Connect Four Game using Minmax-A-B and Alpha-Beta Search Algorithms

Madhusudhan Ramakrishnaiah

(A04735682)

1. INTRODUCTION

1.1 Problem Statement

The most important thing for humanity to do right now is to invent true artificial intelligence (AI): machines or software that can think and act independently in a wide variety of situations. Once we have artificial intelligence, it can help us solve all manner of other problems.

Games and artificial intelligence have a long history together. Even since before artificial intelligence was recognized as a field, early pioneers of computer science wrote game-playing programs because they wanted to test whether computers could solve tasks that seemed to require "intelligence". Alan Turing, arguably the principal inventor of computer science, (re)invented the Minimax algorithm and used it to play Chess. (As no computer had been built yet, he performed the calculations himself using pen and paper.) Arthur Samuel was the first to invent the form of machine learning that is now called reinforcement learning; he used it in a program that learned to play Checkers by playing against itself. Much later, IBM's Deep Blue computer famously won against the reigning grandmaster of Chess, Gary Kasparov, in a much-publicized 1997 event. Currently, many researchers around the world work on developing better software for playing the board game Go, where the best software is still no match for the best humans.

The two reasons that games appeared to be a good domain to explore machine intelligence. The first reason is they provide a structured task in which it is very easy to measure success or failure. The second reason is they did not obviously require large amounts of knowledge. They were thought to be solvable by straightforward search from the starting state to a winning position. The first of these reasons remains valid and accounts for continued interest in the area of game playing by machine. Unfortunately, the second is not true for any but the simplest games. The reason that second statement is not valid is because the search tree will grow exponentially. It is not possible to search the entire search tree to find the best move. The next problem is how to compare the moves to find the best solution. Thus, the three main problems are creating the search tree, managing the tree depth, and comparing the moves.

Develop programs by implementing algorithms MINMAX-A-B (Rich & Knight) and ALPHA-BETA-SEARCH (Russell & Norvig) in C or C++, language. Devise Deep-Enough (use some heuristics as given in Rich and Knight’s book) and Move-Gen functions.

Using "Connect Four" game as an example to test your program and Devise Deep-Enough, Move-Gen, and at least one evaluation function per person.

1.2 Solution

The solution to the problem is that using 2 algorithms one being more efficient that the other.

MinmaxAB Search Algorithm – We implemented this search technique as the algorithm was described in Rich & Knight text book. This uses depth first search technique. The deep enough function is used to control the depth search from exponential growth by having a check on the depth of the tree. The utility or evaluation function is used to choose the best move. The utility or evaluation function is applied in the leaf node and is propagated up in the tree.

This is implemented as a recursive function wherein we basically consider 2 cases

1st case – Base case

In this case we set a certain depth until which the nodes of the tree is generated and traversed to find the most optimal node by using the evaluation function.

This is achieved basically for example if we consider a depth cut-off of 2 (i.e depth starts from 0 and ends at 2) then the leaf nodes at depth 2 is assigned utility values that is computed by the evaluation function and these values are returned back up the tree up to the root node from where we will be able to find the most probable node that has a chance of winning in the future.

2nd case – Recursive case

In this case we iterate through the 7 children nodes using a for loop and call the minmax function recursively until the 1st case i.e base case evaluates to be true.

This function finds the most optimal node by using the variable values, and comparing use threshold and pass threshold starting from the leaf nodes and then backing up to the root node.

This above procedure is repeated multiple times until a player has won or tie has been reached.

AlphaBeta Search Algorithm – This is implemented by using the algorithm as described in Russell & Norvig text book. This also uses depth first search technique. The deep enough function is used to control the depth search from exponential growth by having a check on the depth of the tree. The utility or evaluation function is used to choose the best move. The utility or evaluation function is applied in the leaf node and is propagated up in the tree.

The main difference between this search technique and the above algorithm is that this traverses only to those parts (i.e branches) of the tree from where the optimal nodes might get selected and prunes away the un-necessary branches by calculating the alpha and beta value and then comparing them. Therefore, this search technique will be more efficient than MinmaxAB search.

This is implemented as a recursive function wherein we basically consider 3 cases

1st case – Base case

This case uses the deep enough function to check whether the depth cut-off is reached or not. If it is reached then it return the utility values of all the leaf nodes at that depth.

2nd case – Recursive case

This case is for player ‘X’, it calls itself recursively for all the 7 children nodes by iterating through the for loop and then checking for the values returned by the algorithm and then comparing the best value among nodes, and then calculates alpha and beta value and prunes those branches according to the alpha and beta values compared.

3rd case – Recursive case

This case is for player ‘O’. It has same functionality as described above but has minor differences (i.e the best value initially set and the condition for pruning the branches of tree) it calls itself recursively for all the 7 children nodes by iterating through the for loop and then checking for the values returned by the algorithm and then comparing the best value among nodes, and then calculates alpha and beta value and prunes those branches according to the alpha and beta values compared.

This above procedure is repeated multiple times until a player has won or tie has been reached.

2. CONTRIBUTION

Madhusudhan's Contribution:

1. As a team we researched about the Connect4 game and researched about developing it in a game.

2. As a team we worked on developing MinMax A-B and Alpha Beta Search Algorithm.

3. Developed his Evaluation function.

4. Helped in integrating the code.

5. Analyzed the evaluation function with the team members evaluation function and tabulated.

Harsha's Contribution:

1. As a team we researched about the Connect4 game and researched about developing it in a game.

2. As a team we worked on developing MinMax A-B and Alpha Beta Search Algorithm.

3. Developed an Evaluation function.

4. Helped in integrating the code.

5. Analyzed the evaluation function with the team members evaluation function and tabulated

Santhosh's Contribution:

1. As a team we researched about the Connect4 game and researched about developing it in a game.

2. As a team we worked on developing MinMax A-B and Alpha Beta Search Algorithm.

3. Developed an Evaluation function.

4. Helped in integrating the code.

5. Analyzed the evaluation function with the team members evaluation function and tabulated.

3. GAME DESCRIPTION

3.1 Rules

Connect-Four is a game for two persons. Both players have 21 identical men. In the standard form of the game, one set of men is yellow, and the other set is red. The game is played on a vertical, rectangular board consisting of 7 vertical columns of 6 squares each. If a man is put in one of the columns, it will fall down to the lowest unoccupied square in the column. As soon as a column contains 6 men, no other man can be put in the column. Putting a man in one of the columns is called: a move.

The players make their moves in turn. There are no rules stating that the player with, for instance, the yellow men should start. Since it is confusing to have to identify for each new game the colour that started the game, we will assume that the sets of men are colored white and black instead of yellow and red. Like chess and checkers (and unlike go) it is assumed that the player playing the white men will make the first move.

Both players will try to get four connected men, either horizontally, vertically or diagonally. The first player who achieves one such group of four connected men, wins the game. If all 42 men are played and no player has achieved this goal, the game is drawn.

3.2 Mathematical complexity

To get an idea about the complexity of the game an estimate is presented of the number of different positions which can be achieved, if the game is played according to the rules. A position which can occur during a game is called a legal position, while a position which cannot be achieved is called illegal.

Each square can be in one of three states: empty, white or black. Therefore, it is easy to see that the number of possible positions is at most 342 ($\geq$ 1020). This upper bound is a very crude one, and can be brought into better proportions

4. SEARCH ALGORITHMS

4.1 Alpha - Beta Search:

Alpha–beta pruning is a search algorithm that seeks to decrease the number of nodes that are evaluated by the minimax algorithm in its search tree. It is an adversarial search algorithm used commonly for machine playing of two-player games. It stops completely evaluating a move when at least one possibility has been found that proves the move to be worse than a previously examined move. Such moves need not be evaluated further.

The algorithm of Alpha Beta Pruning:

function ALPHA-BETA-SEARCH(state) **returns** an action
v ←MAX-VALUE(state,−∞,+∞)
return the action in ACTIONS(state) with value v

function MAX-VALUE(state,α, β) **returns** a utility value
if TERMINAL-TEST(state) **then return** UTILITY(state)
v ←−∞
for each a **in** ACTIONS(state) **do**
v ←MAX(v, MIN-VALUE(RESULT(s,a),α, β))
if v ≥ β **then return** v
α←MAX(α, v)
return v

function MIN-VALUE(state,α, β) **returns** a utility value
if TERMINAL-TEST(state) **then return** UTILITY(state)
v ←+∞
for each a **in** ACTIONS(state) **do**
v ←MIN(v, MAX-VALUE(RESULT(s,a) ,α, β))
if v ≤ α **then return** v
β←MIN(β, v)
return v

4.2 MinimaxAB

The *minimax A-B search procedure* is a depth- first, depth-limited search procedure. It is a MiniMax algorithm with the implementation of Alpha Beta pruning to prune the nodes that cannot possibly influence the final decision.

Algorithm: MINIMAX-A-B(Position, Depth, Player, Use-Thresh, Pass-Thresh)

1. If DEEP-ENOUGH(*Position*, *Depth),* then return the structure

VALUE = *STATlC (Position, Player);*

PATH = nil

2. Otherwise, generate one more ply of the tree by calling the function MOVE - GEN (Position, Player) and setting SUCCESSORS to the list it returns.

3.If SUCCESSORS is empty, there are no moves to be made; return the same structure that would have been returned if DEEP-ENOUGH had returned TRUE.

4. If SUCCESSORS is not empty, then go through it, examining each element and keeping track of the best one. This is done as follows.

For each clement SUCC of SUCCESSORS:

(a) Set RESULT-SUCC to

MINIMAX-A-B(SUCC, *Depth* + 1, OPPOSlTE *(Player),*

- *Pass-Thresh,* - *Use-Thresh).*

(b) Set NEW-VALUE to - VALUE(RESULT-SUCC).

(c) If NEW-VALUE> *Pass-Thresh,* then we have found a successor that is better than any that have been examined so far. Record this by doing the following.

(i) Set *Pass-Thresh* to NEW-VALUE.

(ii) The best known path is now from CURRENT to SUCC and then on to the appropriate

path from SUCC as determined by the recursive call to MINIMAX-A-B. So set BEST-PATH to

the result of attaching SUCC to the front of PATH(RESULT-SUCC).

(d) If *Pass-Thresh* (reflecting the current best value) is not better than *Use-Thresh,* then we should

stop examining this branch. But both thresholds and values have been inverted.

So if *Pass-Thresh* >= *Use-Thresh,* then return immediately with the value

VALUE = *Pass-Thresh*

PATH = BEST-PATH

5. Return the structure

VALUE = *Pass-Thresh*

PATH = BEST-PATH

The MiniMax A-B algorithm uses mov generator to produce the successor nodes from the current position. The algorithm then uses evaluation function to find the goodness of the position. The goodness of the node helps to find the best node. The process is continued until it reaches the leaf node. In the leaf node, the node's value is compared with the value of the Pass Threshold and updates Pass Threshold if the value is greater than the Pass Threshold and backs up the tree. The Pass Threshold value is compared with Use Threshold value at each parent node and if the Pass Threshold is greater than the Use Threshold then the successor nodes of that Parent node will not be considered. This process is recursively done until the root's Pass Threshold value is updated with the best Value. The path from which the best Value is obtained will be the Best Path which leads to the Victory.

The Minimax A-B algorithm is much efficient because of the usage of Alpha Beta search algorithm. It reduces the search time because it doesn't consider the paths that are not useful.

4.3 Reason for using Pruning Algorithm

the pruning algorithm is used because it is more efficient than the minmaxAB algorithm.

when AlphaBeta search algorithm is used it prunes those parts of tree (i.e branches of the tree) where there is no possibility of selecting the value from that branch of the tree.

Therefore, by avoiding the un-necessary branches this pruning algorithm saves the execution time and also the memory required by traversing and expanding only those nodes (branch) of the tree which shows potential in being the optimal node for our game.

5. Evaluation Function

An evaluation function returns the estimate of the expected utility of the game from a given position. The performance of a game playing program depends upon the quality of the evaluation function. It assigns approximate material value for
each position.

Basic criteria,
1.Evaluation function must agree with the utility function on terminal states.
2.It must not take too long.
3. It should accurately reflect the actual chances of winning.

5.1 Madhusudhan's Evaluation Function:

```
int tree::evaluation_m(){

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}
};

//since the sum of values of half board is 138.
int initial_value = 138;
int value = 0;

if(ob->player == 'X'){
//for lhs diagonal win
for(int r=0;r<3;r++){
for(int c=6;c>2;c--){
char c1 = ob->board[r][c];
char c2 = ob->board[r+1][c-1];
char c3 = ob->board[r+2][c-2];
char c4 = ob->board[r+3][c-3];
if((c1 == 'X' && c2 == 'X') || (c2 == 'X' && c3 == 'X') || (c3 == 'X' && c4 == ' ')){
value = 100;
}
}
}
```

```
//for rhs diagonal win
for(int r=0;r<3;r++){
for(int c=0;c<4;c++){
char c1 = ob->board[r][c];
char c2 = ob->board[r+1][c+1];
char c3 = ob->board[r+2][c+2];
char c4 = ob->board[r+3][c+3];
if((c1 == 'X' && c2 == 'X') || (c2 == 'X' && c3 == 'X') || (c3 == 'X' && c4 == ' ')){
value = 100;
}
}
}

for(int i = 0; i < 6; i++){
for(int j = 0; j < 7; j++){
if (this->ob->board[i][j] == 'X')
value += evaluationTable[i][j];
else if (this->ob->board[i][j] == 'O')
value -= evaluationTable[i][j];
}
}
}

if(ob->player == 'O'){

for(int r=0;r<3;r++){
for(int c=6;c>2;c--){
char c1 = ob->board[r][c];
char c2 = ob->board[r+1][c-1];
char c3 = ob->board[r+2][c-2];
char c4 = ob->board[r+3][c-3];
if((c1 == 'O' && c2 == 'O') || (c2 == 'O' && c3 == 'O') || (c3 == 'O' && c4 == ' ')){
value = -100;
}
}
}

//for rhs diagonal win
for(int r=0;r<3;r++){
for(int c=0;c<4;c++){
char c1 = ob->board[r][c];
char c2 = ob->board[r+1][c+1];
char c3 = ob->board[r+2][c+2];
```

```
        char c4 = ob->board[r+3][c+3];
        if((c1 == 'O' && c2 == 'O') || (c2 == 'O' && c3 == 'O') || (c3 == 'O' && c4 == ' ')){
        value = -100;
        }
        }
        }

        for(int i = 0; i < 6; i++){
        for(int j = 0; j < 7; j++){
        if(this->ob->board[i][j] == 'X')
        value += evaluationTable[i][j];
        else if (this->ob->board[i][j] == 'O')
        value -= evaluationTable[i][j];
        }
        }
        }
        return initial_value + value;
}
```

In my above evaluation function, I have considered the following cases

The first thing was that I created a 2-D array same as the connect four game board dimensions with number of rows and columns being 6 and 7 respectively. Then I gave values to each cell of the board i.e for each of the 6*7 = 42 cells of the board.

The values that I assigned was depending on the number of possible ways that a player can win by making his move in that cell. By number of ways, I mean there are generally 13 maximum possible ways a player can win by connecting the four cells that primarily being
Vertical win, horizontal win, left diagonal win and right diagonal win.

So,

C0	C1	C2	C3	C4	C5	C6
X			X			
X		X				
X	X					
X	X	X	X			

Suppose the player makes his move by placing an 'X' in the 0th row and 0th col now the number of possible ways that player can win is 3 so I assign the value 3 into that cell.

In the above board the '**X**' (in bold character) is the 0th row and 0th col and if the player makes his move by placing it that cell then the number of possible ways he can win is marked in normal 'X' s (i.e. not in bold) i.e. one being vertical, second being horizontal and the third being left diagonal.

Let’s consider one more example

C0	C1	C2	C3	C4	C5	C6
	X			X		
	X		X			
	X	X				
X	**X**	X	X	X		
X	X					

Now it is clear that as we move away from the edges of the board and towards the center cells the number of ways of possible win will increase.

So, if player ‘X’ makes his move by placing X on 1st row 1st column then he has a total of 6 possible ways of winning.

Working of my (Madhusudhan’s) evaluation function:

I declared and initialized a 2-D matrix(array) for depecting the board of the coonect four game With row and cols being initialized to 6 and 7 respectively.
Then I set a initial value to 138 which is just the addition of all the values that I assigned to each cell but I considered only one half of the board since there are 2 players so the other half is for the other player.

I am using one more variable initially set to 0 this is just to assign the value according to the description below.(in the last sentence)

Then I check whether the current player is ‘X’ or ‘O’ depeding on which I assign positive or negative values to the variable sum in my code.

Other than that the conditions, cases are all the same for both players.

Now after checking who is the current player it will iterate through the board using the neseted for loop for 3 cases primarily

For the 1st one :

It will only check for the left diagonal wins if there is any possibility then it assigns the value 100 to the variable sum.

2nd one:

It will only check for the right diagonal wins if there is any possibility then it assigns the value 100 to the variable sum.

3rd one:

It checks all the cells and depending on whether in each cell if there is ‘X’ then adds the initially assigned values of each cell to the variable sum else If there is ‘Y’ then subtracts the initially assigned values of each cell from the variable sum.

After these cases are executed then it will return the sum of the initial value i.e 138 and the variable sum from the evaluation function as a return value.

For player ‘O’

The above 3 steps are performed for the player ‘O’ also but the only difference is the variable sum will be assigned a value -100 for the 1st case and 2nd case.

5.2 Harsha's Evaluation Function

```
int tree::evaluation_h(){

int value = 0;

if(ob->player == 'X'){
for(int i=0;i<6;i++){
for(int j=0;j<4;j++){
//for continuous row
//first checking whether not everything is empty
if(ob->board[i][j]!= ' ' || ob->board[i][j+1]!= ' ' || ob->board[i][j+2]!= ' ' || ob->board[i][j+3]!= ' '){
if((ob->board[i][j] == 'X' && ob->board[i][j+1] == 'X') && (ob->board[i][j+2]==' ')){
value = 10;
}
}
}
}

for(int i=0;i<3;i++){
for(int j=0;j<7;j++){
//for continuous col
//first checking whether not everything is empty
if(ob->board[i][j]!= ' ' || ob->board[i][j+1]!= ' ' || ob->board[i][j+2]!= ' ' || ob->board[i][j+3]!= ' '){
if((ob->board[i][j] == 'X' && ob->board[i+1][j] == 'X') && (ob->board[i+2][j]==' ')){
value = 10;
}
}
}
}

//for lhs diagonal win
for(int r=0;r<3;r++){
for(int c=6;c>2;c--){
char c1 = ob->board[r][c];
char c2 = ob->board[r+1][c-1];
char c3 = ob->board[r+2][c-2];
char c4 = ob->board[r+3][c-3];
//board[r][c]==board[r+1][c-1]==board[r+2][c-2]==board[r+3][c-3]
//cout<<board[r][c]<<" "<<board[r+1][c-1]<<" "<<board[r+2][c-2]<<" "<<board[r+3][c-3]<<endl;
```

```
if(c1!= ' ' || c2!= ' ' || c3!= ' ' || c4!= ' '){
if((c1 == 'X' && c2 == 'X') || (c2 == 'X' && c3 == 'X') || (c3 == 'X' && c4 == 'X')){
//cout<<"win"<<endl;
value = 15;
}
}
}
}

//for rhs diagonal win
for(int r=0;r<3;r++){
for(int c=0;c<4;c++){
char c1 = ob->board[r][c];
char c2 = ob->board[r+1][c+1];
char c3 = ob->board[r+2][c+2];
char c4 = ob->board[r+3][c+3];
//board[r][c]==board[r+1][c+1]==board[r+2][c+2]==board[r+3][c+3]
//cout<<board[r][c]<<" "<<board[r+1][c-1]<<" "<<board[r+2][c-2]<<" "<<board[r+3][c-
3]<<endl;
if(c1!=' '||c2!=' '||c3!=' '||c4!=' '){
if((c1 == 'X' && c2 == 'X') || (c2 == 'X' && c3 == 'X') || (c3 == 'X' && c4 == 'X')){
//cout<<"win"<<endl;
value = 15;
}
}
}
}
}

else if(ob->player == 'O'){
for(int i=0;i<6;i++){
for(int j=0;j<4;j++){
//for continuous row
//first checking whether not everything is empty
if(ob->board[i][j]!= ' ' || ob->board[i][j+1]!= ' ' || ob->board[i][j+2]!= ' ' || ob-
>board[i][j+3]!= ' '){
if((ob->board[i][j] == 'O' && ob->board[i][j+1] == 'O') && (ob->board[i][j+2]==' ')){
value = 10;
}
}
}
}
for(int i=0;i<3;i++){
```

```
for(int j=0;j<7;j++){
//for continuous col
//first checking whether not everything is empty
if(ob->board[i][j]!= ' ' || ob->board[i][j+1]!= ' ' || ob->board[i][j+2]!= ' ' || ob->board[i][j+3]!= ' '){
if((ob->board[i][j] == 'O' && ob->board[i+1][j] == 'O') && (ob->board[i+2][j]==' ')){
value = 10;
}
}
}
}

//for lhs diagonal win
for(int r=0;r<3;r++){
for(int c=6;c>2;c--){
char c1 = ob->board[r][c];
char c2 = ob->board[r+1][c-1];
char c3 = ob->board[r+2][c-2];
char c4 = ob->board[r+3][c-3];
//board[r][c]==board[r+1][c-1]==board[r+2][c-2]==board[r+3][c-3]
//cout<<board[r][c]<<" "<<board[r+1][c-1]<<" "<<board[r+2][c-2]<<" "<<board[r+3][c-3]<<endl;
if(c1!= ' ' || c2!= ' ' || c3!= ' ' || c4!= ' '){
if((c1 == 'O' && c2 == 'O') || (c2 == 'O' && c3 == 'O') || (c3 == 'O' && c4 == 'O')){
//cout<<"win"<<endl;
value = 15;
}
}
}
}

//for rhs diagonal win
for(int r=0;r<3;r++){
for(int c=0;c<4;c++){
char c1 = ob->board[r][c];
char c2 = ob->board[r+1][c+1];
char c3 = ob->board[r+2][c+2];
char c4 = ob->board[r+3][c+3];
//board[r][c]==board[r+1][c+1]==board[r+2][c+2]==board[r+3][c+3]
//cout<<board[r][c]<<" "<<board[r+1][c-1]<<" "<<board[r+2][c-2]<<" "<<board[r+3][c-3]<<endl;
if(c1!=' '||c2!=' '||c3!=' '||c4!=' '){
if((c1 == 'O' && c2 == 'O') || (c2 == 'O' && c3 == 'O') || (c3 == 'O' && c4 == 'O')){
```

```
        //cout<<"win"<<endl;
        value = 15;
        }
        }
        }
        }

        }
        this->set_heuristic_value(value);
        //cout<<value<<endl;
        return value;
}
```

Working of the above (Harsha's) evaluation function:

In my evaluation function, the Player X starts the first move and it will have seven possible moves to place the X. The Player X will search the board and if two consecutive X is found and the space next to the X is empty, it will choose that board as best and proceed in that board. The Player O will follow the similar move to choose the best move. The wining is considered if either X or O have consecutive four's in the board. The function will have 4 ways to find the wining move. The Vertical Check will check whether the column in the board has consecutive four's, the Horizontal Check will check whether the row in the board has consecutive four's and the right and left diagonal Check will check whether the consecutive fours are in diagonal.

5.3 Santhosh's Evaluation Function

```
int tree::evaluation_s(){

int X4 = 0, X3 = 0,X2 = 0;
int O3 = 0, O2 = 0;
long int eval;

// checking the rows for 4X
for(int i=0 ; i < 6; i++)
{
for(int j = 0 ;j < 7; j = j++)
{
if('X' == ob->board[i][j] && 'X' == ob->board[i][j+1] && 'X' == ob->board[i][j+2] &&
'X' == ob->board[i][j+3])
{
X4++;
}
```

```
}
}

// Checking the rows for 3X
for(int i = 0; i < 6; i++)
{
for(int j = 0; j < 5; j = j++)
{
if('X' == ob->board[i][j] && 'X' == ob->board[i][j+1] && 'X' == ob->board[i][j+2])
{
X3++;
}
}
}

// Checking the rows for 2X
for(int i = 0; i < 6; i++)
{
for(int j = 0; j < 6;j++)
{
if('X' == ob->board[i][j] && 'X' == ob->board[i][j+1])
{
X2++;
}
}
}

// Checking the rows for 3O
for(int i = 0; i < 6; i++)
{
for(int j = 0; j < 5; j = j++)
{
if('O' == ob->board[i][j] && 'O' == ob->board[i][j+1] && 'O' == ob->board[i][j+2])
{
O3++;
}
}
}

// Checking the rows for 2O
for(int i = 0; i < 6; i++)
{
for(int j = 0; j < 6; j = j++)
```

```
{
if('O' == ob->board[i][j] && 'O' == ob->board[i][j+1])
{
O2++;
}
}
}

// checking for column 4X
for(int i = 0; i < 3; i++)
{
for(int j = 0; j < 7; j++)
{
if('X' == ob->board[i][j] && 'X' == ob->board[i+1][j] && 'X' == ob->board[i+2][j] &&
'X' == ob->board[i+3][j])
{
X4++;
}
}
}

// checking for column 3X
for(int i = 0; i < 2; i++)
{
for(int j = 0; j < 7; j++)
{
if('X' == ob->board[i][j] && 'X' == ob->board[i+1][j] && 'X'== ob->board[i+2][j])
{
X3++;
}
}
}

// Checking for column 2X
for(int i = 0; i < 1; i++)
{
for(int j = 0;j < 7; j++)
{
if('X' == ob->board[i][j] && 'X' == ob->board[i+1][j])
{
X2++;
}
}
```

```
}

// Checking for column 3O
for(int i = 0; i < 2; i++)
{
for(int j = 0; j < 7; j++)
{
if('O' == ob->board[i][j] && 'O' == ob->board[i+1][j] && 'O'== ob->board[i+2][j])
{
O3++;
}
}
}

// Checking for column 2O
for(int i = 0; i < 1; i++)
{
for(int j = 0; j < 7; j++)
{
if('O' == ob->board[i][j] && 'O' == ob->board[i+1][j])
{
O2++;
}
}
}

// Checking for diagonal (Positive slope) 4X
for(int i = 0; i < 3; i++)
{
for(int j = 0; j < 4; j++)
{
if('X' == ob->board[i][j] && 'X' == ob->board[i+1][j+1] && 'X' == ob->board[i+2][j+2]
&& 'X' == ob->board[i+3][j+3])
{
X4++;
}
}
}
// Checking for diagonal (Positive slope) 3X
for(int i = 0; i < 2; i++)
{
for(int j = 0; j < 3;j++)
{
```

```
if('X' == ob->board[i][j] && 'X' == ob->board[i+1][j+1] && 'X' == ob->board[i+2][j+2])
{
X3++;
}
}
}
// Checking for diagonal (positive slope) 2X
for(int i = 0; i < 1; i++)
{
for(int j = 0; j < 2;j++)
{
if('X' == ob->board[i][j] && 'X' ==ob->board[i+1][j+1])
{
X2++;
}
}
}
// Checking for diagonal (Positive slope) 3O
for(int i = 0; i < 2; i++)
{
for(int j = 0; j < 3;j++)
{
if('O' == ob->board[i][j] && 'O' == ob->board[i+1][j+1] && 'O' == ob->board[i+2][j+2])
{
O3++;
}
}
}
// Checking for diagonal (positive slope) 2O
for(int i = 0; i < 1; i++)
{
for(int j = 0; j < 2;j++)
{
if('O' == ob->board[i][j] && 'O' == ob->board[i+1][j+1])
{
O2++;
}
}
}
// Checking for diagonal (Negative slope) 4X
for(int i = 3; i < 6; i++)
{
for(int j = 4; j < 7; j++)
```

```
{
if('X' == ob->board[i][j] && 'X' == ob->board[i-1][j+1] && 'X' == ob->board[i-2][j+2]
&& 'X' == ob->board[i-3][j+3])
{
X4++;
}
}
}
// Checking for diagonal (Negative slope) 3X
for(int i = 4; i < 6; i++)
{
for(int j = 5; j < 7; j++)
{
if('X' == ob->board[i][j] && 'X' == ob->board[i-1][j+1] && 'X' == ob->board[i-2][j+2])
{
X3++;
}
}
}
// Checking for diagonal (Negative slope) 2X
for(int i = 5; i < 6; i++)
{
for(int j = 6; j < 7; j++)
{
if('X' == ob->board[i][j] && 'X' == ob->board[i-1][j+1])
{
X2++;
}
}
}
// Checking for diagonal (Negative slope) 3O
for(int i = 4; i < 6; i++)
{
for(int j = 5; j < 7; j++)
{
if('O' == ob->board[i][j] && 'O' == ob->board[i-1][j+1] && 'O' == ob->board[i-2][j+2])
{
O3++;
}
}
}
// Checking for diagonal (Negative slope) 2O
for(int i = 5; i < 6; i++)
```

```
        {
        for(int j = 6; j < 7; j++)
        {
        if('O' == ob->board[i][j] && 'O' == ob->board[i-1][j+1])
        {
        O2++;
        }
        }
        }
        eval = ( X4 * 100000 + X3 * 100 + X2 * 10 ) - (O3 * 100 + O2 * 10);
        return eval;
}
```

Working of the above (Santosh's) evaluation function:

8 * X3 + 4 * X2 + X1 – (8 * O3 + 4 * O2 + O1)

Xn is the number of blocks of 4 (4 consecutive memory locations in any direction) which have n X pieces and no O pieces.

X3 = No of - rows which have 3 X's

O3 = No of - rows which have 3 O's

X2 = No of - rows which have 2 X's

O2 = No of - rows which have 2 O's

X1 = No of - rows which have 1 X

O1 = No of - rows which have 1 O

Let us consider a sample board configuration as follows

	O					
X	**X**	**O**		**X**		
O	**X**	X	**O**	**O**		

Values for features:

X3 = 0;

X2 = 0;

X1 = 7;

O3 = 1;

O2 = 1;

O1 = 8;

Evaluation

(8 * 0 + 4 * 0 + 7) – (8 * 1 + 4 * 1 + 8) = -13;

This assumes that Max is playing X. So, this state favors O since its evaluation is a negative value

6. CODE

6.1 Main.cpp

```
#include <iostream>
#include <ctime>

#include "MinmaxAB.h"
#include "TreeNodes.h"
#include "ConnectFourBoard.h"
#include "AlphaBeta.h"
using namespace std;

/*variable that can be accessed by files
  TreeNodes.h and .cpp, and main function
*/
int num_nodes_generated, num_nodes_expanded,game_path_length,start_time,stop_time;

/*made as a global variable since it must be accessible to
  Minmax() fn and also AlphaBeta() fn
*/
char evalfn_choice;

/* for calling minmaxAB function
   creates a tree object,board object and minmaxab object
   using a while loop keeps calling the minmaxAB function
   repeatedly until the condition checkPlayerWon evaluates
   to true. */
void MinMax(){

   int moves_made = 0;
   tic *board_ob = new tic;

   cout<<endl;
   cout<<"initial board state before starting game with(Initial Node config/Root Node) "<<moves_made<<" moves made"<<endl;
   cout<<"-------------------------------------------------------"<<endl;

   board_ob->display_board(board_ob);
   board_ob->player = 'X';

   start_time = clock();
    while(!board_ob->checkPlayerWon()){

        tree *head = new tree;
```

```
        *(head->ob) = *board_ob;

        if(head->ob->isBoardEmpty()){

            int random_col_index = head->ob->makeFirstMove();
            head->ob->board[0][random_col_index] = 'X';
            head->ob->player = 'X';
            moves_made++;
            game_path_length++;
            cout<<"After 1st move board state(Max's move/Root Node):"<<endl;
            head->ob->display_board(head->ob);
        }

        minmaxA min_ob;
        //assigns the users choice of evaluation function chosen
        min_ob.eval_choice = evalfn_choice;
        int x = min_ob.minmaxAB(head,0,1000,-1000,head->ob->player);
        moves_made++;
        game_path_length++;

        //to get the optimal node - Move Gen function
        head->move_gen(board_ob,moves_made);
    }
    stop_time = clock();
}

/* for calling minmaxAB function
   creates a tree object,board object and alphabeta object
   using a while loop keeps calling the alphabeta function
   repeatedly until the condition checkPlayerWon evaluates
   to true. */
void AlphaBeta(){

   int moves_made = 0;

   tic *board_ob = new tic;

   cout<<endl;
   cout<<"initial board state before starting game with(Initial Node config/Root Node)
"<<moves_made<<" moves made"<<endl;
   cout<<"------------------------------------------------------"<<endl;

   board_ob->display_board(board_ob);
   board_ob->player = 'X';

   start_time = clock();
```

```
  while(!board_ob->checkPlayerWon()){

      tree *head = new tree;
      *(head->ob) = *board_ob;

      if(head->ob->isBoardEmpty()){

          int random_col_index = head->ob->makeFirstMove();
          head->ob->board[0][random_col_index] = 'X';
          head->ob->player = 'X';
          moves_made++;
          game_path_length++;
          cout<<"After 1st move board state(Max's move/Root Node):"<<endl;
          head->ob->display_board(head->ob);
      }

      alphabeta ab_ob;
      //assigns the users choice of evaluation function chosen
      ab_ob.eval_choice = evalfn_choice;
      int x = ab_ob.alpha_beta(head,0,head->ob->player,1000,-1000);
      moves_made++;
      game_path_length++;

      //to get the optimal node - Move Gen function
      head->move_gen(board_ob,moves_made);
  }
  stop_time = clock();
}

int main()
{
   int algo_choice;
   cout<<"Select Algorithm and evaluation function to play the game with ?"<<endl;
   cout<<endl;
   cout<<"  1 - MinmaxAB"<<endl;
   cout<<"  2 - AlphaBeta"<<endl;
   cout<<endl;
   cin>>algo_choice;
   cout<<endl;

   cout<<"  M/m - Madhusudhan's eval function"<<endl;
   cout<<"  H/h - Harsha's eval function"<<endl;
   cout<<"  S/s - Santosh's eval function"<<endl;
   cout<<endl;
   cin>>evalfn_choice;
   cout<<endl;
```

```
char final_choice;
//for my final_choice
if((algo_choice == 1) && (evalfn_choice == 'M' || evalfn_choice == 'm'))
   final_choice = 'A';

if((algo_choice == 2) && (evalfn_choice == 'M' || evalfn_choice == 'm'))
   final_choice = 'B';


//for Harsha final_choice
if((algo_choice == 1) && (evalfn_choice == 'H' || evalfn_choice == 'h'))
   final_choice = 'C';

if((algo_choice == 2) && (evalfn_choice == 'H' || evalfn_choice == 'h'))
   final_choice = 'D';


//for Santosh final_choice
if((algo_choice == 1) && (evalfn_choice == 'S' || evalfn_choice == 's'))
   final_choice = 'E';

if((algo_choice == 2) && (evalfn_choice == 'S' || evalfn_choice == 's'))
   final_choice = 'F';


switch(final_choice){

case 'A':
   MinMax();
   break;

case 'B':
   AlphaBeta();
   break;


case 'C':
   MinMax();
   break;

case 'D':
   AlphaBeta();
   break;


case 'E':
```

```
        MinMax();
        break;

    case 'F':
        AlphaBeta();
        break;

    default:
        cout<<"No case matched! Please select options properly."<<endl;
    }

  return 0;
}
```

6.2 ConnectFourBoard.h

```
#ifndef CONNECTFOURBOARD_H_INCLUDED
#define CONNECTFOURBOARD_H_INCLUDED

/* used to create an abstract data type
   Move which contains 2 fields one for row index
   and the other for column index*/
struct Move{
    int row_index;
    int col_index;
};

class tic{

public:

    /*this pointer to a pointer variable is used to dynamically create
     a 2-d array for the connect-four board and the dimensions
     are given as rows-6, and columns-7
    */
    char ** board;
    //this is used for row size of the board and is set to 6
    int row_size;
    //this is used for col size of the board and is set to 7
    int col_size;
    //this is used to know who the player is at that particular board
    char player;
    /*this is used to dynamically create an ADT moves array using concept
    of structures
    */
    Move *moves;

    //Member functions of this class
    tic();
    bool isBoardEmpty();
    int makeFirstMove();
    void checkPossibleMoves();
    bool checkPlayerWon();
    void operator=(tic);
    void display_board(tic *x);
};
#endif // CONNECTFOURBOARD_H_INCLUDED
```

6.3 ConnectFourBoard.cpp

```
#include <iostream>
#include <ctime>
#include <cstdlib>

#include "ConnectFourBoard.h"
using namespace std;

/* a constructor that initializes all
   the member variables of this class
   creates a 2-d array dynamically for
   connect-four board using new operator
   then initializes all the cells to empty
   character
   creates dynamically a structure variable
   of type Move  and assigns its members
   row_inde and col_index to -1
   sets the player char variable to 'n'
*/
tic::tic(){

    row_size = 6;
    col_size = 7;

    board = new char* [row_size];
    for(int i=0;i<row_size;i++)
        board[i] = new char [col_size];

    for(int i=0;i<row_size;i++){
        for(int j=0;j<col_size;j++)
            board[i][j]=' ';
    }

    moves = new Move[7];
    for(int i=0;i<7;i++){
        moves[i].col_index = -1;
        moves[i].row_index = -1;
    }

    player = 'n';
}

/*a boolean function that returns false if the
  board is not empty else returns true
*/
```

```
bool tic::isBoardEmpty(){

    for(int i=0;i<row_size;i++){
        for(int j=0;j<col_size;j++){
            if(board[i][j]!=' ')
                return false;
        }
    }
    return true;
}

/* this function is used to
   make the first move by randomly
   choosing a column
*/
int tic::makeFirstMove(){

    int random_col;
    srand(time(NULL));
    random_col = rand()%7+1;
    if(random_col == 7)
        random_col=4;
    return random_col;
}

/*this is used to check all the possible moves
  that can be made by the player 1st case considered is
  checking for whether there are empty cells
  in 0th row,if present then stores that col location
  to col_index of the move's structure member variable
  2nd case considered is checking for all other empty cells
  and if empty, checking whether the below cell is empty
  or not if empty then not storing this location else storing this location
  in col_index and row_index of the structure move
*/
void tic::checkPossibleMoves(){

    int c=0;

    //checking whether empty place is there for the 0th row
    for(int col=0;col<col_size;col++){
        if(board[0][col]==' '){
            this->moves[c].row_index = 0;
            this->moves[c].col_index = col;
            c++;
        }
```

```
    }

    //checking for all other rows for empty places
    for(int row=1;row<row_size;row++){
        for(int col=0;col<col_size;col++){
            if(board[row][col]==' '){
                if(board[row-1][col]!=' '){
                    (moves+c)->col_index=col;
                    (moves+c)->row_index=row;
                     c++;
                }
            }
        }
    }
}

/* this is a boolean function that return true
   if a player has won else return false
   it considers 4 cases
     1st for continuous 4 in a row
     2nd for continuous 4 in vertical
     3rd for right diagonal win
     4th for left diagonal win
*/
bool tic::checkPlayerWon(){

    bool win=false;
    char c1,c2,c3,c4;

    //for checking row wins
    for(int r=0;r<row_size;r++){
        for(int c=0;c<4;c++){
            if(!win){
                c1 = board[r][c];
                c2 = board[r][c+1];
                c3 = board[r][c+2];
                c4 = board[r][c+3];
                //checking for board[r][c]==board[r][c+1]==board[r][c+2]==board[r][c+3]
                if(c1!=' '||c2!=' '||c3!=' '||c4!=' '){
                    if((c1 == c2)&& (c2 == c3) && (c3 == c4)){
                        win = true;
                        break;
                    }
                }
            }
        }
```

```
}

//for checking col wins
if(!win){
    for(int r=0;r<3;r++){
        for(int c=0;c<col_size;c++){
            c1 = board[r][c];
            c2 = board[r+1][c];
            c3 = board[r+2][c];
            c4 = board[r+3][c];
            //checking for board[r][c]==board[r+1][c]==board[r+2][c]==board[r+3][c]
            if(c1!=' '||c2!=' '||c3!=' '||c4!=' '){
                if((c1 == c2)&& (c2 == c3) && (c3 == c4)){
                    win = true;
                    break;
                }
            }
        }
    }
}

//looking from right end of array rhs-diagonal
if(!win){
    for(int r=0;r<3;r++){
        for(int c=6;c>2;c--){
            c1 = board[r][c];
            c2 = board[r+1][c-1];
            c3 = board[r+2][c-2];
            c4 = board[r+3][c-3];
            //checking for board[r][c]==board[r+1][c-1]==board[r+2][c-2]==board[r+3][c-3]
            if(c1!=' '||c2!=' '||c3!=' '||c4!=' '){
                if((c1 == c2)&& (c2== c3) && (c3== c4)){
                    //cout<<"win"<<endl;
                    win=true;
                    break;
                }
            }
        }
    }
}
//looking form left end of array lhs-diagonal
if(!win){
    for(int r=0;r<3;r++){
        for(int c=0;c<4;c++){
            c1 = board[r][c];
            c2 = board[r+1][c+1];
```

```
            c3 = board[r+2][c+2];
            c4 = board[r+3][c+3];
            //checking for board[r][c]==board[r+1][c+1]==board[r+2][c+2]==board[r+3][c+3]
            if(c1!=' '||c2!=' '||c3!=' '||c4!=' '){
               if((c1 == c2)&& (c2== c3) && (c3== c4)){
                  //cout<<"win"<<endl;
                  win=true;
                  break;
               }
            }
         }
      }
   }
   return win;
}

/* overloading = operator to copy one
   board to another board object
*/
void tic::operator=(tic rhs){

   for(int i=0;i<row_size;i++){
      for(int j=0;j<col_size;j++){
         this->board[i][j] = rhs.board[i][j];
      }
   }

   for(int i=0;i<7;i++){
      this->moves[i].row_index = rhs.moves[i].row_index;
      this->moves[i].col_index = rhs.moves[i].col_index;
   }
   this->player = rhs.player;
}

/* this function is used for displaying
   board contents
*/
void tic::display_board(tic *x){
   cout<<endl;
   for(int i=5;i>=0;i--){
      for(int j=0;j<7;j++){
         cout<<x->board[i][j]<<" | ";
      }
      cout<<endl;
      for(int k=0;k<7;k++)
         cout<<"----";
```

```
        cout<<endl;
    }
    cout<<endl;
}
```

6.4 TreeNodes.h

```
#ifndef TREENODES_H_INCLUDED
#define TREENODES_H_INCLUDED

#include <iostream>
#include "ConnectFourBoard.h"
using namespace std;

extern int num_nodes_generated,num_nodes_expanded,game_path_length,start_time,stop_time;

class tree{

public:

    //stores the heuristic value of the node i.e board
    int heuristic_value;
    /*to iterate through the children of a node and
    there are exactly 7 possible children for each node
    */
    int num_children;
    //this tores the pointers to its 7 children nodes
    tree *children[7];
    /*an object from the class tic which holds the
    board configuration of each node
    */
    tic *ob;

    //Member functions of tree class
    tree();
    void create_node(char);
    void set_heuristic_value(int);
    void add_all_children();
    bool deep_enough(int);
    int evaluation_m();
    int evaluation_s();
    int evaluation_h();
    void copy_board_status(tic &);
    void helper();
    void display_contents();
    int getOptimalNode();
    void move_gen(tic *,int);
};
#endif // TREENODES_H_INCLUDED
```

6.5 TreeNodes.cpp

```
#include <cstdlib>
#include <ctime>

#include "TreeNodes.h"
using namespace std;

/* a constructor for the tree class that
   initializes all member variables of tree
   class
*/
tree::tree(){

    heuristic_value = -2000;
    num_children = 0;
    for(int i=0;i<7;i++){
        children[i] = NULL;
    }
    //creates pointer for new tic object
    ob = new tic();

}

/* this function creates nodes dynamically
   by using the new operator and assigns the
   address of the node returned by new
   operator to the pointer array that is
   member of each node(i.e tree object)
*/
void tree::create_node(char p){

    children[num_children++] = new tree();
}


/* sets the heuristic value using the
   variable value to nodes (i.e tree object)
*/
void tree::set_heuristic_value(int value){

    heuristic_value = value;
}

/* this function creates 7 nodes(i.e for child's)
   dynamically by using the create_node function
```

```
   above, assigns the player for each of those
   nodes and calls check_possible_moves function
   defined in the ConnectFourBoard.cpp file
   for all the possible moves that can be made by
   the player and assigns them
*/
void tree::add_all_children(){

    char p;
    int row,col;

    for(int i=0;i<7;i++){

        create_node('n');

        /*does the same task as above create_node('n');
        num_children++;
        this->children[i] = new tree();
        */

        *(children[i]->ob) = *(this->ob);


        /*checks parents node move i.e 'X'/'O' and makes
          child's move with the other character(Player)
          also sets the child's player to be that of
          the opposite of parent nodes player
        */

        if(this->ob->player == 'X'){
            p = 'O';
            this->children[i]->ob->player = p;
        }

        else if(ob->player == 'O'){
            p = 'X';
            this->children[i]->ob->player = p;
        }

        //if(ob->player == 'n')
        else{
            p = 'X';
            this->children[i]->ob->player = p;
        }
        /* keeps a count of the total number of
        nodes that is generated
```

```
        */
        num_nodes_generated++;
        children[i]->ob->checkPossibleMoves();
        row = children[i]->ob->moves[i].row_index;
        col = children[i]->ob->moves[i].col_index;
        children[i]->ob->board[row][col] = p;
    }
}

/* this function is used for a depth cut-off
   it has 3 cases where in 1st checks for
   whether heuristic value is set or not
   2nd checks whether a particular depth
   has been reached or not or if a player
   has won the game then return true
   3rd case being that both above cases
   failed that calls the above add_all_children
   function and return false once the control
   is returned back from the add_all_children
   function
*/
bool tree::deep_enough(int depth){

    if(heuristic_value!=-2000)
        return heuristic_value;

    else if(depth == 2 || ob->checkPlayerWon()){
            if(ob->checkPlayerWon()){
                /*so that the last terminal node is
                  also added to the tree i.e
                  board containing win configuration */
                this->add_all_children();
            }
            return true;
    }

    else{
        num_nodes_expanded++;
        this->add_all_children();
        return false;
    }
}

/* this is my (Madhusudhan) Evaluation function
   which is briefly described in my project report
   it creates a 2-d array same as connect-four board
```

```
   dimensions, assigns values to each cells of the
   board, depending on whether the player is Max/Min
   assigns values to them and returns the sum of
   initial_value and the value
*/

int tree::evaluation_m(){

        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}
                       };

        //since the sum of values of half board is 138.
        int initial_value = 138;
        int value = 0;

        if(ob->player == 'X'){
            //for lhs diagonal win
            for(int r=0;r<3;r++){
                for(int c=6;c>2;c--){
                    char c1 = ob->board[r][c];
                    char c2 = ob->board[r+1][c-1];
                    char c3 = ob->board[r+2][c-2];
                    char c4 = ob->board[r+3][c-3];
                    if((c1 == 'X' && c2 == 'X') || (c2 == 'X' && c3 == 'X') || (c3 == 'X' && c4 == ' ')){
                        value = 100;
                    }
                }
            }

            //for rhs diagonal win
            for(int r=0;r<3;r++){
                for(int c=0;c<4;c++){
                    char c1 = ob->board[r][c];
                    char c2 = ob->board[r+1][c+1];
                    char c3 = ob->board[r+2][c+2];
                    char c4 = ob->board[r+3][c+3];
                    if((c1 == 'X' && c2 == 'X') || (c2 == 'X' && c3 == 'X') || (c3 == 'X' && c4 == ' ')){
                        value = 100;
                    }
                }
```

```
        }

        for(int i = 0; i < 6; i++){
            for(int j = 0; j < 7; j++){
                if (this->ob->board[i][j] == 'X')
                    value += evaluationTable[i][j];
                else if (this->ob->board[i][j] == 'O')
                    value -= evaluationTable[i][j];
            }
        }
    }

    if(ob->player == 'O'){

        for(int r=0;r<3;r++){
            for(int c=6;c>2;c--){
                char c1 = ob->board[r][c];
                char c2 = ob->board[r+1][c-1];
                char c3 = ob->board[r+2][c-2];
                char c4 = ob->board[r+3][c-3];
                if((c1 == 'O' && c2 == 'O') || (c2 == 'O' && c3 == 'O') || (c3 == 'O' && c4 == ' ')){
                    value = -100;
                }
            }
        }

        //for rhs diagonal win
        for(int r=0;r<3;r++){
            for(int c=0;c<4;c++){
                char c1 = ob->board[r][c];
                char c2 = ob->board[r+1][c+1];
                char c3 = ob->board[r+2][c+2];
                char c4 = ob->board[r+3][c+3];
                if((c1 == 'O' && c2 == 'O') || (c2 == 'O' && c3 == 'O') || (c3 == 'O' && c4 == ' ')){
                    value = -100;
                }
            }
        }

        for(int i = 0; i < 6; i++){
            for(int j = 0; j < 7; j++){
                if(this->ob->board[i][j] == 'X')
                    value += evaluationTable[i][j];
                else if (this->ob->board[i][j] == 'O')
                    value -= evaluationTable[i][j];
            }
```

```
            }
        }
        return initial_value + value;
}

//santosh's eval fn
int tree::evaluation_s(){

    int X4 = 0, X3 = 0,X2 = 0;
    int O3 = 0, O2 = 0;
    long int eval;

    // checking the rows for 4X
    for(int i=0 ; i < 6; i++)
    {
        for(int j = 0 ;j < 7; j = j++)
        {
            if('X' == ob->board[i][j] && 'X' == ob->board[i][j+1] && 'X' == ob->board[i][j+2] &&
'X' == ob->board[i][j+3])
            {
                X4++;
            }
        }
    }

    // Checking the rows for 3X
    for(int i = 0; i < 6; i++)
    {
        for(int j = 0; j < 5; j = j++)
        {
            if('X' == ob->board[i][j] && 'X' == ob->board[i][j+1] && 'X' == ob->board[i][j+2])
            {
                X3++;
            }
        }
    }

    // Checking the rows for 2X
    for(int i = 0; i < 6; i++)
    {
        for(int j = 0; j < 6;j++)
        {
            if('X' == ob->board[i][j] && 'X' == ob->board[i][j+1])
            {
                X2++;
            }
```

```
        }
    }

    // Checking the rows for 3O
    for(int i = 0; i < 6; i++)
    {
        for(int j = 0; j < 5; j = j++)
        {
            if('O' == ob->board[i][j] && 'O' == ob->board[i][j+1] && 'O' == ob->board[i][j+2])
            {
                O3++;
            }
        }
    }

    // Checking the rows for 2O
    for(int i = 0; i < 6; i++)
    {
        for(int j = 0; j < 6; j = j++)
        {
            if('O' == ob->board[i][j] && 'O' == ob->board[i][j+1])
            {
                O2++;
            }
        }
    }

   // checking for column 4X
   for(int i = 0; i < 3; i++)
   {
        for(int j = 0; j < 7; j++)
        {
            if('X' == ob->board[i][j] && 'X' == ob->board[i+1][j] && 'X' == ob->board[i+2][j] &&
'X' == ob->board[i+3][j])
            {
                X4++;
            }
        }
   }

   // checking for column 3X
   for(int i = 0; i < 2; i++)
   {
        for(int j = 0; j < 7; j++)
        {
            if('X' == ob->board[i][j] && 'X' == ob->board[i+1][j] && 'X'== ob->board[i+2][j])
```

```
            {
                X3++;
            }
        }
    }

    // Checking for column 2X
    for(int i = 0; i < 1; i++)
    {
        for(int j = 0;j < 7; j++)
        {
            if('X' == ob->board[i][j] && 'X' == ob->board[i+1][j])
            {
                X2++;
            }
        }
    }

    // Checking for column 3O
    for(int i = 0; i < 2; i++)
    {
        for(int j = 0; j < 7; j++)
        {
            if('O' == ob->board[i][j] && 'O' == ob->board[i+1][j] && 'O'== ob->board[i+2][j])
            {
                O3++;
            }
        }
    }

    // Checking for column 2O
    for(int i = 0; i < 1; i++)
    {
        for(int j = 0; j < 7; j++)
        {
            if('O' == ob->board[i][j] && 'O' == ob->board[i+1][j])
            {
                O2++;
            }
        }
    }

    // Checking for diagonal (Positive slope) 4X
    for(int i = 0; i < 3; i++)
    {
        for(int j = 0; j < 4; j++)
```

```
        {
          if('X' == ob->board[i][j] && 'X' == ob->board[i+1][j+1] && 'X' == ob->board[i+2][j+2]
&& 'X' == ob->board[i+3][j+3])
          {
            X4++;
          }
        }
    }
    // Checking for diagonal (Positive slope) 3X
    for(int i = 0; i < 2; i++)
    {
        for(int j = 0; j < 3;j++)
        {
          if('X' == ob->board[i][j] && 'X' == ob->board[i+1][j+1] && 'X' == ob->board[i+2][j+2])
          {
            X3++;
          }
        }
    }
    // Checking for diagonal (positive slope) 2X
    for(int i = 0; i < 1; i++)
    {
        for(int j = 0; j < 2;j++)
        {
          if('X' == ob->board[i][j] && 'X' ==ob->board[i+1][j+1])
          {
            X2++;
          }
        }
    }
    // Checking for diagonal (Positive slope) 3O
    for(int i = 0; i < 2; i++)
    {
        for(int j = 0; j < 3;j++)
        {
          if('O' == ob->board[i][j] && 'O' == ob->board[i+1][j+1] && 'O' == ob->board[i+2][j+2])
          {
            O3++;
          }
        }
    }
    // Checking for diagonal (positive slope) 2O
    for(int i = 0; i < 1; i++)
    {
        for(int j = 0; j < 2;j++)
        {
```

```
            if('O' == ob->board[i][j] && 'O' == ob->board[i+1][j+1])
            {
                O2++;
            }
        }
    }
    // Checking for diagonal (Negative slope) 4X
    for(int i = 3; i < 6; i++)
    {
        for(int j = 4; j < 7; j++)
        {
            if('X' == ob->board[i][j] && 'X' == ob->board[i-1][j+1] && 'X' == ob->board[i-2][j+2]
&& 'X' == ob->board[i-3][j+3])
            {
                X4++;
            }
        }
    }
    // Checking for diagonal (Negative slope) 3X
    for(int i = 4; i < 6; i++)
    {
        for(int j = 5; j < 7; j++)
        {
            if('X' == ob->board[i][j] && 'X' == ob->board[i-1][j+1] && 'X' == ob->board[i-2][j+2])
            {
                X3++;
            }
        }
    }
    // Checking for diagonal (Negative slope) 2X
    for(int i = 5; i < 6; i++)
    {
        for(int j = 6; j < 7; j++)
        {
            if('X' == ob->board[i][j] && 'X' == ob->board[i-1][j+1])
            {
                X2++;
            }
        }
    }
    // Checking for diagonal (Negative slope) 3O
    for(int i = 4; i < 6; i++)
    {
        for(int j = 5; j < 7; j++)
        {
            if('O' == ob->board[i][j] && 'O' == ob->board[i-1][j+1] && 'O' == ob->board[i-2][j+2])
```

```
            {
                O3++;
            }
        }
    }
    // Checking for diagonal (Negative slope) 2O
    for(int i = 5; i < 6; i++)
    {
        for(int j = 6; j < 7; j++)
        {
            if('O' == ob->board[i][j] && 'O' == ob->board[i-1][j+1])
            {
                O2++;
            }
        }
    }
    eval = ( X4 * 100000 + X3 * 100 + X2 * 10 ) - (O3 * 100 + O2 * 10);
    return eval;
}

//harsha's eval fn
int tree::evaluation_h(){

    int value = 0;

    if(ob->player == 'X'){
        for(int i=0;i<6;i++){
            for(int j=0;j<4;j++){
                //for continuous row
                //first checking whether not everything is empty
                if(ob->board[i][j]!= ' ' || ob->board[i][j+1]!= ' ' || ob->board[i][j+2]!= ' ' || ob-
>board[i][j+3]!= ' '){
                    if((ob->board[i][j] == 'X' && ob->board[i][j+1] == 'X') && (ob->board[i][j+2]=='
')){
                        value = 10;
                    }
                }
            }
        }

        for(int i=0;i<3;i++){
            for(int j=0;j<7;j++){
                //for continuous col
                //first checking whether not everything is empty
                if(ob->board[i][j]!= ' ' || ob->board[i][j+1]!= ' ' || ob->board[i][j+2]!= ' ' || ob-
>board[i][j+3]!= ' '){
```

```
                    if((ob->board[i][j] == 'X' && ob->board[i+1][j] == 'X') && (ob->board[i+2][j]=='
'))){
                        value = 10;
                    }
                }
            }
        }

        //for lhs diagonal win
        for(int r=0;r<3;r++){
            for(int c=6;c>2;c--){
                char c1 = ob->board[r][c];
                char c2 = ob->board[r+1][c-1];
                char c3 = ob->board[r+2][c-2];
                char c4 = ob->board[r+3][c-3];
                //board[r][c]==board[r+1][c-1]==board[r+2][c-2]==board[r+3][c-3]
                //cout<<board[r][c]<<"          "<<board[r+1][c-1]<<"          "<<board[r+2][c-2]<<"
"<<board[r+3][c-3]<<endl;
                if(c1!= ' ' || c2!= ' ' || c3!= ' ' || c4!= ' '){
                    if((c1 == 'X' && c2 == 'X') || (c2 == 'X' && c3 == 'X') || (c3 == 'X' && c4 == 'X')){
                        //cout<<"win"<<endl;
                        value = 15;
                    }
                }
            }
        }

        //for rhs diagonal win
        for(int r=0;r<3;r++){
            for(int c=0;c<4;c++){
               char c1 = ob->board[r][c];
               char c2 = ob->board[r+1][c+1];
               char c3 = ob->board[r+2][c+2];
               char c4 = ob->board[r+3][c+3];
                //board[r][c]==board[r+1][c+1]==board[r+2][c+2]==board[r+3][c+3]
                //cout<<board[r][c]<<"          "<<board[r+1][c-1]<<"          "<<board[r+2][c-2]<<"
"<<board[r+3][c-3]<<endl;
                if(c1!=' '||c2!=' '||c3!=' '||c4!=' '){
                    if((c1 == 'X' && c2 == 'X') || (c2 == 'X' && c3 == 'X') || (c3 == 'X' && c4 == 'X')){
                        //cout<<"win"<<endl;
                        value = 15;
                    }
                }
            }
        }
    }
```

```
    else if(ob->player == 'O'){
            for(int i=0;i<6;i++){
                for(int j=0;j<4;j++){
                    //for continuous row
                    //first checking whether not everything is empty
                    if(ob->board[i][j]!= ' ' || ob->board[i][j+1]!= ' ' || ob->board[i][j+2]!= ' ' || ob->board[i][j+3]!= ' '){
                        if((ob->board[i][j] == 'O' && ob->board[i][j+1] == 'O') && (ob->board[i][j+2]==' ')){
                            value = 10;
                        }
                    }
                }
            }
            for(int i=0;i<3;i++){
            for(int j=0;j<7;j++){
                //for continuous col
                //first checking whether not everything is empty
                if(ob->board[i][j]!= ' ' || ob->board[i][j+1]!= ' ' || ob->board[i][j+2]!= ' ' || ob->board[i][j+3]!= ' '){
                    if((ob->board[i][j] == 'O' && ob->board[i+1][j] == 'O') && (ob->board[i+2][j]==' ')){
                        value = 10;
                    }
                }
            }
        }

        //for lhs diagonal win
        for(int r=0;r<3;r++){
            for(int c=6;c>2;c--){
                char c1 = ob->board[r][c];
                char c2 = ob->board[r+1][c-1];
                char c3 = ob->board[r+2][c-2];
                char c4 = ob->board[r+3][c-3];
                //board[r][c]==board[r+1][c-1]==board[r+2][c-2]==board[r+3][c-3]
                //cout<<board[r][c]<<"            "<<board[r+1][c-1]<<"            "<<board[r+2][c-2]<<"            "<<board[r+3][c-3]<<endl;
                if(c1!= ' ' || c2!= ' ' || c3!= ' ' || c4!= ' '){
                    if((c1 == 'O' && c2 == 'O') || (c2 == 'O' && c3 == 'O') || (c3 == 'O' && c4 == 'O')){
                        //cout<<"win"<<endl;
                        value = 15;
                    }
                }
            }
```

```
        }

        //for rhs diagonal win
        for(int r=0;r<3;r++){
            for(int c=0;c<4;c++){
                char c1 = ob->board[r][c];
                char c2 = ob->board[r+1][c+1];
                char c3 = ob->board[r+2][c+2];
                char c4 = ob->board[r+3][c+3];
                //board[r][c]==board[r+1][c+1]==board[r+2][c+2]==board[r+3][c+3]
                //cout<<board[r][c]<<"              "<<board[r+1][c-1]<<"              "<<board[r+2][c-2]<<"
"<<board[r+3][c-3]<<endl;
                if(c1!=' '||c2!=' '||c3!=' '||c4!=' '){
                    if((c1 == 'O' && c2 == 'O') || (c2 == 'O' && c3 == 'O') || (c3 == 'O' && c4 == 'O')){
                        //cout<<"win"<<endl;
                        value = 15;
                    }
                }
            }
        }

    }
    this->set_heuristic_value(value);
    //cout<<value<<endl;
    return value;
}

/* this function copies one nodes board configuration
    to the other nodes board it the board's configuration
    that needs to be copied as a reference parameter
    passed to the function and copies all the contents of the
    rhs board object to the lhs board ob
*/
void tree::copy_board_status(tic &tc){

    for(int i=0;i<ob->row_size;i++){
        for(int j=0;j<ob->col_size;j++){
            this->ob->board[i][j] = tc.board[i][j];
        }
    }

    for(int i=0;i<7;i++){
        this->ob->moves[i].col_index = tc.moves[i].col_index;
        this->ob->moves[i].row_index = tc.moves[i].row_index;
    }
}
```

```
/* this function return a particular child node
   which has the same heuristic value as that of
   its parent node i.e it return the optimal child
   node's index (i.e 'i' in our case) that is
   most probable to win in the future
*/
int tree::getOptimalNode(){

    for(int i=0;i<7;i++){
        if(children[i]->heuristic_value == this->heuristic_value && this->heuristic_value != -2000)
            return i;
    }
    return -1;
}

/* this function checks whether the terminal node has been reached
   and calls display_contents function
*/
void tree::helper(){

    char c1,c2,c3,c4;

    for(int r=0;r<3;r++){
        for(int c=6;c>2;c--){
            c1 = ob->board[r][c];
            c2 = ob->board[r+1][c-1];
            c3 = ob->board[r+2][c-2];
            c4 = ob->board[r+3][c-3];
            //board[r][c]==board[r+1][c-1]==board[r+2][c-2]==board[r+3][c-3]
            if((c1 == c2 && c2 == c3 && c4 == ' ' && c1 == this->ob->player && this->ob-
>board[r+2][c-3]!= ' ')){
                //cout<<"RHS matched"<<endl;
                this->ob->board[r+3][c-3] = this->ob->player;
                display_contents();
                exit(0);
            }
        }
    }

    //looking form left end of array lhs-diagonal
    for(int r=0;r<3;r++){
        for(int c=0;c<4;c++){
            c1 = ob->board[r][c];
            c2 = ob->board[r+1][c+1];
            c3 = ob->board[r+2][c+2];
```

```
            c4 = ob->board[r+3][c+3];
            //board[r][c]==board[r+1][c+1]==board[r+2][c+2]==board[r+3][c+3]
            if((c1 == c2 && c2 == c3 && c4 == ' ' && c1 == this->ob->player && this->ob->board[r+2][c-3]!= ' ')){
                //cout<<"LHS matched"<<endl;
                this->ob->board[r+3][c+3] = this->ob->player;
                display_contents();
                exit(0);
            }
        }
    }
}

/* this function calls the above getOptimalNode function
   stores the child's index and copies this child node's
   board configuration to other created board by using
   overloaded = operator to copy or else when a terminal node
   i.e a win state is reached it calls the helper function
*/
void tree::move_gen(tic *board_ob,int moves_made){
    int child_number = this->getOptimalNode();

    if(child_number != -1){
        //since child number can be 0 - 6 so counter=-1
        *(board_ob) = *(this->children[child_number]->ob);
        if(moves_made == 2 || moves_made == 6 || moves_made == 10 || moves_made == 15){
            cout<<"Board state after "<<moves_made<<" moves:"<<endl;
            board_ob->display_board(board_ob);
        }
    }

    else if(child_number == -1)
        this->helper();
}


/* this function is used for displaying the
   results of the connect-four game
*/
void tree::display_contents(){

   int stop_time = clock();
   int total_time = (stop_time - start_time);
   double memory_used = 0.0;

   cout<<this->ob->player<<" Won!"<<endl;
```

```
   cout<<endl;
   cout<<"Terminal(win) state of the board:"<<endl;
   ob->display_board(ob);
   cout<<endl;
   cout<<"Number of nodes generated: "<<num_nodes_generated<<endl;
   // +1 is for the root node since it is also expanded from the empty board
   cout<<"Number of nodes expanded: "<< num_nodes_expanded + 1 <<endl;
   cout<<"Total length of game path: "<<game_path_length<<endl;
   cout<<"Memory size used by 1 node is: 268 bytes"<<endl;
   cout<<"Total size of memory used by program: "<<(268*num_nodes_generated)<<" bytes"<<endl;
   cout<<"Total execution Time: "<<(total_time) / double(CLOCKS_PER_SEC)<<"'s"<<endl;

}
```

6.6 MinmaxAB.h

```
#ifndef MINMAXAB_H_INCLUDED
#define MINMAXAB_H_INCLUDED


#include "TreeNodes.h"

class minmaxA{

public:

    //member variable
    char eval_choice;

    //Member functions of this class
    minmaxA();
    int minmaxAB(tree *,int,int,int,char);


};

#endif // MINMAXAB_H_INCLUDED
```

6.7 MinmaxAB.cpp

```
#include<iostream>

#include "MinmaxAB.h"
#include "ConnectFourBoard.h"
#include "TreeNodes.h"
using namespace std;

minmaxA::minmaxA(){
   eval_choice = ' ';
}

/* this function is implemented from the algorithm given
  in Rich & Knight  text book.
  this function is used to find the node which is more probable to win
  in the future if the player plays the move returned by this function.

  in this function there are 2 cases 1st being base case which is executed
  when the depth cut-off that is reached, 2nd being recursive case that
  iterates through the loop for all the 7 children and recursively calls
  itself again and again until the base case condition evaluates to be true

  More complete description of how this function works is explained in project report
*/
int minmaxA::minmaxAB(tree *tob,int depth,int use_threshold,int pass_threshold,char player){

   int value;
   int new_value;
   char new_player;

   //gets executed if my eval function is selected by user
   if(eval_choice == 'M' || eval_choice == 'm'){
      if(tob->deep_enough(depth)){
         value = tob->evaluation_m();
         if(player == 'O')
            value = -value;
            tob->set_heuristic_value(value);
         return value;
      }
   }

   //gets executed if santosh's eval fn is chosen
   else if(eval_choice == 'S' || eval_choice == 's'){
      if(tob->deep_enough(depth)){
         value = tob->evaluation_s();
```

```
            if(player == 'O')
                value = -value;
                tob->set_heuristic_value(value);
            return value;
        }
    }

    //gets executed if harsha's eval fn is used
    else{
        if(tob->deep_enough(depth)){
            value = tob->evaluation_h();
            if(player == 'O')
                value = -value;
                tob->set_heuristic_value(value);
            return value;
        }
    }



    int value1=0;
    for(int i=0;i<7;i++){
        //checks the board player of the tob node and not the tob->children[] nodes
        if(player == 'X')
            new_player = 'O';
        if(player == 'O')
            new_player = 'X';

        value1 = minmaxAB(tob->children[i],depth+1,-pass_threshold,-use_threshold,new_player);
        new_value = -value1;

        if(new_value > pass_threshold){
            //holds the optimal nodes board config tree ob
             tob->set_heuristic_value(i);
             pass_threshold = new_value;
        }

        if(pass_threshold >= use_threshold){
            value1 = pass_threshold;
            return value1;
        }
    }
        value1 = pass_threshold;
        return value1;
}
```

6.8 AlphaBeta.h

```
#ifndef ALPHABETA_H_INCLUDED
#define ALPHABETA_H_INCLUDED

#include <iostream>
#include "TreeNodes.h"
using namespace std;

class alphabeta{

public:

    //member variables
     char eval_choice;

    //Member functions of alphabeta class
    alphabeta();
    int alpha_beta(tree*, int, char, int, int);


};
#endif // ALPHABETA_H_INCLUDED
```

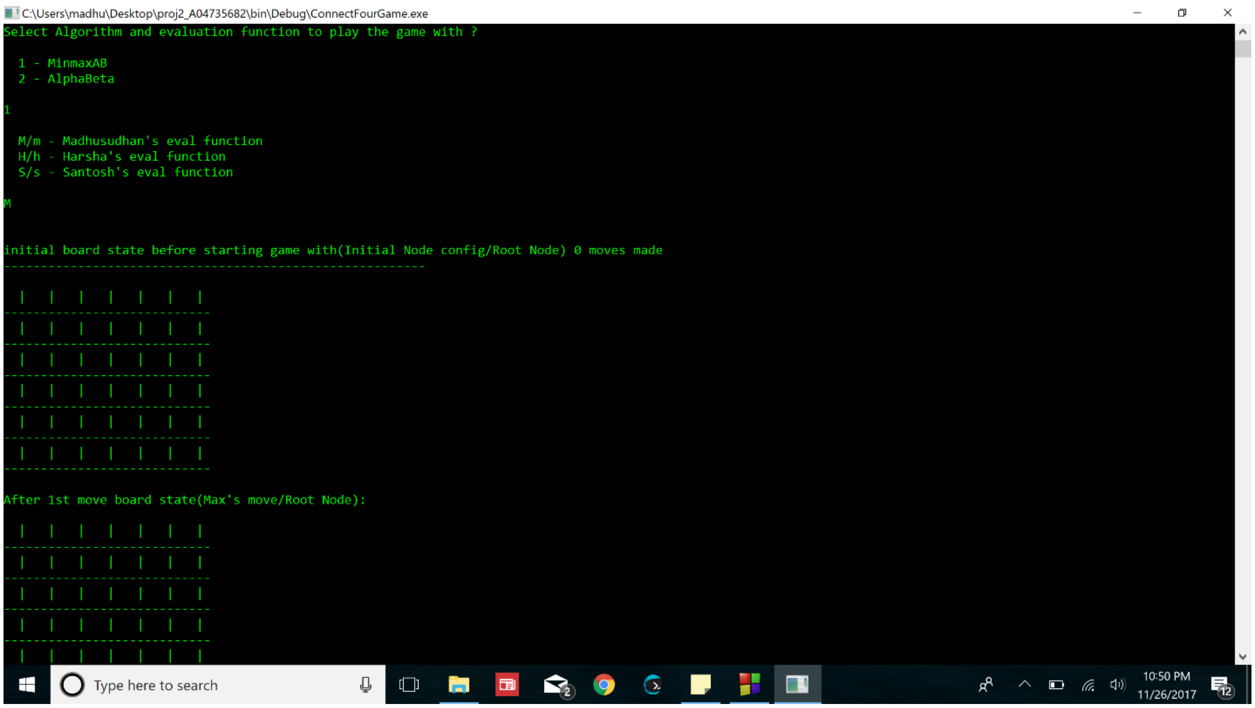
6.9 AlphaBeta.cpp

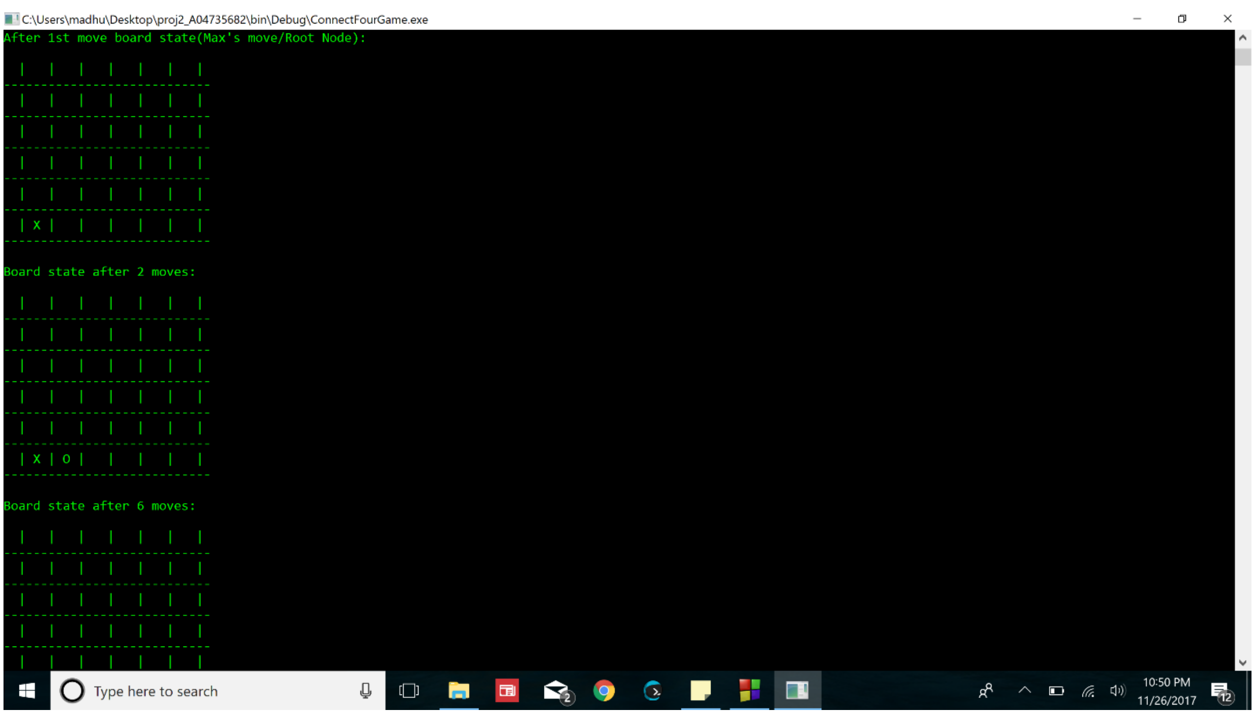
```
#include <iostream>

#include "AlphaBeta.h"
#include "TreeNodes.h"
#include "ConnectFourBoard.h"
using namespace std;

//a constructor with no functionality
alphabeta::alphabeta(){
    eval_choice = ' ';
}

/* this function is implemented from the algorithm given
   in Russell & Norvig text book.
   this function is used to find the node which is more probable to win
   in the future if the player plays the move returned by this function.

   in this function there are 3 cases 1st being base case , 2nd being recursive case
   for player 'X' i.e Max player and the 3rd being recursive case for player 'Y'
   i.e Min player

   More complete description of how this function works is explained in project report
*/
int alphabeta::alpha_beta(tree* tob, int depth, char player, int alpha, int beta){

    //gets executed if my eval function is selected by user
    if(eval_choice == 'M' || eval_choice == 'm'){
        if(tob->deep_enough(depth))
            return tob->evaluation_m();
    }

    //gets executed if santosh's eval fn is chosen
    else if(eval_choice == 'S' || eval_choice == 's'){
            if(tob->deep_enough(depth))
                return tob->evaluation_s();
    }

    //gets executed if harsha's eval fn is used
    else{
        if(tob->deep_enough(depth))
            return tob->evaluation_h();
    }
```

```
    if(player == 'X'){
        int best_value = -100,value;
        for(int i=0;i<7;i++){
            if(tob->children[i] == NULL)
                continue;
            value = alpha_beta(tob->children[i],depth+1,player,alpha,beta);
            best_value = (best_value > value)? best_value:value;
            alpha = (alpha > best_value)? alpha:best_value;

            if(beta <= alpha)
                break;
        }
        tob->set_heuristic_value(best_value);
        return best_value;
    }

    else{
        int best_value = +100,value;
        for(int i=0;i<7;i++){
            if(tob->children[i] == NULL)
                continue;
            value = alpha_beta(tob->children[i],depth+1,player,alpha,beta);
            best_value = (best_value < value)? best_value:value;
            beta = (beta < best_value)? beta:best_value;

            if(beta < alpha)
                break;
        }
        tob->set_heuristic_value(best_value);
        return best_value;
    }
}
```

7. OUTPUT (using my [Madhusudhan] evaluation function)

7.1 Output screen shots for Minmax-AB (using my evaluation function for depth - 2):





C:\Users\madhu\Desktop\proj2_A04735682\bin\Debug\ConnectFourGame.exe

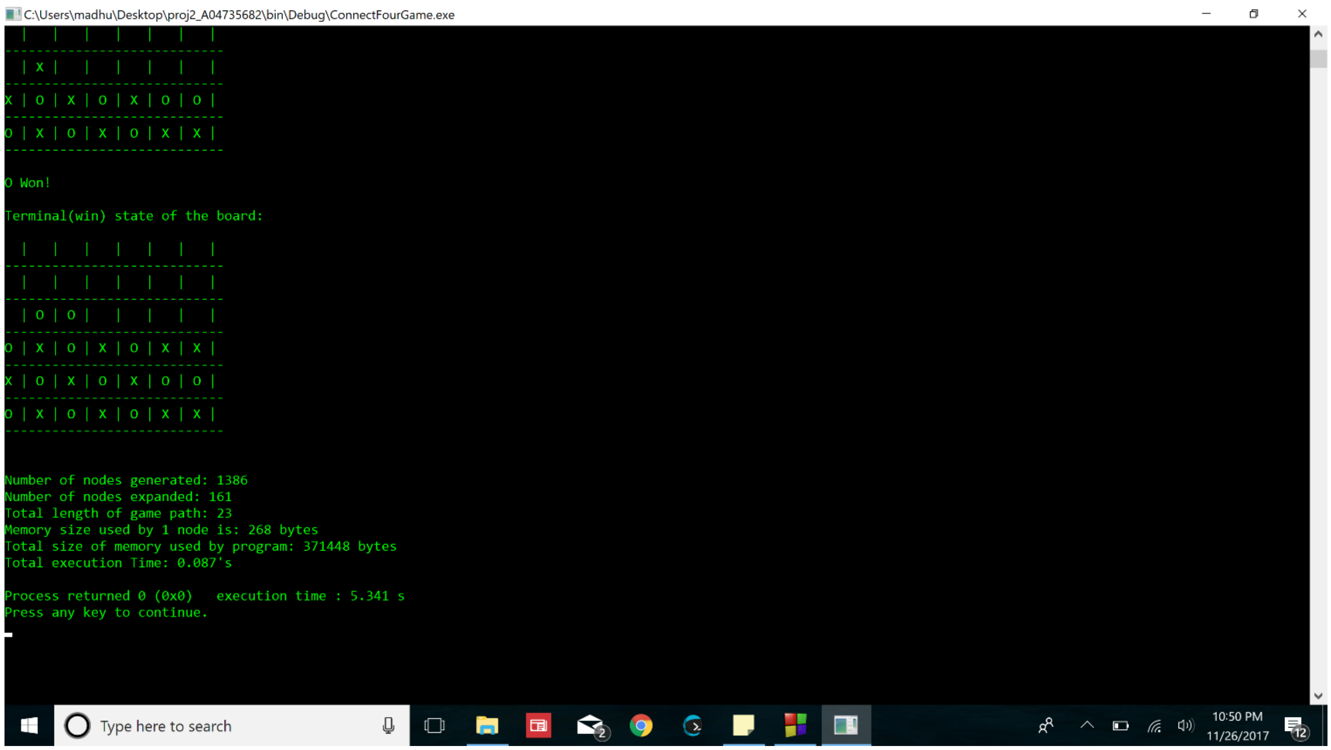
```
Board state after 6 moves:

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

Board state after 10 moves:

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

Board state after 15 moves:

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

Type here to search 10:50 PM 11/26/2017

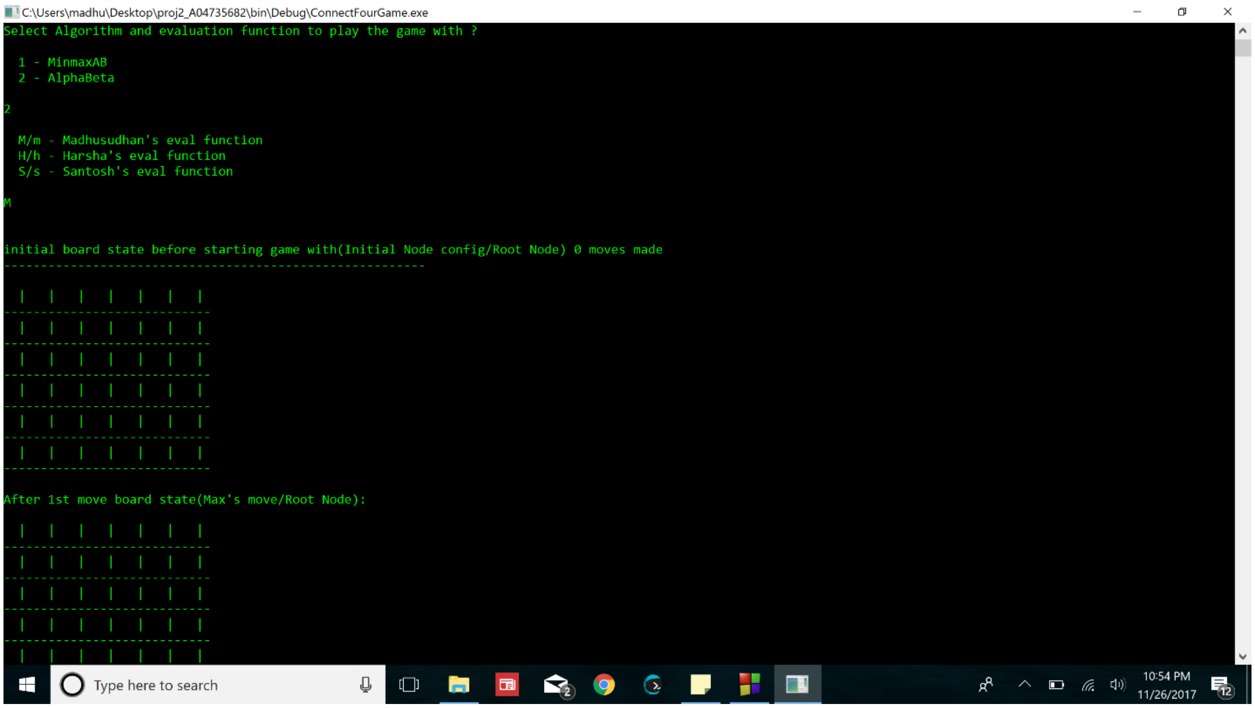
C:\Users\madhu\Desktop\proj2_A04735682\bin\Debug\ConnectFourGame.exe

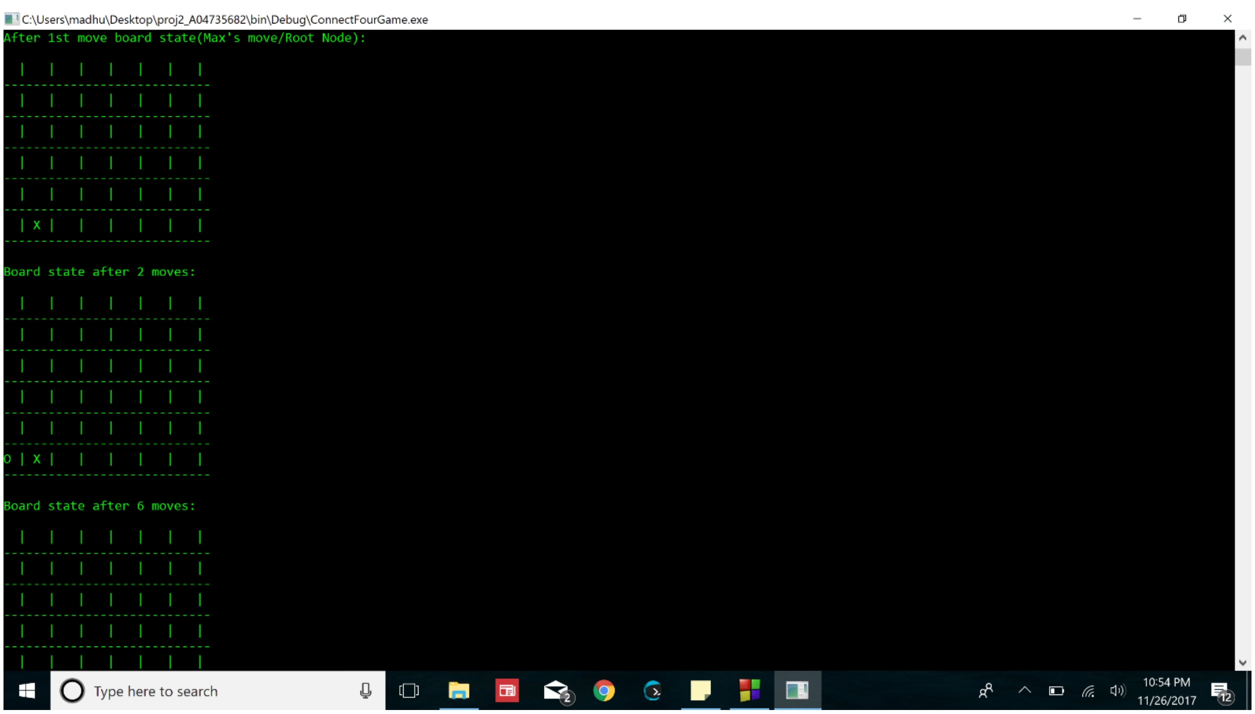
```
Board state after 15 moves:

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

O Won!

Terminal(win) state of the board:

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


Number of nodes generated: 1386
Number of nodes expanded: 161
Total length of game path: 23
Memory size used by 1 node is: 268 bytes
Total size of memory used by program: 371448 bytes
Total execution Time: 0.087's

Process returned 0 (0x0)   execution time : 5.341 s
```

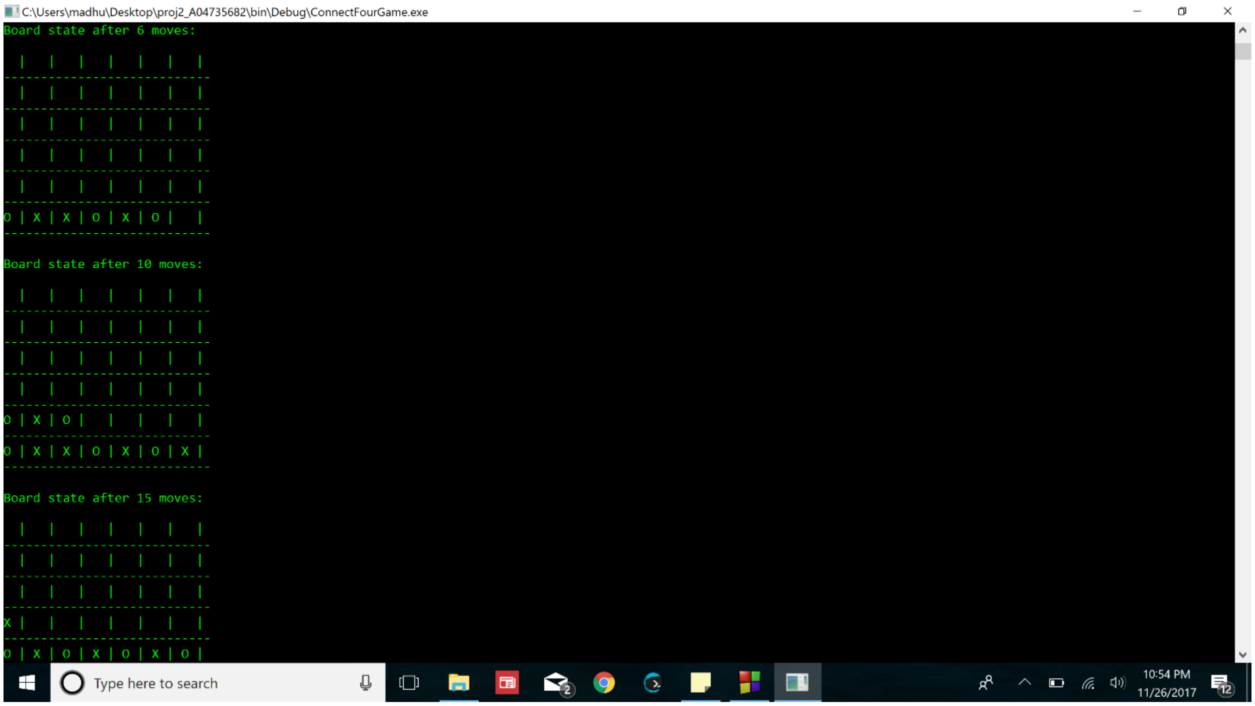
Type here to search 10:50 PM 11/26/2017


C:\Users\madhu\Desktop\proj2_A04735682\bin\Debug\ConnectFourGame.exe
| X |
X | O | X | O | X | O | O |
O | X | O | X | O | X | X |
O Won!
Terminal(win) state of the board:
| O | O |
O | X | O | X | O | X | X |
X | O | X | O | X | O | O |
O | X | O | X | O | X | X |
Number of nodes generated: 1386
Number of nodes expanded: 161
Total length of game path: 23
Memory size used by 1 node is: 268 bytes
Total size of memory used by program: 371448 bytes
Total execution Time: 0.087's
Process returned 0 (0x0)   execution time : 5.341 s
Press any key to continue.
Type here to search
10:50 PM
11/26/2017

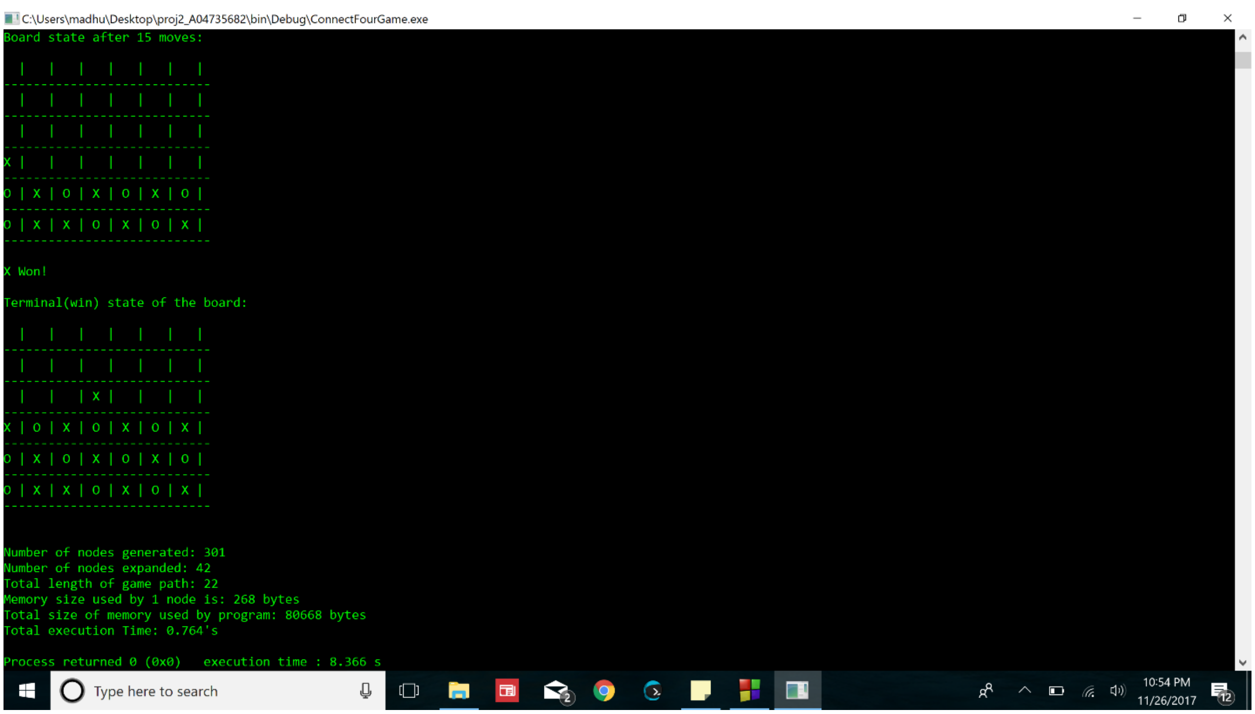
7.2 Output screen shots for Alpha-Beta-Search (using my evaluation function for depth - 2):



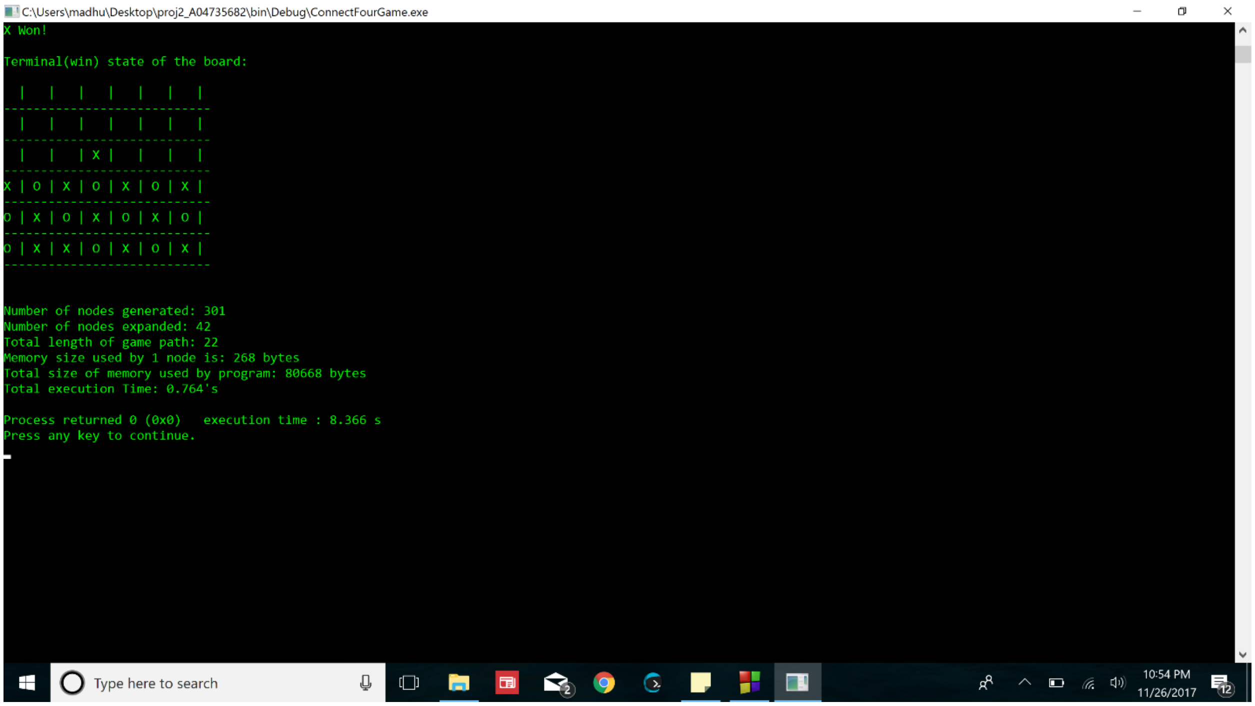




```
C:\Users\madhu\Desktop\proj2_A04735682\bin\Debug\ConnectFourGame.exe
Board state after 6 moves:

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

Board state after 10 moves:

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

Board state after 15 moves:

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



```
C:\Users\madhu\Desktop\proj2_A04735682\bin\Debug\ConnectFourGame.exe
Board state after 15 moves:

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

X Won!

Terminal(win) state of the board:

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


Number of nodes generated: 301
Number of nodes expanded: 42
Total length of game path: 22
Memory size used by 1 node is: 268 bytes
Total size of memory used by program: 80668 bytes
Total execution Time: 0.764's

Process returned 0 (0x0)   execution time : 8.366 s
```


C:\Users\madhu\Desktop\proj2_A04735682\bin\Debug\ConnectFourGame.exe
X Won!
Terminal(win) state of the board:
		X			
X | O | X | O | X | O | X |
O | X | O | X | O | X | O |
O | X | X | O | X | O | X |
Number of nodes generated: 301
Number of nodes expanded: 42
Total length of game path: 22
Memory size used by 1 node is: 268 bytes
Total size of memory used by program: 80668 bytes
Total execution Time: 0.764's
Process returned 0 (0x0)   execution time : 8.366 s
Press any key to continue.
Type here to search
10:54 PM
11/26/2017

8. ANALYSIS 1 (using my [Madhusudhan] evaluation function)

8.1 Comparison of algorithms when depth of search tree is 2:

The output of MinmaxAB search algorithm

O Won!

Number of nodes generated: 1358

Number of nodes expanded: 166

Total length of game path: 23

Memory size used by 1 node is: 268 bytes

Total size of memory used by program: 363944 bytes

Total execution Time: 0.122's

Process returned 0 (0x0)   execution time : 5.707 s

Press any key to continue.

The output of AlphaBeta search algorithm

X Won!

Number of nodes generated: 301

Number of nodes expanded: 42

Total length of game path: 22

Memory size used by 1 node is: 268 bytes

Total size of memory used by program: 80668 bytes

Total execution Time: 0.079's

Process returned 0 (0x0)   execution time : 3.242 s

Press any key to continue.

	MinmaxAB algorithm	**AlphaBeta algorithm**
Total length of the game path	23	22
total number of nodes generated	1358	301
number of nodes expanded	166	42
Execution time (seconds)	0.122	0.079
Memory used by the program (bytes)	363944	80668

8.2 Comparison of algorithms when depth of search tree is 4:

The output of MinmaxAB search algorithm

X Won!

Number of nodes generated: 35791

Number of nodes expanded: 3599

Total length of game path: 38

Memory size used by 1 node is: 268 bytes

Total size of memory used by program: 9591988 bytes

Total execution Time: 0.303's

Process returned 0 (0x0)   execution time : 7.170 s

Press any key to continue.

The output of AlphaBeta search algorithm

O Won!

Number of nodes generated: 658

Number of nodes expanded: 91

Total length of game path: 25

Memory size used by 1 node is: 268 bytes

Total size of memory used by program: 176344 bytes

Total execution Time: 0.416's

Process returned 0 (0x0)   execution time : 3.807 s

Press any key to continue.

	MinmaxAB algorithm	**AlphaBeta algorithm**
Total length of the game path	38	25
total number of nodes generated	35791	658
number of nodes expanded	3599	91
Execution time (seconds)	0.303	0.416
Memory used by the program (bytes)	9591988	176344

9. ANALYSIS 2 (For Harsha's evaluation function)

9.1 Comparison of algorithms when depth of search tree is 2

The output of MinmaxAB search algorithm

X Won!

Number of nodes generated: 210

Number of nodes expanded: 25

Total length of game path: 3

Memory size used by 1 node is: 268 bytes

Total size of memory used by program: 56280 bytes

Total execution Time: 0.028's

Process returned 0 (0x0)   execution time : 3.415 s

Press any key to continue.

The output of AlphaBeta search algorithm

X Won!

Number of nodes generated: 161

Number of nodes expanded: 23

Total length of game path: 11

Memory size used by 1 node is: 268 bytes

Total size of memory used by program: 43148 bytes

Total execution Time: 0.21's

Process returned 0 (0x0)   execution time : 2.929 s

Press any key to continue.

	MinmaxAB algorithm	AlphaBeta algorithm
Total length of the game path	3	11
total number of nodes generated	210	161
number of nodes expanded	25	23
Execution time (seconds)	0.028	0.21
Memory used by the program (bytes)	56280	43148

9.2 Comparison of algorithms when depth of search tree is 4

The output of MinmaxAB search algorithm

X Won!

Number of nodes generated: 3150

Number of nodes expanded: 375

Total length of game path: 5

Memory size used by 1 node is: 268 bytes

Total size of memory used by program: 844200 bytes

Total execution Time: 0.063's

Process returned 0 (0x0)   execution time : 3.198 s

Press any key to continue.

The output of AlphaBeta search algorithm

X Won!

Number of nodes generated: 336

Number of nodes expanded: 47

Total length of game path: 12

Memory size used by 1 node is: 268 bytes

Total size of memory used by program: 90048 bytes

Total execution Time: 0.04's

Process returned 0 (0x0)   execution time : 3.054 s

Press any key to continue.

	MinmaxAB algorithm	AlphaBeta algorithm
Total length of the game path	5	12
total number of nodes generated	3150	336
number of nodes expanded	375	47
Execution time (seconds)	0.063	0.04
Memory used by the program (bytes)	844200	90048

10. ANALYSIS 3 (using Santosh's evaluation function)

10.1 Comparison of algorithms when depth of search tree is 2

The output of MinmaxAB search algorithm

O Won!

Number of nodes generated: 2709

Number of nodes expanded: 206

Total length of game path: 14

Memory size used by 1 node is: 268 bytes

Total size of memory used by program: 726012 bytes

Total execution Time: 0.041's

Process returned 0 (0x0)   execution time : 3.198 s

Press any key to continue.

The output of AlphaBeta search algorithm

X Won!

Number of nodes generated: 252

Number of nodes expanded: 36

Total length of game path: 12

Memory size used by 1 node is: 268 bytes

Total size of memory used by program: 67536 bytes

Total execution Time: 0.048's

Process returned 0 (0x0)   execution time : 3.198 s

Press any key to continue.

	MinmaxAB algorithm	**AlphaBeta algorithm**
Total length of the game path	14	12
total number of nodes generated	2709	252
number of nodes expanded	206	36
Execution time (seconds)	0.041	0.048
Memory used by the program (bytes)	726012	67536

10.2 Comparison of algorithms when depth of search tree is 4

The output of MinmaxAB search algorithm

X Won!

Number of nodes generated: 3390

Number of nodes expanded: 365

Total length of game path: 5

Memory size used by 1 node is: 268 bytes

Total size of memory used by program: 908520 bytes

Total execution Time: 0.019's

Process returned 0 (0x0)   execution time : 3.198 s

Press any key to continue.

The output of AlphaBeta search algorithm

X Won!

Number of nodes generated: 308

Number of nodes expanded: 43

Total length of game path: 11

Memory size used by 1 node is: 268 bytes

Total size of memory used by program: 82544 bytes

Total execution Time: 0.036's

Process returned 0 (0x0)   execution time : 3.054 s

Press any key to continue.

	MinmaxAB algorithm	**AlphaBeta algorithm**
Total length of the game path	5	11
total number of nodes generated	3390	308
number of nodes expanded	365	43
Execution time (seconds)	0.019	0.036
Memory used by the program (bytes)	908520	82544

11. RESULT OF ANALYSIS

By comparing the analysis 1 (that uses my evaluation function) with analysis 2 (uses Harsha's evaluation function) and analysis 3 (uses Santosh's evaluation function)
the following things can be inferred.

1. When the depth of the search tree in increased the amount of computation increases in all aspects i.e. number of nodes generated and expanded, total execution time, memory size used by the program.

2. When the depth of the search tree in increased both the players make more intelligent moves and efficiency of them, in choosing the most optimal node and then making the move according to it increases significantly.

3. The 2nd point made above can be proven to be correct form the 1st point above.
As there will be a increase in the number nodes generated and expanded, is one of the main factors that shows us that both are trying to make more intelligent moves when depth is increased.

4. The 3 evaluation functions used for analysis – 1, 2 and 3 shows us that the above points inferred is true because these points are all true in all the analysis that we have made.

12. CONCLUSION

The implementation of MinmaxAB and AlphaBeta search algorithms has showed me that there is a much significant importance of Artificial Intelligence in the field of games i.e. game theory also. It has also proved that the AlphaBeta search algorithm is better than MinMaxAB algorithm for game playing.

It has helped me to gain a clearer understanding and how these algorithms can be applied practically on our daily activities.
This has a lot of practical applications not only in Connect-Four but also other games where the search tree grows very large to even understand. In such cases these algorithms can be used /implemented to search the tree in a very less time than other algorithms in a simple manner and conclude which move is best suitable for this play in a game.

I would like to thank Dr.Moonis Ali for explaning these concepts and help me making understand the algorithms in class from which I could implement the algorithm of both MinmaxAB and AlphaBeta Search algorithms in this Connect-Four game project.

13. REFERENCES

1. Artificial intelligence, by E.Rich, K.Knight, K.B.Nair

2. Artificial Intelligence, A modern approach: by Peter Norvig and Stuart Russell

3. https://en.wikipedia.org/wiki/Connect_Four

4. https://en.wikipedia.org/wiki/Alpha–beta_pruning

5. https://en.wikipedia.org/wiki/Minimax