Project 2
Checkers Game Using Minmax AB and Alpha Beta Search AI Methods

CS 5346: ADVANCED ARTIFICIAL INTELLIGENCE

Udaya Ramya Godavarthy
TEXAS STATE UNIVERSITY | TXST ID : A05070102

Table of Contents

Abstract

This project aims to implement two popular Artificial Intelligence (AI) methods Minmax-A-B from "Artificial Intelligence by Rich and Knight" and Alpha-Beta-Search from "Artificial Intelligence: A modern approach by Russel and Norvig". Using these techniques, a simple 8x8 checkers game is implemented in C++ programming language.

The main objective of the project is to provide a clear description of these algorithms, Deep-Enough, Move-Gen functions and optimal evaluation functions that will return a value for a node which will have a high chance of winning the game.

To devise these methodologies first, a general description of the Checkers game is provided to understand the moves and winning strategy. Then, the program is executed using every possible combinations of evaluation functions and the two AI methods. Finally, analysis of the results is presented that shows the performance of each algorithm.

Keywords: *artificial intelligence, Minmax-AB, Alpha-Beta-Search, evaluation function*

Introduction

Game theory is a branch of mathematics used to model the interaction between different players in a context with predefined rules and outcomes. It is an extensively explored field of computer science and can be applied in different ambit of artificial intelligence such as multi-agent AI systems, imitation and reinforcement learning, and adversary training in generative adversarial networks (GANs).

Games involving two players requires a large set of moves. It is very difficult to provide a win with brute force method. Therefore, various AI methods are used to get an effective move with less time and with minimum nodes. For example, in the game of chess each player on an average requires 30 moves per player. This will require a search tree of around 3080 positions. Hence, there should a technique which is used to get the optimal solution with less complexity for such games.

This project will provide two Algorithms – Alpha-Beta-Search and Minimax-AB which are commonly used techniques for increasing efficiency of two player games. We implemented these algorithms using Checkers games and analyzed the performance of each method.

Problem Statement

Develop programs by implementing algorithms MINMAX-A-B (Rich & Knight) and ALPHA-BETA-SEARCH (Russell & Norvig) in C or C++ language. Devise Deep-Enough and Move-Gen functions. Use Checkers game as an example to test the programs. Devise at least one evaluation function per person.

Execute the programs for two depths (e.g., 2 and 4) and analyze and tabulate the performance of each algorithm, each depth, and each evaluation function by tabulating the total length of the game path, total number of nodes generated and expanded, execution time, the size of memory used by the program, and winning/losing statistics.

The program should play with a computer to a computer as follows.:

Four runs with depth 2:

- MinMax-A-B with Evl. Function #1 **Verses** Alpha-Beta with Evl. Function #1
- MinMax-A-B with Evl. Function #2 **Verses** Alpha-Beta with Evl. Function #2
- MinMax-A-B with Evl. Function #1 **Verses** MinMax-A-B with Evl. Function #2
- Alpha-Beta with Evl. Function #1 **Verses** Alpha-Beta with Evl. Function #2

Four runs with depth 4:

- MinMax-A-B with Evl. Function #1 **Verses** Alpha-Beta with Evl. Function #1
- MinMax-A-B with Evl. Function #2 **Verses** Alpha-Beta with Evl. Function #2
- MinMax-A-B with Evl. Function #1 **Verses** MinMax-A-B with Evl. Function #2
- Alpha-Beta with Evl. Function #1 **Verses** Alpha-Beta with Evl. Function #2

Analyze the results and determine which algorithm and which evaluation function performs better for which depth.

We are expected to develop the program as a team. After the program is developed and tested for a sample example, each member of the team will work separately to do the following.

- Choose an evaluation function.
- Execute the programs and generate his/her tables.
- You must include execution displays in the report
- Analyze the results and write the conclusions you draw from these analyses.
- Write his/her project report explaining the problem, domain, methodology, source code implementation, source code. description of evaluation functions, any special approaches used, display of executions, the paths generated by your program for each run, analyses of the program, analysis of results, and conclusions.
- Demonstrate the execution of the program, if asked.

Team Members and Respective Contributions

Team Members:

Udaya Ramya Godavarthy, Govind C Narra, and Benila Jerald Xavier

Designing Stage:

In designing stage, we all came together to understand the project details. Numerous brain-storming sessions were done to gather the necessary information.

- We first understood the checkers game play, its rules and saw some sample AI games available online.
- We then revised the algorithms from the modules taught in class and from the textbooks mentioned in the project description.
- After this, we came up with the necessary functions required for implementation and divided the work accordingly.

Programming Stage:

In programming stage, we came up with the following main functions:

(a) **Designing the checkers board function**: The initial implementation of the board was designed by Govind. After that it was modified by Benila and myself by running the program with some sample inputs.

(b) **Minmax AB Algorithm**: MinMax AB algorithm from Rich and Knight book was implemented by Benila.

(c) **Alpha-Beta Search Algorithm:** I worked on designing the Alpha-Beta search algorithm from Russel and Norvig.

(d) **Evaluation Functions**: Each one of us came up with three evaluation functions. Evaluation Function 1 was designed by Govind, 2nd evaluation function by Benila and I made the 3rd evaluation function.

(d) **Game Play**: A computer vs computer function is implemented by me that plays the game using all combinations of evaluation functions and algorithms.

Finally, we all came together to write the main function and did the analysis of the program.

Min Max AB Algorithm from Rich and Knight

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

1. *If DEEP-ENOUGH(Position, Depth), then return the structure VALUE = STATIC (Position, Player);*
   PATH = nil
2. Otherwise, generate one more ply of the tree by calling the function MOVEGEN(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, OPPOSITE *(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 he appropriate pathfrom 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

Alpha-Beta Search Algorithm from Russel and Norvig

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))

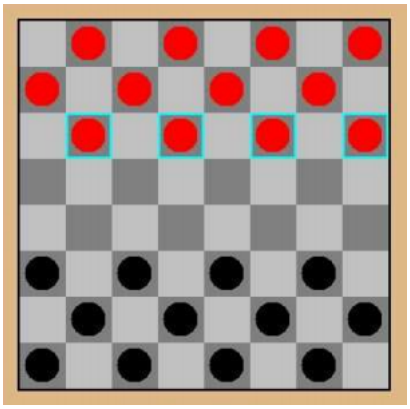
if v ≤ α **then return** v

β←MIN(β, v)

return v

Checkers Game

Checkers is a very popular game all over the world. The first attempts to build the first English draughts computer program were in the early 1950s. In 2007, it was published that the program "Chinook" was able to "solve" the 8X8 board from all possible positions.



The game of checkers is considered a complicated game with 10^20 possible legal positions in the English draughts version (8*8 board) alone (much more on higher dimensions).

Rules of the Game:

- Two players (we have considered 'X' and 'O')
- Computer 1 is 'X' and moves first and Computer 2 is 'O' and moves second.
- 12 pieces for each player which are arranged as shown in the figure.
- Pieces may only move diagonal space forward towards their opponent's pieces in the beginning of the game.
- An opponent's piece is captured by a jumping over it by moving two pieces diagonally in the direction of the opponent piece if it is empty.
- The captured opponent piece is removed from the board.
- When a piece reaches the opponent's side last row it is converted to king.
- King has an advantage moving forward and backward diagonally.
- There is no limit to how many king pieces a player may have.
- If all the opponent's pieces are captured the game ends or to limit the time the game ends after certain moves (say 100). The player with maximum number of pieces left wins.

N-ary Game Tree:

For finding the optimal move of the checker board that can be made by the computer, N-ary Game tree is generated based on the contemporary state of the checker board by fixing the depth. The N-ary Game tree is generated as follows.

Step 1: The initial node that is the root node consists of the contemporary state of the checker board.

Step 2: And the forthcoming levels of the tree consist of all possible moves that can be made by the machine and the opponent alternatively.

Step 3: Step 2 is repeated until the game tree of depth n gets generated.

Step 4: Heuristic value is calculated for all the leaf nodes of the game tree.

Evaluation Functions

An evaluation function, also known as a heuristic evaluation function or static evaluation function, is a function used by game-playing computer programs to estimate the value or goodness of a position (usually at a leaf or terminal node) in a game tree.

We made 3 evaluation functions one by each team member. Evaluation Function 1 was created by myself and evaluation 2 by Govind and evaluation 3 by Benila.

Evaluation Function 1 – By Ramya

The goal of the game is to capture all of the opponent's pieces or block all the moves made the opponent such that no more moves are left. Thus, I created an evaluation function which is a weighted linear function giving weights to different features of the game.

- **Piece Difference**: It is the sum of pieces of the max player – sum of pieces by opponent player giving weight 1 to this feature.
- **King Piece Difference**: It is sum of king pieces of the max player – sum of pieces by opponent player multiplied by 2 giving it a weight 2 for this feature.
- **Possible Moves**: This feature considers all forward possible moves that can be made towards opponent's direction by pawns and kings. Also, all possible moves by kings in backward direction. Giving a weight 3 to this feature.
- **Possible Captures**: This feature considers all captures possible by pawns and kings in all possible directions. Giving a weight 4 to this feature.

Hence, the overall evaluation function is computed by the following equation:

EVALUATION FUNCTION =

1* (Piece Difference) + 2*(KingPiece Difference) + 3*(Each Forward Move Difference + Each King Backward Move Difference) + 4*(Each Capture Difference + Each Capture in Backward By King)

```
if ((opt == Option1 && evalp1 == 1) || (opt == Option2 && evalp2 == 1))
    {
        for(int i=0;i<8;i++)
        {
            for(int j=0;j<8;j++)
            {
                // Piece Count
                if(CheckerBoard[i][j] == PLAYER1)
                    player1Count++;
                else if(CheckerBoard[i][j] == PLAYER2)
                    player2Count++;
                //King Piece Count
                if(CheckerBoard[i][j] == PLAYER1_KING)
                    player1Count += 2;
                else if (CheckerBoard[i][j] == PLAYER2_KING)
                    player2Count += 2;
```

```
                //Possible Moves towards Opponents Direction for Player 1
                if(CheckerBoard[i][j] == PLAYER1 || CheckerBoard[i][j] ==
PLAYER1_KING)
                {
                    if((CheckerBoard[i+1][j-1] == EMPTY_SPACE //checks
right diagonal
                       || CheckerBoard[i+1][j+1] == EMPTY_SPACE) // checks
left diagonal
                       && i+1 < 8 && j-1 > -1 && j+1 < 8)
                        player1Count += 3;

                    // Possible Capture towards Opponents Direction
                    if((CheckerBoard[i+1][j-1] == PLAYER2 ||
                        CheckerBoard[i+1][j-1] == PLAYER2_KING) &&
                       CheckerBoard[i+2][j-2] == EMPTY_SPACE &&
                       i+2 < 8 && i+1 < 8 && j-1 > -1 && j-2 > -1)
                        player1Count += 4;
                    if((CheckerBoard[i+1][j+1]==PLAYER2 ||
                        CheckerBoard[i+1][j+1] == PLAYER2_KING) &&
                        CheckerBoard[i+2][j+2] == EMPTY_SPACE &&
                       i+2 < 8 && i+1 < 8 && j+1 < 8 && j+2 < 8)
                        player1Count += 4;
                }

                // Moves in backward direction by king
                if(CheckerBoard[i][j] == PLAYER1_KING)
                {
                    if((CheckerBoard[i-1][j-1] == EMPTY_SPACE //checks
right diagonal
                       || CheckerBoard[i-1][j+1] == EMPTY_SPACE) // checks
left diagonal
                       && i-1 > -1 && j-1 > -1 && j+1 < 8)
                        player1Count += 3;

                    // Capture in backward direction by king
                    if((CheckerBoard[i-1][j-1] == PLAYER2 ||
                        CheckerBoard[i-1][j-1] == PLAYER2_KING) &&
                       CheckerBoard[i-2][j-2] == EMPTY_SPACE &&
                       i-2 > -1 && i-1 > -1 && j-1 > -1 && j-2 > -1)
                        player1Count += 4;
                    if((CheckerBoard[i-1][j+1] == PLAYER2 ||
                        CheckerBoard[i-1][j+1] == PLAYER2_KING) &&
                        CheckerBoard[i-2][j+2] == EMPTY_SPACE &&
                       i-2 > -1 && i-1 > -1 && j+1 < 8 && j+2 < 8)
                        player1Count += 4;
                }

                //Possible Moves towards Opponents Direction for Player 2
                if(CheckerBoard[i][j] == PLAYER2 || CheckerBoard[i][j] ==
PLAYER2_KING)
                {
                    if((CheckerBoard[i-1][j-1] == EMPTY_SPACE //checks
right diagonal
                       || CheckerBoard[i-1][j+1] == EMPTY_SPACE) // checks
left diagonal
```

```
                        && i-1 > -1 && j-1 > -1 && j+1 < 8)
                         player2Count += 3;
                    // Possible Capture towards Opponents Direction
                    if((CheckerBoard[i-1][j-1] == PLAYER1 ||
                         CheckerBoard[i-1][j-1] == PLAYER1_KING) &&
                        CheckerBoard[i-2][j-2] == EMPTY_SPACE &&
                        i-2 > -1 && i-1 < -1 && j-1 > -1 && j-2 > -1)
                         player2Count += 4;
                    if((CheckerBoard[i-1][j+1] == PLAYER1 ||
                         CheckerBoard[i-1][j+1] == PLAYER1_KING) &&
                         CheckerBoard[i-2][j+2] == EMPTY_SPACE &&
                        i-2 > -1 && i-1 < -1 && j+1 < 8 && j+2 < 8)
                         player2Count += 4;
                }
                // Moves in backward direction by king
                if(CheckerBoard[i][j] == PLAYER2_KING)
                {
                    if((CheckerBoard[i+1][j-1] == EMPTY_SPACE //checks
right diagonal
                        || CheckerBoard[i+1][j+1] == EMPTY_SPACE) // checks
left diagonal
                        && i+1 < 8 && j-1 > -1 && j+1 < 8)
                         player2Count += 3;

                         // Capture in backward direction by king
                    if((CheckerBoard[i+1][j-1] == PLAYER1 ||
                         CheckerBoard[i+1][j-1] == PLAYER1_KING) &&
                        CheckerBoard[i+2][j-2] == EMPTY_SPACE &&
                        i+2 < 8 && i+1 < 8 && j-1 > -1 && j-2 > -1)
                         player2Count += 4;
                    if((CheckerBoard[i+1][j+1]==PLAYER1 ||
                         CheckerBoard[i+1][j+1] == PLAYER1_KING) &&
                         CheckerBoard[i+2][j+2] == EMPTY_SPACE &&
                        i+2 < 8 && i+1 < 8 && j+1 < 8 && j+2 < 8)
                         player2Count += 4;
                }
            }
        }
    }
```

Evaluation Function - By Govind

Govind considered taking Player 1 is positive and Player 2 is negative, taking count of all pieces with more weightage towards kings and adding it with player count difference , how close player is to become king and some randomness.

```
    else if ((opt == 1 && evalp1 == 2) || (opt == 2 && evalp2 == 2))
    {
        int a1 = 0, a2 = 0, b = 0, c = 0;
        for (int i = 0; i < 8; i++)
            for (int j = 0; j < 8; j++)
            {
                if (CheckerBoard[i][j] == PLAYER1)
                {
```

```
                    a1 += 2;
                    if (i == 0)
                        b += 9;
                    else b += i;
                    c += 1;
                }
                else if (CheckerBoard[i][j] == PLAYER2)
                {
                    a2 -= 2;
                    if (i == 7)
                        b -= 9;
                    else b -= (7 - i);
                    c -= 1;
                }
                else if (CheckerBoard[i][j] == PLAYER1_KING)
                {
                    a1 += 3;
                    c += 1;
                }
                else if (CheckerBoard[i][j] == PLAYER2_KING)
                {
                    a2 -= 3;
                    c -= 1;
                }
            }
        a1 *= 10000000;
        a2 *= 10000000;
        b *= 100000;
        c *= 1000;
        int e = rand() % 100;
        if (player == PLAYER2)
            e = -e;
        return a1 + a2 + b + c + e;
    }
```

Evaluation Function - By Benila

Benila made the evaluation function giving an advantage to the position of the pieces on the board. She gave different weights to pieces based on their position. The overall evaluation function is a sum of the center position + edge position + my piece + my king

```
else if ((opt == 1 && evalp1 == 3) || (opt == 2 && evalp2 == 3))
    {
        //EVALUATION FUNCTION 3
        //Evaluation function 3 -  different weights are given to pieces
based on their poisition
        //center position + edge position + my piece + myking
        for (int i = 0; i < 8; i++)
        {
            for (int j = 0; j < 8; j++)
            {
                if(CheckerBoard[i][j] == PLAYER1)
                {
```

```
                    player1Count++;
                    //if piece in edge square
                    if( (i==3) && ((j==0) || (j==7)))
                        player1Count += 0.5;
                    //if piece in one of the four center squares
                    if (((i==3) && ((j == 2) || (j== 4))) || ((i==4) &&
((j==3)||(j==5))))
                        player1Count += 0.75;

                }else if(CheckerBoard[i][j] == PLAYER2){

                    player2Count++;
                    //if piece in edge square
                    if( (i==4) && ((j==0) || (j==7)))
                       player2Count += 0.5;
                    //if piece is in one of the four center squares
                    if (((i==3) && ((j == 2) || (j== 4))) || ((i==4) &&
((j==3)||(j==5))))
                       player2Count += 0.75;
                }
                if(CheckerBoard[i][j] == PLAYER1_KING){

                    player1Count += 2;
                    //if piece in edge square
                   if( (i==3) && ((j==0) || (j==7)))
                       player1Count += 0.5;
                    //if piece is in one of the four center squares
                    if (((i==3) && ((j == 2) || (j== 4))) || ((i==4) &&
((j==3)||(j==5))))
                        player1Count += 0.75;

                }else if (CheckerBoard[i][j] == PLAYER2_KING){

                    player2Count += 2;
                    //if piece in edge square
                   if((i==4) && ((j==0) || (j==7)))
                      player2Count += 0.5;
                    //if piece in one of the four center squares
                    if (((i==3) && ((j == 2) || (j== 4))) || ((i==4) &&
((j==3)||(j==5))))
                         player2Count += 0.75;
                }

            }
        }
    }
    if (opt == 1)
        return player1Count - player2Count;
    return player2Count - player1Count;
```

Analysis of Programs

Table 1: Computer 1 vs Computer 2 with Alpha Beta Search Algorithm – Depth 2

	Computer 1 Alpha Beta Search Evaluation Function 1										
Computer 2 Alpha Beta Search		Nodes Generated		Nodes Expanded		Time Taken		Memory Consumption		Game Length	Winner
	Comp	1	2	1	2	1	2	1	2		
	Eval 1	13302	19938	562	282	35803	44346	23604	11844	101	Tie
	Eval 2	5191	1759	104	183	13875	5106	4368	7686	51	Comp 1 X[1]
	Eval 3	11231	7526	243	228	24718	18073	10206	9576	60	Comp 2 O[2]

Table 2: Computer 1 vs Computer 2 with MinMax AB Algorithm – Depth 2

	Computer 1 MinMax AB Evaluation Function 1										
Computer 2 MinMax AB		Nodes Generated		Nodes Expanded		Time Taken		Memory Consumption		Game Length	Winner
	Comp	1	2	1	2	1	2	1	2		
	Eval 1	13012	21737	757	273	32223	48624	31794	11466	101	Tie
	Eval 2	5242	1607	130	208	20370	7110	5460	8736	101	Tie
	Eval 3	13860	12753	503	315	40453	32092	21126	13230	101	Tie

Table 3: Computer 1 (Alpha Beta Search) vs Computer 2 (MinMax AB) – Depth 2

	Computer 1 Alpha Beta Search Evaluation Function 1										
Computer 2 MinMax AB		Nodes Generated		Nodes Expanded		Time Taken		Memory Consumption		Game Length	Winner
	Comp	1	2	1	2	1	2	1	2		
	Eval 1	16255	23033	632	298	37391	51526	26544	12516	101	Tie
	Eval 2	5192	1558	105	183	17164	7849	4410	7686	52	O[2]
	Eval 3	15316	12009	446	332	35785	30014	18732	13944	101	Tie

Table 4: Computer 1 (MinMax AB) vs Computer 2 (Alpha Beta Search) – Depth 2

	Computer 1 MinMax AB Evaluation Function 1									
Computer 2 Alpha Beta Search	Nodes Generated		Nodes Expanded		Time Taken		Memory Consumption		Game Length	Winner
Comp	1	2	1	2	1	2	1	2		
Eval 1	11533	14900	502	241	35291	33142	21084	10122	101	Tie
Eval 2	5191	1759	104	183	18326	5934	4368	7686	51	X[1]
Eval 3	12952	12414	471	298	36066	28443	19782	12516	101	Tie

Table 5: Computer 1 vs Computer 2 with Alpha Beta Search– Depth 4

	Computer 1 Alpha Beta Search Evaluation Function 1									
Computer 2 Alpha Beta Search	Nodes Generated		Nodes Expanded		Time Taken		Memory Consumption		Game Length	Win Player
Comp	1	2	1	2	1	2	1	2		
Eval 1	306602	332268	249	478	603370	683807	10458	20076	101	Tie
Eval 2	228244	38087	173	251	492692	84777	7266	10542	55	X[1]
Eval 3	343590	323369	620	289	668801	30014	26040	12138	101	Tie

Table 6: Computer 1 vs Computer 2 with MinMax AB Algorithm – Depth 4

	Computer 1 MinMax AB Evaluation Function 1									
Computer 2 MinMax AB	Nodes Generated		Nodes Expanded		Time Taken		Memory Consumption		Game Length	Win
Comp	1	2	1	2	1	2	1	2		
Eval 1	82036	82896	242	174	209384	204682	10164	7308	70	O[2]
Eval 2	74664	22024	171	150	184187	58389	7182	6300	52	O[2]
Eval 3	74937	54731	179	149	182233	136192	7518	6258	54	O[2]

Table 7: Computer 1 (Alpha Beta Search) vs Computer 2 (MinMax AB) – Depth 4

	Computer 1 Alpha Beta Search Evaluation Function 1										
Computer 2 MinMax AB		**Nodes Generated**		**Nodes Expanded**		**Time Taken**		**Memory Consumption**		**Game Length**	**Winner**
	Comp	**1**	**2**	**1**	**2**	**1**	**2**	**1**	**2**		
	Eval 1	196067	57949	147	218	537533	171088	6174	9156	49	X[1]
	Eval 2	200936	28810	148	235	516670	85367	6216	9870	51	X[1]
	Eval 3	196067	51797	147	218	486760	133744	6174	9156	49	X[1]

Table 8: Computer 1 (MinMax AB) vs Computer 2 (Alpha Beta Search) – Depth 2

	Computer 1 MinMax AB Evaluation Function 1										
Computer 2 Alpha Beta Search		**Nodes Generated**		**Nodes Expanded**		**Time Taken**		**Memory Consumption**		**Game Length**	**Winner**
	Comp	**1**	**2**	**1**	**2**	**1**	**2**	**1**	**2**		
	Eval 1	77289	208723	306	127	191087	422327	12852	5334	48	O[2]
	Eval 2	166365	48218	241	214	463515	126557	10122	8988	66	O[2]
	Eval 3	62595	115547	218	95	158759	262857	9156	3990	54	O[2]

- With the analysis reports we can observe that with Evaluation Function 1 which considers possible moves and jumps there were 8 wins out of 24 combinations of runs.
- Evaluation Function 2 had 3 wins out of all combinations played with Evaluation Function 1 only.
- Evaluation Function 3 had 3 wins when played with Evaluation Function 1 only.
- The analysis shows that when Evaluation Function 1 considers possible moves and jumps with piece and king piece difference the Alpha Beta Search performs equivalent to MinMax AB algorithm. Hence there are either tie or wins.
- Player wins when the opponent player's moves are blocked, or all the pieces are captures.
- When a player has king pieces it can move forward and backward hence the player takes that advantage and ties the game.

Sample Output

At the terminal after running following commands:

1. g++ -std=c++17 *.cpp
2. ./a.out

A sample game player between Computer 1 – Alpha Beta Search with Evaluation Function 1

And Computer 2 – MinMax AB with Evaluation Function 2

```
 Welcome to Checkers Game
                 LET'S START!!
Enter your choice for Computer 1
1.Computer 1 - Alpha-Beta-Search
2.Computer 1 - MinMaxAB
Choose one of the above options: 1
Choose ANY ONE of the below evaluation function for computer 1

1.Eval 1
2.Eval 2
3.Eval 3
1
Enter your choice for Computer 2:
1.Computer 2 - Alpha-Beta-Search
2.Computer 2 - MinMaxAB
1
Choose ANY ONE of the below evaluation function for computer 2
1.Eval 1
2.Eval 2
3.Eval 3
2

Choose the depth of algorithms (2 or 4):
2
```

Some board displays to show how the players move:

```
=========================
After Computer O Turn board is:

=========================
-------------------
R/C 0 1 2 3 4 5 6 7
-------------------
 |0|_ _ _ _ _ _ _ _
 |1|_ _ X _ X _ _ _
 |2|_ X _ _ _ X _ X
 |3|O _ _ _ X _ X _
 |4|_ X _ X _ X _ X
 |5|X _ O _ O _ O _
 |6|_ O _ O _ O _ O
 |7|O _ O _ O _ O _

=========================
After Computer X Turn board is:
```

```
==========================
------------------
R/C 0 1 2 3 4 5 6 7
------------------
 |0|_ _ _ _ _ _ _ _
 |1|_ _ X _ X _ _ _
 |2|_ _ _ _ _ X _ X
 |3|O _ X _ X _ X _
 |4|_ X _ X _ X _ X
 |5|X _ O _ O _ O _
 |6|_ O _ O _ O _ O
 |7|O _ O _ O _ O _

==========================
After Computer O Turn board is:

==========================
------------------
R/C 0 1 2 3 4 5 6 7
------------------
 |0|_ _ _ _ _ _ _ _
 |1|_ _ X _ X _ _ _
 |2|_ _ _ _ _ X _ X
 |3|O _ X _ X _ X _
 |4|_ X _ X _ X _ X
 |5|X _ O _ O _ O _
 |6|_ O _ O _ O _ O
 |7|O _ O _ O _ O _

==========================
After Computer X Turn board is:

==========================
------------------
R/C 0 1 2 3 4 5 6 7
------------------
 |0|_ _ _ _ _ _ _ _
 |1|_ _ X _ _ _ _ _
 |2|_ _ _ X _ X _ X
 |3|O _ X _ X _ X _
 |4|_ X _ X _ X _ X
 |5|X _ O _ O _ O _
 |6|_ O _ O _ O _ O
 |7|O _ O _ O _ O _

==========================
After Computer O Turn board is:

==========================
------------------
R/C 0 1 2 3 4 5 6 7
------------------
 |0|_ _ _ _ _ _ _ _
 |1|_ _ X _ _ _ _ _
 |2|_ O _ X _ X _ X
```

```
 |3|_ _ X _ X _ X _
 |4|_ X _ X _ X _ X
 |5|X _ O _ O _ O _
 |6|_ O _ O _ O _ O
 |7|O _ O _ O _ O _

=========================
After Computer X Turn board is:

=========================
-------------------
R/C 0 1 2 3 4 5 6 7
-------------------
 |0|_ _ _ _ _ _ _ _
 |1|_ _ _ _ _ _ _ _
 |2|_ _ _ X _ X _ X
 |3|X _ X _ X _ X _
 |4|_ X _ X _ X _ X
 |5|X _ O _ O _ O _
 |6|_ O _ O _ O _ O
 |7|O _ O _ O _ O _

=========================
Length of Game Path:51
Total Number of Nodes Generated by Computer 1: 5191
Total Number of Nodes Generated by Computer 2: 1759
Total Number of Nodes Expanded by Computer 1: 104
Total Number of Nodes Expanded by Computer 2: 183
Time taken by Computer 1: 12580 microseconds
Time taken by Computer 2: 4882 microseconds
Computer 1's Memory Consumption was: 4368bytes
Computer 2's Memory Consumption was: 7686bytes
Computer 1's Evaluation Function is: 1
Computer 2's Evaluation Function is: 2
Computer X[1] Won the game
```

Player X[1] i.e. the computer 1 has blocked all the moves of computer 2 O[2] hence won the game.

Source Code Implementation

The code contains files main.cpp, CheckerBoard.cpp, Algorithm.cpp, Moves.cpp, EvaluationFunction.cpp, and header files CheckerBoard.h, Algorithm.h

Main.cpp

This file contains the function of computer vs computer game and the driver code which is the user interface.

```
 #include <iostream>
#include <vector>
#include <algorithm>
#include <chrono>
#include "CheckerBoard.h"
#include "Algorithm.h"

using namespace std;
using namespace std::chrono;
int evalp1;
int evalp2;
int comp1,comp2;
int depth;

int opt;
int TotalElements = 0;
int TotalElementsp1 = 0; // Total Elements Expanded by Player 1
int TotalElementsp2 = 0;  // Total Elements Expanded by Player 2
int TotalElementsGenp1; // Total Elements Generated by Player 1
int TotalElementsGenp2;  // Total Elements Generated by Player 2
int ElementSize = 42;
int TotalMemp1; // Total Memory used by Player 1
int TotalMemp2;  // Total Memory used by Player 2
double Player1Time = 0; // time taken by Player 1
double Player2Time = 0; // time taken by Player 2
int moveCount = 0;
vector<Element> bestPath;
void printEval();

/*---- Function:PrintBoard()-----*
Purpose: Prints the board
Input: board[][]
Return Type: void */

void printBoard(char boardsetup[][8])
{
    cout << endl << "=========================" << endl;
    for (int i = 0; i < 8; i++)
    {
        if (i == 0)
        {
            cout << "--------------------" << endl;
            cout << "R/C 0 1 2 3 4 5 6 7" << endl;
            cout << "--------------------";
            cout << endl;
        }
```

```
        cout << " |" << i << "|";
        for (int j = 0; j < 8; j++)
        {
            cout << boardsetup[i][j] << " ";
        }
        cout << endl;
    }
    cout << endl << "========================" << endl;
}


//******  COMPUTER VS COMPUTER ******//

/* Game Functions*/
/*---- Function:Computer Vs Computer -----*
//Purpose: Game playing function Computer Vs Computer using MinMaxAB or
Alpha Beta Search
//Input: board[][],row,col,flag,compare row,compare col,compare
board[][],compare flag
//Return Type: void */
void comp1vscomp2(int &row, int &column, int &nrow, int &ncolumn, bool
&flag, bool &flag2, char CheckerBoard[][8])
{
    while (flag2)
    {
        opt = 1;
        Element test;
        test.setBoard(CheckerBoard);
        test.setPlayer(PLAYER1);
        test.move = 0;
        auto start = high_resolution_clock::now();
        if(comp1 == 1)
        alpha_beta_search(test, CheckerBoard);
        else if (comp1 == 2)
        Element c1 = MinMaxAB(test, test.move, test.player, 120, -100);
        auto stop = high_resolution_clock::now();
        auto duration = duration_cast<microseconds>(stop - start);
        Player1Time += duration.count();
        if (comp1 == 2)
        {
        for (int k = bestPath.size() - 1; k >= 0; k--)
        {
            if (bestPath[k].move == 1)
            {
                for (int i = 0; i < 8; i++)
                {
                    for (int j = 0; j < 8; j++)
                    {
                        //find child with move 0 and then store it into
board
                        CheckerBoard[i][j] =
bestPath[k].CheckerBoard[i][j];
                    }
                }
                break;
            }
```

```
        }
        }
        cout << "After Computer X Turn board is:" << endl;
        moveCount++;
        printBoard(CheckerBoard);
        Element test1;
        test1.setBoard(CheckerBoard);
        test1.setPlayer(PLAYER2);
        test1.move = 0;
        test1.moves();
        //Subtract the size of moves() from TotalElements
        TotalElements -= test1.childs.size();
        TotalElementsp1 += TotalElements;
        TotalElementsGenp1 += test1.childs.size();
        TotalMemp1 = ElementSize * TotalElementsGenp1;
        TotalElements = 0;
        if (test1.terminal())
        {
            cout << "Length of Game Path:" << moveCount << endl;
            cout << "Total Number of Nodes Generated by Computer 1: " <<
TotalElementsp1 << endl;
            cout << "Total Number of Nodes Generated by Computer 2: " <<
TotalElementsp2 << endl;
            cout << "Total Number of Nodes Expanded by Computer 1: " <<
TotalElementsGenp1 << endl;
            cout << "Total Number of Nodes Expanded by Computer 2: " <<
TotalElementsGenp2 << endl;
            cout << "Time taken by Computer 1: " << Player1Time << "
microseconds" << endl;
            cout << "Time taken by Computer 2: " << Player2Time << "
microseconds" << endl;
            cout << "Computer 1's Memory Consumption was: " << TotalMemp1
<< "bytes" <<endl;
            cout << "Computer 2's Memory Consumption was: " << TotalMemp2
<< "bytes" <<endl;
            printEval();
            int w = test1.evaluate();
            if (w == 100)
            {
                cout << "Computer X[1] Won the game" << endl;
                exit(0);
            }
        }
        if (moveCount >= 101)
        {
            cout << "Oops!! Game is  Tie" << endl;
            cout << "Length of Game Path:" << moveCount << endl;
            cout << "Total Number of Nodes Generated by Computer 1: " <<
TotalElementsp1 << endl;
            cout << "Total Number of Nodes Generated by Computer 2: " <<
TotalElementsp2 << endl;
            cout << "Total Number of Nodes Expanded by Computer 1: " <<
TotalElementsGenp1 << endl;
            cout << "Total Number of Nodes Expanded by Computer 2: " <<
TotalElementsGenp2 << endl;
```

```
            cout << "Time taken by Computer 1: " << Player1Time << "
microseconds" << endl;
            cout << "Time taken by Computer 2: " << Player2Time << "
microseconds" << endl;
            cout << "Computer 1's Memory Consumption was: " << TotalMemp1
<< endl;
            cout << "Computer 2's Memory Consumption was: " << TotalMemp2
<< endl;
            printEval();
            exit(0);
        }

        if(comp2 == 2)
        {bestPath.clear();}
        opt = 2;
        Element test2;
        test2.setBoard(CheckerBoard);
        test2.setPlayer(PLAYER2);
        test2.move = 0;
        start = high_resolution_clock::now();
        if(comp2 == 1)
        alpha_beta_search(test2, CheckerBoard);
        else if(comp2 == 2)
        Element c2 = MinMaxAB(test2, test2.move, test2.player, 120, -100);
        stop = high_resolution_clock::now();
        duration = duration_cast<microseconds>(stop - start);
        Player2Time += duration.count();
        if(comp2 == 2)
        {
        for (int k = bestPath.size() - 1; k >= 0; k--)
        {
            if (bestPath[k].move == 1)
            {
                for (int i = 0; i < 8; i++)
                {
                    for (int j = 0; j < 8; j++)
                    {
                        //find child with move 0 and then store it into
board
                        CheckerBoard[i][j] =
bestPath[k].CheckerBoard[i][j];
                    }
                }
                break;
            }
        }
        }
        cout << "After Computer O Turn board is:" << endl;
        moveCount++;
        printBoard(CheckerBoard);
        Element test3;
        test3.setBoard(CheckerBoard);
        test3.setPlayer(PLAYER1);
        test3.move = 0;
        test3.moves();
        //Subtract the size of moves() from TotalElements
```

```
        TotalElements -= test3.childs.size();
        TotalElementsp2 += TotalElements;
        TotalElementsGenp2 += test3.childs.size();
        TotalMemp2 = ElementSize * TotalElementsGenp2;
        TotalElements = 0;
        if (test3.terminal())
        {
            cout << "Length of Game Path:" << moveCount << endl;
            cout << "Total Number of Nodes Generated by Computer 1: " <<
TotalElementsp1 << endl;
            cout << "Total Number of Nodes Generated by Computer 2: " <<
TotalElementsp2 << endl;
            cout << "Total Number of Nodes Expanded by Computer 1: " <<
TotalElementsGenp1 << endl;
            cout << "Total Number of Nodes Expanded by Computer 2: " <<
TotalElementsGenp2 << endl;
            cout << "Time taken by Computer 1: " << Player1Time << "
microseconds" << endl;
            cout << "Time taken by Computer 2: " << Player2Time << "
microseconds" << endl;
            cout << "Computer 1's Memory Consumption was: " << TotalMemp1
<< endl;
            cout << "Computer 2's Memory Consumption was: " << TotalMemp2
<< endl;
            printEval();
            int w = test3.evaluate();
            if (w == 100)
            {
                cout << "Computer O[2] Won the game" << endl;
                exit(0);
            }
        }
        if (moveCount >= 101)
        {
            cout << "Oops!! Game is  Tie" << endl;
            cout << "Length of Game Path:" << moveCount << endl;
            cout << "Total Number of Nodes Generated by Computer 1: " <<
TotalElementsp1 << endl;
            cout << "Total Number of Nodes Generated by Computer 2: " <<
TotalElementsp2 << endl;
            cout << "Total Number of Nodes Expanded by Computer 1: " <<
TotalElementsGenp1 << endl;
            cout << "Total Number of Nodes Expanded by Computer 2: " <<
TotalElementsGenp2 << endl;
            cout << "Time taken by Computer 1: " << Player1Time << "
microseconds" << endl;
            cout << "Time taken by Computer 2: " << Player2Time << "
microseconds" << endl;
            cout << "Computer 1's Memory Consumption was: " << TotalMemp1
<< endl;
            cout << "Computer 2's Memory Consumption was: " << TotalMemp2
<< endl;
            printEval();
            exit(0);
        }
    }
```

```
}

//****** END OF SECTION 5: COMPUTER VS COMPUTER  ******//

void printEval()
{
        cout << "Computer 1's Evaluation Function is: " << evalp1 << endl;
        cout << "Computer 2's Evaluation Function is: " << evalp2 << endl;
}


//****** START OF SECTION 6: MAIN DRIVER FUNCTION ******//

int main()
{
    int row, column, nrow, ncolumn;
    bool flag = false;
    bool flag2 = true;

    char CheckerBoard[8][8] =
    {
        {EMPTY_SPACE, PLAYER1, EMPTY_SPACE, PLAYER1, EMPTY_SPACE, PLAYER1,
EMPTY_SPACE, PLAYER1},
        {PLAYER1, EMPTY_SPACE, PLAYER1, EMPTY_SPACE, PLAYER1, EMPTY_SPACE,
PLAYER1, EMPTY_SPACE},
        {EMPTY_SPACE, PLAYER1, EMPTY_SPACE, PLAYER1, EMPTY_SPACE, PLAYER1,
EMPTY_SPACE, PLAYER1},
        {EMPTY_SPACE, EMPTY_SPACE, EMPTY_SPACE, EMPTY_SPACE, EMPTY_SPACE,
EMPTY_SPACE, EMPTY_SPACE, EMPTY_SPACE},
        {EMPTY_SPACE, EMPTY_SPACE, EMPTY_SPACE, EMPTY_SPACE, EMPTY_SPACE,
EMPTY_SPACE, EMPTY_SPACE, EMPTY_SPACE},
        {PLAYER2, EMPTY_SPACE, PLAYER2, EMPTY_SPACE, PLAYER2, EMPTY_SPACE,
PLAYER2, EMPTY_SPACE},
        {EMPTY_SPACE, PLAYER2, EMPTY_SPACE, PLAYER2, EMPTY_SPACE, PLAYER2,
EMPTY_SPACE, PLAYER2},
        {PLAYER2, EMPTY_SPACE, PLAYER2, EMPTY_SPACE, PLAYER2, EMPTY_SPACE,
PLAYER2, EMPTY_SPACE},
    };

    cout << "\t\t Welcome to Checkers Game" << endl;
    cout << "\t\t LET'S START!!" << endl;
    cout<<"Enter your choice for Computer 1"<<endl;
    cout<<"1.Computer 1 - Alpha-Beta-Search"<<endl;
    cout<<"2.Computer 1 - MinMaxAB"<<endl;
    cout << "Choose one of the above options: ";
    do
    {
        cin >> comp1;
    } while (!(comp1 >= 1 && comp2 <= 2));

    cout << "Choose ANY ONE of the below evaluation function for computer
1" << endl << endl;
    cout << "1.Eval 1" << endl;
    cout << "2.Eval 2" << endl;
    cout << "3.Eval 3" << endl;
    do
```

```
    {
        cin >> evalp1;
    } while (!(evalp1 >= 1 && evalp1 <= 3));

    cout<<"Enter your choice for Computer 2:"<<endl;
    cout<<"1.Computer 2 - Alpha-Beta-Search"<<endl;
    cout<<"2.Computer 2 - MinMaxAB"<<endl;

    do
    {
        cin >> comp2;
    } while (!(comp2 >= 1 && comp2 <= 2));

    cout << "Choose ANY ONE of the below evaluation function for computer
2" << endl;
    cout << "1.Eval 1" << endl;
    cout << "2.Eval 2" << endl;
    cout << "3.Eval 3" << endl;
    do
    {
        cin >> evalp2;
    } while (!(evalp2 >= 1 && evalp2 <= 3));

    cout<<"\nChoose the depth of algorithms (2 or 4): "<<endl;
    cin >> depth;

    comp1vscomp2(row, column, nrow, ncolumn, flag, flag2, CheckerBoard);

}

//****** END MAIN DRIVER FUNCTION ******//
//**************** END OF CHECKERS GAME ***************//
```

Algorithm.h

```
#ifndef Algorithm_h
#define Algorithm_h

//****** Functions for the Algorithms ******//

extern vector<Element> bestPath;

// Minmax AB Declaration
Element MinMaxAB(Element& e, int depth, char player, int useThresh, int
passThresh);

//Alpha Beta Search Declaration.
void alpha_beta_search(Element state, char[][8]);

//Alpha Beta Search Max Function Declaration
int max_value(Element& state, int alpha, int beta);

//Alpha Beta Search Min Function Declaration
int min_value(Element& state, int alpha, int beta);

extern int depth;
```

```
#endif /* Algorithm_h */
```

Algorithm.cpp

```
#include <stdio.h>
#include "CheckerBoard.h"
#include "Algorithm.h"
//****** Functions for the Algorithms ******//

/*
Alpha-Beta-Search
Implemented From Text Book Chater 5 Adversarial searches
*/
/*---- Function:terminal()-----*
Purpose: If there are no moves then returns true else false
Return Type: bool */

bool Element::terminal() const
{

    if (childs.empty())
        return true;

    if(move >= depth)
        return true;

        return false;
}

void alpha_beta_search(Element state, char array2D[][8])
{
    state.v = max_value(state, -1000, 1000);
    for (int i = 0; i < state.childs.size(); i++)
    {
        if (state.childs[i].v == state.v)
        {
            for (int h = 0; h < 8; h++)
            {
                for (int w = 0; w < 8; w++)
                {
                    array2D[h][w] = state.childs[i].CheckerBoard[h][w];
                }
            }
            return;
        }
    }
}

int max_value(Element& state, int alpha, int beta)
{
    state.moves();
    if (state.terminal())
    {
        state.v = state.evaluate();
        return state.evaluate();
```

```
    }
    state.v = -100;
    for (int i = 0; i < state.childs.size(); i++)
    {
        state.v = max(state.v, min_value(state.childs[i], alpha, beta));
        if (state.v >= beta)
        {
            return state.v;
        }
        alpha = max(alpha, state.v);
    }
    return state.v;
}

int min_value(Element& state, int alpha, int beta)
{
    state.moves();
    if (state.terminal())
    {
        state.v = state.evaluate();
        return state.evaluate();
    }
    state.v = 50;
    for (int i = 0; i < state.childs.size(); i++)
    {
        state.v = min(state.v, max_value(state.childs[i], alpha, beta));
        if (state.v <= alpha)
        {
            return state.v;
        }
        beta = min(beta, state.v);
    }
    return state.v;
}


/*

MinMax AB Algorithm
Implemented from Rich and Knight Chapther 12

*/

bool DeepEnough(int d)
{
    if (depth == 2)
        return d >= 2;
    else
    return d >= 4;
}

//*************************************************************************
//minimaxAB: function for the implementation of rich and knight algorithm
//             gets the board position, current depth, player name,
usethresh
```

```
//          passthresh as parameters and returns the bestvalue and the
bestpath
//          for next move
//p:    current node being examined
//depth: level at which the node is examined
//player: name of the player at this level
//usethresh: lower bound value that a player may be assigned
//passthresh: upperbound value that a opponent may be assigned
//*************************************************************************
Element MinMaxAB(Element &e, int depth,char player,int useThresh,int
passThresh)
{
    //vector<Element>path;
    int newValue = 0;

    //checks if the depth has been reached
    if(DeepEnough(depth)){

        e.moves();
        e.v = e.evaluate();
        if (opt == 1)
            if (e.player == PLAYER2)
                e.v *= -1;
        if (opt == 2)
            if (e.player == PLAYER1)
                e.v *= -1;
        return e;
    }

    e.moves();
    vector<Element> successors = e.childs;

    //checks if there is any move for the board position
    if (successors.empty()) {

        e.evaluate();
        if (opt == 1)
        {
            //PLAYER1 is Max
            //PLAYER2 is Min
            if (e.player == PLAYER2)
                e.v *= -1;
        }
        if (opt == 2)
        {
            //PLAYER2 is Max
            //PLAYER1 is Min
            if (e.player == PLAYER1)
                e.v *= -1;
        }

        return e;
    }

    //loop through sucessors
    for(auto & successor : successors){
```

```
        Element succ = successor;

        Element resultSucc;

        resultSucc = MinMaxAB(succ,succ.move,succ.player,-1*passThresh,-
1*useThresh);
        newValue = -1*(resultSucc.v);

        //if newvalue is greater than cut off no further children will be
explored
        if(newValue > passThresh){

            passThresh = newValue;
            if (depth  == 0)
                bestPath.push_back(succ);
        }
        //condition to cut off search
        if(passThresh >= useThresh){
            succ.v = passThresh;
            e = successor;
            return e;
        }

    }
    e.v = passThresh;
    return e;
}

//****** Functions for the Algorithms ******//
```

CheckerBoard.h

```
#ifndef CheckerBoard_h
#define CheckerBoard_h
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

// Defining the Player Variables
#define EMPTY_SPACE  '_'
#define PLAYER1  'X'
#define PLAYER2  'O'
#define PLAYER1_KING 'K' // Player King 1
#define PLAYER2_KING 'L' // Player King 2

extern int opt;
extern int evalp1;
extern int evalp2;
extern int TotalElements;

class Element
{
```

```
public:

    char CheckerBoard[8][8]{}, player{};
    vector<Element>childs;
    int v{};
    int move{};

    //****** SECTION 1 : Functions for the EVALUATION FUNCTION ******//
    int  evaluate();
    int EvalValue();
   // int DefendingNeighbors(char state[][8], int row, int col);
   // void printEval();
    //*************************************//

    //****** SECTION 2: Functions for the BOARD SETUP,PLAYER AND DISPLAY
******//
    void setBoard(char b[][8]);
    void setPlayer(char);
    //*************************************//

    //****** SECTION 3: Functions for the CHECKERS GAME ******//
    bool terminal() const;
    void action(char newboard[][8]);
    void moves();
    static bool compare(char[][8], char[][8]);
    static bool ValidPosition(char CheckerBoard[][8], int row, int col);

    static bool leftCorner(char CheckerBoard[][8], int row, int col, char
player);
    static bool rightCorner(char CheckerBoard[][8], int row, int col, char
player);
    static bool Leftcheck(char CheckerBoard[][8], int row, int col, char
player);
    static bool Rightcheck(char CheckerBoard[][8], int row, int col, char
player);

    void jump(char[][8], int, int, char);
    void leftJump(char newboard[][8], int row, int col, char player, int&,
int&);
    void rightJump(char newboard[][8], int row, int col, char player, int&,
int&);


    static bool checkKing(int row, char player);
    static bool kingLeftCorner(char newboard[][8], int row, int col, char
player);
    static bool kingRightCorner(char newboard[][8], int row, int col, char
player);
    static bool kingTopLeftCorner(char newboard[][8], int row, int col,
char player);
    static bool kingTopRightCorner(char newboard[][8], int row, int col,
char player);

    void jumpKing(char[][8], int, int, char);
    void kingLeftTop(char newboard[][8], int row, int col, char player,
int& crow, int& ccol);
```

```
    void kingRightTop(char newboard[][8], int row, int col, char player,
int& crow, int& ccol);
    void kingLeftJump(char newboard[][8], int row, int col, char player,
int& crow, int& ccol);
    void kingRightJump(char newboard[][8], int row, int col, char player,
int& crow, int& ccol);
    //*************************************//
};

#endif /* CheckerBoard_h */
```

CheckerBoard.cpp

```
#include <stdio.h>
#include <iostream>
#include "CheckerBoard.h"

using namespace std;
//******  Functions for the BOARD SETUP,PLAYER AND DISPLAY ******//

/*---- Function:SetBoard()-----*
Purpose: Set the board with respective names for each player.
Input:board[][]
Return Type: void */

void Element::setBoard(char b[][8])
{
    for (int i = 0; i < 8; i++)
    {
        for (int j = 0; j < 8; j++)
        {
            CheckerBoard[i][j] = b[i][j];
        }
    }
}

/*---- Function:SetPlayer()-----*
Purpose: Set the player as p
Input: char p
Return Type: void */

void Element::setPlayer(char p)
{
    player = p;
}

//****** Functions for the BOARD SETUP,PLAYER AND DISPLAY ******//

//****** Functions for the CHECKERS GAME ******//

/*---- Function:compare()-----*
Purpose: Used for comparing two boards.
Input: board1[][],board2[][]
```

```
Return Type: bool - if the boards are not equal then returns false else
true */

bool Element::compare(char s[][8], char d[][8])
{
    for (int i = 0; i < 8; i++)
    {
        for (int j = 0; j < 8; j++)
        {
            if (s[i][j] != d[i][j])
                return false;
        }
    }
    return true;
}

/*---- Function:compare()-----*
Purpose: Position of the piece should be always within limits and should be
empty
         where you are about to make a move
Input: checkerboard[][],row,col
Return Type: bool - returns true if purpose is satified else false */

bool Element::ValidPosition(char CheckerBoard[][8], int row, int col)
{
    if (!(row >= 0 && row <= 7 && col >= 0 && col <= 7 &&
CheckerBoard[row][col] == EMPTY_SPACE))
        return false;
    return true;
}


/*---- Function:leftCorner()-----*
Purpose: To check whether left corner jump is possible or not.
Input: checkerboard[][],row,col,player
Return Type: bool - Returns true if you cannot make a left jump from the
current
                    position (row,col) else false */

bool Element::leftCorner(char CheckerBoard[][8], int row, int col, char
player)

{
    int mrow, mcol;
    if (player == PLAYER1)
    {
        mrow = row + 2;
        mcol = col - 2;
        if (ValidPosition(CheckerBoard, mrow, mcol))

        {
            if (CheckerBoard[mrow - 1][mcol + 1] == PLAYER2 ||
CheckerBoard[mrow - 1][mcol + 1] == PLAYER2_KING)
            {
                return false;
            }
```

```
        }
    }
    else if (player == PLAYER2)
    {
        mrow = row - 2;
        mcol = col - 2;
        if (ValidPosition(CheckerBoard, mrow, mcol))

        {
            if (CheckerBoard[mrow + 1][mcol + 1] == PLAYER1 ||
CheckerBoard[mrow + 1][mcol + 1] == PLAYER1_KING)
            {
               return false;
            }
        }
    }
    return true;
}

/*---- Function:rightCorner()-----*
Purpose: To check whether right corner jump is possible or not.
Input: checkerboard[][],row,col,player
Return Type: bool - Returns true if you cannot make a right jump from the
current
                    position (row,col) else false */

bool Element::rightCorner(char CheckerBoard[][8], int row, int col, char
player)

{
    int mrow, mcol;
    if (player == PLAYER1)
    {
        mrow = row + 2;
        mcol = col + 2;
        if (ValidPosition(CheckerBoard, mrow, mcol))

        {
            if (CheckerBoard[mrow - 1][mcol - 1] == PLAYER2 ||
CheckerBoard[mrow - 1][mcol - 1] == PLAYER2_KING)
            {

                return false;
            }
        }
    }
    else if (player == PLAYER2)
    {
        mrow = row - 2;
        mcol = col + 2;
        if (ValidPosition(CheckerBoard, mrow, mcol))

        {
            if (CheckerBoard[mrow + 1][mcol - 1] == PLAYER1 ||
CheckerBoard[mrow + 1][mcol - 1] == PLAYER1_KING)
            {
```

```
                return false;
            }
        }
    }
    return true;
}

/*---- Function:Leftcheck()-----*
Purpose: To check whether leftmove is possible or not.
Input: checkerboard[][],row,col,player
Return Type: bool - Returns true if you can move left from the current
position
                    (row,col) else false */


bool Element::Leftcheck(char CheckerBoard[][8], int row, int col, char
player)

{
    int mrow=0, mcol = 0;
    if (player == PLAYER1)
    {
        mrow = row + 1;
        mcol = col - 1;
    }
    else if (player == PLAYER2)
    {
        mrow = row - 1;
        mcol = col - 1;
    }
    return ValidPosition(CheckerBoard, mrow, mcol);
}

/*---- Function:Rightcheck()-----*
Purpose: To check whether rightmove is possible or not.
Input: checkerboard[][],row,col,player
Return Type: bool - Returns true if you can move right from the current
position
                    (row,col) else false */

bool Element::Rightcheck(char CheckerBoard[][8], int row, int col, char
player)
{

    int mrow = 0, mcol = 0;
    if (player == PLAYER1)
    {
        mrow = row + 1;
        mcol = col + 1;
    }
    else if (player == PLAYER2)
    {
        mrow = row - 1;
        mcol = col + 1;
    }
```

```
    return ValidPosition(CheckerBoard, mrow, mcol);
}

/*---- Function:jump()-----*
Purpose: To perform leftjump or rightjump by checking left and right corner
         is performed by the piece.
Input: checkerboard[][],row,col,player
Return Type: void */


void Element::jump(char CheckerBoard[][8], int row, int col, char player)
{

    int mrow, mcol;
    mrow = row;
    mcol = col;
    char newboard[8][8];
    //create a copy of board
    copy(&CheckerBoard[0][0], &CheckerBoard[0][0] + 8 * 8,
&newboard[0][0]);
    leftJump(newboard, row, col, player, mrow, mcol);
    /*
    Check whether the newboard is changed --- compare newboard with
checkerboard based on left and right corner
    If changed then perform action
    */
    if (!compare(CheckerBoard, newboard) && leftCorner(newboard, mrow,
mcol, player) && rightCorner(newboard, mrow, mcol, player))

    {
        //Perform Action
        action(newboard);
    }

    // copy back board in newboard and call jumpright else call jumpight
    copy(&CheckerBoard[0][0], &CheckerBoard[0][0] + 8 * 8,
&newboard[0][0]);
    rightJump(newboard, row, col, player, mrow, mcol);
    /*
     Check whether the newboard is changed --- compare newboard with
checkerboard based on left and right corner
     If changed then perform action
     */
    if (!compare(CheckerBoard, newboard) && leftCorner(newboard, mrow,
mcol, player) && rightCorner(newboard, mrow, mcol, player))

    {
        //Perform Action
        action(newboard);
    }
    copy(&CheckerBoard[0][0], &CheckerBoard[0][0] + 8 * 8,
&newboard[0][0]);
}

/*---- Function:leftJump()-----*
```

```
Purpose: To perform leftjump and check if the new position will make the
current player a king or not
Input: checkerboard[][],row,col,player,reference to row,col
Return Type: void */

void Element::leftJump(char newboard[][8], int row, int col, char player,
int& crow, int& ccol)
{
    int mrow, mcol;
    if (player == PLAYER1)
    {
        mrow = row + 2;
        mcol = col - 2;
        if (ValidPosition(newboard, mrow, mcol))

        {
            if (newboard[mrow - 1][mcol + 1] == PLAYER2 || newboard[mrow -
1][mcol + 1] == PLAYER2_KING)
            {
                //Can make a left jump
                newboard[mrow - 1][mcol + 1] = EMPTY_SPACE;
                newboard[row][col] = EMPTY_SPACE;
                //Checking if the new position will make the current player
a king or not
                crow = mrow;
                ccol = mcol;
                if (checkKing(mrow, player))
                    newboard[mrow][mcol] = PLAYER1_KING;
                else
                {
                    newboard[mrow][mcol] = PLAYER1;
                    jump(newboard, mrow, mcol, player);
                }
            }
        }
    }
    else if (player == PLAYER2)
    {
        mrow = row - 2;
        mcol = col - 2;
        if (ValidPosition(newboard, mrow, mcol))

        {
            if (newboard[mrow + 1][mcol + 1] == PLAYER1 || newboard[mrow +
1][mcol + 1] == PLAYER1_KING)
            {
                //Can make a left jump
                newboard[mrow + 1][mcol + 1] = EMPTY_SPACE;
                newboard[row][col] = EMPTY_SPACE;
                //Checking if the new position will make the current player
a king or not
                crow = mrow;
                ccol = mcol;
                if (checkKing(mrow, player))
                    newboard[mrow][mcol] = PLAYER2_KING;
                else
```

```
                {
                    newboard[mrow][mcol] = PLAYER2;
                    jump(newboard, mrow, mcol, player);
                }
            }
        }
    }
}

/*---- Function:rightJump()-----*
Purpose: To perform rightJump and check if the new position will make the
current player a king or not
Input: checkerboard[][],row,col,player,reference to row,col
Return Type: void */

void Element::rightJump(char newboard[][8], int row, int col, char player,
int& crow, int& ccol)
{
    int mrow, mcol;
    if (player == PLAYER1)
    {
        mrow = row + 2;
        mcol = col + 2;
        if (ValidPosition(newboard, mrow, mcol))

        {
            if (newboard[mrow - 1][mcol - 1] == PLAYER2 || newboard[mrow -
1][mcol - 1] == PLAYER2_KING)
            {
                // can make a left jump
                newboard[mrow - 1][mcol - 1] = EMPTY_SPACE;
                newboard[row][col] = EMPTY_SPACE;
                crow = mrow;
                ccol = mcol;
                //Checking if the new position will make the current player
a king or not
                if (checkKing(mrow, player))
                    newboard[mrow][mcol] = PLAYER1_KING;
                else
                {
                    newboard[mrow][mcol] = PLAYER1;
                    jump(newboard, mrow, mcol, player);
                }
            }
        }
    }
    else if (player == PLAYER2)
    {
        mrow = row - 2;
        mcol = col + 2;
        if (ValidPosition(newboard, mrow, mcol))

        {
            if (newboard[mrow + 1][mcol - 1] == PLAYER1 || newboard[mrow +
1][mcol - 1] == PLAYER1_KING)
            {
```

```
                //can make a left jump
                newboard[mrow + 1][mcol - 1] = EMPTY_SPACE;
                newboard[row][col] = EMPTY_SPACE;
                crow = mrow;
                ccol = mcol;
                //Checking if the new position will make the current player
a king or not
                if (checkKing(mrow, player))
                    newboard[mrow][mcol] = PLAYER2_KING;
                else
                {
                    newboard[mrow][mcol] = PLAYER2;
                    jump(newboard, mrow, mcol, player);
                }
            }
        }
    }
}


/*---- Function:checkKing()-----*
Purpose: To check whether that piece is a king depending upon where it is
on the board.
Input: row,player
Return Type: bool -- return true if it is king else false */


bool Element::checkKing(int row, char player)
{
    if (player == PLAYER1 && row == 7)
    {
        return true;
    }
    if (player == PLAYER2 && row == 0)
    {
        return true;
    }
    return false;
}

/*---- Function:kingLeftCorner()-----*
Purpose: To check for the king position if it falls in left corner.
Input: board[][],row,col,player
Return Type: bool -- return true if the proper move is made else false */

bool Element::kingLeftCorner(char newboard[][8], int row, int col, char
player)
{
    int mrow, mcol;
    if (player == PLAYER1)
    {
        mrow = row + 2;
        mcol = col - 2;
        if (ValidPosition(newboard, mrow, mcol))

        {
```

```
            if (newboard[mrow - 1][mcol + 1] == PLAYER2 || newboard[mrow -
1][mcol + 1] == PLAYER2_KING)
            {
                return false;
            }
        }
    }
    else if (player == PLAYER2)
    {
        mrow = row + 2;
        mcol = col - 2;
        if (ValidPosition(newboard, mrow, mcol))

        {
            if (newboard[mrow - 1][mcol + 1] == PLAYER1 || newboard[mrow -
1][mcol + 1] == PLAYER1_KING)
            {
                return false;
            }
        }
    }
    return true;
}

/*---- Function:kingLeftCorner()-----*
Purpose: To check for the king position if it falls in right corner.
Input: board[][],row,col,player
Return Type: bool -- return true if the proper move is made else false */


bool Element::kingRightCorner(char newboard[][8], int row, int col, char
player)
{
    int mrow, mcol;
    if (player == PLAYER1)
    {
        mrow = row + 2;
        mcol = col + 2;
        if (ValidPosition(newboard, mrow, mcol))

        {
            if (newboard[mrow - 1][mcol - 1] == PLAYER2 || newboard[mrow -
1][mcol - 1] == PLAYER2_KING)
            {
                return false;
            }
        }
    }
    else if (player == PLAYER2)
    {
        mrow = row + 2;
        mcol = col + 2;
        if (ValidPosition(newboard, mrow, mcol))

        {
```

```
            if (newboard[mrow - 1][mcol - 1] == PLAYER1 || newboard[mrow -
1][mcol - 1] == PLAYER1_KING)
            {
                return false;
            }
        }
    }
    return true;
}

/*---- Function:kingTopLeftCorner()-----*
Purpose: To check for the king position if it falls in top left corner.
Input: board[][],row,col,player
Return Type: bool -- return true if the proper move is made else false */

bool Element::kingTopLeftCorner(char newboard[][8], int row, int col, char
player)
{
    int mrow, mcol;
    if (player == PLAYER1)
    {
        mrow = row - 2;
        mcol = col - 2;
        if (ValidPosition(newboard, mcol, mcol))

        {
            if (newboard[mrow + 1][mcol + 1] == PLAYER2 || newboard[mrow +
1][mcol + 1] == PLAYER2_KING)
            {
                return false;
            }
        }
    }
    else if (player == PLAYER2)
    {
        mrow = row - 2;
        mcol = col - 2;
        if (ValidPosition(newboard, mrow, mcol))

        {
            if (newboard[mrow + 1][mcol + 1] == PLAYER1 || newboard[mrow +
1][mcol + 1] == PLAYER1_KING)
            {
                return false;
            }
        }
    }
    return true;
}

/*---- Function:kingTopRightCorner()-----*
Purpose: To check for the king position if it falls in top right corner.
Input: board[][],row,col,player
Return Type: bool -- return true if the proper move is made else false */
```

```
bool Element::kingTopRightCorner(char newboard[][8], int row, int col, char
player)
{
    int mrow, mcol;
    if (player == PLAYER1)
    {
        mrow = row - 2;
        mcol = col + 2;
        if (ValidPosition(newboard, mrow, mcol))

        {
            if (newboard[mrow + 1][mcol - 1] == PLAYER2 || newboard[mrow +
1][mcol - 1] == PLAYER2_KING)
            {
                return false;
            }
        }
    }
    else if (player == PLAYER2)
    {
        mrow = row - 2;
        mcol = col + 2;
        if (ValidPosition(newboard, mrow, mcol))

        {
            if (newboard[mrow + 1][mcol - 1] == PLAYER1 || newboard[mrow +
1][mcol - 1] == PLAYER1_KING)
            {
                return false;
            }
        }
    }
    return true;
}


/*---- Function:kingLeftTop()-----*
Purpose: To check for the king position if it falls in left top.
Input: board[][],row,col,player,refernce to row,col
Return Type:void*/


void Element::kingLeftTop(char newboard[][8], int row, int col, char
player, int& crow, int& ccol)
{
    int mrow, mcol;
    if (player == PLAYER1)
    {
        mrow = row - 2;
        mcol = col - 2;
        if (ValidPosition(newboard, mrow, mcol))

        {
            if (newboard[mrow + 1][mcol + 1] == PLAYER2 || newboard[mrow +
1][mcol + 1] == PLAYER2_KING)
            {
```

```
                newboard[mrow + 1][mcol + 1] = EMPTY_SPACE;
                newboard[row][col] = EMPTY_SPACE;
                crow = mrow;
                ccol = mcol;
                newboard[mrow][mcol] = PLAYER1_KING;
                jumpKing(newboard, mrow, mcol, player);
            }
        }
    }
    else if (player == PLAYER2)
    {
        mrow = row - 2;
        mcol = col - 2;
        if (ValidPosition(newboard, mrow, mcol))

        {
            if (newboard[mrow + 1][mcol + 1] == PLAYER1 || newboard[mrow +
1][mcol + 1] == PLAYER1_KING)
            {
                //Then i can make a left jump
                newboard[mrow + 1][mcol + 1] = EMPTY_SPACE;
                newboard[row][col] = EMPTY_SPACE;
                crow = mrow;
                ccol = mcol;
                newboard[mrow][mcol] = PLAYER2_KING;
                jumpKing(newboard, mrow, mcol, player);
            }
        }
    }
}


/*---- Function:kingRightTop()-----*
Purpose: To check for the king position if it falls in right top.
Input: board[][],row,col,player,refernce to row,col
Return Type: void */

void Element::kingRightTop(char newboard[][8], int row, int col, char
player, int& crow, int& ccol)
{
    int mrow, mcol;
    if (player == PLAYER1)
    {
        mrow = row - 2;
        mcol = col + 2;
        if (ValidPosition(newboard, mrow, mcol))

        {
            if (newboard[mrow + 1][mcol - 1] == PLAYER2 || newboard[mrow +
1][mcol - 1] == PLAYER2_KING)
            {
                newboard[mrow + 1][mcol - 1] = EMPTY_SPACE;
                newboard[row][col] = EMPTY_SPACE;
                crow = mrow;
                ccol = mcol;
                newboard[mrow][mcol] = PLAYER1_KING;
```

```
                jumpKing(newboard, mrow, mcol, player);
            }
        }
    }
    else if (player == PLAYER2)
    {
        mrow = row - 2;
        mcol = col + 2;
        if (ValidPosition(newboard, mrow, mcol))

        {
            if (newboard[mrow + 1][mcol - 1] == PLAYER1 || newboard[mrow +
1][mcol - 1] == PLAYER1_KING)
            {
                //Then i can make a left jump
                newboard[mrow + 1][mcol - 1] = EMPTY_SPACE;
                newboard[row][col] = EMPTY_SPACE;
                crow = mrow;
                ccol = mcol;
                newboard[mrow][mcol] = PLAYER2_KING;
                jumpKing(newboard, mrow, mcol, player);
            }
        }
    }
}

/*---- Function:kingLeftJump()-----*
Purpose: King jump to the left is made
Input: board[][],row,col,player,reference to col,row
Return Type: void */

void Element::kingLeftJump(char newboard[][8], int row, int col, char
player, int& crow, int& ccol)
{
    int nrow, ncol;
    if (player == PLAYER1)
    {
        nrow = row + 2;
        ncol = col - 2;
        if (ValidPosition(newboard, nrow, ncol))

        {
            if (newboard[nrow - 1][ncol + 1] == PLAYER2 || newboard[nrow -
1][ncol + 1] == PLAYER2_KING)
            {
                newboard[nrow - 1][ncol + 1] = EMPTY_SPACE;
                newboard[row][col] = EMPTY_SPACE;
                crow = nrow;
                ccol = ncol;
                newboard[nrow][ncol] = PLAYER1_KING;
                jumpKing(newboard, nrow, ncol, player);
            }
        }
    }
    else if (player == PLAYER2)
    {
```

```
        nrow = row + 2;
        ncol = col - 2;
        if (ValidPosition(newboard, nrow, ncol))

        {
            if (newboard[nrow - 1][ncol + 1] == PLAYER1 || newboard[nrow -
1][ncol + 1] == PLAYER1_KING)
            {
                //Then i can make a left jump
                newboard[nrow - 1][ncol + 1] = EMPTY_SPACE;
                newboard[row][col] = EMPTY_SPACE;
                crow = nrow;
                ccol = ncol;
                newboard[nrow][ncol] = PLAYER2_KING;
                jumpKing(newboard, nrow, ncol, player);
            }
        }
    }
}

/*---- Function:kingRightJump()-----*
Purpose: King jump to the right is made
Input: board[][],row,col,player,reference to col,row
Return Type: void */

void Element::kingRightJump(char newboard[][8], int row, int col, char
player, int& crow, int& ccol)
{
    int nrow, ncol;
    if (player == PLAYER1)
    {
        nrow = row + 2;
        ncol = col + 2;
        if (ValidPosition(newboard, nrow, ncol))

        {
            if (newboard[nrow - 1][ncol - 1] == PLAYER2 || newboard[nrow -
1][ncol - 1] == PLAYER2_KING)
            {
                newboard[nrow - 1][ncol - 1] = EMPTY_SPACE;
                newboard[row][col] = EMPTY_SPACE;
                crow = nrow;
                ccol = ncol;
                newboard[nrow][ncol] = PLAYER1_KING;
                jumpKing(newboard, nrow, ncol, player);
            }
        }
    }
    else if (player == PLAYER2)
    {
        nrow = row + 2;
        ncol = col + 2;
        if (ValidPosition(newboard, nrow, ncol))

        {
```

```
            if (newboard[nrow - 1][ncol - 1] == PLAYER1 || newboard[nrow -
1][ncol - 1] == PLAYER1_KING)
            {
                //Then i can make a left jump
                newboard[nrow - 1][ncol - 1] = EMPTY_SPACE;
                newboard[row][col] = EMPTY_SPACE;
                crow = nrow;
                ccol = ncol;
                newboard[nrow][ncol] = PLAYER2_KING;
                jumpKing(newboard, nrow, ncol, player);
            }
        }
    }
}



/*---- Function:jumpKing()-----*
Purpose: King jump is made by checking whether it can jump to
left,right,lefttop or righttop
Input: board[][],row,col,player
Return Type: void */


void Element::jumpKing(char CheckerBoard[][8], int row, int col, char
player)
{
    int crow, ccol;
    crow = row;
    ccol = col;
    char newboard[8][8];
    copy(&CheckerBoard[0][0], &CheckerBoard[0][0] + 8 * 8,
&newboard[0][0]);

    kingLeftJump(newboard, row, col, player, crow, ccol);
    if (!compare(CheckerBoard, newboard) && kingLeftCorner(newboard, crow,
ccol, player) && kingRightCorner(newboard, crow, ccol, player) &&
kingTopLeftCorner(newboard, crow, ccol, player) &&
kingTopRightCorner(newboard, crow, ccol, player))
    {
        //Perform Action
        action(newboard);
    }
    copy(&CheckerBoard[0][0], &CheckerBoard[0][0] + 8 * 8,
&newboard[0][0]);

    kingRightJump(newboard, row, col, player, crow, ccol);
    if (!compare(CheckerBoard, newboard) && kingLeftCorner(newboard, crow,
ccol, player) && kingRightCorner(newboard, crow, ccol, player) &&
kingTopLeftCorner(newboard, crow, ccol, player) &&
kingTopRightCorner(newboard, crow, ccol, player))
    {
        //Perform Action
        action(newboard);
    }
```

```
    copy(&CheckerBoard[0][0], &CheckerBoard[0][0] + 8 * 8,
&newboard[0][0]);

    kingLeftTop(newboard, row, col, player, crow, ccol);
    if (!compare(CheckerBoard, newboard) && kingLeftCorner(newboard, crow,
ccol, player) && kingRightCorner(newboard, crow, ccol, player) &&
kingTopLeftCorner(newboard, crow, ccol, player) &&
kingTopRightCorner(newboard, crow, ccol, player))
    {
        //Perform Action
        action(newboard);
    }
    copy(&CheckerBoard[0][0], &CheckerBoard[0][0] + 8 * 8,
&newboard[0][0]);

    kingRightTop(newboard, row, col, player, crow, ccol);
    if (!compare(CheckerBoard, newboard) && kingLeftCorner(newboard, crow,
ccol, player) && kingRightCorner(newboard, crow, ccol, player) &&
kingTopLeftCorner(newboard, crow, ccol, player) &&
kingTopRightCorner(newboard, crow, ccol, player))
    {
        //Perform Action
        action(newboard);
    }
    copy(&CheckerBoard[0][0], &CheckerBoard[0][0] + 8 * 8,
&newboard[0][0]);
}

//****** Functions for the CHECKERS GAME ******//
```

Moves.cpp

```
#include <stdio.h>
#include <iostream>
#include "CheckerBoard.h"

using namespace std;

/*---- Function:action()-----*
Purpose: Move one place ahead by setting to newboard and setting to the
player
Input: newboard[][]
Return Type: void */

void Element::action(char newboard[][8])
{
    Element child1;
    child1.setBoard(newboard);
    char newplayer = (player == PLAYER1) ? PLAYER2 : PLAYER1;
    child1.setPlayer(newplayer);
    child1.move = move + 1;
    childs.push_back(child1);
    TotalElements++;
}
```

```
/*---- Function:moves()-----*
Purpose: Defining each moves a piece can take. Leftdiagnoal check,right
diagnoal check and check if its king
         and then performs the action.
Return Type: void */

void Element::moves()
{
    for (int i = 0; i < 8; i++)
    {
        for (int j = 0; j < 8; j++)
        {
            if (CheckerBoard[i][j] != EMPTY_SPACE)
            {
                if (player == PLAYER1 && CheckerBoard[i][j] ==
PLAYER1)//Computer normal player
                {
                    char newboard[8][8];
                    copy(&CheckerBoard[0][0], &CheckerBoard[0][0] + 8 * 8,
&newboard[0][0]);

                    //Left diagnoal check
                    if (Leftcheck(CheckerBoard, i, j, player))
                    {
                        // then can move left
                        if (checkKing(i + 1, player))
                            newboard[i + 1][j - 1] = PLAYER1_KING;
                        else
                            newboard[i + 1][j - 1] = PLAYER1;
                        newboard[i][j] = EMPTY_SPACE;
                        //Perform an Action
                        action(newboard);
                        copy(&CheckerBoard[0][0], &CheckerBoard[0][0] + 8 *
8, &newboard[0][0]);
                    }
                    //Right diagnoal check
                    if (Rightcheck(CheckerBoard, i, j, player))
                    {
                        //then can move right
                        if (checkKing(i + 1, player))
                            newboard[i + 1][j + 1] = PLAYER1_KING;
                        else
                            newboard[i + 1][j + 1] = PLAYER1;
                        newboard[i][j] = EMPTY_SPACE;
                        //Perform Action
                        action(newboard);
                        copy(&CheckerBoard[0][0], &CheckerBoard[0][0] + 8 *
8, &newboard[0][0]);
                    }
                    //Check jumps
                    jump(CheckerBoard, i, j, player);
                }
                else if (player == PLAYER2 && CheckerBoard[i][j] ==
PLAYER2) //user normal player
                {
                    char newboard[8][8];
```

```
                    copy(&CheckerBoard[0][0], &CheckerBoard[0][0] + 8 * 8,
&newboard[0][0]);

                    //Left diagnoal check
                    if (Leftcheck(CheckerBoard, i, j, player))
                    {
                        //then can move left
                        if (checkKing(i - 1, player))
                            newboard[i - 1][j - 1] = PLAYER2_KING;
                        else
                            newboard[i - 1][j - 1] = PLAYER2;
                        newboard[i][j] = EMPTY_SPACE;
                        //Perform Action
                        action(newboard);
                        copy(&CheckerBoard[0][0], &CheckerBoard[0][0] + 8 *
8, &newboard[0][0]);
                    }
                    //Right diagnoal Check
                    if (Rightcheck(CheckerBoard, i, j, player))
                    {
                        //then can move right
                        if (checkKing(i - 1, player))
                            newboard[i - 1][j + 1] = PLAYER2_KING;
                        else
                            newboard[i - 1][j + 1] = PLAYER2;
                        newboard[i][j] = EMPTY_SPACE;
                        //Perform Action
                        action(newboard);
                        copy(&CheckerBoard[0][0], &CheckerBoard[0][0] + 8 *
8, &newboard[0][0]);
                    }
                    //Check jumps
                    jump(CheckerBoard, i, j, player);
                }
                //When Computer player turned king
                if (player == PLAYER1 && CheckerBoard[i][j] ==
PLAYER1_KING)
                {
                    char newboard[8][8];
                    copy(&CheckerBoard[0][0], &CheckerBoard[0][0] + 8 * 8,
&newboard[0][0]);
                    //Check PLAYER1's left diagnoal
                    if (Leftcheck(CheckerBoard, i, j, PLAYER1))
                    {
                        //then can move left
                        newboard[i + 1][j - 1] = PLAYER1_KING;
                        newboard[i][j] = EMPTY_SPACE;
                        //Perform Action
                        action(newboard);
                        copy(&CheckerBoard[0][0], &CheckerBoard[0][0] + 8 *
8, &newboard[0][0]);
                    }
                    //Check PLAYER1's right diagnoal
                    if (Rightcheck(CheckerBoard, i, j, PLAYER1))
                    {
                        //then can move right
```

```
                        newboard[i + 1][j + 1] = PLAYER1_KING;
                        newboard[i][j] = EMPTY_SPACE;
                        //Perform Action
                        action(newboard);
                        copy(&CheckerBoard[0][0], &CheckerBoard[0][0] + 8 *
8, &newboard[0][0]);
                    }
                    //Check PLAYER2's left diagnoal
                    if (Leftcheck(CheckerBoard, i, j, PLAYER2))
                    {
                        //then can move left
                        newboard[i - 1][j - 1] = PLAYER1_KING;
                        newboard[i][j] = EMPTY_SPACE;
                        //Perform Action
                        action(newboard);
                        copy(&CheckerBoard[0][0], &CheckerBoard[0][0] + 8 *
8, &newboard[0][0]);
                    }
                    //Check PLAYER2's right diagnoal
                    if (Rightcheck(CheckerBoard, i, j, PLAYER2))
                    {
                        //I can move right
                        newboard[i - 1][j + 1] = PLAYER1_KING;
                        newboard[i][j] = EMPTY_SPACE;
                        //Perform Action
                        action(newboard);
                        copy(&CheckerBoard[0][0], &CheckerBoard[0][0] + 8 *
8, &newboard[0][0]);
                    }
                    //Checking for jumps
                    jumpKing(CheckerBoard, i, j, player);
                }
                if (player == PLAYER2 && CheckerBoard[i][j] ==
PLAYER2_KING) //User player turned king
                {
                    char newboard[8][8];
                    copy(&CheckerBoard[0][0], &CheckerBoard[0][0] + 8 * 8,
&newboard[0][0]);
                    //Check PLAYER1's left diagnoal
                    if (Leftcheck(CheckerBoard, i, j, PLAYER1))
                    {
                        //then can move left
                        newboard[i + 1][j - 1] = PLAYER2_KING;
                        newboard[i][j] = EMPTY_SPACE;
                        //Perform Action
                        action(newboard);
                        copy(&CheckerBoard[0][0], &CheckerBoard[0][0] + 8 *
8, &newboard[0][0]);
                    }
                    //Check PLAYER1's right diagnoal
                    if (Rightcheck(CheckerBoard, i, j, PLAYER1))
                    {
                        //then can move right
                        newboard[i + 1][j + 1] = PLAYER2_KING;
                        newboard[i][j] = EMPTY_SPACE;
                        //Perform Action
```

```
                        action(newboard);
                        copy(&CheckerBoard[0][0], &CheckerBoard[0][0] + 8 *
8, &newboard[0][0]);
                    }
                    //Check PLAYER2's left diagnoal
                    if (Leftcheck(CheckerBoard, i, j, PLAYER2))
                    {
                        //then can move left
                        newboard[i - 1][j - 1] = PLAYER2_KING;
                        newboard[i][j] = EMPTY_SPACE;
                        //Perform Action
                        action(newboard);
                        copy(&CheckerBoard[0][0], &CheckerBoard[0][0] + 8 *
8, &newboard[0][0]);
                    }
                    //Check PLAYER2's right diagnoal
                    if (Rightcheck(CheckerBoard, i, j, PLAYER2))
                    {
                        //then can move right
                        newboard[i - 1][j + 1] = PLAYER2_KING;
                        newboard[i][j] = EMPTY_SPACE;
                        //Perform Action
                        action(newboard);
                        copy(&CheckerBoard[0][0], &CheckerBoard[0][0] + 8 *
8, &newboard[0][0]);
                    }
                    //Checking for jumps
                    jumpKing(CheckerBoard, i, j, player);
                }
            }
        }
    }
}
```

EvaluationFunction.cpp

```
#include <stdio.h>
#include <iostream>
#include "CheckerBoard.h"
//#include "Enum.h"

using namespace std;

//****** Functions for the EVALUATION FUNCTION ******//
/*---- Function:Evaluate()-----*
Purpose: Evaluates the options and returns the number for that particular
player
Return Type: int - returns the value for that particular player. */


int Element::evaluate()
{
    if (opt == 1)
```

```
    {
        if (childs.empty())
        {
            if (player == PLAYER1)
                return -100;
            else
                return 100;
        }
    }
    else if (opt == 2)
    {
        if (childs.empty())
        {
            if (player == PLAYER1)
                return 100;
            else
                return -100;
        }
    }
    return EvalValue();
}

/*---- Function:EvalValue()-----*
Purpose: 3 Evaluation Functions are defined here. These are designed upon
considering giving weightage to player's kings more than the pawns and also
considered their defending neighbours.
Return Type: int - returns the value for that particular piece. */


int Element::EvalValue()
{
    int player1Count = 0, player2Count = 0;
    // EVALUATION FUNCTION 1
    // (Piece Difference) + 2*(KingPiece Difference) + 3*(Each Forward Move
Difference + Each King Backward Move Difference) + 4*(Each Capture
Difference + Each Capture in Backward By King)
    if ((opt == 1 && evalp1 == 1) || (opt == 2 && evalp2 == 1))
    {
        for(int i=0;i<8;i++)
        {
            for(int j=0;j<8;j++)
            {
                // Piece Count
                if(CheckerBoard[i][j] == PLAYER1)
                    player1Count++;
                else if(CheckerBoard[i][j] == PLAYER2)
                    player2Count++;
                //King Piece Count
                if(CheckerBoard[i][j] == PLAYER1_KING)
                    player1Count += 2;
                else if (CheckerBoard[i][j] == PLAYER2_KING)
                    player2Count += 2;

                //Possible Moves towards Opponents Direction for Player 1
                if(CheckerBoard[i][j] == PLAYER1 || CheckerBoard[i][j] ==
PLAYER1_KING)
```

```
                {
                    if((CheckerBoard[i+1][j-1] == EMPTY_SPACE //checks
right diagonal
                       || CheckerBoard[i+1][j+1] == EMPTY_SPACE) // checks
left diagonal
                       && i+1 < 8 && j-1 > -1 && j+1 < 8)
                        player1Count += 3;

                    // Possible Capture towards Opponents Direction
                    if((CheckerBoard[i+1][j-1] == PLAYER2 ||
                        CheckerBoard[i+1][j-1] == PLAYER2_KING) &&
                       CheckerBoard[i+2][j-2] == EMPTY_SPACE &&
                       i+2 < 8 && i+1 < 8 && j-1 > -1 && j-2 > -1)
                        player1Count += 4;
                    if((CheckerBoard[i+1][j+1]==PLAYER2 ||
                        CheckerBoard[i+1][j+1] == PLAYER2_KING) &&
                        CheckerBoard[i+2][j+2] == EMPTY_SPACE &&
                       i+2 < 8 && i+1 < 8 && j+1 < 8 && j+2 < 8)
                        player1Count += 4;
                }

                // Moves in backward direction by king
                if(CheckerBoard[i][j] == PLAYER1_KING)
                {
                    if((CheckerBoard[i-1][j-1] == EMPTY_SPACE //checks
right diagonal
                       || CheckerBoard[i-1][j+1] == EMPTY_SPACE) // checks
left diagonal
                       && i-1 > -1 && j-1 > -1 && j+1 < 8)
                        player1Count += 3;

                    // Capture in backward direction by king
                    if((CheckerBoard[i-1][j-1] == PLAYER2 ||
                        CheckerBoard[i-1][j-1] == PLAYER2_KING) &&
                       CheckerBoard[i-2][j-2] == EMPTY_SPACE &&
                       i-2 > -1 && i-1 > -1 && j-1 > -1 && j-2 > -1)
                        player1Count += 4;
                    if((CheckerBoard[i-1][j+1] == PLAYER2 ||
                        CheckerBoard[i-1][j+1] == PLAYER2_KING) &&
                        CheckerBoard[i-2][j+2] == EMPTY_SPACE &&
                       i-2 > -1 && i-1 > -1 && j+1 < 8 && j+2 < 8)
                        player1Count += 4;
                }

                //Possible Moves towards Opponents Direction for Player 2
                if(CheckerBoard[i][j] == PLAYER2 || CheckerBoard[i][j] ==
PLAYER2_KING)
                {
                    if((CheckerBoard[i-1][j-1] == EMPTY_SPACE //checks
right diagonal
                       || CheckerBoard[i-1][j+1] == EMPTY_SPACE) // checks
left diagonal
                       && i-1 > -1 && j-1 > -1 && j+1 < 8)
                        player2Count += 3;
                    // Possible Capture towards Opponents Direction
                    if((CheckerBoard[i-1][j-1] == PLAYER1 ||
```

```
                        CheckerBoard[i-1][j-1] == PLAYER1_KING) &&
                       CheckerBoard[i-2][j-2] == EMPTY_SPACE &&
                       i-2 > -1 && i-1 < -1 && j-1 > -1 && j-2 > -1)
                        player2Count += 4;
                    if((CheckerBoard[i-1][j+1] == PLAYER1 ||
                        CheckerBoard[i-1][j+1] == PLAYER1_KING) &&
                        CheckerBoard[i-2][j+2] == EMPTY_SPACE &&
                       i-2 > -1 && i-1 < -1 && j+1 < 8 && j+2 < 8)
                        player2Count += 4;
                }
                // Moves in backward direction by king
                if(CheckerBoard[i][j] == PLAYER2_KING)
                {
                    if((CheckerBoard[i+1][j-1] == EMPTY_SPACE //checks
right diagonal
                       || CheckerBoard[i+1][j+1] == EMPTY_SPACE) // checks
left diagonal
                       && i+1 < 8 && j-1 > -1 && j+1 < 8)
                        player2Count += 3;

                        // Capture in backward direction by king
                    if((CheckerBoard[i+1][j-1] == PLAYER1 ||
                        CheckerBoard[i+1][j-1] == PLAYER1_KING) &&
                       CheckerBoard[i+2][j-2] == EMPTY_SPACE &&
                       i+2 < 8 && i+1 < 8 && j-1 > -1 && j-2 > -1)
                        player2Count += 4;
                    if((CheckerBoard[i+1][j+1]==PLAYER1 ||
                        CheckerBoard[i+1][j+1] == PLAYER1_KING) &&
                        CheckerBoard[i+2][j+2] == EMPTY_SPACE &&
                       i+2 < 8 && i+1 < 8 && j+1 < 8 && j+2 < 8)
                        player2Count += 4;
                }
            }
        }
    }

    //Player 1 is positive and Player 2 is negative, taking count of all
pieces with more weightage towards kings and adding it with player count
difference , how close player is to become king and some randomness
    // EVALUATION FUNCTION 2
    else if ((opt == 1 && evalp1 == 2) || (opt == 2 && evalp2 == 2))
    {
        int a1 = 0, a2 = 0, b = 0, c = 0;
        for (int i = 0; i < 8; i++)
            for (int j = 0; j < 8; j++)
            {
                if (CheckerBoard[i][j] == PLAYER1)
                {
                    a1 += 2;
                    if (i == 0)
                        b += 9;
                    else b += i;
                    c += 1;
                }
                else if (CheckerBoard[i][j] == PLAYER2)
                {
```

```
                    a2 -= 2;
                    if (i == 7)
                        b -= 9;
                    else b -= (7 - i);
                    c -= 1;
                }
                else if (CheckerBoard[i][j] == PLAYER1_KING)
                {
                    a1 += 3;
                    c += 1;
                }
                else if (CheckerBoard[i][j] == PLAYER2_KING)
                {
                    a2 -= 3;
                    c -= 1;
                }
            }
        a1 *= 10000000;
        a2 *= 10000000;
        b *= 100000;
        c *= 1000;
        int e = rand() % 100;
        if (player == PLAYER2)
            e = -e;
        return a1 + a2 + b + c + e;
    }

    else if ((opt == 1 && evalp1 == 3) || (opt == 2 && evalp2 == 3))
    {
        //EVALUATION FUNCTION 3
        //Evaluation function 3 -  different weights are given to pieces
based on their poisition
        //center position + edge position + my piece + myking
        for (int i = 0; i < 8; i++)
        {
            for (int j = 0; j < 8; j++)
            {
                if(CheckerBoard[i][j] == PLAYER1)
                {
                    player1Count++;
                    //if piece in edge square
                    if( (i==3) && ((j==0) || (j==7)))
                        player1Count += 0.5;
                    //if piece in one of the four center squares
                    if (((i==3) && ((j == 2) || (j== 4))) || ((i==4) &&
((j==3)||(j==5))))
                        player1Count += 0.75;

                }else if(CheckerBoard[i][j] == PLAYER2){

                    player2Count++;
                    //if piece in edge square
                    if( (i==4) && ((j==0) || (j==7)))
                       player2Count += 0.5;
                    //if piece is in one of the four center squares
```

```
                    if (((i==3) && ((j == 2) || (j== 4))) || ((i==4) &&
((j==3)||(j==5))))
                        player2Count += 0.75;
                }
                if(CheckerBoard[i][j] == PLAYER1_KING){

                    player1Count += 2;
                    //if piece in edge square
                   if( (i==3) && ((j==0) || (j==7)))
                       player1Count += 0.5;
                    //if piece is in one of the four center squares
                    if (((i==3) && ((j == 2) || (j== 4))) || ((i==4) &&
((j==3)||(j==5))))
                        player1Count += 0.75;

                }else if (CheckerBoard[i][j] == PLAYER2_KING){

                    player2Count += 2;
                    //if piece in edge square
                   if((i==4) && ((j==0) || (j==7)))
                      player2Count += 0.5;
                    //if piece in one of the four center squares
                    if (((i==3) && ((j == 2) || (j== 4))) || ((i==4) &&
((j==3)||(j==5))))
                         player2Count += 0.75;
                }

            }
        }
    }
    if (opt == 1)
        return player1Count - player2Count;
    return player2Count - player1Count;
}


//****** Functions for the EVALUATION FUNCTION ******//
```