Implementing the Minmax Algorithm and Alpha-Beta pruning to analyze the performance statistics.

KALAH GAME

Meghana Bodduluri [A04898234]

Table of Contents

ABSTRACT

The objective of our project is to implement Minmax and Alpha beta pruning algorithms on Kalah game and analyzing the result by using different evaluation functions.

This project is study work for better understanding of these algorithms and analysis of the performance which includes documentation of statistics by trying the combinations of algorithm and evaluation function. The statistics include following details.

- Number of Nodes Expanded
- Number of Nodes Generated
- Total Memory Occupied
- Execution Time

These properties of an algorithm are calculated for the following sample cases of combinations to find out the best case scenario and worst case scenario.

- Minmax Versus Minmax
  - Evaluation 2 for player1 and Evaluation 1 for player 2
  - Evaluation 2 for player1 and standard evaluation function for player 2
  - Standard evaluation function for player 1 and Evaluation 1 for player 2
- Alpha Beta pruning Versus Alpha Beta pruning
  - Evaluation 2 for player1 and Evaluation 1 for player 2
  - Evaluation 2 for player1 and standard evaluation function for player 2
  - Standard evaluation function for player 1 and Evaluation 1 for player 2
- Alpha Beta pruning Versus Minmax
  - Evaluation 2 for player1 and Evaluation for player 2
  - Evaluation 2 for player1 and standard evaluation function for player 2
  - Standard evaluation function for player 1 and Evaluation 1 for player 2

These cases are repeated taking player2 plays first.

INTRODUCTION

Game theory algorithm is an extensively explored field of computer science and more particularly of artificial intelligence as it is a domain where common methods of reasoning are to be applied, while offering a closed world, completely described by the game rules. More over the quality of the play is easy to evaluate quantitatively by observing the matches outcomes.

One way of looking at all the search procedures is that they are essentially generate-and- test procedures in which the testing is done after varying amounts of work by the generator. Because of their adversarial nature in 2 player games, as values are passed back up, different assumptions must be made at levels where the program chooses the move and at the alternating levels where the opponent chooses, A* procedure is inadequate for two-person games.

We have two Algorithms – Minimax and Alpha-Beta Pruning which are most commonly used methods for Two-player game implementation.

FORMAL DESCRIPTION OF THE PROBLEM

To Develop programs by implementing algorithms MINMAX-A-B and ALPHA-BETA-SEARCH in Java language. Devise Deep-Enough and Move-Gen functions.

Using Kalah game as an example to test the program.

Goals of the project:

1. Develop Minimax and Alpha-Beta-Pruning algorithms.
2. Develop Kalah Game and its various Evaluation Functions.
3. Use this game to provide an analysis.
4. Comparing the evaluation functions efficiency by using same algorithm strategy for both the players, but different evaluation functions.
5. Evaluating the time and space complexity for both the algorithms. Provide statistics between the two algorithms in terms of the following

- Number of Nodes Expanded
- Number of Nodes Generated
- Total Memory Occupied
- Execution Time

6. Win/Lose Statistics: This gives the count of wins/loose when re-run with same first player or different first player. This will be covered as part of Result Analysis. And not by the Program.

Input for the project

1. Input.txt file contains board details about Cutting depth, number of pits, number of stones in pit initially and number of players.
2. Initially, user have to select the choice between minmax and alpha beta algorithm as player1.
3. select the choice between minmax and alpha beta algorithm as player 2.
4. Select the choice for evaluation function for player1.
5. Select the choice for evaluation function for player2.
6. Value for who starts the game first(player1/player2).

Output of the project

1. Displays the game board for every move selected for player.
2. Evaluation values generated for each node.
3. Displays the name of the optimum node generated.
4. If no moves are available, displays which player won the game.
5. Prints the statistics values calculated for number of nodes generated, number of nodes expanded, memory used.

TEAM MEMBERS

Team Involved:

Divya Viswanathan(A04859120)

Meghana Bodduluri(A04898234)

Contribution of each:

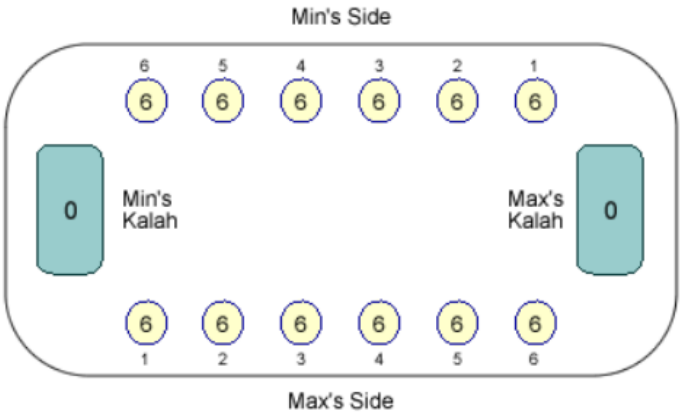
Kalah Game: Divya and I studied the Kalah game flow and its rules were learnt individually. Various Evaluation Functions were designed individually. After this, considering the game strategy we, deduced the evaluation function where one evaluation function works for free turns, as well as to capture stones and other looks for best move to capture maximum number of stones.

Program: I was involved majorly in implementation of Alpha-Beta Algorithm. I took a lot of references for the algorithm flow and logic, while Divya worked in the implementation of Minmax Algorithm. We then worked on integration of the two modules together. We reviewed each-others code in terms of following: -

- Following the coding standards
- Generalization of code by combining the algorithm and move gen functions into one class and utility functions into one class.
- Output statements displayed in a better way for better understanding of game.
- In-line comments for the logic applied in program for easy understanding the flow of game.
- Optimizing the lines of code by defining common functions separately and called by algorithm individually.
- Using try, catch blocks to handle exceptions.

KALAH GAME

Kalah is an abstract strategy game invented in 1940 by William Julius Champion, Jr. The notation (m,n)-Kalaha refers to Kalaha with m pits per side and n stones in each pit. In 2000, Kalaha was solved for all $m \leq 6$ and $n \leq 6$.



Game Components

- Board with 2 rows of 6 holes
- 72 colored stones
- The pits on the left and the right side of the board, each belongs to a Player and can be called as Player A's Kalah and Player B's Kalah.

Objective of game

The objective of the game is to collect as many pieces as possible before one of the player's side gets cleared

Capture a stone

- If you place the last stone of your turn into an empty hole on your side of the board, you capture all the pieces in the hole directly across from it on your opponent's side of the board.
- Take all the captured stones and place them in your kalah.

End of the game

- As soon as all the small six holes on one side of the board have been emptied, the game ends.
- The player who still has stones on his side of the board when the game ends captures all those stones too.
- The players now count their stones in their kalah and the player with the most stones wins the game.

GAME PLAYING

We first consider games with two players, whom we call MAX and MIN for reasons that will soon become obvious. MAX moves first, and then they take turns moving until the game is over. At the end of the game, points are awarded to the winning player and penalties are given to the loser. A game can be formally defined as a kind of search problem with the following elements:

- S0: The initial state, which specifies how the game is set up at the start. We will have a total of 72 stones (or pieces) and these 72 stones will be divided equally into all of those (6+6) pits. Player who sits south of board starts the game.
- PLAYER(s): Defines which player has the move in a state(Max/Min).
- ACTIONS(s): Returns the set of legal moves in a state.(getallMoves() function).
- RESULT(s, a): The transition model, which defines the result of a move.
- TERMINAL-TEST(s): A terminal test, which is true when the game is over and false otherwise. States where the game has ended are called terminal states.
- UTILITY(s, p) : utility function (also called an objective function or payoff function), defines the final numeric value for a game that ends in terminal states for a player p.

An **evaluation function**, also known as a heuristic evaluation function or static evaluation function, is a function used by game-playing programs to estimate the value or goodness of a position in the minimax and related algorithms. For Kalah Game, an evaluation function is built on these strategies.

Evaluation Function1(Play Defensive)

If you can't capture opponent's stones, then make a move on your turn that prevents your pieces from being captured by moving stones into the opponent's empty pit. This will not maximize stones going into your own kalah, but it can stop your opponent from capturing your stones.

You should also try to remove the opponent's ability to move more than once during his turn by dropping a stone in the hole that would have allowed your opponent to end his turn in his mancala.

Evaluation Function2(Empty wisely your own holes)

Create empty holes on your side of the board where your opponent's hole is not empty in order to capture his/her stones.

Empty your rightmost hole already early in the game as this is directly next to your kalah zone. Whenever you pick up a single stone from that hole as your move, you will score a point and get another move. Your next move should be to drop stones into your mancala for another free point, and then move again.

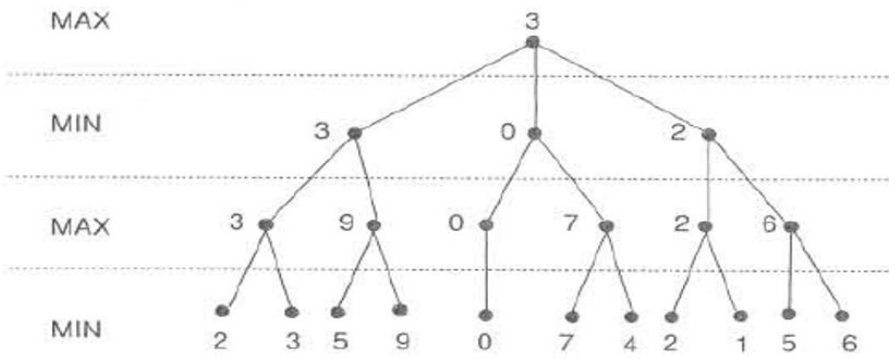
Evaluation functions used in program:

1. (#Stones in 1st player Kalah + #Stones in 1st player pits) - (#Stones in 2nd player Kalah + #Stones in 2nd player pits)
2. (number of my empty pits) - (number of opponent empty pits)
3. Standard Evaluation- (#Stones in 1st player Kalah - #Stones in 2nd player Kalah)

MINIMAX ALGORITHM

Description:

Minimax is a kind of backtracking algorithm that is used in decision making and game theory to find the optimal move for a player, assuming that your opponent also plays optimally. It is widely used in two player turn-based games.



In Minimax the two players are called maximizer and minimizer. The **maximizer** tries to get the highest score possible while the **minimizer** tries to do the opposite and get the lowest score possible.

Every board state has a value associated with it. In a given state if the maximizer has upper hand then, the score of the board will tend to be some positive value. If the minimizer has the upper hand in that board state then it will tend to be some negative value. The values of the board are calculated by some heuristics which are unique for every type of game.

Minmax Algorithm Flow

MINIMAX (CURRENT, 0, PLAYER-ONE)

if PLAYER-ONE is to move, or

MINIMAX (CURRENT, 0, PLAYER-TWO)

if PLAYER-TWO is to move

- MINIMAX (Position, Depth, Player)

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

VALUE = STATIC (Position, Player);

PATH = nil

This indicates that there is no path from this node and that its value is that determined by the static evaluation function.

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

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

4. If SUCCESSORS is not empty, then examine each element in turn and keep track of the best one. This is done as follows:

Initialize BEST-SCORE to the minimum value that STATIC can return. It will be updated to reflect the best score that can be achieved by an element of SUCCESSORS.

For each element SUCC of SUCCESSORS, do the following:

(a) Set RESULT-SUCC to

MINIMAX (SUCC, Depth + 1, OPPOSITE (Player))

This recursive call to MINIMAX will actually carry out the exploration of SUCCC.

(b) Set NEW-VALUE to - VALUE (RESULT-SUCC). This will cause it to reflect the merits of the

position from the opposite perspective from that of the next lower level.

(c) If NEW-VALUE> BEST-SCORE, 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 BEST-SCORE to NEW-VALUE.

(ii) The best-known path is now from CURRENT to SUCC and then on to the appropriate path down from SUCC as determined by the recursive call to MINIMAX. So, set BEST-PATH to the result of attaching SUCC to the front of PATH (RESULT-SUCC).

5. Now that all the successors have been examined, we know the value of Position as well as which path to take from it. So, return the structure

VALUE = BEST-SCORE

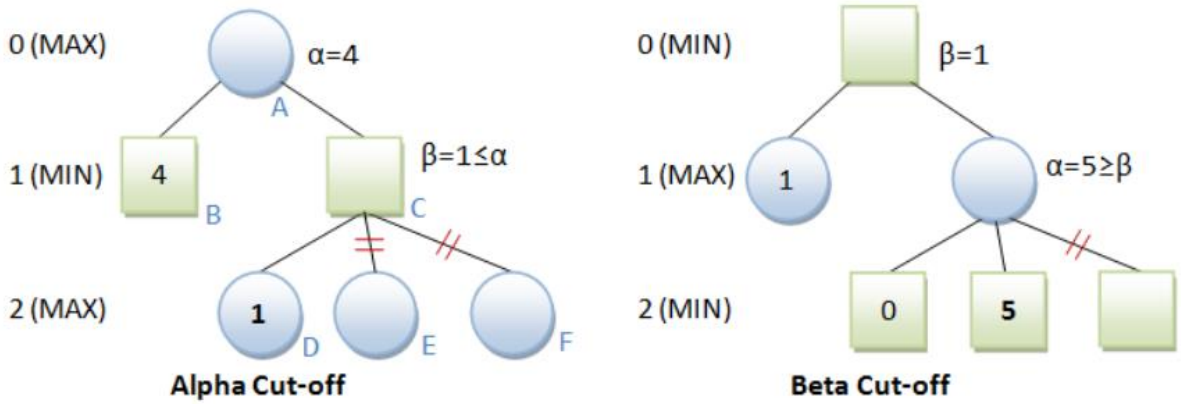
PATH = BEST-PATH

ALPHA BETA PRUNING

Description:

Alpha-Beta pruning is an optimization technique for minimax algorithm. It reduces the computation time by a huge factor. This allows us to search much faster and even go into deeper levels in the game tree. It cuts off branches in the game tree which need not be searched because there already exists a better move available. It is called Alpha-Beta pruning because it passes 2 extra parameters in the minimax function, namely alpha and beta.

Let's define the parameters alpha and beta.
Alpha is the best value that the **maximizer** currently can guarantee at that level or above.
Beta is the best value that the **minimizer** currently can guarantee at that level or above.



Alpha cutoff

- The initial call starts from **A**. The value of alpha here is **-INFINITY** and the value of beta is **+INFINITY**. These values are passed down to subsequent nodes in the tree. At **A** the maximizer must choose max of **B** and **C**, so **A** calls **B** first.
- **B** returns 4 to **A**. At **A**, alpha = max(-INF, 4) which is 4. Now the maximizer is guaranteed a value of 4 or greater. **A** now calls **C** to see if it can get a higher value than 4.
- At **C**, alpha = 4 and beta = +INF. **C** calls **D**
- **D** returns 1 to **C**. alpha = max(4, 1) which is still 4.
- At **C**, beta = min(+INF, 1). The condition beta <= alpha becomes true as beta = 1 and alpha = 4. So, it breaks and it does not even have to compute the nodes.

Beta cutoff

- The initial call starts from **A**. The value of alpha here is **-INFINITY** and the value of beta is **+INFINITY**. These values are passed down to subsequent nodes in the tree. At **A** the minimizer must choose min of **B** and **C**, so **A** calls **B** first.
- **B** returns 1 to **A**. At **A**, beta = min(+INF, 1) which is 1. Now the minimizer is guaranteed a value of 1 or lesser. **A** now calls **C** to see if it can get a lower value than 1.
- At **C**, beta = 1 and alpha = -INF. **C** calls **D**
- **D** returns 0 to **C**. beta = min(1,0) which makes beta = 0. **C** calls **E**
- **E** returns 5 to **C**. alpha = max(-INF,5) which makes alpha=5. The condition alpha>= beta becomes true as beta = 1 and alpha = 5. So, it breaks and it does not even have to compute the other nodes.

Alpha-Beta Algorithm Flow

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

1. If DEEP-ENOUGH(Position, Depth), then return the structure VALUE = STATlC (Position, Player);PATH = nil

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

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

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

(i) Set RESULT-SUCC to MINIMAX-A-B(SUCC, Depth + 1, OPPOSlTE (Player), - Pass-Thresh, - Use-Thresh).

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

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

i. Set Pass-Thresh to NEW-VALUE.

ii. The best-known path is now from CURRENT to SUCC and then on to the appropriate path from SUCC as determined by the recursive call to MINIMAX-A-B. So, set BEST-PATH to the result of attaching SUCC to the front of PATH(RESULT-SUCC).

(iv) 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 Structure

VALUE = Pass-Thresh

PATH = BEST-PATH

IMPLEMENTED PROGRAM

Kalah1.java

```
package kalah1;
import java.io.BufferedReader;
import java.io.FileInputStream;
import java.io.IOException;
import java.io.InputStreamReader;
import java.util.Scanner;

/**
 *
 * @author meghana
 */
public class Kalah1 {

    public static void main(String[] args) throws IOException{
        FileInputStream fileInputStream=null;
        InputStreamReader inputStreamReader=null;
        BufferedReader bufferedReader=null;

        try{
            fileInputStream=new FileInputStream(args[0]);
            inputStreamReader=new InputStreamReader(fileInputStream);
            bufferedReader=new BufferedReader(inputStreamReader);
            Scanner input =new Scanner(System.in);

            //Read input file
            int cuttingDepth=Integer.parseInt(bufferedReader.readLine().trim());
```

```
String[] boardPlayer2=bufferedReader.readLine().trim().split(" ");
String[] boardPlayer1=bufferedReader.readLine().trim().split(" ");

int boardSize=boardPlayer2.length;
int[] board2=new int[boardSize];
int[] board1=new int[boardSize];

for(int i=0;i<boardSize;i++){
   board2[i]=Integer.parseInt(boardPlayer2[i].trim());
   board1[i]=Integer.parseInt(boardPlayer1[i].trim());
}

int kalah2=Integer.parseInt(bufferedReader.readLine().trim());
int kalah1=Integer.parseInt(bufferedReader.readLine().trim());

Helper util=new Helper();
KalahBoard next=null;
Algorithms.Minimax mm;
Algorithms.AlphaBeta ab;
//Call Minimax or AlphaBeta method based on value specified in input

System.out.println("----------------------------------------");
System.out.println(" Welcome to KALAH Game ");
System.out.println("----------------------------------------");
System.out.println();
System.out.println("Algorithms");
System.out.println("-----------");
System.out.println("1. Minimax");
System.out.println("2. AlphaBetaPruning");
```

```
System.out.println("Choose algorithm for your 1st player : ");
int player1algo = Integer.parseInt(input.nextLine().trim());
System.out.println("Choose algorithm for your 2nd player :");
int player2algo = Integer.parseInt(input.nextLine().trim());


System.out.println();
System.out.println("Evaluation Functions");
System.out.println("--------------------");
System.out.println("1. (#Stones in 1st player Kalah + #Stones in 1st player pits) - (#Stones in 2nd player Kalah + #Stones in 2nd player pits) ");
System.out.println("2. (number of my emptypits) - (number of opponent empty pits)  ");
System.out.println("Standard Evaluation-  (#Stones in 1st player Kalah -  #Stones in 2nd player Kalah) ");
System.out.println("Choose evaluation function for 1st player : ");
int player1eval = Integer.parseInt(input.nextLine().trim());
System.out.println("Choose evaluation function for 2nd player : ");
int player2eval = Integer.parseInt(input.nextLine().trim());


System.out.println();
System.out.println("1. Player #1");
System.out.println("2. Player #2");
System.out.println("Who goes first? : ");
int currentplayer = Integer.parseInt(input.nextLine().trim());


  // untill game reaches end
do{
   KalahBoard g=new KalahBoard(currentplayer, board2, board1, kalah2, kalah1);
   //If player1 starts game
   if(currentplayer == 1)
   {
      //selected algo is minimax
```

```
    if(player1algo == 1)
    {
       mm=new Algorithms.Minimax(cuttingDepth);
       next=mm.minimax(g, player1eval);

       board1=next.getPlayer1_Board();
       board2=next.getPlayer2_Board();
       kalah1=next.getPlayer1_Kalah();
       kalah2=next.getPlayer2_Kalah();

       currentplayer=2;
    }
    //selected algo alphabeta pruning
    else
    {
       ab=new Algorithms.AlphaBeta(cuttingDepth);
       next=ab.alphabeta(g, player1eval);

       board1=next.getPlayer1_Board();
       board2=next.getPlayer2_Board();
       kalah1=next.getPlayer1_Kalah();
       kalah2=next.getPlayer2_Kalah();

       currentplayer=2;
    }


}
//If player2 starts the game
else
```

```
    {
      if(player2algo == 1)
       {
          mm=new Algorithms.Minimax(cuttingDepth);
          next=mm.minimax(g, player2eval);

          board1=next.getPlayer1_Board();
          board2=next.getPlayer2_Board();
          kalah1=next.getPlayer1_Kalah();
          kalah2=next.getPlayer2_Kalah();

          currentplayer=1;
       }
       else
       {
          ab=new Algorithms.AlphaBeta(cuttingDepth);
          next=ab.alphabeta(g, player2eval);

          board1=next.getPlayer1_Board();
          board2=next.getPlayer2_Board();
          kalah1=next.getPlayer1_Kalah();
          kalah2=next.getPlayer2_Kalah();

          currentplayer=1;
       }
    }
}while(!util.GameOver(next));

if(next.getPlayer2_Kalah()>next.getPlayer1_Kalah())
    System.out.println("PLAYER - 2 WON THE GAME");
```

```
            else if(next.getPlayer2_Kalah()< next.getPlayer1_Kalah())
                System.out.println("PLAYER - 1 WON THE GAME");
            else
                System.out.println("Its a Tie");

            System.out.println("\nStatistics");
            System.out.println("------------------------------------------------");
            System.out.println("PLAYER-1");
            System.out.println("No of Nodes Expanded by Player 1 is "+ Statistics.ExpandedNodes1);
            System.out.println("No of Nodes Generated by Player 1 is "+ Statistics.noofNodes1);
            System.out.println("Memory Used : " + Statistics.noofNodes1*4 + "Bytes");
            System.out.println("------------------------------------------------");
            System.out.println("PLAYER-2");
            System.out.println("No of Nodes Expanded by Player 2 is "+ Statistics.ExpandedNodes2);
            System.out.println("No of Nodes Generated by Player 2 is "+ Statistics.noofNodes2);
            System.out.println("Memory Used : " + Statistics.noofNodes2*4 + "Bytes");
        }
        catch(Exception ex){
            System.out.println("Exception occured : " + ex);
            ex.printStackTrace();
        }
        finally{
            bufferedReader.close();
            inputStreamReader.close();
            fileInputStream.close();
        }
    }
}
```

Algorithms.java

```
package kalah1;

import java.io.BufferedWriter;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.OutputStreamWriter;
import java.util.ListIterator;

public class Algorithms {

    /**
     *
     * @author meghana
     */
    /* Get next best move based on Alpha Beta technique
     * And traverse log shows the exploration of game
     * tree in order to select best next move.
     */
    public static class AlphaBeta {

        Helper helper = new Helper();
        private final int depth;
        private static int count = 0;

        public AlphaBeta(int depth) {
            super();
            this.depth = depth;
        }
```

```
    //Game tree exploration
    public KalahBoard alphabeta(KalahBoard g, int eval) throws IOException {
        KalahBoard nextstate = null;
        try {
            KalahBoard.Node root = new KalahBoard.Node("root", g, Double.NEGATIVE_INFINITY,
Double.NEGATIVE_INFINITY, Double.POSITIVE_INFINITY, 0, true, null);
            getAllMoves_AB(root, eval);
            nextstate = helper.nextState(root);
        } catch (Exception ex) {
            System.out.println("Exception in AlphaBeta : " + ex.getMessage());
        }
        return nextstate;
    }

    /* Get all the valid moves based on current state
     * and select next best move
     */
    public void getAllMoves_AB(KalahBoard.Node n, int eval) throws IOException {
        count = n.depth;
        //Leaf Node
        if (count >= depth && !n.game.getGetAnotherTurn()) {
            if (eval == 1) {
                n.eval = helper.eval1(n.game);
            } else if (eval == 2) {
                n.eval = helper.eval2(n.game);
            } else {
                n.eval = helper.eval(n.game.getPlayer(), n.game.getPlayer1_Kalah(),
n.game.getPlayer2_Kalah());
            }
            return;
```

```
}
boolean valid = false;
if (count == depth && n.game.getGetAnotherTurn()) {
   valid = true;
}
while ((count < depth || valid) && !helper.GameOver(n.game)) {
   helper.expansion(n);
   valid = false;
   ListIterator<KalahBoard.Node> listIterator = n.children.listIterator();
   while (listIterator.hasNext()) {
      KalahBoard.Node temp = listIterator.next();
      temp.alpha = n.alpha;
      temp.beta = n.beta;
      getAllMoves_AB(temp, eval);
      if (n.max) {
         if (temp.eval > n.eval) {
            n.eval = temp.eval;
         }
         if (!prun(n)) {
            if (temp.eval > n.alpha) {
               n.alpha = temp.eval;
            }
         }
      } else {
         if (temp.eval < n.eval) {
            n.eval = temp.eval;
         }
         if (!prun(n)) {
            if (temp.eval < n.beta) {
               n.beta = temp.eval;
```

```
                }
            }
        }
        if (prun(n)) {
            break;
        }
    }
    count++;
  }
}

//Check pruning condition
public boolean prun(KalahBoard.Node n) {
  if (n.max) {
    if (n.beta <= n.eval) {
      //Pruning
      return true;
    } else {
      return false;
    }
  } else {
    if (n.alpha >= n.eval) {
      //Pruning
      return true;
    } else {
      return false;
    }
  }
}
```

```
/* Get next best move based on Minimax technique
 * And traverse log shows the exploration of game
 * tree in order to select best next move.
 */
public static class Minimax {

    Helper helper = new Helper();
    private final int depth;
    private static int count = 0;

    public Minimax(int depth) {
        super();
        this.depth = depth;
    }

    //Game tree exploration
    public KalahBoard minimax(KalahBoard g, int eval) {
        KalahBoard nextstate = null;
        try {
            KalahBoard.Node root = new KalahBoard.Node("root", g, Double.NEGATIVE_INFINITY, 0,
true, null);
            getAllMovesMin(root, eval);
            nextstate = helper.nextState(root);
        } catch (Exception ex) {
            System.out.println("Exception in Minimax : " + ex);
            ex.printStackTrace();
        }
        return nextstate;
    }
```

```
    /* Get all the valid moves based on current state
     * and select next best move
     */
    public void getAllMovesMin(KalahBoard.Node n,int eval) throws IOException {
        count = n.depth;
        if (count >= depth && !n.game.getGetAnotherTurn()) {
            if (eval == 1) {
                n.eval = helper.eval1(n.game);
            } else if (eval == 2) {
                n.eval = helper.eval2(n.game);
            } else {
                n.eval = helper.eval(n.game.getPlayer(), n.game.getPlayer1_Kalah(),
n.game.getPlayer2_Kalah());
            }
            return;
        }
        boolean valid = false;
        if (count == depth && n.game.getGetAnotherTurn()) {
            valid = true;
        }
        while ((count < depth || valid) && !helper.GameOver(n.game)) {
            helper.expansion(n);
            valid = false;
            ListIterator<KalahBoard.Node> listIterator = n.children.listIterator();
            while (listIterator.hasNext()) {
                KalahBoard.Node temp = listIterator.next();
                getAllMovesMin(temp, eval);
                if (n.max) {
                    if (n.eval < temp.eval) {
```

```
                    n.eval = temp.eval;
                }
            } else {
                if (n.eval > temp.eval) {
                    n.eval = temp.eval;
                }
            }
        }
        count++;
    }
  }
}
```

Helper.java

```
package kalah1;
import java.io.*;
import java.util.ListIterator;

class Statistics{
   public static int noofNodes1, noofNodes2;
   public static int ExpandedNodes1, ExpandedNodes2;

   public Statistics()
   {
      noofNodes1 = noofNodes2 = ExpandedNodes1 = ExpandedNodes2 = 0;
   }
}
```

```
//Class with common functionality used by other classes
public class Helper {

    int[] localBoard1, localBoard2;
    int localKalah1, localKalah2;

    //Copy values in local variables
    public void setLocals(KalahBoard x){
        this.localKalah1=x.getPlayer1_Kalah();
        this.localKalah2=x.getPlayer2_Kalah();
        localBoard1=new int[x.getBoardSize()];
        localBoard2=new int[x.getBoardSize()];
        System.arraycopy(x.getPlayer1_Board(), 0, localBoard1, 0, x.getBoardSize());
        System.arraycopy(x.getPlayer2_Board(), 0, localBoard2, 0, x.getBoardSize());
    }

    /* The following function comprises of the moves made
     * game argument is current state of game
     * player indicates current player who is playing
     * index indicate the hole from which stones needs to be moved
     *
     * This function returns the new state of game after moving stones
     * for given player from given hole index
     */
    public KalahBoard moveStones(KalahBoard game, int player, int index){
        KalahBoard state=new KalahBoard(game);
        state.setGetAnotherTurn(false);
        setLocals(state);
        if(player==1){
            if(localBoard1[index]>0){
```

```
int stones=localBoard1[index];
localBoard1[index]=0;
int j=index+1;
boolean reverse=false;
while(stones>0){
   if(j==state.getBoardSize()){
      if(stones==1)
         state.setGetAnotherTurn(true);
      localKalah1++;
      stones--;
      reverse=true;
      j=state.getBoardSize()-1;
   }
   else if(j<0){
      j=0;
      reverse=false;
   }
   else if(reverse){
      localBoard2[j]++;
      stones--;
      j--;
   }
   else{
      if(stones==1 && localBoard1[j]==0){
         localKalah1+=localBoard2[j]+1;
         localBoard2[j]=0;
         stones--;
      }
      else{
         localBoard1[j]++;
```

```
                stones--;
                j++;
            }
        }
    }
}
else{
    if(localBoard2[index]>0){
        int stones=localBoard2[index];
        localBoard2[index]=0;
        int j=index-1;
        boolean reverse=true;
        while(stones>0){
            if(j<0){
                if(stones==1)
                    state.setGetAnotherTurn(true);
                localKalah2++;
                stones--;
                reverse=false;
                j=0;
            }
            else if(j==state.getBoardSize()){
                j=state.getBoardSize()-1;
                reverse=true;
            }
            else if(!reverse){
                localBoard1[j]++;
                stones--;
                j++;
```

```
            }
            else{
                if(stones==1 && localBoard2[j]==0){
                    localKalah2+=localBoard1[j]+1;
                    localBoard1[j]=0;
                    stones--;
                }
                else{
                    localBoard2[j]++;
                    stones--;
                    j--;
                }
            }
        }
    }
}
if(Player1_SideEmpty(localBoard1)){
    for(int i=0;i<localBoard2.length;i++){
        localKalah2+=localBoard2[i];
        localBoard2[i]=0;
    }
    state.setGetAnotherTurn(false);
}
if(Player2_SideEmpty(localBoard2)){
    for(int i=0;i<localBoard1.length;i++){
        localKalah1+=localBoard1[i];
        localBoard1[i]=0;
    }
    state.setGetAnotherTurn(false);
}
```

```
    state.setPlayer1_Board(localBoard1);
    state.setPlayer2_Board(localBoard2);
    state.setPlayer1_Kalah(localKalah1);
    state.setPlayer2_Kalah(localKalah2);
    return  state;
}

/*
 * x is current state of game
 *
 * The function returns if the game is over or not on current state of game
 */
public boolean GameOver(KalahBoard x){
    int[] temp1=new int[x.getBoardSize()];
    int[] temp2=new int[x.getBoardSize()];
    System.arraycopy(x.getPlayer1_Board(), 0, temp1, 0, x.getBoardSize());
    System.arraycopy(x.getPlayer2_Board(), 0, temp2, 0, x.getBoardSize());

    boolean side1Empty=true;
    boolean side2Empty=true;
    for(int i=0;i<x.getBoardSize();i++){
        if(temp1[i]>0){
            side1Empty=false;
            break;
        }
    }
    for(int i=0;i<x.getBoardSize();i++){
        if(temp2[i]>0){
            side2Empty=false;
```

```
            break;
        }
    }
    return (side1Empty || side2Empty);
}

/*
 * player is current player
 * Kalah1 is number of stones in player 1's kalahpit
 * Kalah2 is number of stones in player 2's kalahpit
 *
 * This function returns the heuristic (Eval value) for current player
 * which helps in deciding the next best move
 */
public int eval(int player, int kalah1, int kalah2){
    if(player==1){
        return kalah1-kalah2;
    }
    else{
        return kalah2-kalah1;
    }
}

//Evaluates total number of stones in each pit + Kalah
public int eval1(KalahBoard g1){
    setLocals(g1);
    int sum1=0, sum2=0;
    for(int i=0; i< localBoard1.length; i++)
        sum1+=localBoard1[i];
    sum1+=localKalah1;
```

```
    for(int i=0; i< localBoard2.length; i++)
        sum2+=localBoard2[i];
    sum2+=localKalah2;
    if(g1.getPlayer()==1){
        return sum1-sum2;
    }
    else{
        return sum2-sum1;
    }
}

//Evaluates the number of empty pits on each side
public int eval2(KalahBoard g1){
    setLocals(g1);
    int count1=0, count2=0;
    for(int i=0; i< localBoard1.length; i++)
    {
        if(localBoard1[i]==0)
        count1++;
    }
    for(int i=0; i< localBoard2.length; i++)
    {
        if(localBoard2[i]==0)
        count2++;
    }
    if(g1.getPlayer()==1){
        return count1-count2;
    }
    else{
        return count2-count1;
```

```
    }
}

/*
 * player is current player
 * index is hole index for current player on the game board
 *
 * This function returns the name for hole
 * For below representation
 *                Player 2
                    1   2   3   4   5
                   ---------------------
                  A|  |  |  |  |  |
                  |          -------------
                  B|  |  |  |  |  |
                   ---------------------
                                Player 1

                Example For player 1 second hole will be represented as B3
 */
public String getName(int player, int index){
    String name="";
    if(player==1){
        name="A".concat(Integer.toString(index+1));
    }
    else{
        name="B".concat(Integer.toString(index+1));
    }
    return name;
}
```

```
/*
 * name is String represenation of hole
 *
 * This function is opposite of getName,
 * it returns the player and index of hole
 *
 * For below representation
 *                Player 2
                    1   2   3   4   5
                   ---------------------
                  A|  |  |  |  |  |
                  |  -------------
                  B|  |  |  |  |  |
                   ---------------------
                                  Player 1

                  Example : A4 will be player 2's 3rd hole
 */
public int[] decodeName(String name){
    int[] returnValue=new int[2];// returnValue[0] is player number and returnValue[1] is index in board
    if(name.charAt(0)=='A'){
        returnValue[0]=1;
    }
    else{
        returnValue[0]=2;
    }
    int temp=Integer.parseInt(name.substring(1));
    returnValue[1]=temp-2;
    return returnValue;
```

```
}

/*
 * n is some node which contain  state of game
 * This function expands the current state into further valid states
 * and create node for add all the valid states and
 * add them as children of current node
 */
public void expansion(KalahBoard.Node n){
   //System.out.println("MoveStones function is called");
   if(!(GameOver(n.game))){
      int currentPlayer;
      if(n.game.getPlayer()==1)
         Statistics.ExpandedNodes1++;
      else Statistics.ExpandedNodes2++;
      if(n.max)
         currentPlayer=n.game.getPlayer();
      else{
         if(n.game.getPlayer()==1)
            currentPlayer=2;
         else
            currentPlayer=1;
      }

      int[] checkValidMove=new int[n.game.getBoardSize()];
      if(currentPlayer==1){
         System.arraycopy(n.game.getPlayer1_Board(), 0, checkValidMove, 0, n.game.getBoardSize());
      }
      else{
         System.arraycopy(n.game.getPlayer2_Board(), 0, checkValidMove, 0, n.game.getBoardSize());
```

```
}
for(int i=0;i<n.game.getBoardSize();i++){
   if(checkValidMove[i]>0){
      KalahBoard temp=new KalahBoard(moveStones(n.game, currentPlayer, i));
      KalahBoard.Node xn=new KalahBoard.Node();
      xn.parent=n;
      xn.game=new KalahBoard(temp);
      xn.name=getName(currentPlayer, i);
      char startChar=n.name.charAt(0);
      if (xn.name.charAt(0)==startChar){
         xn.depth=n.depth;
      }
      else{
         xn.depth=n.depth+1;
      }
      if(temp.getGetAnotherTurn()){
         xn.max=n.max;
      }
      else{
         xn.max=!n.max;
      }
      if(xn.max){
         xn.eval=Double.NEGATIVE_INFINITY;
      }
      else{
         xn.eval=Double.POSITIVE_INFINITY;
      }
      xn.alpha=n.alpha;
      xn.beta=n.beta;
      n.children.add(xn);
```

```
                if(currentPlayer==1)
                    Statistics.noofNodes1++;
                else Statistics.noofNodes2++;
            }
        }
    }
}

/* Convert Double.MAX_VALUE and Double.MIN_VALUE
 * as Infinity and Negative Infinity to print them
 * into traverse log.
 */
public String evalToString(double eval){
    if(Double.isInfinite(eval)){
        return Double.toString(eval);
    }
    else{
        return Integer.toString((int)eval);
    }
}

/*
 * Get next best move from current game state which is represented in root Node.
 */
public KalahBoard nextState(KalahBoard.Node root) throws IOException{
    ListIterator<KalahBoard.Node> listIterator=root.children.listIterator();
    while(listIterator.hasNext()){
        KalahBoard.Node temp=listIterator.next();
        System.out.println("Expanding Player #"+temp.game.getPlayer()+" Node #"+temp.name+" :: eval
"+temp.eval);
```

```
        if(temp.eval==root.eval){
            KalahBoard move=null;
            if(temp.game.getGetAnotherTurn()){
                System.out.println("--------------------------");
                System.out.println("Optimum choice : Node "+temp.name);
                printBoard(temp.game);
                System.out.println("Free move!");
                KalahBoard.Node x=getNextState(temp);
                move=new KalahBoard(x.game);
                System.out.println("--------------------------");
                System.out.println("Optimum choice : Node "+x.name);
                printBoard(move);

            }
            else{
                move=new KalahBoard(temp.game);
                System.out.println("--------------------------");
                System.out.println("Optimum choice : Node "+temp.name);
                printBoard(move);
            }
            return move;
            //break;
        }
    }
    return null;
}

/*
 * Helper function to get next State (Best valid move)
 */
```

```
    public KalahBoard.Node getNextState(KalahBoard.Node n) throws IOException{
        KalahBoard.Node nextMove=null;
        //System.out.println("Free move!");
        ListIterator<KalahBoard.Node> listIterator=n.children.listIterator();
        while(listIterator.hasNext()){
            KalahBoard.Node temp=listIterator.next();
            System.out.println("Expanding Player #"+temp.game.getPlayer()+" Node #"+temp.name+" :: eval
"+temp.eval);
            if(temp.eval==n.eval){
                if(temp.game.getGetAnotherTurn()){
                    System.out.println("-------------------------");
                    System.out.println("Optimum choice : Node "+temp.name);
                    printBoard(temp.game);
                    System.out.println("Free move!");
                    nextMove=getNextState(temp);
                }
                else{
                    nextMove=new KalahBoard.Node(temp);
                    break;
                }

            }
        }
        //System.out.println("Optimum choice : Node "+nextMove.name);
        //printBoard(nextMove.game);
        return nextMove;
    }

    /*
     * Print next state of game in required format.
```

```
*/
public void printBoard(KalahBoard move) throws IOException{
    System.out.println("Board after playing the best move by Player #" + move.getPlayer());

    int[] temp1=new int[move.getBoardSize()];
    int[] temp2=new int[move.getBoardSize()];
    System.arraycopy(move.getPlayer1_Board(), 0, temp1, 0, move.getBoardSize());
    System.arraycopy(move.getPlayer2_Board(), 0, temp2, 0, move.getBoardSize());
    String s1="";
    String s2="";
    System.out.println("-----------------------------------");
    System.out.print("PLAYER-1  ");
    for(int i=0;i<move.getBoardSize();i++){
        System.out.print(temp1[i] + " ");
    }
    System.out.print("   " +move.getPlayer1_Kalah());
    System.out.println();
    System.out.print("PLAYER-2  ");
    for(int i=0;i<move.getBoardSize();i++){
        System.out.print(temp2[i] + " ");
    }
    System.out.print("   " +move.getPlayer2_Kalah());
    System.out.println();
    System.out.println("-----------------------------------");
    System.out.println();
    for(int i=0;i<move.getBoardSize();i++){
        s1+=Integer.toString(temp1[i])+" ";
        s2+=Integer.toString(temp2[i])+" ";
    }
}
```

```
    /*
     * Check if all the holes on player 1's side are empty
     * which is one of the condition for end of game.
    */
    public boolean Player1_SideEmpty(int[] boardPlayer1){
        for(int i=0;i<boardPlayer1.length;i++){
            if(boardPlayer1[i]>0)
                return false;
        }
        return true;
    }

    /*
     * Check if all the holes on player 2's side are empty
     * which is one of the condition for end of game.
    */
    public boolean Player2_SideEmpty(int[] boardPlayer2){
        for(int i=0;i<boardPlayer2.length;i++){
            if(boardPlayer2[i]>0)
                return false;
        }
        return true;
    }
}
```

Kalahboard.java

```
package kalah1;

//Represent KalahBoard state

import java.util.LinkedList;
import java.util.List;
import java.util.ListIterator;

public class KalahBoard {
    private int player;
    private int[] board1,board2;
    private int kalah1, kalah2;
    private int boardSize;
    private boolean getAnotherTurn;

    //Default Constructor
    public KalahBoard(){

    }

    //Constructor for board creation
    public KalahBoard(int player, int[] board2, int[] board1, int kalah2, int kalah1){
        this.player=player;
        this.boardSize=board2.length;
        this.board2=new int[boardSize];
        this.board1=new int[boardSize];
        for(int i=0;i<boardSize;i++){
            this.board2[i]=board2[i];
            this.board1[i]=board1[i];
```

```
    }
    this.kalah2=kalah2;
    this.kalah1=kalah1;
}

//Copy Constructor
public KalahBoard(KalahBoard x){
    player=x.getPlayer();
    boardSize=x.getBoardSize();
    this.board2=new int[boardSize];
    this.board1=new int[boardSize];
    //Copies source array from position 0 to destination array from position 0
    System.arraycopy(x.getPlayer2_Board(), 0, this.board2, 0, boardSize);
    System.arraycopy(x.getPlayer1_Board(), 0, this.board1, 0, boardSize);
    kalah2=x.getPlayer2_Kalah();
    kalah1=x.getPlayer1_Kalah();
    getAnotherTurn=x.getGetAnotherTurn();
}

//Returns the board size
public int getBoardSize(){
    return boardSize;
}
//Sets the boardSize
public void setBoardSize(int boardSize){
    this.boardSize=boardSize;
}
//Returns the current player
public int getPlayer(){
    return player;
```

```
}
//Sets the next player
public void setPlayer(int player){
    this.player=player;
}
//Returns the player1's board
public int[] getPlayer1_Board(){
    return board1;
}
//Copies the current board of Player1 to this.board1
public void setPlayer1_Board(int[] board1){
    System.arraycopy(board1, 0, this.board1, 0, boardSize);
}
//Gets Player2's board
public int[] getPlayer2_Board(){
    return board2;
}
//Copies the current board of Player2 to this.board2
public void setPlayer2_Board(int[] board2){
    System.arraycopy(board2, 0, this.board2, 0, boardSize);
}
//Gets Player1's Kalah total
public int getPlayer1_Kalah(){
    return kalah1;
}
//Sets the current kalah to Player1's Kalah
public void setPlayer1_Kalah(int kalah1){
    this.kalah1=kalah1;
}
//Gets Player2's Kalah total
```

```
public int getPlayer2_Kalah(){
    return kalah2;
}
//Sets current kalah total to player2's kalah
public void setPlayer2_Kalah(int kalah2){
    this.kalah2=kalah2;
}

public boolean getGetAnotherTurn(){
    return getAnotherTurn;
}

public void setGetAnotherTurn(boolean getAnotherTurn){
    this.getAnotherTurn=getAnotherTurn;
}

//Node class to represent game state
public static class Node {

    String name;
    KalahBoard game = null;
    double eval;
    double alpha;
    double beta;
    int depth;
    boolean max;
    Node parent;
    List<Node> children;

    //Default Constructor
```

```
    public Node() {
        super();
        this.alpha = Double.NEGATIVE_INFINITY;
        this.beta = Double.POSITIVE_INFINITY;
        this.children = new LinkedList<Node>();
    }

    //Constructor for Minimax method
    public Node(String name, KalahBoard game, double eval, int depth, boolean max, Node parent) {
        super();
        this.name = name;
        this.game = new KalahBoard(game);
        this.eval = eval;
        this.alpha = Double.NEGATIVE_INFINITY;
        this.beta = Double.POSITIVE_INFINITY;
        this.depth = depth;
        this.max = max;
        this.parent = parent;
        this.children = new LinkedList<Node>();
    }

    //Constructor for alpha beta method
    public Node(String name, KalahBoard game, double eval, double alpha, double beta, int depth,
boolean max, Node parent) {
        super();
        this.name = name;
        this.game = new KalahBoard(game);
        this.eval = eval;
        this.alpha = alpha;
        this.beta = beta;
```

```
            this.depth = depth;
            this.max = max;
            this.parent = parent;
            this.children = new LinkedList<Node>();
        }

        //Copy Constructor
        public Node(Node x) {
            super();
            name = x.name;
            game = new KalahBoard(x.game);
            eval = x.eval;
            alpha = x.alpha;
            beta = x.beta;
            depth = x.depth;
            max = x.max;
            parent = x.parent;
            children = new LinkedList<Node>();
            ListIterator<Node> listIterator = x.children.listIterator();
            while (listIterator.hasNext()) {
                children.add(listIterator.next());
            }
        }
    }
}
```

RESULT/OUTPUT

Output1:

At cutting depth 1, the following is the output for minmax algorithm for player 1 using evaluation function 1 and player2 using evaluation function 2.

run:

Welcome to KALAH Game

Algorithms

---

1. Minimax

2. AlphaBetaPruning

Choose algorithm for your 1st player :

1

Choose algorithm for your 2nd player :

1

Evaluation Functions

---

1. (#Stones in 1st player Kalah + #Stones in 1st player pits) - (#Stones in 2nd player Kalah + #Stones in 2nd player pits)

2. (number of my empty pits) - (number of opponent empty pits)

Standard Evaluation- (#Stones in 1st player Kalah - #Stones in 2nd player Kalah)

Choose evaluation function for 1st player :

1

Choose evaluation function for 2nd player :

2

1. Player #1

2. Player #2

Who goes first? :

2

Expanding Player #2 Node #B1 :: eval 0.0

Optimum choice : Node B1

Board after playing the best move by Player #2

PLAYER-1 7 7 7 7 7 6   0

PLAYER-2 0 6 6 6 6 6   1

Expanding Player #1 Node #A1 :: eval 10.0

Optimum choice : Node A1

Board after playing the best move by Player #1

PLAYER-1 0 8 8 8 8 7   1

PLAYER-2 0 6 6 6 6 7   1

Expanding Player #2 Node #B2 :: eval -1.0

Optimum choice : Node B2

Board after playing the best move by Player #2

PLAYER-1 1 9 9 9 8 7   1

PLAYER-2 1 0 6 6 6 7   2

Expanding Player #1 Node #A1 :: eval 18.0

Optimum choice : Node A1

Board after playing the best move by Player #1

PLAYER-1 0 10 9 9 8 7 1

PLAYER-2 1 0 6 6 6 7 2

Expanding Player #2 Node #B1 :: eval -1.0

Optimum choice : Node B1

Board after playing the best move by Player #2

PLAYER-1 0 10 9 9 8 7 1

PLAYER-2 0 0 6 6 6 7 3

Free move!

Expanding Player #2 Node #B3 :: eval -1.0

Optimum choice : Node B3

Board after playing the best move by Player #2

PLAYER-1 1 11 10 9 8 7 1

PLAYER-2 1 1 0 6 6 7 4

Expanding Player #1 Node #A1 :: eval 22.0

Optimum choice : Node A1

Board after playing the best move by Player #1

PLAYER-1 0 12 10 9 8 7 1

PLAYER-2 1 1 0 6 6 7 4

Expanding Player #2 Node #B1 :: eval -1.0

Optimum choice : Node B1

Board after playing the best move by Player #2

PLAYER-1 0 12 10 9 8 7 1

PLAYER-2 0 1 0 6 6 7 5

Free move!

Expanding Player #2 Node #B2 :: eval -2.0

Expanding Player #2 Node #B4 :: eval -1.0

Optimum choice : Node B4

Board after playing the best move by Player #2

PLAYER-1 1 13 10 9 8 7 1

PLAYER-2 1 2 1 0 6 7 6

Expanding Player #1 Node #A1 :: eval -2.0

Expanding Player #1 Node #A2 :: eval 20.0

Optimum choice : Node A2

Board after playing the best move by Player #1

PLAYER-1 2 0 11 10 9 8 6

PLAYER-2 2 0 2 1 7 8 6

Expanding Player #2 Node #B1 :: eval -2.0

Expanding Player #2 Node #B3 :: eval -2.0

Expanding Player #2 Node #B4 :: eval -2.0

Expanding Player #2 Node #B5 :: eval -1.0

Optimum choice : Node B5

Board after playing the best move by Player #2

PLAYER-1 3 1 11 10 9 8 6

PLAYER-2 3 1 3 2 0 8 7

Expanding Player #1 Node #A1 :: eval 20.0

Expanding Player #1 Node #A2 :: eval 24.0

Optimum choice : Node A2

Board after playing the best move by Player #1

PLAYER-1 3 0 12 10 9 8 6

PLAYER-2 3 1 3 2 0 8 7

Expanding Player #2 Node #B1 :: eval -1.0

Optimum choice : Node B1

Board after playing the best move by Player #2

PLAYER-1 4 1 12 10 9 8   6

PLAYER-2 0 1 3 2 0 8   8

Expanding Player #1 Node #A1 :: eval 24.0

Optimum choice : Node A1

Board after playing the best move by Player #1

PLAYER-1 0 2 13 11 10 8   6

PLAYER-2 0 1 3 2 0 8   8

Expanding Player #2 Node #B2 :: eval -1.0

Optimum choice : Node B2

Board after playing the best move by Player #2

PLAYER-1 0 2 13 11 10 8   6

PLAYER-2 0 0 3 2 0 8   9

Expanding Player #1 Node #A2 :: eval 28.0

Optimum choice : Node A2

Board after playing the best move by Player #1

PLAYER-1 0 0 14 12 10 8 6

PLAYER-2 0 0 3 2 0 8 9

Expanding Player #2 Node #B3 :: eval -1.0

Optimum choice : Node B3

Board after playing the best move by Player #2

PLAYER-1 0 0 14 12 10 8 6

PLAYER-2 1 1 0 2 0 8 10

Free move!

Expanding Player #2 Node #B1 :: eval -1.0

Optimum choice : Node B1

Board after playing the best move by Player #2

PLAYER-1 0 0 14 12 10 8 6

PLAYER-2 0 1 0 2 0 8 11

Free move!

Expanding Player #2 Node #B2 :: eval -2.0

Expanding Player #2 Node #B4 :: eval -2.0

Expanding Player #2 Node #B6 :: eval -1.0

Expanding Player #2 Node #B2 :: eval -2.0

Expanding Player #2 Node #B4 :: eval -2.0

Expanding Player #2 Node #B6 :: eval -1.0

Optimum choice : Node B6

Board after playing the best move by Player #2

PLAYER-1 1 1 14 12 10 8 6

PLAYER-2 2 2 1 3 1 0 11

Expanding Player #1 Node #A1 :: eval 28.0

Expanding Player #1 Node #A2 :: eval 32.0

Optimum choice : Node A2

Board after playing the best move by Player #1

PLAYER-1 1 0 15 12 10 8 6

PLAYER-2 2 2 1 3 1 0 11

Expanding Player #2 Node #B1 :: eval -2.0

Expanding Player #2 Node #B2 :: eval -1.0

Optimum choice : Node B2

Board after playing the best move by Player #2

PLAYER-1 1 0 15 12 10 8 6

PLAYER-2 3 0 1 3 1 0 12

Free move!

Expanding Player #2 Node #B1 :: eval -1.0

Optimum choice : Node B1

Board after playing the best move by Player #2

PLAYER-1 2 1 15 12 10 8   6

PLAYER-2 0 0 1 3 1 0   13

Expanding Player #1 Node #A1 :: eval 32.0

Optimum choice : Node A1

Board after playing the best move by Player #1

PLAYER-1 0 2 16 12 10 8   6

PLAYER-2 0 0 1 3 1 0   13

Expanding Player #2 Node #B3 :: eval -2.0

Expanding Player #2 Node #B4 :: eval -1.0

Optimum choice : Node B4

Board after playing the best move by Player #2

PLAYER-1 0 2 16 12 10 8   6

PLAYER-2 0 1 2 0 1 0   14

Expanding Player #1 Node #A2 :: eval 10.0

Expanding Player #1 Node #A3 :: eval 22.0

Optimum choice : Node A3

Board after playing the best move by Player #1

PLAYER-1 1 3 1 14 12 10 7

PLAYER-2 1 2 3 1 2 1 14

Expanding Player #2 Node #B1 :: eval -1.0

Optimum choice : Node B1

Board after playing the best move by Player #2

PLAYER-1 1 3 1 14 12 10 7

PLAYER-2 0 2 3 1 2 1 15

Free move!

Expanding Player #2 Node #B2 :: eval -1.0

Optimum choice : Node B2

Board after playing the best move by Player #2

PLAYER-1 1 3 1 14 12 10 7

PLAYER-2 1 0 3 1 2 1 16

Free move!

Expanding Player #2 Node #B1 :: eval -1.0

Optimum choice : Node B1

Board after playing the best move by Player #2

PLAYER-1 1 3 1 14 12 10   7

PLAYER-2 0 0 3 1 2 1   17

Free move!

Expanding Player #2 Node #B3 :: eval -1.0

Optimum choice : Node B3

Board after playing the best move by Player #2

PLAYER-1 1 3 1 14 12 10   7

PLAYER-2 1 1 0 1 2 1   18

Free move!

Expanding Player #2 Node #B1 :: eval -1.0

Optimum choice : Node B1

Board after playing the best move by Player #2

PLAYER-1 1 3 1 14 12 10   7

PLAYER-2 0 1 0 1 2 1   19

Free move!

Expanding Player #2 Node #B2 :: eval -1.0

Expanding Player #2 Node #B2 :: eval -1.0

Expanding Player #2 Node #B4 :: eval -1.0

Expanding Player #2 Node #B3 :: eval -1.0

Optimum choice : Node B3

Board after playing the best move by Player #2

PLAYER-1 1 3 1 14 12 10   7

PLAYER-2 2 1 0 1 2 1   17

Free move!

Expanding Player #2 Node #B1 :: eval -1.0

Expanding Player #2 Node #B4 :: eval -1.0

Expanding Player #2 Node #B3 :: eval -1.0

Optimum choice : Node B3

Board after playing the best move by Player #2

PLAYER-1 1 3 1 14 12 10   7

PLAYER-2 1 3 0 1 2 1   16

Free move!

Expanding Player #2 Node #B1 :: eval -1.0

Optimum choice : Node B1

Board after playing the best move by Player #2

PLAYER-1 1 3 1 14 12 10   7

PLAYER-2 0 3 0 1 2 1   17

Free move!

Expanding Player #2 Node #B2 :: eval -1.0

Expanding Player #2 Node #B2 :: eval -1.0

Expanding Player #2 Node #B4 :: eval -1.0

Optimum choice : Node B4

Board after playing the best move by Player #2

PLAYER-1 1 3 1 14 12 10   7

PLAYER-2 0 2 4 0 2 1   15

Expanding Player #1 Node #A1 :: eval 24.0

Optimum choice : Node A1

Board after playing the best move by Player #1

PLAYER-1 0 4 1 14 12 10   7

PLAYER-2 0 2 4 0 2 1   15

Expanding Player #2 Node #B2 :: eval -1.0

Optimum choice : Node B2

Board after playing the best move by Player #2

PLAYER-1 0 4 1 14 12 10   7

PLAYER-2 1 0 4 0 2 1   16

Free move!

Expanding Player #2 Node #B1 :: eval -1.0

Optimum choice : Node B1

Board after playing the best move by Player #2

PLAYER-1 0 4 1 14 12 10 7

PLAYER-2 0 0 4 0 2 1 17

Free move!

Expanding Player #2 Node #B3 :: eval -1.0

Expanding Player #2 Node #B3 :: eval -1.0

Optimum choice : Node B3

Board after playing the best move by Player #2

PLAYER-1 1 4 1 14 12 10 7

PLAYER-2 2 1 0 0 2 1 17

Expanding Player #1 Node #A1 :: eval 24.0

Expanding Player #1 Node #A2 :: eval 22.0

Expanding Player #1 Node #A3 :: eval 26.0

Optimum choice : Node A3

Board after playing the best move by Player #1

PLAYER-1 1 4 0 15 12 10 7

PLAYER-2 2 1 0 0 2 1 17

Expanding Player #2 Node #B1 :: eval -1.0

Optimum choice : Node B1

Board after playing the best move by Player #2

PLAYER-1 2 4 0 15 12 10 7

PLAYER-2 0 1 0 0 2 1 18

Expanding Player #1 Node #A1 :: eval 28.0

Optimum choice : Node A1

Board after playing the best move by Player #1

PLAYER-1 0 5 0 15 12 10 8

PLAYER-2 0 1 0 0 2 1 18

Expanding Player #2 Node #B2 :: eval -1.0

Optimum choice : Node B2

Board after playing the best move by Player #2

PLAYER-1 0 5 0 15 12 10 8

PLAYER-2 0 0 0 0 2 1 19

Expanding Player #1 Node #A2 :: eval 28.0

Optimum choice : Node A2

Board after playing the best move by Player #1

PLAYER-1 0 0 1 16 13 11 9

PLAYER-2 0 0 0 0 2 1 19

Free move!

Expanding Player #1 Node #A3 :: eval 28.0

Optimum choice : Node A3

Board after playing the best move by Player #1

PLAYER-1 0 0 0 17 13 11 9

PLAYER-2 0 0 0 0 2 1 19

Expanding Player #2 Node #B5 :: eval -1.0

Optimum choice : Node B5

Board after playing the best move by Player #2

PLAYER-1 0 0 0 17 13 11 9

PLAYER-2 0 0 0 1 0 1 20

Expanding Player #1 Node #A4 :: eval 12.0

Expanding Player #1 Node #A5 :: eval 16.0

Optimum choice : Node A5

Board after playing the best move by Player #1

PLAYER-1 1 1 1 18 0 12   12

PLAYER-2 1 1 1 2 0 2   20

Expanding Player #2 Node #B1 :: eval -1.0

Optimum choice : Node B1

Board after playing the best move by Player #2

PLAYER-1 1 1 1 18 0 12   12

PLAYER-2 0 1 1 2 0 2   21

Free move!

Expanding Player #2 Node #B2 :: eval -1.0

Optimum choice : Node B2

Board after playing the best move by Player #2

PLAYER-1 0 1 1 18 0 12   12

PLAYER-2 0 0 1 2 0 2   23

Expanding Player #1 Node #A2 :: eval 16.0

Optimum choice : Node A2

Board after playing the best move by Player #1

PLAYER-1 0 0 2 18 0 12   12

PLAYER-2 0 0 1 2 0 2   23

Expanding Player #2 Node #B3 :: eval -1.0

Optimum choice : Node B3

Board after playing the best move by Player #2

PLAYER-1 0 0 2 18 0 12   12

PLAYER-2 0 0 0 2 0 2   24

Expanding Player #1 Node #A3 :: eval 16.0

Optimum choice : Node A3

Board after playing the best move by Player #1

PLAYER-1 0 0 0 19 0 12   13

PLAYER-2 0 0 0 2 0 2   24

Expanding Player #2 Node #B4 :: eval -1.0

Optimum choice : Node B4

Board after playing the best move by Player #2

PLAYER-1 0 0 0 19 0 12   13

PLAYER-2 0 0 1 0 0 2   25

Expanding Player #1 Node #A4 :: eval -4.0

Expanding Player #1 Node #A6 :: eval 4.0

Optimum choice : Node A6

Board after playing the best move by Player #1

PLAYER-1 1 1 1 20 0 0   16

PLAYER-2 1 1 2 1 0 3   25

Expanding Player #2 Node #B1 :: eval 0.0

Optimum choice : Node B1

Board after playing the best move by Player #2

PLAYER-1 1 1 1 20 0 0   16

PLAYER-2 0 1 2 1 0 3   26

Free move!

Expanding Player #2 Node #B2 :: eval -1.0

Expanding Player #2 Node #B3 :: eval -1.0

Expanding Player #2 Node #B4 :: eval 0.0

Optimum choice : Node B4

Board after playing the best move by Player #2

PLAYER-1 1 1 1 20 0 0 16

PLAYER-2 0 1 3 0 0 3 26

Expanding Player #1 Node #A1 :: eval 4.0

Optimum choice : Node A1

Board after playing the best move by Player #1

PLAYER-1 0 2 1 20 0 0 16

PLAYER-2 0 1 3 0 0 3 26

Expanding Player #2 Node #B2 :: eval 0.0

Optimum choice : Node B2

Board after playing the best move by Player #2

PLAYER-1 0 2 1 20 0 0 16

PLAYER-2 0 0 3 0 0 3 27

Expanding Player #1 Node #A2 :: eval 2.0

Expanding Player #1 Node #A3 :: eval 6.0

Optimum choice : Node A3

Board after playing the best move by Player #1

PLAYER-1 0 2 0 21 0 0 16

PLAYER-2 0 0 3 0 0 3   27

Expanding Player #2 Node #B3 :: eval 0.0

Optimum choice : Node B3

Board after playing the best move by Player #2

PLAYER-1 0 2 0 21 0 0   16

PLAYER-2 1 1 0 0 0 3   28

Free move!

Expanding Player #2 Node #B1 :: eval 0.0

Optimum choice : Node B1

Board after playing the best move by Player #2

PLAYER-1 0 2 0 21 0 0   16

PLAYER-2 0 1 0 0 0 3   29

Free move!

Expanding Player #2 Node #B2 :: eval 0.0

Expanding Player #2 Node #B2 :: eval 0.0

Optimum choice : Node B2

Board after playing the best move by Player #2

PLAYER-1 0 2 0 21 0 0   16

PLAYER-2 2 0 0 0 0 3   28

Expanding Player #1 Node #A2 :: eval 4.0

Optimum choice : Node A2

Board after playing the best move by Player #1

PLAYER-1 0 0 1 22 0 0   16

PLAYER-2 2 0 0 0 0 3   28

Expanding Player #2 Node #B1 :: eval 0.0

Optimum choice : Node B1

Board after playing the best move by Player #2

PLAYER-1 1 0 1 22 0 0   16

PLAYER-2 0 0 0 0 0 3   29

Expanding Player #1 Node #A1 :: eval 6.0

Expanding Player #1 Node #A3 :: eval 8.0

Optimum choice : Node A3

Board after playing the best move by Player #1

PLAYER-1 1 0 0 23 0 0   16

PLAYER-2 0 0 0 0 0 3   29

Expanding Player #2 Node #B6 :: eval -1.0

Optimum choice : Node B6

Board after playing the best move by Player #2

PLAYER-1 1 0 0 23 0 0 16

PLAYER-2 0 0 0 1 1 0 30

Expanding Player #1 Node #A1 :: eval 8.0

Optimum choice : Node A1

Board after playing the best move by Player #1

PLAYER-1 0 0 0 23 0 0 17

PLAYER-2 0 0 0 1 1 0 30

Expanding Player #2 Node #B4 :: eval 0.0

Optimum choice : Node B4

Board after playing the best move by Player #2

PLAYER-1 0 0 0 23 0 0 17

PLAYER-2 0 0 0 0 1 0 31

Expanding Player #1 Node #A4 :: eval -18.0

Optimum choice : Node A4

Board after playing the best move by Player #1

PLAYER-1 2 1 1 1 2 2   19

PLAYER-2 2 2 2 2 3 2   31

Expanding Player #2 Node #B1 :: eval -1.0

Expanding Player #2 Node #B2 :: eval 0.0

Optimum choice : Node B2

Board after playing the best move by Player #2

PLAYER-1 2 1 1 1 2 2   19

PLAYER-2 3 0 2 2 3 2   32

Free move!

Expanding Player #2 Node #B1 :: eval 0.0

Optimum choice : Node B1

Board after playing the best move by Player #2

PLAYER-1 3 2 1 1 2 2   19

PLAYER-2 0 0 2 2 3 2   33

Expanding Player #1 Node #A1 :: eval -18.0

Expanding Player #1 Node #A2 :: eval -18.0

Expanding Player #1 Node #A3 :: eval -18.0

Expanding Player #1 Node #A4 :: eval -18.0

Expanding Player #1 Node #A5 :: eval -12.0

Optimum choice : Node A5

Board after playing the best move by Player #1

PLAYER-1 3 2 1 1 0 3   20

PLAYER-2 0 0 2 2 3 2   33

Free move!

Expanding Player #1 Node #A1 :: eval -18.0

Expanding Player #1 Node #A2 :: eval -18.0

Expanding Player #1 Node #A3 :: eval -18.0

Expanding Player #1 Node #A4 :: eval -12.0

Optimum choice : Node A4

Board after playing the best move by Player #1

PLAYER-1 3 2 1 0 0 3   24

PLAYER-2 0 0 2 2 0 2   33

Expanding Player #2 Node #B3 :: eval -2.0

Expanding Player #2 Node #B4 :: eval -1.0

Optimum choice : Node B4

Board after playing the best move by Player #2

PLAYER-1 3 0 1 0 0 3   24

PLAYER-2 0 0 3 0 0 2   36

Expanding Player #1 Node #A1 :: eval -10.0

Optimum choice : Node A1

Board after playing the best move by Player #1

PLAYER-1 0 1 2 0 0 3   25

PLAYER-2 0 0 3 0 0 2   36

Expanding Player #2 Node #B3 :: eval 0.0

Optimum choice : Node B3

Board after playing the best move by Player #2

PLAYER-1 0 1 2 0 0 3   25

PLAYER-2 1 1 0 0 0 2   37

Free move!

Expanding Player #2 Node #B1 :: eval 0.0

Optimum choice : Node B1

Board after playing the best move by Player #2

PLAYER-1 0 1 2 0 0 3   25

PLAYER-2 0 1 0 0 0 2   38

Free move!

Expanding Player #2 Node #B2 :: eval 0.0

Expanding Player #2 Node #B2 :: eval -1.0

Expanding Player #2 Node #B6 :: eval -2.0

Optimum choice : Node B2

Board after playing the best move by Player #2

PLAYER-1 0 1 2 0 0 3   25

PLAYER-2 0 0 0 0 0 2   39

Expanding Player #1 Node #A2 :: eval -10.0

Optimum choice : Node A2

Board after playing the best move by Player #1

PLAYER-1 0 0 3 0 0 3   25

PLAYER-2 0 0 0 0 0 2   39

Expanding Player #2 Node #B6 :: eval -1.0

Optimum choice : Node B6

Board after playing the best move by Player #2

PLAYER-1 0 0 3 0 0 3   25

PLAYER-2 0 0 0 0 1 0   40

Expanding Player #1 Node #A3 :: eval -12.0

Optimum choice : Node A3

Board after playing the best move by Player #1

PLAYER-1 0 0 0 1 1 4   25

PLAYER-2 0 0 0 0 1 0   40

Expanding Player #2 Node #B5 :: eval Infinity

Optimum choice : Node B5

Board after playing the best move by Player #2

PLAYER-1 0 0 0 0 0 0   30

PLAYER-2 0 0 0 0 0 0   42

PLAYER - 2 WON THE GAME

Statistics

PLAYER-1

No of Nodes Expanded by Player 1 is 426

No of Nodes Generated by Player 1 is 1229

Memory Used : 4916Bytes

PLAYER-2

No of Nodes Expanded by Player 2 is 288

No of Nodes Generated by Player 2 is 1962

Memory Used : 7848Bytes

BUILD SUCCESSFUL (total time: 8 seconds)

ANALYSIS OF THE RESULT

MINMAX Vs MINMAX

For a cutting depth of 1

Combinations	First Player	Players	Algo selection	Evaluation function	Nodes generated	Nodes Expanded	Win/lose	Execution time	Memory used
Game 1	Player1	Player1	Minmax	Eval1	70	14	Lose	6 sec	280Bytes
		Player2	Minmax	Eval2	71	19	win		284Bytes
Game2	Player1	Player1	Minmax	Eval2	95	24	win	5 sec	380Bytes
		Player2	Minmax	standard	74	19	lose		296Bytes
Game 3	Player1	Player1	Minmax	standard	47	14	Win	6 sec	188Bytes
		Player2	Minmax	Eval1	56	12	Lose		224Bytes

Combinations	First Player	Players	Algo selection	Evaluation function	Nodes generated	Nodes Expanded	Win/lose	Execution time	Memory used
Game 1	Player2	Player1	Minmax	Eval1	158	35	Lose	5 sec	632Bytes
		Player2	Minmax	Eval2	140	36	win		560Bytes
Game2	Player2	Player1	Minmax	Eval2	77	19	lose	9 sec	308Bytes
		Player2	Minmax	standard	100	21	win		400Bytes
Game 3	Player2	Player1	Minmax	standard	74	20	Win	6 sec	296Bytes
		Player2	Minmax	Eval1	68	17	Lose		272Bytes

- According to analysis we can say that the combination of Minmax algorithm and Evaluation function1 always loses which is worst case.
- The best case for the combination of Minmax algorithm is evaluation function 2 if the player1 plays first else standard evaluation function.

For a cutting depth of 2

Combinations	First Player	Players	Algo selection	Evaluation function	Nodes generated	Nodes Expanded	Win/lose	Execution time	Memory used
Game 1	Player1	Player1	Minmax	Eval1	1131	354	Lose	23 sec	4524Bytes
		Player2	Minmax	Eval2	1614	265	win		6456Bytes
Game2	Player1	Player1	Minmax	Eval2	1061	313	lose	5 sec	4244Bytes
		Player2	Minmax	standard	1413	277	win		5652Bytes
Game 3	Player1	Player1	Minmax	standard	2791	415	Win	7 sec	11164
		Player2	Minmax	Eval1	2035	625	Lose		8140Bytes

Combinations	First Player	Players	Algo selection	Evaluation function	Nodes generated	Nodes Expanded	Win/lose	Execution time	Memory used
Game 1	Player2	Player1	Minmax	Eval1	1229	426	Lose	6 sec	4916Bytes
		Player2	Minmax	Eval2	1962	288	win		7848Bytes
Game2	Player2	Player1	Minmax	Eval2	1381	290	lose	12 sec	5524Bytes
		Player2	Minmax	standard	1340	310	win		5360Bytes
Game 3	Player2	Player1	Minmax	standard	1229	333	Win	5 sec	4916Bytes
		Player2	Minmax	Eval1	1594	245	Lose		6376Bytes

- According to analysis we can say that the combination of Minmax algorithm and Evaluation function1 always loses which is worst case.
- The best case for the combination of Minmax algorithm is standard evaluation function.

Alpha-Beta Pruning Vs Alpha-Beta Pruning

For a cutting depth of 1

Combinations	First Player	Players	Algo selection	Evaluation function	Nodes generated	Nodes Expanded	Win/lose	Execution time	Memory used
Game 1	Player1	Player1	Alphabeta	Eval1	70	14	Lose	13 sec	280Bytes
		Player2	Alphabeta	Eval2	71	19	win		284Bytes
Game2	Player1	Player1	Alphabeta	Eval2	95	24	win	8 sec	380Bytes
		Player2	Alphabeta	standard	74	19	lose		296Bytes
Game 3	Player1	Player1	Alphabeta	standard	47	14	Win	6 sec	188Bytes
		Player2	Alphabeta	Eval1	56	12	lose		224Bytes

Combinations	First Player	Players	Algo selection	Evaluation function	Nodes generated	Nodes Expanded	Win/lose	Execution time	Memory used
Game 1	Player2	Player1	Alphabeta	Eval1	158	35	Lose	8 sec	632Bytes
		Player2	Alphabeta	Eval2	140	36	win		560Bytes
Game2	Player2	Player1	Alphabeta	Eval2	77	19	lose	6 sec	308Bytes
		Player2	Alphabeta	standard	100	21	win		400Bytes
Game 3	Player2	Player1	Alphabeta	standard	74	20	Win	27 sec	296Bytes
		Player2	Alphabeta	Eval1	68	17	lose		272Bytes

- According to analysis we can say that the combination of Alphabeta algorithm and Evaluation function1 always loses which is worst case.
- The best case for the combination of Alphabeta algorithm is evaluation function 2 if the player1 plays first else standard evaluation function.

For a cutting depth of 2

Combinations	First Player	Players	Algo selection	Evaluation function	Nodes generated	Nodes Expanded	Win/lose	Execution time	Memory used
Game 1	Player1	Player1	Alphabeta	Eval1	998	272	Lose	4 sec	3992Bytes
		Player2	Alphabeta	Eval2	1248	237	win		4992Bytes
Game2	Player1	Player1	Alphabeta	Eval2	938	250	lose	8 sec	3752Bytes
		Player2	Alphabeta	standard	1138	243	win		4552Bytes
Game 3	Player1	Player1	Alphabeta	standard	1579	277	Win	8 sec	6316Bytes
		Player2	Alphabeta	Eval1	1334	350	lose		5336Bytes

Combinations	First Player	Players	Algo selection	Evaluation function	Nodes generated	Nodes Expanded	Win/lose	Execution time	Memory used
Game 1	Player2	Player1	Alphabeta	Eval1	1318	277	Lose	10 sec	5272Bytes
		Player2	Alphabeta	Eval2	1337	300	win		5348Bytes
Game2	Player2	Player1	Alphabeta	Eval2	1258	268	lose	5 sec	5032Bytes
		Player2	Alphabeta	standard	1284	266	win		5136Bytes
Game 3	Player2	Player1	Alphabeta	standard	870	219	Win	5 sec	3480Bytes
		Player2	Alphabeta	Eval1	1003	166	lose		4012Bytes

- According to analysis we can say that the combination of Alphabeta algorithm and Evaluation function1 always loses which is worst case.
- The best case for the combination of Alphabeta algorithm is standard evaluation function.

Alpha Beta Vs Minmax

- At cutting depth1 the number of nodes generated and expanded does not differ for both minmax and alpha-beta pruning algorithm. Hence comparing the combination of both at cutting depth 2

Combinations	First Player	Players	Algo selection	Evaluation function	Nodes generated	Nodes Expanded	Win/lose	Execution time	Memory used
Game 1	Player1	Player1	Alphabeta	Eval2	382	47	Lose	13 sec	1528Bytes
		Player2	Minmax	Eval1	262	75	win		1048Bytes
Game2	Player1	Player1	Alphabeta	Eval2	1061	250	lose	13 sec	4244Bytes
		Player2	Minmax	standard	1138	277	win		4552Bytes
Game 3	Player1	Player1	Alphabeta	standard	2791	291	Win	14 sec	11164
		Player2	Minmax	Eval1	1384	625	lose		5536Bytes

Combinations	First Player	Players	Algo selection	Evaluation function	Nodes generated	Nodes Expanded	Win/lose	Execution time	Memory used
Game 1	Player2	Player1	Alphabeta	Eval2	1227	193	Lose	11 sec	1528Bytes
		Player2	Minmax	Eval1	930	270	win		1048Bytes
Game2	Player2	Player1	Alphabeta	Eval2	1922	331	lose	8 sec	7688Bytes
		Player2	Minmax	standard	1461	430	win		5844Bytes
Game 3	Player2	Player1	Alphabeta	standard	1229	291	Win	5 sec	4916Bytes
		Player2	Minmax	Eval1	1003	245	lose		4012Bytes

CONCLUSION

- With regard to the above observations we can see that the player which chooses standard evaluation always win irrelevant to the Algorithm.
- So, the best win possibilities are always with standard evaluation function.
- Now, comparing eval function1 and function 2, We can observe that in some cases eval1 wins over eval2 and vice versa in rest other, like the sample shown below

Combinations	**First Player**	**Players**	**Algo selection**	**Evaluation function**	**Nodes generated**	**Nodes Expanded**	**Win/lose**	**Execution time**	**Memory used**
Game 1	Player1	Player1	Minmax	Eval2	382	47	Lose	9 sec	1528Bytes
		Player2	Minmax	Eval1	262	75	win		1048Bytes
Game2	Player1	Player1	Minmax	Eval1	1131	354	Lose	6 sec	4524Bytes
		Player2	Minmax	Eval2	1614	265	win		6456Bytes

- Hence according to the observations, regardless of who ever might be the first player and algorithm, in the combination of (player1 as Eval2) vs (player2 as Eval1) always player2+eval1 wins
- In the combination of (player1 as Eval1) vs (player2 as Eval2) always player2+eval2 wins

Best choice to win game

1. (player1/player2) + Standard evaluation function + (Minmax/Alphabeta pruning)
2. Player2 + (Eval1/Eval2) +(Minmax/Alphabeta pruning)

Worst case to lose game

1. Player1 + Eval1 + (Minmax/Alphabeta pruning)

REFERENCES

- https://www.geeksforgeeks.org/minimax-algorithm-in-game-theory-set-1-introduction/
- https://www.researchgate.net/publication/2911672_Solving_Kalah
- https://www.thesprucecrafts.com/how-to-win-at-mancala-basic-strategy-411832
- https://en.wikipedia.org/wiki/Mancala
- https://en.wikipedia.org/wiki/Minimax