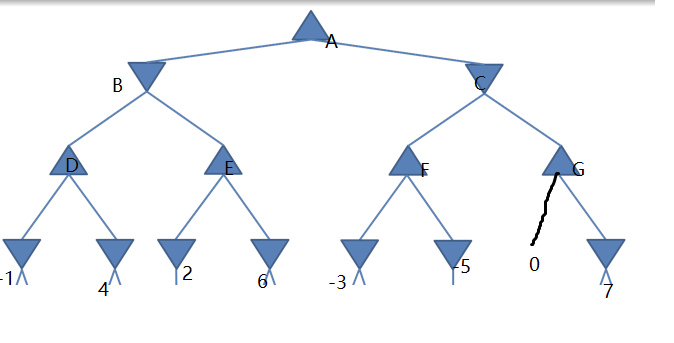
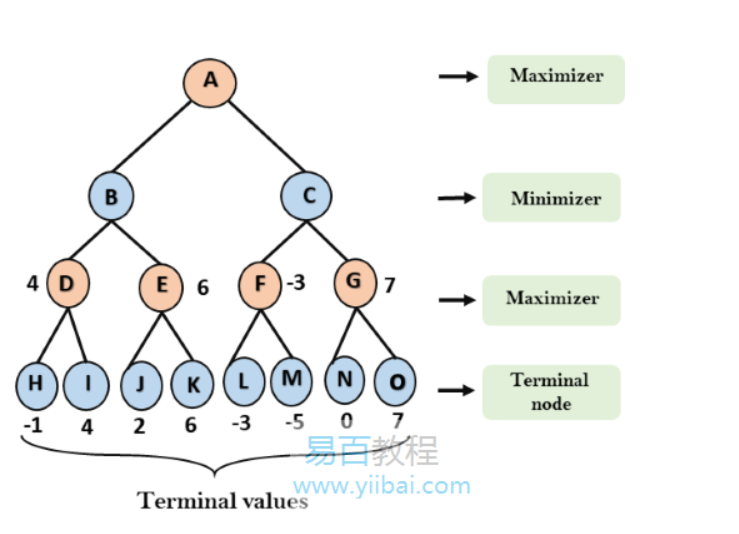
**AI课上代码实现 第三次试验 硬件一班 王倩倩 171491121**

**3 最大最小算法**

最大最小算法常用于零和博弈游戏中，零和博弈指参与博弈的各方，在严格竞争下，一方的收益必然意味着另一方的损失，博弈各方的收益和损失相加总和永远为“零”，双方不存在合作的可能。博弈游戏，目的是寻找最优的方案使得自己能够利益最大化。

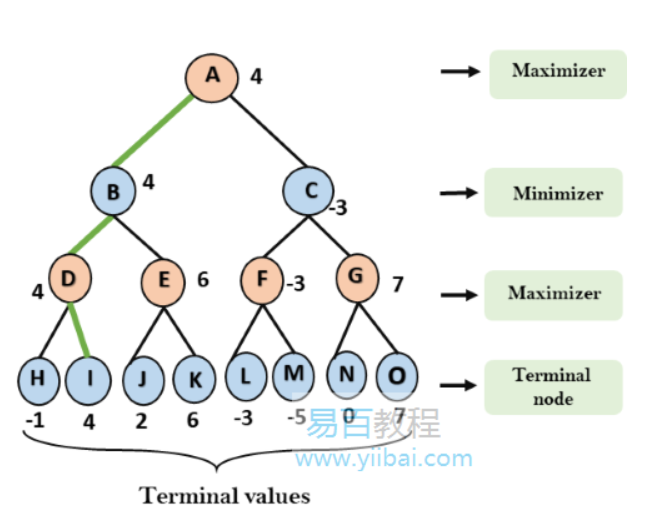


**第1步：** 在第一步中，算法生成整个游戏树并应用效用函数来获取终端状态的效用值。在下面的树形图中，下面来看看A是树的初始状态。

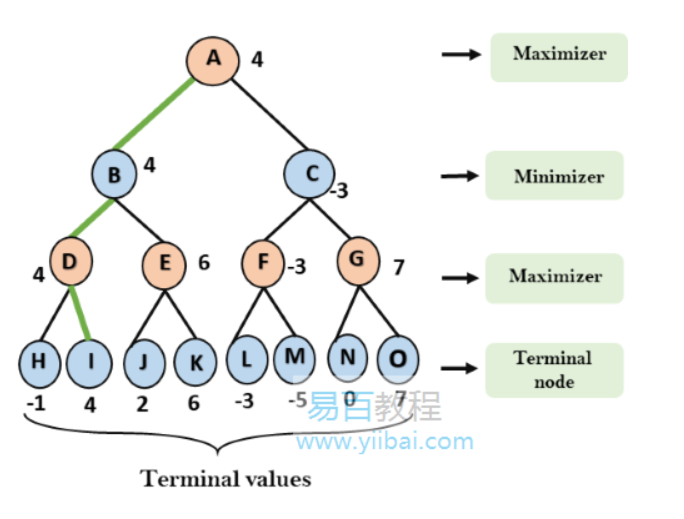


**第2步**：

* 对于节点 D max(-1,- -∞) => max(-1,4)= 4
* 对于节点 E max(2, -∞) => max(2, 6)= 6
* 对于节点 F max(-3, -∞) => max(-3,-5) = -3
* 对于节点 G max(0, -∞) = max(0, 7) = 7

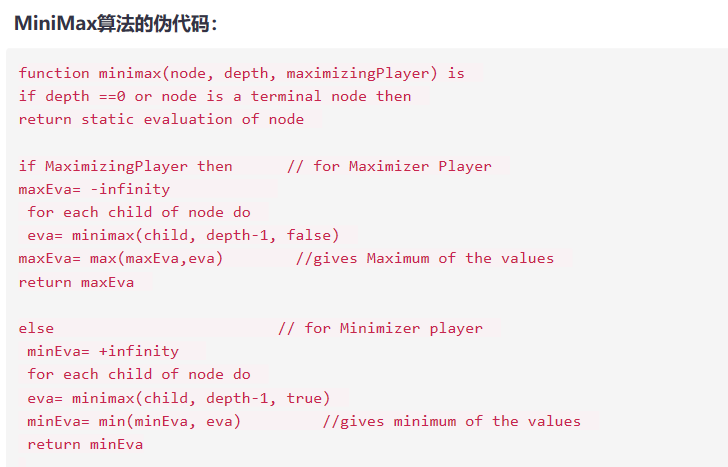


**第三步：**对于节点A ，max(4, -3)= 4



#### Mini-Max算法的属性

* Complete-Min-Max算法是完整的。它肯定会在有限搜索树中找到解决方案(如果存在)。
* 如果两个对手都以最佳方式进行游戏，则Optimal-Min-Max算法是最佳的。
* 时间复杂度 - 当它为游戏树执行DFS时，Min-Max算法的时间复杂度为O(b^m)，其中b是游戏树的分支因子，m是树的最大深度。
* 空间复杂度 - Mini-max算法的空间复杂度也类似于DFS，即O(bm)。



1. **α-β剪枝算法**

**1.问题描述**

将最小最小算法优化，就形成了α-β算法。

在上述的极大极小算法中，MIN和MAX过程将所有的可能性省搜索树，然后再从端点的估计值倒推计算，这样的效率非常低下。而α-β算法的引入可以提高运算效率，对一些非必要的估计值进行舍弃。其策略是进行深度优先搜索，当生成结点到达规定深度时，立即进行静态估计，一旦某一个非端点的节点可以确定倒推值的时候立即赋值，节省下其他分支拓展到节点的开销。

剪枝规则：

（1） α剪枝，任一极小层节点的β值不大于他任一前驱极大值层节点的α值，即α（前驱层）≥β（后继层），则可以终止该极小层中这个MIN节点以下的搜索过程。这个MIN节点的倒推值确定为这个β值。

（2） β剪枝，任一极大层节点的α值不小于它任一前驱极小值层节点的β值，即β（前驱层）≤α（后继层），则可以终止该极大值层中这个MAX节点以下的搜索过程。这个MAX节点的倒推值确定为这个α值。

α-β剪枝建立在两个假设上：

1. 整个博弈过程属于零和博弈，即一方的收益必然意味着另一方的损失，博弈双方的收益和损失相加总和永远是零，双方不存在任何合作的可能。粗俗理解就是你赚了我就亏了。
2. 博弈双方足够聪明，即每一方在决策时总会选择使自己利益最大化的决策。粗俗理解就是，假设是在下棋，那么大家下棋时大家走的都是固定的且是对自己最好的，下多少局也是这样。
3. 在极大极小算法中，MIN和MAX过程将所有的可能性省搜索树，然后再从端点的估计值倒推计算，这样的效率非常低下。而α-β算法的引入可以提高运算效率，对一些非必要的估计值进行舍弃。其策略是进行深度优先搜索，当生成结点到达规定深度时，立即进行静态估计，一旦某一个非端点的节点可以确定倒推值的时候立即赋值，节省下其他分支拓展到节点的开销。

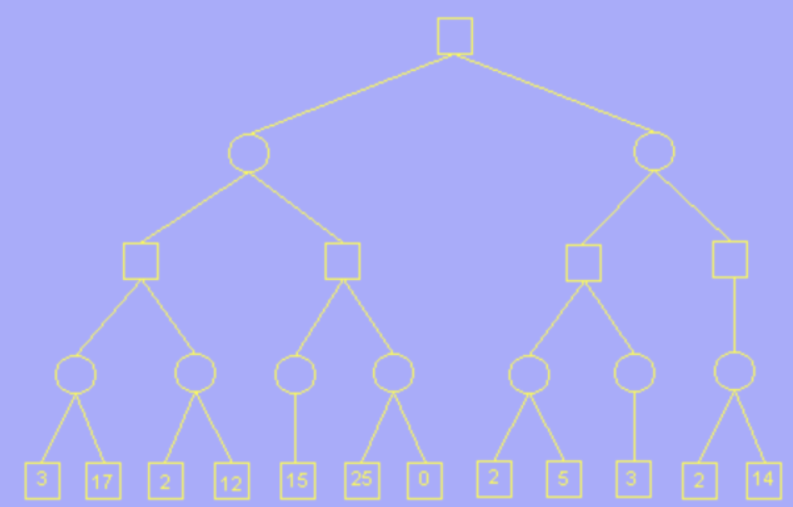
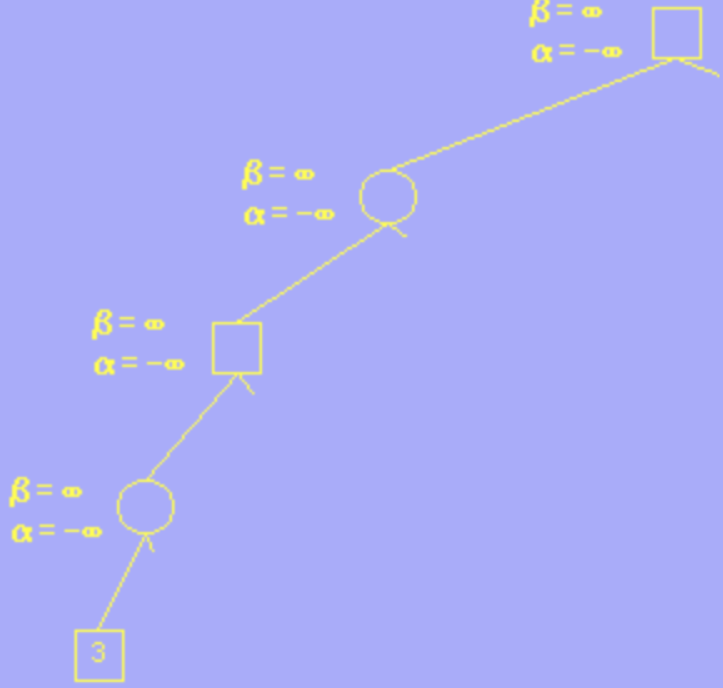
剪枝规则：

（1） α剪枝，任一极小层节点的β值不大于他任一前驱极大值层节点的α值，即α（前驱层）≥β（后继层），则可以终止该极小层中这个MIN节点以下的搜索过程。这个MIN节点的倒推值确定为这个β值。

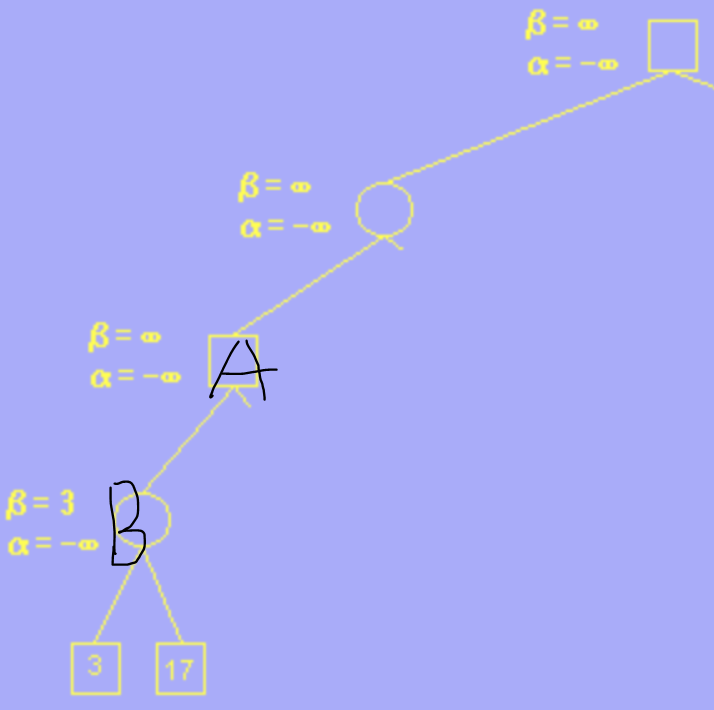
（2） β剪枝，任一极大层节点的α值不小于它任一前驱极小值层节点的β值，即β（前驱层）≤α（后继层），则可以终止该极大值层中这个MAX节点以下的搜索过程。这个MAX节点的倒推值确定为这个α值。

**2.算法流程**

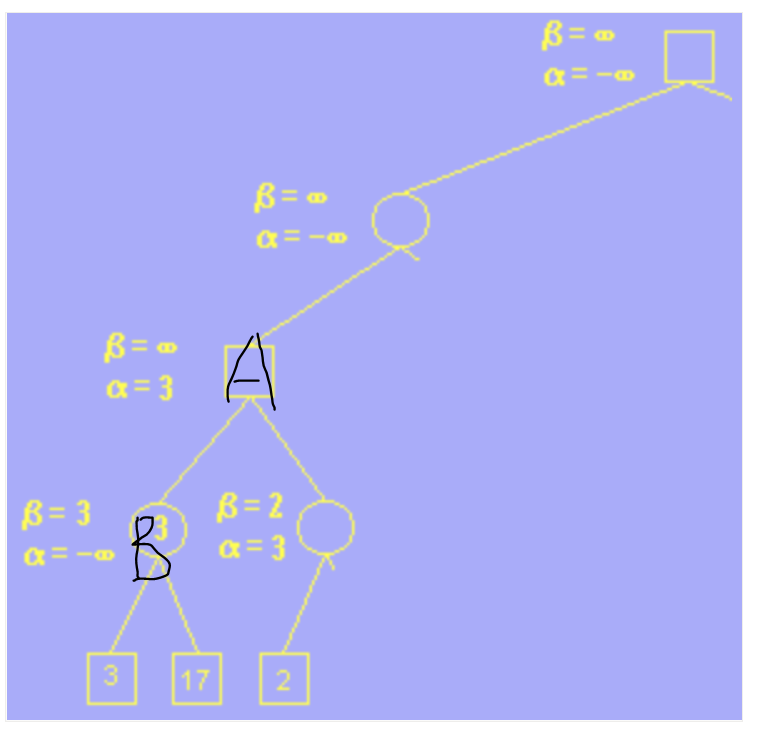
下图左一是整颗搜索树，我们设初始值α为负无穷大，β为正无穷大。

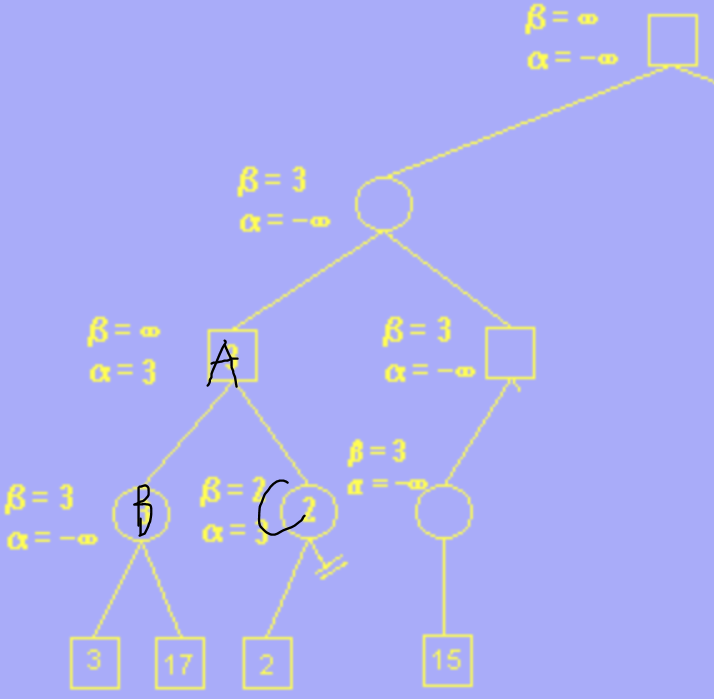
对于B而言，它要使A获得最小值，所以她修改β为3



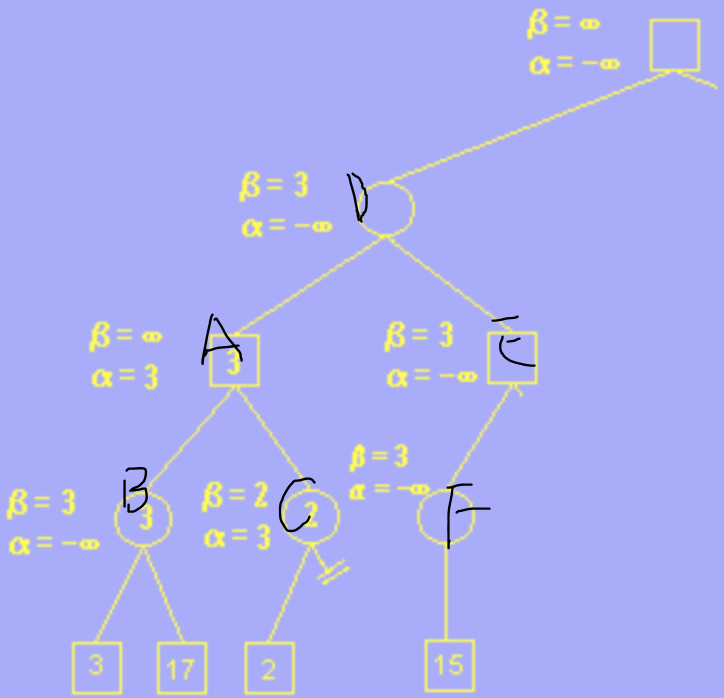
而对于A 而言，他要使自己获利最大，因此他要修改自己的最小值，他遇到比α大的值的时候，他要修改自己的α值。他修改了自己的α值为3



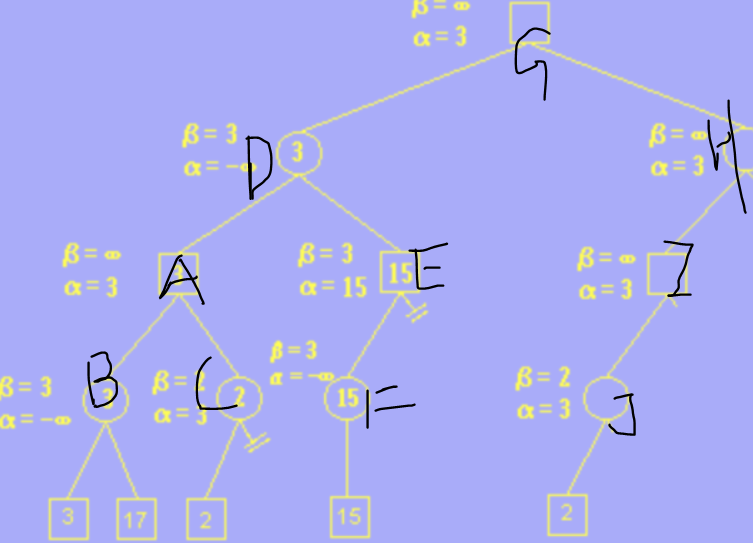
B使得A获利只有2，α=3,  β=2, α > β, 说明A只要选择C, 则c必然可以使得A的获利少于B。

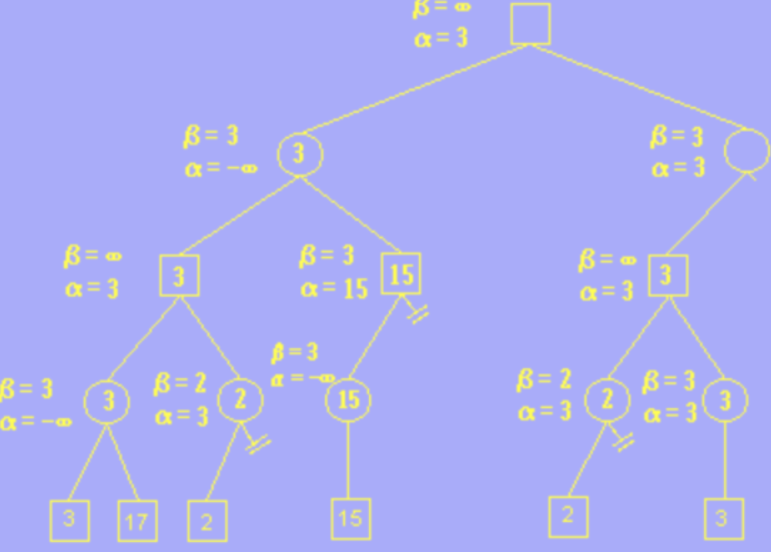


D要使得A利益最小,则E不能使得A的获利大于β, 也就是3. 但是若D选择E,第三层可以选择A使得A获利为15, α=15,  β=3, α > β, 故不需要再考虑E, 因为D不会选择E。

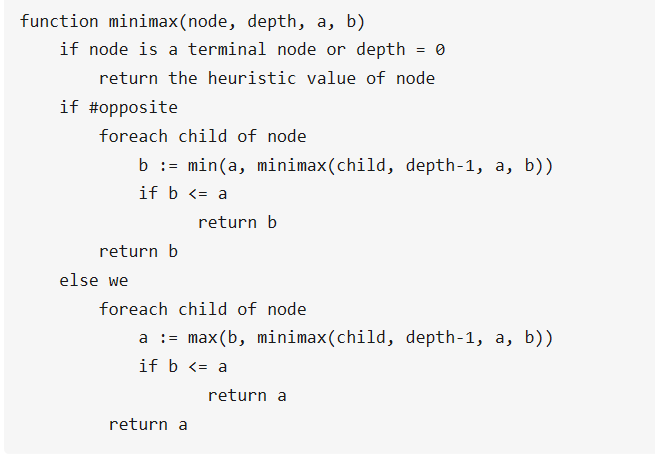


G要使自己获利最大，则要修改自己的最小值



然后一步步地进行下去...最终达到剪枝的目的。

**3.伪代码**



1. 代码实现

#include <stdio.h>

#include<conio.h>

#define COM -1

#define MAN 1

#define STEP 9

#define DRAW 0

#define ROW 3

#define COL 3 //三行三列的棋盘

#define MAX\_NUM 1000;

struct Move

{

int x;

int y;

};//结构体move下棋位置

//棋盘

int board[3][3] = { { 0,0,0 },

{ 0,0,0 },

{ 0,0,0 }

};

int tempBoard[3][3] = { { 0,0,0 },

{ 0,0,0 },

{ 0,0,0 }

};

//玩家=1

int player = MAN;

//最好的一步（x，y）

Move bestMove;

//当前深度

int currentDepth;

//谁先走

bool MAN\_first = true;

//判断输赢

int isWin() {

int i, j;

for (int i = 0; i < 3; i++)

{

if (board[i][0] + board[i][1] + board[i][2] == 3)

return 1;

else if (board[i][0] + board[i][1] + board[i][2] == -3)

return -1;

}//判断横线方向上谁赢了

for (int j = 0; j < 3; j++)

{

if (board[0][j] + board[1][j] + board[2][j] == 3)

return 1;

else if (board[0][j] + board[1][j] + board[2][j] == -3)

return -1;

}//判断反对角线谁赢了

if (board[0][0] + board[1][1] + board[2][2] == 3 || board[0][2] + board[1][1] + board[2][0] == 3)

return 1;

else if (board[0][0] + board[1][1] + board[2][2] == -3 || board[0][2] + board[1][1] + board[2][0] == -3)

return -1;//判断对角线谁赢了

else return 0;

}

//评估函数

int evaluteMap() {

bool flag = true;

int i, j;

if (isWin() == COM)

return MAX\_NUM;//如果计算机赢了，返回最大值，com为-1

if (isWin() == MAN)

return -MAX\_NUM;//如果计算机输了，返回最小值 ，man为1

int count = 0;//该变量用来表示评估函数的值

//将棋盘中的空格填满自己的棋子，既将棋盘数组中的0变为1

for (i = 0; i < 3; i++)

for (j = 0; j < 3; j++)

{

if (board[i][j] == 0)

tempBoard[i][j] = COM;

else

tempBoard[i][j] = board[i][j];

}

//电脑一方

//计算每一行中有多少行的棋子连成3个的

for (i = 0; i < 3; i++)

count += (tempBoard[i][0] + tempBoard[i][1] + tempBoard[i][2]) / 3;

for (i = 0; i < 3; i++)

count += (tempBoard[0][i] + tempBoard[1][i] + tempBoard[2][i]) / 3;

count += (tempBoard[0][0] + tempBoard[1][1] + tempBoard[2][2]) / 3;

count += (tempBoard[2][0] + tempBoard[1][1] + tempBoard[0][2]) / 3;

//将棋盘中的空格填满对方的棋子，既将棋盘数组中的0变为-1

for (i = 0; i < 3; i++)

for (j = 0; j < 3; j++)

{

if (board[i][j] == 0)

tempBoard[i][j] = MAN;

else tempBoard[i][j] = board[i][j];

}

//玩家

//计算每一行中有多少行的棋子连成3个的

for (i = 0; i < 3; i++)

count += (tempBoard[i][0] + tempBoard[i][1] + tempBoard[i][2]) / 3;

for (i = 0; i < 3; i++)

count += (tempBoard[0][i] + tempBoard[1][i] + tempBoard[2][i]) / 3;

count += (tempBoard[0][0] + tempBoard[1][1] + tempBoard[2][2]) / 3;

count += (tempBoard[2][0] + tempBoard[1][1] + tempBoard[0][2]) / 3;

return count;

}

void makeMove(Move curMove)

{

board[curMove.x][curMove.y] = player;

player=(player==COM)? MAN : COM;

}

void unMakeMove(Move curMove) {

board[curMove.x][curMove.y] = 0;

player = (player == COM) ? MAN : COM;

}

//得到有空位的集合

int getMoveList(Move moveList[]){

int moveCount = 0;

int i, j;

for (i = 0; i < COL; i++){

for (j = 0; j < ROW; j++){

if (board[i][j] == 0){

moveList[moveCount].x = i;

moveList[moveCount].y = j;

moveCount++;

}

}

}

return moveCount; //返回一共多少个空的位置

}

int miniMaxsearch(int depth){

int value; //估值

int bestValue = 0; //最好的估值

int moveCount = 0;

int i;

int m, n;

Move moveList[9];//保存可以下子的位置

if (isWin() == COM || isWin() == MAN){

return evaluteMap(); //一般是返回极大极小值

}

//如果搜索深度耗尽 ， 返回估值

if (depth == 0){

return evaluteMap();//估值给了谁？

}

//根据不同的玩家 进行赋值

if (COM == player){

bestValue = -MAX\_NUM;

}

else if (MAN == player){

bestValue = MAX\_NUM;

}

//一共多少步

moveCount = getMoveList(moveList);

for (i = 0; i < moveCount; i++){

Move curMove = moveList[i];

makeMove(curMove);

value = miniMaxsearch(depth - 1);

unMakeMove(curMove);

if (player == COM){

if (value > bestValue){

bestValue = value;

if (depth == currentDepth){

bestMove = curMove;

}

}

}

else if (player == MAN){

if (value < bestValue){

bestValue = value;

if (depth == currentDepth){

bestMove = curMove;

}

}

}

}

//找出能走的最好的一步

return bestValue;

}

//打印棋盘 电脑X ，玩家O

void printBoard() {

int i, j;

for (i = 0; i < COL; i++){

printf("-------------\n");

for (j = 0; j < ROW; j++){

if (board[i][j] == COM){

printf("| X ");

}

else if (board[i][j] == MAN){

printf("| O " );

}

else{

printf("| ");

}

}

printf("|\n");

}

printf("-------------\n");

}

//电脑下棋

void com\_play(){

miniMaxsearch(currentDepth);

board[bestMove.x][bestMove.y] = COM;

}

//玩家下棋

void man\_play(){

int x, y;

printf("请输入位置坐标 0 0为左上角 2 2为右下角 \n");

scanf("%d", &x);

scanf("%d", &y);

while (x < 0 || x > 2 || y < 0 || y > 2){

printf("您输入的坐标错误，请重新输入:x:(0~2) , y:(0~2)\n");

scanf("%d", &x);

scanf("%d", &y);

}

while (board[x][y] != 0){

printf("该位置已有棋，请重新输入:\n");

scanf("%d", &x);

scanf("%d", &y);

}

board[x][y] = MAN;

}

void setFirst(){

char c;

printf("\nDo you want to play first? y -你先走 , n-电脑先走");

for (c = getche(); c != 'Y'&&c != 'y'&&c != 'N'&&c != 'n'; c = getche());

if (c == 'n' || c == 'N') {

MAN\_first = false;

}

printf("\n");

}

int main(){

currentDepth = 9;

int step = 1;

setFirst();

printBoard();

if (MAN\_first){

player = MAN;

for (step = 1; step <= STEP; )

{

man\_play();

printBoard();

if (player == isWin()) {

printf("您获胜了\n");

break;

}

step++;

currentDepth--;

if (step == 10) {

printf("平局 ~~~");

break;

}

player = (player == COM) ? MAN : COM;

com\_play();

printBoard();

if (player == isWin()) {

printf("电脑获胜\n");

break;

}

step++;

currentDepth--;

player = (player == COM) ? MAN : COM;

}

}

else{

player = COM;

for (step = 1; step <= STEP;){

com\_play();

printBoard();

if (player == isWin()) {

printf("电脑获胜\n");

break;

}

step++;

currentDepth--;

if (step == 10) {

printf("平局 ~~~");

break;

}

player = (player == COM) ? MAN : COM;

man\_play();

printBoard();

if (player == isWin()) {

printf("您获胜了\n");

break;

}

step++;

currentDepth--;

player = (player == COM) ? MAN : COM;

}

}

getch();

return 0;

}

