人工智能课程大作业二

α-β剪枝的四子棋策略

沈磊
物理系 2015 级

一、四子棋背景介绍

在人工智能领域的研究中，博弈问题一直是一个代表性问题，研究历史也是非常久远。因为在通常大家的认识当中，博弈问题是比较能够考验一个人的智力水平的。如果计算机和人进行博弈，是一种典型的人工智能的体现。而且随着博弈问题的不断研究，研究出了很多人工智能方面的算法，也极大地推动了机器智能的进步。

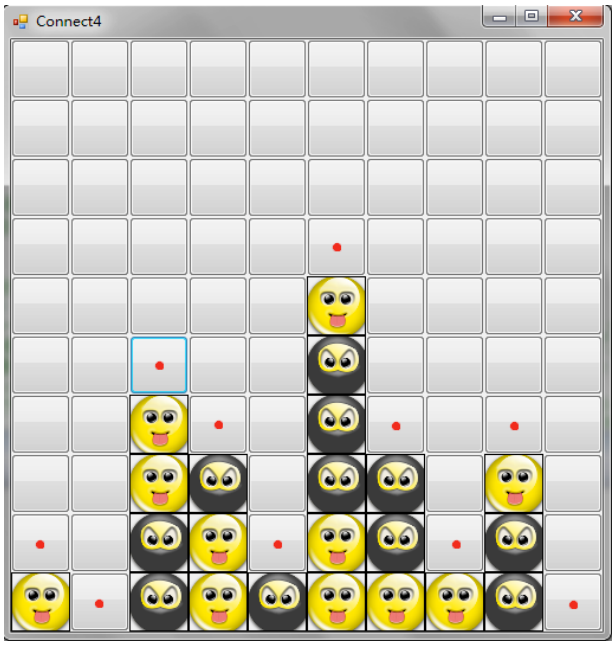
棋类问题是博弈问题的一个典型例子。在棋类博弈中，竞技双方的条件相对公平，对方的情况是透明的，双方的局势评价标准是一样的。这类问题需要用到的就是各种搜索算法，简而言之，谁对场上局势把握更准确，谁能想的比对方更远，谁就更有优势。所以在本次大作业中主要用到的就是**极小极大算法**和**α-β剪枝算法**。

棋类问题有很多，围棋最复杂，中国象棋、国际象棋、五子棋、四子棋复杂程度依次递减。棋类博弈过程可以看成是一个庞大的博弈树，刚刚说的“复杂”指的是搜索的博弈树每展开一步都的多少，子节点多就是“复杂”，子节点少就是“简单”。所以围棋每拓展依次会有很多节点，而四子棋拓展依次节点不会那么多（当然这和棋盘大小有关系）。所以本次作业以四子棋作为我学习人工智能搜索算法的一个检验。

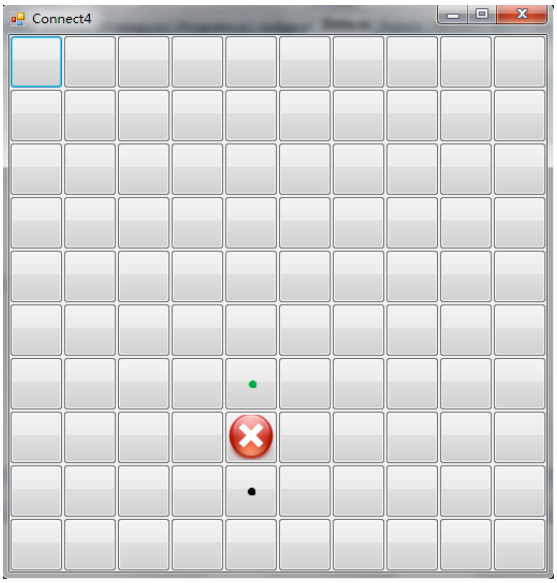
二、四子棋规则及算法分析

1、规则介绍

本次作业的四子棋规则不同于普通四子棋的规则。如图:



首先棋局是不定大小的长宽在[9,12]之间。每一步棋只能走每一列的顶部可走位置，如果满了，不可走。胜负判定还是水平、竖直和两个斜方向，任意成四连子即获胜。除此之外，还有一个不可落子点的约束，如图：



其他要求都不变，只在到达该不可落子点之后就跳过，下一步可走在该不可落子点的上面，也即图中的黑色和绿色点是先后可走的点。

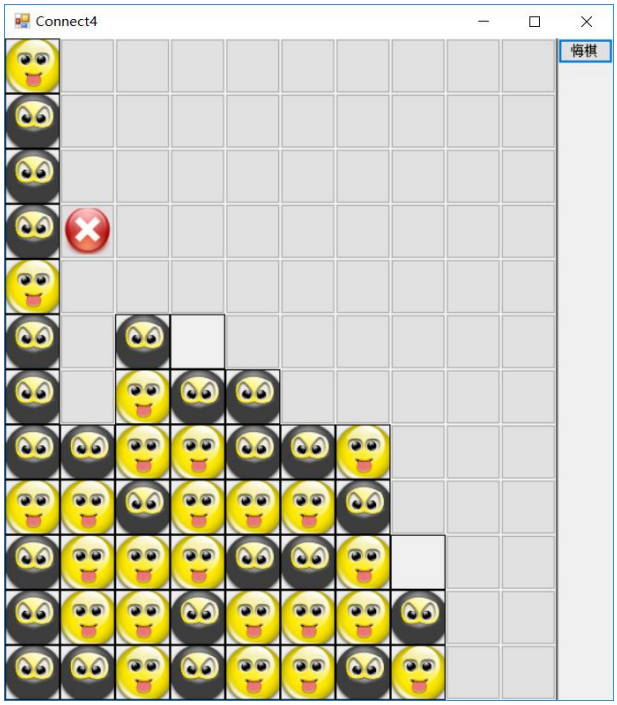
普通的四子棋，如果棋盘比较小，根据先后手有必胜策略。但是在本次作业中，由于不可落子点的不确定约束，必胜策略不是很容易应用。而且落子有方向要求（会在评价函数设计部分描述），所以和常规四子棋也不一样。

2、算法分析

此类型博弈问题，经常用的算法是极小极大搜索，α-β剪枝算法，蒙特卡洛树搜索算法。本次作业采用极小极大搜索和α-β剪枝算法。

（1）、极小极大搜索算法

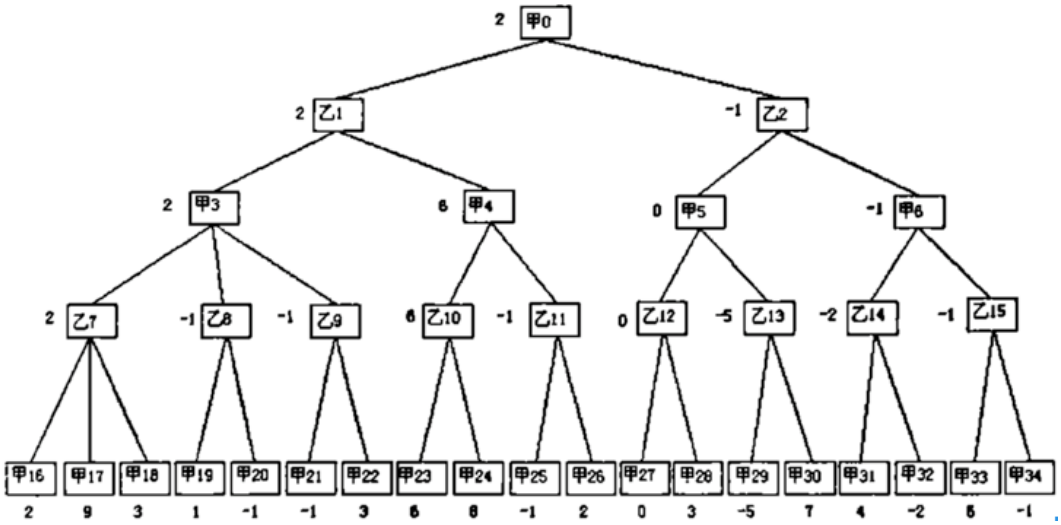
假设我们先让两个人进行四子棋博弈，过程如图：



我方和敌方轮流走子。我在走子的时候会对场上形势做一个判断：先找有没有成四连子的走法，如果有，那我就赢了。如果没有，再找有没有有利于我方的走法，如果有，再看看该步会不会让对方“更容易”赢，如果不会有利于对方，那可以走改点。如果有利于对方，看看别的点能不能，同样按照此判断。

注意上述走子思考方式有几个关键点：1、双方都清楚场上形势，都可以对场上形势做一个评价，而且假设双方同等厉害，双方对局势的评价结果应该是一样的（可用相同的评价函数）。2、同样的棋局，对我方有利，那么就对敌方不利。所以在走子的时候我方就要走评价函数的极大点，敌方走评价函数的极小点。3、我方在走子的时候要往后考虑几步，这样才能走出更加有利于我方的走法。

所以就引出了我们引用的基本算法：极小极大搜索算法。如图：



正如上图所示，假设评价函数越大就是对甲方越有利，评价函数越小就是对乙方越有利。所以可以称甲的层为 max 层，乙的层为 min 层。在博弈过程中，甲要使得评价函数越大越好，所以从甲层到乙层是搜索极大值，而反过来，乙层到甲层是搜索极小值。还有一点非常重要，下棋双方往往对后来子节点的评分会更加准确。因为越往后，双方的输赢就越明显。所以每

个节点的分值应当尽量由子节点来决定，此即所谓评价函数的反向传播方法，该方法也是α-β剪枝的基础。当考虑完若干层之后，甲就选他的子节点评价函数值最大的，即最有利于甲的；反过来，乙就选择评价函数最小的加点。上述过程就是极小极大搜索算法过程。

（2）、α-β剪枝算法

上述搜索算法很好，假设评价函数得当，计算机的计算足够强大，考虑的层数足够多，那么棋力可以预期会非常强。普通玩家大概可以到 4 层，比较好的选手可以考虑 6 层，更好的选手可以考虑到 8 层（当然上述说的是在棋局评价函数相当的时候）。

为什么大多数人考虑最多也不到 10 层呢？因为，这个博弈搜索树是指数级增长的，之后每增长一级，数量会是之前的倍数（围棋每一层状态数那么多，可想而知有多难）。所以不能这样一直加深考虑的层数。

考虑一种情况，当我走一步棋的时候有很明显对我方有利的走法（比如制胜走法），那么我就只在这一支上面拓展就行了。同理，如果这一步明显更有利于对方，那么我就不用在这一支上拓展了。这样可以减少计算量，而且和全部拓展效果是一样的。这种可以不考虑某些分支的算法就是著名的α-β剪枝算法。

用算法的语言描述就是：我们把 max 层的极小值叫做α，把 min 层极大值叫做β。当 max 层的极小值α大于其子节点 min 层的极大值β，那么该分支就可以不用再接着拓展了，这叫α剪枝；当 min 层的极大值β小于于其子节点 man 层的极小值α，那么该分支就可以不用再接着拓展了，这叫β剪枝。过程如图所示：

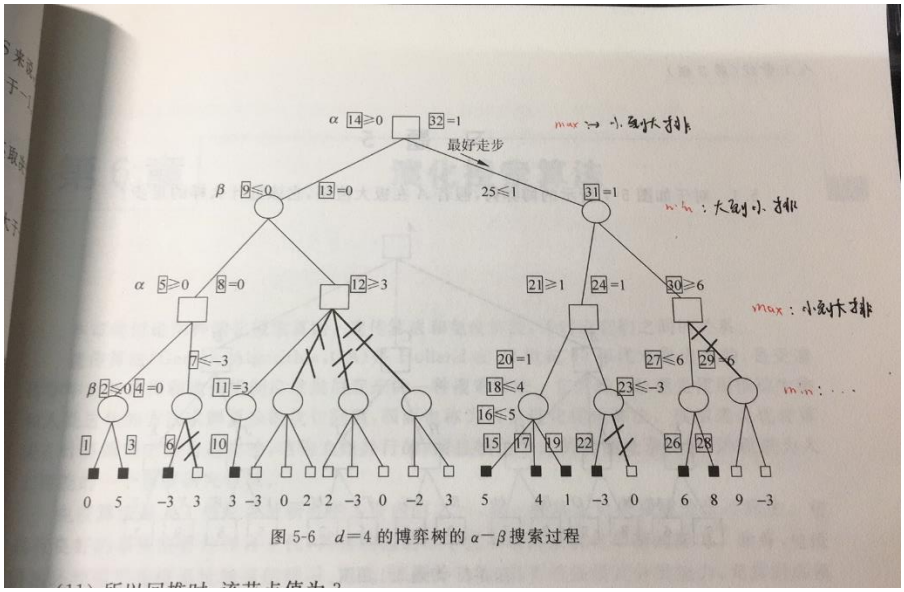
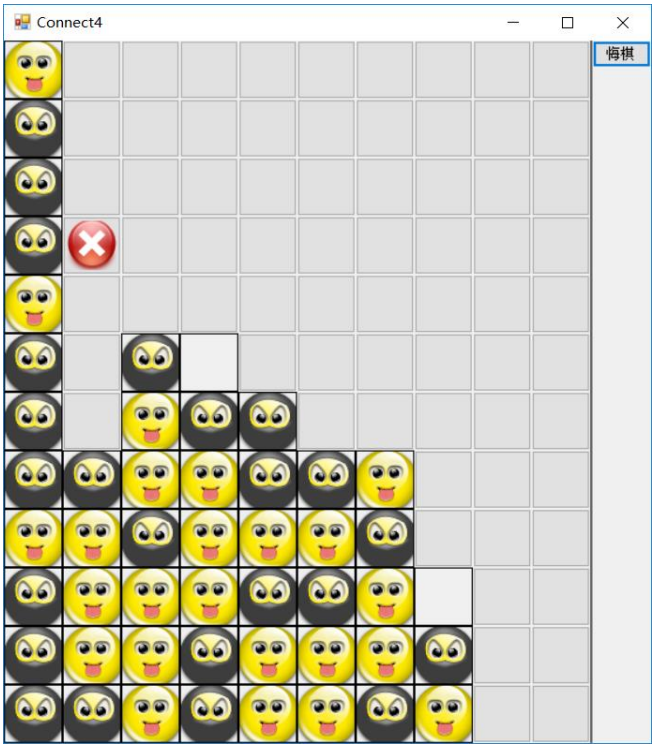


图 5-6　$d=4$ 的博弈树的 $\alpha-\beta$ 搜索过程

大体原则就是，在能把函数值传递给父节点就今早传递，这样就可以跟其他的子节点进行剪枝判断。

（3）、本次作业算法设计

本次作业采用的就是极小极大搜索算法基础上的α-β剪枝算法。该方法中有几个关键点很重要：1、评价函数要尽量好，好的评价函数意味着对局势把握好。2、层数尽可能多。3、剪枝效果尽可能好。



（a）、评价函数：

通过我不断和已有策略进行下棋，总结出对于棋盘局势的评级函数。具体评价包含以下一个子的形状：四连子，活三，活二，活一，死三，死二，死一。“活”即为最大可走的窗口大于 3。“死”就是最大窗口小于等于 3。每个分数成倍递减（具体可看代码）。

很重要的一点，棋的走法是数值向下的，横竖和两个斜线方向的四个方向不等价的。竖直方向的活子永远都是“半活”（最下面不能走子），所以评价函数在数值方向相对其他方向比较小。

很重要一点，评价函数永远一般都不会是尽善尽美的。**这时候有一种局势会非常重要，也就是“杀棋”**。只要把局盘中的杀棋给出，其实其他状态的评分不是非常接近完美，也可以用过加深层数的办法来增加杀棋出现的概率，这样就可以把其他无关痛痒的局势的评分影响降到最低，这样会更加接近真实的评价值。

（b）负极大值搜索：

细想一下，甲方和乙方的评价函数都是一样的，把甲子和乙子的评分相减，正的就是甲子，负的评分，就是乙子的评分。这样就可以运用一种从α-β剪枝算法改进的一种负极大搜索算法。因为两种极值搜索方式的算法形式是一样的，这样就可以把极大值搜索和极小值搜索写成一个函数，每次变换角色的时候，只要把甲的参数交换传给乙来调用负极大搜索函数就可以了（可看后面实现部分）。

三、代码实现和算法改进

1、实现过程

（1）、设置全局变量用于更新保存每次拓展的情况

对于 board，top，lastX，lastY 都是在拓展和评价函数计算的时候要用的，设成全局变量。

```cpp
#include <iostream>
#include <conio.h>
#include <atlstr.h>
#include <ctime>
#include <cmath>
#include <cstdlib>
#include "Point.h"
#include "Strategy.h"
#include "Judge.h"

using namespace std;

static int lastx;
static int lasty;
static int next_step[2]; //存储下一步的走步
static int *mytop;  //存储每一次拓展后的top
static int **myboard;  // 存储每次拓展后的board
static int DEPTH = 8; //设置剪枝深度


int evaluate_board(bool type, const int M, const int N); //局面评价函数
bool hasThreatPoint(bool type, int x, int y, const int M, const int N);  //判断一个点是否是威胁点
bool userWin(const int x, const int y, const int M, const int N, int* const* board);  //用户胜
bool machineWin(const int x, const int y, const int M, const int N, int* const* board);  //策略胜
bool isTie(const int N, const int* top);  //平局


int negmaxsearch(bool mytype,int mydepth, int myalpha, int mybeta, const int M, const int N){ ... }
```

Strategy (全局范围)

```cpp
    */
    //Add your own code below
    /* ... */
    ////////////////////////////////////////////////////////////////////
    //我的代码：          //
    board[noX][noY] = -1; //把不可落子点与无落子点区别出来
    //初始化myboard
    myboard = new int *[M];
    for (int i = 0; i < M; i++) {
        myboard[i] = new int[N];
        for (int j = 0; j < N; j++) {
            myboard[i][j] = board[i][j];
        }
    }
    //_cprintf("%d\t ", myboard[noX][noY]);
    //初始化mytop
    mytop = new int [N];
    for (int i = 0; i < N; i++) {
        mytop[i] = top[i];
    }
    //初始化lastx, lasty
    lastx = lastX;
    lasty = lastY;
    //_cprintf("%d\t%d\t ", lastx,lasty);
    //处理必杀棋和威胁棋
    int xindex, yindex;
    bool isabcut = true; //标记，用于判断用不用ab剪枝
    for (int i = 0; i < N; i++) {
        if (mytop[i] == 0) continue;
        else {
                xindex = mytop[i] - 1;
```

(2)、策略方制胜位置和地方威胁位置判断

如果有制胜位置，立刻走；然后如果有威胁位置，立刻堵上。

```
    lasty = lastY;
    //_cprintf("%d\t%d\t ", lastx,lasty);
    //处理必杀棋和威胁棋
    int xindex, yindex;
    bool isabcut = true; //标记，用于判断用不用ab剪枝
    for (int i = 0; i < N; i++) {
        if (mytop[i] == 0) continue;
        else {
                xindex = mytop[i] - 1;
                yindex = i;
                if (hasThreatPoint(true, xindex, yindex, M, N)) { //判断必杀棋
                    next_step[0] = mytop[i] - 1;
                    next_step[1] = i;
                    isabcut = false;
                }
                else {
                    if (hasThreatPoint(false, xindex, yindex, M, N)) { //判断威胁棋
                        next_step[0] = mytop[i] - 1;
                        next_step[1] = i;
                        isabcut = false;
                    }
                }
            }
        }

    //在策略先手时，把第一颗子放中间远离不可落子点的位置
    if (!(lastX>=0 && lastX<M && lastY >=0 && lastY< N)) {
```

(3)、如果策略方先手，走远离 noY 位置，并且尽量靠近棋盘中心

```
    //在策略先手时，把第一颗子放中间远离不可落子点的位置
    if (!(lastX>=0 && lastX<M && lastY >=0 && lastY< N)) {
        if ((N - 1 - noY) > noY) {
            next_step[0] = mytop[N-4] - 1;
            next_step[1] = N-4;
            isabcut = false;
        }
        else {
            next_step[0] = mytop[3] - 1;
            next_step[1] = 3;
            isabcut = false;
        }
        //_cprintf("lastX = %d\nlastY = %d\n", lastX, lastY);
    }
    if (isabcut) {
        int score = negmaxsearch(true, DEPTH, -999999999, 999999999, M, N);
    }
    //_cprintf("%d\t ", score);
    //把得到的下一步走法传给x,y
    x = next_step[0];
    y = next_step[1];
```

(4)、如果上述特殊情况都没有，那么就是蛮长的 ab 剪枝的过程了

```
    }
    if (isabcut) {
        int score = negmaxsearch(true, DEPTH, -999999999, 999999999, M, N);
    }
    //_cprintf("%d\t ", score);
    //把得到的下一步走法传给x,y
    x = next_step[0];
    y = next_step[1];
    delete[]mytop;
    for (int i = 0; i < M; i++) {
        delete[] myboard[i];
    }
    delete[]myboard;

    ////////////////////////////////////////////////////////////////////////////
    /*
        不要更改这段代码
    */
    clearArray(M, N, board);
    return new Point(x, y);
}
```

(5)、下图就是 ab 剪枝的负极大算法的实现

通过递归拓展分支，当达到局面确定或是达到拓展深度，再一层一层递归回原始节点。

```
int negmaxsearch(bool mytype,int mydepth, int myalpha, int mybeta, const int M, const int N) {
    int score=0;
    //判断局面；
    if (machineWin(lastx, lasty, M, N, myboard) || userWin(lastx, lasty, M, N, myboard) || mydepth == 0|| isTie(N,mytop)) {
        return evaluate_board(mytype,M, N);
    }
    //剪枝排序！越靠近lasty,优先拓展，可以极大提升效率。深度可再加深2层！
    int* sortindex = new int[N];
    int count = 0;
    for (int delta = 0; delta < N; delta++){ ... }

    //节点拓展
    for (int i = 0; i < N; i++) {
        if (mytop[sortindex[i]] == 0) continue; //满的列不拓展
        //备份拓展前的点
        int backx = lastx;
        int backy = lasty;
        int *backtop = new int [N];
        for (int j = 0; j < N; j++) {
            backtop[j] = mytop[j];
        }
        int **backboard = new int*[M];
        for (int p = 0; p < M; p++) {
            backboard[p] = new int[N];
            for (int q = 0; q < N;q++) {
                backboard[p][q] = myboard[p][q];
            }
        }
        //更新mytop和myboard,lastx,lasty
        lastx = mytop[sortindex[i]] - 1;
        lasty = sortindex[i];
        if (mytype) myboard[mytop[sortindex[i]] - 1][sortindex[i]] = 2;
        else myboard[mytop[sortindex[i]] - 1][sortindex[i]] = 1;
        if (mytop[sortindex[i]] == 1) {
            mytop[sortindex[i]] = 0;
        }
        else if (myboard[mytop[sortindex[i]]-2][sortindex[i]] == -1) mytop[sortindex[i]] -= 2; //不可落子点，更新mytop
```

```
    //alpha-beta剪枝
    if (score > myalpha) {
        if (mydepth == DEPTH) {
                next_step[0] = mytop[sortindex[i]] - 1;
                next_step[1] = sortindex[i];
        }
        if (score >= mybeta) {
            return mybeta;
        }
        myalpha = score;
    }

delete[]sortindex;
return myalpha;
}
```

（6）、局面的评价函数就是通过计算 top 的 2 子（策略方）和 1 子（用户方）的子的评分，相减。因为影响棋局都多数是 top 的位置，只需计算 top 各子的得分即可。

```
    ////////////////////
    //2子点的评价函数
int evaluate_each_point2(int x, int y, const int M, const int N){ ... }

int evaluate_each_point1(int x, int y, const int M, const int N){ ... }

    //type=true:查看该位置是否必胜棋;type=false:查看威胁棋
bool hasThreatPoint(bool type, int x, int y, const int M, const int N){ ... }
    //棋盘的评价函数
int evaluate_board(bool type,const int M,const int N){ ... }
```

2、算法改进

（1）、杀棋的判断

```
312     //考虑横向
313     //计算2子
314     int leftIndex2 = y - 1; //最左端超出2的位置
315     while (leftIndex2 >= 0){ ... }
321     int rightIndex2 = y + 1; //最右端超出2的位置
322     while (rightIndex2 < N){ ... }
328     int horicount2 = rightIndex2 - leftIndex2 - 1; //先算出2的连子数
329
330     int leftIndex22 = leftIndex2;
331     while (leftIndex22 >= 0){ ... }
338     int rightIndex22 = rightIndex2;
339     while (rightIndex22 < N){ ... }
346     int horiwidth2 = rightIndex22 - leftIndex22 - 1; //最大走子范围，再判断是不是活字
347     switch (horicount2) {
348     case 1:
349         if (horiwidth2 > 3) score2 += 2000;
350         else score2 += 500;
351         break;
352     case 2:
353         if (mytop[rightIndex2] == x + 1 && mytop[leftIndex2] == x + 1) { //!!!判断杀棋走法
354             score2 += 9999;
355             break;
356         }
357         if (horiwidth2 > 3) score2 += 5000;
358         else score2 += 1000;
359         break;
360     case 3:
361         if (mytop[rightIndex2]==x+1 && mytop[leftIndex2] == x+1) {  //!!!判断杀棋走法
362             score2 += 999999;
363             break;
364         }
365         if (horiwidth2 > 3) score2 += 10000;
366         else score2 += 2000;
367         break;
368     default:
369         return  99999999;
```

上图是所谓的杀棋，对局势起到至关重要作用的棋局，给予很大的分数，可以增强评价函数的准确性（原因在之前分析过了）。

（2）、攻击系数

```
724 bool hasThreatPoint(bool type, int x, int y, const int M, const int N){ ... }
795 //棋盘的评价函数
796 int evaluate_board(bool type,const int M,const int N) {
797     int temp_score2 = 0;
798     int temp_score1 = 0;
799     int xindex, yindex;
800     for (int i = 0; i < N; i++) {
801         if (mytop[i] == M) continue;
802         else {
803             xindex = mytop[i];
804             yindex = i;
805             //if (myboard[xindex][yindex] == 2) temp_score2 += evaluate_each_point2(xindex, yindex, M, N);
806             //if (myboard[xindex][yindex] == 1) temp_score1 += evaluate_each_point1(xindex, yindex, M, N);
807             //判断走的该步是不是危险走步
808             if (myboard[xindex][yindex] == 2) {
809                 temp_score2 += evaluate_each_point2(xindex, yindex, M, N);
810                 if (mytop[i] > 0) {
811                     if (hasThreatPoint(false, xindex - 1, yindex, M, N)) temp_score2 -= 9999999;
812                 }
813             }
814             if (myboard[xindex][yindex] == 1) {
815                 temp_score1 += evaluate_each_point1(xindex, yindex, M, N);
816                 if (mytop[i] > 0) {
817                     if (hasThreatPoint(true, xindex - 1, yindex, M, N)) temp_score1 -= 9999999;
818                 }
819             }
820             //////
821         }
822     }
823     if (type) return (int(0.5*temp_score2) - temp_score1);
824     else return (temp_score1 - int(0.5*temp_score2));
825 }
```

红框中的 0.5 是所谓的攻击系数。Temp_score2 是策略方的得分，系数 0.5 小于 1，表示策略方比较更多的考虑地方的走步，这样的设计可以使得在该评价函数下的走步策略不是那么“刚性”大，比较多的考虑敌方走步，可以弥补评价函数的缺陷。

（3）、判断最后子节点是不是危险走步

```
int evaluate_board(bool type,const int M,const int N) {
    int temp_score2 = 0;
    int temp_score1 = 0;
    int xindex, yindex;
    for (int i = 0; i < N; i++) {
        if (mytop[i] == M) continue;
        else {
            xindex = mytop[i];
            yindex = i;
            //if (myboard[xindex][yindex] == 2) temp_score2 += evaluate_each_point2(xindex, yindex, M, N);
            //if (myboard[xindex][yindex] == 1) temp_score1 += evaluate_each_point1(xindex, yindex, M, N);
            //判断走的该步是不是危险走步
            if (myboard[xindex][yindex] == 2) {
                temp_score2 += evaluate_each_point2(xindex, yindex, M, N);
                if (mytop[i] > 0) {
                    if (hasThreatPoint(false, xindex - 1, yindex, M, N)) temp_score2 -= 9999999;
                }
            }
            if (myboard[xindex][yindex] == 1) {
                temp_score1 += evaluate_each_point1(xindex, yindex, M, N);
                if (mytop[i] > 0) {
                    if (hasThreatPoint(true, xindex - 1, yindex, M, N)) temp_score1 -= 9999999;
                }
            }
            //////
        }
    }
    if (type) return (int(0.5*temp_score2) - temp_score1);
    else return (temp_score1 - int(0.5*temp_score2));
}
```

如图的在到达中子节点的时候，判断一下是不是危险走步，用于对评价函数进行校正。细细一想，其实相当于考虑的层数更深了。

（4）、**ab 剪枝的排序算法**：提高剪枝效率

```
int negmaxsearch(bool mytype,int mydepth, int myalpha, int mybeta, const int M, const int N) {
    int score=0;
    //判断局面；
    if (machineWin(lastx, lasty, M, N, myboard) || userWin(lastx, lasty, M, N, myboard) || mydepth == 0|| isTie(N,mytop))
        return evaluate_board(mytype,M, N);
    }
    //剪枝排序！ 越靠近lasty,优先拓展，可以极大提升效率。深度可再加深2层！
    int* sortindex = new int[N];
    int count = 0;
    for (int delta = 0; delta < N; delta++) {
        if (count == N) break;
        for (int j = 0; j < N; j++) {
            if (abs(j - lasty) == delta) {
                sortindex[count] = j;
                count++;
                if (count == N) break;
            }
        }
    }

    //节点拓展
    for (int i = 0; i < N; i++) {
        if (mytop[sortindex[i]] == 0) continue; //满的列不拓展
        //备份拓展前的点
        int backx = lastx;
        int backy = lasty;
        int *backtop = new int [N];
        for (int j = 0; j < N; j++) {
            backtop[j] = mytop[j];
        }
        int **backboard = new int*[M];
        for (int p = 0; p < M; p++) {
```

其实 ab 剪枝的效率和子节点的排序有关系的。直观的想法，如果先考虑优秀的节点，那么剪枝效果会很好。**根据理论推算，最佳剪枝效率可使得每层的等效节点数是全拓展节点数的平方根。**也就是，如果本来每层要搜 10000 步，最佳剪枝可以每层等效搜 100 步。效率极大提升。

图中的按照靠近上一步走棋的附近开始拓展，这样可以使原本 6 层的深度，拓展到 8 层。

3、测试结果

使用我的策略与测试策略对弈 10 轮，结果如下：

结果（相对策略）	胜	负	平
100	0.45	0.55	0
90	0.6	0.4	0
80	0.7	0.3	0
70	0.95	0.05	0
60	0.85	0.15	0

54	0.9	0.1	0
40	0.95	0.05	0
34	0.9	0.1	0
20	0.9	0.1	0
10	0.9	0.1	0

可见在我设计的评价函数基础下，8 层的搜索深度效果还是非常好的。

4、收获体会

通过调参发现

（1）、评价函数的重要性非常大，评价函数的好坏直接影响策略的性能。所以我的上述改进算法很多是用来弥补评价函数的，效果非常好!

（2）、评价函数、拓展深度、攻击系数三者是相互补充，相互影响的。通过调参发现，有些参数会明显增强性能。这个归根结底还是优化评价函数。

（3）、攻击系数越大，策略刚性越大，越容易在早期获胜，越到晚期，劣势越大。

（4）、ab 剪枝节点排序算法可以非常好的提升剪枝效率。搜索深度从原来 6 层，增加到 8 层。