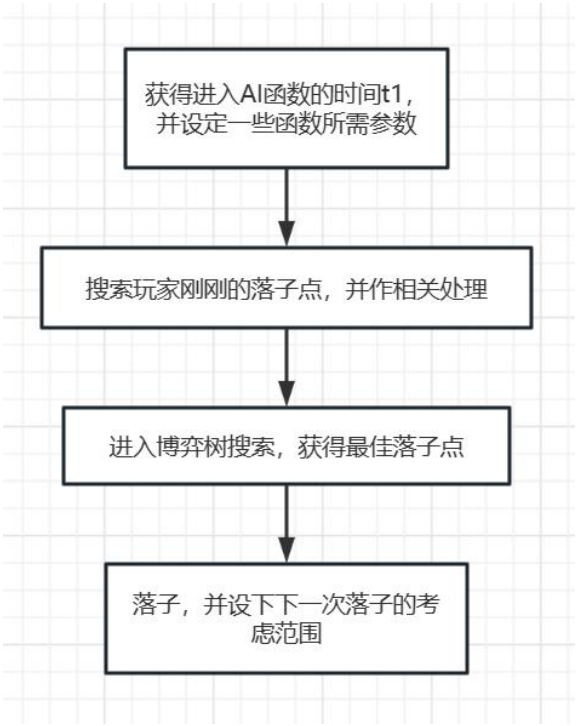
五子棋说明文档

戴玉鸣 2021K8009929004

以下主要对五子棋程序人机对战部分机器落子（以下以 AI 指代）实现过程进行说明：

进入 AI 函数后的主要流程如下：



PART1

PART2

PART3

PART4

下面对几部分的具体实现进行说明：

PART1

首先，调用 Linux 的<time.h>库，使用 time 函数获得进入函数的时间（严格说是从 1970 至今的秒数），如下图。这之后将用于严格限制搜索时间。

```
time_t t1;
t1=time(0);
```

之后，函数需要的一些参数将进行设定。

```
alpha[0]=-2000000;
beta[0]=2000000;
```

alpha 和 beta 数组的零号元素进行初始化，将用于剪枝

```
if(sign==1){
    corr=1;
}else if(sign==-1){
    corr=2;
}
// 对博弈树搜索所需参数进行初始化
flooor=0;
extend=0;
noreturn=0;
```

corr 为修正用，便于根据搜索深度和 AI 所执棋子颜色，在博弈树搜索中落下正确颜色的棋

flooor 为搜索深度，初始为 0 层；extend 为额外增加的最大搜索深度限制，初始也为 0；noreturn 标识是否向上返回某条博弈树分枝的 alpha 和 beta 值

PART2

以下以 AI 执黑为例：

```
if(sign==1){
    for(i=0;i<15;i++){
        for(j=0;j<15;j++){
            // 找到对手刚刚落子点后，设定下一步AI的落子
            if(inBoard[i][j]==4){
                inBoard[i][j]=2;
                partScoreUpdate(i,j);
                xMax[1]=i+RANGE;
                xMin[1]=i-RANGE;
                yMax[1]=j+RANGE;
                yMin[1]=j-RANGE;
                chessKey^=zobristWriteMap[i][j];
            }else if(inBoard[i][j]==0){
                hengMAX=i;
                zongMAX=j;
            }
        }
    }
```

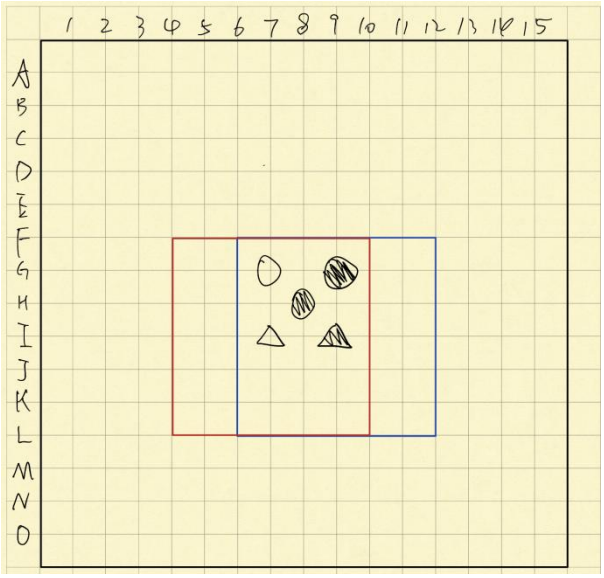
遍历棋盘，当搜索到玩家刚刚落子的点后，更新该落子点及其周围棋子的分数[1]，并且设下 AI 下两次落子的落子范围限制[2]，以及更新棋局特征值[3]

注释说明：

1. 棋子分数设定为根据该子所在四条直线（竖，横，左斜，右斜）上是否有活三活四等棋形以及各有多少，分多种情况给予不同的分数。为方便全局分（即表征局势优劣的分数）计算，相同情况黑子分数为正，白子分数为负。

因为落下棋子后不可避免的将会影响周围棋子的分数，所以每次落子后都会调用 partScoreUpdate 函数更新受到影响的棋子的分数。

2. 为避免博弈树搜索时考虑大量不必要的落子点，每层假想落子只考虑在上两次落子点周围落子。xMax 即代表下两次落子的 x 坐标的最大值，其余参数含义同理。

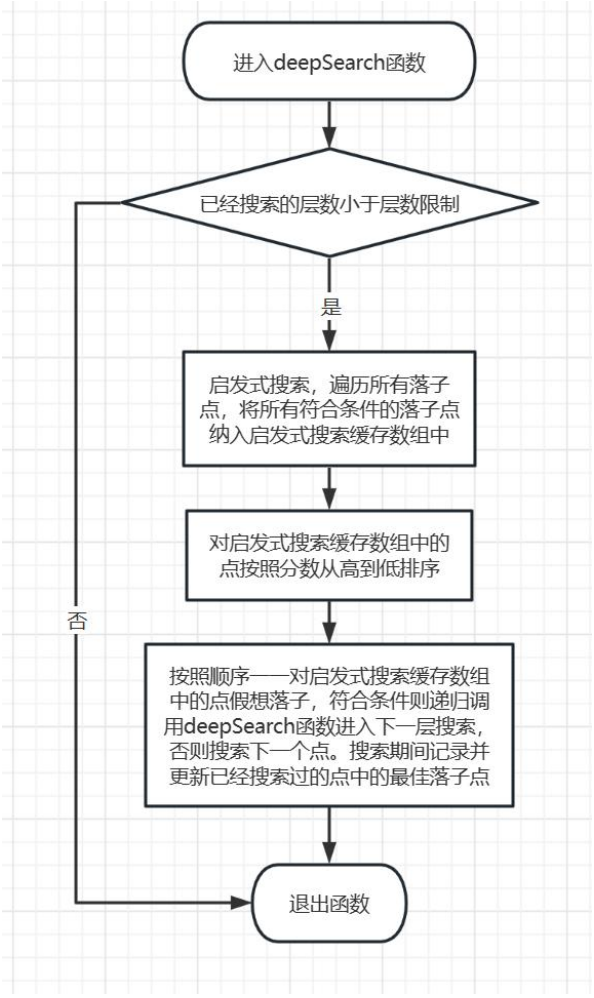


如图所示，黑白二色三角形为上两次落子点，那么白子下一次落子只会考虑在红色**或**蓝色框中的点上落子（其余特殊情况之后将说明）

3. 因为用到了 zobrist 置换表，本程序需要一个 64 位数代表不同的整体棋局。具体实现及作用之后将说明。

PART3

博弈树搜索是 AI 函数的核心。以下为简要流程图：



启发式搜索部分：

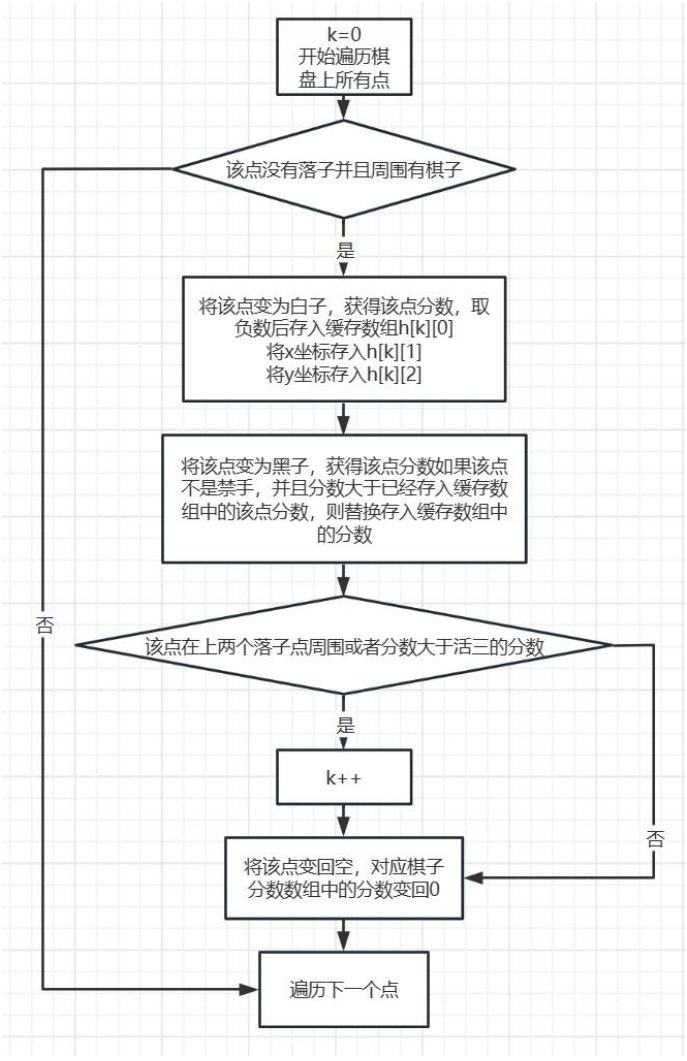
因为棋盘上可选的点实在是太多，而博弈树每层能考虑的点太少，所以必须要有一个筛选的机制。

启发式搜索的思路是遍历棋盘上的每个点，如果该点是空的（即没有落子），那么便假想在该点落一个白子并获得其棋子分数，随后再落一个黑子获得其分数，取两者最高连同坐标存入缓存数组中。如果这个点在上两个落子的限制范围内，或者它的分数大于活三的分数（即虽然不在落子范围内但仍有很高的价值），那么 k 将指向数组下一个位置，否则会被下一个遍历到的点的数据覆盖。

之所以要取最高分，是因为如果只考虑自己的分数的话，可能会漏看虽然对自己的子分数不高，但对方下了会分数很高的点。这样的点如果自己堵上也是很有价值的。

在以缓存数组中的分数作为排序依据进行快速排序后，正式的博弈树搜索将直接按序对缓存数组中存入的点进行搜索。因为这些点已经按照分数从高到低排过序，而分数越高的点越可能是最优选点，所以启发式搜索可以有效提高剪枝的效率。

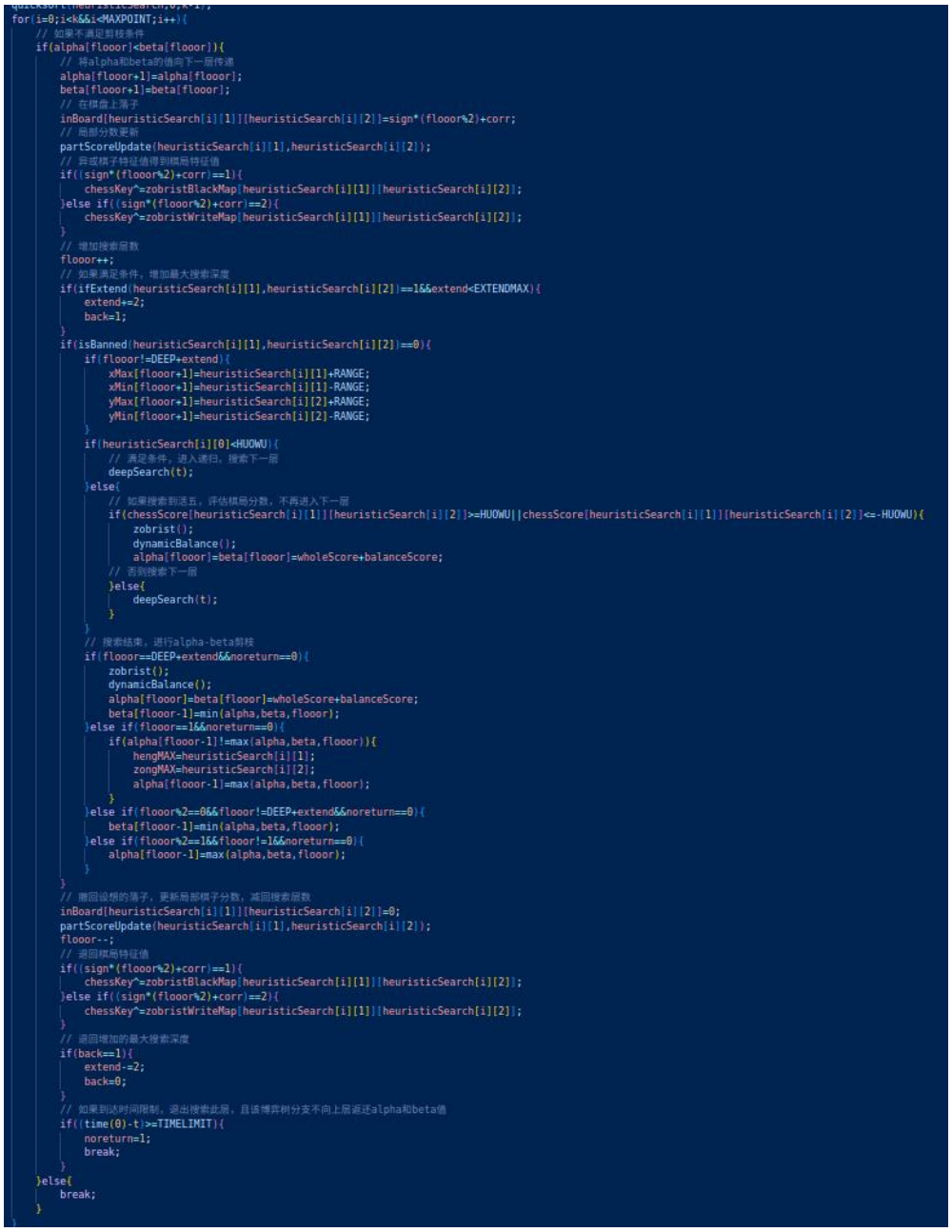
启发式搜索的流程图：



正式搜索部分：

正式搜索的思路较为复杂，限于篇幅流程图无法描述。下面将参照原码进行思路说明。

概览：



```
for(i=0;i<k&&i<MAXPOINT;i++){
    // 如果不满足剪枝条件
    if(alpha[flooor]<beta[flooor]){
        // 将alpha和beta的值向下一层传递
        alpha[flooor+1]=alpha[flooor];
        beta[flooor+1]=beta[flooor];
        // 在棋盘上落子
        inBoard[heuristicSearch[i][1]][heuristicSearch[i][2]]=sign*(flooor%2)+corr;
        // 局部分数更新
        partScoreUpdate(heuristicSearch[i][1],heuristicSearch[i][2]);
        // 异或棋子特征值得到棋局特征值
        if((sign*(flooor%2)+corr)==1){
            chessKey^=zobristBlackMap[heuristicSearch[i][1]][heuristicSearch[i][2]];
        }else if((sign*(flooor%2)+corr)==2){
            chessKey^=zobristWriteMap[heuristicSearch[i][1]][heuristicSearch[i][2]];
        }
        // 增加搜索层数
        flooor++;
        // 如果满足条件，增加最大搜索深度
        if(ifExtend(heuristicSearch[i][1],heuristicSearch[i][2])==1&&extend<EXTENDMAX){
            extend+=2;
            back=1;
        }
        if(isBanned(heuristicSearch[i][1],heuristicSearch[i][2])==0){
            if(flooor!=DEEP+extend){
                xMax[flooor+1]=heuristicSearch[i][1]+RANGE;
                xMin[flooor+1]=heuristicSearch[i][1]-RANGE;
                yMax[flooor+1]=heuristicSearch[i][2]+RANGE;
                yMin[flooor+1]=heuristicSearch[i][2]-RANGE;
            }
            if(heuristicSearch[i][0]<HUOWU){
                // 满足条件，进入递归，搜索下一层
                deepSearch(t);
            }else{
                // 如果搜索到活五，评估棋局分数，不再进入下一层
                if(chessScore[heuristicSearch[i][1]][heuristicSearch[i][2]]>=HUOWU||chessScore[heuristicSearch[i][1]][heuristicSearch[i][2]]<=-HUOWU){
                    zobrist();
                    dynamicBalance();
                    alpha[flooor]=beta[flooor]=wholeScore+balanceScore;
                // 否则搜索下一层
                }else{
                    deepSearch(t);
                }
            }
            // 搜索结束，进行alpha-beta剪枝
            if(flooor==DEEP+extend&&noreturn==0){
                zobrist();
                dynamicBalance();
                alpha[flooor]=beta[flooor]=wholeScore+balanceScore;
                beta[flooor-1]=min(alpha,beta,flooor);
            }else if(flooor==1&&noreturn==0){
                if(alpha[flooor-1]!=max(alpha,beta,flooor)){
                    hengMAX=heuristicSearch[i][1];
                    zongMAX=heuristicSearch[i][2];
                    alpha[flooor-1]=max(alpha,beta,flooor);
                }
            }else if(flooor%2==0&&flooor!=DEEP+extend&&noreturn==0){
                beta[flooor-1]=min(alpha,beta,flooor);
            }else if(flooor%2==1&&flooor!=1&&noreturn==0){
                alpha[flooor-1]=max(alpha,beta,flooor);
            }
        }
        // 撤回设想的落子，更新局部棋子分数，减回搜索层数
        inBoard[heuristicSearch[i][1]][heuristicSearch[i][2]]=0;
        partScoreUpdate(heuristicSearch[i][1],heuristicSearch[i][2]);
        flooor--;
        // 返回棋局特征值
        if((sign*(flooor%2)+corr)==1){
            chessKey^=zobristBlackMap[heuristicSearch[i][1]][heuristicSearch[i][2]];
        }else if((sign*(flooor%2)+corr)==2){
            chessKey^=zobristWriteMap[heuristicSearch[i][1]][heuristicSearch[i][2]];
        }
        // 返回增加的最大搜索深度
        if(back==1){
            extend-=2;
            back=0;
        }
        // 如果到达时间限制，退出搜索此层，且该博弈树分支不向上层返还alpha和beta值
        if((time(0)-t)>=TIMELIMIT){
            noreturn=1;
            break;
        }
    }else{
        break;
    }
}
```

下面正式进行说明：

```
for(i=0;i<k&&i<MAXPOINT;i++){
    // 如果不满足剪枝条件
    if(alpha[flooor]<beta[flooor]){
        // 将alpha和beta的值向下一层传递
        alpha[flooor+1]=alpha[flooor];
        beta[flooor+1]=beta[flooor];
        // 在棋盘上落子
        inBoard[heuristicSearch[i][1]][heuristicSearch[i][2]]=sign*(flooor%2)+corr;
        // 局部分数更新
        partScoreUpdate(heuristicSearch[i][1],heuristicSearch[i][2]);
        // 异或棋子特征值得到棋局特征值
        if((sign*(flooor%2)+corr)==1){
            chessKey^=zobristBlackMap[heuristicSearch[i][1]][heuristicSearch[i][2]];
        }else if((sign*(flooor%2)+corr)==2){
            chessKey^=zobristWriteMap[heuristicSearch[i][1]][heuristicSearch[i][2]];
        }
        // 增加搜索层数
        flooor++;
        // 如果满足条件，增加最大搜索深度
        if(ifExtend(heuristicSearch[i][1],heuristicSearch[i][2])==1&&extend<EXTENDMAX){
            extend+=2;
            back=1;
        }
```

首先，如果缓存数组中存有的所有点都被搜索完或者已经搜索过的点的数量达到限制，都会退出循环（即退出搜索），这是为了保证整体的搜索速度，每层搜索的点每增加一个，搜索时间都将指数增加，不利于满足每步的时限。

接着进行剪枝[1]条件判断。如果 alpha[flooor]大于等于 beta[flooor]，代表这条分支之后的点都不会比已经搜索到的点更好，可以直接剪去。如果没有发生剪枝，便将该层的 alpha 和 beta 值向下一层传递，将遍历到的缓存点对应的棋盘坐标变为黑子或者白子（具体颜色根据层数和 AI 先后手而定），并且更新该子周围棋子的分数和棋局特征值，之后 flooor++，表示搜索层数增加一层。

因为大多数情况冲四都会要求对方堵上成五的点，相当于一步棋限制了两步棋的落点，所以如果遇到冲四的点，extend+=2，表示最大搜索深度增加两层，从而能够多预测两步看到冲四之后的变化。

```
if(isBanned(heuristicSearch[i][1],heuristicSearch[i][2])==0){
    if(flooor!=DEEP+extend){
        xMax[flooor+1]=heuristicSearch[i][1]+RANGE;
        xMin[flooor+1]=heuristicSearch[i][1]-RANGE;
        yMax[flooor+1]=heuristicSearch[i][2]+RANGE;
        yMin[flooor+1]=heuristicSearch[i][2]-RANGE;
    }
    if(heuristicSearch[i][0]<HUOWU){
        // 满足条件，进入递归，搜索下一层
        deepSearch(t);
    }else{
        // 如果搜索到活五，评估棋局分数，不再进入下一层
        if(chessScore[heuristicSearch[i][1]][heuristicSearch[i][2]]>=HUOWU||
            zobrist();
            dynamicBalance();
            alpha[flooor]=beta[flooor]=wholeScore+balanceScore;
        // 否则搜索下一层
        }else{
            deepSearch(t);
        }
    }
```

如果这次落子不是禁手，会首先给之后两次落子设下落子范围。

之后如果这次落子点在缓存数组中对应的分数没有大于活五的分数，则递归调用 deepSearch 函数进入下一层搜索。

如果分数大于活五的话，有两种可能性：这步成了活五，或者这步堵了对方的活五。

第一种可能性的话，说明棋局已经结束了，这时候不能再往下一层搜索了，而应该立刻评估当前的棋局分数[2]，并进入 alpha-beta 剪枝的流程.

第二种可能性的话，则也递归调用 deepSearch 函数进入下一层搜索。

```
// 搜索结束，进行alpha-beta剪枝
if(flooor==DEEP+extend&&noreturn==0){
    zobrist();
    dynamicBalance();
    alpha[flooor]=beta[flooor]=wholeScore+balanceScore;
    beta[flooor-1]=min(alpha,beta,flooor);
}else if(flooor==1&&noreturn==0){
    if(alpha[flooor-1]!=max(alpha,beta,flooor)){
        hengMAX=heuristicSearch[i][1];
        zongMAX=heuristicSearch[i][2];
        alpha[flooor-1]=max(alpha,beta,flooor);
    }
}else if(flooor%2==0&&flooor!=DEEP+extend&&noreturn==0){
    beta[flooor-1]=min(alpha,beta,flooor);
}else if(flooor%2==1&&flooor!=1&&noreturn==0){
    alpha[flooor-1]=max(alpha,beta,flooor);
}
```

该部分为 alpha-beta 剪枝的具体代码实现，思路参考将在之后给出。

注意，所有的 alpha 和 beta 传值都要先满足条件 noreturn==0，原因将在下面给出。

```
// 撤回设想的落子，更新局部棋子分数，减回搜索层数
inBoard[heuristicSearch[i][1]][heuristicSearch[i][2]]=0;
partScoreUpdate(heuristicSearch[i][1],heuristicSearch[i][2]);
flooor--;
// 退回棋局特征值
if((sign*(flooor%2)+corr)==1){
    chessKey^=zobristBlackMap[heuristicSearch[i][1]][heuristicSearch[i][2]];
}else if((sign*(flooor%2)+corr)==2){
    chessKey^=zobristWriteMap[heuristicSearch[i][1]][heuristicSearch[i][2]];
}
// 退回增加的最大搜索深度
if(back==1){
    extend-=2;
    back=0;
}
// 如果到达时间限制，退出搜索此层，且该博弈树分支不向上层返还alpha和beta值
if((time(0)-t)>=TIMELIMIT){
    noreturn=1;
    break;
}
```

alpha-beta 剪枝结束后，代表该点的搜索已经全部完成。所以进行将该点变回空、更新周围棋子分数、层数退回等“善后工作”。

特别需要注意的是，如果 back==1，说明该点是冲四，增加了两层最大搜索深度限制，此刻需要退回，即 extend-=2。

之后再次调用 time 函数获得当前时间，减去 t 后得到直到此时 AI 函数的执行时间，如果超时的话，将 noreturn 赋值为 1，同时退出该层搜索。

如果没有超出时间限制，则继续搜索该层下一个点。

noreturn 的作用是：如果检测到超出时间限制，我们需要立刻退出博弈树搜索，并在目前搜索到的最优位置落子。但此时不能直接使用 goto 语句跳出递归，因为还没有进行过假想落子的“善后”。所以必须要一层一层退出，只是每层不再搜索下一个点而已。

在这个过程中，如果没有 noreturn 的限制，每层的 alpha 和 beta 值还是正常向上一层传递。然而此刻正在搜索的这条分支大概率没有搜索完全，返回上去的 alpha 和 beta 值是不安全的，即可能存在该分支实际的结果比已经搜索过的分值更差的可能性，这可能会导致一个不好的点被认为是好的点，从而替换掉已经搜索到的最好的点。为了避免这种情况，如果超出时限，则不再进行 alpha-beta 剪枝。

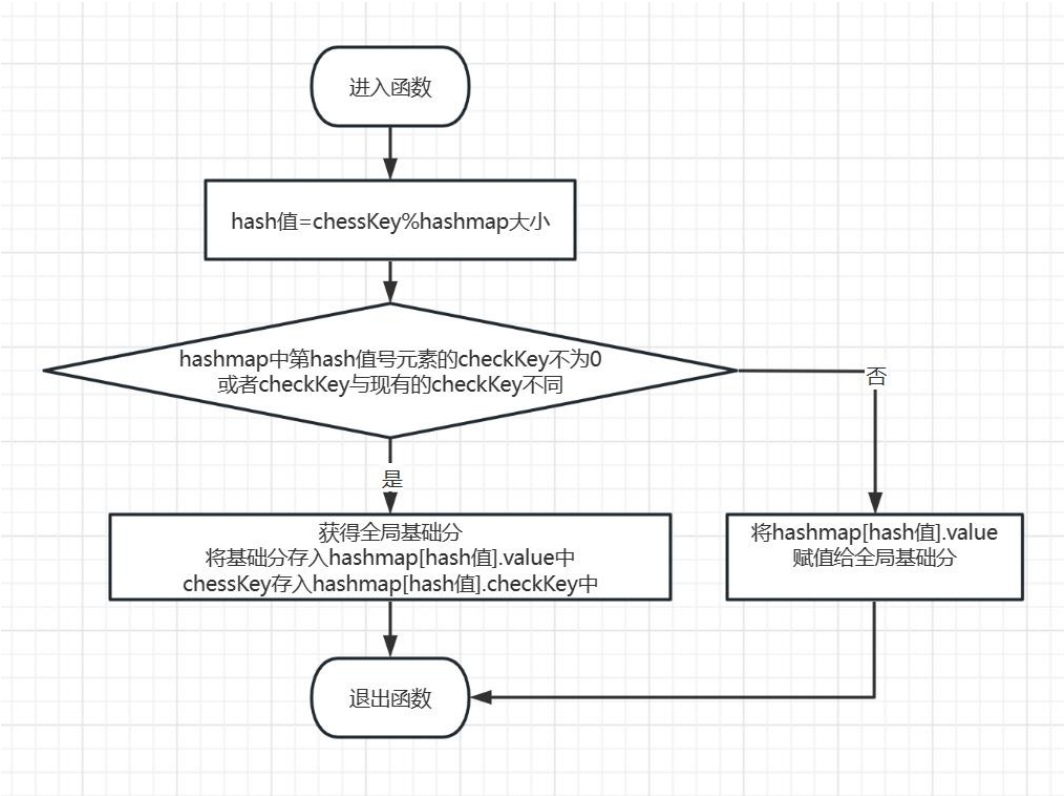
注释说明：

1. alpha-beta 剪枝是一种安全的剪枝策略，即剪去不必要的博弈树分支的同时保证最终搜索到的点仍然是最好的点。同时 alpha-beta 剪枝在向上层传值的过程实际上实现了 max-min 取值。具体的实现参考了 bilibili 网站上的视频【Alpha-Beta 剪枝算法（人工智能）】https://www.bilibili.com/video/BV1Bf4y11758/?share_source=copy_web&vd_source=33b639a524e8f2e0d55b5f73f40d1f42。

2. 当前的棋局分数分为两部分，一是直接将全部棋子分数直接加和得到的基础分，一是根据棋局优劣进行动态平衡的平衡分。

获得基础分用到了 zobrist 置换表。简要说明如下：创建两个大小为棋盘落子点数量的数组，将其用 64 位随机数填满，分别代表每个落子点的黑白子特征值。创建一个参数 chessKey，每次落子后，就将 chessKey 与落子点（颜色和坐标）对应的特征值异或一下，如果这个点变回空，也异或一下。这样每种棋局都有一个对应的 chessKey，在高质量 64 位随机数的前提下不同棋局之间的 chessKey 可以认为不可能冲突。

在搜索博弈树的过程中，每遇到需要获得棋局基础分的时候，就调用函数 void zobrist()



首先将代表棋局的 chessKey 对 hashmap（此处为 zobristMap）的长度求余，获得对应的 hash 值，查找 hashmap 对应 hash 值的元素，如果该 hash 值未存储过分数，则获取一遍当前的局势基础分存入 hashmap 中。如果该 hash 值已经存储了一个分数，则先对比下校验值（也就是棋局特征值 chessKey）是否相同，如果相同则直接使用存在 hashmap 中的基础分，如果不同则还是重新获取一遍。

Zobrist 置换表的优势在于每次落子的异或操作几乎不花费时间，但博弈树搜索的过程中会经常遇到重复的棋局，利用 zobrist 置换表可以大量节省重复计算相同棋局基础分的时间。

Zobrist 置换表并没有使用拉链法来避免 hash 冲突的情况，因为 hashmap 的大小开的很大，几乎不会遇到 hash 冲突。经过实际检查，一次博弈树搜索最多搜索不超过 20w 个点（包括重复的点），而 hashmap 的大小设定为 21474895（随便找的一个质数），远大于搜索的点的数量。

zobrist 置换表的命中率高达 50%左右。可以通过查看 matchTimes 和 totalTimes 实时知晓命中次数和总搜索次数，为了程序美观正式程序没有在屏幕上显示这两个参数。

平衡分部分，当 AI 认为当前局势有利的时候，将倾向于下出自己分数更高的棋子；当 AI 认为当前局势不利的时候，将倾向于下出阻止对手获得高分的棋子。具体实现如下：

```
// 设定平衡因子
factor=(float)2*wholeScore/(float)DENOMINATOR;
if(factor>0.2){
    factor=0.2;
}else if(factor<-0.3){
    factor=-0.3;
}
```

设定一个平衡因子 factor，其数值为当前局势基础分乘二后除以一个基值。factor 大致在-0.5 到 0.5 之间波动，为了避免进攻/防守倾向过高，factor 设有上限 0.2 和下限-0.3。

己方的棋子分数权重将增加一个 factor，对手棋子分数将减少一个 factor，即

局势分=（1+factor）*我方棋子总分-（1-factor）*敌方棋子总分

而经过简单处理后可知

局势分=（我方棋子总分-敌方棋子总分）+factor*（我方棋子总分+敌方棋子总分）

第一项就是局势基础分，那么要想获得平衡后的局势分，我们只需要基础分加上第二项即可。第二项即是平衡分 balanceScore。

```
// 动态平衡分数评估函数
int dynamicBalance(){
    int i,j;
    balanceScore=0;
    for(i=0;i<SIZE;i++){
        for(j=0;j<SIZE;j++){
            if(inBoard[i][j]==1){
                balanceScore+=chessScore[i][j];
            }else{
                balanceScore+=(-chessScore[i][j]);
            }
        }
    }
    // 对最终分数进行平衡
    balanceScore=factor*balanceScore;
}
```

PART4

因为在 alpha-beta 剪枝向上层传值的过程中，搜索到的最优落子点的坐标已经存入全局变量 hengMax 和 zongMax 中，所以只需在该坐标上落子即可。

```
// 进行落子
if(sign==1){
    inBoard[hengMAX][zongMAX]=1;
    partScoreUpdate(hengMAX,zongMAX);
    inBoard[hengMAX][zongMAX]=3;
    chessKey^=zobristBlackMap[hengMAX][zongMAX];
}else if(sign==-1){
    inBoard[hengMAX][zongMAX]=2;
    partScoreUpdate(hengMAX,zongMAX);
    inBoard[hengMAX][zongMAX]=4;
    chessKey^=zobristWriteMap[hengMAX][zongMAX];
}
// 对下一次落子设定落子考虑范围
xMax[0]=hengMAX+RANGE;
xMin[0]=hengMAX-RANGE;
yMax[0]=zongMAX+RANGE;
yMin[0]=zongMAX-RANGE;
```

在最优落子点落子后，更新周围棋子分数，并且获得最新的棋局特征值

为之后两次落子给出落子范围限制

AI 函数部分思路借鉴了 lihongxun945 在 github 上的 blog，特此感谢。
Issues • lihongxun945/myblog • GitHub