C语言课程设计五子棋AI实验报告

——五子棋Alpha\_cat

电子科技大学 2023080909007 姜皓裕

1. 设计需求

0、本五子棋AI名为Alpha\_cat

1. 由于五子棋AI先手的胜率较大，且黑子禁手的规则较复杂，所以仅仅设计业余版本（无禁手）并且Alpha\_cat执后手白子。
2. 本次设计侧重关注算法部分，将精确计算设置为一个重要部分，前端优化较为简陋，仅仅保留了基本的UI界面。
3. 主要算法
4. Minimax算法
5. alpha\_beta剪枝算法
6. 启发式搜索
7. 主要创作过程
8. **棋谱设计**（与评估函数密切相关，该部分侧重介绍棋谱设计的具体流程，部分原因将在评估函数一部分介绍）

通过一晚上的五子棋规则学习，我发现棋谱设计十分关键，棋谱设计的精细程度直接关系到评估函数的精确程度。棋谱设计的主要思路是分为连五，活四，眠四，活三，眠三，活二，眠二，活一，眠一，分别列举可能出现的棋谱。具体过程如下：

1. 将局面上的同色棋子成块的分割，这样可以避免同一部分棋子分数的重复计算，如下图：



B

A

其中A和A后面一个黑色棋子组成一个黑色棋子块，B和B后面一个白色棋子组成一个白色棋子块，在后续评估函数中，每个棋子块作为一个整体进行评估，可以有效减少重复，并且尽量减少各个棋子块之间的影响。

2）关注某种颜色出现的第一个棋子前面一个棋子，以这个棋子为标准进行一次分类。



B

A

比如若计算以A为开头的黑色棋子块（及A与A之后的一个黑色棋子），则需要从A之前的一个位置开始匹配，即从A前的空白位置开始匹配，而在棋谱设计阶段，也应从A前一个位置（即空格位置）开始分类。同理，若关注以B为起始的白色棋子块（即B和B后面一个棋子），应从B前的一个棋子（即黑色棋子）开始分类。

3）具体设计：

首先需要明确如下几点：

1. 异色棋子和围棋边界的效果是一样的，我们称之为“堵”
2. 同色棋子称之为“同”
3. 没有棋子称之为“空”
4. 由于我们关注的是同色棋子组成的棋子块，所以这个棋子块的容量最多不会超过5，而我们又要额外关注每个棋子块前一个棋子，所以我们所寻找的棋谱最长不会超过6.

接下来解释分类过程

Ⅰ、首先根据连续同色棋子的个数分类，有n=5，4，3，2，1五种情况，n为该棋谱中最长的连续同色棋子数目。比如下图：



其中黑色棋子块中有3个连续棋子，那么它属于以3为第一次分类的棋谱

Ⅱ、根据棋子块的前一个棋子进行分类，棋子块的前一个棋子有“空”和“堵”两种情况（因为棋子块是同色连续的，所以一个棋子块的前面不可能出现“同”的情况）。

Ⅲ、根据连续棋子后的两个位置进行分类，但是保证总的棋子数不超过6



比如上图这种情况，如果不检测连续棋子后面的两个位置，那么得到的结果是眠一，而实际上它是一个眠二。

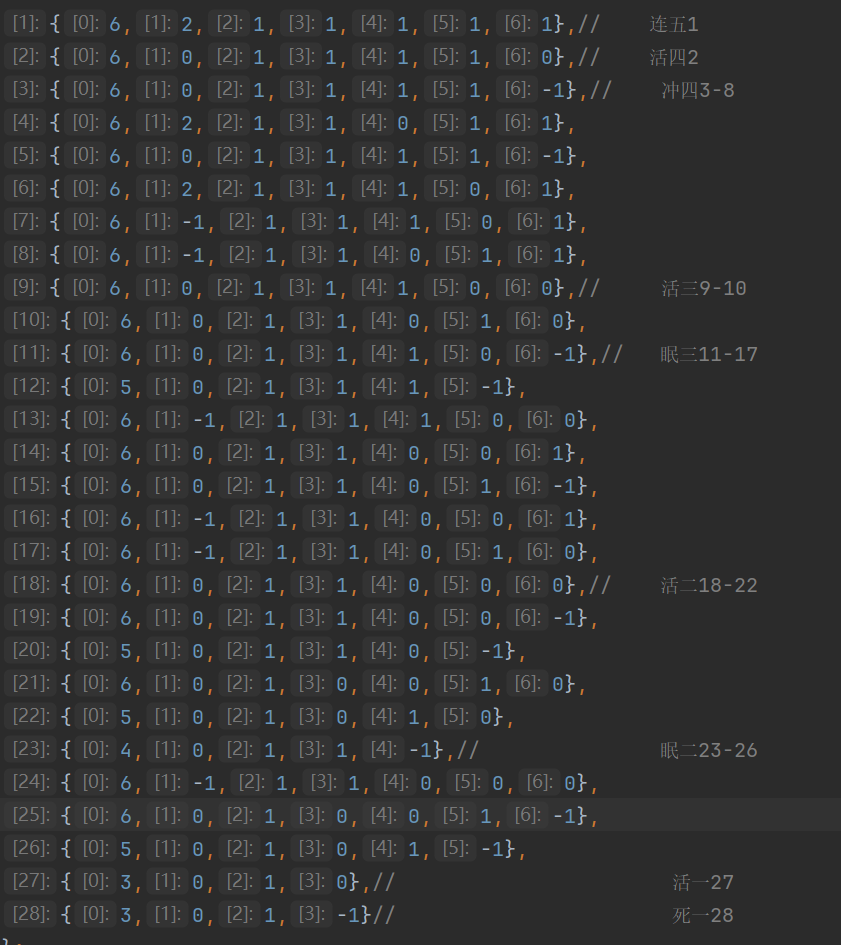
特殊地，某些活二或眠二棋谱检测连续棋子后面三个。比如下图：



关注左边第一个棋子向右三个，这其实是一个活二或眠二，而非活一。

Ⅳ、消除重复的棋谱。有左冲四和右冲四之分，而在棋谱设计中只需要设计一个方向上的冲四，因为在后续评估函数中会有两个方向的搜索。

Ⅵ、棋谱的存储。棋谱通过一个二维数组进行存储，其中每个一维数组的第一个数n表示该棋谱有多少个棋子，后面n个数表示棋子，其中1表示目标同色棋子，-1表示目标异色棋子和棋盘边界（“堵”），0表示空格，2表示“堵”和“空”均可。



1. **评估函数**
2. （已优化过）评估的范围。

每落一子，只改变该落子4个方向上的值，即只将该落子4个方向上的棋子拿出进行匹配。

1. 分值存储的方式。

用一个三维数组存储，每层一个二维数组存放分数，一共四层。若匹配行分值，则将分值存储在每行的第一个元素中；若匹配列分值，则将列分值存储在每一列的第一个元素；若匹配左上到右下斜线的分值，则将分值存储到左斜上第一个元素中；若匹配右上到左下斜线的分值，则将分值存储到右斜上第一个元素中。用三维数组存储可能会造成空间的浪费，因为每层二维数组相当于仅仅使用了两个边元素，但是将整个二维数组（即棋盘）表示出来有利于寻找第一个元素，而且每次递归都是使用自动存储期变量，当搜索四层的时候，只会出现4个三维数组，故忽略空间损耗。

这样的存储方式有提取和计算总分值较为方便的优点，但是也存在一些缺点，比如，落子时其实仅仅影响4个方向上部分棋子的分数，其实可以仅仅重新计算这些有影响的棋子的分数，但是如果把分数全都加在一个位置，就无法仅仅计算有影响的位置的棋子来更改分数。除此之外，本程序所用的围绕围绕棋子块正反算分的算分方法（下面介绍）也使得仅仅计算有影响的棋子的分数变得十分困难（这是因为很有可能从一个棋子块的中部截断而无法保持棋子块的完整性）。

1. 寻找每个方向第一个元素的方式——利用棋盘坐标的优良性质。

假设落子坐标（i，j），则有：

Ⅰ、横向算分：令layer=1，do\_i=i,do\_j=1;

Ⅱ、纵向算分：令layer=2，do\_i=1,do\_j=j；

Ⅲ、左上到右下算分：令layer=3， do\_i=i>j?(i-j+1):1,do\_j=i>j?1:(j-i+1);

Ⅳ、右上到左下算分：令layer=4，

do\_i=i+j<16?1:(i+j-15),do\_j=i+j<16?(i+j-1):15;

则每次匹配出正确的棋谱，则让score[layer][do\_i][do\_j]加上相对应的分数。

1. 评估函数与棋谱匹配的联系

从上面棋谱的存储可以看出，主要是通过“1”，“-1”，“0”，“2”这几个数字进行处理的，其中“1”表示目标同色棋子（可能是白子也可能是黑子，这取决于当前轮到哪一方落子了），“-1”表示目标异色棋子和棋盘边界（“堵”），“0”表示空格（“空”），“2”表示可以是“堵”也可以是“空”。

除此之外，程序中还建立了一个由数字形成的二维数组作为界面中棋局的映射，在这个二维数组中，“1”表示玩家落的黑子，“-1”表示Alpha\_cat落的白子，“0”表示“空”，“3”表示棋盘边界。

这样，我们便可以发现一个联系评估函数和棋谱的很好的性质，我们可以设置一个state，state的数值就是棋局映射二维数组的数值，比如轮到玩家落的黑子的state=1，轮到Alpha\_cat落的白子的state=-1，当匹配的时候，先将某个方向上的待匹配棋子转移到test一维数组中，然后将test一维数组中的每一个元素都\*state。比如，当前轮到Alpha\_cat下棋，则state=-1，假设test数组={0，-1，-1，1}（即test数组中有四个待匹配棋子，其中第二个和第三个-1表示Alpha\_cat落的白子，最后一个1表示玩家落的黑子），那么每个数值\*state后得到的结果是test={0，1，1，-1}，这个时候test数组的意义就变化了，其中1代表目标同色棋子（轮到Alpha\_cat下了，所以目标同色棋子是白色，也就是1代表了白色），-1代表目标异色棋子（玩家），这样就和所设置的棋谱1表示目标同色棋子，-1表示目标异色棋子的逻辑相匹配了。而这个方法的数学原理实际上就是（-1）\*（-1）=1，1\*1=1，0\*1=0，但是由于棋局中用3来表示棋盘边界，所以转化之后-3和3都表示棋盘边界，这个无伤大雅，因为棋局中只有一个3表示棋盘边界，而没有-3表示其他内容，所以将3和-3都判断成棋盘边界即可。

5）评估函数匹配的具体流程——正反计算取最大值。

Ⅰ、最初设想

最初是想以每一个棋子为开头，向后延伸若干棋子，再和已知棋谱进行匹配，如果匹配成功则加分。这样的匹配模式会有如下几个缺点：（1）无法检测反向棋子，比如如果棋谱中包含了左冲四，而没有右冲四，那么仅仅左冲四会匹配加分，而右冲四不会匹配加分。（2）（1）中的问题可以由增加匹配模型得以解决，但是使模型匹配变得冗长。（3）以每一个棋子开头实际上忽略了该棋子前面一个棋子对这个棋子以及这个棋子后面棋子的影响，比如搜索出连续三个同色棋子，如果不确定这个棋子块前面的一个棋子，则无法确定该棋子块是活三还是眠三，这大大降低了分数计算的精确度。（4）如果将每一个棋子为开头全部匹配一遍，那么不仅会大大增加计算量，而且会在某些情况下影响分数的计算，比如在极端的情况下，眠二的数量很多的时候，每个眠二都会计算2遍，有可能最终该棋局的分数可能会超过一个活三的棋局分数，这样就会对落子产生影响。

Ⅱ、从棋子块前一个位置开始检测——解决Ⅰ中问题（3）

每次检测的时候，如果是检测到“空”，则在该方向上跳到下一个棋子进行检测，如果不是空，则将标记调整至非空棋子的前一个位置，这个位置可以通过方向数组进行调整，具体如下：

start\_i=i-direct[num][0]

start\_j=j-direct[num][1]

其中，（i，j）是当前检测到的位置，num代表方向，[num][0]表示i的增量，[num][1]表示j的增量假设检测方向是从左上到右下，则有[num][0]=1，那么向左上方向移动一个位置,[num][1]=1,那么start\_i=i-1，start\_j=j-1，就实现了向前一个位置的定位。

通过匹配第一个非空棋子的前一个棋子，就能更好地拟合棋谱，更精确地算分。

Ⅲ、围绕棋子块进行正反算分——解决Ⅰ中问题（1）（4）

若根据Ⅰ中的最初设想进行评估，则可能会重复算分，这会带来两个比较严重的后果：一是时间消耗大，而是算分有误差。本程序的解决方案是把棋局分割成棋子块进行正反算分（棋子块就是连续的同色棋子）。

①首先，阐述正反算分。正如其名，这个方法就是在找到第一个非空的棋子之后，先正向算分，并标记第一个异色棋子或者空格或者棋盘边界（意味着一个棋子块的终结，第一个异色棋子或者空格或者棋盘边界的位置记作end），这时需要进一步分类，如果是空格则继续将棋局上的棋子存储到test待检测数组中（如果存储数量==6，则不继续存储，test数组的最大容量是6），但如果是异色棋子或者棋局边界，那么结束test数组的存储，意味着一次搜索结束。这么操作的原因是：如果是空格阻断则可能成为其他的棋型，如果是异色棋子或者棋盘边界，那么说明棋子块被彻底阻断了，不可能产生其他的棋型。然后将test数组与模型进行匹配，得到算分结果记为score\_1.

接着进行反向算分，以end的位置开始进行反向算分，此时，end的位置是目标棋子块的后一个位置，从这个位置向前将棋局上每一个棋子输入到test数组中，其方法和正向算分一致，匹配算分后记作score\_2.

然后取max{score\_1,score\_2}，这是让每一个棋子块都只有一次分数，最后将这个分数按照3）中的方式存储到score三维数组中。这样就解决了（1）中所说左右冲四的问题。

②其次阐释围绕棋子块操作。

起初，我在网上查找资料，有的资料是将已经参与过算分的棋子进行标记，在后续算分中不再使用这些标记过的棋子，但是我发现这种方式有一些弊端会导致出错，如下图：

从左数三个棋子可以形成一个眠三，如果按照网上部分人所说的标记算过分的棋子，那么，在一次算分中，左边三个被当成眠三计算并且标记，右边三个会在下一次算分中被当作活三处理，但是显然右边数四个已形成活四，分数应该更高，所以网上这种标记的方法我认为还是不够精确。经过分析，我认为这种问题的出现整体性的割裂造成的，本身这种活四是由两部分形成的，但是屏蔽掉了一部分，就不是活四了。为了保证每次算分的整体性，我认为应当保留每一个棋子块左右两部分的其他棋子块，为了在尽量删除重复计算的棋子而尽量不影响匹配和算分的准确性，我认为可以将一串棋子按照棋子块进行分割，对每一个棋子块进行一次算分。

由①中可知，在正向算分的过程中，标记了end位置，以end为起点再次进行了反向算分，在两次算分的过程中，目标棋子块参与了两次算分，取最大的一次，也就是对于这个棋子最优的选择。

同时，虽然是正反两次算分，但是大部分相同的棋型并不会被重复计算两次，只有高度对称的棋型才会被计算两次，例如：

显然在正向计算左侧三个棋子的时候会计算出一个冲四，但当反向计算最后一个棋子的时候并不会计算出一个冲四，这是因为在棋谱中仅记录了{-1，1，1，1，0，1}，但是反向计算的时候得到的棋型是{0，1，0，1，1，1}，没有这个模型所以不会按照冲四计算。

③围绕棋子块正反算分的利弊

以②中例子来说，对于左侧三个单独的黑子，每一个子都被计算了1次，这就相当于还是存在重复计算，但是右侧的3个子仅仅计算了一次，并且活四也可以被检测出来。也就是说，围绕棋子块正反算分的方法有利：能更加清楚的检测出棋型，部分减少了重复。有弊：对于某些特殊棋型，还是会出现重复计算的现象。

不过有一些方法可以尽量缩减弊端而突出优势，比如本程序采取的策略：直接将以上例子中左侧三个棋子如此形成的棋型从匹配模型中删去，也就是说匹配左侧三个棋子的时候，当检测左数第一个棋子的时候，它会将test数组变为{0，1，0，1，0，1}我们认为这个棋型不存在，即得分为零。而在模型中设置活二{0，1，0，1，0}，其分数与眠三分数一致，这样就检测不出整体的眠三，但能检测出部分活二，以活二分数替代眠三分数，尽量使重复减少。

Ⅳ、关于算分函数的部分优化

①斜线死角优化

在斜线上，靠近棋盘角落的位置有一些地方不可能形成连五，所以在算分的过程中遇到这些地方直接跳过。

②死棋优化

死棋优化分成两种：确定死棋，仅结尾死棋。

确定死棋：两个异色棋子或者异色棋子和棋盘边界中间的棋子数目不超过4个，也就是说不会形成连五，那么直接将这种情况舍弃，不再匹配。

仅结尾死棋：只有在test数组结尾是异色棋子或者棋盘边界，这个时候是需要算分的，因为反向算分可能仍有分数。

③匹配模型时的优化

匹配模型的时候我们知道两个非常重要的数字——待匹配test数组中的棋子的数目以及模型中棋子的数目（存储到每个一维数组的第一位了）。

我们可以先比较这两个数目，再决定是否进行进一步的匹配工作。假设test数组中待匹配数目为n，模型中棋子的数目为m，若n<m则直接舍弃，不进行匹配，因为test数组的容量最大为6，如果n<m则必有n<6，那么一定是死棋优化的结果，那么test数组的最后一位一定是异色棋子或者棋盘边界。如果n>=m则一定可以进行匹配，因为模型可能是test数组的子集，例如：test={0，1，0，1，0，1}，而在上文中提到，这个眠三由于重复问题而从模型中删去了，但是模型中保留了活二{0，1，0，1，0}，显然n=6，m=5，可以进行匹配，并且这个活二可以在眠三中被检测出来。这样先判断个数再逐个匹配可以优化一些时间。

1. **Minimax算法**

这个没有什么可说的，就是很普通的Minimax算法。

1. **Alpha\_Beta剪枝优化**

这也没啥可说的，就是很普通的Alpha\_Beta剪枝优化。

有一个地方和解决必胜局面调戏问题相关联，将Alpha值设置为INT\_MIN+1,Beta值设置为INT\_MAX-1,如果遇到高于某个分数并且值非常大并且是在第一层搜索（即电脑下完第一次棋），直接返回Alpha=INT\_MAX，此时Alpha>Beta，剩余直接剪掉。

1. **启发式搜索**
2. 初步了解启发式搜索

网上人额外使用一个相对粗略的算分模型，比如计算在某范围内有多少个同色的棋子，以此来大致计算某一落点的得分，在这里强调棋子的单独得分和局面的总体得分的不同。我的第一反应是，用额外的算分函数会增加空间消耗，时间消耗，最重要的是要写很多行代码，另外，用一个相对粗略的模型还不能很好地拟合棋局，所以我就希望用棋局函数直接代替粗略的棋子函数，这样不仅省时省力，而且还提高了启发式搜索的精确度。

1. 主要问题

利用棋局函数直接给棋子算分其实不难，但是启发式搜索需要存储每一个可能落子的棋子分数，直接使用棋局算分函数的话，需要先对每个可能的落子算分，在对走完这步子的局面算分，这样就额外增加了不少多余的算分步骤，会增加不少时间消耗。

1. 解决方案

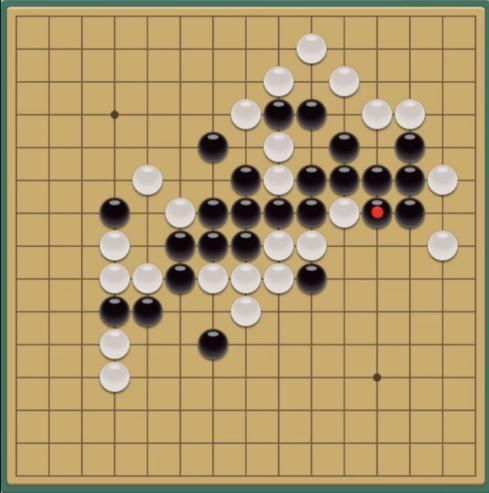
由于落子只会影响该落子位置4个方向上的分数，而对于棋子的评估也是从这四个方向上进行的，也就是说，棋盘分数的改变就是棋子的分数，即棋子分数=Δ棋局分数。

因此，我们只需要在一开始启发式搜索对每个落子进行评估的时候调用一次评估函数，并且记录Δ棋局分数，在启发式搜索快排的时候，根据Δ棋局分数进行排序，选择最优落点，而在计算下完最优子后的局面则可通过新棋局分数=原棋局分数+Δ棋局分数 计算得到，不需要再次调用评估函数进行计算。

1. **解决必输局面摆烂问题和必胜局面调戏问题**
2. 必输局面调戏问题

Ⅰ、问题描述：

当玩家形成双活三或者局面出现活四的时候，Alpha\_cat会判定自己必输，此时Alpha\_cat会下出看似摆烂的棋子。



Ⅱ、原因分析：

Alpha\_cat会走一个评估分数最大的一步，当局面必输的时候，Alpha\_cat无论怎么走都不会消除对方成五的大比分，所以会走出使自己分数最大的一步棋，比如自己形成一个活四或者活三，尽可能减少对方带来的负分。

Ⅲ、最初设想：

将成五分成两类，一个是两边没有棋子的成五，另一种是两边有棋子的成五，令后者的分数略低于前者的分数。这样Alpha\_cat会趋向于堵死活五。

Ⅳ、出现bug：

活四不堵问题解决了，但是双活三问题仍然存在，仍然不堵双活三。

Ⅴ、再次分析：

活四不堵和双活三不堵的本质都是一样的——必输局面下形成一个最有利于自己的局面。因此，可以将两种情况一起处理。但是，双活三的棋局复杂，难以从分数的角度处理。

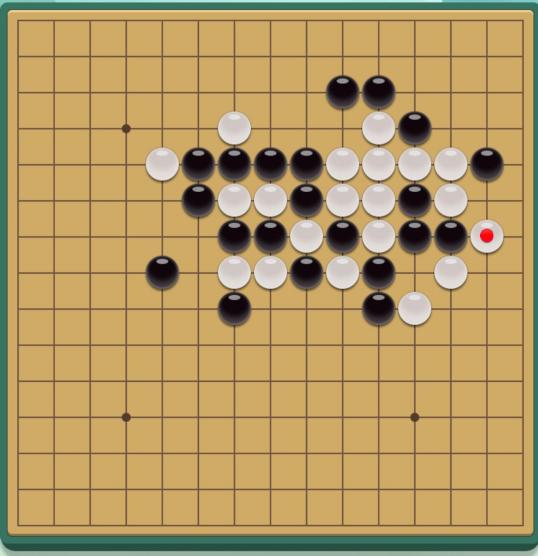
Ⅵ、解决方案：

所有棋局分数的最大值都可以通过alpha值返回，所以检测alpha值就可以判断是不是必输局面，如果是必输局面，则重新对棋局上的每一个位置进行一层算分，并且这次算分仅计算玩家的棋子分数，这样就可以摒弃电脑棋子分数带来的影响，最后选择玩家棋子分数绝对值最小的一步，标志着堵死对方分数最大的棋型。

1. 解决必胜局面调戏问题

Ⅰ、问题描述：

当Alpha\_cat下多个棋可以获胜，比如一个活四和一个活三同时存在时，Alpha\_cat会先形成两个活四“调戏玩家”，在形成成五，而不是直接形成成五。



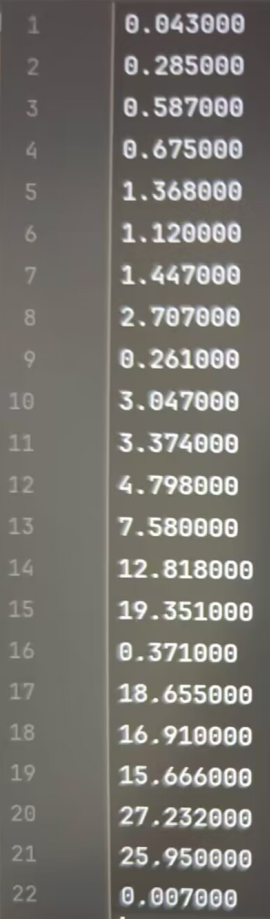
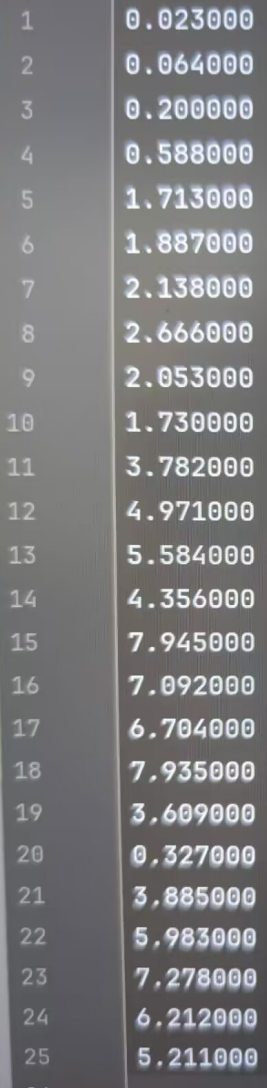
Ⅱ、解决方案：

利用Alpha\_Beta剪枝。将Alpha的初始值设置为INT\_MIN+1,Beta的初始值设置为INT\_MAX-1，当遇到非常大的分数并且是位于第一层搜索之后（即Alpha\_cat刚下完棋）的时候，就直接Alpha=INT\_MAX,此时后续直接剪枝。

7、最终优化

把debug模式改成release模式，快了三到五倍。

优化结果如下图（左图是debug，右图是release）：

1. **实验感受**
2. 最大的收获是对自己的代码实现能力有了更强的信心
3. 掌握了几种debug的方式：设置断点（数据量较小的时候）；打印日志（数据量较大的时候），直接把重要数据输出。
4. 明白了注释的重要性
5. 程序设计的过程中学会了代码架构，需要想清需求，设计代码，最后迭代优化。