Bait游戏实验报告

余孟凡（231240002、231240002@smail.nju.edu..cn）

(南京大学 计算机科学与技术系, 南京 210093)

摘 要: 读懂一个程序

编写一个基本的DFS接口,并逐渐优化。

学习并使用MiniMax搜索算法。

学习并使用MiniMax搜索（AlphaBeta剪枝）算法。

学习大项目的规范化注释。

1. minmax\_decider函数探究
   1. minmax\_decider函数的主要任务是为AI玩家选择最佳移动。它接受三个参数

### **game**: 当前的游戏状态

### **max\_depth**: 搜索的最大深度，限制了算法的搜索范围，以防止计算时间过长。

### **maximizing\_player**: 布尔值，指示当前玩家是最大化玩家（AI）还是最小化玩家（对手）。

## 函数逻辑

### 递归终止条件：

首先，函数检查是否达到最大搜索深度（max\_depth == 0）或游戏是否结束（game.is\_game\_over()）。如果是，则调用evaluate\_game\_state(game)评估当前游戏状态，并返回该评估值和None（表示没有移动）。

### 获取合法移动:

valid\_moves = game.get\_valid\_moves()获取当前游戏状态下所有合法的移动。

### 最大化玩家的逻辑:

如果当前是最大化玩家，则初始化max\_eval为负无穷（float("-inf")），并设置best\_move为None。

遍历所有合法移动：

为每个移动创建一个新的游戏状态（new\_game），复制当前游戏的棋盘和当前玩家。

在新游戏中执行当前移动（new\_game.make\_move(\*move)）。

递归调用minmax\_decider，将max\_depth减1，并将maximizing\_player设置为False（表示接下来是最小化玩家的回合）。

如果返回的评估值（eval）大于当前的max\_eval，则更新max\_eval和best\_move。

最后，返回最大评估值和最佳移动。

### 最小化玩家的逻辑:

如果当前是最小化玩家（对手），则初始化min\_eval为正无穷（float("inf")），并设置best\_move为None。

同样，遍历所有合法移动，创建新的游戏状态，执行移动，并递归调用minmax\_decider，这次将maximizing\_player设置为True。

如果返回的评估值（eval）小于当前的min\_eval，则更新min\_eval和best\_move。

返回最小评估值和最佳移动。

## 合并最大化和最小化逻辑

### 在传统的MiniMax算法中，通常会将最大化和最小化的逻辑分开实现，分别使用两个函数。这里通过一个布尔开关（maximizing\_player）将两者合并为一个函数。这样做的好处是：

### 代码简洁性: 只需一个函数来处理两种情况，减少了代码重复。

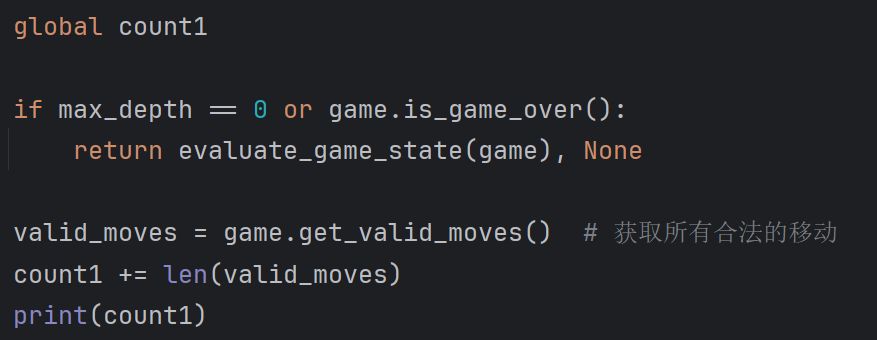
### 易于维护: 逻辑集中在一个地方，便于理解和修改。

### 灵活性: 可以轻松地在递归中切换角色，只需改变一个参数。

# Alphabeta剪枝：

## 我们首先来阐述alphabeta剪枝的原理：

首先我们明确minmax算法是为了决策树问题而产生的，但是该算法的问题在于：当深度增加时每增加一层深度就要付出指数倍的代价，由于棋盘情况的多样性，这一数字的基数已经很大：我们来简单做个测试：



为所有局面增加一个计数功能，可以很方便的看到递归搜索次数：

#### 6层深度时：第一步16250次搜索，第二步77983次搜索，第三部335851次搜索，第四步1848690次搜索，再往后我电脑顶不住了。开始尝试降低深度：

#### 5层深度时：第一步2477次搜索，第二步10511次搜索，第三部41838次搜索，第四步98293次搜索，第五步220438次搜索，第六步346239次搜索，第七步422841次搜索，第八步620750次搜索，第九步1099663次搜索，再往后仍然跑不动了。

#### 4层深度时：第一步427次搜索，第二步1552次搜索，第三部4963次搜索，第四步19378次搜索，第五步34493次搜索，第六步56727次搜索，第七步106449次搜索，第八步133135次搜索，第九步166673次搜索，第十步214634次搜索……最终结果赢了，不过赢得是乱下的我。

#### 到网上找了一个7个难度的AI，看看这个4层的能打到哪一步：<https://www.heibaiqi123.com/>

Emmmm,电脑搜索0.02秒最低一级的情况下，这个4层搜索都打不过，那么再降低层数已经没有必要了。

## Alphabeta剪枝的原理：

### Alphabeta剪枝是基于minmax剪枝。我们知道minmax是在本方层选择值最大的节点，在对方层选择值最小的节点。

那么如果我在本方层时，节点的值一开始是负无穷，然后过程中单调递增。如果我在对方层时，节点的值一开始是正无穷，然后过程中单调递减。那么这个搜索过程中，

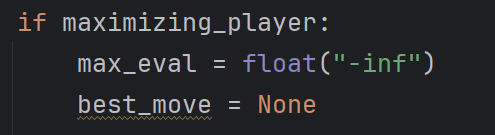
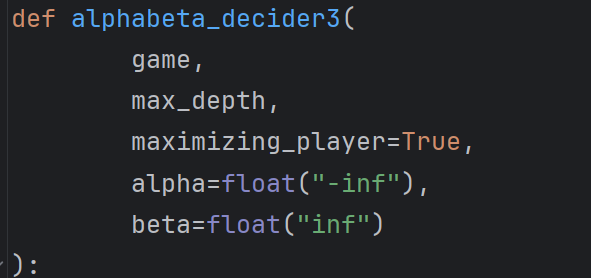
#### 最大层节点下面的第一个最小层节点搜出了x，而其他的节点搜出的值要小于x：由于最小层的节点只可能越搜越小，因此这一分支不可能再为父节点更新起到任何帮助了，于是我们就把这一节点剪枝掉了。同理，当我们更新完最大层的节点之后，要看他的父节点会不会认为这个节点对他起不了任何帮助了。

这就是alphabeta算法我理解的核心思路：只搜索对更新过程有用的节点。

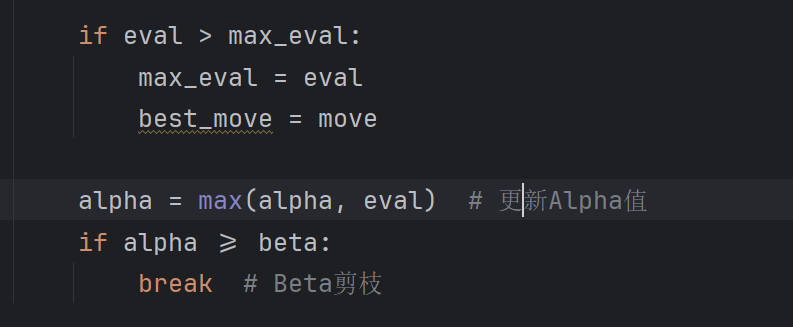
## 代码实现：

### 我们主要看在minmax算法上的更改：

#### 参数初始化：

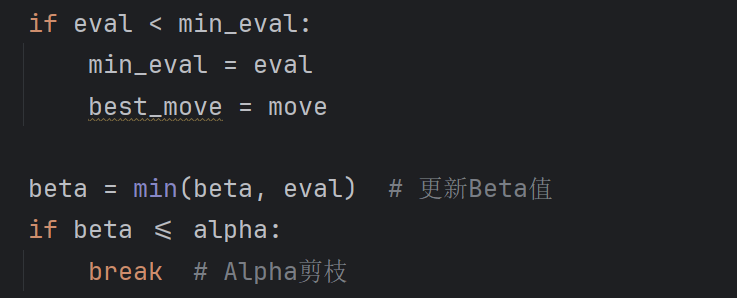


#### 当我们身处最大层节点时：



我们需要递增的更新alpha的值，但正如上文所说的，一旦这个值大于父节点获得的最小值了，那么他会毫不犹豫的被父节点抛弃（剪枝）。

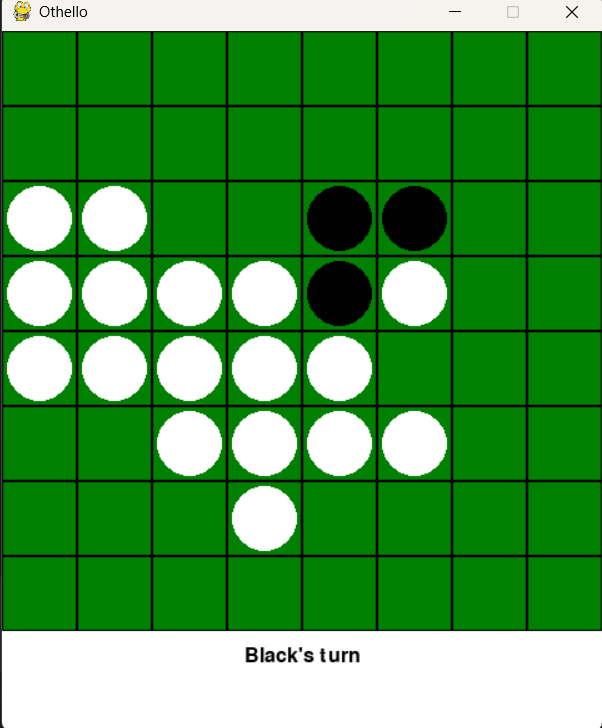
#### 当我们身处最小层节点时：



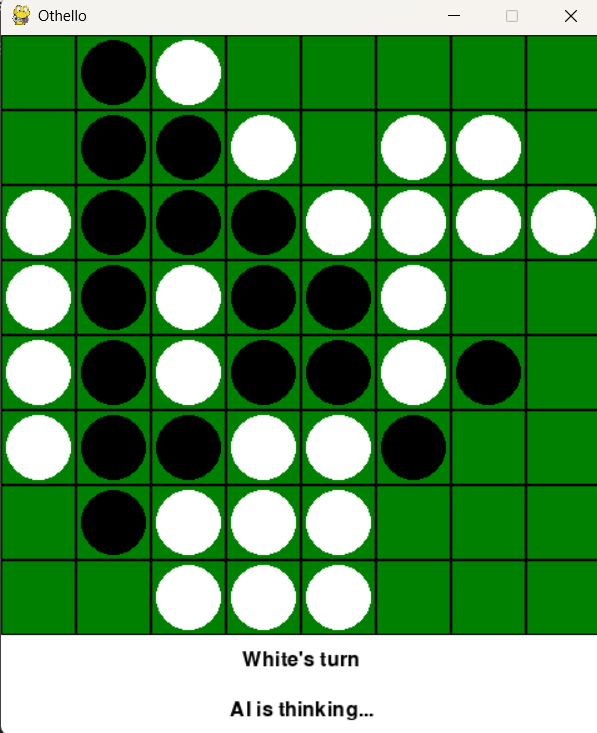
同理。

## 测试时发现了严重问题：前面反应速度确实还可以，但是到了后期本来以为我已经完蛋了，结果却突然降智，最后白白送了我这局：

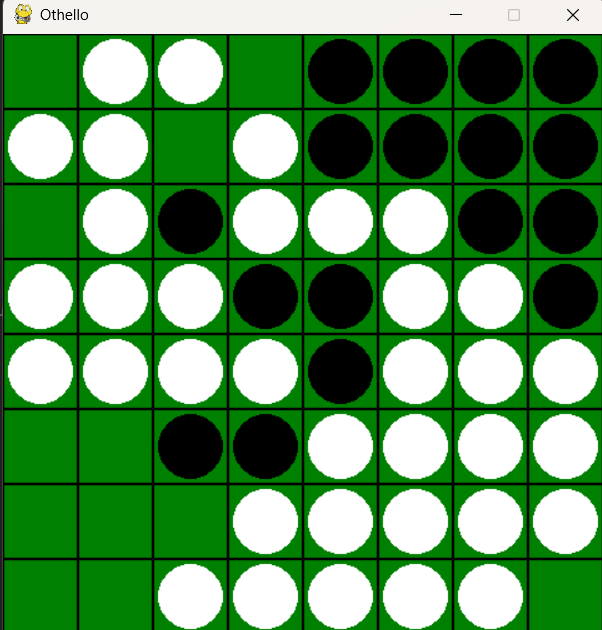
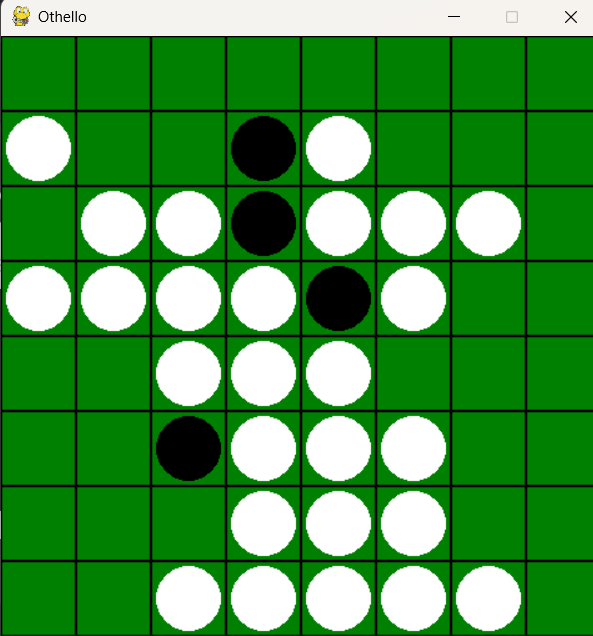
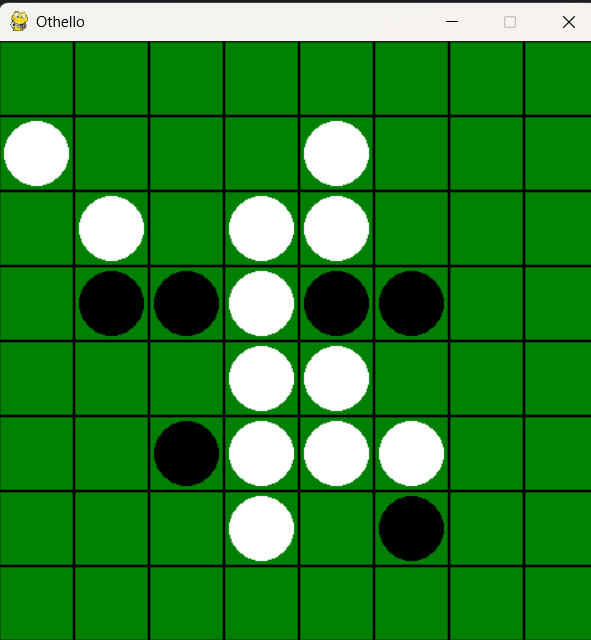
### 前期局面：



中期局面：



最后的画面没截下来，但是是黑赢了。保险起见，我们再来一把：



又是最后被黑棋两步占完边赢了。

我不禁在想：这到底是什么问题？

目前给我的选项有两种可能：一种是搜索深度导致的评估局面不准确，另一种是评估函数本身就存在问题。

当然这是在第二个问题下面，我们先看看是不是第一种的问题。（此工作放到了第三板块去做）

### 但是在做这个工作之前，我们还是要先看一下对战网上AI的测试：剪枝后的算法对战难度1，2都有碾压性优势，但是问题出现了：规则不一致，在我们的框架代码中，遇到了如果有一方无路可走的情况，那么直接计算输赢，但是网上的规则另一方继续走直到这一方有位置可以走棋。

这加深了我对第二种情况的猜想。

### 比较剪枝速度：我们还按照2.1.1的方法对搜索次数进行统计：从6层深度开始：

1.2983 2.13125 3.36488 4.107804 5.163729 6.430735 7.256634 8.362306 9.562260 10.708954 11.880267 12.1116463 13.1420543 14.1860345 15.2220457 16.2654067 17.3212859 18.3509456 19.3794762 20.3992342 21.4012978 22.4082526 23.4148440 24.4161004 25.白赢

（这次没有发生上面突然降智的问题，这下问题又不确定了）

可以看出相比于非剪枝的情况，游戏可以顺利完成，搜索量级降低了一个指数级。

### 那么此时我们可以向下继续探索深度，当深度为7时：对战AI第一级：

1.11630 2.41578 3.121918 4.257407 5.735574 6.1125560 7.2043800 8.2839390 9.3542517 10.4167981 11.4548279 12.4667421 13.4858598 14.4926054 15.5183090 16.5395372 17.5482138 18.5636432 19.5751280 20.5881136 21.5921546 22.5936089 23.5943853 24.5949841 25.5951077 26.5951492 27.5951528 28.黒赢

可以看到，深度为7时最后的总搜索次数与深度为6时的总搜索次数还在一个量级，这是因为到了中期的局面再往下模拟的话可落子的位置大多数都被剪枝剪掉了，因此数量级不会太爆炸。

可以看到计算的峰值基本都在中期，一手棋计算接近百万级的局面，此时计算时间在20秒左右，对于这个游戏来说，这是可以接受的。

接下来对战AI第二级，仍然赢了（很强啊），最终测试到第四关失败。

我们可以看到最多也就只需要下30步，因此理论上搜索深度开到30是一定可以寻找到最优解的，而我们肉眼观察基本上在10手左右就能够看出胜负手，因此我认为开到10以上的搜索深度就可以。

但是开到搜索深度为10的第一步就已经搜索了1036692次，并且这一手棋的位置与6，7步时不一样，因此我在此处作出这样一个猜测：这一步的结果是这个游戏的最优解。

## 进一步改进

### 查询有关这个游戏的论文，我发现一篇USTC本科生于07年发表在个人主页上的论文：<http://home.ustc.edu.cn/~baj/publications/concluding2007-Bai.pdf>

这篇论文提出了两种对alphabeta算法进行优化的方法：置换表和历史表：

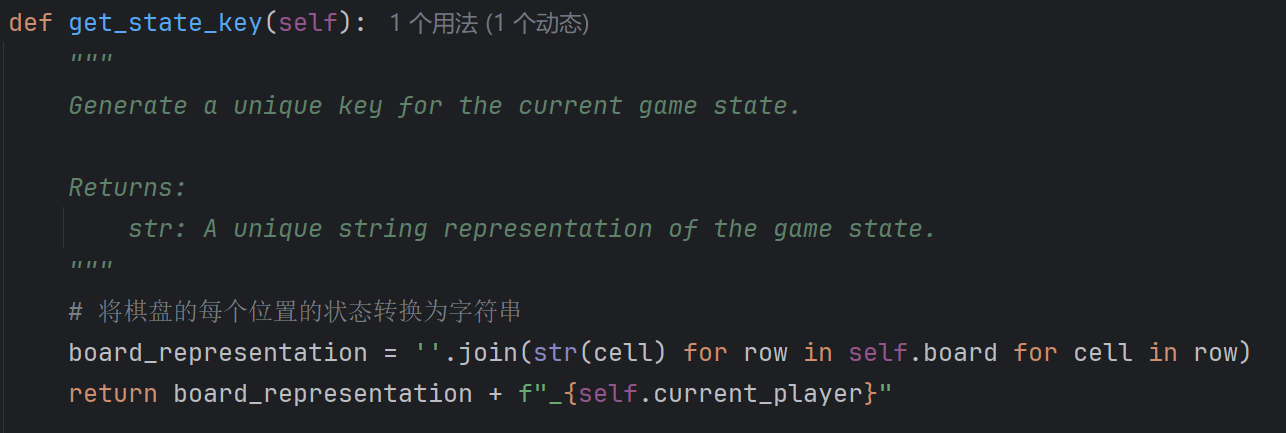
#### 置换表的基本概念：置换表通常使用一个哈希表（字典）来实现，查找：在进行 Alpha-Beta 搜索时，算法首先检查当前游戏状态是否已经存在于置换表中。如果存在，直接返回存储的评估值，避免重复计算。存储：如果当前状态不在置换表中，算法会计算该状态的评估值，并将其存储到置换表中，以便后续使用。更新：在搜索过程中，可能会遇到相同的游戏状态（例如，由于不同的移动顺序导致的状态重复）。在这种情况下，算法可以根据需要更新置换表中的值，以确保存储的是最佳评估值。

### 置换表的实现：



我们使用一个字典来作为置换表的实现，然后通过查找评估值来实现剪枝。

具体的哈希值实现方法：



将棋盘的每个位置（0、1、-1）串联成一个长字符串。通过将当前玩家信息（1或-1）附加到字符串的末尾，可以确保不同玩家的状态也有不同的键。这意味着相同的棋盘状态在两个不同玩家的轮次下会被视作不同的状态。

我们现在来比较一下搜索次数：

1.11656 2.46475 3.105354 4.259477 5.628318 6.1044145 7.1329606 8.2237515 9.3096456 10.3810406 11.5557107 12.7176063

可以看到相比于之前，在7手之后的搜索步数明显减少，这是由于很多局面在曾经已经被评估过，可以直接剪枝了。但是到了第11手位置，剪枝后的方案搜索次数反而要比剪枝之前更高，到了15手之后才恢复正常

#### 这是为什么？再次研究，我认为问题出在哈希函数的设计上：哈希函数我们目前使用棋盘状态+当前玩家，但是这并没有考虑到深度带来的影响。哈希函数仍需进一步改进。

### 我们现在来完成历史表的实现：

历史表记录了在博弈过程中每个动作的成功率和优先级。通过分析历史数据，算法可以优先考虑那些在过去表现良好的动作，从而提高搜索效率。记录表现：在每次搜索过程中，当算法评估一个动作时，会根据该动作的表现更新历史表。例如，如果某个动作导致了较好的结果（如成功剪枝或获得高评估值），则该动作的优先级会提高。优先搜索：在进行下一次搜索时，算法会根据历史表中的记录优先考虑表现良好的动作。这种优先级排序可以显著提高剪枝的效率，从而减少需要评估的节点数量。



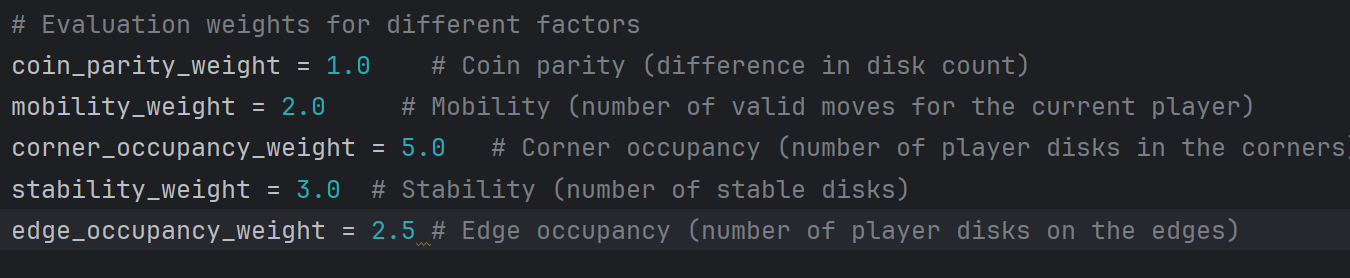
在每次搜索结束后，通过 update\_history\_table 函数更新历史表，记录当前最佳移动的表现。在获取合法移动后，使用 sort\_moves\_by\_history 函数根据历史表中的值对移动进行排序，以优先考虑表现好的移动。

进行测试……

历史表的运行时间较为夸张，来到了一步棋一分钟的时间，这是不可接受的，放弃此种方法。

# 改进evaluate\_game\_state函数

## 我们先来看看原版evaluate\_game\_state的评估方案：



使用了以下五个评估方法：

棋子差异权重: 1.0 这一部分计算当前玩家和对手之间的棋子数量差 (coin\_parity)。正值表示 AI 玩家拥有更多棋子，这是一种有利的状态。

行动能力权重: 2.0 行动能力通过计算当前玩家的有效行动数量，并减去对手的有效行动数量来评估。较高的行动能力表示更好的选择。

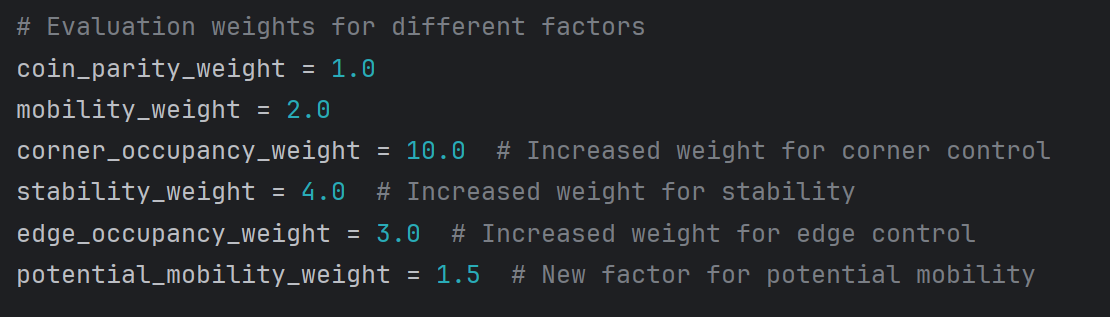
角落占据权重: 5.0 这部分计算当前玩家在角落的棋子数量。角落在 Othello 中非常重要，因为它们是稳定的位置，无法被翻转。

稳定性权重: 3.0 稳定性指的是稳定棋子的数量（无法被翻转的棋子）。calculate\_stability(game) 函数根据游戏规则计算这一点

边缘占据权重: 2.5 这部分累加计算棋盘边缘的棋子数量。边缘上的棋子也可以获得更稳定的位置，尽管它们的价值低于角落。

最终将这些值加和形成当前局面的得分。

对于改进方法，我现在也没有什么好的思路，查询论文：[1]我们发现，我们可以增加Potential Mobility这一项（潜在行动力），这表示与对手棋子相邻的空白位置数量，这个因素可以帮助更好地评估未来可能的行动机会。



首先修改参数，增加了角落的权重，因为根据这么多盘的总结，占住角落基本等于守下了一片区域。（参数先预设成这样，然后逐渐调整）



行动力的统计：我们来统计所有的可下位置，并返回个数，将其作为新增的参数。

详细的调整可以用线性回归的方法来学习，不过目前没有大量对局样本，自己对弈太耗时间，先放一下。

# MTD(f)算法

## MTD（f）是什么？

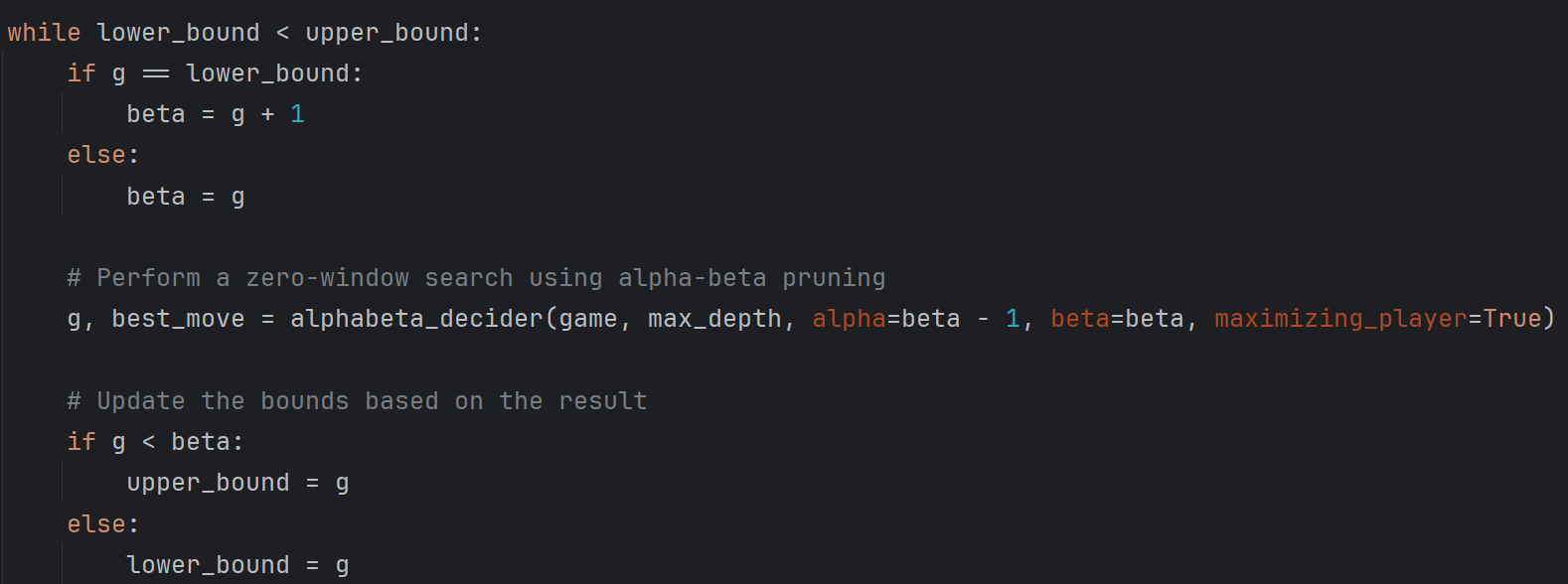
搜索结果：MTD(f)的核心思想是进行多次的零窗口搜索，每次更新估值直到找到局面的最优值。每次执行零窗口搜索时，算法都使用之前的返回值作为新的猜测值，从而逐渐收敛到最优解。这种方法的关键在于最初的猜测值应该接近真实值，因为MTD(f)的效率高度依赖于初始猜测的准确性。算法流程中，模型将依据返回的结果动态调整当前的上界和下界，直到两者相同，最终确定最优值。

读完这段话，我有两个概念不理解：什么时候“零窗口搜索”，如何收敛？

那么需要从“吸出搜索”这一概念开始逐渐理解：在进行搜索之前，算法会设定一个期望的评估值（通常是基于之前的搜索结果或启发式评估函数的输出），并定义一个小的窗口（例如，期望值±某个小值）。这就是窗口的概念。在搜索过程中，算法首先尝试在这个小窗口内找到最佳的移动。如果在这个范围内找到一个值，算法会进行剪枝，从而避免不必要的搜索。如果在小窗口内没有找到合适的值，算法会扩大搜索范围，重新进行搜索。这种动态调整窗口的方式可以有效地减少搜索的时间。

而对于零窗口来说：相比较宽的搜索窗口,零窗口可以产生更多的剪枝,可以通过置换表,使得之前的搜索不需重新进行,从而将多个零窗口的搜索粘合.

我们来看代码逻辑：



lower\_bound 和 upper\_bound 分别初始化为负无穷和正无穷，用于限制评估值的范围，g 被赋值为初始猜测值 guess。

只要下界小于上界，就继续进行搜索。β值的设定:如果当前评估值 g 等于下界，则将 beta 设为 g + 1，以避免搜索停滞。否则，将 beta 设为当前的评估值 g。

零窗口搜索: 调用 alphabeta\_decider 函数进行零窗口搜索，返回新的评估值 g 和最佳移动 best\_move。

alpha=beta - 1: 设置α值为β值减1，表示当前的最小可能值。

边界更新:如果新的评估值 g 小于当前的 beta，则更新上界为 g，表示可以缩小搜索范围。

否则，更新下界为 g，表示需要扩大搜索范围。

## 其与minimax算法的异同

### 相同点

博弈树搜索: 两者都使用博弈树的结构来评估不同的移动选择，通过递归地评估每个节点的值来决定最佳策略。

评估函数: 在两种算法中，评估函数用于评估叶节点的状态，以便在没有完全搜索整个树的情况下做出决策。

## 不同点

### Minimax即使经过剪枝，仍然需要遍历大量的节点，而mtd（f）可以通过使用零窗口搜索和转移表，能够在更小的搜索范围内进行多次搜索，从而实现更高效的剪枝。

### Minimax算法:在每次搜索中，通常不会重用之前的搜索结果，而MTD(f)算法:利用转移表存储已搜索节点的结果，能够在后续搜索中重用这些信息，从而提高效率。

[1]彭之军.计算机博弈算法在黑白棋中的应用[J].现代信息科技,2021,5(17):73-77+81.DOI:10.19850/j.cnki.2096-4706.2021.17.018.