黑白棋AI分享

2021310792 安頔

游戏规则

以下摘抄自百度百科，如果想快速上手游戏规则，建议前往https://www.saiblo.net/game/1/ranklist进行快速人机对局体验。

黑白棋的棋盘是一个有8*8方格的棋盘。下棋时将棋下在空格中间，而不是像围棋一样下在交叉点上。开始时在棋盘正中有两白两黑四个棋子交叉放置，黑棋总是先下子 。

下子的方法

把自己颜色的棋子放在棋盘的空格上，而当自己放下的棋子在横、竖、斜八个方向内有一个自己的棋子，则被夹在中间的全部翻转会成为自己的棋子。并且，只有在可以翻转棋子的地方才可以下子。

棋规

1．棋局开始时黑棋位于e4和d5，白棋位于d4和e5。

2．黑方先行，双方交替下棋。

3．一步合法的棋步包括：在一个空格新落下一个棋子，并且翻转对手一个或多个棋子。

4．新落下的棋子与棋盘上已有的同色棋子间，对方被夹住的所有棋子都要翻转过来。可以是横着夹，竖着夹，或是斜着夹。夹住的位置上必须全部是对手的棋子，不能有空格。

5．一步棋可以在数个方向上翻棋，任何被夹住的棋子都必须被翻转过来，棋手无权选择不去翻某个棋子。

6．除非至少翻转了对手的一个棋子，否则就不能落子。如果一方没有合法棋步，也就是说不管他下到哪里，都不能至少翻转对手的一个棋子，那他这一轮只能弃权，而由他的对手继续落子直到他有合法棋步可下。

7．如果一方至少有一步合法棋步可下，他就必须落子，不得弃权。

8．棋局持续下去，直到棋盘填满或者双方都无合法棋步可下。

胜负判定

当棋盘填满或者双方都无合法棋步下的时候，计算双方盘面上的棋子数，棋子多的一方获胜，如果棋子数相等则平局。

黑白棋基本概念

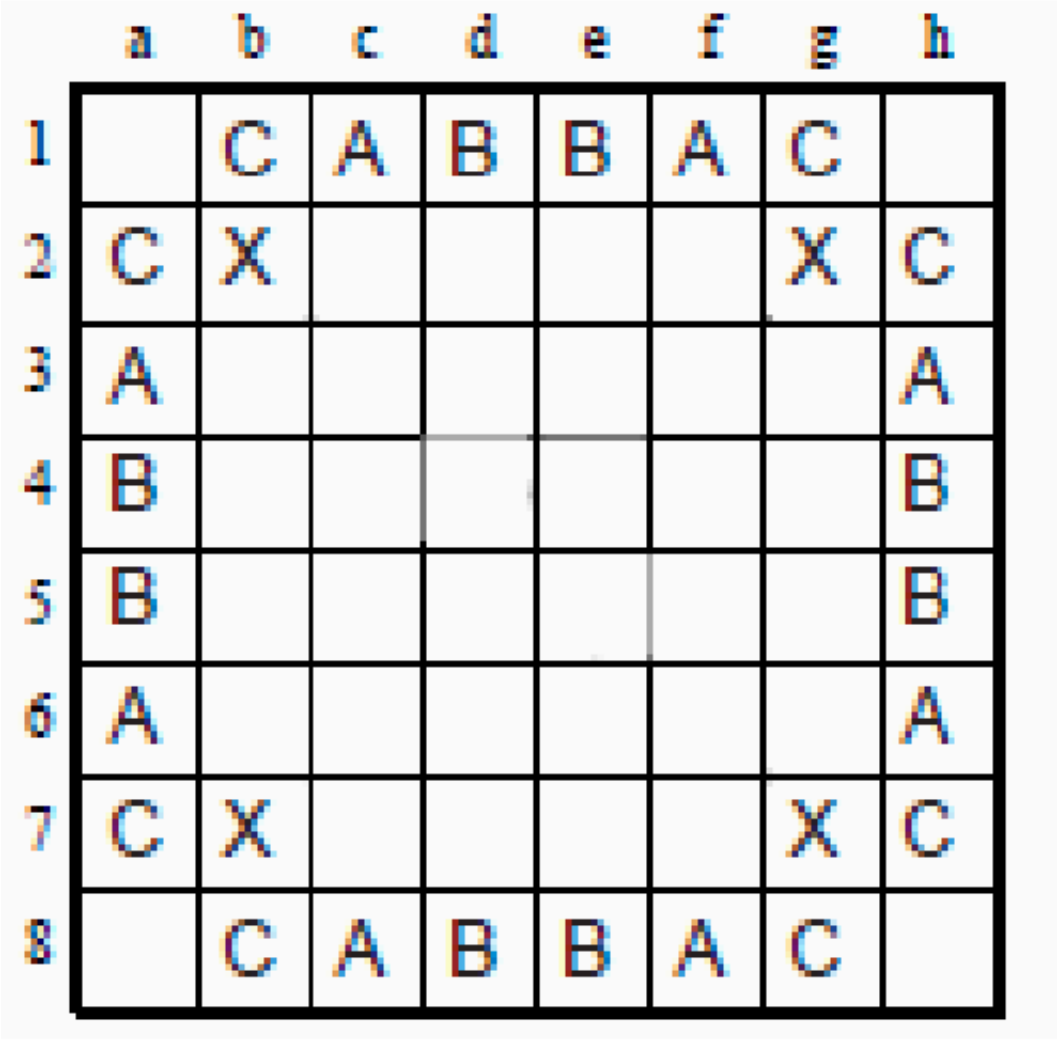
在黑白棋**人类选手**的对战中，有几个相对比较重要的概念；

稳定子

在之后的任何情况下都不会被翻转的棋。稳定子的判定在中后盘是一个相对复杂的问题，但是有一些位置可以非常简单的判定是否是稳定子，实际情况中往往用这些位置的判定来近似计算稳定子数目。

显然，棋盘的角部不论是哪方，都一定是稳定子。当有个棋子在行/列/左上-右下斜线/右上-左下斜线四条线上都至少相邻一个己方稳定子或者不可下的点(棋盘边界外)，那它必然也是稳定子。

根据以上理论可以发现，占到角部的玩家相对来说更有优势，因为可以依托角部的稳定子来逐步扩张自己的稳定子直到对局结束，因此黑白棋的一个主要玩法就是占角。

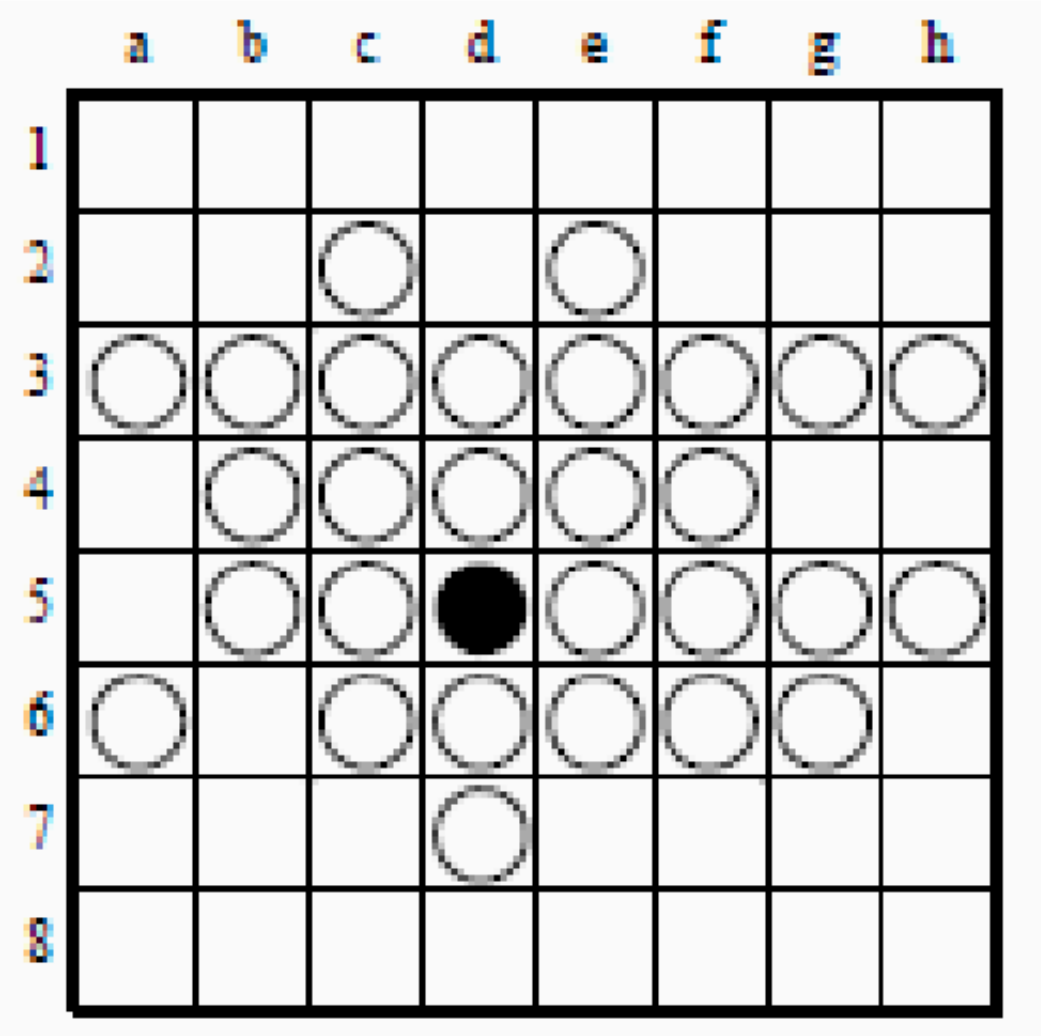


而己方能占角的前提是对方在图上的C点或者X点有棋子，所以在没有占角的时候，大部分情况下在C点或X点是比较差的下法。又因为在开始的时候，双方在对角线中间上有子，逐步扩张到边界，因此在前中期过早的下在X点比C点更危险，更容易让对手占角。

在纯随机走法的AI里加入对C点、X点和角部的判断，已经可以轻松打赢原来的纯随机AI了，但是这还远远不够。

行动力

在很多黑白棋新手的眼里，在棋盘上有很多子就等价于有优势，这是完全错误的。因为在棋盘上的子是随时有可能被翻转的，最终用来判定胜负的不是子的数量，而是稳定子的数量（因为只有当棋盘满了或者双方都没地方可以下的情况下才能开始胜负判定）思考一下，下面这个棋局里，是黑棋优势还是白棋优势呢?



既然我们希望能够有尽可能多的稳定子，按照上面关于稳定子的讨论，我们会发现我们的目标其实是占领角部然后扩张。如果你的对手也知道角部很重要，C点和X点不是比较好的选择这些相对简单的黑白棋道理，我们应该如何赢过他呢?

一个显然的想法是让他当前可以下的点只有X点或者C点，他就必须下在X点或C点，这样我们下一步就可以占领角部了。从这里我们很容易想到，如果定义行动力为某一方当前局面可以落子的点，那么行动力越低，可选择的点就越少，就越容易走出恶手葬送棋局。

所以在行棋过程中，通常情况下希望己方行动力变多，对方行动力变少，来尽量迫使对方走在对自己不利的位置上。

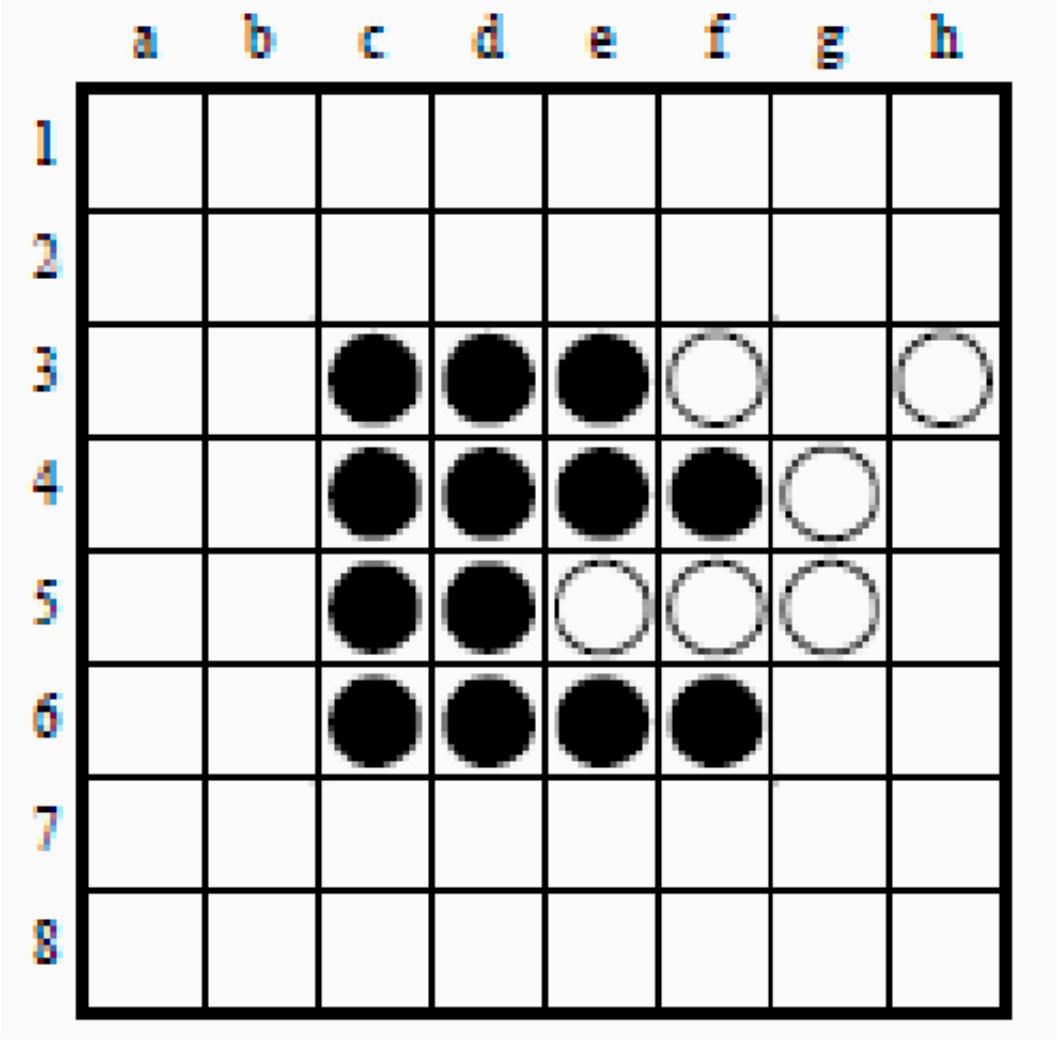
在上面的加了稳定子判断的AI中再加入对于行动力的判断，可以较为轻松的击败只有稳定子判断的AI了。另外，很容易想到，在对局的前中期，行动力相对比较重要，而在后期已经占领了角部之后，扩展稳定子相对更加重要了。

散度理论，凝聚手和发散手

一个棋子散度定义为它八个方向上空格子的数量，例如在**行动力**章节的图中，中间那个黑子的散度为0。如果一个子散度为0，在黑白棋中通常称为**内部子**，否则称为**边界子**。由相连的边界子形成的区域通常称为**墙**。根据行动力判断可知，在行棋过程中，**边界子**甚至**墙**都是相对不好的棋形，因为容易增加对手的行动力，同时降低自己的行动力。

因为在**人类对战**中，下若干步后的行动力变化是不好计算的，因此往往用棋盘上自己子的散度和来近似估计对方的行动力，用对方子的散度和近似估计自己的行动力，不难发现这个估计是永远大于等于实际的行动力的，是有一定偏差的，相对易于计算。采用这种估计的话，我们实际上就希望我们下的棋翻转的棋子的散度和尽可能的小。

在黑白棋中，通常把翻转的棋子散度和为0的棋步称为**凝聚手**，否则称为**发散手**，在大部分情况下，凝聚手都是好手。



以上图为例，白先，g3只反转了f4一颗黑子，翻转的黑子散度和为0，因此g3是步凝聚手。那么g3是不是个好手呢？我们会发现白棋下g3之后，黑棋只能下h2，相应的，白棋就可以下h1占领角部。

在之前的AI中加入对于凝聚手和发散手的判断，会发现AI的棋力并没有非常明显的提升。这是为什么呢？因为散度理论，凝聚手和发散手本质上还是最优化行动力，**人类对战**中用这套理论来简化和估计行动力的，而在AI中，因为算力足够来计算最真实的行动力，所以完全不需要用这套理论来进行估计。

奇偶性理论

奇偶性理论的思想是，对于棋盘上的一个闭合区域，我们总是希望可以在对方落子后，再落子将对方的子吃回。因此我们希望棋盘上的闭合区域大小尽可能都是偶数，这样我们总能落到闭合区域的最后一个点，结论则是我们应该尽可能的在大小为奇数的闭合区域内落子。实际对战中，会发现这个理论在残局处理的时候相当有效。

审视我们的开局，会发现开局的时候，棋盘上只有一片闭合区域，而且大小为偶数，从奇偶性理论来分析，应该是后手占据一定的优势。事实上也正是如此，在更小的棋盘下，可以穷尽所有可能的状态数来判断胜负，会发现白棋必胜。

然而在AlphaZero训练好的结果上看，8*8的黑白棋对战结果平均情况为32:32，也就是平局，白棋并没有特别明显的优势，这是为什么呢？这是因为黑白棋中存在弃权情况，也就是当一方没有合法棋步后，将跳过这一方的行动，让另一方下棋。一旦出现这种情况，奇偶性将被逆转。黑棋可以通过构造这种情况跳过自己的棋步或者对方的棋步来逆转奇偶获取优势（事实上，对于零和全知且确定性的棋盘游戏，一定存在黑棋必胜或者白棋必胜这样的结论的，这从侧面说明AlphaZero也远远没达到传说中神之一手的级别）

在残局中加入奇偶性判断，优先在奇数大小闭合区域上落子，会发现在残局表现上比之前的AI要好，也能够较为轻松的战胜之前的AI。

更多的黑白棋理论

在了解上面这些知识之后，已经足够你成为较为厉害的黑白棋棋手了。如果你希望学习更多的黑白棋知识，如不平衡边、锚点、四通陷阱等等，推荐阅读Brian Rose的《黑白棋指南》。但是由于那些知识与具体的棋形相关，是人类在长期对战中总结出的经验，用来编写AI一方面提升并不显著，另一方面代码量和代码难度也会明显上升，在这里也就不多做介绍。

估值函数

简单的估值函数

通过以上理论的结合，可以对当前棋盘或者某一手的决策进行价值估计。

例如，对于一个局面S，定义稳定子数目为x，行动力为y，估值函数为$value(S)$，$value(S)=\alpha x-\beta y$，其中$\alpha,\beta$是人为设定的参数，通常情况下随着棋盘上子数的增多，α逐渐变大，β逐渐变小。

对于局面S的一个决策点A，估值函数定义为$v(A|S)$，假设下在决策点A后局面变为S'，可以让$v(A|S)=v(S')-v(S)$，这样可以获得每个决策的价值，每次贪心选取最高价值的决策作为当前决策。如果想加入奇偶性的判断，修改$v(A|S)$的定义式即可。

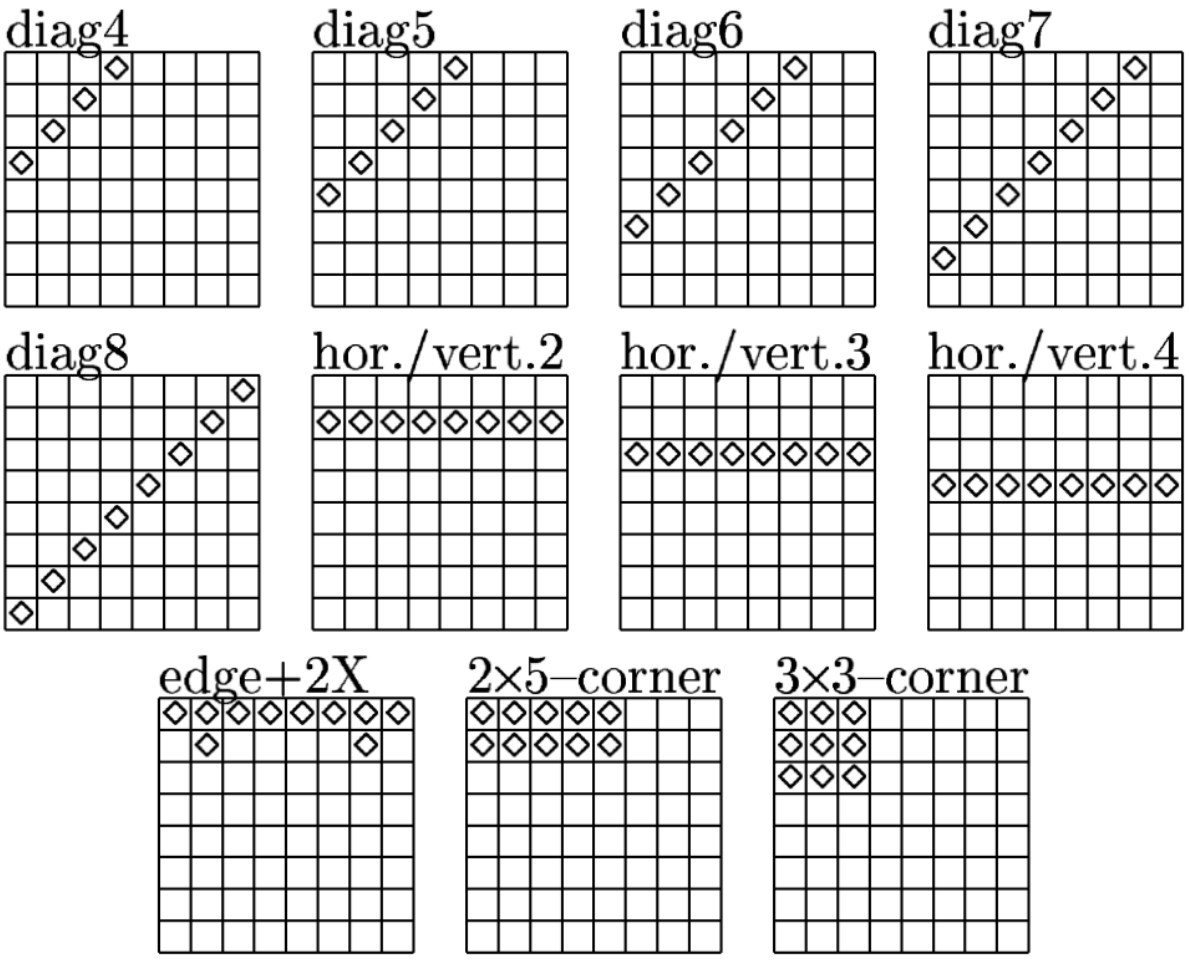
如果只判断一步棋直接贪心，可以轻松战胜随机走子的AI。搭配上搜索(后面会详细阐述)就已经有相当不错的棋力了，这也就是90年代战胜国际象棋世界冠军的“深蓝”的主要思路(估价函数+搜索)。

但是这个AI思路存在一些严重的问题，第一个是非常依赖人类智慧，需要人类从对战经验中总结棋盘特征，如稳定子、行动力等等，并且调整α,β等参数，如果估计偏差较大，AI的强度就会很低。

第二个是无法保证偏差的收敛，以黑白棋为例，从局面S走一步定义为S_1，走两步定义为S_2，依次类推，对于任意一个局面S我们有估值函数$v(S)$，假设其真实值为$v_{real}(S)$，偏差定义为$|v(S)-v_{real}(S)|$，偏差收敛则需要满足$|v(S_m)-v_{real}(S_m)|\leq|v(S_n)-v_{real}(S_n)|(m>n)$，如果不能保证偏差的收敛，则搜索之后若干步得到的最优策略未必会优于对当前步进行贪心，我们的估值函数难以通过搜索算法进一步提升AI实力。

模板估值函数

90年代，Logistello(一个黑白棋AI)异军突起，以压倒性的优势击败了其他黑白棋AI，并且战胜人类世界冠军。其主要思想是不再通过人类智慧手工调整参数，而是通过定义一些模板，让程序自己学习参数来调优。



上图是Logistello用来估值的主要模板(Logistello后面又加入了一些角部模板来强化角部的估值)，不妨按顺序用$p_1, p_2, \ldots\ldots, p_{11}$依次代表上述模板，这些模板对应的棋盘状态可以用一个3进制数字来描述(例如0表示黑子,1表示白子,2表示空)，以p_{11}为例，因为模板里有9个点，因为可以用小于3^9的3进制数字来描述，也就是这个模板最多有3^9种状态。

对于模板p_i，状态为s_i的情况我们可以定义一个权重w_{p_i,s_i}，每个模板可能并不只对应棋盘的一个区域，以p_{11}为例，左上，右上，左下，右下四个区域都可以同样用这个模板的权重进行计算，再比如p_7，对应第三行，第六行以及原棋盘转置后的的第三行，第六行四个区域。设第i个模板对应着m_i个区域，第i个模板第j个区域的状态是s_{ij}，权重为$w_{i,s_{ij}}$。这样表示的好处是可以充分提取棋盘上的信息，同时利用棋盘对称性共享权重，减少了参数数量。

整个棋盘状态S的估值定义为所有模板对应区域权重的线性相加，即

$$v(S) = \Sigma_{i=1}^{11} \Sigma_{j=1}^{m_i} w_{i,s_{ij}}$$

我们一共有$\Sigma_{i=1}^{11} 3^{|p_i|}$个权重$w$需要确定，其中$|p_i|$是模板$p_i$覆盖的棋子数。这么多的权重参数显然不能人工一个个思考和设定，我们需要一些更为巧妙的思路。

考虑当棋盘上接近64 (比如60, 61, 62, 63) 个子的时候，双方都按照最优策略走的结果是可以通过暴力搜索得到的 (如果这里不理解最优策略, 可以阅读后面搜索部分的搜索树一节)。不妨假设我们是黑棋，可以定义$v_{real}(S)$就是最优策略到达终局时候黑棋数量减去白棋数量，注意因为暴力搜索到了终局状态，因此得到的是真实值而非估值。

我们可以随机生成足够多的60，61，62，63个子的局面，并且搜索获得对应的v_{real}值，带入之前的线性相加的方程里可以得到一个线性方程组，其中w_{p_i,s_i}为求解量。由于权重参数w的数量过于庞大，难以用时间复杂度为$O(n^3)$的高斯消元法进行求解。但由于该线性方程组的系数矩阵是稀疏矩阵，存在更快的数值求解方案，可以用高斯-塞德尔迭代法、牛顿迭代法等方法进行快速求解。实际实践中考虑到数值稳定性问题，同时可以近乎无限采样生成方程式的特点，采用了简单的梯度下降方法迭代求解。

具体做法为，每次随机生成一个局面S，暴力搜索得到$v_{real}(S)$，使用当前权重参数计算$v(S)$，定义损失函数为$\frac{1}{2}(v_{real}(S)-v(S))^2$，可以得到损失函数对每个参与计算的权重参数$w$的偏导为$v(S)-v_{real}(S)$，更新参与计算的权重参数为

$$w'_{i,s_{ij}}=w_{i,s_{ij}}-(v(S)-v_{real}(S))*lr$$

其中$w'_{i,s_{ij}}$是更新后的参数，lr表示学习率，实际实现中取$1e-3$。

这样迭代若干轮之后，参数权重的值会逐渐收敛。尽管我们有几十万的参数权重，但是要解的线性方程组有近乎无穷多的方程数目且规律性不明显，因此用得到的结果估计仍然是有偏的。实际实现中随机生成了若干局60，61，62，63个子的局面，使用训练好的参数权重进行训练，计算损失函数$L_1=\frac{1}{2}(v_{real}(S)-v(S))^2$和$L_2=|v_real(S)-v(S)|$，得到$L_1$平均为11左右，$L_2$平均为3.8。换言之，在最多还差4个子终局的时候，用这种估值方案估值出来的平均子力偏差只有3.8，这已经是一个非常不错的估值了。

之后我们逆向迭代该过程，假设以60-63个子的局面估计为真实值，用同样的方法可以计算56-59个子的局面估计参数权重，然后再以56-59个子的局面估计为真实值，计算52-55个子的局面估计参数权重，反复迭代下去，可以得到在对应棋盘各种状态的参数权重。为什么不直接去求56-59，52-55……这些局面的真实值呢？这是因为真实值需要我们搜索到最终局面才能确定，而随着搜索深度的增加，搜索的时间开销将指数级增长，会使得生成局面数据和训练的过程的用时大大增加，甚至难以忍受。

实际训练细节

在实际训练中，可以认为之前得到的参数权重已经包含了某些局部特征的相对准确的估值，后面逆向迭代的过程实际上是对这个参数权重的finetune，一方面使参数权重更适应当前子数情况的估价，另一方面可以补充一些在之前模型中没有出现过的模板状态的权重。

基于这个想法，可以先对60-63个子的情况做大量数据的训练 (实际训练中使用了5亿盘局面数据)，采用较高的学习率 (实际训练中取值1e-3)。然后在之后56-59，52-55……d的训练中，使用上一次的模型进行权重初始化，采用相对较少的数据 (实际训练中每一轮使用了5kw盘局面数据)，并采用相对较低的学习率 (实际训练中取值5e-5)。

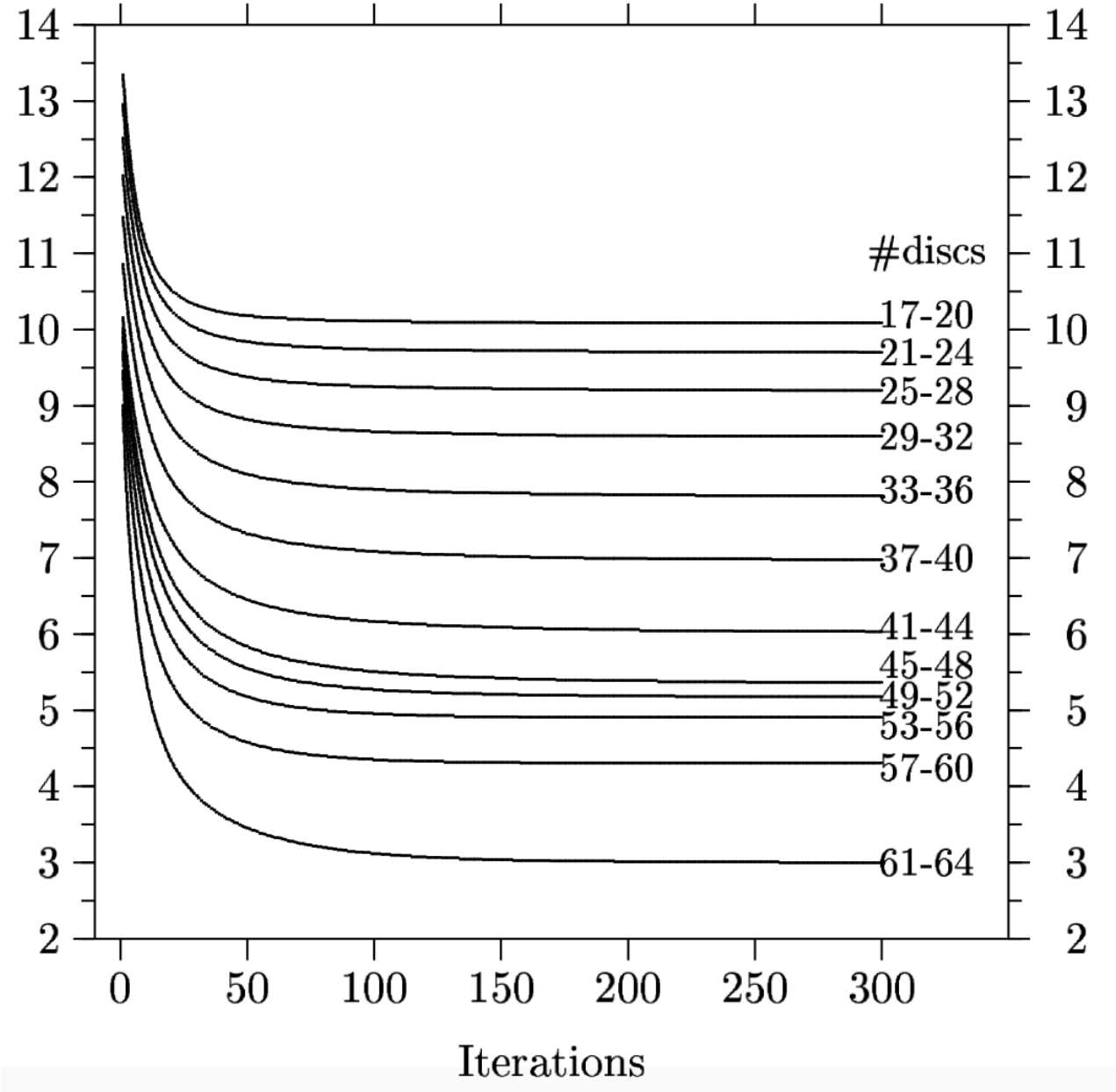
训练本身并不是非常吃算力，但是随机生成局面S并搜索得到$v_{real}(S)$是相对比较慢的过程。为了加速训练，以训练56-59这个阶段为例，每次随机走子生成有56个子的棋盘状态，暴力搜索这个棋盘状态的$v_{real}(S)$，并将搜索过程中每个棋盘状态都视为训练数据进行训练，这样可以有效扩充训练数据，大大提高训练速度，但缺点是会使得训练数据之间存在关联性，训练达到收敛的效果可能会有所下降，同时会更容易使得训练过程中损失函数产生震荡现象。

另外，考虑黑白棋棋盘的对称性，很容易发现对棋盘进行左右上下翻转，转置，转置+左右上下翻转得到的棋盘情况跟原始棋盘完全一样的。这里需要注意的是，因为黑白棋初始棋盘有四个子，初始棋盘的状态在左右翻转和上下翻转后情况和原始棋盘并不一致，所以并不能使用上下和左右翻转进行数据增强。在实际训练过程中，对于每个棋盘，可以同时用原棋盘、左右上下翻转棋盘，转置棋盘，转置+左右上下翻转四种棋盘状态作为数据，提升了训练数据数目，同时在一定程度上优化了训练结果的收敛速度和稳定性。

训练结果

在每个阶段训练结束后，随机生成对应阶段的若干个棋盘状态，搜索得到真实值，计算训练模型得到的结果和真实值的损失函数。

以上一个阶段的估价作为真实值，当前阶段的L_2损失为稳定在10-13，L_1损失稳定在$3.5-4.5$，每个阶段的训练结果都是类似情况。相比下图中Logistello原论文展示的训练结果 (每一组discs纵轴展示的是以上一组作为真实值的L_1损失函数)更加优秀，这是因为相比90年代，现代硬件的算力有了非常夸张的进步，允许使用更多的训练数据和更有效的训练方法。



这样我们就得到了若干组对应棋盘不同子数的参数权重，这个估值基于数据驱动，相比人类智慧的估值更加准确，并且从训练的整个过程中，可以发现估值偏差是具有收敛性的，搜索深度越深得到的估值就会越准确。

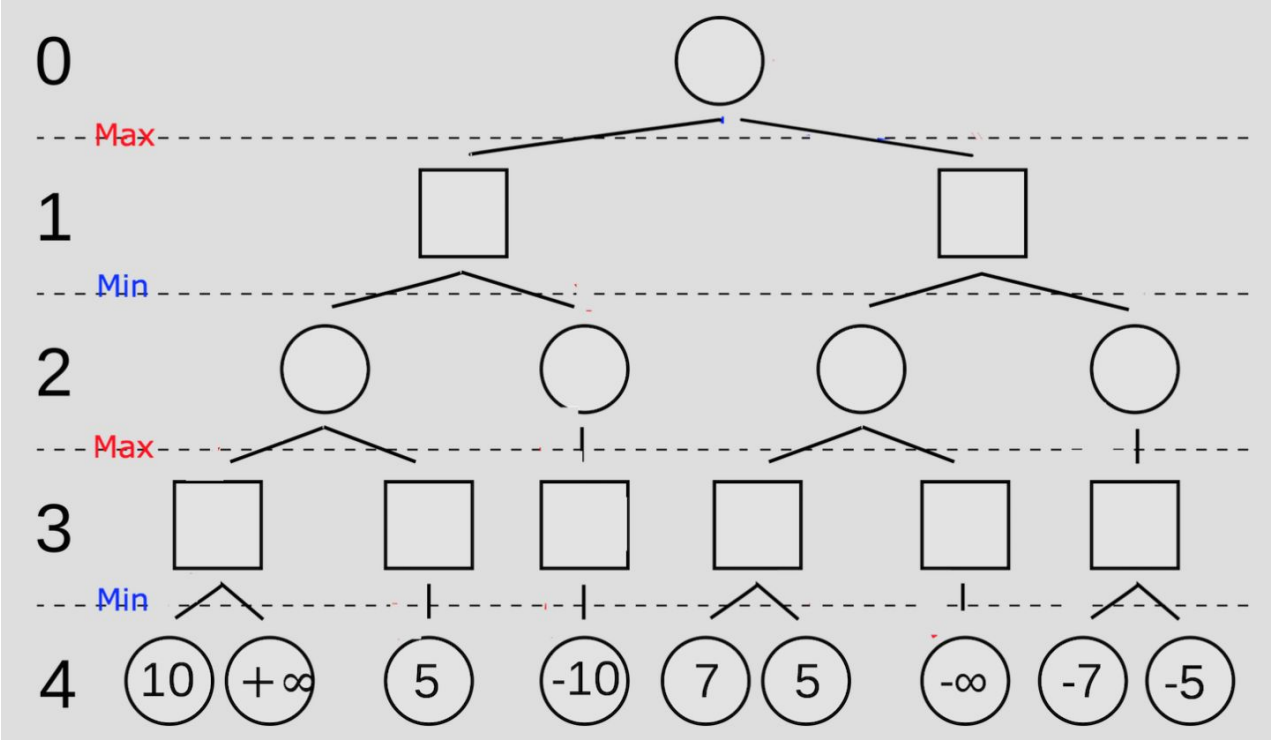
极大极小搜索及其相关优化

搜索树

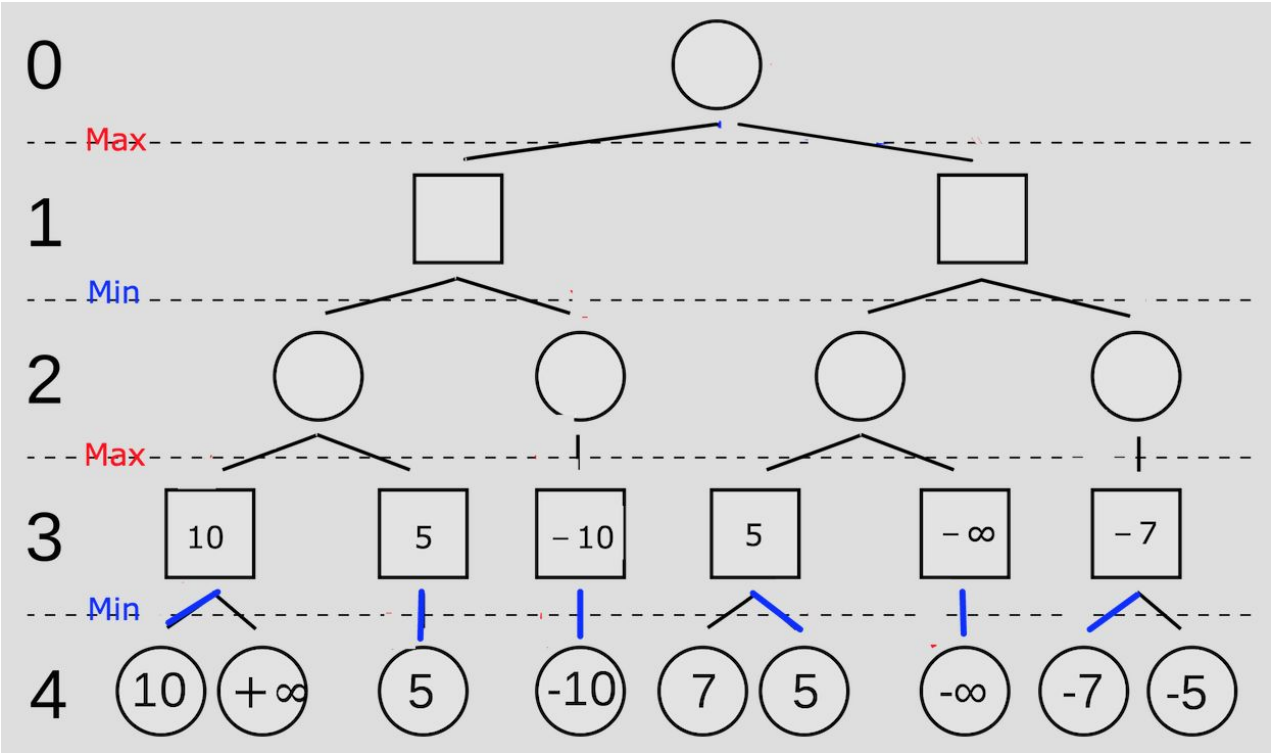
在讨论极大极小搜索前，首先要对搜索树有一个相对直观的认识。对于任何一个状态，当前下棋方会有若干种决策，在黑白棋游戏中，每个决策会导向一个新的状态，如果我们把每个状态看成一个节点，决策看成边，因为在黑白棋中，棋盘棋子数量是单调递增的，因此状态不可能循环。所以整个黑白棋游戏可以看成一个以初始局面状态为根节点，以结束状态为叶节点的超大树型结构，通常称之为搜索树。

极大极小搜索

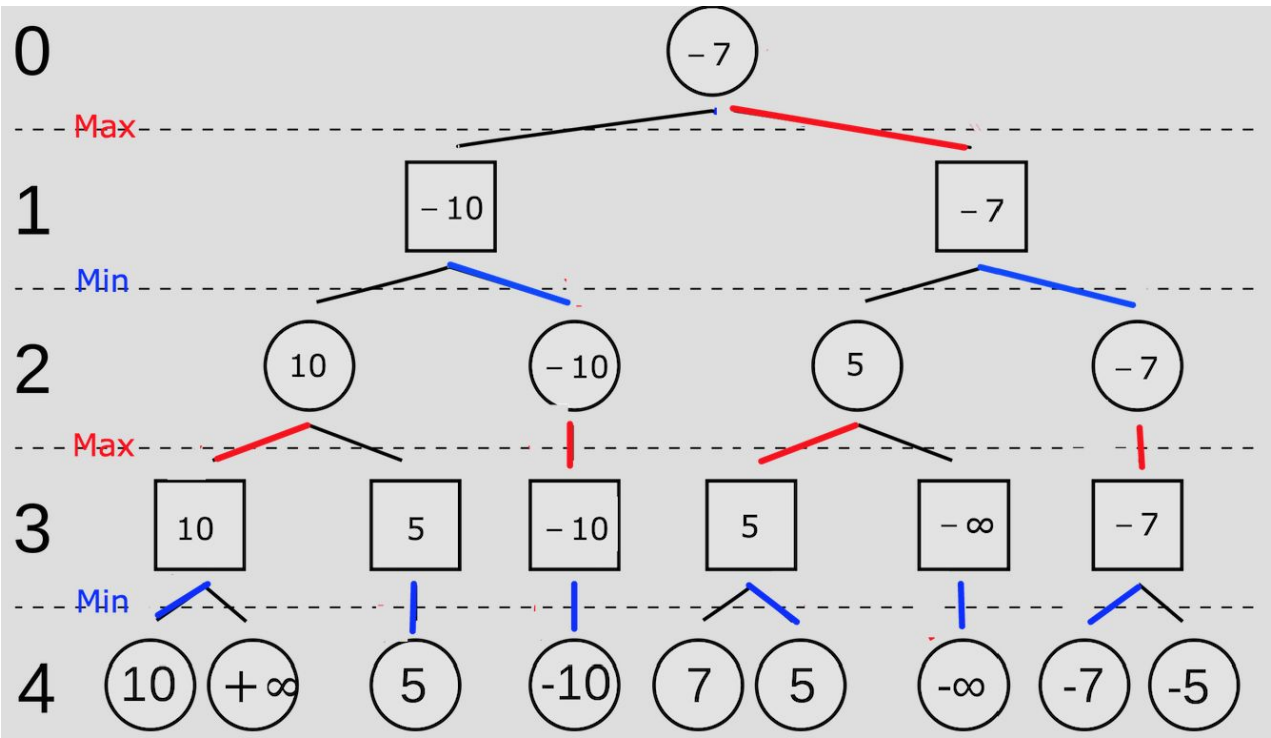
黑白棋的搜索树节点规模大约是10^{28}，是难以完全搜索展开的。为了方便讨论，我们以下图这棵相对简单的搜索树为例，其中偶数层为先手行动回合，奇数层为后手行动回合，第四层为叶节点，也就是棋盘终止状态，叶节点上的数字是**先手方视角下**的对局结果。那么，在最开始，先手方应该采用哪个决策呢?



不妨从树的底层开始思考，比如当前是第三层的第一个节点，后手方行动，肯定希望结果越小越好，因此会采用结果为10的决策，同理，我们可以得到第三层的节点值依次为10，5，-10，5，$-\infty$，-7，如下图所示



第二层该先手方行动了，肯定希望行动后的结果越大越好，即选取后继节点值较大的那个。这样可以一路倒退到根节点，得到如下图结果



那么当前状态即根节点情况下，先手方应选取右边的那个决策方案。这也极大极小搜索的主要思想，即某一方优先选择结果最大的决策，而对手会优先选择结果最小的决策，**注意这里的结果指的是某一方的视角**。之前的模板估值对于v_{real}的搜索也正是采用这种极大极小搜索思想。

根据这种极大极小搜索思想，我们很容易意识到，对于双人零和全知且决策转移固定的游戏 (如围棋、象棋、国际象棋、黑白棋、五子棋等)，是一定有先手/后手必胜的结论的。AlphaZero在这些游戏上自我先后手对弈的结果并不能展现出某一方碾压性的优势，说明AlphaZero还远远没有达到所谓的“围棋上帝”的地步，人类在博弈游戏上的研究仍然任重道远。

因为大部分情况下，我们不可能将搜索树完全展开，拓展到真实的终局状态，也就是叶节点，因此实际实现中经常会限制搜索深度，当达到搜索深度后使用估值函数来近似代替真实值，因此极大极小搜索的AI实力非常依赖于估值函数的准确性和偏差收敛性。

Alpha-Beta剪枝

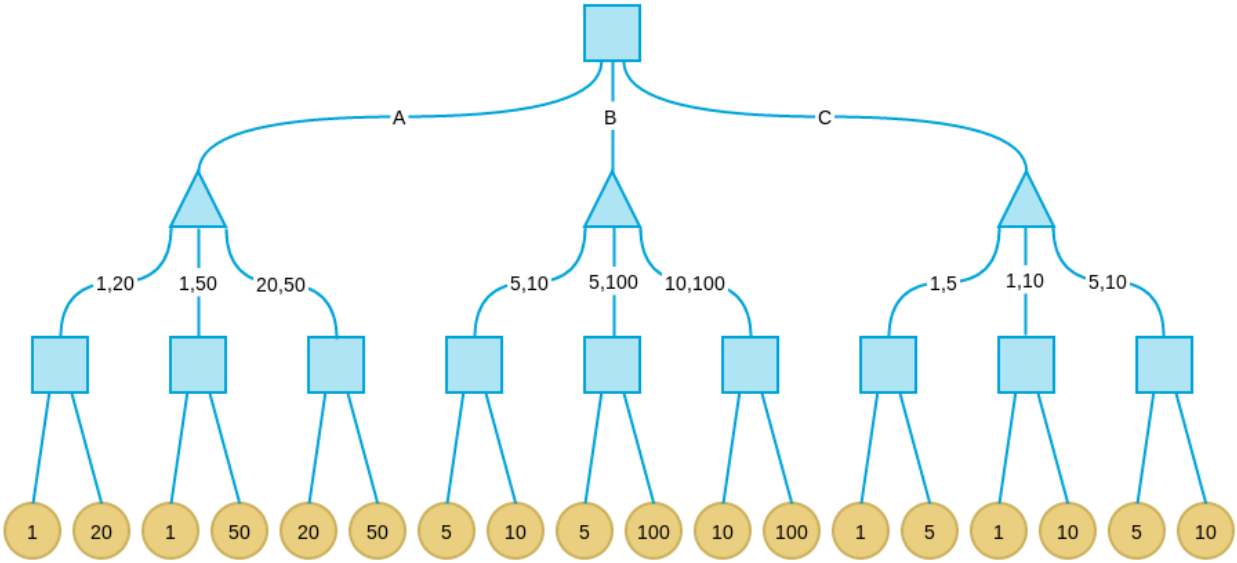
为了便于讲解Alpha-Beta剪枝算法，我们先考虑一个比较简单的游戏：

有三个盘子A、B和C，每个盘子分别放有三张纸币。A放的是1、20、50，B放的是5、10、100，C放的是1、5、20。单位均为“元”。有甲、乙两人，两人均对三个盘子和上面放置的纸币有可以任意查看。游戏分三步：

1. 甲从三个盘子中选取一个。
2. 乙从甲选取的盘子中拿出两张纸币交给甲。
3. 甲从乙所给的两张纸币中选取一张，拿走。

其中甲的目标是最后拿到的纸币面值尽量大，乙的目标是让甲最后拿到的纸币面值尽量小。

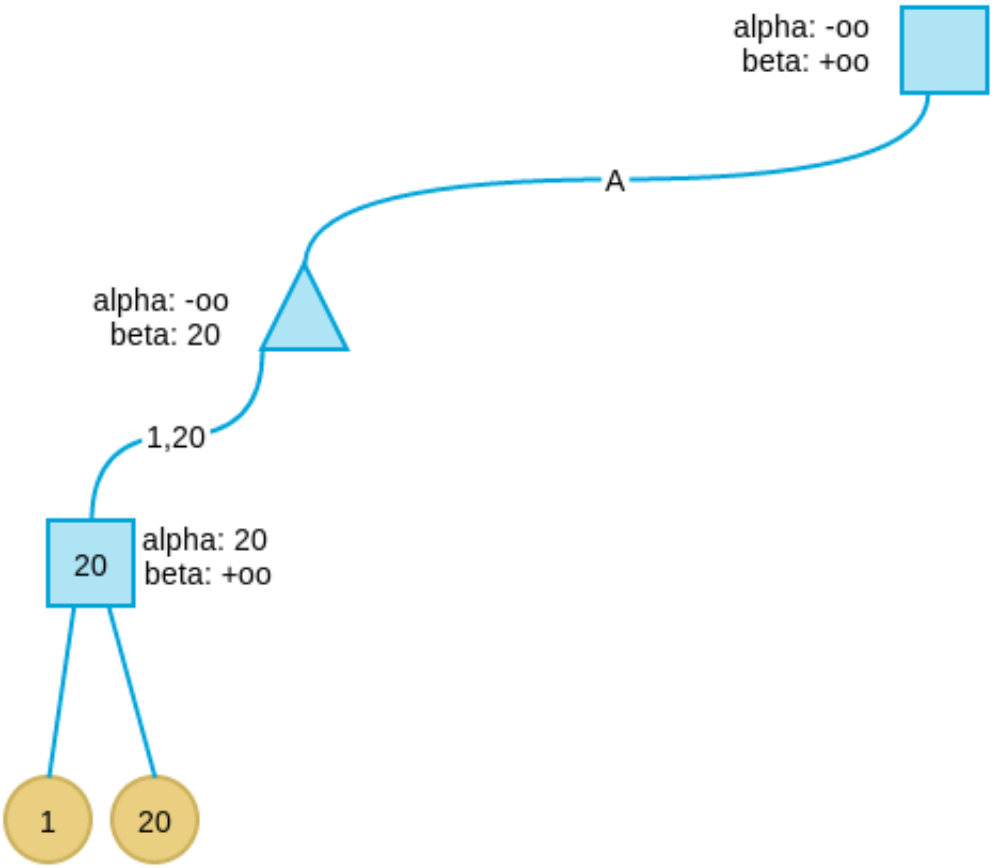
因为整个游戏的状态空间和决策空间都比较小，我们可以得到如下的一棵搜索树



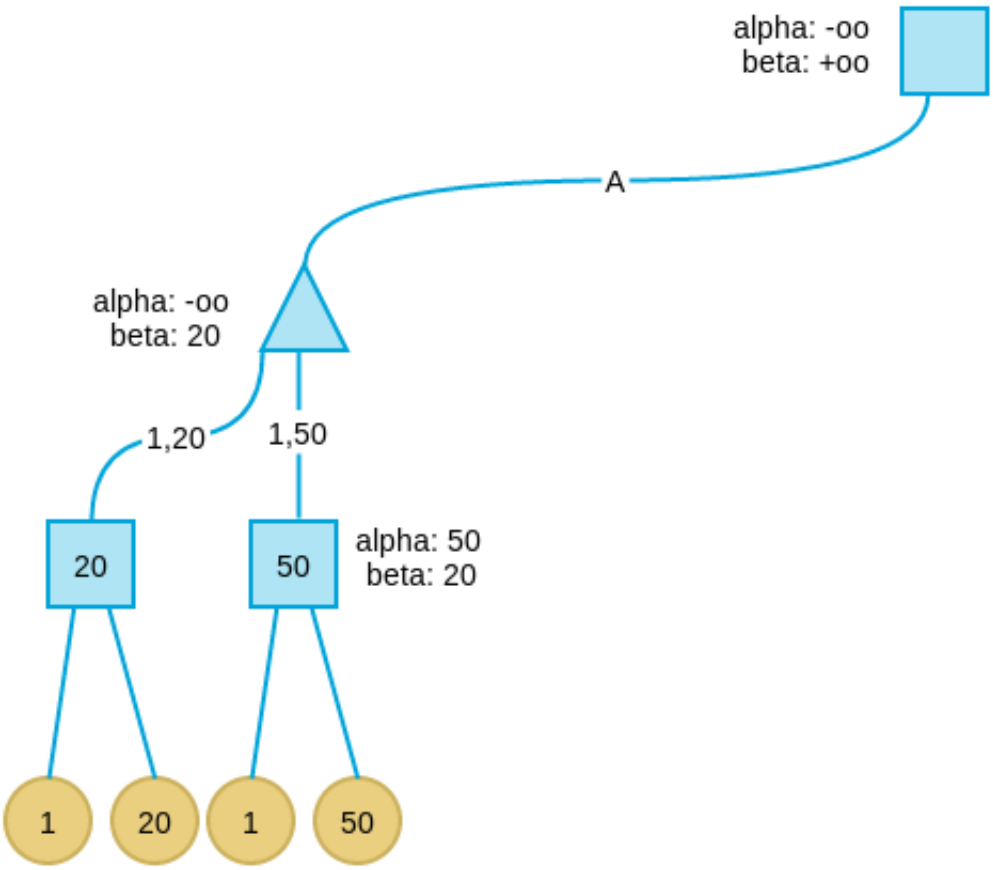
如果采用朴素的极大极小搜索，我们需要扩展出31个状态节点才能得到一个最优决策结果。

Alpha-Beta剪枝的思路是在深度优先搜索过程中，给每个节点存储Alpha值和Beta值，其中Alpha表示这个节点取值的下界，Beta表示这个节点取值的上界，初始时分别为$-\infty, \infty$。在向下拓展过程中，拓展节点继承父节点的Alpha和Beta值，当搜索到指定深度或者终止局面时回溯，对于MAX节点 (即选取后继节点最大值的节点) 更新Alpha值，对于MIN节点 (即选取后继节点最小值的节点) 更新Beta值，如果有一个节点出现了$Alpha \geq Beta$的情况，则这个节点不会是上层节点的最优选择，可以不进行后续的搜索拓展。

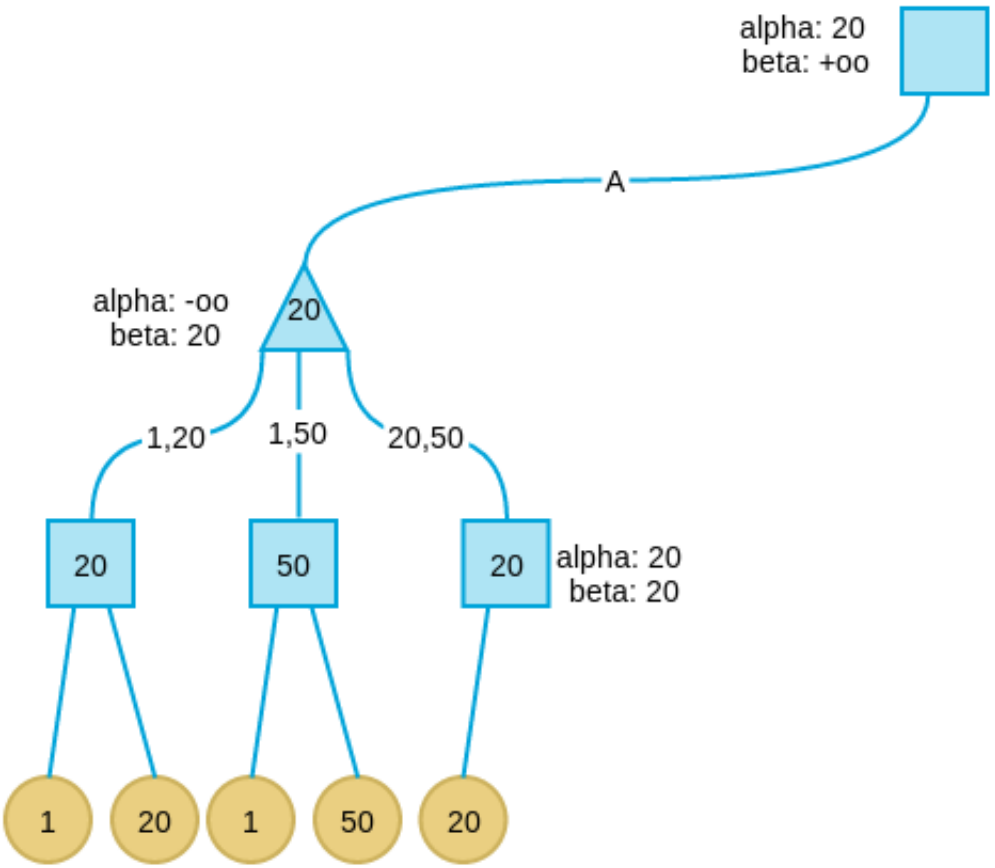
对上图使用深度优先搜索，会先得到下图的拓展结果。请注意偶数层节点希望结果越大越好，奇数层节点希望结果越小越好 (层数从零开始计数)，按照极大极小搜索思想更新节点的上下界，如果不是很能理解，请务必按照下图的情况手推一下。



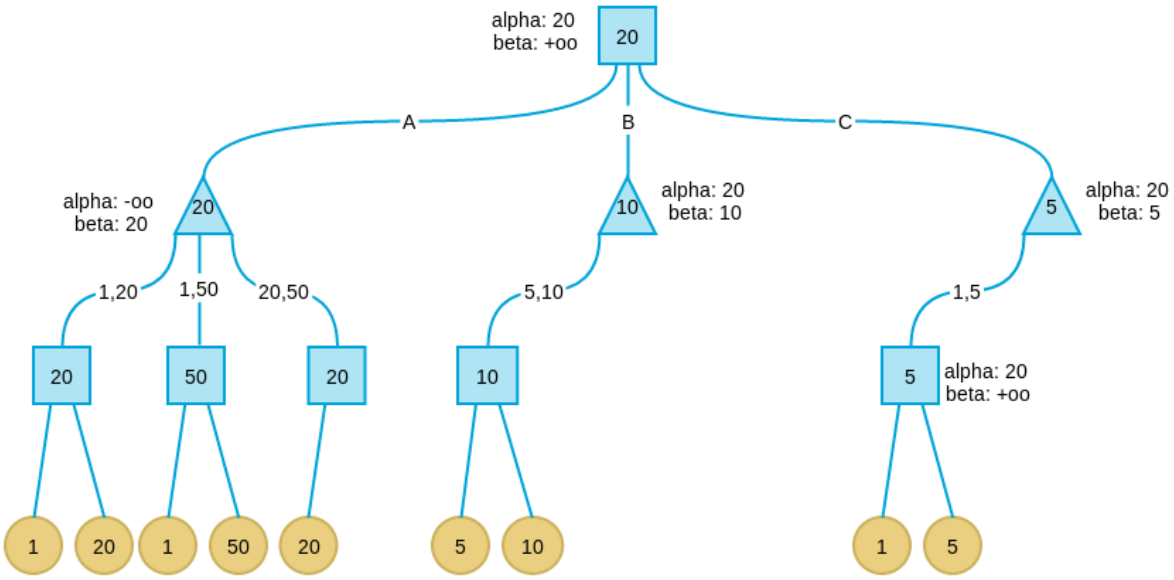
继续进行搜索，可以得到如下图拓展结果。注意拓展节点要继承父节点的Alpha和Beta，并在此基础上进行更新。



接着搜索，拓展结果如下图，注意此时第二层的第三个节点出现了$Alpha \geq Beta$，它对第一层的节点更新不可能有任何贡献，这是因为不论第二层第三个节点如何继续搜索，其下界至少是20了，而第一层的第一个节点根据极大极小搜索思想，应该取后继节点中的下界的最小值，在搜索过程中这个值已经是20了，因此第二层第三个节点必定不会影响上层节点的搜索结果了，也就没有必要继续扩展了。



按照这个剪枝思想，最终搜索过程整体如下：



后两个搜索分支是如何更新的，以及为什么可以剪枝，如果对Alpha-Beta剪枝还不是很理解，建议手推一下。最后我们会发现甲选取A盘是最优的，相比朴素的极大极小搜索，即Alpha-Beta剪枝只拓展了18个节点，大大减少了搜索计算量。将该剪枝方法推广到黑白棋也是同样适用的，实际实践中黑白棋中盘朴素的极大极小搜索只能搜4-5步，加入Alpha-Beta剪枝后可以搜6-7步，优化非常显著。

注意Alpha-Beta剪枝是无损剪枝，不影响搜索的正确性。在相同搜索深度下，加Alpha-Beta剪枝和不加Alpha-Beta剪枝得到的极大极小搜索结果是一样的。

决策排序优化

分析Alpha-Beta剪枝过程会发现，每次剪枝实际上是拿父节点搜索得到的部分解和当前节点搜索得到的部分解进行比较然后剪枝。搜索得到的部分解越接近于实际解，Alpha-Beta剪枝发生的概率就会越高。这也就意味着，Alpha-Beta剪枝的效率和每一层的搜索顺序是相关的，我们可以按照一定顺序搜索，使得我们尽可能快的找到接近于实际解的部分解。

考虑到我们对于每个棋盘状态都有个估值，一个简单的排序方案是：每个节点向下拓展的时候，如果这个节点需要取后继节点的MAX，我们按照后续棋盘状态估值从大到小进行拓展；如果这个节点需要取后继节点的MIN，则按照棋盘状态估值从小到大进行拓展。

在加上这个决策排序优化后，AI比之前朴素的Alpha-Beta剪枝的搜索深度又多了一层，实际实践中中盘可以搜7-8步。

Negamax

传统的极大极小搜索+Alpha-Beta剪枝需要对每个节点记录Alpha和Beta值，而且需要区分MAX节点和MIN节点，在调试和代码编写上不是很直观，也会增加比较多的分支跳转语句，影响程序性能。

Negamax的想法是，既然要交替取MAX和MIN，如果我每次都对后继节点取负，那么我每个节点的操作实际上就都变成了对后继节点取MAX操作（如果不理解的话，可以简单画个搜索树手推一下）。由于没有了MIN操作，每个节点也不需要保留(Alpha, Beta)这个上下界窗口，只需要保留一个当前部分搜索的最大值即可。跟Alpha-Beta剪枝思想一样，这种情况下也可以根据节点和父节点保留的部分搜索最大值来进行剪枝。

注意Negamax只是极大极小搜索一种相对更为简洁的表达方式，并不会影响极大极小搜索本身的正确性，也不会减少或者增多搜索拓展的节点数目。

迭代加深搜索

Saiblo上的黑白棋AI是限制时间的，每一步限时3s，如果在3s内AI没有做出决策，则直接超时判负。如果在极大极小搜索中采用固定的搜索深度，由于黑白棋中盘局面变化相对复杂，容易搜索超时，而残局的时候变化又相对简单，可能面临搜索深度不够深，时间利用不充分的问题。

这可以根据棋盘上子的数量来人为设置每种情况的搜索深度，但是一方面人为设置的深度有可能不够深，导致AI没有完全利用计算资源，另一方面这种人为设置需要固定评测机的性能指标，如果评测机本身发生了变化 (比如换了台更慢的评测机)，原来的设置就会导致超时了。

我们不难发现极大极小搜索的节点数是指数级扩张的，假设每个节点只有n种决策，对应不同的后继局面，那么搜索深度为k的朴素极大极小搜索拓展节点数为n^k，搜索深度为1，2，3，.....，k-1层的总扩展节点数为$n^1+n^2+n^3+.....+n^{k-1}$，当$n\geq 2$的时候，有$n^1+n^2+n^3+.....+n^{k-1}\leq n^k$。

这也就意味着，我们可以从一个较低的深度开始搜索，逐步增加搜索深度并更新最优决策，利用计时函数来判断是否快要超时，如果快要超时，则停止当前深度的搜索，使用上一个深度的搜索结果作为决策，这样的算法叫做迭代加深搜索。

迭代加深搜索仅仅只是增加了一点微不足道的常数，但是可以在时间限制下尽可能的压榨算力，在加入迭代加深搜索后，尽管决策复杂的中盘的最终搜索深度仍然为8，但是残局搜索深度最高可以达到14了，大大增加了AI下残局的实力。

零窗搜索优化

为了表达简洁易懂，这里采用Negamax的表示方式。对于Negamax，我们会发现每个节点记录的部分搜索最大值在搜索过程中是递增的。这个记录的部分搜索最大值通常情况下初始化为一个很小很小的值。如果我们在初始化的时候给他一个值$\hat{v}$会发生什么事情呢?

假设在完全搜索下的最终结果为v，如果$v \geq \hat{v}$，在搜索过程中我们初始化的值最终一定会被更新为v，不会影响我们最终搜索的正确性。如果$v < \hat{v}$，这样初始化的值不会被搜索结果更新覆盖，从而导致结果不正确。

因此，最好的情况是$v = \hat{v}$，这时候除了最佳决策那个分支之外，其他的分支都会被Alpha-Beta快速剪枝掉，大大加速整个搜索过程。最坏的情况是$v < \hat{v}$，这样就不得不将初始化的值$\hat{v}$变小重新进行搜索。换言之，只要能够合理的设置初始值，就可以对整体搜索进行加速。

如何设置初始值呢？一个可行的想法是我们贪心的按照当前的状态估值向后模拟直到最大搜索深度，以那时候的局面状态估值作为初始值，可以保证$v \geq \hat{v}$，不会出现重新搜索的情况，这个想法实际上就是我们之前的**决策排序优化**。

第二种想法是通过二分或者迭代方法试探性的选择初始值$\hat{v}$，通过散列表来加速重复搜索过程，这种方法被称为**MDT-f**算法。

第三种想法是利用当前局面状态估值进行初始化，试探性的进行搜索，如果初始值没有被更新，就将初始值设定为很小很小的值，然后再进行一次搜索，这种方法被称为**PVS**算法。

因为**PVS**和**MDT-f**算法都会出现再次搜索的情况，因此可以使用散列表减少重复状态搜索次数。

对于黑白棋这个游戏而言，简单的**决策排序优化**已经可以很大程度上减少搜索节点数了，尽管**PVS**和**MDT-f**算法确实可以使得搜索节点数进一步缩小，但是搜索结果更新变得更复杂了，同时散列表插入和查询也需要额外的时间开销。实际测试中整体性能并没有特别明显的提升。

散列表优化

散列表也叫哈希表，具体数据结构算法在这里不再赘述，可以查阅相关资料进行学习。散列表在这里主要是优化对于同一状态的重复搜索，这种情况多出现在迭代加深搜索、PVS搜索和MDT-f算法中。实际实现中需要谨慎的设计散列函数，如果冲突过多会极大的影响性能。经尝试，加入散列表优化确实可以减少搜索节点数，但是因为散列表本身也有时间开销，因此性能也没有非常大的提升。

多重概率剪枝

以上的剪枝方法都是无损剪枝，即剪枝并不会影响最终结果的正确性。而多重概率剪枝是有损的剪枝，也就是说它可能会丢失最优解，但是可以大大加深搜索深度和速度。Logistello使用的也正是多重概率剪枝。

多重概率剪枝的主要思想是如果我需要搜索一个较大的比如12层的深度，我可以先搜索一个较小的比如4层的深度得到一个结果，通过设定阈值过滤掉那些在4层搜索明显不优的节点，对剩下的节点进行较大深度的搜索。我们可以对搜索过程中的每一个浅层节点应用这个算法，可以大大减少搜索分支数。

这个算法之所以有损是因为浅层搜索中那些被过滤掉的节点是有可能在深层次的搜索中成为最优决策的，直接丢弃这些节点会导致最优解丢失。

显然，如果估值函数满足偏差收敛性，且足够准确，通过合理的设置过滤阈值，丢掉最优解的概率是很小的。又因为从迭代搜索那一节那里可以知道浅层搜索相比深层搜索的时间开销几乎可以忽略不计，多重剪枝算法在没有引入太大时间开销的情况下实现了相对较为准确的剪枝，实际实现中通常可以使得搜索深度增加2-4。

实际实现细节

在最终实现中，使用了Negamax+Alpha-Beta剪枝+决策排序优化+迭代加深搜索，多重概率剪枝尽管可以增加搜索深度，但是实际测试中发现过滤阈值不好设定，容易丢失最优解 (也可能因为我写挂了，代码有点复杂)，所以在Saiblo上的版本就没有使用多重概率剪枝。

黑白棋游戏中，并不是完全先后手交替的，存在某一方没有办法行动被跳过的情况，需要在这里进行特殊处理（这里并不满足Alpha-Beta剪枝的条件，按照极大极小搜索理论更新结果的时候也需要注意是极大还是极小，Negamax需要注意是否取负）。

搜索通常用函数递归形式实现，但是因为递归深度较小，每一层递归记录信息较多，为了进一步加速搜索过程，减少反复的内存创建和销毁开销，采用了用栈结构模拟递归的实现方式，并且对分支跳转顺序进行了一些优化。

位运算优化

分析整个算法的时间瓶颈，会发现大部分的时间开销都集中在对于棋盘状态的估值函数和棋盘当前可走位置的查询上。这两个查询是可以通过位运算来进行优化的。

棋盘表示

棋盘上任意一个位置可以用0，1，2进行表示，0表示先手，1表示后手，2表示空。这样一个位置我们可以用2-bit进行表示，8*8的棋盘可以用两个64-bit的无符号长整型来进行表示。

对于棋盘的基本操作，如黑白计数，黑白反转，棋盘转置，棋盘水平翻转，棋盘竖直翻转，棋盘斜对角线提取都可以用位运算来快速完成。

估值函数查询

按照棋盘表示方法，我们可以很快速查询在同一行连续的几个子对应的4进制表示。对棋盘进行转置，我们就可以快速查询在同一列连续的几个子对应的4进制表示。对棋盘进行斜对角线提取，可以快速查询在同一斜线上连续的几个子对应的4进制表示。

这样我们就可以快速查询棋盘状态对于每个模板的值了，这样可以大大加速估值函数查询过程，实际测试中加速比约为5x

可走位置查询

预处理出只有一行8个位置所有可能的状态，以及这些状态对应的可走位置的二进制表示，分别提取每一行，每一列，每一斜线的状态，利用预处理结果可以计算出所有的可走位置，对这些可走位置取并集即可。实际测试中加速比约为10x

后续可能的改进

迭代优化参数

在模板估值方法里，采用每4步训练一组参数权重的方式，这是因为暴力搜索更多层的话时间开销过大。但是在AI实现中，会发现在残局状态下，AI可以达到14-15的搜索深度，如果我们训练的时候采用的不是上一阶段有偏的估计值，而是棋盘终止状态无偏的估计值，这样训练出来的参数权重会更准确。

基于这个想法，我们利用训练好的参数进行极大极小搜索，可以得到更加准确的数据用于训练新的参数权重，而更准确的参数权重又会更有利于极大极小搜索过程中的Alpha-Beta剪枝，加速搜索过程，增加搜索深度。反复迭代该过程直到收敛，这样可以得到更优的参数权重

更复杂的估值模型

模板估值本质上是利用模板提取棋盘特征，然后进行线性回归的过程。这是个相对简单的估值模型，Logistello采用这种模型也是因为90年代的算力限制。我们可以采用更复杂的估值模型，加入CNN，非线性变换等模块提高模型的表征能力，从而更好的对真实结果进行拟合。

但是需要注意的是，估值模型越复杂，计算某个棋盘状态的估值时间开销就越大，外层的搜索深度也会因此受到限制，最后不一定会达到更好的效果。

加入对策略的估值模型

在对局面状态进行估值的同时，我们也可以对策略进行估值，模型输入是当前棋盘状态或者根据当前棋盘状态提取的特征，输出是棋盘上每个位置落子的概率。最简单的节省算力的训练方案是收集棋力较强的黑白棋AI的棋谱，以这些棋谱的棋盘状态和落子选择为数据，训练策略估值模型。

有了策略估值模型后，我们可以尝试性采用一些有损的剪枝方案，比如只搜索策略估值模型给出的概率前五的走法来加速搜索过程。

参考资料

1. Logistello论文https://skatgame.net/mburo/ps/improve.pdf
2. WZebra官网 http://radagast.se/othello/
3. 《黑白棋指南》https://cloud.tsinghua.edu.cn/f/da407a93922f428498d8/