重力四子棋实验报告

2020211543 杨喜凯 经硕 201

目录

一、 算法介绍

重力四子棋是一种完全信息博弈，对应的 AI 算法主要包括 Alpha-beta 剪枝以及蒙特卡洛树搜索 (MCTS)。Alpha-beta 剪枝算法更为简练，但问题在于我们需要寻找一个优良的局面评估函数；而蒙特卡洛树搜索的方式并不需要对重力四子棋有先验知识，其通过每个节点的模拟结果来评估节点的优劣，因此其适应性更强，可以应用于不同的问题中，因此在本项目中我选择了蒙特卡洛树搜索的方式。

1、蒙特卡洛树搜索

蒙特卡洛树搜索的简要步骤如下：

(1)选择：从根节点出发，在搜索树上自上而下迭代式执行一个子节点选择策略，直至找到当前最为紧迫的可扩展节点为止，一个节点是可扩展的，当且仅当其所对应的状态是非停

止状态，且拥有未被访问过的子状态；在此处如果一个节点存在子节点还未作为模拟的起始点，那么该节点便可以作为可扩展节点。

(2)扩展：根据当前可执行的行动，向选定的节点上添加一个子节点以扩展搜索树；在重力四子棋的场景下，扩展便是指我们在当前根节点下生成一个子节点，并将该子节点作为模拟的初始点。

(3)我们对该子节点进行模拟，我们模拟的方式即每一步都是随机落子，直至有一方获胜或者达到平局即结束

(4)我们对此次模拟的结果进行回传，沿着模拟的起点开始回传直至根节点，更新沿途节点的 visit times 以及 score，便于我们后续计算其信息上限值。

以上(1)-(4)循环进行，直至达到计算时间的上限(此处我们选取 time_limit = 2.93)

最终我们从根节点的所有子节点中挑选效果最好的子节点作为我们此次的落子点。

2、UCB 算法

UCB 算法即信心上界算法，是一类解决多臂老虎机问题的算法的总称，UCB 算法的核心在于平衡继续探索以及当前收益二者之间的关系，其具体表达式如下所示：

$$I_j = \bar{X}_j + C * \sqrt{\frac{2\ln(n)}{T_j(n)}}$$

其中，$\bar{X}_j$是手臂j所获得的回报的均值，n是到当前这一时刻为止所访问的总次数，$T_j(n)$是手臂j到目前为止总共所访问的次数。C 是参数，C 越小代表我们越保守。

对于我们的重力四子棋问题而言，$\bar{X}_j$是指节点 j 所获得的回报的均值，该回报通常与该节点关联的模拟结果有关；n 是到当前这一时刻所访问该节点的父节点的总次数，$T_j(n)$是指当当前这一时刻访问该节点的总次数。

3、UCT 算法

UCT 算法是将 UCB1 算法思想用于蒙特卡罗树搜索的特定算法，其与只利用蒙特卡罗方法构建搜索树的方法的主要区别如下：

可落子点的选择不是随机的，而是根据 UCB1 的信心上限值来进行选择，如果可落子没有被访问，则其信心上限值为正无穷大。

当模拟结束后确定最终选择结果时，我们不再仅仅根据胜率做出判断，而是兼顾信心上限

所给出的估计值。除此之外，由于UCT算法兼顾了极大极小的思想，我们也可以选择每个节点的visit times来作为评判的标准。

二、具体实现

在本项目中我主要实现了两个类Node和Board，其中Node代表蒙特卡洛树搜索过程中的节点，其记录父节点/子节点/信息上限值/访问次数/棋盘位置等信息；

Board类则负责记录棋盘信息以及实现相应的UCT算法，先将其简要介绍如下：

Init：初始化棋盘信息，相当于起到构造函数的作用；

treePolicy：相当于蒙特卡洛树搜索中的(1)-(2)步骤，返回一个可以作为此次模拟起点的节点；

defaultPolicy：进行一次模拟并返回此次模拟的reward；

backward：从模拟起点开始进行回溯此次模拟的结果，注意此处存在一个易错点便是在回溯的过程中，父节点和子节点相当于对手方的关系，因此其reward应该取相反值。

search：将上述函数封装，实现信息上限树算法，经过约2.9s的仿真模拟后，返回一个落子点

三、若干尝试与优化

在写代码的过程中，除去最为基本的UCT算法以外，我还尝试了若干改进，其中以失败或者无效的尝试居多，但也不乏一些改进能够提高模型的效果，现将其陈述如下：

首先，我修改了defaultPolicy函数(也就是模拟下棋过程)，在原先双方随机下棋的基础上做了细小的改进，具体来说，在模拟过程中如果当前的player发现了自己的必胜点(该位置可使当前player直接获胜)，那么便不再随机选取落子点，而是直接下在必胜点上；我在代码中以注释的形式保留了这部分逻辑的处理；不过遗憾的是，我粗略的观察了一下结果发现这样的逻辑并没有改善我最终的结果，因此最终仍然是采取随机下棋的方式

除此之外，另外一个尝试便是在search函数之前加一个简单的判断，如果当前的可落子点中存在必胜点或者对方的必胜点，那么从逻辑上来说我们必定要下在这一点上(不然的话如果对手是理智的那么我们必败)，因此此时我们便不再进行 search，而是直接返回该点；

从逻辑上来说这一个判断似乎可以有效改善我们的效果，但实际上似乎并没有什么改善，我们在和一些比较高级的AI样例下棋的过程中会发现，往往在最后的棋盘中，即使我们“堵死”了对手的一个必胜点，对手往往存在两个必胜点，也就是说，这样的局面我们是”无力回天”的；而对于相对简单的AI样例而言，实际上我们并不太需要这种逻辑，因为我们的模拟过程便足以给我们较好的表现。于是最终我们也没有采取这样的先决判断条件。

在 treePolicy(寻找扩展节点)的函数中，我们在原逻辑的基础上做了细微的修改，我们在遍历可落子点的过程中如果我们发现了必胜点(落于该点可直接获胜),那么如果这个位置对应的节点还未被拓展过，那么我们则在该位置上生成节点(而不是像之前一样随机选取位置生成节点)；如果该位置的节点已经被拓展过了，那么我们便不再拓展新的节点，而是直接对该必胜点做模拟以及回溯，在实际代码的逻辑中其实对该节点我们并未做模拟，而是直接回溯了一个reward，因为我们已知其必胜。

除此之外，我们发现在进行扩展的过程中每次生成一个新节点都需要 new 操作开辟一片新的内存，但new操作本身较为耗时，因此我们在Board类中维护了一个全局变量tree，相当于提前开辟了足够大的空间存放Node类型的对象，每次需要使用时便可以从中提取一个Node对象即可。

最后我们的改进是对于回溯过程中reward的改进,起初我们对于胜利和失败的reward分别是1和-1，并且在回溯过程中所有的节点相应的更新值都是1和-1；在代码中我们的处理逻辑是，对于获胜节点，我们给予的reward是5，对于获胜节点的父节点，给予的reward是-4，接下来回溯过程中的reward序列为3,-2,1,-1,1,-1.....依此类推。这样处理是一种很朴素的想法，对于一个离必胜点很远的节点，如果我们给它的 reward 和给必胜点的reward 是相同的值，那么实际上是不合理的，因为我们如果选取了这个”很远”的节点，我们还是有一定概率输掉比赛，因为我们无需给它一个很大的reward来保证我们一定选这个节点，但如果我们下一步要走的棋子中存在必胜点，那么我们理应给必胜点一个很大的reward来保证我们一定能选到这一点。因此距离必胜点越远，对应的 reward 绝对值越小，而当绝对值等于1时便不再递减；如果将回溯的过程看成是一种时间序列数据的话，我们可以认为其自相关性最多只能延展5步棋。

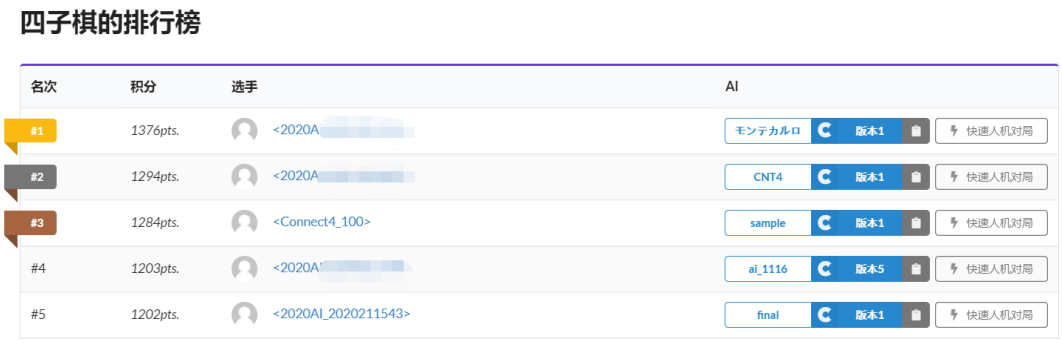
四、模型表现

将模型置于测试网站上观察其对战结果，我们首先将其与82-90共五个AI测试样例进

行对战，结果如下所示，共十战九胜，可以看到在于 Connect4_86 对战的过程中输掉了一局。

接下来我们将其与 92-100 共五个测试样例对战，结果如下所示，共十战六胜，可以看到与 94 号测试样例对决时连败两局，但与 100 号样例对决时却一胜一负；并且在与一些序号较小的样例对决时也会出现失败的情况，可以从此看出我们的模型还是存在一定方差的。

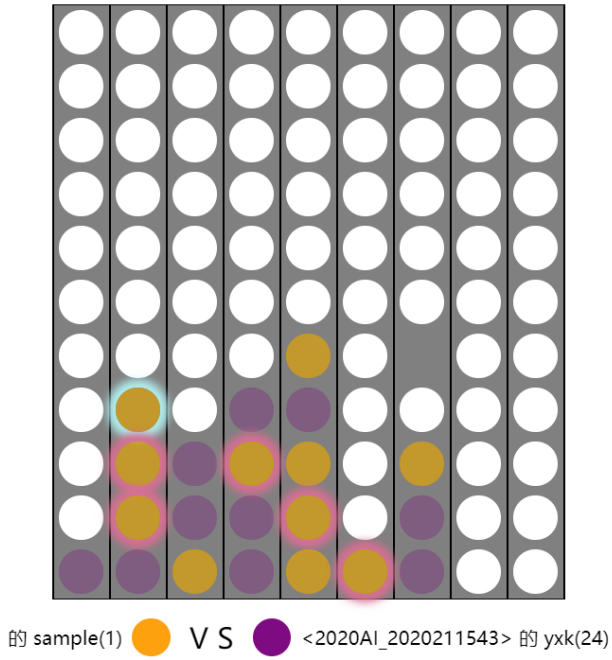
截止到 11 月 20 日 19 时，也就是提交作业的当晚，我在天梯的排名位于第五名的位置。当然随着后续有新的同学加入，可能排名会有波动。



五、不足与总结

我们在 treePolicy 的实现过程中提及到了必胜点处的剪枝;实际上从人类的角度来说，“必落子点”的范围要远远大于“必胜点”；很多情况下对方只连起来两步棋，如果这时候下一步两边都没有我们的棋去“堵住”的话，那么对方必胜；

我们假设上图是一个棋盘最后一行的情况，假设此时双方只下了三个棋，下一个棋该由 1 来走，如果 1 不堵住 2 的两侧的话，接下来无论 1 再怎么走，2 都可以必胜。但这种局面实际上比较难以评估,或者说我没有想出比较好的方法来提前“预判”必胜或者必败的场景;在我比较早期的程序版本中，就经常会出现在这种场景下下不对棋的情况。



在我初始阶段的程序中,还经常出现如上图所示的情形,可以看到接下来是紫色方下棋,但是不论我们下在第二列还是第三列，对于黄色方而言都是必胜的局面，但这种情形的预判显然要比上一种情形复杂得多。而且这种棋局的出现往往是比较复杂的多因素共同作用的结果。不过令人欣慰的是，在后面调试程序的过程中，随着代码逻辑的完善，这种棋局出现次数有很大程度的减少。

在测试网站与其他同学“对决”的结果中,如果我们仔细观察一些比自己更强的“棋手”

的表现，会发现他们每步棋的耗时可能并不是一直都处于 2.8 或者 2.9 附近，而是会有一部分棋子的落子时间非常短，而我的每步棋的耗时都是稳定在 2.9s 附近；我猜测更优秀的“棋手”可能基准模型是蒙特卡洛树搜索，但可能在优化过程中使用了一些例如估值函数的方法来帮助他们进行剪枝；因此如果我们今后能够在蒙特卡洛树搜索这种“暴力”方法上加入合适的 domain knowledge，例如上面我们提到的“预判”棋局，相信能够显著提升我们的效果。

对于运行效率的优化，在本次项目中落子是有时间限制的，因此运行效率显得格外重要，如果运行效率高，模拟次数增多，那么理应会获得更优良的效果；在本次大作业中我们用全局变量的方式来减少了频繁 new 操作带来的开销；我在之前浏览资料时发现可以用二进制的形式来表示棋盘中的一行或者一列，从而可以通过位运算的方法来实现时间复杂度的优化。但限于时间有限我并没有实现这一功能，但相信如果用位运算的形式来处理棋盘(比如判断是否获胜)，那么算法的整体效率应该会有显著提升。

除此之外，还有一些其他可以改进的地方，例如我们程序的一些超参数，比如 UCB 算法中的 C 如何确定，search 过程中 computation budget 应该取多少秒既可以保证效果又不至于超时，调整这部分超参数相信也会给我们模型带来提升。

这次大作业耗费了我比较多的时间，开发的时间大约花费有一整天，但最初下棋的表现并不尽人意，而且因为测试网站并不会报告代码中的错误，因此我花费了若干个晚上来调试代码。其中也是走过了很多的“坑”，记忆最为深刻的经历便是从测试网站上把棋盘手动输进代码里，然后在 debug 模式下一行一行的运行，看 treePolicy 是否选对了节点，看 defaultPolicy 模拟的过程中有没有漏下什么步骤，然后自己去思考为什么某一步棋没有下对以及有什么改进的方法。

实际上感觉本次项目 debug 难度还是比上一次的拼音输入法要难很多，我们需要考虑很多不同的情况，因为每一步的落子，实际上都是经过了十几万次模拟后选出的最优子节点；之所以落子点不合适，可能是因为我们代码运行效率低导致模拟次数不够，从而没选出合适的节点，这时我们便应该去思考如何改进效率而不是去研究代码逻辑；也有可能是因为我们代码内部逻辑有些细微的错误，但这种错误实际上很难去发现，我有一个印象比较深刻的错误便是在 treePolicy 的函数中，如果当前节点的子节点已经全部被拓展过一次并且需要选择最优子节点，那么在选完最优子节点之后我们需要更新我们的棋盘以及相应信息(相当于在这一点下了棋)，但最初我在 debug 的时候并没有在这一点附近仔细观察棋盘的变化，也没有意识到这个问题，从而导致花了很久才发现这个问题。

当然，这次大作业我也收获了很多；我本科并非计算机相关专业，工程能力较薄弱，这次项目很大程度上提升了我的工程能力，也锻炼了我 debug 复杂逻辑的能力。当看到自己写的下棋程序能够和 AI 测试样例有“一战之力”的时候，内心还是挺高兴的，觉得很值得。感谢马老师和助教学长学姐这次的大作业机会，感谢悉心教导！感谢 Saiblo 网站的负责同学！