**黑白棋实现（Mini AlphaGo）**

**实验背景：**

黑白棋 (Reversi)，也叫翻转棋，是一个经典的策略性游戏。

一般棋子双面为黑白两色，故称“黑白棋”。因为行棋之时目标为将对方棋子翻转，变为己方棋子，故又称“翻转棋” (Reversi) 。黑白棋多使用 8x8 的棋盘，由两人执黑子和白子轮流下棋，终局子多方为胜方。

因为黑白棋的进行过程中，仅有棋子的翻转与交替落子行为，而不存在类似于围棋的吃子行为，因而对于 8\*8 64个落子点而言，黑白棋总能在64步内得出结果。在本实验中，我们使用‘X’来表示黑棋的落子，‘O’表示白棋的落子，并默认黑棋先手，且黑白双方下棋者可以在棋局结束后任意互换，从而进行游戏。

**详细规则：**  
 1.黑方先行，在有合理落子的前提下双方交替下棋。

2.其中一步合理的棋步包括：

2.1 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；

2.2新落下的棋子必须落在可夹住对方棋子的位置上，对方被夹住的所有棋子都要翻转过来

2.3夹住的位置上必须全部是对手的棋子，不能有空格；

2.4一步棋可以在数个（横向，纵向，对角线）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转某个棋子。

3.如果一方没有合法棋步，即没有方法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。

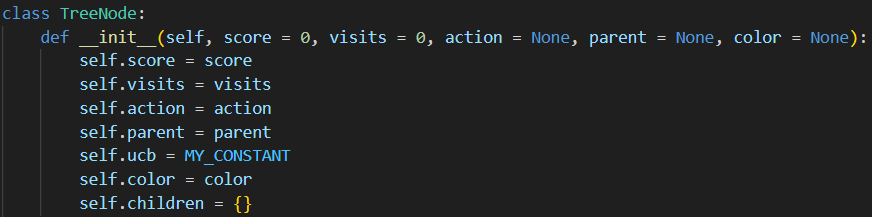
4.如果一方至少有一步合法棋步可下，他就必须落子，不得弃权。

5.棋局持续下去，直到棋盘填满或者双方都无合法棋步可下。

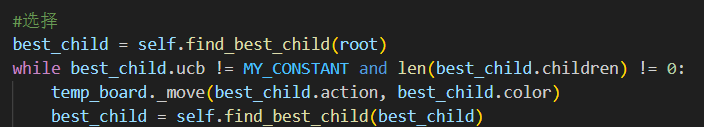
6.如果某一方落子时间超过 1 分钟 或者 连续落子 3 次不合法，则判该方失败。

**具体实现：**

本实验要求使用“蒙特卡洛树搜索算法”实现 miniAlphaGo for Reversi。在实现蒙特卡洛树搜索算法前，我先声明了蒙特卡洛树可能需要使用的数据结构：

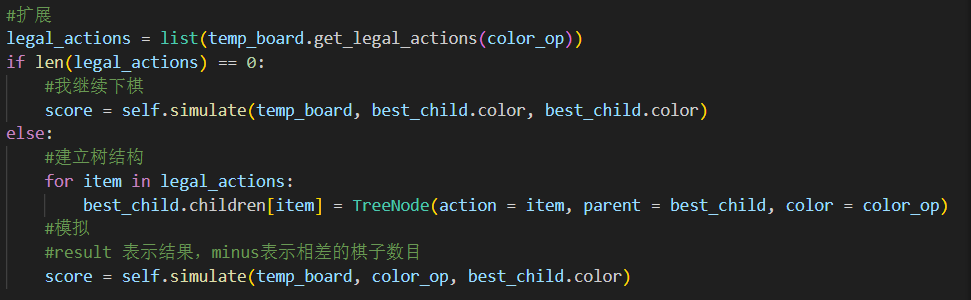
 其中 score属性用于记录当前节点的累计得分；visits用于记录当前节点相应的访问次数；action表示节点对映的具体行动；parent则记录节点的父亲节点，便于后续更新过程的进行；color记录了当前节点对映的棋子类型（‘X’或者‘O’）；children则记录了其对应的子节点。

为了便于debug的过程，我也在该class下实现了print\_tree函数，可选的输出第一层children（用于判断ucb计算情况与首轮选择）或整棵树的状态。

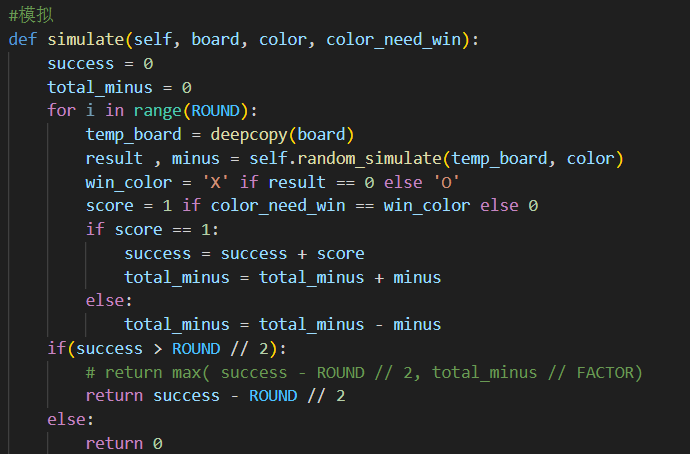
在class AIPlayer的实现中，我则仿照蒙特卡洛算法，先对当前已知的最优策略进行选择：

find\_best\_child函数即通过比较当前节点的所有子节点的ucb值，在其中选择最大的ucb子节点最为最佳选择对象。需要特别注意的是，在每次选择完最佳子节点后，我们需要再棋盘上相应的进行落子操作，仿造棋盘的进行。因为在仿造棋盘进行的过程中，棋盘本身会被修改，从而我们需要对棋盘进行深拷贝后再加利用。为了便捷后续操作，我也根据self.color得出了对手当前持有的棋子颜色。

接着是扩展阶段，在这一阶段，我们需要分两种情况考虑。一方面是扩展的点已经是叶子节点，即另一方在当前节点没有后续落子空间，这对应了游戏结束和我方继续落子两种局面，均可以使用simulate过程来进行模拟。另一方面便是需要进行新节点的拓展：

首先需要遍历所有可能的落子点，添加至当前节点的children列表中，对蒙特卡洛树进行进一步的建立；接着通过simulate函数模拟棋局直至终局状态，从而获得相应的分数。

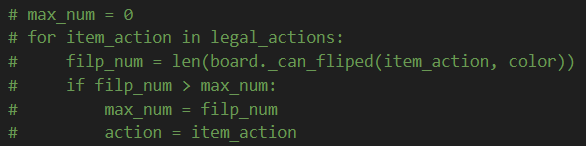
在simulate函数中，我支持旋转进行一定的轮次，来模拟每一步的终局状态。一方面是考虑如果直接使用随机或其他方法模拟至游戏结束，可能会有较大的不确定性，不能很好的反映对映扩展点的真实情况；另一方面，进行多轮的尝试，可能可以使用给多的计分方式，而不只是通过输赢和判断得分与否，或许可以引入胜子数量来更新这一得分情况。

为了可以引入更多的simulate策略，我仅把simulate函数作为一个进入模拟进程的借口：

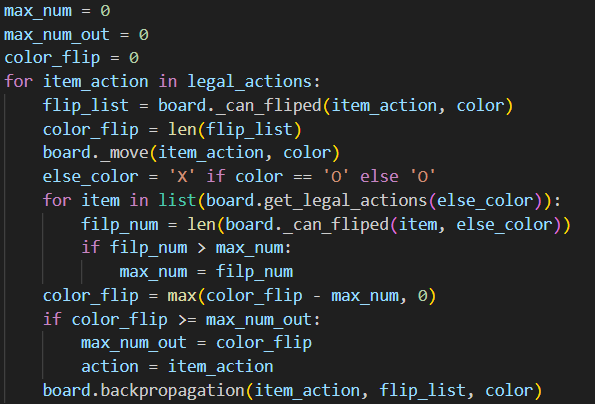
同时，我维护了success与total\_minus两个变量来记录总循环次数中我在模拟中获胜的次数以及赢得棋子的净值。在simulate函数中，我也考虑了多个模拟策略。

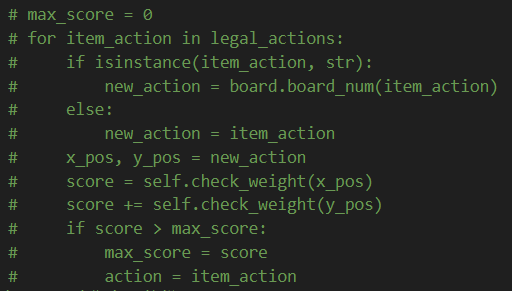
最简单的，即随机落子策略，使用随机模拟得到终局结果。

第二种，翻转对方棋子最多策略：

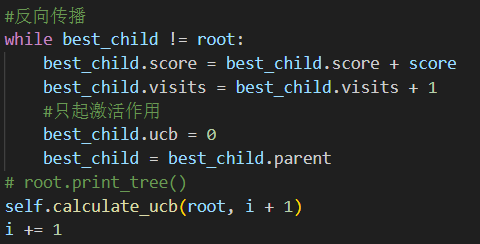
每次选择可以翻转对方最多棋子的策略，在每一步贪心获得更好的局面。如果配合以胜子作为得分的计算策略，可以促使蒙特卡洛树更加倾向于选择可以得到大胜结果的落子方向。

第二种的变种，当前步翻转净棋子最多，这个策略不再仅仅考虑当前翻转棋子数达到最大，其更向后看一步，即落子之后，考虑对方可以将局面翻转的能力大小，通过当前步可以收获的最大净棋子数，来选择合适的地方落子。需要特别注意的是，推测对手的落子需要改变棋局的情况，因而计算结束后需要使用backpropagation() 函数来退回当前的模拟情况。

第三种，边缘偏好性：

通过观察终局结果可以看到，获胜或获大胜方往往占据了更多的外围空间，特别是边缘空间，因而我对落子的可行空间进行人为的打分，并使用check\_weight 这一打分函数对当前的落子点进行评价，最倾向于边缘坐标，更倾向于1/3/6/8行与1/3/6/8列，来为终局附近的翻转局面提供更多的计算便利性。

在进行完simulate的模拟之后，便是结果回传与ucb数值更新阶段：

在此，我先更新当前路径上节点的score与visits属性，并将ucb属性激活（本来为MY\_CONSTANT常量，定义为无穷大）。接着，我使用calculate\_ucb函数更新root节点子节点的ucb数值，再进行后续循环，再后续每次find\_best\_child前，我们都要使用calculate\_ucb函数来更新当前层的children的ucb属性，这样单层的更新可以减少不必要的时间消耗。

当计时终止后，我们在此对root节点进行find\_best\_child函数，挑选最佳的落子点位即可得到对应的action。