黑白棋 (Mini AlphaGo)

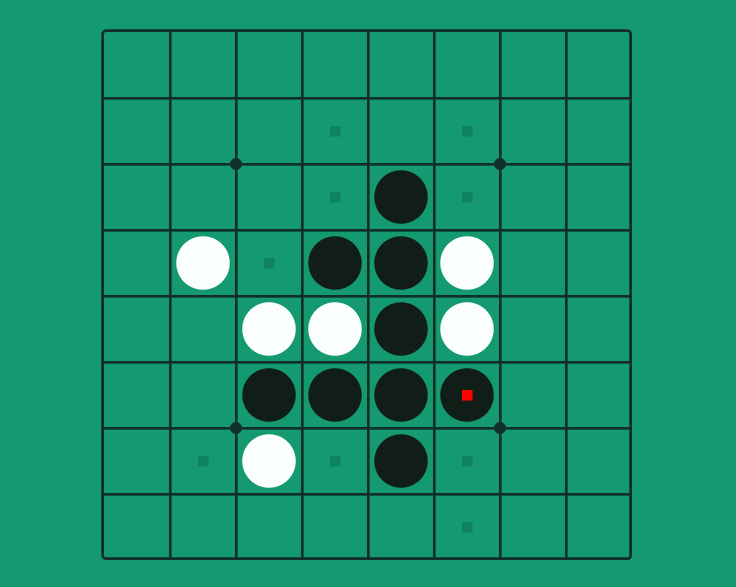
3220104382 吴俊亨、3220102094 王子扬、3220101111洪晨辉

**1 内容概述**

黑白棋 (Reversi)，也叫苹果棋，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”。因为行棋之时将对方棋子翻转，则变为己方棋子，故又称“翻转棋” (Reversi) 。  
 棋子双面为红、绿色的称为“苹果棋”。它使用 8x8 的棋盘，由两人执黑子和白子轮流下棋，最后子多方为胜方。

棋盘如图所示。



下棋规则：

* **黑方先行，双方交替下棋。**

一步合法的棋步包括：

* + 在一个空格处落下一个棋子，并且翻转对手一个或多个棋子；
  + 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，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；
  + 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某个棋子。

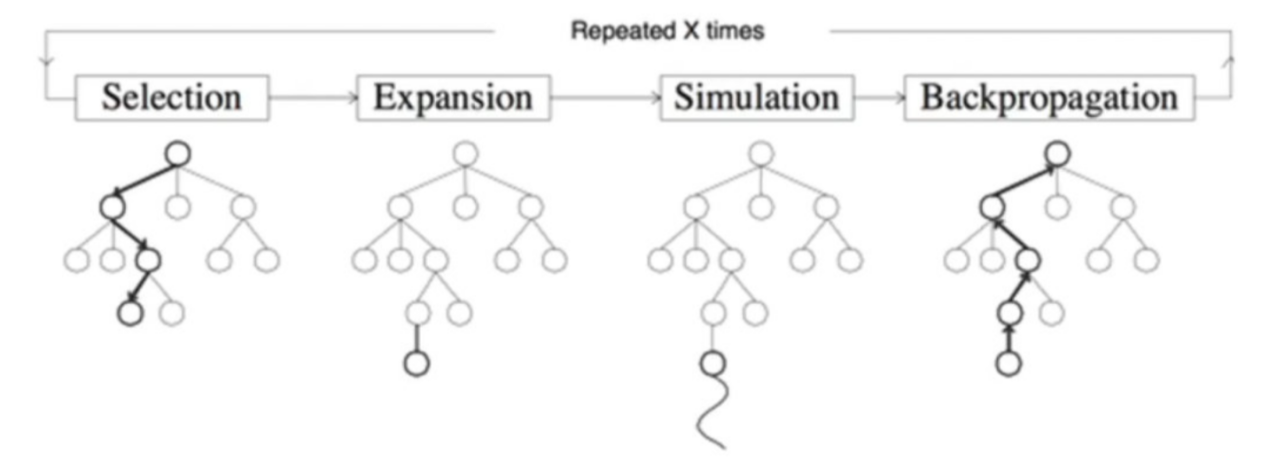
注意事项：

* 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。
* 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。
* 棋局持续下去，直到棋盘填满或者双方都无合法棋步可下。
* 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。

**2 蒙特卡洛树**

蒙特卡洛树搜索（MCTS）是一种经典的树[搜索算法](https://so.csdn.net/so/search?q=%E6%90%9C%E7%B4%A2%E7%AE%97%E6%B3%95&spm=1001.2101.3001.7020" \t "https://songjian.blog.csdn.net/article/details/_blank)，名镇一时的 AlphaGo 的技术背景就是结合蒙特卡洛树搜索和深度策略价值网络，因此击败了当时的围棋世界冠军。它对于求解这种大规模搜索空间的博弈问题极其有效，因为它的核心思想是把资源放在更值得搜索的分枝上，即算力集中在更有价值的地方。

MCTS算法主要有四个步骤，分别是选择、扩展、模拟和回溯。



**2.1 选择（Selection）**

未访问：还没有评估过当前局面；

未完全展开：被评估过至少一次，但是子节点（下一步的局面）没有被全部访问过，可以进一步扩展；  
 完全展开：子节点被全部访问过。

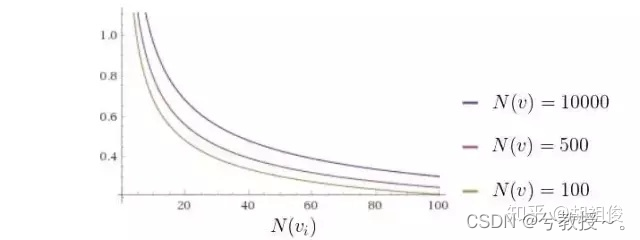
从根节点开始，递归选择最优的子节点，最终到达一个叶子结点。

节点的优劣由UCB公式（Upper Confidence Bounds）决定。

其中，是该节点赢的次数，是该节点模拟过的次数，c是一个常数，通常取2。我们通过改变c的值改变算法效率。

指的是节点的胜率，但是胜率大并不能说明这个节点就更优。

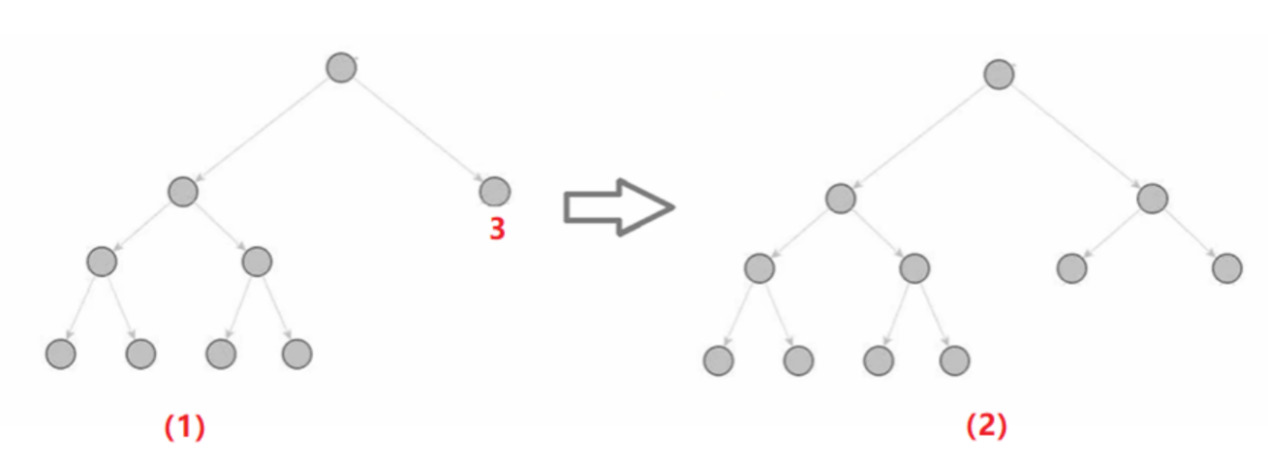
函数的图像如下。



我们可以看出，随着访问次数N(vi)的增加，加号后面的值越来越小，因此我们的选择会更加倾向于选择那些还没怎么被统计过的节点，这样一来可以防止尽管这个节点不怎么好，但是一开始随机走子的时候赢了一盘，就会一直走这个节点的情况出现。

因此我们每次选择的过程如下——从根节点出发，遵循minmax原则，每次选择己方 UCT 值最优的一个节点，向下搜索，直到找到一个「未完全展开的节点」，未完全展开的节点一定有未访问的子节点，随便选一个进行扩展。

**2.2 扩展（Expansion）**

如果当前叶子结点不是终止节点，那么就创建一个或者更多的子节点，选择其中一个进行扩展，统计信息为「0/0」，然后进入下一步模拟。   


**2.3 模拟（Simluation）**

在某一结点用随机策略进行游戏，又称 playout 或 rollout。从扩展节点开始，运行一个模拟的输出，直到博弈游戏结束。比如，从该扩展节点出发，模拟了十次，最终胜利九次，那么该扩展节点的得分就会比较高。

**2.4 回溯（Backpropagation）**

又称反向传播，就是从子节点开始，沿着刚刚向下的路径往回走，沿途更新各个父节点的统计信息。在Rollout中我们计算出了value之后，我们需要返回这个value，那么对于它所有的父节点，它们的探索次数全部+1，它们的value也会进行一个累加，整个算法会repeate很多次，直到蒙特卡洛树能够给出当前状态下最好的一个解答。

**2.5 算法终结**

* 游戏内棋手给定限制时间
* 人工给定固定的迭代次数

**3 研究意义和展望**

**3.1 研究意义**

基于蒙特卡洛树的黑白棋算法是一种新型的人工智能算法，它可以用来解决黑白棋这种复杂的博弈问题。相比传统的博弈树搜索算法，基于蒙特卡洛树的算法更加高效，能够在更短的时间内找到最优解，同时还具有一定的随机性，能够避免陷入局部最优解。

基于蒙特卡洛树的黑白棋算法的研究意义主要体现在以下几个方面：

* 提高黑白棋博弈的智能化水平。基于蒙特卡洛树的算法能够自动学习和优化策略，可以不断提高棋手的水平，从而使黑白棋博弈更加智能化。
* 推动人工智能的发展。基于蒙特卡洛树的算法是一种新型的人工智能算法，其研究和应用可以推动人工智能的发展，同时也可以为其他领域的研究提供借鉴和参考。
* 拓展蒙特卡洛树算法的应用范围。基于蒙特卡洛树的算法不仅可以应用于黑白棋博弈，还可以应用于其他博弈问题、路径规划问题等领域，其应用范围非常广泛。
* 总之，基于蒙特卡洛树的黑白棋算法是一种非常有前途的人工智能算法，其研究和应用具有重要的意义和价值。

**3.2 算法展望**

在文献*Monte Carlo Tree Search in Hex*中，作者提到了一些基于蒙特卡洛树的黑白棋算法的优化方法。

* 启发式策略：在树的扩展过程中，使用启发式策略来选择最有可能导致胜利的子节点进行扩展，从而减少搜索空间。
* 快速模拟：在随机模拟过程中，使用一些快速的模拟方法来加速搜索过程，如快速移位、快速判断胜负等。
* 并行计算：使用多线程或分布式计算等方法来加速搜索过程，提高算法效率。
* 此外，还有其他一些算法优化方法，如使用更好的状态表示方法、改进评估函数等。这些优化方法可以提高基于蒙特卡洛树的黑白棋算法的表现，使其具有更好的效果和可扩展性。

**4 实验目的**

1. 根据上述游戏规则编写python代码在计算机上实现黑白棋游戏
2. 不依赖现成的包、工具或接口使用【蒙特卡洛树搜索算法】实现 miniAlphaGo for Reversi。

**5 实验内容（代码后附）**

****一、初始化与首次扩展****

1. ****开始****：启动算法流程。
2. ****棋盘参数设置****：设定BOARD\_SIZE（棋盘大小为 8）、PLAYER\_NUM（根据玩家执子情况确定，如 -1 或 1）、COMPUTER\_NUM（与PLAYER\_NUM 对应，如 1 或 -1）、MAX\_THINK\_TIME（电脑最大思考时间，如 5）等参数。
3. ****节点树初始化****：
   1. 检查 PATHROOT是否为空。
   2. 若为空，调用expand函数：
      1. 针对当前棋盘状态（初始棋盘），通过 possible\_positions 函数获取计算机执子时的可落子位置列表。
      2. 对于每个可落子位置，创建一个节点元组 (temp, 0, 0, [])（temp为位置坐标，初始模拟次数为 0，奖励为 0，子节点列表为空），并将这些元组组成的列表作为 PATHROOT 的初始值。
   3. 对 PATHROOT 中的每个初始节点：
      1. 复制当前棋盘状态得到current\_board。
      2. 调用updateBoard函数在current\_board上执行该节点的落子操作（将计算机棋子放置在相应位置）。
      3. 进行 20 次循环（for i in range(1, 21)）：
         1. 复制current\_board得到current\_board2。
         2. 调用find\_playout函数，传入current\_board2和与当前节点执子情况相反的棋子（-playerNum），进行模拟对局，并根据结果更新该节点的奖励值（若模拟对局获胜，reward += 1）。
      4. 更新 PATHROOT 中该节点的信息，包括模拟次数（t\_playout = 10）和更新后的奖励值。
4. ****进入循环搜索阶段****

****二、循环搜索（受最大遍历次数等条件限制）****

****选择（Selection）步骤****

* 1. 从根节点（PATHROOT）开始。
  2. 对于 PATHROOT 中的每个子节点：
     + 获取节点元组(name, nplayout, reward, childrens)。
     + 检查nplayout是否为 0，若为 0 则设置为 1；同时初始化循环次数t（若为 0 则设置为 1）。
     + 调用ucb函数计算该节点的 UCB 值：(reward / nplayout) + ucb\_c \* math.sqrt(math.log(t) / nplayout)，其中ucb\_c为预设的常数（如 0.7）。
  3. 根据最大最小搜索策略（在find\_path函数中实现）：
     + 初始化current\_path为空列表child为PATHROOT，parent\_playout为 0，并标记isMCTSTurn为 True（表示电脑方回合）。
     + 进入循环：
       - 检查child是否为空，若为空则跳出循环。
       - 初始化maxidxlist为[0],maxval根据isMCTSTurn确定（电脑方为 -1，玩家方为 2），cidx为 0。
       - 对于child中的每个节点元组：
         * 计算该节点的 UCB 值（同上述计算方式）。
         * 根据isMCTSTurn判断：

若为电脑方回合：

若当前节点 UCB 值大于等于maxval：

若等于maxval，将当前节点索引cidx添加到maxidxlist。

若大于maxval，更新maxidxlist为[cidx]，并更新maxval为当前节点 UCB 值。

若为玩家方回合：

若当前节点 UCB 值小于等于maxval：

若等于maxval，将当前节点索引cidx添加到maxidxlist。

若小于maxval，更新maxidxlist为[cidx]，并更新maxval为当前节点 UCB 值。

* + - * + cidx自增 1。
      * 从maxidxlist中随机选择一个索引maxidx。
      * 获取child[maxidx]的节点元组信息，将其parent添加到current\_path，更新parent\_playout为该节点的t\_playout，并将child更新为该节点的t\_childrens，同时切换isMCTSTurn的状态（not (isMCTSTurn)）。
  1. 确定一条从根节点到叶节点的路径current\_path，进入扩展步骤。

****扩展（Expansion）步骤****

* 1. 取current\_path中的最后一个节点（即选择的叶节点）。
  2. 调用possible\_positions函数，判断该叶节点是否还有可扩展的子节点：
     + 若有可扩展子节点：
       - 调用expand函数：
         * 针对当前棋盘状态（current\_board，通过之前的落子操作得到），获取可落子位置列表。
         * 对于每个可落子位置，创建一个新的节点元组 (temp, 0, 0, [])，并将这些元组组成的列表作为新的子节点列表。
       - 记录新扩展的子节点数量randomTime（len(t\_childrens) \* 10），并初始化rewardSum为 0，进入模拟步骤。
     + 若无可扩展子节点，直接进入模拟步骤，使用当前叶节点对应的棋盘状态进行模拟。

****模拟（Simulation）步骤****

* 1. 针对新扩展的节点或选择的叶节点（若无可扩展子节点）：
  2. 开始模拟对局（在find\_playout函数内）：
     + 初始化depth为 0，进入循环（while(depth < 120)）：
       - 调用possible\_positions函数获取当前可落子位置列表turn\_positions。
       - 检查len(turn\_positions)是否为 0：
         * 若为 0，切换轮次（tile = -tile），再次调用possible\_positions函数获取对方可落子位置列表neg\_turn\_positions。

若len(neg\_turn\_positions)也为 0，调用eval\_board函数根据当前棋盘状态判断胜负（比较双方棋子数量），并结束模拟，返回胜负结果。

若len(neg\_turn\_positions) 不为 0，将turn\_positions更新为neg\_turn\_positions。

* + - * 从turn\_positions中随机选择一个位置 temp（temp = turn\_positions[random.randrange(0, len(turn\_positions))]）。
      * 调用updateBoard函数在当前棋盘上执行落子操作（updateBoard(tep\_board, tile, temp[0], temp[1])）。
      * 切换轮次（tile = -tile），depth自增 1。
    - 循环结束后（达到最大模拟深度 120 次递归），调用eval\_board函数根据当前棋盘状态判断胜负，返回胜负结果。
  1. 记录模拟结果（胜负情况）：
     + 若模拟对局获胜，rewardSum += 1。
     + 对新扩展的每个子节点（如果存在）：
       - 循环一定次数（如 10 次）：
         * 复制当前棋盘状态得到child\_board。
         * 在child\_board上执行当前子节点的落子操作（updateBoard(child\_board,tempTile, child\_parent[0], child\_parent[1])，tempTile为当前执子情况）。
         * 调用find\_playout函数，传入child\_board和与当前执子情况相反的棋子（tempNegTile），进行模拟对局，并根据结果更新当前子节点的奖励值（若模拟对局获胜，reward += 1）。
  2. 完成所有模拟后，进入回溯步骤。

****回溯（Backpropagation）步骤****

* 1. 根据模拟结果，沿着之前选择的路径current\_path反向传播：
  2. 从PATHROOT开始：
     + 对于current\_path中的每个节点：
       - 遍历PATHROOT中的每个节点元组，找到与当前路径节点匹配的节点（通过位置坐标判断）。
       - 更新该节点的模拟次数（t\_playout += randomTime，randomTime 为新扩展节点的模拟次数或 0，如果没有新扩展节点）和奖励值（reward += rewardSum，rewardSum 为模拟获胜的总次数）。
  3. 回溯完成后，检查是否达到最大遍历次数（30 次）：
     + 若未达到，返回选择步骤继续下一轮循环搜索。
     + 若达到，进入决策阶段。

****三、决策阶段****

1. 在完成规定次数（30 次）的循环搜索后，遍历PATHROOT中的每个节点元组：
   * 获取节点元组(parent,t\_playout,reward,t\_childrens)。
   * 检查t\_playout是否大于 0 且(reward/t\_playout) 是否大于当前最大平均奖励 max\_avg\_reward：
     + 若满足条件，更新mt\_result为当前节点的parent，并更新max\_avg\_reward为(reward/t\_playout)。
2. 调用coordinate\_to\_label函数将mt\_result（最终选择的最佳落子位置坐标）转换为便于展示的字符串格式（如 "G6"），作为最终决策结果输出。

****四、更新与持续决策（如果游戏未结束）****

1. 根据实际落子情况更新棋盘状态：
   * 调用updateBoard函数在实际棋盘上执行落子操作，将计算机棋子放置在决策阶段确定的最佳落子位置。
   * 调用updateAIBoard函数更新棋盘状态表示（用于算法内部记录）。
2. 更新节点树 PATHROOT：
   * 调用updatePathRoot函数：
     + 比较当前棋盘与上一次保存的棋盘（self.last\_AIboard），找出上一次棋盘上有而当前棋盘上消失的 -1 位置（代表玩家上次落子位置）。
     + 对于这些位置，在self.last\_AIboard上调用updateBoard函数执行玩家的落子操作，并更新 PATHROOT，移除与这些位置相关的节点及其子树，同时更新剩余节点的信息（如模拟次数、奖励值等），以反映最新的棋盘状态。
3. 返回循环搜索阶段，进行下一轮决策，直至游戏结束。

****五、算法优点****

**1.高效模拟资源分配**：

**1.1深度控制**：find\_playout 函数设最大递归深度 120 次，如棋盘局面复杂，某分支无深度限制会持续探索，浪费资源。此限制使算法在有限时间内适度深入多种局面，合理分配资源，提升搜索效率。

**1.2广度拓展**：mctsNextPosition 函数循环遍历限 30 次，开局时众多落子选择，此广度限制让算法在一定范围探索多种情况，把握全局局势，避免局部最优，控制时间。

**2.模拟次数精准调配**：

**2.1初始评估**：初始化时，新扩展节点借 mctsNextPosition 相关逻辑进行特定次数（如 20 次）模拟，如在新节点扩展处，快速获取潜力信息，初步筛选落子选择，节省后续资源与时间。

**2.2动态调整**：循环搜索中，依据节点情况调模拟次数。像 find\_playout 函数多次调用时，前期胜率高或关键节点获更多资源，边缘或低胜率节点则减少次数，使资源集中于关键决策节点，提高模拟效率与决策准确性。

**3.剪枝高效决策**：

****3.1基于 UCB 算法的隐式剪枝****

在你的代码中，mctsNextPosition函数里通过ucb函数计算节点的 UCB 值来选择路径。随着模拟的进行，那些 UCB 值低的节点（比如胜率低或者已经被大量访问但效果不佳的节点）逐渐不被优先选择，从而达到类似剪枝的效果，减少了在这些不太有希望的分支上的探索。

****3.2基于搜索深度的剪枝****

find\_playout函数中的最大递归深度限制（120 次）是一种典型的基于深度的剪枝策略。当模拟对局达到这个深度后，就会停止深入这个分支并返回结果，避免在过深的分支上浪费计算资源，这与代码中的实际应用紧密相关，用于控制模拟的深度，从而有效利用计算资源。

****3.3基于游戏规则和局面的剪枝****

代码中的isok函数用于检测位置是否可以落子，possible\_positions函数用于返回一个颜色棋子可落子位置。通过这些函数，将不符合游戏规则或者没有可落子可能的位置排除在搜索范围之外，这在代码中是直接的剪枝操作，能够快速缩小搜索空间，提高算法效率。

**4.信息驱动精准抉择**：

**4.1初始信息**：算法起始用 countTile 函数统计棋盘棋子数量，获基本信息，为后续搜索决策打基础，早期评估落子方向可行性，节省决策时间。

**4.2动态信息**：updateBoard 函数记录模拟对局落子后棋盘变化，包括棋子翻转与位置更新，实时反馈给搜索决策，使算法依棋局动态调策略，避免无效路径，增强决策精准与时效。

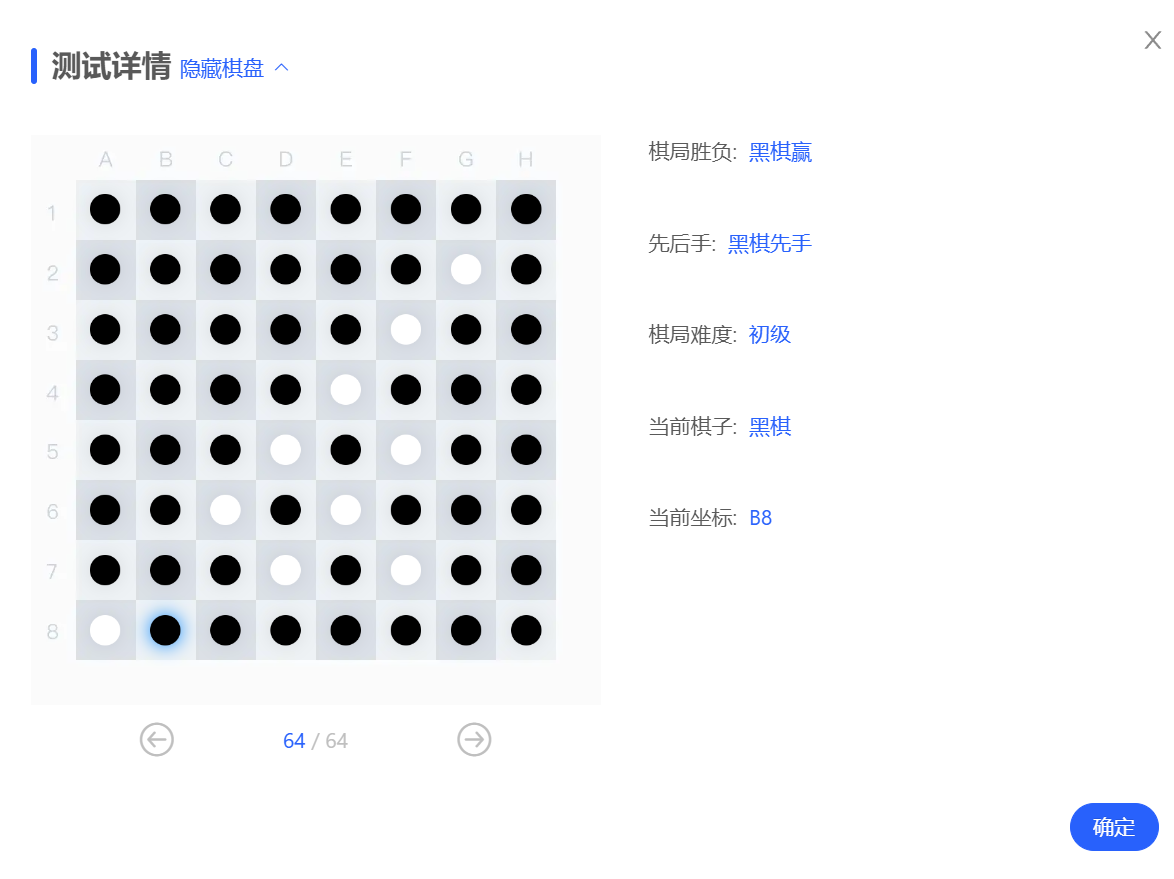
**4.3节点信息**：PATHROOT 节点信息如模拟次数、获胜次数等，是决策关键。mctsNextPosition 函数综合分析这些信息与 UCB 算法，快速筛选最优落子位置，如决策阶段依此确定局面下最佳落子，科学精准决策，高效应对复杂场景，提升获胜概率。

**6 测试结果与对比**

经过测试，我们小组的模型已经可以战胜高级难度的对手：

而在对弈初级、中级难度的对手时，结果如下：

本次战胜初级对手44目：

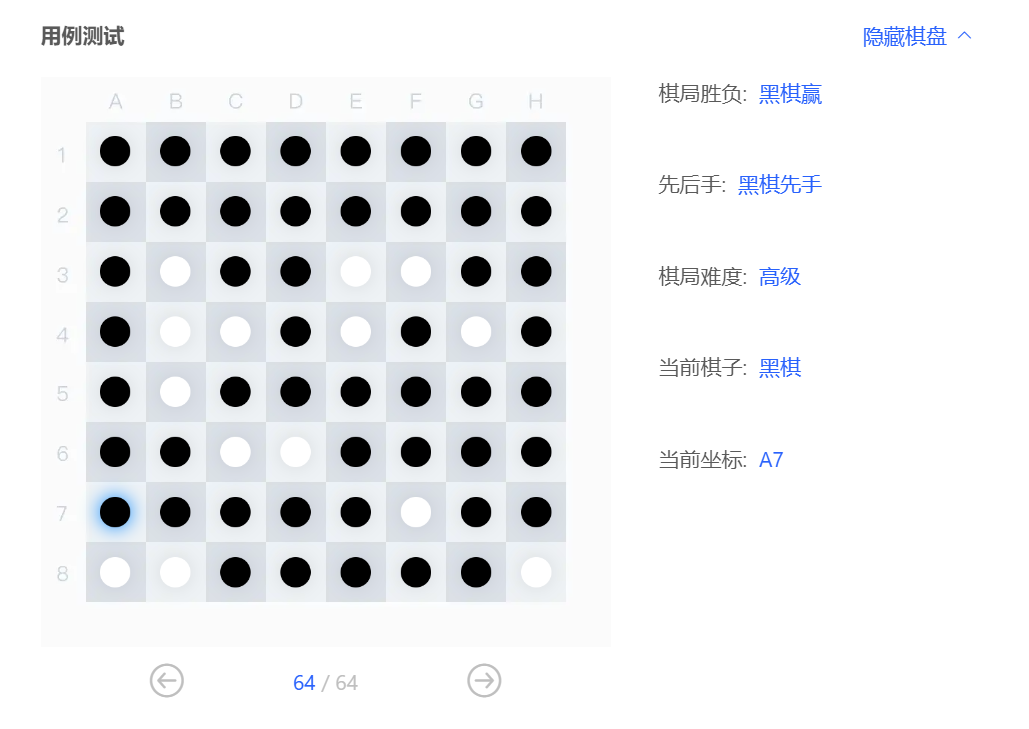


本次战胜中级对手28目：



最终在与高级难度的对手对弈中，赢得了36目：





由此可见，我们的模型显示出了强大的性能，在对战三级难度对手时均有稳定可观的胜率。

**7 实验缺陷与改进**

**7.1 节点树遍历效率较低：**

在处理节点树PATHROOT时，如在updatePathRoot函数中，采用简单的遍历方式查找特定节点，随着节点数量增多，时间复杂度会显著增加。可通过建立索引结构（如以节点坐标建字典索引）、利用哈希表、改造为平衡搜索树或结合分层存储与缓存机制等方式优化节点查找，降低时间复杂度。

**7.2 UCB 算法稳定性局限：**

UCB算法在计算节点价值时，依赖于模拟次数和奖励值。当模拟次数较少时，计算结果可能波动较大，导致节点选择不稳定，进而影响整个算法的决策稳定性。例如在游戏初期，各节点模拟数据较少，UCB值的可信度较低，可能使算法做出不够准确或一致的决策。可进一步研究根据搜索阶段动态调整ucb\_c参数的策略来进行改进。

**8 总结感悟**

吴俊亨：刚开始接触这个项目时，蒙特卡洛树算法对我来说是个不小的挑战。但我没有退缩，通过查阅大量资料和向学长学姐请教，我慢慢摸清了它的工作方式。之后，我将重点放在与算法优势相关的超参数调整上，经过反复的试验和优化，成功构建出高性能的模型。这个过程让我收获颇丰，不仅学会了如何攻克知识难点，还提升了自己解决实际问题的能力。我会将这次的经验运用到今后的学习中，不断提升自己的专业素养。

王子扬：这次实验让我切实掌握了蒙特卡洛树算法在黑白棋游戏中的运用。从最初设计程序时的迷茫，到逐渐理解算法原理并将其应用于优化对战算法，每一步都凝聚着自己的努力。过程中，与电脑内置程序对弈屡屡失败以及遭遇运行时间超限的问题，确实让我感到沮丧，但也激发了我深入探究的决心。通过不断排查问题、尝试不同的方法，我最终克服了困难，这使我对课堂所学的理论知识有了更直观、深入的理解，也让我明白遇到困难不能轻易放弃，坚持下去总会找到解决办法。

洪晨辉：在本次实验里，我深刻体会到了超参数对模型训练效果的关键作用。一开始，调整超参数就像在黑暗中摸索，一个小的改动可能带来意想不到的结果，甚至因为其他超参数的影响而适得其反，这让我有些不知所措。但幸运的是，小组讨论给了我新的思路，大家一起分析、尝试，逐渐找到了超参数之间的平衡，使模型性能得到显著提升。通过这次经历，我明白了团队协作的重要性，也学会了在复杂的参数调整中保持耐心和细心，这对我今后的学习和研究有着很大的帮助。

**附录：代码**

import time

import random

import math

import copy

BOARD\_SIZE = 8 #棋盘行数与列数

PLAYER\_NUM = -1 #在board中代表玩家的数字

COMPUTER\_NUM = 1 #在board中代表带电脑的数字

MAX\_THINK\_TIME = 5  #电脑的最大思考时间

# 表示棋盘坐标点的8个不同方向坐标，比如方向坐标[0][1]则表示坐标点的正上方。

direction = [(-1, 0), (-1, 1), (0, 1), (1, 1), (1, 0), (1, -1), (0, -1), (-1, -1)]

BLACK\_NUM = -1 #代表黑棋的数字

WHITE\_NUM = 1 #代表白棋的数字

PATHROOT = [] #节点树

#蒙特卡洛算法部分：

# 返回棋子数

def countTile(board,tile):

    stones = 0

    negstones = 0

    for i in range(0, BOARD\_SIZE):

        for j in range(0, BOARD\_SIZE):

            if board[i][j] == tile:

                stones+=1

            elif board[i][j] == -tile:

                negstones+=1

return stones,negstones

# 返回一个颜色棋子可落子位置

def possible\_positions(board, tile):

    positions = []

    for i in range(0, BOARD\_SIZE):

        for j in range(0, BOARD\_SIZE):

            if board[i][j] != 0:

                continue

            if isok(board, tile, i, j):

                positions.append((i, j))

return positions

#检测对应位置是否在棋盘

def isOnBoard(x, y):

return x >= 0 and x <= 7 and y >= 0 and y <= 7

#检测该位置是否可以落子

def isok(board,tile,i,j):

    change = -tile

    if(board[i][j]!=0):

        return False

    for xdirection, ydirection in direction:

        x, y = i, j

        x += xdirection

        y += ydirection

        if isOnBoard(x, y) and board[x][y] == change: #该点朝一dirction方向相邻一个棋子，且相邻的棋子为可以被翻转的数字

            # 一直走到出界或不是对方棋子的位置

            while board[x][y] == change:

                x += xdirection

                y += ydirection

                if not isOnBoard(x, y):

                    break

            # 出界了，则直接进行下一个方向的查找

            if not isOnBoard(x, y):

                continue

            # 是自己的棋子，中间的所有棋子都要翻转

            if board[x][y] == tile:

                return True

return False

# 是否是合法走法，如果合法返回需要翻转的棋子列表

def updateBoard(board, tile, i, j):

    change = -tile

    need\_turn = [] # 要被翻转的棋子

    for xdirection, ydirection in direction:

        x, y = i, j

        x += xdirection

        y += ydirection

        if isOnBoard(x, y) and board[x][y] == change: #该点朝一dirction方向相邻一个棋子，且相邻的棋子为可以被翻转的数字

            # 一直走到出界或不是对方棋子的位置

            while board[x][y] == change:

                x += xdirection

                y += ydirection

                if not isOnBoard(x, y):

                    break

            # 出界了，则直接进行下一个方向的查找

            if not isOnBoard(x, y):

                continue

            # 是自己的棋子，中间的所有棋子都要翻转

            if board[x][y] == tile:

                while True:

                    x -= xdirection

                    y -= ydirection

                    # 回到了起点则结束

                    if x == i and y == j:

                        break

                    # 需要翻转的棋子

                    need\_turn.append([x, y])

    # 翻转棋子

    board[i][j] = tile

    for x, y in need\_turn:

        board[x][y] = tile

return len(need\_turn)

# 上面的代码用于返回要被翻转的棋子个数，这里的代码用于将更新后的棋盘输出

def updateAIBoard(board, tile, i, j):

    change = -tile

    need\_turn = [] # 要被翻转的棋子

    for xdirection, ydirection in direction:

        x, y = i, j

        x += xdirection

        y += ydirection

        if isOnBoard(x, y) and board[x][y] == change: #该点朝一dirction方向相邻一个棋子，且相邻的棋子为可以被翻转的数字

            # 一直走到出界或不是对方棋子的位置

            while board[x][y] == change:

                x += xdirection

                y += ydirection

                if not isOnBoard(x, y):

                    break

            # 出界了，则直接进行下一个方向的查找

            if not isOnBoard(x, y):

                continue

            # 是自己的棋子，中间的所有棋子都要翻转

            if board[x][y] == tile:

                while True:

                    x -= xdirection

                    y -= ydirection

                    # 回到了起点则结束

                    if x == i and y == j:

                        break

                    # 需要翻转的棋子

                    need\_turn.append([x, y])

    # 翻转棋子

    board[i][j] = tile

    for x, y in need\_turn:

        board[x][y] = tile

return board

def updatePathRoot(i,j):

    global PATHROOT

    for n\_tuple in PATHROOT:

        #找到最佳路径中此节点对应的子节点

        parent, t\_playout, reward, t\_childrens = n\_tuple

        if i == parent[0] and j == parent[1]:

            PATHROOT = t\_childrens

            Break

# 蒙特卡洛树搜索

def mctsNextPosition(board,ucb\_c,playerNum): #棋盘、ucb公式中常数c的值

    def ucb(node\_tuple, t): #t为进行循环的次数

        #  返回各个结点用于进行ucb算法的值

        name, nplayout, reward, childrens = node\_tuple #四个值分别对应 落点位置、模拟对局次数 、赢的次数、子节点

        if nplayout == 0: #避免意外情况

            nplayout = 1

        if t == 0:#避免意外情况

            t = 1

        #reward 是赢的次数 nplayout是模拟对局次数,cval是常数

        return (reward / nplayout) + ucb\_c \* math.sqrt(math.log(t) / nplayout)

    def find\_playout(tep\_board, tile, depth=0): #对tep\_board进行了系列随机落点后，返回最终结果胜负

        def eval\_board(tep\_board): #比较二者的棋子数目，判断胜负

            tileNum,negTilenum = countTile(tep\_board,playerNum)

            if tileNum > negTilenum:

                #tile代表的棋胜

                return True

                #tile代表的棋负

            return False

        while(depth<120):#进行最多120次递归后返回结果

            turn\_positions = possible\_positions(tep\_board, tile)

            if len(turn\_positions) == 0: #如果没位置下棋，切换到对手回合

                tile = -tile

                neg\_turn\_positions = possible\_positions(tep\_board, tile)

                if len(neg\_turn\_positions) == 0: #对方也没位置下棋，结束游戏

                    return eval\_board(tep\_board)

                else:

                    turn\_positions = neg\_turn\_positions

            temp = turn\_positions[random.randrange(0, len(turn\_positions))] # 随机放置一个棋子

            updateBoard(tep\_board, tile, temp[0], temp[1])

            # 转换轮次

            tile = -tile

            depth+=1

        return eval\_board(tep\_board)

    #扩展结点,返回tep\_board的子节点数组

    def expand(tep\_board, tile):

        positions = possible\_positions(tep\_board, tile)

        result = []

        for temp in positions:

            result.append((temp, 0, 0, []))

        return result

    def find\_path(root):

        current\_path = []

        child = root

        parent\_playout = 0

        for item in child: #计算父节点遍历过的次数

            name, nplayout, reward, childrens = item

            parent\_playout+=nplayout

        isMCTSTurn = True

        while True:

            if len(child) == 0: #无可落子的位置,直接结束

                break

            maxidxlist = [0]

            cidx = 0

            if isMCTSTurn:

                maxval = -1

            else:

                maxval = 2

            for n\_tuple in child:  #对每一个可落子的位置进行最大最小搜索

                #实现最大最小搜索，电脑选择最大值，玩家选择最小值

                if isMCTSTurn:

                    #ucb返回各个结点的值，之后就依靠这个值来进行最大最小算法

                    cval = ucb(n\_tuple, parent\_playout)

                    if cval >= maxval:

                        # 获取子结点中值最大的一项,并记录其id(即cidx)

                        if cval == maxval:

                            maxidxlist.append(cidx)

                        else:

                            maxidxlist = [cidx]

                            maxval = cval

                else:

                    cval = ucb(n\_tuple, parent\_playout)

                    if cval <= maxval:

                        #获取子节点中值最小的一项

                        if cval == maxval:

                            maxidxlist.append(cidx)

                        else:

                            maxidxlist = [cidx]

                            maxval = cval

                cidx += 1

            # 从最值结点中随机选择一处落子

            maxidx = maxidxlist[random.randrange(0, len(maxidxlist))]

            parent, t\_playout, reward, t\_childrens = child[maxidx]

            current\_path.append(parent)

            parent\_playout = t\_playout

            # 选择子节点进入下一次循环

            child = t\_childrens

            isMCTSTurn = not (isMCTSTurn)

        # 返回根据最大最小规则选择出来的一条路径

        return current\_path

    global PATHROOT #节点树

    if len(PATHROOT) == 0:

        PATHROOT = expand(board,playerNum)

        for index, rootChild in enumerate(PATHROOT):

            current\_board =  copy.deepcopy(board) #current\_board记录在某处落子后的棋盘

            parent, t\_playout, reward, t\_childrens = rootChild

            updateBoard(current\_board, playerNum, parent[0] , parent[1]) #对落子于此处的棋盘进行随机落子，使得能对其使用ucb算法（避免除以0的情况）

            t\_playout =10

            reward = 0

            for i in range(1,21):

                current\_board2 = copy.deepcopy(current\_board) #current\_board2是用来进行随机落点判断胜负的临时表盘

                isWon = find\_playout(current\_board2, -playerNum) #tile表示下一步谁执行

                if(isWon):

                    reward+=1

            PATHROOT[index] = (parent,t\_playout,reward,t\_childrens)

    #记时，防止循环时间过长

    start\_time = time.time()

    slectTime = 0 #选择过程耗费的时间

    expendTime = 0#扩展过程耗费的时间

    simulationTime = 0 #模拟过程耗费的时间

BackTime = 0 #回溯过程耗费的时间

    simulationTimes = 0

looptime = 0

    for loop in range(0, 30): #总的遍历

        looptime += 1

        current\_board =  copy.deepcopy(board) #current\_board记录在某处落子后的棋盘

        # 思考最大时间限制

        if (time.time() - start\_time) >= MAX\_THINK\_TIME:

            Break

        # current\_path是一个放置棋子的位置列表，根据此列表进行后续操作

        tempStartTime = time.time() #选择过程

        current\_path = find\_path(PATHROOT) # find\_path返回:ucb算法基于root蕴含的信息,获取的最佳路径(从头结点开始的，最佳子节点在各级child数组中的index数组)，

        tempEndTime = time.time()

        slectTime += tempEndTime-tempStartTime

        tile = playerNum

        for temp in current\_path:

            #将通过ucb算法得到的路径整合到一个初始棋盘中

            updateBoard(current\_board, tile, temp[0], temp[1]) #最终current\_board为根据路径落子得到的棋盘

            tile = -tile

        #扩展与模拟过程

        child = PATHROOT

        randomTime = 0 #进行随机落子的盘数

        rewardSum = 0 #胜利总次数

        for temp in current\_path:

            #遍历最佳路径

            idx = 0

            for n\_tuple in child:

                #找到最佳路径中此节点对应的子节点

                parent, t\_playout, reward, t\_childrens = n\_tuple

                if temp[0] == parent[0] and temp[1] == parent[1]:

                    break

                idx += 1

            if temp[0] == parent[0] and temp[1] == parent[1]:

                if len(t\_childrens) == 0:

                    #找到路径的叶子结点，进行拓展

                    tempStartTime = time.time()

                    t\_childrens = expand(current\_board, tile) #扩展过程

                    tempEndTime = time.time()

                    expendTime += tempEndTime-tempStartTime

                    randomTime = len(t\_childrens) \*10 #进行随机落子的盘数

                    rewardSum = 0 #胜利总次数

                    tempStartTime = time.time() #模拟过程

                    for index, rootChild in enumerate(t\_childrens):#对落子于此处的棋盘进行随机落子，使得能对其使用ucb算法（避免除以0的情况）

                        child\_board =  copy.deepcopy(current\_board) #current\_board记录在某处落子后的棋盘

                        child\_parent, child\_playout, reward, child\_childrens = rootChild

                        tempTile = tile

                        tempNegTile = -tempTile

                        updateBoard(child\_board, tempTile, child\_parent[0] , child\_parent[1])

                        child\_playout =10

                        reward = 0

                        for i in range(1,21):

                            current\_board2 = copy.deepcopy(child\_board) #current\_board2是用来进行随机落点判断胜负的临时表盘

                            simulationTimes+=1

                            isWon = find\_playout(current\_board2, tempNegTile) #tile表示下一步谁执行

                            if(isWon):

                                reward+=1

                        rewardSum+=reward

                        t\_childrens[index] = (child\_parent,child\_playout,reward,child\_childrens)

                    tempEndTime = time.time()

                    simulationTime += tempEndTime-tempStartTime

                #应用修改到本体

                child[idx] = (parent, t\_playout, reward, t\_childrens)

            #继续处理子结点

            child = t\_childrens

        if randomTime!=0:

            tempStartTime = time.time() #反向传播过程

            child = PATHROOT

            for temp in current\_path:

                #遍历最佳路径

                idx = 0

                for n\_tuple in child:

                    #找到最佳路径中此节点对应的子节点

                    parent, t\_playout, reward, t\_childrens = n\_tuple

                    if temp[0] == parent[0] and temp[1] == parent[1]:

                        break

                    idx += 1

                if temp[0] == parent[0] and temp[1] == parent[1]:

                    #找到了对应的结点，修改场数与胜利数

                    t\_playout += randomTime

                    reward += rewardSum

                    #应用修改到本体

                    child[idx] = (parent, t\_playout, reward, t\_childrens)

                #继续处理子结点

                child = t\_childrens

            tempEndTime = time.time()

            BackTime += tempEndTime-tempStartTime

    max\_avg\_reward = -1

    mt\_result = (0, 0)

    for n\_tuple in PATHROOT:

        parent, t\_playout, reward, t\_childrens = n\_tuple

        if (t\_playout > 0) and (reward / t\_playout > max\_avg\_reward):

            mt\_result = parent

            max\_avg\_reward = reward / t\_playout

    print(mt\_result)

return mt\_result

#形式化输出格式转换部分：

#将输入的棋盘转换为算法可以识别的0，1，-1字典形式

def convert\_board(board):

    AIboard = {}

    for row\_index in range(8):

        AIboard[row\_index] = {}  # 先为每一行创建一个空字典，用于存放该行各列的棋子对应值

        for col\_index in range(8):

            element = board[row\_index][col\_index]

            if element == '.':

                AIboard[row\_index][col\_index] = 0

            elif element == 'X':

                AIboard[row\_index][col\_index] = -1

            elif element == 'O':

                AIboard[row\_index][col\_index] = 1

    return AIboard

#将算法输出的坐标转换为如G6的字符串输出

def coordinate\_to\_label(coord):

    # 定义列的字母映射

    columns = {0: 'A', 1: 'B', 2: 'C', 3: 'D', 4: 'E', 5: 'F', 6: 'G', 7: 'H'}

    # 解析坐标，获取列和行的索引

    col\_index, row\_index = coord

    # 获取列的字母

    row\_letter = columns[row\_index]

    result = row\_letter + str(col\_index + 1)

    # 构建输出的标签格式，如 "G6"

return result

class AIPlayer:

    """

    AI 玩家

    """

    def \_\_init\_\_(self, color):

        """

        玩家初始化

        :param color: 下棋方，'X' - 黑棋，'O' - 白棋

        """

        #初始化上次棋盘记录

        self.last\_AIboard = None

        #初始化玩家执黑or执白

        self.color = color

    def get\_move(self, board):

        """

        根据当前棋盘状态获取最佳落子位置

        :param board: 棋盘

        :return: action 最佳落子位置, e.g. 'A1'

        """

        if self.color == 'X':

            PLAYER\_NUM = 1

            COMPUTER\_NUM = -1

        else:

            PLAYER\_NUM = -1

            COMPUTER\_NUM = 1

        if self.color == 'X':

            player\_name = '黑棋'

        else:

            player\_name = '白棋'

        print("请等一会，对方 {}-{} 正在思考中...".format(player\_name, self.color))

        # 将传入的board转换为算法可识别的格式

        AIboard = convert\_board(board)

        if self.last\_AIboard is not None:

            # 用于记录旧AIboard中-1的位置（坐标形式，如 (row, col)）

            old\_minus\_ones\_positions = []

            # 用于记录新AIboard中-1的位置（坐标形式，如 (row, col)）

            new\_minus\_ones\_positions = []

            # 统计旧AIboard中-1的数量

            old\_minus\_ones\_count = 0

            # 统计新AIboard中-1的数量

            new\_minus\_ones\_count = 0

            for row in range(len(self.last\_AIboard)):

                for col in range(len(self.last\_AIboard[row])):

                    if self.last\_AIboard[col][row] == 0:

                        old\_minus\_ones\_count += 1

                        old\_minus\_ones\_positions.append((row, col))

            for row in range(len(AIboard)):

                for col in range(len(AIboard[row])):

                    if AIboard[col][row] == 0:

                        new\_minus\_ones\_count += 1

                        new\_minus\_ones\_positions.append((row, col))

            # 找出缺少的 -1 在字典中的具体行列数（即那些在旧AIboard中是-1但在新AIboard中不是的位置）

            missing\_minus\_ones\_positions = [pos for pos in old\_minus\_ones\_positions if pos not in new\_minus\_ones\_positions]

            for pos in missing\_minus\_ones\_positions:

                updateBoard(self.last\_AIboard, PLAYER\_NUM, pos[1], pos[0])

                updatePathRoot(pos[1], pos[0])  # 更新pathRoot

        # 获取此时机器可以落子的结点

        mcts\_possibility = len(possible\_positions(AIboard, COMPUTER\_NUM))

        print(possible\_positions(AIboard, COMPUTER\_NUM))

        # 判断机器是否有棋可下

        if mcts\_possibility == 0:

            return None  # 如果没位置可下，返回None表示无合适走法

        # 根据mcts算法获取落子位置

        stone\_pos = mctsNextPosition(AIboard, 0.7, COMPUTER\_NUM)

        updateBoard(AIboard, COMPUTER\_NUM, stone\_pos[0], stone\_pos[1])

        AIboard = updateAIBoard(AIboard, COMPUTER\_NUM, stone\_pos[0], stone\_pos[1])

        updatePathRoot(stone\_pos[0], stone\_pos[1])  # 更新pathRoot

        # 这里可以根据需求决定是否返回格式化后的位置字符串，如果需要可调用coordinate\_to\_label函数转换

        action = coordinate\_to\_label(stone\_pos)

        self.last\_AIboard = AIboard  # 更新保存的上次的操作后的棋盘

        return action