人工智能大作业第一阶段报告：基于蒙特卡洛搜索树的 AI

林子开 21307110161 鞠扬 21300180032 苏宇骢 212300180030

2023 年 10 月 28 日

摘要 基于蒙特卡洛搜索树设计了四子棋博弈 AI。本报告说明了算法框架和具体的 Python 实现，汇总了 AI 的对战情况，并介绍了本小组的创新与不足。

目录

1 导言

四子棋是如下的二人零和博弈游戏：两名玩家在 7×6 的棋盘上轮流行动，每次选择未满的一列落子；棋子会自动下落到该列棋子的底部；率先完成四子连成一线（无论是行、列或对角线）的玩家获胜；如果所有列都填满而仍未分出胜负，则双方战平。在四子棋游戏中，如果先手下棋，则存在完美的必胜的策略。但是，完美的落子策略需要大量的存算代价，不切实际。因此，我们利用蒙特卡罗树搜索算法（MCST）设计了一个 AI 模型，该模型通过计算大量的模拟，能够帮助我们在四子棋博弈中逼近完美落子策略。

2 算法框架简介

蒙特卡罗树搜索算法（MCTS）分为四个阶段：选择，扩展，模拟，回溯。以下是我们蒙特卡洛树搜索算法的示意图：

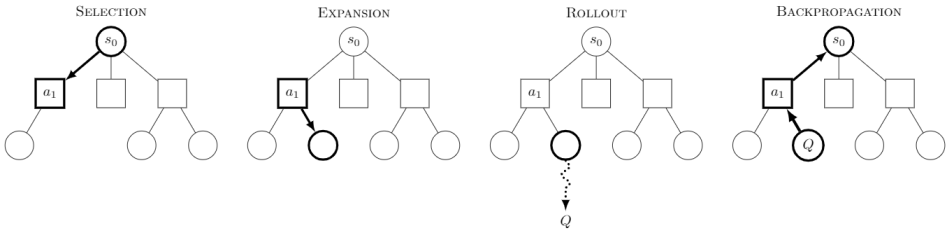


图 1: 蒙特卡洛搜索树示意图：选择、扩展、模拟、回溯

第一阶段：选择 Selection　从根节点 R 开始，按照 UCT 准则（详后）递归选择最优的子节点，最终到达一个叶子节点 L。根节点为目前游戏状态，叶子节点 L 为一个有潜在子节点的任意节点（未经过模拟），或者某个代表棋局结束的节点。

第二阶段：扩展 Expansion　除非任意一方的输赢使得游戏在 L 结束，否则创造一个或多个子节点并从中选择一个节点 C。子节点只要是由 L 定义的任意有效的移动即可。

第三阶段：模拟 Simulation　再从节点 C 开始用随机策略进行一轮游戏，直到分出胜负，或者平局。

第四阶段：回溯 Back-propagation　利用模拟游戏的结果，基于 negamax 思想，更新由 C 至 R 路径上的经验胜率。

下图是我们组设计 MCTS 算法时参考的算法框架，图片来源于北京大学《人工智能原理》的 mooc 网课，特此感谢。

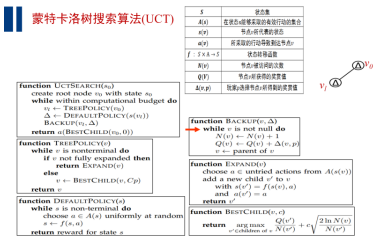


图 2: 蒙特卡洛搜索树算法框架（来源：北京大学《人工智能导论》mooc 网课）

3　算法实现：Python codes 详细介绍

3.1　基础工作：节点 node，搜索树 MCTree，主函数 MCST-agent 与其他辅助函数

node 类　我们组首先定义了一个 node 类，node 类是一个用于表示游戏状态的节点的类，存储了以下属性：父节点、当前棋局状态、未扩展的潜在子节点，、已扩展的子节点、当前棋手（先手/后手）、从父节点到达该节点所采取的落子 action、当前节点的访问次数、相对于父节点的胜利次数（基于 Negamax 思想，详后）、当前棋局是否结束等。

此外，node 类还包含了以下方法：isTerminal()，判断该节点是否为终止节点。fullyExpanded()：判断该节点是否被完全扩展。findBestChild(n, c)：基于 UCT 从所有子节点中选取最具有探索价值的子节点。expand()：扩展当前节点，随机生成当前节点的一个子节点。simulate()：对当前节点进行蒙特卡洛模拟。backup(reward)：反向传播，使用 negamax 策略。

node 类的主要作用是在蒙特卡洛树搜索算法中表示游戏状态的节点，用于存储和更新游戏状态的信息。此外，蒙特卡洛模拟和反向传播也是基于 node 类进行的。

MCTree 类 基于 node 类，我们定义了 MCTree 类，这个类是用于构建蒙特卡洛树，具有 root 根节点。MCTree 类还包含了 findNodeToSimulate(n, c)，该方法从根节点开始递归地进行寻找最具有探索价值的节点，根据 UCT 值，返回一个节点。

MCTree 类的主要作用是构建和管理蒙特卡洛树，我们的蒙特卡洛模拟算法是基于 MCTree 类进行的。

主函数 MCST-agent 主函数 MCST-agent 接收两个参数：obs 和 config。obs 是当前游戏的状态，config 则是游戏的配置信息。主函数按以下步骤选择落子位置：

- Step 1：主函数首先读入棋盘状态和配置信息，包括当前玩家的标记和棋盘的列数等。同时，将一维的棋盘转换为二维的棋盘。
- Step 2：如果我方是第一次走子，则直接在棋盘中间区域落子（**理由详后**），不进行蒙特卡洛搜索；否则，则根据当前棋盘状态为本次落子分配超额思考时间（**分配方式详后**）。
- Step 3：主函数检查当前棋盘是否只有一列可下，如果是，则没有进行蒙特卡洛搜索的必要，直接返回该列的索引。
- Step 4：主函数检查是否存在落子位置可以使我方立刻获胜，如果有，则没有进行蒙特卡洛搜索的必要，直接返回能使我方胜利的落子位置的索引。
- Step 5：接着，函数检查是否存在某个落子位置，**如果对手在此位置落子，则对手将获胜**，如果存在这样的落子位置，则立刻返回该位置的索引，**阻止对手获胜**。
- Step 6：如果当前棋局通过了以上检查，则说明需要进行蒙特卡洛模拟。该函数将为蒙特卡洛树创建一个根节点。然后按照以下步骤（**具体介绍详后**）进行蒙特卡洛模拟，直到消耗完本次思考时间：
  * Step 6.1 基于 UCT 准则选择一个节点
  * Step 6.2 对该节点进行扩展
  * Step 6.3 从新扩展的节点出发，快速模拟落子，直到棋局结束
  * Step 6.4 根据模拟结果，基于 Negamax 思想，反向传播，更新各节点的统计信息
- Step 7：函数选择根节点的具有最大胜率的子节点，返回该子节点索引。如果在该步骤出现任何意外的错误，函数将随机选择一个可以落子的列返回，防止因为函数报错而输掉比赛。

辅助函数：落子 put-a-piece 函数负责落子，接受三个参数：board 表示当前的棋盘状态，col 表示要落子的列，player 表示玩家是先手/后手的标识（用 1 或 2 标志）。该函数返回落子后的棋盘状态。

辅助函数：获取可行列 get-legal-move 函数负责判断还有哪些列可以落子。该函数接受一个参数 board，表示当前的棋盘状态，以列表形式返回所有可以落子的列的索引。

辅助函数：棋局胜负判断 get-winner 函数负责判断在给定的棋盘状态下，是否有玩家获胜。该函数接受两个参数：board 表示当前的棋盘状态，action 表示最近一次落子所在的列。由于在两个相邻的棋盘状态之间只相差一个落子，因此，只需要检查该棋子分别检查**棋子所在的列、行、以及两个对角线方向上的连续棋子个数**（如图3所示），即可判断棋局的胜负，**从而节省判断用时，为蒙特卡洛模拟留出更多的时间**。如果任何一个方向上的连续棋子个数大于等于设定的连续棋子个数（inarow），则返回该棋子所属的玩家（1 或 2）。如果棋盘已经摆满且没有玩家获胜，则返回 0 表示平局。如果以上条件都不满足，则返回 None 表示游戏还未结束。

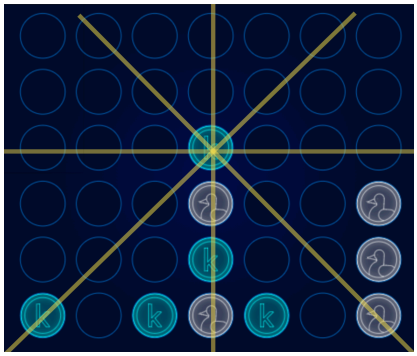


图 3: get-winner 函数检查当前棋局胜负的示意图，四条直线相交处为刚刚落下的棋子

3.2 选择：基于 UCT 准则

在节点的选择过程中，我们使用 UCT 来评估每个节点的“探索价值”。

我们使用的 UCT 公式为

$$\frac{w_i}{n_i} + c\sqrt{\frac{2\log(t)}{n_i}}$$

其中超参数 c 是一个常数，c 越大则搜索树会越宽，但也会越浅，c 一般取 1。该公式前一项表示当前节点的经验胜率（基于极大似然估计），后一项代表前节点的胜率的不确定性，可以反映出当前节点的“探索价值”（当该节点的模拟次数较少时，认为该节点值得继续被探索）。其中 w_i 表示当前节点曾经取胜的次数；n_i 表示当前节点曾经的模拟次数，w_i/n_i 可表示该节点经验胜率大小）；t 表示总共的模拟次数。注意到当 n_i 较小时，后一项会较大，说明此时该节点被探索的次数较少，胜率的不确定性仍然较大，该节点具有更多的“探索价值”。

UCT 值综合考虑了节点的获胜次数和访问次数，能防止出现某些最初被判断为高胜率的节点被探索多次，而某些最初被判定为低胜率的节点几乎不再被探索的极端情况。一言以蔽之，UCT 能够有效平衡探索和利用。

由于四子棋是一个**二人的零和博弈问题**，因此，**相邻两层节点存储的经验胜率的意义是相反的**。简单而言，每一个节点（根节点除外）的经验胜率都是用于辅助它的父节点（而不是它自己）进行选择，该经验胜率表示从父节点出发，采取某个能到达该子节点的 action 后的**父节点胜率**。这样就引出了 negamax 的思想，而 negamax 在本质上是一种极大极小策略。在选择阶段，**每向下递归一层，棋手的身份就变化一次**（先手变后手，后手变先手），并且根据当前棋手的身份选择对于该身份而言最优的 action。质言之，在选择阶段，我们将对手也假想为**绝对理性人**，在任何一次选择中都会选择对自己而言最优的落子方式。

此外，我们小组经过观察，发现基于蒙特卡洛的 AI，可能由于随机化的原因，有时出现“考虑不周”的情况，尤其在棋局的后半程，没有及时“发现”对手即将连线的趋势。因此，我们设定 **超参数 c 随着棋局的进行，逐渐线性地从 1 增加到 1.1**，以促进 AI 在比赛的后半程（此时比较邻近比赛终点，潜在的待扩展节点较少，树高会相应地下降），主动拓宽蒙特卡洛搜索树的宽度，优先考虑更多的可能性（而不是优先考虑搜索树的深度），这样能尽早发现并遏制对手连线的趋势。这是我们小组在博弈策略上提出的一种自适应的改进。

3.3 扩展：随机选择

在节点的扩展过程中，从当前节点未被尝试过且可行的动作列表中随机选取一个 action，然后落子，并由此得到其子节点（表征新的棋盘状态），然后返回新生成的子节点。

3.4 模拟：快速落子

在模拟阶段，我们从给定节点出发，按照均匀分布从目前可行的落子位置中抽取某个位置，进行走子，然后转变棋手身份，重复上述过程，直到本次模拟的棋局结束（平局，胜利，或失败）。然后返回棋局的结果。

3.5 反向传播：基于 Negamax 思想

在反向传播中，我们从模拟的起始节点出发，既有 Negamax，依次更改各个祖先节点的经验胜率。**若模拟结果是平局，则各祖先节点的胜利次数都增加 0.5，模拟次数增加 1。如果模拟结果不是平局，则各祖先节点的胜利次数依次交替地增加 0 或者 1，而模拟次数统一增加 1。**

4 实战结果汇总

kaggle 比赛对战情况：截至 2023 年 10 月 28 日晚上 22 点，我们的 AI（提交编码为 34265584）获得的分数为 980.3 分。

与内置 random 棋手对战情况：我们与本地 Kaggle 环境内置的“random”棋手（5 次先手，5 次后手）进行了 10 个回合的博弈，全部胜利。

与内置 negamax 棋手对战情况：我们与本地 Kaggle 环境内置的“Negamax”棋手进行了 10 个汇合的博弈（5 次先手，5 次后手），9 次胜利（5 次先手，4 次后手），1 次平局（后手）。

5 创新点：策略优化与算法提速

在本小节中，我们将我们提出的优化和提速方法总结如下：

引入 numba 库进行提速 我们引入了 numba 库对我们的模拟过程进行提速。经测试，在联想笔记本电脑 Yoga 上，没有引入 numba 前，每秒模拟次数的数量级在 10^3 左右，在引入 numba 库进行加速后，每秒模拟次数的数量级提升到了 10^4 左右。由于我们对各个节点的胜率估计基于极大似然法，因此，越多的模拟次数，越能减小方差，则越能得到稳健的胜率估计，也就越逼近完美的必胜策略。

先查看必胜节点 我们在进行模拟前，会先查看是否存在某个落子位置，可以使我方必胜，如果存在这样的位子，则直接落子，赢得比赛。这样既可以减少模拟时间消耗，也能保证我方 AI 一定胜利，防止由于蒙特卡洛模拟的随机性而忽视（尽管概率很小）这个必胜的落子位置。

及时阻止对手连线 在进行蒙特卡洛模拟之前，AI 也会检查是否存在某个位置，当对手在此位置落子时，对手必胜。如果存在这样的位子，则 AI 在此位置落子，尽可能能阻止对手获胜。不过，这并不能一定阻止对手获胜，一方面，可能存在不只这样的一个对手必胜的落子位置，我方落子只能占据一个位置，而对手落子于另一个位置则同样能获胜。另一方面，可能我方在该位置落子后，又创造了另一个可以使对方必胜的落子位置，此时我方仍然无法获胜。因此，该策略只能尽可能地阻止对手胜利。

第一步基于经验观察，提高胜算，节省时间。 我们在观察同班同学设计的高分数 AI 的基础上，发现第一步落子于中间区域更具有胜算，因此，我们不为第一步落子分配时间，而直接改为下在棋盘的中间区域。该策略来源于人类的观察经验。此外，这样也不占用总的超额思考时间，能为之后的思考留下更多的超额时间。

自适应的超参数 c 增长机制 我们对超参数 c 采取了自适应的增长策略。定义系数

$$k = \frac{n}{N}$$

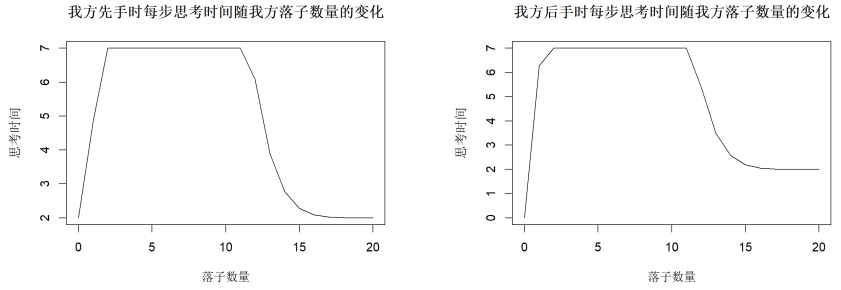
其中 n 是棋盘上已有的双方棋子数之和，而 N 是棋盘上所有的落子位置 (在本次博弈中，$N = 6 \times 7 = 42$)，我们设定 $c = 1 + 0.1 \times k$，则随着棋局的进行，超参数 c 会逐渐从 1 增大到 1.1，也就是 AI 会逐渐更倾向于探索那些模拟次数较少的节点。此时，AI 倾向于优先拓宽搜索树的宽度（而不是深度），我们认为，这样有助于 AI 在棋局的后半程“看到”更多可能性，提前“意识到”对手可能的连线趋势，进而阻止对方获胜。

巧妙的超额时间分配函数 我们观察了本班其他同学的 AI 对战结果，发现大多数比赛都会在棋子填满 3/4 的棋盘之前结束，因此，我们认为，**应该将大多数的超额时间分配在棋局的前半程**，以免棋局结束时还有大量未用的超额时间，造成浪费。此外，越靠近棋局终点，则能探索的潜在节点会越少，**需要的探索时间应该随比赛的进行，按照指数下降的**方式进行分配。最后，由于一开始的探索空间非常之大，而且双方博弈的随机性也很大，我们认为在**一开始就分配大量的超额时间并不明智**，所以我们为一开始的几个节点分配的时间也是较少的。

我们设计的时间分配函数为：

$$t = t_{\text{step}} + \min(k * t_{\text{total}}, 5)$$

其中，t 是为本次思考分配的超额时间，t_{step} 是系统默认的每步思考时间（默认为 2 秒），t_{total} 是剩余的总超额时间，k 的含义与上文一致，用于衡量棋局的进度。基于上述分配机制，我们得到时间分配的关系图如下：



(a) 先手　　(b) 后手

图 4: AI 的每步思考时间

基于这样的时间分配函数，我们的 AI 在前 10 次落子之后，消耗掉大约 43 秒的超额时间，占 3/4 的总超额时间（共 60 秒）。

6 不足与可能的改进

在我们的算法中，每一次探索都重新建立一个蒙特卡洛搜索树，无法利用之前的模拟结果所获得的信息。可以的改进方法是：构建一个集合，用于存储上一轮走子时根节点的子节点。当下一次进行落子时，从上一次的落子构建的蒙特卡洛树中找到对应的子节点，以该子节点作为蒙特卡洛树的根节点，则可充分利用之前模拟的信息。

在我们的模拟中，“近端”与“远端”的输赢，具有相同的权重。比如，20 步之后才会发生的胜利，和 2 步之后就会发生的胜利，对于根节点而言意义是一样的。但是，越靠近根节点的棋局结果，具有越多的信息量。比如，如果 AI 发现 2 步之后就能胜利，AI 可以更倾向于往这个棋局结果去进行落子，则有可能更迅速地获胜。因此，可以借鉴马尔可夫决策模型，考虑在反向传播和模拟过程中引入衰减系数，使得 AI 对靠近当前状态的输赢更加敏感。

尽管我们引入了比较巧妙的时间分配机制，但该机制的最优性仍然值得探讨，比如每次分配的超额时间上限设定为 5 秒是否合适？此外，按照指数下降的方式分配后半程的超额时间，有可能会对后半程的博弈不利。

我们引入了线性的超参数 c 的增长机制。但是，c 从 1 增长到 1.1 是否不足，或是太多，从而导致树太窄，或太宽？此外，c 越大，要想达到同样的深度，则需要更多的思考时间，但按照我们的时间分配机制，棋局的后半程的思考时间迅速减少，这可能与我们的超参数 c 的增长机制相违背。