四子棋实验

熊泽恩 计24

2024 年 5 月 28 日

目录

1 基于 UCT 算法的四子棋策略 **2**

1.1 MCTS 算法 . . . 2

1.2 UCB1 公式 . . . 2

1.3 UCT 算法原理 . . . 3

1.4 UCT 算法实现 . . . 3

1.4.1 UCTSearch . . . 3

1.4.2 TreePolicy . . . 4

1.4.3 Expand . . . 4

1.4.4 DefaultPolicy . . . 4

1.4.5 Backup . . . 4

1.4.6 BestChild . . . 5

1.5 实现细节 . . . 5

2 特殊策略 **5**

2.1 将军/解围策略 . . . 5

2.2 中心优势 . . . 5

2.3 调整权值 . . . 6

3 统计数据 **6**

4 总结与感想 **7**

1 基于 UCT 算法的四子棋策略

我的四子棋策略基于 UCT 算法，即基于上限置信区间的蒙特卡洛树搜索算法，它由 MCTS 算法和 UCB1 算法组成。

1.1 MCTS 算法

蒙特卡洛树搜索算法(Monte Carlo Tree Search) 由四部分构成：

1. 选择(Selection)：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。我们称一个节点是**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；
2. 扩展(Expansion)：根据当前可执行的行动，向选定的节点上添加一个（或多个）子节点以扩展搜索树；
3. 模拟(Simulation)：根据默认策略在扩展出来的一个（或多个）子节点上执行蒙特卡罗模拟，并确定节点的估计值；
4. 反向传播(Backpropagation)：根据模拟结果向上依次更新祖先节点的估计值，并更新其状态。

以下是 MCTS 算法的示意图：

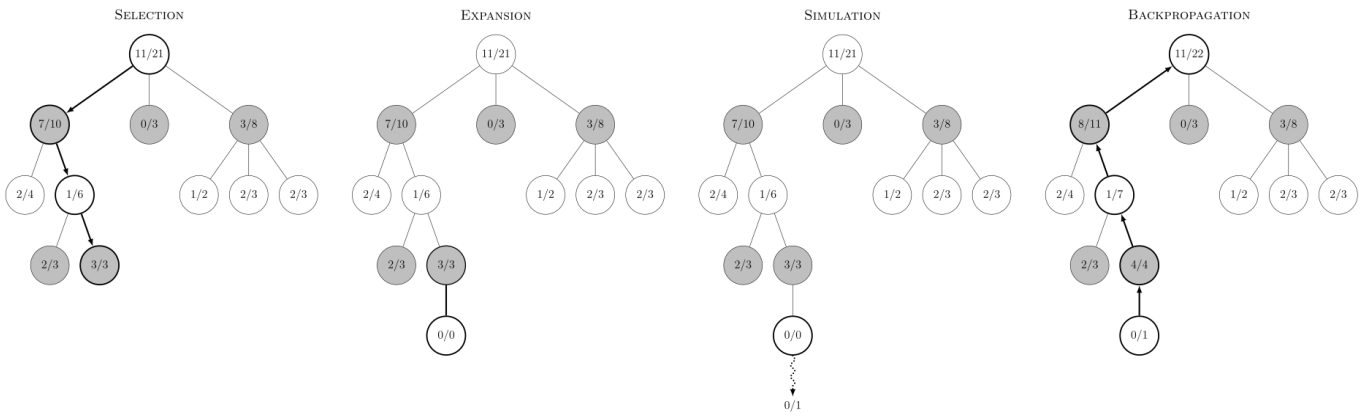

图 1: MCTS 算法示意图 [1]

当确定了扩展节点之后，算法以该节点为根进行大量的随机模拟，并根据模拟的结果确定在该根节点下如何落子，并更新祖先节点的估计值，该策略一般称为默认策略(Default Policy)。在平凡情况下，算法在每一次模拟中从根节点的状态开始，随机地选择一个落子点 x 进行落子，接下来不断地交替随机落子直至棋局结束，并根据结束时双方的胜负状态来确定当前状态下落子点 x 的估计值。当完成大量的模拟之后，对每一种落子点的估计值将变得更为准确，算法以此来决定最终的落子，并向上依次更新祖先节点的估计值。

事实上，我最后采取了特殊策略，并没有完全按照 “随机地选择一个落子点 x” 进行落子，而是

1. 如果当前局面下有可以直接获胜的落子点，则直接落子；
2. 如果对手有可以直接获胜的落子点，则直接落子阻止；
3. 否则，由于 “靠中间的落子点更容易导致赢局”，因此按照某一权值分布选择落子点，中间的权值更高，边缘的权值更低。

1.2 UCB1 公式

上限置信区间(Upper Confidence Bound 1) 公式是解决多臂老虎机问题的一种经典算法，它的核心思想是在探索和利用之间取得平衡，即在已知的信息中选择最优的行动，同时也要保证对未知的行动进行探索。UCB1

公式如下：

$$I(i) = \bar{X}_i + \sqrt{\frac{2\ln n}{T_i(n)}} \tag{1}$$

其中，$\bar{X}_i$ 是手臂 i 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，$T_i(n)$ 是手臂 i 到目前为止总共所访问的次数。可以理解为，(1) 式中前一部分 X_i 就是对到目前为止已经搜集到的知识的价值，而后一部分则可以看作是尚未充分探索过的节点需要继续探索的必要性。

1.3 UCT 算法原理

我最终结合 MCTS 算法和 UCB1 公式，使用了基于上限置信区间的蒙特卡洛树搜索算法 (Upper Confidence Bound 1 Applied to Tree, UCT) 来解决四子棋问题。 UCT 算法的核心在于使用UCB公式来指导搜索过程，通过计算每个可选步骤的置信上界来决定下一步的行动。这种方法结合了先深后广的搜索策略，即先对每个步骤进行深度搜索，然后根据UCB公式的计算结果来扩展搜索范围，从而在有限的计算资源内找到相对较优的解决方案。

此时 UCT 中 UCB1 算法的公式如下：

$$I_v(v') = \frac{Q(v')}{N(v')} + c\sqrt{\frac{2\ln N(v)}{N(v')}} \tag{2}$$

其中 v 是当前节点，v' 是 v 的子节点，$v \to v'$ 表示的就是一次决策(模拟)。 $Q(\cdot)$ 是回报函数，在四子棋中，我使用**获胜次数**这一指标， $N(\cdot)$ 是节点的访问次数。

c 是一个常数，用于平衡探索和利用；理论最优解为 $c = \sqrt{2}$。

1.4 UCT 算法实现

以下是 UCT 算法的伪代码：其中 TreePolicy 用于选择节点，Expand 用于扩展节点， DefaultPolicy 用于模拟，Backup 用于反向传播，BestChild 用于选择最优子节点。

$s(\cdot)$ 表示节点的状态，$f(\cdot)$ 表示状态转移函数，$N(\cdot)$ 表示节点的访问次数， $Q(\cdot)$ 表示节点的回报。

1.4.1 UCTSearch

UCTSearch 函数是 UCT 算法的主函数，用于执行 UCT 算法。

Algorithm 1: UCT Algorithm - UCTSearch

```
function UCTSearch(s_0)
    create root node v_0 with state s_0
    while within computational budget do
        v_l ← TreePolicy(v_0)
        Δ ← DefaultPolicy(v_l)
        Backup(v_l, Δ)
    end while
    return BestChild(v_0, 0)
end function
```

1.4.2 TreePolicy

TreePolicy 函数用于选择节点，即根据 UCB1 公式选择最优的子节点。

Algorithm 2: UCT Algorithm - TreePolicy

function TreePolicy(v)
  while v is not terminal **do**
    if v has expandable children **then**
      return Expand(v)
    else
      $v \leftarrow$ BestChild(v, c)
    end if
  end while
  return v
end function

1.4.3 Expand

Expand 函数用于扩展节点，即根据当前节点的状态，选择一个未尝试过的动作，并创建一个新的子节点。

Algorithm 3: UCT Algorithm - Expand

function Expand(v)
  choose a random untried action a, the next state is s'
  $s' \leftarrow f(s(v), a)$
  create a new child v' of v with state s'
  return v'
end function

1.4.4 DefaultPolicy

DefaultPolicy 函数用于模拟，即根据默认策略模拟一次决策。

Algorithm 4: UCT Algorithm - DefaultPolicy

function DefaultPolicy(v)
  while the state of v is not a terminal state **do**
    select a legal action a at random
    $s \leftarrow f(s, a)$
  end while
  return the reward for state s
end function

1.4.5 Backup

Backup 函数用于反向传播，即根据模拟的结果向上依次更新祖先节点的估计值。

Algorithm 5: UCT Algorithm - Backup

function BACKUP(v, Δ)
  while $v \neq$ null **do**
    $N(v) \leftarrow N(v) + 1$
    $Q(v) \leftarrow Q(v) + \Delta(v, p)$
    $v \leftarrow$ the parent of v
  end while
end function

1.4.6 BestChild

BestChild 函数用于选择最优子节点，即根据 UCB1 公式选择最优的子节点。

Algorithm 6: UCT Algorithm - BestChild

function BESTCHILD(v, c)
  return $\arg\max_{v' \in \text{ children of } v} \frac{Q(v')}{N(v')} + c\sqrt{\frac{2\ln N(v)}{N(v')}}$
end function

1.5 实现细节

我使用 `C++` 语言实现了 UCT 算法，具体代码位于 `src/UCT.cpp` 中。

2 特殊策略

除了朴素的 UCT 算法之外，在实际实现中，我通过查找资料 [4]，发现一些特殊策略可以提高最终结果的准确性。

2.1 将军/解围策略

在四子棋中，如果对手有可以直接获胜的落子点，那么我们应该优先选择这个落子点，以阻止对手获胜。这一策略称为将军/解围策略。

2.2 中心优势

在 [4] 中提到，中心的落子点更容易导致赢局。类似于围棋，在四子棋的对弈过程中，由于重力原因，四子棋的棋盘中央即为棋盘底部的中央，于是我考虑 defaultPolicy 随机走子的基础上，更改底部各子的权重，增大中部的权重，更容易将棋子下在中部，这样下出妙手的可能性更高，提高了有效模拟的比例。

具体而言，假设棋盘宽度为 w，则定义

$$\text{weight}(i) = \begin{cases} (i+1)^2, & i \leq n/2, \\ \text{weight}(w-i-1), & \text{otherwise.} \end{cases}$$

在 DefaultPolicy 中，我按照这一权值分布选择落子点。具体而言，先随机选取 $w_0 \in [0, \sum_i \text{weight}(i)]$ 的值，再根据权值分布选择第一个 **可行**且满足 $\sum_j \text{weight}(j) \leq w_0$ 的 j。

2.3 调整权值

注意到 UCT 算法中的 c 常数是用于平衡探索和利用的，因此我在实际实现中也对 c 进行了调整。不同 c 值对最终结果的影响如下表所示：

c	胜率
0.500	0.57
0.600	0.6
0.650	0.85
0.707	0.95
1.000	0.6

表 1: 不同 c 值对最终结果的影响

因此我在实际实现中选择了 $c = 0.707$。

3 统计数据

最终在 Saiblo 平台上进行的批量测试结果如下图所示：

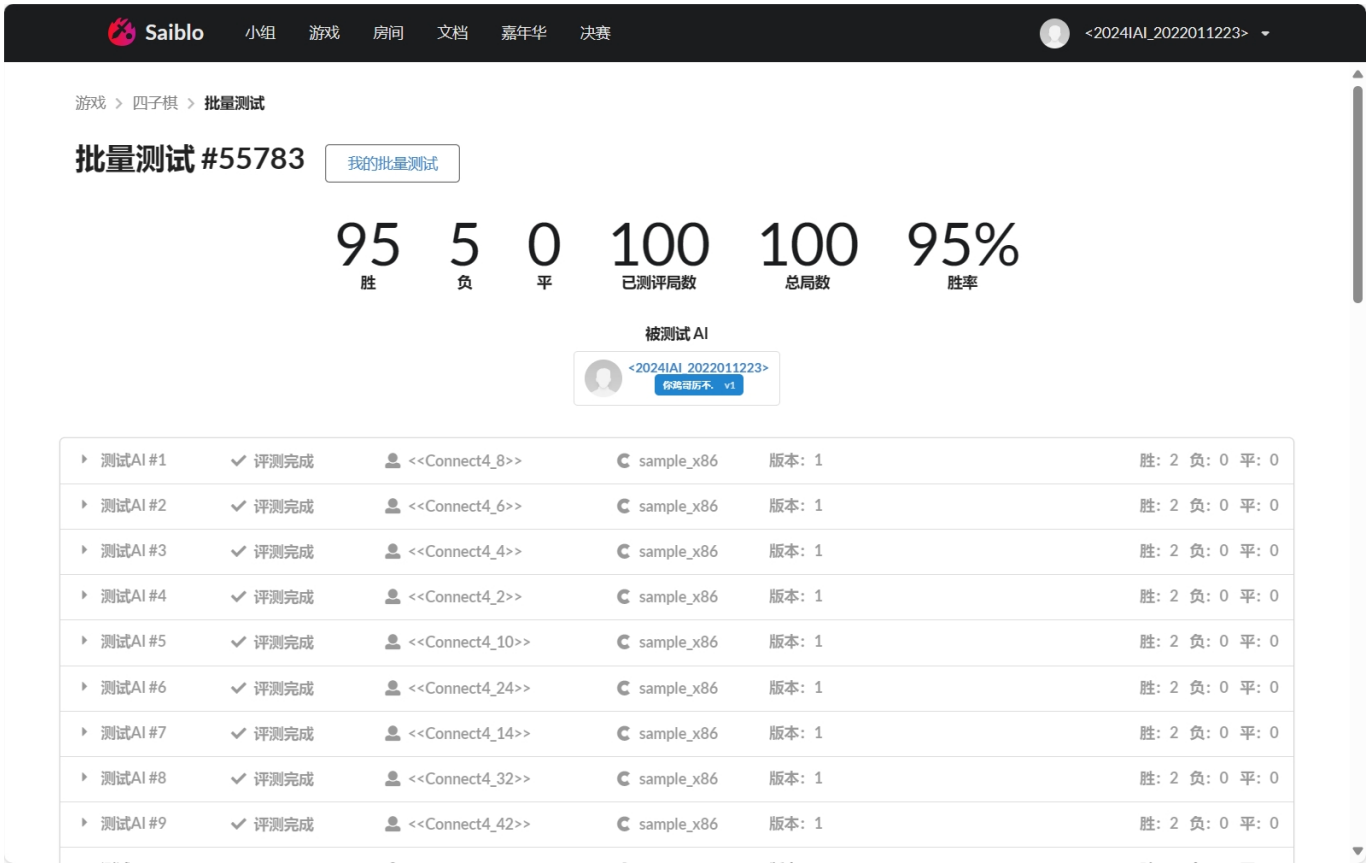

图 2: 统计数据

4 总结与感想

通过本次实验，我对蒙特卡洛树搜索算法有了更深入的了解，同时也学会了如何将其应用到四子棋问题中。在实现过程中，我遇到了很多困难，比如如何设计状态转移函数、如何设计回报函数、如何设计默认策略等等。通过不断地调试和修改，我最终实现了一个相对较好的四子棋策略。

在实现过程中，我发现了一些问题，比如在模拟过程中，由于随机性较大，有时候会出现不合理的落子，导致最终结果不够准确。因此，我在模拟过程中加入了一些特殊策略，比如将军/解围策略、中心优势等等，以提高最终结果的准确性。

参考文献

[1] Robert Moss, *File:MCTS-steps.svg - Wikimedia Commons*, 2020, https://commons.wikimedia.org/w/index.php?curid=88889583.

[2] *Monte Carlo tree search - Wikipedia*, 2023, https://en.wikipedia.org/wiki/Monte_Carlo_tree_search.

[3] Levente Kocsis and Csaba Szepesvári, *Bandit Based Monte-Carlo Planning*, In *Machine Learning: ECML 2006*, Springer, Berlin, Heidelberg, 2006, pp. 282–293, https://doi.org/10.1007/11871842_29.

[4] zhaochenyang20, *IAI_2022/homework/connect-4 at main · zhaochenyang20/IAI_2022 · GitHub*, 2022, https://github.com/zhaochenyang20/IAI_2022/tree/main/homework/connect-4.