四子棋-实验报告

于越洋

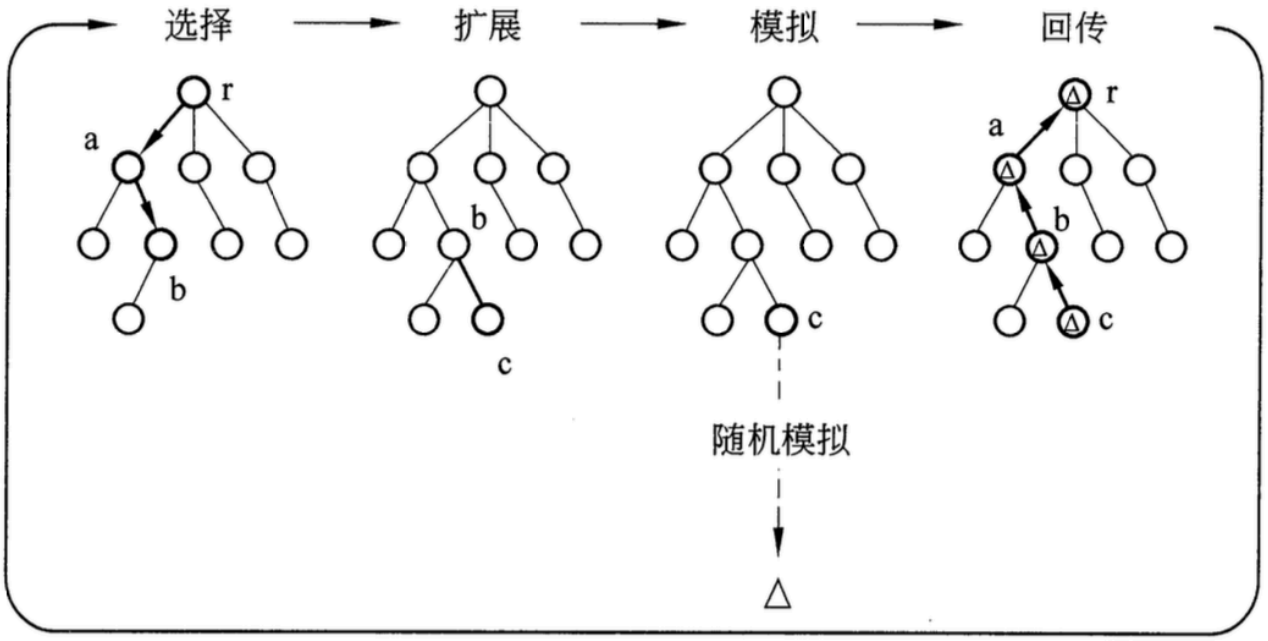
1. 策略

我的四子棋 AI 基于 **UCT 算法**，即结合了 MCTS 算法和 UCB 算法来实现博弈树搜索。

1.1 MCTS 算法

MCTS 算法主要由四部分构成：

- 选择：从根节点出发，按照某种原则自上而下地选择节点，直到第一次遇到一个**未完全扩展**的节点（即还有未生成的子节点）为止；
- 扩展：生成被选中节点的一个子节点，并添加到搜索树中；
- 模拟：对新生成的子节点进行蒙特卡洛随机模拟，即双方轮流落子，直至分出胜负或到达平局，然后根据结果计算收益；
- 回传：将收益向祖先传递，依次向上更新祖先节点的估计值。
  当以上过程重复进行多次后，可以获得根节点的所有子节点的大致收益，此时可以进行**决策**，即选取收益最大的子节点作为本次落子位置。
  下图为 MCTS 算法过程示例，图片取自教材 119 页。



1.2 UCB 算法

上限置信区间算法是解决多臂老虎机问题的经典算法，它既考虑了每种选择的收益，也考虑了每种选择收益的“可靠性”，即充分利用现有的信息，在探索和利用之间寻找平衡，做出最优的选

择。节点信心值由如下公式来计算：

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

其中 $\bar{X}_j$ 表示第 j 个选择到目前为止的平均收益，n 表示所有选择被尝试的总次数，$T_j(n)$ 表示第 j 个选择被尝试的次数，c 为一个可调节的系数，公式计算得到的结果 I_j 即为进行选择时使用的信心值。
式中的 $\bar{X}_j$ 项为收益项，是对现有搜索结果信息的利用；而 $\sqrt{\frac{2\ln n}{T_j(n)}}$ 为探索项，用于评估节点的收益值是否可靠（尝试次数越多，随机因素影响越小，理论上收益越可靠）。在该公式的加持下，算法每次做出选择时，会更倾向于选择收益值较高而模拟次数相对较少的子节点，从而验证该子节点的高收益是否是“真实”的。

1.3 UCT 算法

我的算法结合了上面提到的 MCTS 算法和 UCB 算法，使用**基于信心上限置信区间的蒙特卡洛树搜索算法（UCT 算法）**来解决四子棋问题。
我使用的计算节点信心值的公式为

$$I_j = \frac{Q(v_j)}{N(v_j)} + c\sqrt{\frac{2\ln N(v)}{N(v_j)}}$$

其中 v 表示父节点，也就是即将做出选择的节点，v_j 表示该节点的第 j 个子节点，$Q(v_j)$ 表示该子节点的**净胜利次数**，即获胜次数与获负次数之差，$N(v)$ 和 $N(v_j)$ 分别表示父节点和第 j 个子节点的**模拟次数**，c 是可调节系数，我在代码中选取为 1.2，公式计算得到结果 I_j 作为第 j 个子节点的信心值，在选择时选取信心值最大的子节点。

2. 实现

UCT 算法的实现参考论文A Survey of Monte Carlo Tree Search Methods(https://ieeexplore.ieee.org/document/6145622)，主要逻辑在 `Tree.cpp` 中。

2.1 UCB 算法 - BestChild

`SearchTree::BestChild` 函数用于计算当前节点所有子节点的信心值，并返回信心值最大的子节点作为选择结果，其具体逻辑如下：

function BestChild(v, c):
  bestScore $\leftarrow -1.0$
  bestChild $\leftarrow$ null
  for v' in children of v:
    if $\dfrac{Q(v')}{N(v')} + c\sqrt{\dfrac{2\ln N(v)}{N(v')}} >$ bestScore:
      bestScore $\leftarrow \dfrac{Q(v')}{N(v')} + c\sqrt{\dfrac{2\ln N(v)}{N(v')}}$
      bestChild $\leftarrow v'$
  return beatChild
end function

2.2 选择 - Select

`SearchTree::Select` 函数用于选择用于本次模拟的最优节点，对应 MCTS 算法中的“选择”过程，其具体逻辑如下：

function Select(v):
  while v is not terminal:
    if v have expandable children node:
      return v.expand()
    else
      $v \leftarrow$ BestChild(v)
  return v
end function

2.3 扩展 - expand

`UCTNode::expand` 函数用于对选择的节点进行扩展，对应 MCTS 算法中的“扩展”过程，其具体逻辑如下：

function expand:
  choose the most necessary move m of current node v
  create new state s based on the state of current node
  s.move(m)
  create new child node v' with the new state s
  return v'
end function

2.4 模拟 - Simulate

`SearchTree::Simulate` 函数用于对选择的节点进行随机模拟，对应 MCTS 算法中的“模拟”过程，其具体逻辑如下：

function Simulate(v):
  $s \leftarrow$ state of v
  while s is not a terminal state:
    select a legal move a of s at random
    $s \leftarrow s.\text{move}(a)$
  return the result of state s
end function

2.5 回传 - Backup

`SearchTree::Backup` 函数用于反向传播模拟的结果，对应 MCTS 算法中的“回传”过程，其具体逻辑如下：

function Backup(v, r):
  while v is not null:
    $v.\text{update}(r)$
    $v \leftarrow$ the parent of v
end function

对于节点信息的更新，调用 `UCTNode::update` 函数，由于得到的模拟结果均是相对于己方的，所以该函数判断节点是否是己方节点，以决定**净胜利次数**应当增加还是减少，具体逻辑如下：

function update(r):
  $N(v) \leftarrow N(v) + 1$
  if current node is own node:
    $Q(v) \leftarrow Q(v) + r$
  else
    $Q(v) \leftarrow Q(v) - r$
end function

2.6 搜索 - Search

`SearchTree::Search` 函数是算法的核心函数，用于实现搜索过程，其具体逻辑如下：

function Search:
  while within the time limit:
    $v \leftarrow$ Select($root$)
    $r \leftarrow$ Simulate(v)
    Backup(v, r)
  return the move of BestChild($root$, 0.0)
end function

3. 优化

在普通的 UCT 算法的基础上，我还使用一些特殊策略进行了优化，以进一步提高胜率。

3.1 攻防策略

对于四子棋来说，每当一方有三个棋子连在一起时，仅需再落子一次即可立即取胜，对于这种情况下的落子，完全没必要进行一次蒙特卡洛树搜索。因此在进行搜索前，先判断当前棋局是否有可以让己方一步取胜的落子点，再判断是否有对方可以一步取胜的落子点，只要找到就立即返回对应的点作为落子位置，以实现己方取胜（攻）或解除对方取胜威胁（防）的目的。若并未寻找到这样的落子点，则正常进行搜索，确定落子位置。

3.2 中部优势

考虑到在棋盘中部落子，后续可拓展的方向多于在棋盘两侧落子，便于施展策略，从而更容易取得胜利。因此，在选择落子位置时，使程序有更大概率选择靠近棋盘中部的落子点，应该对胜率提升有一定帮助。在实现上，我采用权重机制，即为棋盘的每一列赋予不同的权重，权重选取使用如下公式：

$$weight_i = \begin{cases} (i+1)^2, i \leq \dfrac{n}{2} \\ weight_{n-i-1}, i > \dfrac{n}{2} \end{cases}$$

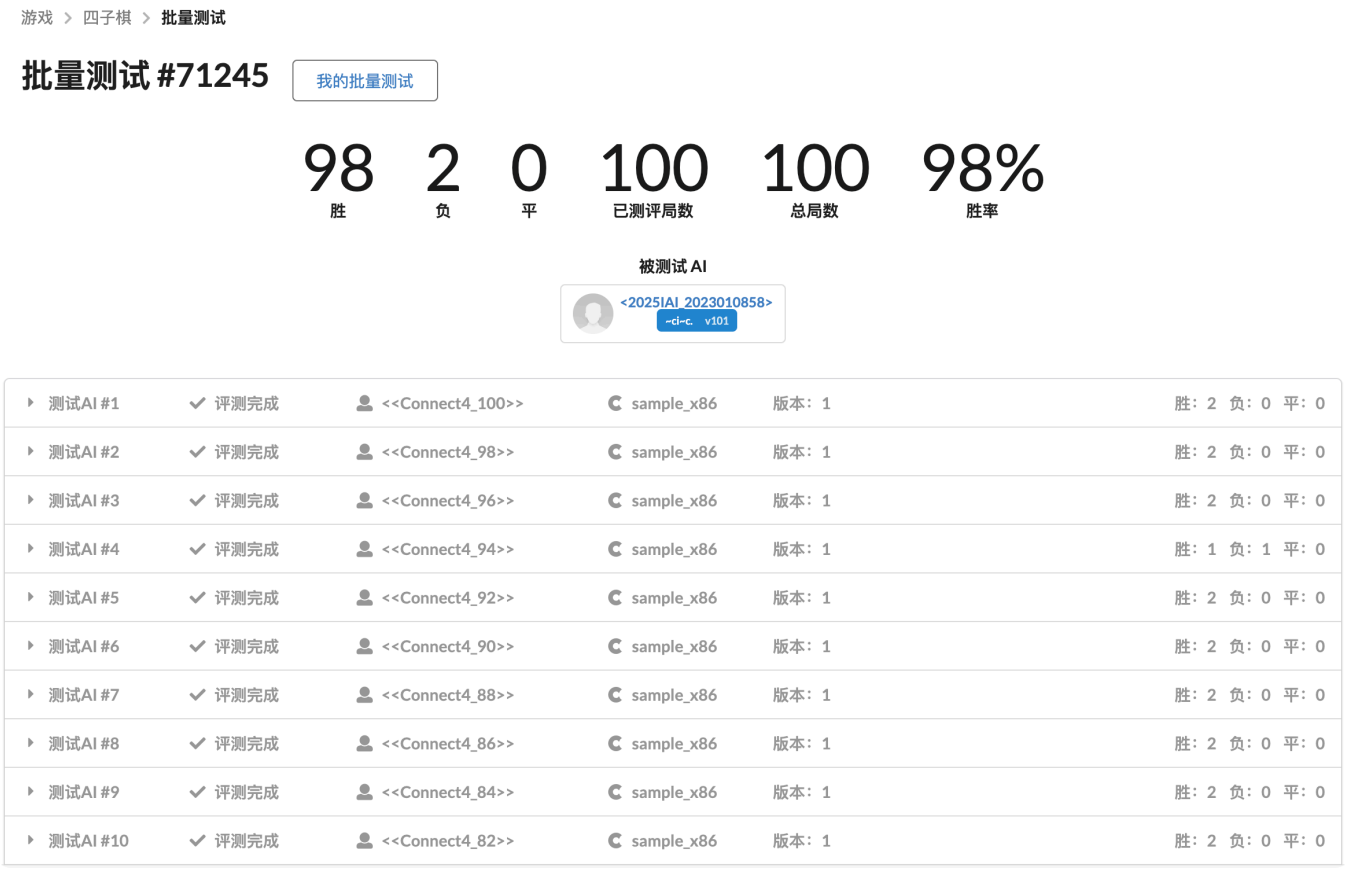
这样在进行随机模拟时，更容易选取棋盘中部的落子点进行落子。

3.3 数据结构

考虑到棋盘上的每个位置只有三种状态：空位、我方落子、对方落子，而重力四子棋所有落子一定在该列的 `top` 下方，因此，以 `top` 数组中的高度为分界线，可将每一列分成两部分，即 `top` 上方（全部为空位）和 `top` 下方（全部为落子点）。这样，对于 `top` 上方的点，仅有一种状态，即空位，可用 0 表示；对于 `top` 下方的点，有我方落子和对方落子两种状态，分别用 0 和 1 表示，如此就可以仅用 1 个 bit 来表示棋盘上每个位置的状态。
我的代码逻辑中只需要获取 `top` 以下的位置状态，即获取落子点属于哪一方从而判断是否有胜局。则在获取状态时，若该位置为 0，即可返回 2，表示有己方落子；若该位置为 1，即可返回 1，表示有对方落子。对于题目限制的不可落子点，在获取状态时做特殊判断，直接返回 0 即可。
此外，为适应新的棋盘存储方式，我在 `State` 类中重写了胜局情况的检查函数，在进横向和对角方向检查时添加了对当前位置是否高于 `top` 的判断。
题目中所给的棋盘最大不会超过 12×12 ，也就意味着一个棋盘最多只需要 144 个 bit 即可完全表示，这样就可以将原来表示棋盘所用的的大小为 mn 的二维整型数组压缩为三个无符号长整数，大幅降低了状态表示的空间消耗，也可以减少状态拷贝时分配空间和进行深拷贝的时间成本，从而更有效地利用搜索时间，增加模拟次数。

4. 统计数据

与系统提供的 50 个 AI 进行对抗，得到的结果为



该批量测试对应的网址为https://www.saiblo.net/batch/71245/
此为最好结果，多次批测的胜率在区间 93%～98% 内波动。

5. 总结

通过本次实验，我对 UCT 算法有了更为深刻的理解，也知道了如何将蒙特卡洛树搜索应用于实际问题之中。在优化算法的过程中，我切实感受到了良好的策略和精心设计的数据结构能够更好地发挥算法的能力，从而取得更高的胜率。