四子棋实验报告

于新雨 计25 2022010841

统计数据

进行5次测量，得到的结果是胜率分别为97%，94%，93%，96%,98% 且都没有平局出现，综合胜率为 95~96%

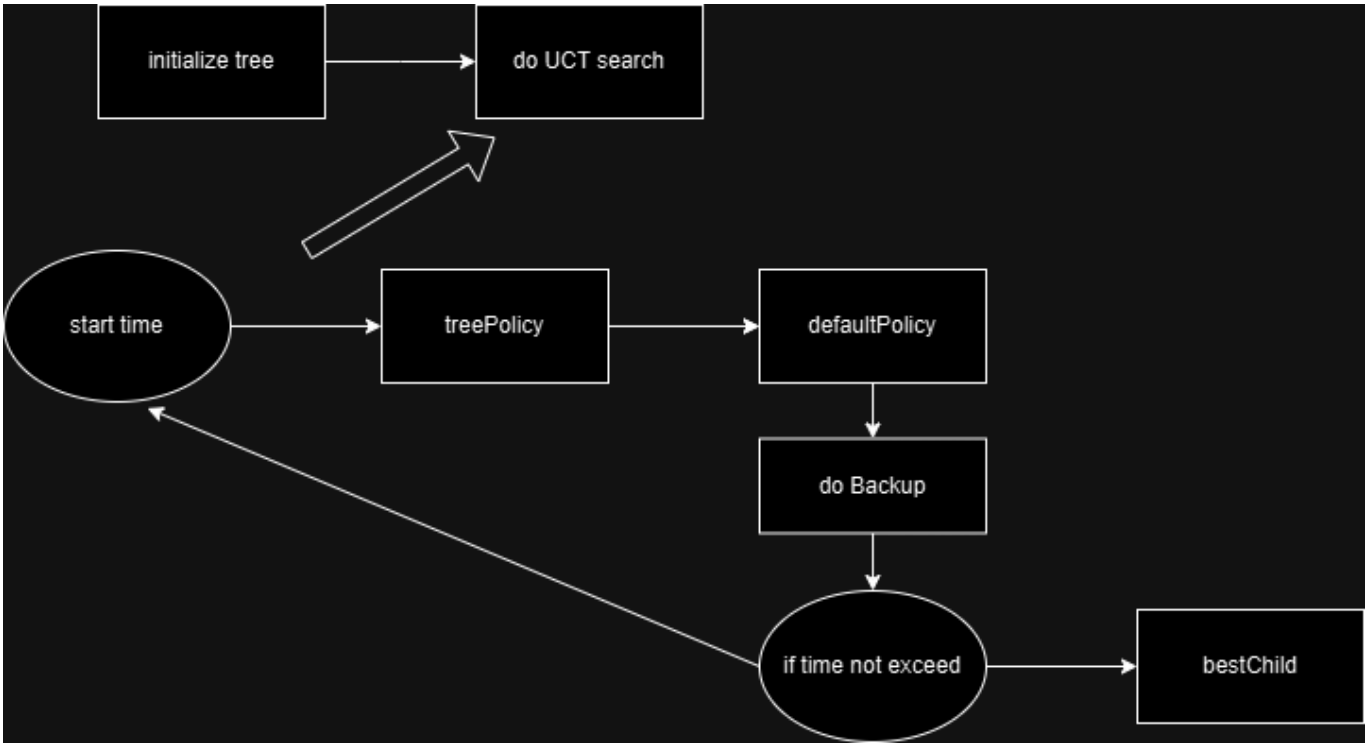
策略说明

总体策略如下：

1. 采用和 100.so 中的实现类似的蒙特卡洛树搜索的方法，来实现基本算法
2. 采用对各列权重的优化来提高准确性
3. 采用下必胜点/堵必输点的方法提高策略稳定性

蒙特卡洛树搜索

总体流程如下



数据结构

`Node`

- **描述**: 表示蒙特卡洛树搜索中的一个节点，用于保存游戏的状态信息以及模拟决策过程。
- **属性**:
  - `num_visits`: 该节点的访问次数。
  - `parent`: 指向父节点的指针。
  - `children`: 子节点数组。

    - `expandables`: 可扩展的动作列表，即还未探索的子节点。
    - `expandables_cnt`: 可扩展的动作数量。
    - `my_x`, `my_y`: 在棋盘上的行和列。
    - `score`: 节点的评分，用于反向传播更新。
    - `board`: 表示游戏当前状态的二维数组。
    - `top`: 表示每列的顶部空闲位置。
    - `M`, `N`: 棋盘的行数和列数。
    - `is_over`: 表示游戏是否结束。
    - `is_tie`: 表示游戏是否平局。
    - `player`: 当前轮到哪位玩家。
    - `weight_x`: 用于权重的数组，辅助上述对各列权重的优化。
- **方法**:
    - `my_rand()`: 根据权重随机选择一个可扩展的列在`expandables`内的编号。
    - `~Node()`：析构函数

`MCTSTree`

- **描述**: 蒙特卡洛树搜索的主体，用于控制搜索过程，并维护树的根节点和结构。
- **属性**:
    - `root`: 树的根节点。
    - `M`, `N`: 棋盘的行数和列数。
    - `top`: 当前每列的顶部空闲位置。
    - `board`: 当前棋盘的状态。
    - `num_nodes`: 树中的节点总数。
    - `num_leafs`: 叶子节点的数量。
    - `weights`: 各个动作的权重。
- **方法**:
    - `UCTSearch()`: 实施 UCT 搜索策略。
    - `expand()`: 在指定节点上执行扩展操作。
    - `backup()`: 根据模拟结果更新路径上的所有节点。
    - `treePolicy()`: 决定在当前节点进行扩展还是继续向下搜索。
    - `defaultPolicy()`: 进行快速模拟，评估局面。
    - `bestChild()`: 选择评分最高的子节点。
    - `bestRootChild()`: 选择根节点下评分最高的子节点。
    - `~MCTSTree()`: 析构函数，负责清理内存。

搜索过程

`UCTSearch` 是 MCTS 算法的核心函数，负责控制搜索过程。函数逻辑包括：

1. **时间检查**：确定是否超过预定时间，若是则停止搜索。该时间设定为2.0s，因为平台测试超过2.2s 时，最高几个点都可能 tle
2. **树策略（Selection）**：通过 `treePolicy` 递归选择最优节点或创建新节点。在存在子节点未经过探索时，随机选择一列进行探索，创建一个新子节点并且计算是否为必胜/必输点，等所有节点都经过探索后，再计算最优子节点通过以下公式
   $$ [\text{UCT} = \frac{w_i}{n_i} + c \sqrt{\frac{\ln N_i}{n_i}}]
   $$

公式参数解释：

$$ (w_i): 节点 (i) 的累计奖励。
(n_i): 节点 (i) 的访问次数。
(N_i): 父节点的访问次数。
(c): 探索常数，观察到 100.so 中设置的参数是1，所以我们也用参数1，用于平衡探索与利用。
$$

3. **默认策略（Simulation）**：通过 `defaultPolicy` 进行游戏模拟直至游戏结束。
4. **反向传播（Backpropagation）**：通过 `backup` 更新节点统计，影响后续选择。

每次迭代通过这四个步骤，最终通过 `bestRootChild` 选择根节点的最佳子节点作为决策输出。其中 `bestRootChild` 是通过score/visit 的次数进行计算

效果

实现monte-carlo tree search 之后，正确率可以达到90%左右

权重优化

和计27 段睿思同学讨论，得到的优化方法是在扩展/模拟列的时候，使得随机数没那么随机，即是更偏向于取靠近中间的数。从人下棋的角度来说，取靠近中间点的能动性可以更高，而在实验过程中，模拟也更加充分。经过多次实验，发现较好的分布是使得最边缘列的选取的概率为中间列选取概率的一半，就像给出的如下权重

```
    int weight_9[9]={4,5,6,7,8,7,6,5,4};
    int weight_10[10]={4,5,6,7,8,8,7,6,5,4};
    int weight_11[11]={5,6,7,8,9,10,9,8,7,6,5};
    int weight_12[12]={5,6,7,8,9,10,10,9,8,7,6,5};
```

效果

实现之后，平均胜率达到93%左右，但是最低也出现过86%胜率的情况

预判断必胜必输点

为了增加算法稳定性，先采取类似于人下棋的时候“堵棋”的策略，就是在每一步前预先遍历所有列判断，如果有直接能赢的点，或者对方如果下就对方能赢的点，就直接下在该点上。可以确定，这样子下不会使得性能更劣，也能在多数情况下规避掉快终局情况下启发式算法带来的随机性问题。而这一步虽然无法防止对方采取“草蛇灰线”的方法来赢，但是确实可以一定程度上削弱随机性，增强算法稳定性

效果

实现之后，达到了平均95%胜率，而且胜率波动在5%左右，有显著改善

方法创新性

策略上创新

和样例中 100.so 中实现的蒙特卡洛搜索相比，我增加了权重优化和预判断必胜必输点的优化，提高了性能

实现方法上创新

- “知己知彼，百战不殆”，为了了解对局 AI 的能力范围和优化水平，我应用**逆向工程**的方法，对部分 AI 进行了破解，其中对 100.so 进行了较深入的逆向，而在对局之前了解对手水平，也贴近于人对战时的表现
- 总体发现：
  - 发现 100.so 只是实现了一个较为朴素的蒙特卡洛搜索，但是相较于我的实现，它的一个优势是减少了动态分配内存的开销，前面也有几个AI 比如 72.so 是通过线程池 创建 Node 的形式减少开销
  - 即是是非常前面的AI 比如30左右的 也都是“五脏俱全” 有基本搜索过程。在60~70+ 左右的一些 AI 性能低的原因是有硬伤，就像设置时限太长导致大概率超时这种的。而后面的 AI 至少逆向出来看上去挺对的，各个流程都也具备，大概率是实现上的细节上的差异
- 过程贴图

```
23    v6 = (double)(int)(std::chrono::_V2::system_clock::now(this) - v5) / 1000000000.0;
24    if ( v6 > *((double *)v1 + 11) )
25      break;
26    MCTSTree::copyStatusStatic(v1, v3, v2, *((const int **)v1 + 5), *((int ***)v1 + 6));
27    v7 = MCTSTree::treePolicy(v1);
28    if ( !v7 )
29    {
30      __printf_chk(1LL, "<<<<<<<<<<<<vl == nullptr>>>>>>>>>>>>\n");
31      break;
32    }
33    if ( *(_BYTE *)(v7 + 41) )
34    {
35      v6 = 0.0;
36      if ( *(_BYTE *)(v7 + 40) )
37        v6 = 1.0;
38    }
39    else
40    {
41      v11 = v7;
42      MCTSTree::defaultPolicy(v1);
43      v7 = v11;
44    }
45    this = v1;
46    MCTSTree::backUp(v1, v7, v6);
47    ++*(_DWORD *)v1;
48  }
49  if ( *((_BYTE *)v1 + 96) )
50    return MCTSTree::bestChild(v1, *((_QWORD *)v1 + 1));
```

原 UCTSearch 反编译出来的结果

```
int* MCTSTree::UCTSearch() {
  auto start = std::chrono::high_resolution_clock::now();
  auto end = start;
  int count = 0;
  while (true) {
    end = std::chrono::high_resolution_clock::now();
    std::chrono::duration<double> elapsed_seconds = end - start;
    if (elapsed_seconds.count() > m_maxTime) {
      break;
    }
    copyStatusStatic(m_currentStatus, m_root);
    auto vl = treePolicy();
    if (vl == nullptr) {
      printf("<<<<<<<<<<<<vl == nullptr>>>>>>>>>>>>\n");
      break;
    }
    if (vl->isEnd) {
      double result = 0;
```

```
        if (vl->isWin) {
          result = 1;
        }
        backUp(vl, result);
      } else {
        defaultPolicy();
        backUp(vl, 0);
      }
      count++;
    }
    if (m_verbose) {
      return bestChild(m_root);
    } else {
      int max = 0;
      auto best = m_root->child;
      auto current = m_root->child;
      while (current != nullptr) {
        if (current->visit > max) {
          max = current->visit;
          best = current;
        }
        current = current->nextSibling;
      }
      return best->move;
    }
  }
```

用 MLM 进行语义推断
可以看到对于单个函数而言，逆向得到的结果已经非常符合我们的实现了，但是因为它语义推断无法进行过程间分析，无法完全推断结构体，其中关于 Node 的结构体信息也较难推断，所以主体部分还是要自己写一遍

- 此外，这种方法也有助于理解该算法，比如说 Node 是如何保存状态的，还有一些细节和参数之类的，也可以逆向得到