2022 年春季学期
数据结构课程设计第 5-7 次课赛道_B_
实验报告

罗旺[1] 唐文鑫[1] 戴轲翰[2] 余越[3]

[1] 软件学院 2020 级 9 班

[2] 软件学院 2020 级 10 班

[3] 软件学院 2020 级 8 班

1. 分工与合作

罗旺：搭建 MCTS 框架，算法优化

唐文鑫：设计价值函数，测试参数

戴轲翰：算法优化

余越：测试参数

2. 参考文献与创新之处

2.1 参考文献:

[1] A Survey of Monte Carlo Tree Search Methods Cameron Browne, Member, IEEE, Edward Powley, Member, IEEE, Daniel Whitehouse, Member, IEEE, Simon Lucas, Senior Member, IEEE, Peter I. Cowling, Member, IEEE, Philipp Rohlfshagen, Stephen Tavener, Diego Perez, Spyridon Samothrakis and Simon Colto

链接：https://ieeexplore.ieee.org/abstract/document/6145622

期刊：IEEE

页码：9（UCT 伪代码）

[2] 郭倩宇,陈优广.基于价值评估的不围棋递归算法[J].华东师范大学学报(自然科学版),2019(01):58-65.

链接：

https://webvpn.jlu.edu.cn/https/77726476706e69737468656265737421fbf952d2243e635930068cb8/kcms/detail/detail.aspx?dbcode=CJFD&dbname=CJFDLAST2019&filename=HDSZ201901007&uniplatform=NZKPT&v=V6m33Xv4mKDzyS2nRb3sz2rjzV_7jkIKxtVzTMTxaixRGMFKkSUN-a52X9Yu6-6q

期刊：华东师范大学学报（自然科学版）

页码：4（权利值的讲解）

2.2 创新之处:

为了提高 MCTS 的模拟效率以提高模拟次数，让 MCTS 得到的解更逼近最优解，我们在原论文的基础上采取了一系列的优化措施，包括调整 UCB 公式中的常数 C，修改蒙特卡洛树结点的数据结构来用空间换时间，优化原文中的扩展策略，修改模拟策略、设计价值评估函数来准确地评估棋面以提高模拟效率等方法（具体会在算法思想中提到），改善了原文中的 MCTS 算法，从最开始 0.8ms 模拟 2000 次左右到最后 0.8ms 能够模拟 2w 次（最后十回合 10w 到 100w 次），在 botzone 中天梯分在 1130 左右

3. 算法思想

3.1 算法描述

在 MCTS 算法的基础上，对算法的时间复杂度进行优化（尽量用空间换时间），加入价值评估策略

算法大体过程为：

1.由当前局面建立根节点，一次生成根节点的下一层全部子节点，分别进行模拟对局；

2.从根节点开始，进行最佳优先搜索；

3.利用 UCB 公式计算每个子节点的 UCB 值，选择最大值的子节点；

4.若此节点不是叶节点，则以此节点作为根节点，重复 2；

5.直到遇到叶节点，如果叶节点未曾经被模拟对局过，对这个叶节点模拟对局；否则为这个叶节点随机生成子节点，并进行模拟对局；

6.将模拟对局的收益(随机落子一定步数后，按一定的评估策略对状态进行评估，返回评出的价值)，按对应颜色更新该节点及各级祖先节点的总价值，同时增加该节点以上所有节点的访问次数；

7.回到 2 除非此轮搜索时间结束或者达到预设循环次数

8.从当前局面的子节点中挑选胜率（价值/访问次数）最高的给出最佳着法。

3.2 算法实现

1.蒙特卡洛搜索树结点的数据结构

```
typedef struct Node  // 蒙特卡洛搜索树结点
{
    /* 棋盘属性 */
    int board[board_size][board_size];  // 棋盘
    int col;  // 该棋盘下一步的颜色(1为白，-1为黑)
    Action action;  // 该棋盘上一步的落子位置
    double value;  // 该状态的价值
    int is_terminal;  // 该状态是否为终止状态
    vector<int> available_list;  //  可扩展的子节点表

    /* 结点属性 */
    int visited_times;  // 该结点的搜索次数
    Node *parent;  // 父节点;

    Node *childs[board_size][board_size];  // 子节点指针数组,新建的孩子结点需要存在对应下标的位置
} Node;
```

每一个蒙特卡洛搜索树结点都保存着一种棋盘状态，包括父结点指针、当前棋盘的布局、下一子由哪方落下、该棋盘上一步的落子位置、赋予该棋盘状态的价值以及访问次数、标记该棋盘是否为终止状态、下一步的可落子点（可扩展到 MCT 中的子结点表），最后这两个属性是我们对于基础 MCTS 的用空间换时间的优化策略，即将重复用到的、耗时的数据给封装到结点中。

3.3 算法具体优化思路以及实现：

1. 模拟过程的策略优化（Simulation）

最开始我们的模拟策略就是随机落子到游戏结束，输赢给予模拟开始的状态对应的结点以相应的价值，然后进行反向传播。结果发现这种策略是非常耗时的，其一就是每模拟一步，就要获取到当前棋盘中的可落子位置，而获取这个可落子位置的函数是一个 n 方复杂度的函数，也就是说总体上这是个复杂度为 n 的三次方的一个过程。其二就是这种耗时的模拟在没有经过大量次数的模拟的情况下它是没有意义的，因为是随机落子，双方都是“傻瓜”，这样模拟完一把的价值不大，模拟的效率不够，况且还耗时。

针对这种情况，我们最开始想到的是直接用价值函数替代模拟过程，直接用价值函数评估出当前棋盘的价值，这种方式对于价值的函数的时间复杂度要求较高，但是好处是省去了大量的模拟时间，让模拟次数大大增加。但是缺点也很大，其一就是对价值函数的正确性的要求比较大，价值函数是否能够正确评估当前的棋面直接决定了模拟的正确性与效率，而好一点的价值评估函数通常又是费时的，价值评估函数在设计的社会需要衡量时间消耗与正确性这两方面，时间消耗太大，减少了模拟次数，或者正确性不好，模拟的效率不高，都会导致模拟的效率下降，得到的解不够好；另外一点就是，在游戏初期，棋面上没有什么子的时候，这个时候价值评估函数就失效了，导致游戏前期评估出来的价值不够准确，得到的解不够好。

基于以上两点问题的思考，我们最终采取了折中的办法，我们先随机落子几步后再对棋盘进行价值评估，一方面解决了游戏前期的棋面上落子少，价值函数不能很好地评估棋面的问；另一方面又减少了随机落子，让模拟次数增加，然后模拟次数的增加，又能减少价值评估函数的不准确性带来的影响(毕竟 MCTS 还是基于大量模拟来得到较优解的，只要模拟次数够，什么问题都能解决)。

然后最后就是模拟层数与价值函数的设计。

首先是模拟层数的设计：

模拟层数多了，会导致模拟时间消耗过大，降低模拟次数；模拟层数少了价值函数评估的正确性会降低。

最开始我们设想的是动态的调整模拟层数，经过我们大量的观看对局，发现一般游戏对局在 69-71 回合结束的比较多，大部分棋局能在 73 回合结束，而我们固定模拟层数经过大量对比测试后（在 botzone 上一把一把的下，每一组测试数据都至少测试了 50 把），发现固定模拟层数为 6 是最佳。又考虑到，后期棋盘上棋子够，不用过多随机落子，仅凭价值函数就能较好地评估棋面。而前期棋盘上没什么子才需要多随机落子来评估。因此我们最开始设计模拟层数公式为

模拟层数 = 7 -（回合数 % 10）

式 1

但是经过测试，效果还是不如固定模拟层数为 6 好，于是更换策略为：

前 30 回合模拟层数为 6，30 回合以后模拟层数为 3

效果依旧不好，最终就确定固定模拟层数为 6 然后进行价值评估

2. 价值评估函数的优化设计

郭倩宇学者曾在 2019 年提出不围棋双方的权力值概念[2]，权力值即为每次对局对手的非法步数。本价值函数的设计正是借鉴了该概念，并且对该概念做出了创新。由于权力值是我方获得的优势的量化，所以在对局中我们要保证自己的优势比对方的优势高，所以就要保证自己的权利值高于对方的权利值。权力值的计算公式如式 2

$$p_w = \sum_{i=1}^{81} N_b^i, p_b = \sum_{i=1}^{81} N_w^i$$

式 2

$$f = \frac{p_1}{p_2}$$

式 3

式 3 中代表的价值函数为己方权力值与对方权力值的比值。

式 3 为本组最后确定的价值函数，但在此之前仍有诸多相关思考，故在下面表达出来。

由于式 3 的取值的范围是 0 到正无穷的，所以我们要对其进行压缩，并且考虑到如果下一步棋，使得自己的权力值比对方的权力值小了，显然我们不能允许这样的行为发生，因此要给它加入惩罚机制，即当下一步棋导致比值小于 1，价值函数的值就要小于零，代表惩罚。对于式 3，价值函数此时的值域为 0 到正无穷，我们可以使用 log 函数对其映射到负无穷到正无穷的区间，再通过 tanh 函数将其映射到-1 到 1 的区间，而由于 log 函数在 x 等于 1 的点区分出大于零和小于零，而 tanh 函数在 x=0 时作为正负的分界点，因此使用 log 和 tanh 的双重映射可以使原来以 1 为惩罚奖励分界点转化为以 0 作为分界点。

在此之前，曾考虑过使用 sigmoid 函数作为第二层映射函数，但由于 sigmoid 函数没有惩罚机制，所以后来替换为 tanh 函数。

$$\tanh = 2sigmoid(2x) - 1 = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}}$$

式 4

故此时的价值函数为：

$$f = \tanh(\log(\frac{p_1}{p_2})) = \frac{e^{\log(\frac{p_1}{p_2})} - e^{\log(\frac{p_2}{p_1})}}{e^{\log(\frac{p_1}{p_2})} + e^{\log(\frac{p_2}{p_1})}}$$

式 5

但由实验测试得知，式 5 的能力明显弱于式 3，在此我总结了一些原因：

1. 在计算 tanh 和 log 的时候由于耗时比较长，所以会降低模拟次数从而降低下棋的能力。

2. tanh 和 log 函数的导数分布并不均匀，可能会导致比值增长相同，但因为在不同区间函数值的增长过高或过低。

综合以上缺陷考虑，最终使用式 3 来作为价值评估函数。

3. 扩展策略的优化（expansion）

最开始按论文中描述的，每选取一个叶节点，就将其一个子结点扩展到蒙特卡洛树中，但我们后面发现，其实每次求解过程中，根节点的下一层结点一定是全在树中的，下两层结点就不一定，所以我们决定一开始就将根节点的下一层结点全部加入到树中，然后进行模拟与反向传播。然后其余的结点就按原策略加入到树中。最终经实验证明，这种策略能够提高百分之 10 的模拟次数，在游戏后期甚至提高了一倍的模拟次数（因为游戏后期模拟一次的时间很短，添加完一层后就没多少结点能添加了，时间大部分用在模拟上了）

4. UCB 公式中的常数 C 的优化

因为 C 值决定的是探索比，C 值越大，模拟就越倾向于未模拟过或者说模拟次数较少的结点；C 值越小，模拟就越倾向于价值大/胜率高的结点。所以对于 MCTS 中要的要一个好的较优解，C 值是很重要的，而且不同游戏的最优 C 值应该是不同的，于是我们经过大量测试（将 C 值从 1.0 到 2.0，步长 0.1 分为 12 个小数，经过大量测试比较，最后得到最优的 C 值为 1.5，而不是论文中的根号 2（1.4）

5.数据结构的空间换时间策略

1. 在选取下一个要扩展的结点的过程中，要重复判断一个结点是否为最终状态(is_terminal)，一个结点是否是最终状态是不变的，于是将 is_terminal 封装到结点的数据结构中，初值为-1 表示未计算过是否是最终状态，0 表示不是最终状态，1 表示是最终状态。计算过一个点是否为最终状态后就将其值保存到 is_terminal 属性中，后面再访问直接访问该属性即可，于是函数设计就是：

```
// 判断是否为终止状态
inline int IsTerminal(Node * node)   // 无需优化
{
    if(node->is_terminal == -1)
    {
        for (int i=0;i<9;i++)  // 检查棋盘是否能够继续落子，若能则不是终止状态
        {
            for (int j = 0; j < 9; j++)
            {
                if (judgeAvailable(node->board, i, j, node->col))
                {
                    node->is_terminal = 0;
                    return 0;
                }
            }
        }
        node->is_terminal = 1;
        return 1;
    }
    else
    {
        return node->is_terminal;
    }
}
```

2. 在选取结点进行扩展的时候，随机选取能够进行扩展的一个子结点加入树总，即每次都要计算这个点的哪些子节点能够加入到树中，然后随机选取一个。因为每次只加入一个子节点进入树中，其余的子节点后续可能也会加入树中，也会计算这个点的哪些子节点能够加入到树中。

基于这个问题，我们将

```
vector<int> available_list;  //  可扩展的子节点表
```

封装到结点中，每次扩展时即为其计算可扩展的子结点表，需要扩展的时候就从表中随机选一个扩展，然后每扩展了一个子节点，就从对应的表中删去。

然后这里还有一个优化点，就是什么时候为结点计算可扩展的子节点表，我们发现，我们在扩展一个结点后，需要对结点进行模拟，而模拟的第一步随机落子，就会计算这个结点的下一个可落子点，即可扩展的子结点表，于是直接将可扩展的子节点表的初始化放入模拟中

```
inline double DefaultPolicy(Node *node, int depth)
{
    int temp_board[board_size][board_size];  // 复制node的棋盘用于模拟
    memcpy( Dst: temp_board,  Src: node->board,  Size: sizeof(temp_board));
    int color = node->col;
    bool flag = true;  // 标记是否使用评估策略进行评估
    for(int i = 0; i < depth; i++)  // 快速随机落子
    {
        vector<int> available_list_;  //临时合法位置表
        for (int i=0;i<9;i++)  //随机选取可落子处落子
            for (int j=0;j<9;j++)
                if (judgeAvailable( board: temp_board, fx: i, fy: j,  col: color))
                    available_list_.push_back(i * 9 + j);
        if(i == 0)  //第一次模拟时为node的available_list赋值
        {
            node->available_list.assign( first: available_list_.begin(),  last: available_list_.end());
        }
        if(available_list_.empty())  // 为最终状态时，直接进行查看颜色进行评估，而不用调用评估策略函数
        {
            flag = false;
            break;
        }
        int result = available_list_[rand() % available_list_.size()];  //随机选取
        temp_board[result / 9][result % 9] = color;  //落子
        color = -color;  //交换下棋者颜色
    }
```

6.其他优化

所有结点加入树中后，不断对根节点的下一层子结点进行模拟

以上就是我们组的不围棋 MCTS 算法的全部优化过程以及实现