四子棋实验报告

计算机系 2019011312 姚建竹

2021 年 4 月 30 日

1 算法思路

我选择使用蒙特卡洛方法和信心上限树算法。在构建 UCT 的过程中，随机模拟四子棋双方进行对战，并对每次模拟的终局情况进行收益的计算，最终表达出每个 UCT 树的节点，从而选择最佳落子点。算法的大致思路如下：

首先根据当前的棋盘状态建立根节点。我设定搜索次数上限为 680000 次。每一次搜索，选择扩展当前的节点或者是根据 best child 算法计算出的最优孩子，其中算法使用下式：

$$\frac{Q(v)}{N(v)} + c\sqrt{\frac{2ln(N(u))}{N(v)}} \tag{1}$$

其中，u 为当前的节点，v 为考虑的子节点，$N(v)$ 为 v 被回溯的次数，$Q(v)$ 为 v 回溯的胜利次数。

获得应当模拟的结点之后，我进行了一次游戏的模拟，即双方轮流下子，直到游戏结束（在其中可以选择使用一些策略）。根据此次模拟的胜利与否选择对该节点的得分进行一定的操作。

最后，在根节点中选择应用上式计算出的得分最高的孩子，采取该孩子的下法，即为我们做出的决策。

2 实现过程

2.1 自定义 UCT 类

定义了两个类：表示当前棋盘状态的 State 类和节点 Mcts Node 类。

State 类：

int board[MAX_SIZE]	一位数组，储存棋盘的状态
int lastX	这个状态上一次下点的横坐标
int lastY	这个状态上一次下点的纵坐标
int last_type	这个状态下一次落子的人
int win_type	这个状态的获胜方
set<int> expand	当前状态可以扩展的纵坐标集合
bool is_terminal()	判断是否已经是终局
void calculate_expand()	计算可以扩展的纵坐标集合
bool can_expand()	判断是否能够扩展

Mcts Node 类：

Mcts_Node* parent	父节点
Mcts_Node* first_child	大儿子
Mcts_Node* next	弟弟指针
Mcts_Node* next_free	在链表内存池中下一个可分配元素的地址
int visited	当前节点被遍历的次数
int win	当前节点胜利次数
State state	当前节点的状态
void set_node_state(State)	设置当前节点的状态
Mcts_Node* new_node(int)	以当前节点为父节点，新建一个孩子节点

对于 State 类，我维护了一个 int 类型数组 board（代表当前阶段的棋盘状态），几个 int 类型分别表示上一次下子的人、位置和胜利的一方。同时我维护了一个 set<int> 类型的成员，为了保存当前棋盘状态可以下的位置。我改写了 Judge.cpp 中提供的终局检测函数，使其仅使用一维数组，效率更高。对于 set<int> 类型的成员，我会扫描每列的最上方元素，当元素为空且不是不可落子点时，可以进行下子。

对于 Mcts Node 类，我维护了它的父亲指针、大儿子指针和弟弟指针。同时为了维护一个链表类型的内存池，我又包含了一个指向下一个可分配节点的指针。同时，维护了两个 int 类型变量储存节点的访问次数和胜利次数（得分）。并且，一个 Mcts Node 应该包含上述的一个 State 类型变量。而 Mcts Node 类也提供了设置 state，增加孩子的成员函数。

2.2 内存池的实现

我最初使用的方法是新建节点时使用 new 创建一个对象，但是我发现在频繁的 new 过程中会浪费比较多的时间，所以我打算使用统一分配内存的方式来创造节点。

我使用了简单的链表内存池的方式。具体方法为：每一个节点内都含有一个指向下一个可分配的节点，同时有一个头节点。当想要一个新的节点时，直接在链表的头部获得一个节点即可，再将头节点指向的下一个可分配节点向下传递一个，即可维护一个可分配节点的链表。当进行析构时，只需要在初始化节点成员的同时，插入到可分配节点的链表头部即可，复杂度均为 $O(1)$。

2.3 算法的实现

使用 MCTS 的算法，框架大致相同

next_state()	将当前状态转移到下一个状态
Mcts_Node* best_child()	计算一个节点的 best_child
Mcts_Node* expand()	扩展一个节点
Mcts_Node* tree_policy()	使用生成树策略
int default_policy()	进行模拟，返回胜利状态
void backup()	回溯
Point uct_search()	建树函数

为了让程序不至于超时，我设置了最大搜索次数为 680000 次，最大搜索时间为 2.7 秒。

2.4 采取的策略

我在落子的时候采取了如下策略:

1. 如果我方下在 (x, y) 处必胜，那么不进行搜索，直接返回 (x,y)
2. 如果对方下在 (x, y) 处必胜，那么不进行搜索，直接返回 (x,y)
3. 如果对方下在 (x-1,y) 处必胜，那么我方一定不会下在 (x,y) 处
4. 如果我方下在 (x-1, y) 处必胜，那么我方一定不会下在 (x,y) 处
5. 如果在同一个横线上存在两个对方的子，同时这两个子的左边和右边同时有小于 2 个自己的子，我方会下在一边防止产生三个连续的子。

针对上面的四个策略，我在不同的时机分别执行，取得了不一样的效果。

首先，我在选择最终落子位置的时候引入了 12345 五种方案。对于 34 方案，我使用了两个 set<int> 收集对应的位置，并在根节点的 state.expand 对象中进行去除，保证了搜索不到对应的节点，达到了删除的效果。

另外，在随机模拟的 default_policy 中我也选择了引入 12 两种策略，不过由于 12 两种策略会比较浪费时间，所以在模拟中引入策略后搜索的次数会有所下降，导致决策的效果较差，所以我针对这两种情况分别进行了实验分析，选择了具有最佳效果的一个方案。而当在随机模拟时同时引入 1234 四种策略时，模拟的次数会发生大幅下降，导致 AI 的表现较差。

3 实验结果

我使用自己的测试脚本，分别对 2-100 的 50 个 ai 进行了 10 轮对局（共 20 次对战），以获得较为准确的胜率结果统计。

3.1 模拟完全随机，落子选择 1234 策略

	2.dylib	4.dylib	6.dylib	8.dylib	10.dylib	12.dylib	14.dylib	16.dylib	18.dylib
轮数	10	10	10	10	10	10	10	10	10
胜率	1	1	1	1	1	1	1	1	1
	20.dylib	22.dylib	24.dylib	26.dylib	28.dylib	30.dylib	32.dylib	34.dylib	36.dylib
轮数	10	10	10	10	10	10	10	10	10
胜率	1	1	1	1	1	1	1	1	1
	38.dylib	40.dylib	42.dylib	44.dylib	46.dylib	48.dylib	50.dylib	52.dylib	54.dylib
轮数	10	10	10	10	10	10	10	10	10
胜率	0.95	0.95	0.5	0.9	0.6	0.35	1	1	1
	56.dylib	58.dylib	60.dylib	62.dylib	64.dylib	66.dylib	68.dylib	70.dylib	72.dylib
轮数	10	10	10	10	10	10	10	10	10
胜率	1	1	1	1	1	1	1	1	1
	74.dylib	76.dylib	78.dylib	80.dylib	82.dylib	84.dylib	86.dylib	88.dylib	90.dylib
轮数	10	10	10	10	10	10	10	10	10
胜率	1	1	1	1	1	1	0.95	0.7	0.95
	92.dylib	94.dylib	96.dylib	98.dylib	100.dylib				
轮数	10	10	10	10	10				
胜率	0.95	0.5	0.9	0.6	0.35				

3.2 模拟时加入启发 12 的策略

	2.dylib	4.dylib	6.dylib	8.dylib	10.dylib	12.dylib	14.dylib	16.dylib	18.dylib
轮数	10	10	10	10	10	10	10	10	10
胜率	1	1	1	1	1	1	1	0.9	1
	20.dylib	22.dylib	24.dylib	26.dylib	28.dylib	30.dylib	32.dylib	34.dylib	36.dylib
轮数	10	10	10	10	10	10	10	10	10
胜率	1	1	1	1	1	1	1	1	1
	38.dylib	40.dylib	42.dylib	44.dylib	46.dylib	48.dylib	50.dylib	52.dylib	54.dylib
轮数	10	10	10	10	10	10	10	10	10
胜率	1	1	1	1	1	1	0.85	0.9	1
	56.dylib	58.dylib	60.dylib	62.dylib	64.dylib	66.dylib	68.dylib	70.dylib	72.dylib
轮数	10	10	10	10	10	10	10	10	10
胜率	0.95	0.95	0.95	0.95	0.85	0.95	0.95	0.9	0.85
	74.dylib	76.dylib	78.dylib	80.dylib	82.dylib	84.dylib	86.dylib	88.dylib	90.dylib
轮数	10	10	10	10	10	10	10	10	10
胜率	0.85	0.8	0.9	0.95	0.85	1	0.75	0.85	0.85
	92.dylib	94.dylib	96.dylib	98.dylib	100.dylib				
轮数	10	10	10	10	10				
胜率	0.95	0.15	0.95	0.8	0.4				

经过统计发现，在模拟时加入了第 1 和第 2 个落子策略，会让 AI 的胜率比完全随机模拟时的胜率有所提高，所以效果更好一些。由于模拟的次数较多，每对 AI 都进行了 20 次对战，所以总体的胜率较高，可以认为较好地完成了四子棋对战的任务。

4 总结

通过本次实验，我对蒙特卡洛方法的 UCT 搜索有了更深的理解，也了解了内存池的简单使用方法。同时，我将其运用到四子棋的策略中，得到了较为理想的效果。