重力四字棋实验报告

计01　容逸朗　2020010869

算法介绍

算法框架

本次实验中，我使用了信心上限树算法（UCT，该算法结合了蒙特卡罗树与信心上限算法），具体的算法内容可以结合课堂参考资料给出的伪代码来解释。

下面的代码是 UCT 的核心框架部分。可以看见，对于当前的（棋盘）状态，我们希望找到最佳的子节点（状态），使得我方在对应位置下棋的收益最高。而具体的搜索过程则可以划分为 4 个部分，也就是选择（TreePolicy）、扩展（Expand）、模拟（DefaultPolicy）、回传（Backup）。

```
function UctSearch(s0)
    以状态 s0 创建根节点 v0
        while 尚未用完计算时长 do:
            vl ← TreePolicy(v0)
            △ ← DefaultPolicy(s(vl))
            Backup(vl, △)
        end while
    return a(BestChild(v0, 0))
```

对于选择（TreePolicy）部分，简单来说就是不断按照特定规则（BestChild）进行深探，直至找到一个未探索过（或子节点尚未被全部探索过）的节点，然后扩展（Expand）之。

```
function TreePolicy(v)
    while 节点 v 不是终止节点 do:
        if 节点 v 是可扩展的 then:
            return Expand(v);
        else:
            v ← BestChild(v, c);
    return v;
```

扩展（Expand）部分较为直观，只需要在当前节点下找到一个未探索过的节点，然后在树上增加此点即可。

注：下面的 state(v) 表示节点 v 对应的状态，A(state(v)) 便代表了在这一状态下的合法走子集合。

```
function Expand(v)
```

选择行动 $a \in A(state(v))$ 中尚未选择过的行动

向节点 v 添加子节点 v'，使得 $s(v') = f(s(v), a), a(v') = a$

return v'

最佳子节点（BestChild）是在选择（TreePolicy）部分中提到的下棋点位选择规则，这里我们通过节点的胜率（经验）和被探索次数（探索）等方面衡量子节点的价值。

注：这里 c 是可变参数，越大的 c 表示算法越重视节点探索。

function BestChild(v, c)

return $argmax_{v' \in children\ of\ v} \left(\frac{Q(v')}{N(v')} + c\sqrt{\frac{2ln(N(v))}{N(v')}} \right)$

模拟（DefaultPolicy）部分故名思义，就是以目标节点为起点，然后随意选择下棋位置，直至双方分出胜负为止的过程。

function DefaultPolicy(s)

while s 不是终止状态 **do:**

以等概率选择行动 $a \in A(s)$

s ← f(s,a)

return 状态 s 的收益

完成模拟过程后得到节点的收益，这时需要使用回传（Backup）方法把对应收益传递到祖先节点上。

注：在每一次的循环之中，可以看到收益（△）的符号是不断变化的，这是因为对于一方有利的节点，对于另外一方则是不利的原因。

function Backup(v, △)

while $v \in NULL$ **do:**

N(v) ← N(v) + 1

Q(v) ← Q(v) + △

△ ← -△

v ← v 的父节点

算法改进

进攻与防守

按照上面介绍的算法完成代码后，我发现这个 AI 的棋艺甚至还不如最简单的样例 AI。在简单观察 AI 下了几局以后，发现 AI 经常会忽略对方的必胜节点，因此我在扩展（Expand）部分中加入了剪枝，让 AI 主动进攻我方的必胜节点，以及防守对方的必胜结点。

需要注意的是，一旦找到满足上述条件的子节点后，便不再需要扩展这一节点，原因在于：

1. 若找到必胜节点，则在该处下棋便可直接取胜；
2. 若找到必败节点，则必定要在该处下棋，否则我方便会直接输掉比赛；
3. 若必败节点有两个或以上，则无论下在哪一个节点都不会改变我方的败局，故此时只记录一个节点便可。

进一步思考

增加上面的策略后，虽然 AI 可以避免因为低级失误（没有把棋下在我方必胜点/对方必胜点）而输掉比赛，但是对于水平较高的 AI，即使加入了上述策略后胜率还是十分低。稍加观察后，发现我方 AI 经常会给对方点炮（即在某一处下棋后使得对方出现必胜节点），从而输掉比赛。

为此，我增加了一个额外的剪枝。对于那些不存在终局子节点（我方/对方不能在一步内终结比赛的节点）的节点，在随机扩展的时候需要多考虑一步（即对方的回合），若对方可以在我方下棋的位置的上方下子而取得胜利，则我方便不应该在该处下棋。

当然，即使找不到不会点炮的子节点，我们还是需要给出一个下棋的位置，否则会出现 TLE。

具体实现

在实验框架的基础上，我增加了 Uct.cpp, Uct.h（信心上限树), Node.cpp, Node.h（树的节点), Timer.h（计时器）等文件，其中信心上限树的实现如上一部分所述，树的结点则记录了该回合下棋方及下子位置等信息，对于计时器则使用了微秒级的计时器，保证时间计算准确。

空间优化

最开始的时候，我在每个节点（Node）中都记下了当前状态的整个棋盘和列顶元素，这样做的后果是 AI 在十个回合以内便会 MLE。

经过简单的分析后，我发现对于每个节点都记下整个棋盘是完全没有必要的，这是因为每次搜索都是从根节点往下搜索的，因此只需要在确定了走子位置以后同步更改一个全局的棋盘便可以做到同样的效果。经过这一优化后，程序需要的内存大幅降低，即时下满整个棋盘也不会导致 MLE。

记忆功能

最开始的时候，每一次主程序调用 getPoint 以后我的 AI 都会重新生成一棵 UCT，但是这样做的话会捨弃之前花费大量时间模拟得来的结果。为了重复利用这些数据，我更改了程序的逻辑：

- 对于每次主程序调用 getPoint，我们都会检查一次棋盘的状态，若棋盘所有数字之和为 0 或 1（我方先行/对方先行），则可以新生成一棵 UCT 树，否则只需要更改 UCT 中的棋盘和列顶位置表即可。
- 由于主程序调用 getPoint 时我们会知道对方上一手下子的位置，因此我们可以让 UCT 的根节点变为行走这一步棋的子节点（若没有合适的子节点或根为空则可以重新创建一个），然后删去其他子节点便成功更新了根节点。
- 需要注意的是，在 AI 选择了行走的位置后也需要同步更改根节点，具体方法同上。

实验结果

利用 saiblo 平台测试 AI，得到如下结果：

AI	测试编号	胜	负	平	总局数	胜率/%
v133	#21230	84	16	0	100	84.00
v133	#21376	95	5	0	100	95.00
v133	#21377	95	5	0	100	95.00
v133	#21450	93	6	0	99	93.94
v133	#21452	85	15	0	100	85.00
v133	#21453	94	6	0	100	94.00
v133	#21455	92	8	0	100	92.00
v133	#21925	90	10	0	100	90.00
v133	#21926	93	7	0	100	93.00
v133	#22502	90	9	1	100	90.00
v133	#22503	93	7	0	100	93.00
v133	#23019	95	5	0	100	95.00
v133	#23021	95	5	0	100	95.00
v133	#23022	95	5	0	100	95.00
v133	#23025	91	9	0	100	91.00
v138	#21712	88	12	0	100	88.00
v138	#21727	92	8	0	100	92.00
v138	#21740	96	4	0	100	96.00
v138	#21741	91	9	0	100	91.00
v138	#21923	90	10	0	100	90.00
v138	#21924	93	7	0	100	93.00
总计	-	1931	168	1	2099	91.99

注：上面的 AI 都使用同样的策略，但 v133 每一回合搜索时间上限为 2500ms，而 v138 则为 2800ms。

一些小发现

搜索的时间对结果的影响并没有想象中的大：

- 每步时限为 2500ms 的程序 和 2800ms 的程序 的效果相若。（平均胜率为 91.45% 左右）