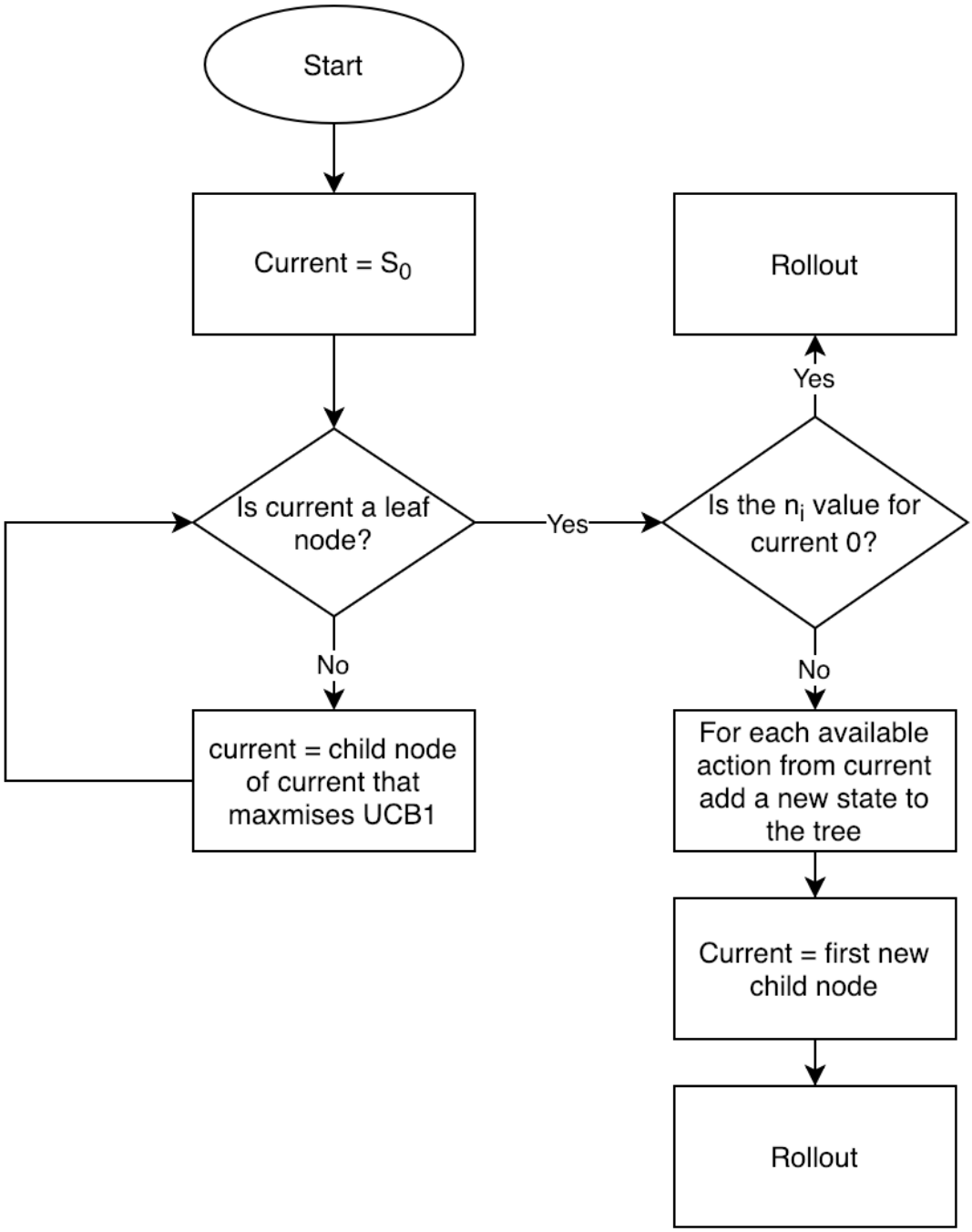
重力四子棋 实验报告

计81 肖光烜 2018011271

算法思路

我采用的是蒙特卡洛搜索，利用信心上限算法进行改进。算法流程如下所示：



其中UCB1（信心上限）表达式如下：

$$UCB1 = V_i + c\sqrt{\frac{\ln N}{n_i}}$$

- V_i 表示当前节点i及其子节点的平均收益（获胜为1、失败为-1）。
- N 为当前节点的父节点被访问的次数。
- n_i 为当前节点i被访问的次数。

网上的教程中，常数c多取2，用以鼓励对未知节点的开拓。但是在我本次的实验中，由于种种优化技巧的加入，对某个节点的一次蒙特卡洛模拟已经能够较好反映当前节点的价值，因此将注意力更多放在有价值的节点上，多将算力分配到对最有可能节点更加深入地斟酌评估是合理的选择。我将常数c取作1，当然也可能有更好的选择，由于时间和算力的限制我没有进行更多参数的评估和选择。

实现设计

在提供的框架代码的基础上，我采用了OOP的设计思路，加入了四个文件：`MCTree.h/.cpp`、`Phase.h`、`constants.h`，分别实现了核心的蒙特卡洛搜索树算法、位运算加速棋盘和实验中常数的保存。本AI代码模块化好，可读性佳。

优化技巧

为了击败`100.so`（和在天梯上不断内卷），我历经苦苦探索进行了许多优化（操作），最终采用的优化有下列几项，比单纯不加优化的信心上限树有了很大的提高。

首先开宗明义，总体的优化思路有两条：

- 性能优化：在有限的时间内能够进行越多的模拟，走子的位置就会越为精准。
- 策略优化：在模拟的过程中更加聪明（具有智能），模拟就会更有意义，模拟的结果就更能反映真实走子价值。

围绕这两方面，我采取了诸多技巧。

位运算存棋盘

位运算存棋盘是性能上的优化。起初我还想用一个unsigned long long存下整个棋盘，但是由于棋盘的最大规模为144，大于64个bit，因此我采用了一个short（2 byte = 16 bits）数组方法存储棋局。例如一个M*N的局面，就可以用两个长度为N的short数组分别存下对手的落子位置和我的落子位置（1表示有落子、0表示无），同时用两个整数维护不可落子位置即可。

这样的好处在于，我可以利用位运算加速对胜负的判断和局面估值等一系列操作（这是在蒙特卡洛模拟中最为耗时的操作，框架自带的`Judge.h/.cpp`太慢了），具体做法见`Phase.hpp`中的`bool alignment(const unsigned short* pos) const;`函数即可明白，在此不再赘述。

节点池

在蒙特卡洛模拟过程中会有大量子节点被开拓，如果用动态内存管理的话显然会非常耗时，于是我在棋局开始之时一口气开了`1.1e7`个节点（`Node`对象），同时写了一个`NodePool`节点池类管理这些节点的行为，这两个类都在`MCTree.h/.cpp`中。

树根移动

这同样是性能上的优化。试想如果每次调用`getPoint()`函数都要重新构造蒙特卡洛树进行局面判断，显然之前的运算就全部浪费掉了。如果对自己的算法足够有信心，那么对方下一招也多有可能采用当前模拟树下一层的某个模拟次数较多的节点，因此下一步的时候不重新建树，而将树根移动到上一步的模拟树对方所对应的下一层招法即可。因为每一步的思考都基于前一步思考的基础上，这样其实变相将每一步的思考时间拉长了，并且也更加符合人类下棋的行为（思维是连续而不是断裂的）。

当然，这样的做法就不会回收废弃的节点了，会面临节点池快速用尽的情况。我目前的做法是：当节点池有用尽的危险时，就不再移动树根而是从0号节点开始重新建树（这会面临中盘由于算力不足被人捡漏的风险，但是据我的观察一般到40多步节点池才会用尽，所以对阵100.so这种鱼腩还是够了）。当然我想会有更加高明的数据结构支持滚动移动树根的操作，这也是我今后优化这个AI的方向之一。

根据估值随机选点

蒙特卡洛模拟过程中随机傻搜当然没有问题，性能是最好的，但是这会导致估计局面过程中错过双方特定的“妙手”，从而对局面有盲目的自信或悲观。因此在选点过程中根据一定的先验分布进行随机选点是值得优化的选择。可以根据人类对四子棋的先验知识对局面中可能的落子进行评价，给可能更好的落点更多权重，从而达到**聪明模拟**的效果。

但同时这也面临性能与“聪明”之间的权衡取舍，过于聪明必定带来的是估值过程中的计算开销（不然还用MCT搜啥），因此设计的估值函数必须要简单、快速而且行之有效，同时不给随机模拟过程带来过大的扰动（不然就不是蒙特卡洛模拟了）。

中和之道

首先最简单容易想到的就是：**棋盘的中路一般都不会太坏**。这是由于棋盘的中路拥有向两侧发展的权利，同时也能限制两侧对手的发展，因此中路在开盘必是兵家必争之地，在随机选点中给中路更多的权重是合理的。

我采用了如下的权重分配：

$$Score(move) = (move\text{距离最近的边界的距离})/3 + 1$$

这样最好的点分值为3，最坏的点分值为1，偏差不太大，不会扭曲蒙特卡洛方法，同时计算快速、带来先验信息。在引入”中和之道“之前我对100.so只有大约50%的胜率，在引入后瞬间击败了100.so，胜率稳定达到90%以上，此简单的估值函数不可谓不强大。

必胜点与救火点

在局面中，同时会出现必胜点和救火点。所谓必胜点，即下在这个位置就能取得胜利，此时选取其他任何点都可能错失良机，从而对局势评价产生偏差。所谓救火点，即在不存在我方必胜点的情况下对方的必胜点，如果此时不下在这里对方的下一步就会抢占此点从而获得胜利。因此在蒙特卡洛模拟的某步中如果存在必胜点和救火点，那此步就不要再随机了，直接下在这里！这种策略使得对局势的模拟变得更为精准。

“三连威胁”的进攻与防守

这个就是我玄学想出来的估值了，没想到还意外好用。在四子棋游戏中，如果己方造成了可能的“三连威胁”局面：即在连续的四个格子中有三个自己的棋子，那么对方下一步肯定就要在剩余的那个位置上进行“救火”，此之谓”三连进攻“。同样，如果对方在某四个连续的格子内已经有2个子了，那么下一步下在另两个位置的任一个就会创造一个”三连威胁“，于是我方如果下另两个位置的任一个就会破坏对方的潜在的“三连进攻”，此之谓“三连防守”。

因此，把某个点的“三连进攻”数和“三连防守”数加起来作为权重，也是一种较好的估值函数。

总结

最终，我通过累加的方法将以上三种估值方法组合起来，达到了不错的效果。当然也面临了性能上的trade-off：完全随机的模拟在我的电脑上每步能进行至少2e6次模拟，但是采用了估值函数之后每步仅能最多做到2e5次搜索，这使得搜索深度大幅降低。但是又由于模拟过程更加”聪明“，再结合先前提到的**移动树根**策略，优秀的局面判断可以不断累加模拟次数，从而抵消估值函数的性能影响，达到更强大的局面模拟精度。

剪枝

在正如先前提到的，如果局面中存在必胜点和救火点，那么下一步必然会走这个位置而不会考虑其他。同样还可以考虑**禁入点**：如果下在这个位置会给对方造出一个必胜点，此点称为**禁入点**。在扩展叶子节点的过程中可以检查下一步是否存在必胜点、救火点、禁入点。将不必要的分叉减掉，可以使每次选择best child的过程得到加速。

其他尝试

更柔和的收益判定

在一开始的时候我曾经考虑过更加柔和的收益判定函数，基于以下思路：**如果当前局面非常不利，那么能够拖得越久造成对方失误的可能性就越大**。于是可以采用以下的收益函数：

$$Value(\text{胜者}) = -Value(\text{败者}) = \frac{\text{局面剩余的步数}}{\text{棋盘的大小}}$$

这样的收益函数追求**速胜**和**慢败**，但是实验来看效果并不好。我的猜想是：利用蒙特卡洛模拟对后期局面的模拟并不精确，这种收益函数会导致后期的胜利和失利对整体收益的刺激不够，因此最终弃用了这种收益函数。

更激进的随机剪枝

在随机模拟过程中，如果像拓展叶子结点一样也引入**禁入点**也是一种容易想到的策略，但是实验看来效果并不好。原因在于禁入点的判断是十分耗时的，会严重拖累模拟的速度，因此没有采用。

对战成绩

我采用要求的测试方法，即每轮交换先后手在相同棋盘上对阵两局，总共对阵5轮10局的方法与助教提供的50个AI进行了对战，以下列出的是有败绩的对战记录，其余未列出的均为全胜。

对手**AI**	胜局	平局	败局	胜率（**%**）
98	9	0	1	90
94	8	0	2	80
92	9	0	1	90
90	8	0	2	80

可见我的AI对助教提供的50个AI是碾压的水平，几乎没有悬念。

总结收获

在本次作业中，我通过亲手实现对蒙特卡洛树搜索算法和信心上限算法有了更加深入的认识，同时对性能的种种优化和对策略的调试也加深了我对限时博弈游戏采用蒙特卡洛树算法的优势和注意事项。除此之外，我在思考策略优化时也研究了$\alpha-\beta$剪枝算法的优势和劣势，对博弈搜索算法有了较为全面的认识。