四子棋大作业实验报告

计 71 张程远

2017011429

目录

1 问题描述 3

2 实验原理 3

3 实验结果 3

4 遇到的问题 4

4.1 内存泄漏 4

4.2 超时 4

5 讨论 4

5.1 负优化 4

5.2 Q-Learning 解四子棋问题的可行性 5

6 致谢与 HonorCode 5

1. 问题描述

重力四子棋是一类智力游戏，双方轮流在棋盘每一列的最底部落子，直到某一方的棋子在某一方向上连成连续 4 子即获得胜利。本次实验要求我们实现一个四子棋对战 AI，完成 AI 的决策部分代码，并与提供的测试策略对战。实验用的四子棋棋盘长宽为[9,12]的随机数，且每局棋会随机产生一个不可落子点。双方程序每步的执行时间为 3 秒，超时即判负。

2. 实验原理

UCT（信心上限树）是蒙特卡洛搜索的一种实现方式。UCT 每次都从根节点开始向下搜索，并考虑根节点的所有子节点。如果所有的子节点都已被发现并搜索过，那么从中选择最佳节点继续重复上述步骤；如果存在节点未被搜索过，则优先搜索此节点。对这一节点的处理方式为，在选择这一节点后进行向下随机模拟，直到达到终止节点获取奖励，之后把奖励返回给这一节点。也就是说，当遇到没有搜索过的节点时，我们暂时采用一种随机结果作为对这一节点的评价。之后不断重复上述过程，直到搜索结束。

所谓最佳节点的评判标准含有两个参数：节点的访问次数以及这个节点累计获得的奖赏。我们认为节点访问次数越少的情况下，获得的奖赏越多，那么这一节点就越好。我在选择最佳节点时采用了下面的公式计算：

$$\mathrm{U} = \frac{V(i) + 1.0}{N(i) + 0.00001} + 0.7 * \sqrt{\frac{2 * \ln(N(i.father) + 1.001)}{N(i) + 0.0001}}$$

其中 U 表示信心值，V(i)表示节点累计得到的奖赏，N(i)表示节点的访问次数，N(i.father)表示节点 i 的父节点的访问次数。信心值最高者即为最佳子节点。给分子分母加上一个ε的原因是使未搜索节点的信心值较大（N(i)==0），从而优先搜索未搜索节点。在博弈树的规模比较小的时候，后一项可以让程序更倾向于选择“父亲访问次数较大但自身访问次数少的节点“，防止某些节点因前期随机到的奖励值过小而被忽视；当博弈树规模不断扩大，整体信心值将主要由前一项所决定。

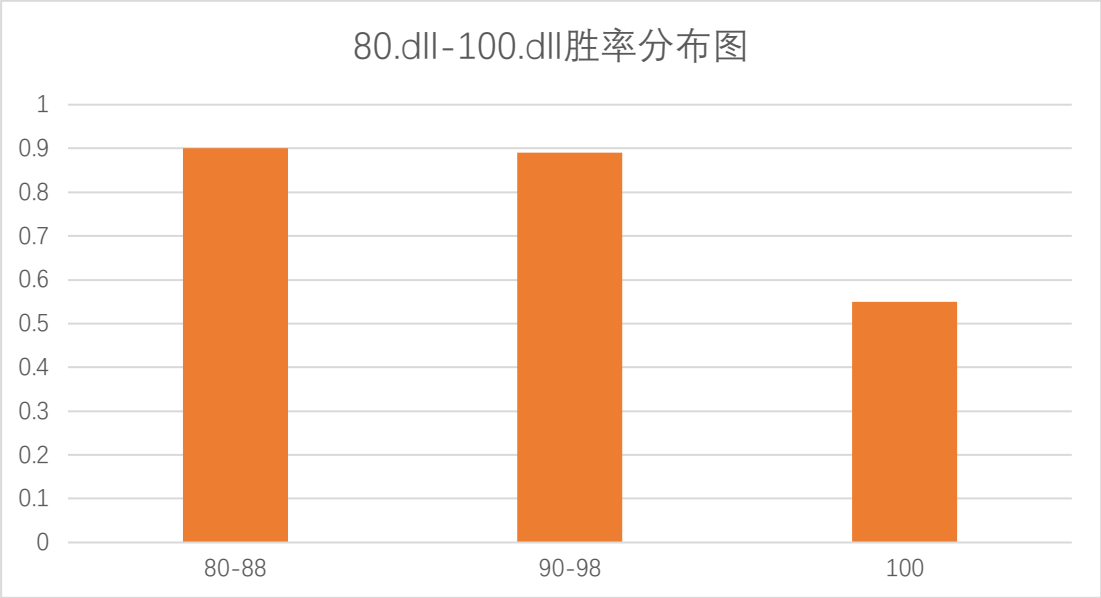
关于奖励的设置，程序将赢棋奖励设定为 1，输棋奖励设定为-1，平局奖励设定为 0。如果下到某一步局面未结束，则把奖励值设置为-3。在向下随机的过程中，直到某一步落子返回奖励值为±1 或 0 才会更新路径节点的奖励值。但这又带来一个问题：我们有时要模拟棋局，模拟时双方所选的策略都应该是最佳策略，这样做出的决定才是正确的。但是我们为对方选择落子位置的时候，所得到的 U 值是负的，这样我们按最大值选出的反而是奖励积累较少的节点。因此我们计算奖励时加入一个参数即当前这一步由谁来下，如果是对方来下，则式子中的 V(i)应该变成-V(i)。

3. 实验结果

下面为与 80.dll-100.dll 对战 10 个 round 之后得到的结果。

编号	先手胜率	后手胜率	总胜率	编号	先手胜率	后手胜率	总胜率
80	0.90	0.70	0.80	**82**	1.00	0.90	0.95

84	1.00	0.80	0.90	**86**	0.90	0.90	0.90
88	1.00	0.90	0.95	**90**	1.00	0.80	0.90
92	1.00	1.00	1.00	**94**	0.90	0.90	0.90
96	0.80	1.00	0.90	**98**	0.90	0.60	0.75
100	0.50	0.60	0.55				



从以上两个图表中可以看出，基于 UCT 实现的策略对抗效果不错，经过 10 轮先后手棋局，对抗策略 80-96 都取得了 80%以上的胜率，平均胜率达到 90%；对抗 98.dll 胜率达到 0.75，与 100.dll 实力基本相当，基本完成了与给定策略对抗的任务。

4. 遇到的问题

4.1 内存泄露

我遇到的内存问题主要由以下两个原因构成：首先 dll 的内存不能在给定函数的外面进行修改。所以程序中规定了一套参数用以复制函数中传入的参数，传入的参数完全不修改只读取，修改只发生在复制的参数中，这样就避免了这个问题；第二个问题是个人的疏忽，程序运行时全局变量的值会随函数每次被调用而改变，因此在 getPoint 函数的开始需要将一些值重置，如节点的使用个数、节点是否有子节点等信息都需要重置。

4.2 超时

为了每手都在 3 秒的时限内下完，需要在函数内设计一个时间限制，当超过时间限制时就中止棋局。最开始设定的时限是 2.9 秒，但是有些过于紧张了，还是会导致超时；之后看到群里的讨论将时间设置为 2.2 秒，但是这样搜索效果比较差，经常没有达到循环步数就跳出了；最后设定时限为 2.6 秒，没有再测出超时的现象，同时也能保持比较好的搜索效果。

5. 讨论

5.1 负优化

虽然在扩展新节点时采用完全随机模拟得到的结果来评价听上去很不靠谱，但实际上这

正是蒙特卡洛搜索所需要的。我曾经尝试过一个“优化“，即在扩展到新节点并开始模拟的第一步时，考察上一轮的 top 数组，并将这一步放在 top 数组中己方棋子的附近。这样打破”随机“的方式实际上是不可取的——对战 100.dll 时并没有体现问题，但对战 98 和 96 时胜率跌到了 4 成左右。实际上，这里的随机模拟已经能够搜到绝大多数情况，因为蒙特卡洛搜索树搜索较多时会倾向于形成单链，这足以快于寻常的剪枝。我将未加优化的程序与其他同学实现的 UCT 程序进行对比，结果也基本五五开。所以这里的随机性看似不靠谱，实则能够大大提高搜索效率。

5.2 Q-Learning 的可行性

Q-Learning 的思想实际上和 UCT 的想法有共通之处，也许可以将其用于解决四子棋问题。Q-Learning 的基本思想是：在 S1 状态下有几个 action 可供选择，我根据 Policy 选择某一个 action A 进入到状态 S2。在 S2 状态，我们只向下模拟一步，看看 S2 能获得什么 Q 值并带来什么样的 reward，并以此更新 Q 值函数 F(S1, A)的值。Policy 通常是一个比例，决定这一步选择 action 时是选择 Q 值最大的行动还是随机一个行动。函数的更新公式如下：

$$\mathrm{F(S,a)} \leftarrow \mathrm{F(S,a)} + \alpha[\mathrm{r} + \gamma max_{a' \in action(S')}(\mathrm{F}(S', a')) - \mathrm{F(S,a)}]$$

其中α为学习率，γ为衰减系数，通常小于 1。α为 1 的时候 F(S,a)与原本的值完全无关。γ体现了状态之间的距离，距离越远的状态对当前状态影响越少。它与蒙特卡洛搜索的核心区别在于，它对于新节点的态度并非像蒙特卡洛那样随机一个结果出来作为其评价，而是直到找到有奖励的节点才自底而上慢慢扩大影响。

Q-Learning 应用到棋局中的方式也与 UCT 基本类似，每个局面都是一个状态，落自选择构成 action 集合，计算 Q 值最大值即为寻找子节点存放奖励的最大者，在为对方选择落子时将 Q 值取负即可。Q-Learning 对于固定棋盘固定禁手的规则应是有效的，但在这里不是非常适用，因为 Q-Learning 节点是逐层之间影响，在一开始会搜索到大量的 0，非常耗时，而每次不同的局面意味着模型要重新训练，因此效果不会很好。不过 Q-Learning 在理论上是一个可行的方式。

6. 致谢与 HonorCode

感谢计 74 的陈果同学，他提醒我在给对手落子的时候要让信心值取负数，我们还讨论了在我的写法中如何确保先搜新节点等细节的实现。算法学习参考下面的网址：

https://blog.csdn.net/u014397729/article/details/27366363

Q-Learning 的部分参考自下面的网址：

https://morvanzhou.github.io/tutorials/machine-learning/reinforcement-learning/2-1-A-q-learning/

最后感谢助教和讨论区的同学提供的答疑解惑！