四子棋实验报告

2017011620 计 73 李家昊

2019 年 5 月 13 日

1 算法基本思路

该实验中采用蒙特卡洛树搜索（Monte Carlo tree search, MCTS）与信心上限树（Upper Confidence Trees, UCT）相结合的算法。标准 MCTS 算法主要分为 Tree Policy, Default Policy, Back Up 三个部分，算法流程如下：

Algorithm 1 General MCTS Algorithm

```
function MctsSearch(s_0)
    create root node v_0 with state s_0
    while within computational budget do
        v_l ← TreePolicy(v_0)
        Δ ← DefaultPolicy(s(v_l))
        BackUp(v_l, Δ)
    return a(BestChild(v_0))
```

1.1 Tree Policy

这一部分完成 MCT 上结点的扩展。若结点未扩展完全，则优先扩展未扩展的结点，并返回；否则，选择 UCB 值最高的孩子，循环上一操作。

每个结点的 UCB 值定义如下：

$$\mathrm{UCB} = \frac{Q}{N} + c \cdot \sqrt{\frac{2 \ln N_p}{N}}$$

其中 Q 为该节点赢的次数，N 为该节点的访问次数，N_p 为父亲结点的访问次数。c 为超参数，用于调节 exploitation 和 exploration，具体调参实验见下文。

1.2 Default Policy (Rollout)

这一部分进行随机模拟。从新拓展的结点开始，双方循环随机落子，直到游戏结束。双方决出胜负，或者平局，计算得到 reward，记为 Δ。

1.3 Backup

向上更新自身及父代的 Q 值，对每个父代结点，访问次数 N 增 1，并用终局的 reward 更新其 Q 值，获胜方给予正 reward，失败方给予负 reward，即

$$Q' = \begin{cases} Q + \Delta, & \text{if won} \\ Q - \Delta, & \text{if lost} \\ Q, & \text{if tied} \end{cases}$$

2 算法优化

2.1 必胜剪枝

实现随机的 MCTS 算法后，发现超过两步的必胜走法很难预测出来，于是考虑剪枝。

正常人下棋时，只要下一步能赢，就必定会落到能赢的位置，而不会考虑其他位置，我们称这种局面为“必胜局面”。

于是考虑剪枝：在下一步能够赢的时候，将赢棋的落子位置设为合法，其他位置设为非法，这样一来，只要出现“必胜局面”，无论在 Tree Policy 还是在 Default Policy 过程中，必定只有一种走法，从而实现剪枝，剪枝数量为 $N-1$，原本在这 $N-1$ 条枝干上，由于 MCTS 的随机性，可能存在非常多的结点。

2.2 必败剪枝的尝试

同样的，当对方下一步要赢的时候，将阻碍对方赢的落子位置设为合法，其他位置设为非法。

但由于有必胜剪枝，即使不进行必败剪枝，若 MCTS 扩展出其他位置，那么对方下一步必定会选择必胜策略，虽然必败剪枝数量也达到 $N-1$，但真正剪掉的结点只有 $2N-2$。这一改进可看似合理，实际上意义不大。

2.3 其他剪枝的尝试

尝试进一步优化，比如当我方走某一步棋时，对方下一步将要赢棋，那么就不走这一步。这么做同样意义不大，因为若走了这一步，对方下一步必走必胜策略，剪枝数量仅为 1，剪掉结点数量仅为 2。

2.4 Discount Factor

人们总是希望在最短的时间内赢棋，在 MCTS 中同理，越深的结点对根节点的胜率贡献越小。

因此，在 Backup 过程中，从初始结点开始，将赢棋的 reward 初始化为 1，每次向上更新父亲的 Q 值时，将 reward 乘上一个衰减系数（Discount Factor），记为 λ，再更新父亲的 Q 值。这里取 $\lambda = 0.99$ 。即对于每个结点，若其高度为 h，则更新的 Q 值大小为

$$|\Delta Q| = \lambda^h$$

优化后，运行效率基本不变，但棋力有了轻微的提升。

2.5 棋盘和 top 数组的共享化

一开始在每个 Node 中都独立存储了一个棋盘和 top 数组，这样做空间消耗极大，且运行效率极低。

考虑到每次沿着 MCT 遍历的时候，棋盘和 top 数组的更新是局部的：从父亲到孩子，只需要在棋盘中增加孩子的落子，并更新 top 数组这一列的值即可；从孩子到父亲，只需要在棋盘中去掉孩子的落子，并更新 top 数

组这一列的值即可。因此将 board 和 top 数组定义为全局共享的，每次迭代时对其进行局部更新，降低了空间消耗，提高了运行效率。

2.6 预先分配内存

new 和 delete 开销非常大，经测试，每次 delete 1M 个结点，消耗时间约为 200ms。为避免 new 和 delete 的开销，在程序开始时，我预先分配了内存池，定义了 5M 个结点的静态数组，在程序运行过程中，不再使用 new 和 delete 操作。

2.7 MCT 的复用

在预先分配内存池中有 5M 个结点，一次落子的搜索必然用不完，若全部清理，则造成浪费，因此可以考虑 MCT 的复用。

每次初始化时，可以根据自己上一步的落子，以及对方上一步的落子 (lastX, lastY)，可以在先前建立的 MCT 中，找到以当前状态为根的一颗子树，这样就省去了再次建树的操作，并且很多结点都被保留下来，在效果上相当于隐性的增加了迭代次数。

若初始化时，内存池预留的结点不足 1M ，则会将内存池清空，并以当前状态为根节点，重新建立 MCT。

2.8 重复状态的消除

MCT 上结点的 State 有很多都是重复的，事实上，只要每一列双方落子的顺序相同，则必然产生重复的 State，这种重复造成了大量的内存开销以及效率开销。

于是采用高效的哈希函数，将 State 映射到哈希表上，每次扩展新结点时，若节点的 State 已在哈希表中出现过，则不再扩展新结点，而是直接沿用旧结点，这样一来，成百上千重复结点的 Q 值和 N 值被综合到一起，在效果上相当于隐性地增加了搜索次数。

2.9 最后的优化

最后的优化就是把上述所有剪枝、Discount Factor、哈希表、MCT 结点复用全部去掉，此时棋力达到顶峰。由此得出难以解释但被实验证实的结论：几乎所有的优化都是劣化，不做任何剪枝的 AI 反而是最优的，即原生 MCTS + UCT 的随机策略反而是最聪明的。

3 调整参数

UCB 算法中，c 值需要根据实际情况而定，这里进行了调参实验。考虑到 100.dll 经常 timeout，无法反映其真实水平，于是这里令 AI 在不同的 c 值下与 98.dll 对战 50 局（先后手分别 25 局），首先进行粗调，胜率如下表所示：

c	0.6	0.7	0.8	0.9
Win	78%	92%	88%	86%

表 1: 不同 c 值下对战 98.dll 的胜率

可以看出，最优 c 值应处于 0.6-0.8 之间，下面进行细调。

c	0.60	0.62	0.64	0.66	0.68	0.70	0.72	0.74	0.76	0.78	0.80
Win	78%	82%	80%	84%	92%	92%	92%	86%	90%	84%	88%

表 2: 不同 c 值下对战 98.dll 的胜率

由表可知，当 c 在 0.7 附近时，胜率最高，符合论文建议的最优值，最终取 $c = 1/\sqrt{2}$。

4 评测结果

设置迭代时间为 2.5 s，迭代次数平均为 1M 次。与所有测例分别对战 50 局（先后手分别 25 局），胜率如下：

AI	Winning Rate	AI	Winning Rate	AI	Winning Rate
2.dll	100%	36.dll	100%	70.dll	100%
4.dll	100%	38.dll	100%	72.dll	96%
6.dll	100%	40.dll	100%	74.dll	98%
8.dll	100%	42.dll	100%	76.dll	98%
10.dll	100%	44.dll	100%	78.dll	100%
12.dll	100%	46.dll	98%	80.dll	96%
14.dll	100%	48.dll	100%	82.dll	98%
16.dll	100%	50.dll	100%	84.dll	92%
18.dll	100%	52.dll	98%	86.dll	98%
20.dll	100%	54.dll	100%	88.dll	94%
22.dll	100%	56.dll	100%	90.dll	94%
24.dll	100%	58.dll	98%	92.dll	90%
26.dll	100%	60.dll	100%	94.dll	96%
28.dll	100%	62.dll	100%	96.dll	94%
30.dll	100%	64.dll	100%	98.dll	90%
32.dll	100%	66.dll	100%	100.dll	92%
34.dll	100%	68.dll	100%		

表 3: 改进后的 AI 与所有测例 AI 的对战结果

改进后的 MCTS 取得了不错的效果，对战编号 80 以上的 AI，胜率在 90% 以上。其中，测例 64.dll 和 66.dll 存在比较严重的 bug，经常导致 Compete.exe 卡死（内存占用约 2G，CPU 占用 0%，且 Ctrl+C 都杀不死），希望最终评测时能保证测例正常运行。

5 总结收获

1. 通过本次实验，我实现了 MCTS 和 UCT 算法，加深了对随机搜索算法的理解，并将其应用到实际问题中。

2. 另外，我还对标准的 MCTS 进行了优化，加上了人的先验知识，但发现未做任何剪枝的 AI 是最聪明的。

3. 感谢这个平台提供的易用接口，以及助教的耐心指导！