重力四子棋：实验报告

刘泓尊　2018011446　计 84

目录

Saiblo 用户名：Escape, AI: version1, 版本：9

源代码及 makefile 放在./src 文件夹下，您可以压缩之后 (src.zip) 上传至 Saiblo 网站进行评测

1 算法思路

考虑到 alpha-beta 剪枝中估价函数的参数难以准确设置，本实验我采用“蒙特卡洛树搜索”结合“信心上限树算法”，在有限时间内随机模拟节点的胜率，加以一些简单的启发性策略，取得了不错的效果。

1.1 算法流程

蒙特卡洛树搜索 (MCTS) 通过一个已知的初始状态，动态地进行状态的搜索，逐步扩展树的节点 (Selection)，每进行一次扩展 (Expansion)，便对该状态进行一次随机模拟 (Default-Policy)，并将模拟结果反向传播 (BackPropagation) 给祖先各节点。

信心上限树算法 (UCT) 将蒙特卡洛树搜索与 UCB 策略进行结合，设置了可以量化的指标 UCB; 公式为:

$$UCB = \frac{Q(v)}{N(v)} + c\sqrt{\frac{2\ln(N)}{N(v)}}$$

其中 $Q(v)$ 为节点 v 的总得分，$N(v)$ 为该节点被回溯的总次数, N 为总次数。UCT 算法每次从根节点开始，选择一个可以扩展的节点，对该节点使用 Default-Policy 进行随机模拟，最后将收益 (胜为 2，平为 1，负为 0) 反向传播给祖先节点，最后选择根节点的儿子中收益最大的一个为落子点。

function UCTSEARCH(s_0)
以状态s_0创建根节点v_0;
while 尚未用完计算时长 **do**:
v_l ←TREEPOLICY(v_0);
Δ←DEFAULTPOLICY(s(v_l));
BACKUP(v_l, Δ);
end while
return a(BESTCHILD(v_0, 0));

1.2 节点信息

每个节点维护信息如下:

Node.h

```
class Node{
    char player, winner;//当前节点玩家，胜者
    int totaltime, wintime;//总模拟次数，胜利次数
    double ucb_a, ucb_b;//计算ucb的辅助信息
    int* top; //当前节点的可落子点, range: 0-12
    Node** childlist;//孩子序列
};
```

1.3 一些优化

1.3.1 内存池

每次扩展节点的时候都新创建一个节点将消耗很多时间,因此,我先开辟了一个 poolSize(=5×10^6) 的 Node 内存池，在每一局开始的时候申请，结束时清空即可。此策略可以使有限时间内的模拟次数提升 1.5 倍左右。

1.3.2 UCT 子树重用

在每一回合都重新建树会消耗大量的时间，同时也丧失了上一回合的模拟结果。所以我在下一回合的时候并不将树完全释放，而是将根节点下移两层到达下一回合的节点，这样可以利用上一回合的计算结果，同时避免了大量析构节点的开销，实现性能和效果的较大提升。加入此优化之后，对战 100.dll 的胜率从 0.6 上升到 0.8。

1.3.3 时间优化

计算 UCB 值需要消耗很多的浮点运算资源，经过观察，我发现 UCB 值可以拆解为两部分，即

$$UCB = UCB_a + UCB_b \cdot X$$

其中:

$$UCB_a = \frac{Q(v)}{N(v)}, \quad UCB_b = \frac{1}{\sqrt{N(v)}}, \quad X = C \cdot \sqrt{2\ln(N)}$$

这样，可以在全局保存 N, 这样每轮模拟的 X 便不需要重复计算。每个节点保存对应的 UCB_a 和 UCB_b，这样便可以节约很多的计算时间。我最终将时限设置为 1.9s.

由于 UCB 的计算精度要求不是很高，我使用了快速倒数根号算法 (fastInvSqrt) 和快速对数 (fastLog) 算法。这些算法可以避免大量浮点运算，进一步节省计算时间。(见 **utils.cpp**)

为了使得选择阶段的策略更加“一致”，我在每个节点保存了 player 变量，如果 player 是我方，其 $UCB_a = Q(v)/N(v)$, 否则 $UCB_a = 1 - Q(v)/N(v)$。这样就可以保证，选择子节点的时候，总是选择 UCB 值最大的那个，使得代码更加简洁。

1.3.4 空间优化

上述“时间优化”在全局保存 N 的值已经可以在每个节点节省一个 int 的大小。此外，我对棋盘的存储做了空间优化，每个节点不必保存其局面下的棋盘，而是设置一个临时棋盘 board 和初始棋盘 init_board, 每次模拟将 board 上选择的落子点标记为 player, 在模拟结束后恢复为 init_board 即可，大大减少了所需内存。

1.3.5 必胜和必败情况

模拟过程 Default-Policy 采用完全随机的落子策略，与人的下棋策略相去甚远，其得到的结果可存在一些无法修正的偏差。因此，我加入了模拟过程中对必胜和必败情况的下棋策略，其余情况依然随机落子。这样，既可以保证蒙特卡洛方法的精髓，又使得模拟情况更加准确。具体做法是: (1) 如果模拟过程中遇到必胜点，就直接选择该落子点。(2) 如果遇到对方的必胜点，我方直接落在该点，阻止对方获胜。加入该启发信息之后，与 100.dll 的胜率从 0.5 提升到了 0.8。

初始过程和选择过程 在开始 UCT 算法之前，也应该进行必胜和必败情况的检测，因为这样可以免去后续的模拟步骤，快速做出反应。即，在轮到我方的时候，就直接检测是否存在必胜点和必败点，如果存在该点，就直接在该点落子。选择过程中同理。

1.4 尝试的改进

1.4.1 走子策略的衡量

朴素的 UCT 算法使用根节点的子节点中胜率最大或者获胜次数最多的节点作为下一步的走子，但是实测发现随机模拟策略存在一定缺陷，不能很好反映长期状况。因此我考虑根节点至其孙子节

点这三层的信息，保证当前走步胜率最大，同时保证对方下一步胜率尽可能小。即

$$\arg\max_{m_j} P(m_j) + \alpha(1 - P(O|m_j))$$

可以看到，上述评价函数综合我方胜率和对方胜率，使得算法尽可能做出有利于自己而不利于对方的决策。对实测和 100.dll 的胜率有 0.1 的提升。

1.4.2 加入更多的启发策略

在模拟过程中，之前仅仅考虑了存在“必胜/必败”落子点的情况，但是此思路也可以扩展到“思考两步”。即检测当前局面是否存在可以“三连”的情况。我尝试使用了此策略，但是效果的提升并不大，基本与改进前相同。推测是蒙特卡洛方法的随机性保证了其结果十分接近实际情况。

1.4.3 加入浅深度 alpha-beta 剪枝

有了“思考两步”的想法，自然就延伸到了可以在模拟过程中采用浅层的 alpha-beta 剪枝。我测试了在每一次模拟过程中，对当前节点做深度为 4 的 alpha-beta 剪枝，实际上等价于思考 3 步，此处的剪枝依然需要恰当的估价函数，经过我的测试，局面评估可以如下设置:

- OOOO 型局面: 200000
- OOOX, OOXO 局面: 50000
- OOXX, OXOX 局面: 10000
- OOXX 局面: 5000
- OXXO 局面: 500

1.4.4 调节常数 C

改进算法表现的最便捷方式便是调节常数 C 了。C 的意义在于: C 较大时更有机会选择兄弟节点 (宽度)，C 较小时则优先扩展深度。我测试了不同 C 下与 90.dll-100.dll 的胜率，最后选择 $C = 1/\sqrt{2} = 0.71$. 即

$$UCB = \frac{Q(v)}{N(v)} + \sqrt{\frac{\ln(N)}{N(v)}}$$

1.5 必胜策略的探索

我从文献中找到，四子棋在存在一种策略可以保证自己不败。实际上，当不可落子点处于比较靠上的位置时，不可落子点对原始必胜策略的影响是很小的，所以原始四子棋的必胜策略对于算法依然有指导意义。必胜策略中很重要的一条规则便是: **阻止对方-OO-情形的产生**。因此我在模拟过程中加入了这种启发信息，在模拟过程中对方出现“-OO-”局面时，我方立即下在两侧，阻止这种情况的出现。加入这种改进之后，对战 100.dll 的胜率又有所上升。这也提醒我们，在 MCTS 算法的随机模拟过程加入适当的启发信息，可以有效提高算法表现。这一点从 AlphaGo 在模拟过程中加入估值网络也可以验证。

2 胜率统计

Saiblo 用户名: Escape, AI: version1, 版本: 9

我使用 Saiblo 平台上的批量测试功能进行对战，下面我列出对战过程 (每个 AI 对战 5 个回合 10 场) 中胜率 <1 的情况，**未列出部分均为“完胜”。整体胜率** 97.8%. 5 次对战的截图见下一页。可以看到，算法的表现十分出色，先手胜率为 1，后手表现略差，面对最强大的 94.dll(于我的 AI 而言) 胜率也达到了 0.8。

未完胜的对手	60.dll	80.dll	86.dll	92.dll	94.dll	96.dll	100.dll
先手胜率	5/5	5/5	5/5	5/5	5/5	5/5	5/5
后手胜率	4/5	4/5	4/5	4/5	3/5	4/5	3/5
总胜率	0.9	0.9	0.9	0.9	0.8	0.9	0.8

图 1: 第 1-5 次测试

我的 AI 在 Saiblo 平台天梯上进行对战，一度获得了第 2 名 (Escape) 的好成绩，下面是对战截图:

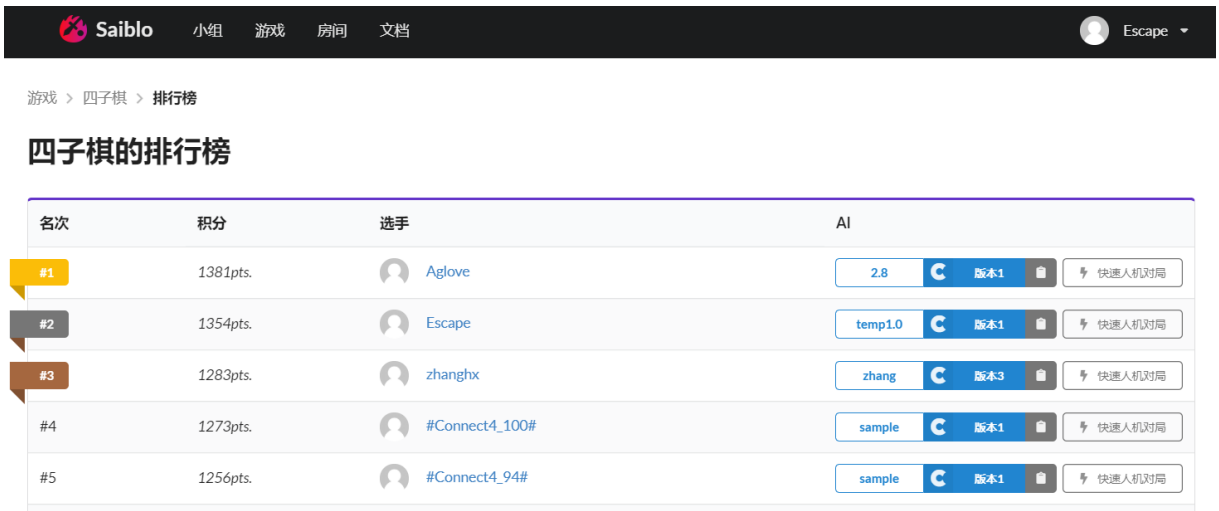


图 2: Saiblo 天梯 rank

3 改进思路

3.1 通过神经网络进行局面的评估

借鉴 alphaGo 的思路，我们可以训练一个神经网络作为估值网络，以当前盘面作为输入，输出该盘面的估值和各个落子点的胜率。此方法也可以用于 alpha-beta 剪枝的估值函数。其效果已经在 alphaGo 中得到了很好的验证。但是重力四子棋和围棋的区别是四子棋的“三连”等棋子聚集现象不一定真的价值很高，因为每次落子都是落在该列最下侧，所以使用带有卷积的残差网络可能拟合效果欠佳，这一猜测有待验证。

3.2 改进 UCB 函数

经过仔细思考，我认为可以针对 UCB 算法做出简单改进。在搜索过程中，不同的后辈节点可能会选择相同的落子点，这些相同选择的节点在一定程度上可以视为“极为相似的”。所以，不妨统计某一节点在模拟过程中，每一个相同落子策略的出现次数 $N\hat{(}v)$ 和由该节点导致的胜利次数 $Q\hat{(}v)$. 由此得到 $U\hat{C}B_a = Q\hat{(}v)/N\hat{(}v)$. 采用插值法修正 UCB 的值为:

$$UCB = \beta \cdot \frac{Q(v)}{N(v)} + (1-\beta) \cdot \frac{Q\hat{(}v)}{N\hat{(}v)} + c \cdot \sqrt{\frac{2\ln(N)}{N(v)}}$$

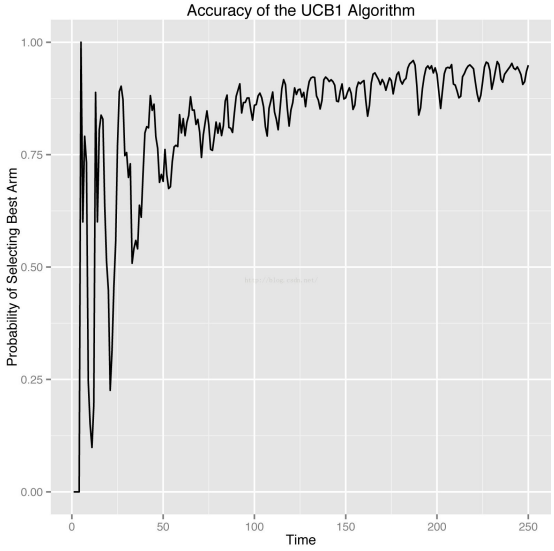
4 发现与感悟

4.1 胜率的动态变化

我观察了多局失败的对战情况，我发现，当 MCTS 给出的胜率 <40% 时，算法有很大的概率最终会失败。当给出胜率 >60% 时，算法有很大的概率会胜利。这一方面检验了 MCTS 的正确性，但另一方面也显示了，在很早的时候，算法便可以预测自己的最终结局。我查阅了一些文献，没有发现可以“提早反败为胜”的策略，但是这一发现引发了我的思考。既然能提前很多步发现自己有大概率失败，那么如果能找到“反败为胜”的方法，最终的败局就有可能被避免。

4.2 选择最优点的概率

经过我的统计发现，尽管随着模拟时间的增长，MCTS 选择最优落子点的概率会不断上升，但是这种上升存在很大的波动性，不是特别稳定。这意味着，MCTS 很可能不会选择最优落子点，进入某种“陷阱”。因此，减小这种波动性或许会成为进一步改进性能的突破点。



5 总结

本次实验我实践了 UCT 算法，对原始版本进行了时间和空间的大幅优化，并改进了 Default-Policy。可以发现，加入了一定启发信息的 UCT 算法表现更为优秀，但是这种启发信息也不宜过多，否则就丧失了“蒙特卡洛”的精髓。通过查阅文献，我也了解了 AlphaGo 的精妙之处，对我的算法提出了一些改进，最终取得了不错的效果。

感谢马老师和助教的悉心指导!