四子棋机器对弈

实验报告

韩 荣

计 71 2016010189

实验二 四子棋机器对弈

韩荣 2016010189 计 71 han-r16@mails.tsinghua.edu.cn 18811669931

Contents

1 问题描述

实现一个四子棋落子策略，能够根据当前棋盘状态，自动选择收益较大的落子点。在与机器动态库的对弈中， 尽可能多得赢得比赛，并进行模型的评价与改进。

2 文件说明

根据策略生成的 dylib：

libStrategy.dylib：

生成的动态库，作用为输入棋盘当前状态，可以返回一个落子点，是机器对弈算法的生成文件，在棋局开始时，将其加载，即可开始对弈。

工程文件包括：

Point.h：

定义 Point 类的头文件，Point 是一个二元组，相当于坐标系中的点。

Judge.h、Judge.cpp：

用于判断对局胜负，但在 AI 实现中，并没有调用此文件，而是调用自身类中成员函数 JudgeWin。

Strategy.h，Strategy.cpp:

策略函数，是比赛调用的接口，可以返回一个点作为下一步落子的策略。

AI.cpp,AI.h:

四子棋落子算法的实现，构造一个 AI 类，通过 GetInfo()函数获取棋盘信息，通过 GetAction()函数返回本步策略。

运行方法：

① 通过 UI 调用动态库：打开四子棋图形界面程序，选择 Computer-Computer 对战，将 libStrategy.dylib 作为 player B 载入棋局，另外将测试文件作为 player A 载入棋局，即可进行对弈测试。

② 通过 Compete 指令调用动态库：按照 Compete 可执行文件的调用方法，在命令行中进行对弈，本实验中，进行 A 先手 5 次，B 先手 5 次共 10 次测试，最后

会返回每盘棋局的结果和最终胜率。

3 模型原理

3.1 蒙特卡罗评估

由于四子棋的变化较多，采用估值函数的方法来评价局势会比较复杂，同时局面也不易用估值函数准确评估，故采用蒙特卡罗评估方法。蒙特卡罗评估方法为在有限时间内通过模拟落子不断搜索状态空间，加以一些启发性策略提高模拟效果。统计当前状况下的胜率，并选择胜率较高的点进行落子。

3.2 模蒙特卡罗树搜索（MCST）

蒙特卡罗树搜索是通过一个已知的初始状态，动态进行状态的探索，逐步拓展树的节点（即不断尝试新的状态），每拓展一个新的状态，便对该状态进行一次随机模拟，将获得的回报反向传播给到达此状态路径上的各个节点，是一种动态的搜索策略。各个节点储存有该节点可以拓展出的子状态、当前模拟次数和当前胜场数、以及棋盘各可落子点的位置等信息，用于后续的计算。其原理图如下。

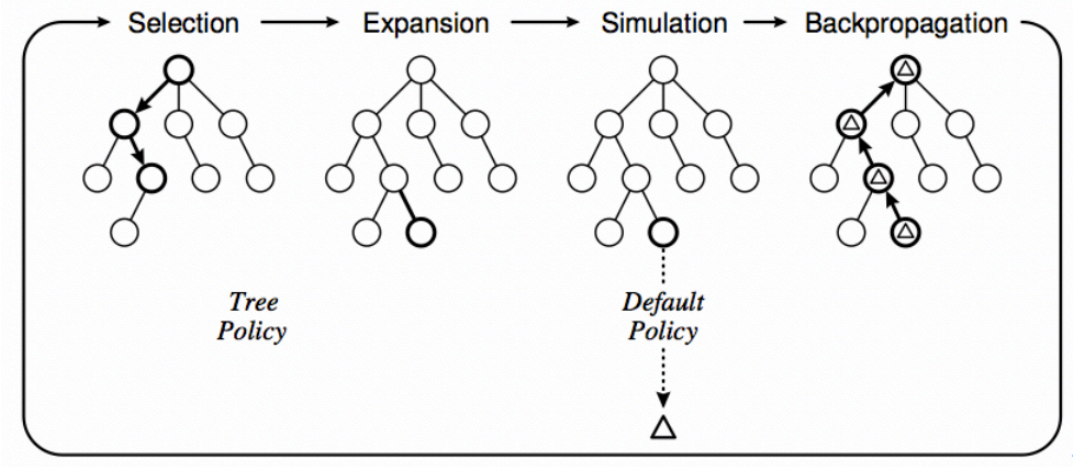

图 1：蒙特卡罗树搜索

由图可见，蒙特卡罗树搜索分为四个步骤：

选择：通过特定的选择标准，从根节点出发，不断选择最优秀的子节点，直到选到一个仍有拓展可能的节点为止。

拓展：对该节点进行一次拓展，获得一个新的子状态，选择该子状态。

模拟：将新的子状态进行模拟（模拟的过程中不会拓展出新的状态），让两方轮流随机落子，加以一些启发性策略，获得的结果中，胜利为 2，平局为 1，失败为 0。

回溯：将结果向父节点传播，更新父节点的胜率、场次等信息，至此完成一次树搜索。

3.3 UCB 算法和 UCT 算法

UCB 算法：多臂老虎机模型，在有限的时间和有限次的探索中，我们应该优先探索已知价值更高的节点，也要兼顾未被探索的节点以获得新的知识。因此，我们在选择节点时不应该只考虑当前已知节点的胜率，还应该适当探索被探索较少的节点，故 UCB 算法给出的信心上限如下：

$$I_j = \bar{X}_j + \text{Cons} * \sqrt{\frac{2\ln(n)}{T_j(n)}}$$

其中

$$P\left(\bar{X}_j \geq \mu_j + c_t\right) \leq t^{-4}$$
$$P\left(\bar{X}_j \leq \mu_j - c_t\right) \leq t^{-4}$$

即我们可以获得一个置信概率很高的结果，其中$\bar{X}_j$为当前的胜率，表示已搜索到的知识的价值，$\sqrt{\frac{2\ln(n)}{T_j(n)}}$表示未探索的节点的价值，n 为总局数，$T_j(n)$为该节点的模拟局数,Cons 是一个常数。

UCT 算法：即使用 UCB 算法的信心上限公式，进行蒙特卡罗树搜索，每次选择已有节点时，选取 UCB 公式计算值最大的节点。

3.4 针对 MCST 的一些优化

（1） 在选择过程中遇到必胜情况

如果一旦遇到了下该点则游戏结束的情况，那么便一定会选择该点，不管该点的父节点有无完全拓展，每次到此，都不会进行新的拓展和

模拟，直接返回结果。

（2） **在模拟过程中遇到必胜的情况**

基本的蒙特卡罗模拟方法为根据随机模拟的结果进行回溯，但仔细思考，发现这样的模拟不是特别优秀，比如在己方已连成三子的之时，若随机落子，则会错失良机，不符合常识；同样地，若对方已连成三个，此时己方不堵，则会拱手让出胜利，不符合常识。因此，在模拟过程中如遇到这两种情况，不应随机落子，而是一定选择争取胜利或者阻止对方胜利。这样可以一定程度上提高蒙特卡罗模拟的质量。

3.5 针对计算时间的一些优化

在计算过程中，发现有一些值是不必要每次都进行计算的，比如进行一次模拟，需要更新全局所有点的 UCB 计算值的后一项，这样做过于繁琐，因为真正需要被改变的其实是这一次搜索路径上的节点信息，因此对 UCB 计算式做一个修改，不将I_j储存在本地，将计算式拆解如下：

$$UCBa = \bar{X}_j = \frac{Nodewins}{T_j(n)}$$

$$UCBb = \sqrt{\frac{1}{T_j(n)}}$$

$$XL = Cons * \sqrt{\ln(n)}$$

可以看出，**UCBa，UCBb 均与总局数无关，可以储存在本地，而将 XL 储存为一个全局变量**，每进行一次搜索就更新一次。在**需要计算某个节点的 UCB 时，才进行计算**，这样可以避免每次对各个节点更新时复杂的根号和除法运算，计算式如下：

$$I_j = \bar{X}_j + \text{Cons} * \sqrt{\frac{2\ln(n)}{T_j(n)}} = UCBa + UCBb * XL$$

3.6 针对空间的一些优化

对于每个节点，如果要储存当前棋盘信息，则会浪费非常多的内存，在开

始读入棋盘信息时，同时为两个**全局棋盘** board 和 baseboard 赋值，然后每个节点仅记录每一列的可落子位置，然后真的在 board 上“下棋”，每一次搜索之后，**再将 board 复原成 baseboard 的状态**，开始一次新的搜索。这样可以大大减少节点所需要的内存，优化了空间。

4 算法测试

将算法生成的动态库与老师提供的 50 个难度不同的动态库进行对弈，通过测试结果，进行参数的调整，达到最优的胜率。

5 模型分析

5.1 MCST 的优势与不足

对于即时策略型的游戏，MCST 相对于$\alpha-\beta$剪枝算法，**不需要手动定义评估函数**，而是通过搜索状态空间来即时生成策略，这对于简化算法设计难度有很大的帮助，通过加入合理的模拟启发，MCST 生成的策略的质量十分高。

但是蒙特卡罗算法无法记忆自己的策略，因此在每一步下棋的时候，都会从节点开始进行重新搜索。不过这一点是可以适当改进的，即通过全局的树来记录各个节点的信息，一步操作之后这个树不会消失，只是将根节点转移为某一个子节点，这样便能记录下上一次模拟的信息，避免重复模拟。但是如果要通过全局的树记录已有模拟的结果，则又会大大提高内存的消耗，可谓有利必有弊，故在本次实验中，仍然保持每次重新搜索的搜索方式。

5.2 结果的记录规则

在记录结果时，每次对局，**节点场次会加 2，如果胜利，则胜利值加 2，平则加 1，负则不增加胜利值**。在计算每一个节点的 UCB 时，

如果当前节点的 player 是己方，则

$$UCBa = \bar{X}_j = \frac{Nodewins}{T_j(n)}$$

如果当前节点的 player 是敌方，则

$$UCBa = \bar{X}_j = \frac{T_j(n) - Nodewins}{T_j(n)}$$

这样一来，在选择环节，双方都会选择各自的最优子节点进行博弈，获得较为可靠的选择结果。

5.3 UCT 信心计算常数的选择

根据判断，在算法采用的计算规则中（5.2 所述），相当于将场次和胜场数同时乘 2，这会导致 UCBb 偏小（即分母乘 2，分子乘 2 后取对数），那么在选择常数时应该稍微考虑大一些的常数。

常数 Cons 的选择需要通过实验获得，在本实验中，针对 90～100 的 dll 进行实验，反复调整 Cons 的值，发现 Cons 的值在 0.7 左右比较合适。

于是选取$\sqrt{2}/2$（即 0.707）作为常数值。

5.4 MCST 的实现

在实现 MCST 的过程中，采用了模块化的思想，将选择、拓展、模拟、回溯分成了三个不同的函数，并对不同的选择结果判断，以下是关键代码及说明：

```
std::vector<Node*> path;
Node *pos = Selection(root,path);//选择一个点并记下路径
int result;
if((pos && pos->winner == 0) || !pos)//如果选到了一个非终止节点
    result = Simulation(pos); //进行模拟
else
    result = 2 * int(pos->winner == My_player);//否则直接更新
if(pos)
    CalculateParameters(pos);//计算该节点的UCBa，UCBb值
Backpropagation(path,result);//沿着路径进行回溯更新
```

6 实验结果

6.1 与各 dylib 对战的结果（共对局 10 个回合，各先手 5 次）

dylib 编号	2	4	6	8	10
胜率	1	1	1	1	1

dylib 编号	12	14	16	18	20
胜率	1	1	1	1	1

dylib 编号	22	24	26	28	30
胜率	1	1	1	1	1

dylib 编号	32	34	36	38	40
胜率	1	1	1	1	1

dylib 编号	42	44	46	48	50
胜率	1	1	1	1	1

dylib 编号	52	54	56	58	60
胜率	1	1	0.9	1	1

dylib 编号	62	64	66	68	70
胜率	1	0.9	1	0.8	1

dylib 编号	72	74	76	78	80
胜率	0.9	0.9	1	0.8	0.8

dylib 编号	82	84	86	88	90
胜率	0.9	1	1	1	0.9

dylib 编号	92	94	96	98	100
胜率	0.9	0.8	0.7	0.8	0.9

6.2 胜率分析及人机对战

可以看见，在与老师给出的 50 个 dylib 对战的结果显示，本算法的策略比较优秀，这激发了我测试 50 个 dylib 强度的兴趣，和同学一起与 dylib 中的 100 进行人机对战，胜率为（1/1），再与自己写的策略生成的动态库进行对战，胜率为（1/4），可见本算法的策略还是存在一些局限性，虽然强度比 100.dylib 大，但在面对深思熟虑的人时，还是会出现错误。远远没有达到“完胜人类”的效果，因此说明本算法还存在明显的改进空间。

6.3 可能的改进方向

上面提到，与人对战时，无论是 100.dylib 还是 libStrategy.lib，均会出现失手的情况，因此模型还应该进行进一步的改进使之更加智能。

① 通过与深思熟虑的人对弈，调节参数 Cons

这是目前想到的最直接也最简单的提高算法强度的方式，但具体的效果未知，因为人的棋力不像 100.dylib 一样固定，因此在与人对战的调试中，“胜率”往往并不是一个准确的参考，这会使得参数调节变得困难。

② 在模拟环节，加入更多的启发信息，如“判断两步后落子”——即简略版的蒙特卡罗树搜索，这样会在每次模拟时提高模拟的质量，但缺点是会增加模拟的时间。

③ 通过神经网络来提高算法的表现，这一点已经在 AlphaGo 与 AlphaZero 中得到了证明，MCST+NerualNetwork 是一个不错的提高强度的算法。

7 收获与总结

7.1 关于本次实验的一些思考

（1）在蒙特卡罗搜索树模拟的过程中加入启发信息的作用？

如在本实验中，通过加入必胜情况时落子的启发，会较好地提高算法的能力，（**原来打 100 胜率约为 30%，改进后为 90%**），这是因为这种模拟的下法更符合常理，因此模拟的效果更好。这也启发我们，在模拟过程中适当加入启发信息，会对整个算法的提升有很大的帮助。但这种启发信息也不应该过多，否则会降低算法速度，同时也减少了模拟的次数，不利于状态空间的探索。

（2）算法能力提升主要的限制在哪里?

主要有两点：一是计算资源，二是计算时间。在有限时间内，不能便历所有状态空间，因此无法获得所有信息。需要将有限的计算资源利用到有价值的状态空间，这样才能在有限的时间内进行最充分的学习，做出最优的选择。

（3）本实验生成的 libStrategy.dylib 真的比 100.dylib 优秀吗？

从比赛结果和人工测试来看，是这样；但从算法策略和消耗时间来看，不是这样。100.dll 能够在几乎没有延迟的情况下做出响应，而本算法需要“思考时间”，约 3s，差距十分明显。可以说 100.dylib 所采用的算法效率远高于本算法，这一点十分值得注意，作为即时策略游戏，反应的时间也应该纳入优化的范围。但目前为止，还没有思考出好的结果，希望能够学习 100.dylib 的迅速反应算法。

7.2 实验收获

在四子棋人机对弈的实验中，我学习了 MCST 和 UCT 的思想和方法，体会到了蒙特卡罗方法在即时性策略游戏的优秀表现。同时在思考提升算法性能的过程中，不断地体会到了方法中选择、拓展、模拟、回溯的各自作用，并进行了针对性的优化。

最后，我将搜索时间设定为 1.8s，这是因为我希望在保持较高正确率的同时，也应该适当兼顾反应的时间，这样的算法才是能够有实际应用价值的算法。

通过本次实验，我学会了机器学习的一种方法，发现著名的 AlphaGo 也是运用了这样的算法，在感叹本算法设计的精妙的同时，也加入了自己的理解使之更佳智能，很有成就感。同时，对于人工智能的理解也更加深刻，在对比与总结过程中，也进行了更多思考，比如对模型的评价方法（本实验中有老师给的 50 个.dylib，但是在科学研究中只能自己构造测试例子）的选择是模型成功的重要因素，也能够因此调节出更适合模型的参数。

7.3 实验小结

本次实验是我完成的第二个“人工智能”的程序，在实验过程中，从对算法随机落子的不信任到适当改进模拟算法，提高了算法的强度，也优化了算法所需要的计算资源，同时 C++代码的实现让我自己动手搭建了一个 UCT 算法，加深了对实验的理解。在测试的过程中，我思考了模型评价、性能优化的方法，找到了自己的模型的不足，并思考了一些进一步改进的方向。