四子棋实验报告

计算机科学与技术 75 班 赵成钢 2017011362

2019 年 5 月

1 基础算法

本次实验我直接采用了信心上限树算法（UCT）为基础框架，UCT 是信心上限算法（UCB1）结合到 Monte Carlo Methods 上的结合算法。

和 Monte Carlo Methods 基础做法是相同的，每个节点代表一个局面的状态，子节点代表通过落子来转移到的下一局面，而不同的是在评价节点时，UCT 并不单纯考虑获胜的概率，而同时考虑信心上限的估计，每个节点综合的得分通过式子给出：$Score(v) = \frac{Q(v)}{V(v)} + c\sqrt{\frac{2\ln(N(v'))}{N(v)}}$，其中的 v' 是 v 的父节点，$Q(v)$ 代表点 v 的为根的子树的所有终结局面的收益的和，$N(v)$ 是节点被访问的次数，这里规定对手赢收益为 -1，平局为 0，胜利为 1。而更重要的是式子的后面一部分，系数 c 为可调参数是整体搜索的策略参数，而后面的部分我们发现子节点作为分母，可以定性的理解为在搜索起步时，每个节点不会因为自己的纯获胜的收益小而被放弃选择，而会因为 $N(v)$ 较小被给予尝试，而随着节点数被尝试的次数越来越多，分母变大，收益就决定了搜索的方向，可谓是平衡了收益和探索。

而对应到具体实现，每次搜索一次，UCT 先选择一个可扩展节点对子局面进行扩展；如果没有可以扩展的就沿着每个节点得分最大的路径走下去。如果走到的节点不是终结态，则通过默认的策略计算这个点的收益，最后得到收益后反馈给所有的祖先节点。为了加速搜索的过程，我并没有每个节点都存储对应的棋盘，而是顺着单次搜索搜索到最后的终结态，最后结束单次搜索后再恢复到开始的状态。

2 优化和尝试

在写完基础算法后，我发现对战 100.dylib 的胜率已经高达 85%，可见 UCT 算法的强大和针对任意场景的鲁棒性。不过，针对四子棋，我还是做了一些有意思的尝试。

2.1 必胜、必堵

最开始时采取默认策略的目的是为了对一个节点有一个正确的估计，而基础算法的估计基于随机，即双方随机走，而为了让估计更加准确，我们需要对局面有初步的认识。最简单的即是，如果我差一个子就胜利了，那我肯定下在那里；如果对手只差一个子就胜利了，我肯定下在那里堵住对手，这样我们就为默认策略减少了随机性，增加了其对局面估计的准确性。其实我还尝试过，随机策略是每次走能走的最中间的棋子，这样根本没有随机性，对局面的估计反而有了很大的偏见，我们需要在这两种极端的情况下做出平衡。

不过这种做法需要大量的判断，也浪费了大量的时间，最后综合考虑我还是没有在代码中体现，但是如果不失为一种好的做法，没有采用的原因就是 UCT 本身也包含了对这种情况的考虑，在搜索次数和估计准确性上我选择了前者，这样直接导致搜索次数多了约 10%。

2.2 自我博弈

这是我尝试过的最有意思的优化，即直接 load 一个其他的策略文件，预测在状态树的第二层的落子，然后在计算的时候只搜那个分支，这样可以减少大部分的分支。我尝试了 load 自己先前的基础版本策略，以为一定可以下过原来的版本，可是结果非常不尽人意，UCT 产生了非常强的偏见性，几乎全部输掉了。

（忽略本段）由此甚至可以产生一个危险的想法：直接把 90 到 100 全部 load 进来和对手下，然后每个策略全部跑一遍，最后投票走位，于是就可以不写 UCT 了。

2.3 评价优先选择中间节点

在选择得分最高的节点时，我最开始的写法无非就是设一个最大值然后如果这个节点得分大于最大值就把最好的节点改成这个值，而实际上在搜索初期的时候，很多情况下有很多最大值的得分，我们需要没有偏见地选择一个节点才能保证探索足够充分，于是我改成了在最大值相同的里面随机选择，可是胜率瞬间掉到了 80%，我发现了这样做搜索次数变少了，因为搜索更加宽泛了。根据经验的，我在所有的最大值中选择了最靠近中线的一个，这样做了之后，对战 100 的胜率直接稳定到了 95%。一个可能的解释就是中线的地方，有更多的可扩展的空间，连成 4 个子的几率更大。

2.4 结果继承

因为本次的测试为一个完整的游戏，而非是简简单单的一步。那么在一次游戏中，我们可以复用上次的搜索树结果，因为落子和对手的落子为对应的树的某一个跟深度为 2 的子树。我输出了复用率，大概为 1% 到 90% 还是在有限的时间内提升了搜索次数的，测试的结果是基于上面的优化胜率稳定到 97.5%，即 40 轮输一次，效果不会差。

2.5 常数优化

进行了必要的内联化、内存对齐、采用单精度浮点数等常数上的优化，搜索次数多了 50% 左右，达到了每步平均 40 万次左右。

2.6 一个误区

最后真正判断操作时，注意要把系数 c 置为 0，我的理解是探索是在搜索过程中体现的，在最后的决定时刻还是要以胜率为依据，而且这个 $\frac{Q(v)}{N(v)}$ 在正则化之后可以直接作为胜率，以便我们观察局势的走向，看着这个数字一点点变化真的是胆战心惊。

3 实验结果

3.1 测试环境

为了可以争取加载 32 位 i386 的 dylib，我降级了 Xcode Command Line Tools 使得支持 32 位程序的编译和运行，同时我认为这样一个小程序用 Xcode 过于繁重，于是写了一个 Makefile，可以直接在 Stragedy 目录下 make，不过请先确保 g++ 是 Apple Clang 版本的而且支持 i386 编译。

下面是运行环境：

1. macOS Mojave 10.14.4
2. Intel Core i7, 2.8 GHz, 16 GB DDR3
3. Xcode Command Line Tools 9.4 (i386 supported)

3.2 对战测试

这次提供了对战的平台，于是我不仅和提供的 dylib 进行了对战，我还和其他同学写的可以 90% 几率战胜 100 的进行了对战，还是很有趣的。

最后，我把时间限制在 2.5 秒，系数取了 0.8，进行了测试。

下面是和提供的进行对抗的结果，其中因为前面的胜率较高可能合并为一项，和每个对战了 40 轮，前面为了节省时间把搜索时间设置为 0.01 秒也几乎是全胜，后面出现失误的时候逐渐提高到 2.7 秒，不过我调试的时候都是和 100 打，最后发现好像 100 不是最强的。

2-48	50-80	82	84	86	88
100%	95.9%	95%	80%	90%	100%
90	92	94	96	98	100
97.5%	97.5%	67.5%	72.5%	72.5%	97.5%

后面，我又找了一位同学（计 71 班刘欣）和他的进行对抗，发现不论是哪个优化后的版本，两人对抗基本上都是五五开的胜率，下的非常胶着，每次都几乎把棋盘下面，还出现了从来没出现过的平局情况，而且我发现了基本上都是先手必胜，如果是先手，胜率在一开始就维持在 60%，后面大概率都在增长，很少会输掉，可见应该存在大概率胜利的某种固定走法，搜索算法也尽可能地探测到了这种走法，非常奇妙。

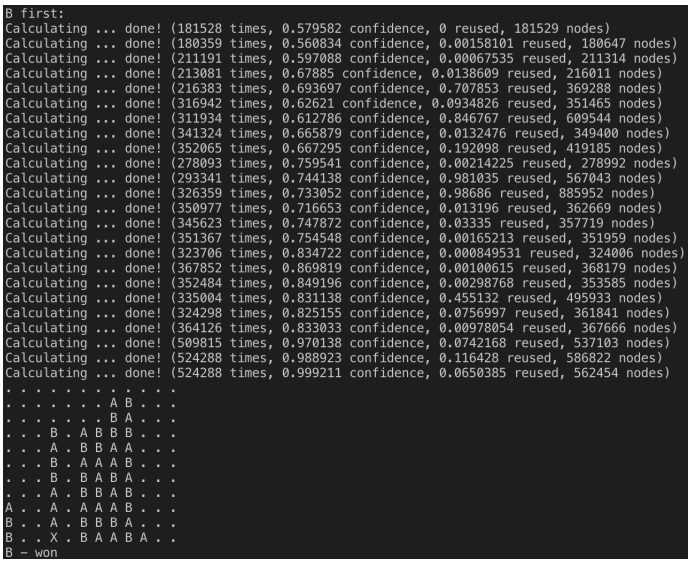

图 1: 测试截图（显示了搜索次数、胜率、节点复用率）

如图 3.2为程序下一盘完整的棋的运行过程。

4 总结

本次我实现了完整的 UCT 算法，并且针对四子棋的问题做了充分的尝试。认识到了算法的美妙。也了解到之前的 Alpha Go 就是基于 Monte Carlo Methods 来做的，另外最大的区别就是它还用神经网络评估局面来估计走子，随后自己和自己对弈还可以更新神经网络实现自我学习。做完本次作业之后，认识到高级的算法距离我们并不遥远，都是基于基础算法的结合和衍生，进一步形成更强大的通用算法来为社会解决实际问题，这个过程非常神奇和美妙。

感谢老师的讲解和助教在微信群中的答疑。