人工智能课题：围棋AI实现报告

F1103028 (5110309046) 王凯南

2014 年 1 月 8 日

1 基本思路

对于棋类游戏最为基本的方法就是min-max搜索，并应用各种合理的方式去剪枝，但此类方法在围棋上却遇到了巨大的困难，其主要原因就是围棋的状态总量过于巨大，目标函数难以确定等等。基于这种现状难和查阅相关资料，我们组采用的方法是基于随机的搜索方法。

这里的随机，主要是指评估状态函数的方式。当我们搜索到一个节点时，该节点必然对应着一个棋盘状态，我们在该状态下，使用随机的策略（只要是合法的走法都有等概率被选择），模拟很多盘，通过胜率来评判这个状态的好坏。

但这样的随机会带来一个问题: 为了提高判断的准确性，我需要模拟很多盘，但这样消耗的时间过多（我们的优化后程序1秒也只能跑10^4盘，但可以生成的节点远不止这么多），反而有些得不偿失。为了解决这个问题，我们采用了一种被称为UCT的方法。在这个方法下，我对于每一个节点，维护的不再是在这个节点上模拟的胜率，而是在这个节点以及其子节点下模拟的胜率。这样，我们对于每个节点仅需要模拟一次，并在对其子节点的模拟之后更新其胜率。后面我们会发现，这样的设定其实更加准确。

2 主题框架以及改善空间

我们给出的伪代码中（具体请见下一页），有一些函数的实现相对固定，例如判断棋局是否结束，裁决结束的棋局的结果等等。但有些函数的实现则大大影响了程序的效率和效果。在这当中，最为重要的莫过于$SelectMove$函数$PlayRandomlyUntilGameOver$函数。在理想的模型中$SelectMove$ 选择的是胜率最大的走法，$PlayRandomlyUntilGameOver$则是按照最优策略进行模拟。而在我们的基本模型中，$SelectMove$选择的是模拟对局胜率最高的节点，$PlayRandomlyUntilGameOver$则是按照随机策略进行。

当模拟次数趋于无穷的时候，基本模型会向着理想模型收敛。因此，在这个基本模型下，我们的目标是在限定的时间内尽可能的接近于理想的模型。

3 模拟对局的策略优化

首先，对于模拟策略而言，我们有三个主要的评判标准。

- 合理性，这样的对局更可能出现在真实的对局中
- 多样性，任何真实的的对局都可能被模拟出来
- 高效性，可以在时间限制内模拟更多的局数

Algorithm 1 Upper Confidence bounds applied to Trees Algorithm 基本框架

Require: $initialBoard$:表示初始棋盘的状态, s_0:表示由此状态导出的搜索树的根节点

Ensure: 下一步的位置

function UctSearch($initialBoard, s_0$)
  while time avalible **do**
    $t \leftarrow 0$
    $board \leftarrow initialBoard$
    while $board.GameNotOver()$ **do**
      if s_t not visited before **then**
        CreatChildrenNodes($s_t, board$)
        $board.PlayRandomlyUntilGameOver()$
      else
        a = SelectMove($s_t, board$)
        $board.PlayMove(a)$
        $s_{t+1} \leftarrow s_t.ChildAt(a)$
        $t \leftarrow t + 1$
      end if
    end while
    $result \leftarrow board.BlackWins()$
    for $i = 0 \rightarrow t$ **do**
      Update($s_i, result$)
    end for
  end while
  return SelectMove($s_0, board$)
end function

在这三条标准中，高效性是比较易于判断的，但其它的标准则很难量化的去把握。再考虑到这三条标准之间冲突性，如何获取一个合理的策略无疑是非常困难的，我们参考了一些论文的方法，在经过测试之后，采取了一种简单高效的“优先级策略”，也就是说，我们定义一些具有优先级的简单策略，如果某一个走法满足当前策略，就执行，否则检查下一个策略。这种方式的好处在于，我们无需考虑策略之间的平衡并且可以在该走法满足某一个策略之后就退出，这大大提高了模拟的速度。当前，我们选择的简单策略按照优先级从高到底分别是

- Atari Defence Move
- Nakade Move
- Fill Board Move
- Pattern Move
- Capture Move
- Random Move
- Pass Move

其中，pattern是由巢子琛实现的，其余部分则是由陈蕾宇实现的，具体的解释可以参阅他们的报告。

4 模拟速度的优化

我们本身的棋盘模型是在网上的brown程序基础上完成的，在10秒内可以完成6000局模拟。而在我门添加策略之后，这个数字则下降到了3000左右。这显然并不能满足我们的要求，我们反思了自己的程序，发现每当我们需要用到“气”相关的内容时，都需要遍历整个一串棋子才能做到。这无疑是非常的浪费时间的，因此，我希望能做到在平均常数的时间内完成对气的维护。

首先，我们不应该对每个有子的位置都维护气，因为每下一步，都会有若干串的棋子同时发生气的改变。因此，我们应该对每一个串维护一个气的值，并且维护一个从位置到串的映射。对于每个有棋的位置，我维护了它对应的串的编号，对于每个串，我维护了它的颜色，气，将这串棋子维护成一个循环链表并记录首尾。

当我落子的时候，如果没有发生提子或者连接的情况，我只需要把周围的位置对应的串的气进行更新即可。如果发生了提子，我们就检查被提走的串，更新和它相连的串的气。我们发现，这里的时间和串的大小是相关的，似乎不再是常数了，但我们又发现，提子越多，棋局的步数就越多，平摊的看，维护提子时的气对于每步而言，仍然是常数的。对于连接（就是把同色的子连在一起）而言，我们只需要把两个链表头尾相接就好了。但问题在于，每个位置对于串的编号必须要统一。一个理论上比较优的方法是采用并查集这种数据结构进行编号的维护，但考虑到常数，我们采用了一种更为简单有效的方式，就是直接把较小的串的位置的编号改成较大的串的编号。不难发现，这个方法的复杂度是log 的，但考虑到串的长度和实际的常数，这样做的效果更好。

我们遇到了另外一个问题是一个空位对于同一个串的两个子只能提供一个气，这个看似简单，实际上则非常困难。因为整个程序最慢的地方是维护连接时的气。被连接的两个串可能有一段公共的气。而我们又只能遍历小的那个串，所以我们需要先扫一下小的串周围有没有空位，再看这个空位的周围有没有大的那个串。常数巨大到可想而知。但如果我们维护一个二维的布尔数组来判断一个空位是否为一个串提供气，这无疑需要一个平方级别的空间和初始化时间。我们可以采用位运算的方法，用一个int表示32个布尔值。这样，

虽然我们是平方的时间，但在除以32之后，其实际效果也等同于线性（也就是平摊到每一步的常数）了。

至于打劫的维护，棋盘的初始化和复制，终局的判断等各种各样的问题，虽然十分繁琐，并数次出错，但终究算不上大的挑战。整个程序大概持续了将近一周的时间，经过大量的测试之后才投入使用。经测试，纯随机模拟的速度提高到了10秒25000局。我又想出了一种优化选择的策略。之前的随机方式是找出所有可行解，然后随机选择一步，但这样需要扫描整个棋盘，我的改进是先随机选一个位置，如果合法就选择，如果20次之后仍然找不着合法解，就扫描整个棋盘。这么做的好处在于，在游戏的大部分时间里，你都可以很快的找到一个合法的位置，并且没有影响到随机性。在优化之后，模拟的速度又一次提高了4倍!

还有一个问题是如何找到所有的capture move，也就是提子。和随机不同，一个棋盘可以提子的位置并不多，我随机找20处想找到一个可以提子的位置并不现实。但扫描整个棋盘确实又太慢了。（当时降到了10秒17000 盘左右）。最后，我们用一个队列维护了所有处于叫吃状态的串，一旦一个串气降至1就加入队列。然后需要检查capture时把那些气不再是1的串去掉。因为我认为在平均情况下，这样的串并不多，因此复杂度可以说非常的小。

在完成各种优化之后，程序的瓶颈落在pattern move 上，因为这部分不是我实现的，并且据说需要相对底层的优化，所以我并没有对其进行改动。放弃对pattern move改动的另一个原因是我发现模拟之外的程序的时间已经将模拟的极限限制在了10秒20000次左右，因此后面的优化可能性价比并不够高。

5 节点选择优化

5.1 RAVE方法

RAVE是rapid action value estimation的简写，又称之为AMAF（All move as first）方法。我们对一个棋盘的一种走法，不仅仅看这一步下这个位置的价值，还要看在将来下这一步的价值（如果可以下的话）。更形式化的说，对于一个节点s和一个走法a，令Q(s, a)表示从节点s经过走法a之后的状态，那么Q(s, a)这个状态的RAVE值（AMAF的评估）就是在状态s及其子树上所有模拟对局中的那些在某一步选择走法a的对局的胜率。我们很容易发现，这样的评价是有问题的，因为一步的价值在若干步之后很可能就失去了，但RAVE值得巨大优势就是其信息量庞大，对于一个节点s，只有其子节点才能更新其胜率，但其所有父节点的子节点都可以为它提供RAVE值得更新。这使得我们还没有访问一个节点或访问次数较少时，很可能已经得到了其相当准确的RAVE值，通过这个rave值来进行选择，可以大大优化了我们选择的策略。当然了，如何处理胜率和RAVE值（AMAF胜率）的平衡，则是一个令人头疼的问题。总的来说AMAF在总体来说更加准确（因为更新的次数多），但在一些关键走法上没有原则（所谓的关键走法，就是指不这么走局势就会大变，之后再走就没用了）。不论偏向哪个都会有问题，所以我想出了一种动态参数的方法。也就是说，在程序搜索的过程中，随机的选择对搜索策略的偏向。这样就不会因为过分注重某一方面而使得程序有明显的漏洞。这个方法在很早之前就完成了，但因为参数问题效果一般（过于偏向胜率），程序本身虽然有很多细节需要注意，但想法上除了动态系数之外都是按照论文上写的，因此就不细述了。

5.2 加入领域知识

这里的领域知识，也就是对我们下的某一步进行一个粗略的认识。我们需要考虑很多内容，例如和边界的距离，和上一步棋的距离，是否符合某个pattern，是否具有atari或者atari defence的性质。（其中pattern的识别是巢子琛完成的）等等。因为这样的判断对一个节点只有一次（而对于模拟程序来说一个节点要模拟整整一盘），因而我们可以处理的更加复杂。因为这里求得的值H仅仅是对与节点选择的开始阶段有一定的影响，随着胜率和RAVE值更新次数的的增加，H的影响会逐渐减弱。另外一种估值则影响一直

相当大，比方说涉及到长串atari，atari defence和capture。这些走法可能改变整个局势，是RAVE所不能控制的，是搜索可能忽视的，因此要给予相当稳定的权重。对于具体的权重，我的把握比较随意，因为这些值最终都会被RAVE和胜率覆盖，我给出的主要信息是相对的大小。但实践中我发现atari defence的重要性不言而喻，这和模拟程序中的的策略是一致的。

5.3 剪枝优化

我对于剪枝一开始还是比较乐观的，但越到后来越保守，因为不论按照什么条件剪纸，总会有剪错的时候。但我们的程序中还是加入了一些保守剪枝（这部分是徐涵完成的），其主体思想是考虑每一个位置终局的占有情况和胜率之间的关系。

6 我的感想

应该说，这次的课题是自从我进入大学以来最为有趣的一次，我也为此付出了很大的精力。围棋的AI和之前的程序最大的不同在于你没有办法高效的调试，程序的结果并不是单纯的对和错的区别，而是好与坏的区别。而这个好与坏，可能也没有办法简单的判断（我并不能下过我写的程序）。比方说，我程序的白棋似乎能稳定的战胜黑棋，直到现在我还不知道是什么问题（但明天大概能处理好）。即使不算gtp相关的接口代码，我们最终版的程序有效代码有将近2000行，如果算上被废弃的版本，这个数字将达到3000。将这么长的代码中的bug一点一点的找出来，既是一种折磨，也是一种乐趣。

关于这个AI的故事实在是太多了，但我实在是写不动了，所以我着重写了一些原创性比较强的地方，而那些从论文中学习的方法，则仅仅是大概的描述一下。在这里，我更想总结一下在这个课题中收获的经验。

第一个经验是关于优化效率上的，我们在处理效率问题上的原则是“专注于瓶颈”，模拟慢，就去优化模拟，模拟里面算气慢就去优化算气，capture move的判断慢就去优化capture move，最后以至于要优化pattern move。但我后来想想，这样做也许有一些过了，也许算气并不需要常数，因为策略的判断比算气的维护慢很多，但执行的次数却是一样的。我们其实也许也没有必要（事实上我们也没有）优化pattern move，因为搜索的时间其实更慢。因此，优化也应该掌握一个度，在有限的时间内，“专注于瓶颈”在分析问题时无可厚非，在实现代码是则更应该注重平衡和统筹。就结果来说，我们的模拟10秒运行的盘数从3000增至15000，但那个程序本身是可以模拟10^5 盘不止的（有的组确实为了利用这巨大的运行能力放弃了策略），现在看来显得有些大材小用了。

第二个经验来自于信息的获取上，我觉得论文的确是最好的材料，尤其是那些有程序背景的论文。我看的论文主要是和MoGo相关，MoGo 是一个非开源的围棋AI，UCT和RAVE的早期论文都和这个程序相关。但我在这个过程中也犯了一些错误，就是看到一两篇就开始码代码，而我的程序又不是非常容易维护，等我发现现有的程序不够好，看到更好的论文以后，发现很难加进去这些东西，然后就只能重写了。重写带来的另一个问题就是分配任务上出现了混乱，很多组员写过的代码都被推掉了，现在的版本一半以上代码都是我完成的，但这并不是什么值得夸赞的事。总结一下就是，要么做好充分的准备，要么让自己的程序便于维护。

关于分工，每个组员都写了自己相关的部分，我也在我的报告中提到了一些（没提到的可能已经被替换了）。但总的来说，大家都非常的努力。

最后，感谢张老师一个学期的辛勤付出，感谢助教对这次课题组织和对我们组的帮助!