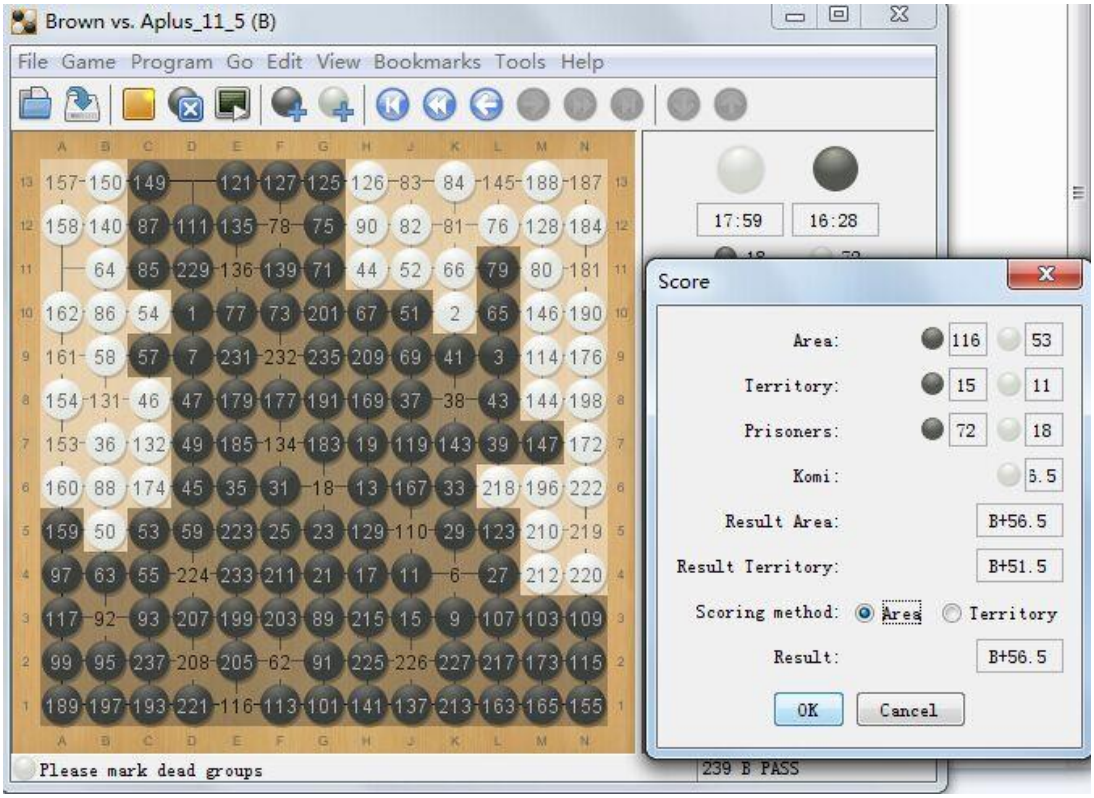

[1]

人工智能课程报告

GiveMeAplus: 陈悦莹 刘静静 仇馨婷 张哲慧

2015.1.17

[1] 封面图片为 Aplus11_5 与上届 undecidable【第一】的对战

目录

目录

前言

本次报告中，我们会详细阐述我们的技术路线，以及整个实现过程。

技术路线这一部分大体分为三节，第一节介绍棋谱定式的使用，棋谱定式可以帮助我方棋子在开局占据优势，就基本思路和具体实现方法均有详细说明。第二节介绍一些基本的优化方法，一个是置换表，通过哈希思想减少查找复杂度，另一个是并查集和气，通过利用并查集，为每一个棋串维护一个气，从而减少遍历棋串的代价。第三节介绍了 UCT，这也是我们技术的核心，通过介绍 UCT 的基本思想，节点排序，策略优化，UCT-RAVE 四个方面介绍我们的 UCT 的思想与方法，其中一部分思想是通过查阅相关文献得到的，有一部分是在具体实践中总结出来的，在文中均有详细阐述

项目历程这一部分介绍了本课程项目的整体开发历程，根据时间顺序介绍了阶段 1，阶段 2 和阶段 3，介绍了我们的项目如何从无到有，不断完善优化，最后得到一个表现稳定的版本，我们的工作大致分为功能方法实现，优化方法加速，debug。不同阶段也是根据这些工作的比例各有不同。阶段一侧重于功能实现，阶段二则是在相对完整的项目上进行完善与优化，寻找问题，排除错误，阶段三则针对最后的比赛进行高强度的优化，找出提高棋力的最优办法。

总体来说，主要思想仍是使用 uct，在这之上，我们又做了很多改进。比如剪枝，排序，策略优化等，这些改进让我们的 UCT 更高效，更聪明，除此以外我们还在棋盘上做了一些优化，增加了棋盘的速度，从而降低时间复杂度，为蒙特卡洛模拟留下了足够的时间。

组员分工

姓名	分工
仇馨婷	• UCT 框架 • 剪枝 • 并查集的实现 • 气的处理
陈悦莹	• UCT 框架 • 策略优化 • 多线程实现
刘静静	• 棋谱定式 • 置换表的实现 • 策略优化 • 节点排序
张哲慧	• pattern 的实现 • UCT-Rave 的实现 • UCT 测试与公式优化

技术路线

棋谱定式

基本思路

围棋定式是经过棋手们长久以来的经验累积，形成在某些情况下双方都会依循的稳妥的下法。我们的程序试图通过棋谱文件的支持，在开局阶段仿真棋手下棋思维，根据不同局面下出相应的应对策略。通过进入程序的时候读取棋谱文件，我们为自己保存了一个棋谱的数据结构，在每个局面要下子的时候，匹配棋谱树看有没有相同的局面，如果有就读取这个局面节点所连接的合理下法的节点。

实现过程

1. “学习”：读取并保存 SGF 棋谱文件上的信息

1.1 SGF 文件

SGF 是为了存储双人棋类对局记录而设计的一种纯文本文件格式，由节点组成并构造成对局树，可以存储对局记录(一系列着子)和实战的变化图。

1.2 SGF 文件处理

我将棋谱用在了开局前期的定式上，找到的是 kogo 定式大全，为了使程序快速读取棋谱文件，我用 Python 处理字符串，去除了评论、注释等无关项。

1.3 棋谱的数据结构

如下图所示，散列表即索引表是为了根据待匹配局面棋子总数，快速找到相应棋子总数的格局，表头与第一个相应棋子总数的格局相连，所有的格局以链表的形式连接起来。格局节点结构含有的数据项为：当前棋盘状态的位数组记录、下一步轮到的颜色（不必要，因为黑白可以调换）、当前棋盘上棋子总数、这个局面在棋谱中出现的次数，儿子为当前局面的其中一个合理的走法，兄弟为与该格局棋子总数相同的另一个格局。格局之间相连的边作为一类节点，彼此之间也是使用儿子兄弟链表的形式保存，儿子指向这一落子方式能到达的格局，兄弟指向他的父节点能下的另一个位置，含有的数据项为：落子位置、下一格局的指针，兄弟节点的指针。两个边节点可能会指向同一个格局节点，表示不同的落子顺序能够到达同一格局。

技术路线

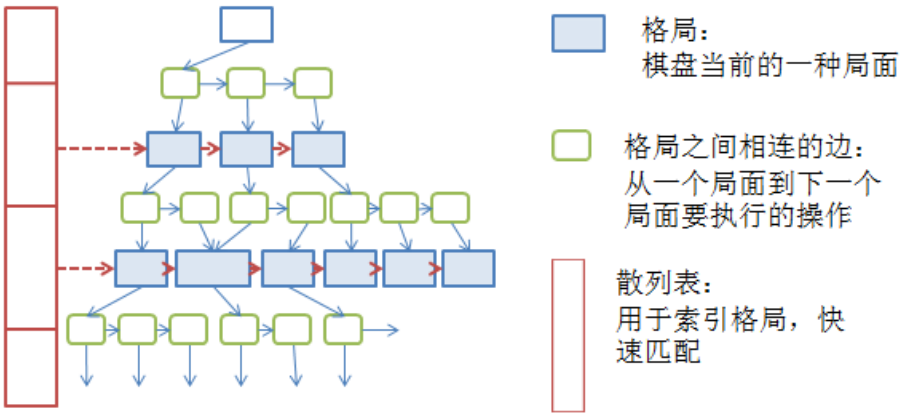

图 1. 棋谱示意图

1.4 SGF 文件的读取与存储

利用栈结构来对分支进行相应的读取，栈保存的是格局节点，基本思路如下：

循环读取字符：

a. 当遇到“(”时，向栈 push 一个空指针；
b. 当遇到“)”时，一个一个出栈，直到遇到一个空指针，将这个空指针出栈，并将当前局面指向栈顶；
c. 当遇到“:”时，再读取一个字符，如果是“B”，说明黑子要落子，继续读取落子位置，然后新建一个格局之间相连的边节点保存落子位置，当前格局落子之后的格局如果能通过索引表发现已经存在，就直接指向它，否则要创建一个新的格局节点，最后调整所有节点之间指针的指向即可，如果是“W”同理；
d. 遇到其它字符可以忽略。

2. “实战”：利用现有信息，匹配局面并选择下一步优秀的落子位置

虽然现有的棋谱文件都是对于 19*19 的棋盘，但是定式文件基本就是在一个角落里下子，所以我们只需要使得一块区域与定式匹配即可。

匹配的方法是，循环选取棋盘上 7*7 大小的盘面，首先数这个 7*7 区域里有多少颗子，然后使用索引表定位到这个棋子总数的所有格局节点，将旋转折叠得到的八个状态分别与每一个格局进行比较，如果有符合的格局，就将这一格局能落子的位置（即他的儿子节点以及儿子节点的兄弟）挑出来。

由于格局的复杂多样，即使是循环使用已经旋转折叠过的不同状态，还是很有可能出现匹配不到格局的情况，对此我做出了一些调整进行让步：一是如果这一棋子总数没有相对应的格局，那就到棋子总数更少的格局里面去搜索，当然不会少于实际总数的一半；二是放松匹配，只要对于棋谱格局上每一个位置的状态（假设是有一颗黑子），实际中的区域必须在这个位置也要有同样的状态（假设也是一颗黑子），那么便判为匹配；三是颜色对调再次匹配。

最后因为有许多候选下法，所以我不是直接输出位置，而是把这些下法加入到一个候选列表里，根据下一格局在棋谱中出现的次数来给出相应的分数，最后将分值排好序传给 uct 统一处理，让模拟来选出最好的下法。

2.1 测试结果

我让程序直接下匹配得到的结果的点，前 20 步所有的格局都能匹配到相应的落子方法，可以明显看出来有围棋定式的走法在里面。大多数在同一棋子总数的格局链表里面就可以匹配到相应的格局。

优化方法

置换表

置换表是棋类 AI 中非常常见的一种方法，配合 zobrist 键值，在哈希表中保存每个格局，每当搜到一个新的格局时，在置换表中查找是否已经存在，如不存在才创建新的格局，这样能够避免了重复格局的创建，在局数相同的情况下，能搜得更深。这部分主要的工作可以分为如何构建置换表，以及如何使用置换表。

Zobrist 键值是一个 64 位的无符号整数型数据，初始化就用 64 位随机数填充 U64 zobrist[2][MAX_BOARDSIZE]，下子和收子就让两个格局做异或运算，基本不会出现两个格局 zobrist 键值冲突的情况。

但是实现置换表需要更改原 uct 的数据结构，增加一类边节点，表示从这个格局到另一个格局做出的改动。增加结构体使 uct 的操作变慢，所带来的损失和置换表带来的提升还得平衡一下。

我尝试在 uct 创建子节点的时候先去查表，如果查到了就直接返回该格局的指针，如果没有查到就新建一个节点，并更新置换表。显然只要最大最小的层数到达 3 层或 3 层以上，就应该会出现重复的格局，层数越多，或者模拟的次数越多，都能使重复的格局变多，命中率变大。我在测试的时候发现置换表的命中率还是相当高的，前期大概有差不多十分之一，后期甚至能达到三分之一，具体的实现过程和测试结果在小组报告中给出。

1. 基本思路

置换表是棋类 AI 中非常常见的一种方法，配合 zobrist 键值，在哈希表中保存每个格局，每当搜到一个新的格局时，在置换表中查找是否已经存在，如不存在才创建新的格局，这样能够避免了重复格局的创建，在局数相同的情况下，能搜得更深。

2. 实现过程

2.1 Zobrist 键值

为了给每个局面标记一个 key 值，我们采用 zobrist 键值，它具有快速匹配、不易冲突、异或运算方便实现落子提子的优点。我们使用的是 64 位整型来表示 zobrist 键值，初始化就用 64 位随机数填充 U64 zobrist[2][MAX_BOARDSIZE]，下子和收子就让两个格局做异或运算，还实现了根据棋盘状态生成相应的 zobrist 值的工具函数。

2.2 uct 数据结构的改变

置换表强调的是格局的不重复，但是一个格局可以由很多种下法来得到，这样原来只记录落子位置的 uct 树就不再适用。类似棋谱的数据结构，我们增加了边节点连接上下两个格局。格局节点保存的数据项有：zobrist 键值、赢数、访问次数、指向的第一个边节点的指针，边节点保存的数据项有：落子位置、指向的格局是否是经过这个边节点创建的（为了不重复回收地址空间导致的报错），儿子是指向的格局，兄弟是上一格局的另一落子点。

数据结构的复杂化，也使得 uct 的速度下降了。

2.3 置换表的数据结构

置换表就是一个哈希表，是哈希节点组成的一维数组，每一个节点包含的数据项有：zobrist 键值、指向的格局的指针、对应格局的棋盘状态。

2.4 查表

在 uct 扩展节点创建子节点的时候，首先生成新节点的 key 值，根据这个 key 值查找哈希表。由于计算机对除法的运算十分耗时，但是移位运算可以很快，所以实现方法是，将置换表的大小设为 2 的 n 次方，ZOBRIST_HASH_TABLE_SIZE_MASK 设为置换表的大小减一，使用如下运算：

```
hash_table_head *head_node = &hash_table[key & ZOBRIST_HASH_TABLE_SIZE_MASK];
```

得到的 head_node 即为置换表的一个节点。如果不为空，再检查棋盘状态确认是不是正是我们需要的格局。如果是可以直接返回指针，否则新建一个边节点和格局节点。

2.5 指针的释放

由于可能会出现两个边节点指向同一个格局节点的情况，所以在释放指针的时候会因为经过不同分支重复释放了同一个格局导致报错。所以我在分配空间的时候就记录了是谁分配的，遵循“谁分配谁释放”原则。

3. 测试结果

经过我多次的盘数、命中次数和深度的测试和比较，大致能够看出置换表的效果如何，下表是其中一次对比测试的结果。

从结果我们可以看出几点：

a. 虽然增加了一个结构体非常耗时，但是经过代码优化之后反而比改之前的 uct 模拟的盘数要更多，对速度的影响在可以接受的范围之内；
b. 哈希命中率比较理想，在前期深度不算很大的时候能有将近十分之一的命中率，到了后期甚至能达到二分之一；
c. 从最大深度来看置换表的优势并没有很明显，深度主要还是和局面有关。当棋盘上能下的子数比较少的时候就可以模拟很深，会有一个深度的顶峰。

技术路线

	9.6（纯 uct+置换表）			9.7（纯 uct）		改了数据结构之前的纯 uct
	盘数	Hash hit	最大深度	最大深度	盘数	盘数
0	121268	21740/372495	3	3	117692	117538
0	119252	19153/341895	3	3	118774	115802
20	96033	19452/264158	3	3	97833	96736
20	95819	17520/254940	3	3	97383	97104
40	93209	22233/223331	3	3	91692	90495
40	93386	18019/207567	3	3	91733	92167
60	95610	24720/195388	4	3	91683	86610
60	95924	18145/158497	4	4	92077	86945
80	104233	27493/173685	4	5	91456	89649
80	105817	18188/138735	4	4	91762	88744
100	107421	18897/117628	6	7	93826	90861
100	108551	17137/118856	5	7	95405	88118
120	116376	15528/79554	7	11	105412	96628
120	117460	15313/78160	6	9	108603	101076
140	132398	19491/75706	9	13	117609	113784
140	136179	21920/79988	8	14	120653	108737
160	160627	16221/61451	13	7	122769	129003
160	161316	17069/62180	11	8	126708	137855
180	160342	2393/5610	10	5	158648	161206
180	158860	3000/6691	5	5	159937	161083
200				13	158395	
200				14	158851	

表 1. 置换表测试数据

技术路线

并查集与气的处理

虽然这个方法很多组都用到，但是作为提高棋盘速度很重要的一个方法，我们还是想在这里做一下阐述。因为每次判断棋串是否为某一个位置的子提供气时，需要遍历整个棋串，这就导致了很多重复浪费的操作，所以用并查集，为每一个棋串维护一个气，是非常重要的。同时，我们在全局棋盘变量维护的是真气，在 uct 模拟时维护的是伪气。前者是为了能为为 uct 结点排序提供正确的信息，因为其中会涉及到吃只剩一口气的子，需要提供气的具体数目。而在 uct 中模拟时，我们没有用到真气的必要，用伪气已经足够了。由于维护真气需要很多操作，所以不再模拟中用真气可以大大增加模拟的局数。

UCT 的实现与优化

对于博弈类游戏，很容易想到的方法就是最大最小树，并用合理的方法剪枝。但这方法并不适用与围棋这类复杂性如此高的棋类游戏。通过上网查阅资料，我们锁定了蒙特卡洛随机搜索方法，简称 MC 方法。

我们如何才能知道哪一步棋很优呢？蒙特卡洛的方法的核心，就是通过对某一棋局随机模拟很多盘得到其胜率。随机模拟很多盘的主要思想，就是在某一个棋局之上，让黑白子采用随机下法对战，直至终盘，如此往复，模拟多盘，我们就能得到这一棋局的胜率。这一棋局是通过原始棋局下某一步得到的，所以这个胜率也就是这一步棋的胜率。胜率最高的那步棋，就是我们接下来要下的那步棋。

虽然蒙特卡洛方法简单易懂，但它的缺点也很明显，在一定时间内只能模拟出一定棋局数，而且没有将对方下一手考虑进去，等于只搜索了一步，而对于博大精深的围棋来说，只考虑一步是远远不够的。

进一步，我们发现结合树蒙特卡洛搜索和 UCB 公式的 UCT 方法。UCB 是一个表示结点价值的公式，不仅包含结点的胜率也涉及到了结点被访问的次数，UCB 公式是对胜率和被访问次数的平衡。胜率越高 UCB 分数越高，被访问的次数越少 UCB 分数也越高，简单理解就是胜率越高说明该结点的赢的可能越高，而被访问的次数少的结点需要给它机会来证明的自己的价值。

根据 UCB 获得 UCT 权重$Q^+(s,a)$计算公式为：

$$Q^+(s,a) = Q(s,a) + c\sqrt{\frac{logN(s)}{N(s,a)}}$$

在最初的实现中，我们的参数 c 设成了定值，然而随着进度的推进，我们发现最初是的 UCB 公式似乎并不那么“好”，于是我们去寻求其他 paper 的帮助，最后在五个公式中跳出了最合适的一个，最后通过论文[1]，我们更换了 UCB 的公式。

$$Q^+(s,a) = Q(s,a) + \sqrt{\hat{V}\frac{logN(s)}{N(s,a)}} + \frac{logN(s)}{N(s,a)}$$

其中$\hat{V}$的计算方式为

技术路线

$$\hat{V} = max\left(0.001, Q(s,a) * \left(1 - Q(s,a)\right)\right)$$

其中$Q(s,a)$表示的就是胜率，也可以看出来，随着胜率的稳定，后一项所占比重增加，其实在检测中发现，大概万盘模拟之后，后一项对权重的影响就十分小了。

具体伪代码：

procedure uctSearch(*s*0)
while time available **do**
  Simulate(*board, s*0)
end while
*board.SetPosition(s*0*)*
return SelectMove*(board, s*0, 0*)*
end procedure

procedure Simulate(*board, s*0)
*board.SetPosition(s*0*)*
[*s*0,..., *sT*] = simulate_game*(board)*
z = get_score *(board)*
updata_node *(*[*s*0,..., *sT*], *z)*
end procedure

procedure simulate_game (*board*)
c = *exploration constant*
t = 0
while not *board.GameOver()* **do**
  st = *board.GetPosition()*
  if *st* ∈ *tree* **then**
    creat_new_child *(st)*
    return [*s*0,..., *st*]
  end if
  a = uct_select *(board, st, c)*
  board.Play(a)
  t = *t* + 1
end while
return [*s*0,..., *st*−1]
end procedure

其中各个子函数在实现都很简单，就不再赘述。

剪枝

如果事先在树搜索时砍去一些预计不会很好的枝，可以减少 uct 的遍历，对预计优秀的结点多一些模拟和搜索。我们尝试只将一些我们认为好的着棋作为 uct 的结点。包括上一步棋周围的某一个范围内

的空点，上一步被吃掉的棋串的位置等等，将其中合法的结点作为 uct 的结点。但尝试下来的结果都不尽人意，都不如之前的 uct 棋力高。围棋博大精深，限制条件太过绝对，反而效果不佳。

但是后来我们想到可以砍掉一些显而易见不够好的下棋点，比如周围 8 个点都没有棋的位置（不让棋下在空旷地方），一定步数下棋谱边缘的点等等。在 uct 不够强的情况下，剪掉一些这类的结点，可以提高一定的棋力。

节点排序

我们在尝试剪枝的时候发现单纯的去除一些结点效果并不是很好，所以我们想到是否能将 uct 的结点做一个评分，也就是对每一个可以选择的着棋方法评分，按照这个评分给 uct 结点排序，分值越高，越先拓展该子节点。我们考虑到的有几下几个方面：吃子、救子、紧气、做真眼、连接、切断以及避免填眼，对于符合这些情况的点我们会赋予一定的权重，具体分数由该点的贡献来决定。

这样可以让 uct 更有针对性，对一些我们认为较好的点进一步拓展搜索。我们也有试过给 uct 子节点的 uctvalue 直接赋值，但是效果并不是很好，所以我们仍采用排序的方法。

多线程

MCTS 的优点之一便是利于并行，分为叶并行，根并行，和树并行，树并行可以充分利用通信带宽，不同线程并行化执行模拟，并共享内存中操作同一棵树。通常，我们需要一个全局锁来保证对树的而修改不会冲突，但在[5]中提到了一种不需要锁的多线程方法，我们基本就是在这一思想下实现的。

节点的 UCT 评估值是通过计数和平均值存储的。平均值通过一个增加的算法来更新，如果通过无锁算法，会导致平均值的更新丢失，也就是计数失效。假设有个线程，读了计数同时被另一个线程进行了写操作，第一个线程将会看到一个错误的状态。在实际中，这个默认更新操作只在很小概率下发生而且对计数和平均值只有很小的影响，所以我们可以忽略这些影响。

策略优化

我们的策略主要有三种使用方式：一是在 uct 之前一些紧急的情况要及时下出对策，比如救子吃子等行为；二是使用策略对棋盘上的点进行评估，挑选出一些比较优秀的点，给出相应的分值，经过排序这些候选位置全都传进 uct 里，依靠大量的模拟最终挑选出最优下法；三是在每一盘模拟里面都加上策略，对完全随机做出一定的改进，不再是两个完全傻子的对弈。

目前项目中实现的技巧主要有：布局，提子，叫吃，紧气，连接，切断，做眼等等，还有在一定步数下不能下到棋盘边缘，一定步数下不能在空旷的地方落子等限制。策略大同小异，下面在 UCT 中的模拟策略部分作以阐述。

1. 基本思想

通过添加领域知识，相比于原来的传统随机模拟来说，好处在于模拟看起来更具战略。我们最开始就想到了救子吃子这种最基本的策略，在后来的调研中，也看到了许多其他人尝试出来具有普适性的模式，便加以改进利用。在[1]中，作者介绍了 4 种 pattern，而且经过测试是比较适合的，所以主体还是这些 pattern。 我们需要注意的是，pattern 仅被用于在找局部解当中，并不一定是全距最优解。因为

技术路线

在蒙特卡洛模拟中，找到一个更优的序列比找到更优的一步棋更为重要，如果将 pattern 应用于全部棋盘，反而会降低准确率。

除去 pattern 之外，其他的策略也十分重要，在[1]中，作者综述了 6 种 pattern，分别是：

- Atari defense move：救子，遍历棋串，找到只有一个气的棋串，决定救不救。
- Nakade move：防止对方做眼，如果有三个连续的空位置，并且被对方棋串围着，则下中间的一步
- Fill board move：找到棋盘上的空位置下
- Pattern move：与 hane，cut，border 等模式进行匹配并下棋
- Capture move：如果对方棋串仅剩一口气则吃掉
- Random Move：随机产生一步，如果是合法位置则下在这一位置

2. 实现方法

我们仅仅考虑再最后一步棋附近匹配 pattern。因为全局匹配代价太大，耗时太多，而局部最优解最有可能是在最后一步附近。

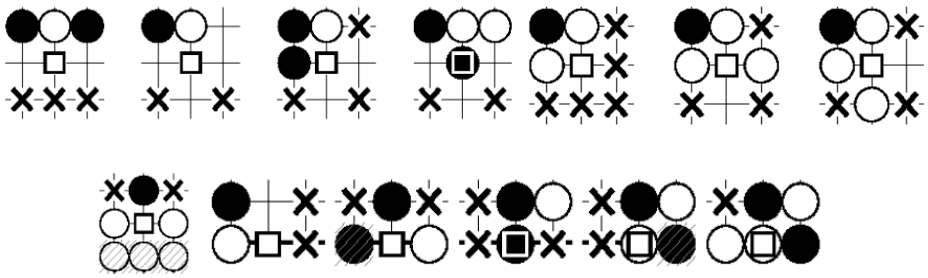

图 2. Pattern 图示[左上四个为 hane，右下四个为 border，其余为 cut][2]

Patern 部分的实现难点主要在于坐标转换与判别，在我们的设计中，根据最后一步子周围八个点是否与设定的 pattern 匹配来进行，匹配过程中不考虑转换问题，坐标转换部分。

坐标转换

由于匹配 pattern 的时候我们假设棋面已经旋转至指定方向，因此就可以不考虑坐标问题，直接与指定 pattern 进行匹配。为了实现坐标转换，首先要明确，我们应当考虑在 3*3 的格子内，我们有三种转换方式，上下翻转，左右翻转和绕对角线翻转，对于这三种旋转方式进行排列组合，我们可以得到 8 种转换。

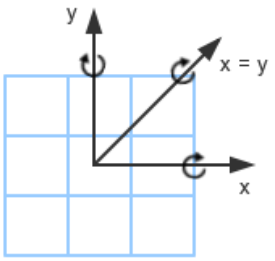

图 3 坐标转换示意图

通过这 8 种转换，我们可以对进行每一种转换后的棋面与 pattern 进行匹配，得到匹配的结果后返回。需要注意的一点是，border 这一 pattern 于其他 pattern 较不同，需要提前判断是否在 border 上，如果不在的话，可以直接跳过这一步骤。具体流程为：

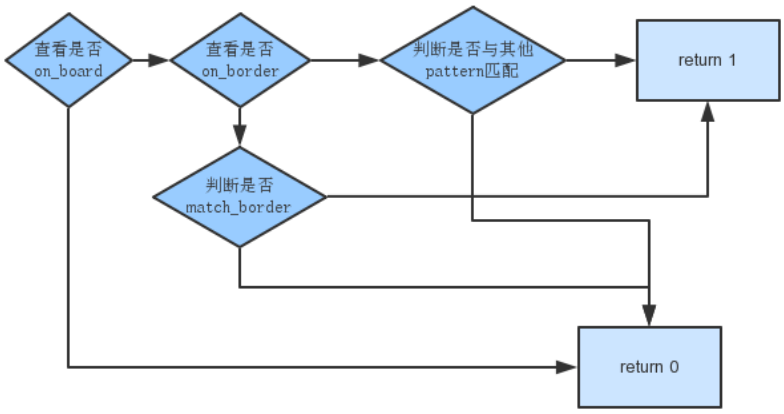

图 4 pattern 匹配部分流程图

而在实践中，我们发现如果棋走的越分散，棋力会有明显下降，所以就去掉了 fill board move, 而且由于加入 pattern 过于影响速度，最后也并没有全程开策略。

UCT-Rave

UCT-RAVE 是 UCT 和 RAVE 两种方法的融合，就 UCT 来说，需要根据 MC 模拟，推算当前状态下最佳的步骤，也就是赢率最大的一步棋，这样的话，对每个节点都要模拟成百上千盘棋局，在有限时间和计算能力下，得到的信息是非常有限的，而 RAVE 则关注于相关节点之间的胜率关系，从而得到一些相关信息，尽管这些相关信息可能并不十分准确，但能够得到一个较准确的估计值。

RAVE 是根据 AMAF 启发式权重进行搜索，这种方法的本质是在于，对于每步棋都有一个权值，这个权值不随执行这步棋的时间改动。其实说这种方法是违背围棋精神的，毕竟在实际的对战中，每一步都是具有很强的时间敏感性的，比如说现在救子吃子都很重要，然后选择了吃子，如果把这一步救子的

权值累加到下一步，就会有一个问题，因为再过一步，救子没救成，已经被对手吃掉了，然后之前的权值其实已经一点价值都没有了。这样在战场上无疑是很致命的错误。

1. 基本思想

为了扬长避短，UCT-RAVE 就是汲取这两种方法的优点，通过对短期权重和长期权重的平衡，得到的一个权值。RAVE 方法的学习能力要比 UCT 快，但是准确率没有那么高，UCT-RAVE 呢，则是通过将快速学习得到的全盘权重，和 MC 模拟得到的准确的实时权重结合，从而得到一个较理想的权值。

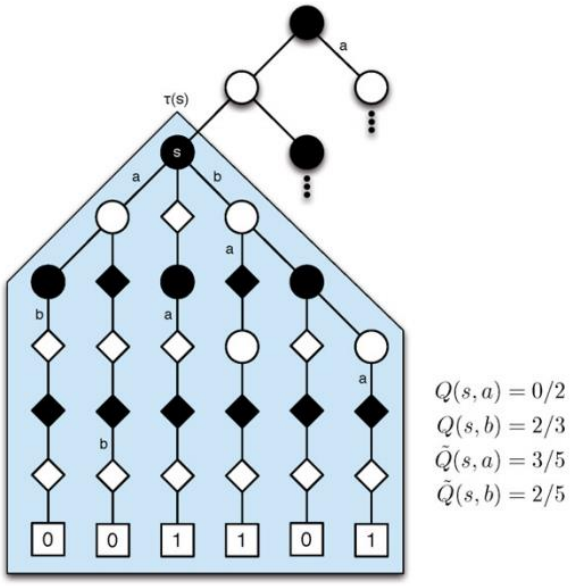

图 5. RAVE 图示[4]

具体的 UCT 和 RAVE 比较可以参见图 1，其中 s 是当前棋盘，a 是当前设置权重的位置，$Q(s,a)$表示 UCT 胜率，$\tilde{Q}(s,a)$表示 RAVE 胜率，$Q_*(s,a)$就是加了权重的 uct-weight 和 rave-weight。

$$Q_*(s,a) = \big(1 - \beta(s,a)\big)Q(s,a) + \beta(s,a)\tilde{Q}(s,a)$$

其中$\beta(s,a)$的计算公式为：

$$\beta(s,a) = \sqrt{\frac{k}{3N(s)+k}}$$

其中 k 是设定的调整权值的参数，当模拟盘数为 k 是，UCT 胜率和 RAVE 胜率权重相同。

UCT-RAVE 的公式就可以表达为：

$$Q_*^+(s,a) = Q_*(s,a) + c\sqrt{\frac{logN(s)}{N(s,a)}}$$

其中 c 是经过调整的参数，关于 c 参数的调整在 uct 部分已经介绍过，这里就不再赘述了。

2. 实现方法

由于在 AMAF 中，不仅仅是当前进行模拟的节点有存储价值，在每盘模拟中的所经过的所有的位置都是有利用价值的，因此在所有的模拟都需要对经过的节点进行统计，如果这保存在一个数组里，然后作为参数传递，肯定代价是很大的，因此我们定了了数组 AMAF_wins []数组，这个数组的 size 就是当前所有棋盘上的位置，根据以当前位置开始的模拟棋局中一个位置是否出现，来判断这个位置的价值。

我们首先定义了一个 AMAF_DEFTH 变量，这个变量是为了调整 AMAF 的深度，在模拟盘数较大的情况下，这个深度意义其实不大，但在模拟盘数较少，比较侧重 AMAF 值得时候，这个参数就十分重要。在 20 步以前和 50 步以后，这个参数都不大，定为 4，因为在开局的时候每一步都差不多，所以模拟本身就不是很有意义，10 步以前我们通过棋谱定式得到下一步棋的位置，10~20 步则基本要靠吃子、救子、和连线来进行，纯 UCT 一般就能得出比较好的点。而 50 步以后，由于每一步都很关键，而且每部棋的时间敏感性很大，所以 AMAF 的深度也很小。

定义 AMAF_DEPTH 变量的意义在于存储相应的节点，在每次模拟结束之后，将这些数据返回至主程序，然后在更新节点权值的时候，更新 AMAF_wins 的数据，UCT 胜率和 AMAF 胜率的计算方法是一样的，得到两种胜率之后通过上文介绍的公式就可以得到最终的权值。在加了 AMAF 的情况下，模拟盘数将近要下降一半，这个代价也是很大的。在单线程情况下，UCT-RAVE 的搜索结果还是要逊色于多线程的纯 UCT 搜索结果。

项目历程

将会在这部分阐述我们具体完成这项工程的过程，也算是我们小小奋斗史的简略记录。在这次合作项目中，我们收获的不止是算法和策略的思考，也有团队合作的宝贵经验。

我们的项目从无到有，从基础到慢慢优化，出于管理以及对比需求，我们一直都有在做版本记录，直至我们比赛之前，有几十个版本，当然并不是指我们有几十种策略，我们的棋力也并不是随着版本号线性增长的，但是我们的想法和工作都累积在这些版本之中。在之后的报告中，我将会选择具有代表性的版本进行说明。我们靠 git 管理代码，项目原本在 github 上后来转到了 bitbucket 上面。

工作大致分为功能方法实现，优化方法加速，debug。三种工作每个时期都有，但是侧重都不太相同，但是总的来说基本思路就是前期集中大量开发功能方法，后期根据测试结果做 debug 和优化。

阶段一：集中完成各类方法

我们四个人都是没有围棋经验的人，对于围棋的知识仅仅停留在知道规则而已，并且对于围棋人工智能的东西也知之甚少，同时因为初期基础代码不便于团队合作所以我们一开始的分工是两个人做开发两个人做理论学习。

在此期间我们完成了 version1.x 和 version 2.x 系列

Version 1.x 基本数据结构和基础棋盘

版本 1.x 系列，是在 brown 的基础上，规划工程数据结构，完成新数据结构下的棋盘。因为 Brown 的本身的数据太过单一，不足以支持我们的工程，我们也用空间换时间的方法新添加了一些记录除去棋盘状态之外的数据。

Version 2.x 基本单线程 Uct

Version 3.x 更换棋盘结构以及加速对棋串的维护

我们在这里做了一个对比试验，利用之前加入的 uct，做 uct 在 brown 上和在 version1.0 上面的性能比较，以 10s 能模拟的盘数作为评判标准。发现之前所抽出的高层结构降低了 uct 的性能至少两倍。所以我们总结原因是结构太复杂，耦合性高，同时可能是因为结构体的嵌套导致分配空间时增大时间开销（by TA）。所以我们决定放弃之前的结构，做了最简单的结构，虽然从代码上来说看起来并不优美，但是效率毕竟还是最重要的。而之后我们一系列的修改也证明这次的重写是有意义的，因为在简单的结构上增删更为高效。

我们用并查集对棋串的维护做了优化，加快了棋盘速度。

Version 4.x uct 多线程优化，导入棋谱，加入四种 pattern

到 4.x 版本我们就基本完成了一开始预设的功能方法。

- 多线程 UCT

- 四种 pattern
- 开局定式
- 棋谱搜索
- 简单技巧，比如提前吃字，立即救子，提气，做假眼等。
- RAVE 方法

此时整个项目的组合方式是：开局定式->棋谱前 20 步->如果有特殊情况用技巧返回点不然 uct 搜索（uct 搜索中有 pattern 做优化）

阶段二：测试寻找问题，排除逻辑 BUG

Version 5.x – version 7.x

利用助教提供的上一届的 exe 反复对战测试观察问题，因为这时候的 debug 难度要比之前都高很多，毕竟每个功能都有可能出现逻辑 bug。顺便也在这个时期观察下其他 AI，猜一下他们使用的方法。

观察点：

- 是否有开局定式
- 是否有 trick，比如提前救子，优先吃子，特定走法
- 是否有 pattern
- 是否使用多线程

GnuAplus_v5 对战记录

GotodieOut	Demon	Gg	IAmHungry	IdiotGo
w+17.5	W+45.5	W+1.5	W+15.5	B+16.5

表 2. GnuAplus_v5 先手对战记录

虽然只是部分，但是几乎是全败的记录，但是从对战中确实发现了不少问题。

以下是我们发现的问题以及解决方法，只做部分摘要

现象	问题	解决
棋局还有可以下的地方却报 illegal 错	我方气判断错误导致合法的地方 illegal	重新计算气
棋下的过于散		重新计算气
棋谱单一		
棋会填自己的眼	因为救子的优先级过高加上气的计算有误导致	重新计算气

表 3. 问题与解决方案记录表

可以很明显的看出，我们大部分问题的根源是对于气的计算。我们重新计算了气，然后同时保留了真气和伪气两种计算方式和对应数据。是为之后优化多留一个选择。

项目历程

阶段三：优化围棋项目

这大概是最难的一部分了。不是实现上的困难而是策略的问题，就是说这阶段做得多不等于做得好，做的东西在不在点上很重要，毕竟测试要花大量的时间。这一阶段里我们以优胜和找到优化方向为目标和上一届的程序模拟了大量的对战。因为前面已经对于我们的技术路线有很详细的解释了，这里对于技术方面就不再赘述了而是从几个重要的组成部分来简述我们的优化过程。

UCT

UCT 无疑是我们整个项目最核心的部分。之前我们都是在为能让他搜的更深更多而做努力——努力减少其他方面带来的开销，甚至不惜放弃 uct 中的策略优化和剪枝。纯 UCT 情况下我们达到 9.5s 内 450K。虽然用纯 UCT 我们已经打败了上届半数以上的成果，但是我们依然觉得我们的 UCT 比他应当表现出的水平要低（这是和别组同学交流而得）。后来我们测试发现并不是模拟的越多越好，虽然加速还是有意义的，但是时间可能应该拿来做别的事情，也根据另外的论文调整了 uct 参数。

策略

我们想过也实现了很多策略，从简单的吃子救子到复杂的前中后场不同策略组合，但是因为测试很难，除非某种组合能 90%赢过纯 UCT 才能说这种组合提高了棋力，但是测试时间太过漫长，我们最终获得的组合也只是一个相对较优的解。因为对 UCT 的重新认识，让我们再一次重视了策略的重要性，也有相关论文给出了较优的排列组合，但是最终我们也是根据自身的情况做了一些调整。

减少时间开支

这一部分一直是我们比较重视的地方，在这里付出的心血也较大。后来我们想了很多方法，比如置换表，GPU 编程等。后期在 UCT 上加上各种策略之后，搜索盘数又成为了我们的一个瓶颈，置换表从数据上来讲，确实达到了 1/10 的重复率，但是实际实验结果是反而因为构建置换表而增加了时间开支。最后也尝试用 GPU 编程，但是因为种种原因没有完成，没有切实的对比数据，这里也就不再赘述了。后期所做的优化真正起作用的还是对数据结构的优化。

以上就是我们完成这个项目的过程概述了，这里毕竟是在做完之后完成的报告，看起来还有些条理，在实际开发过程中，还有太多无法一一表述清楚的纠结过程，我们用了很多的时间走了很多的弯路也遇到了很多的坑，很多应该是可以避免的，但是总的来说项目合作还是很愉快的。

总结与后记

技术报告到这里就接近尾声了，这次项目开发是我们大学以来首次去实现一个博弈类项目，而且最后的分数也有很大比重在最后的竞技成绩中，是十分有新意的。这种博弈思想不仅体现在我们的程序中，也体现在了我们的实现过程里，每一次版本的更新，方法的改进，其实也都是跟之前自己的博弈，有时失败，有时胜利，虽说失败让人懊恼，但毕竟总体仍是螺旋上升的。

我们的项目从无到有，从基础到慢慢优化，并不是每一次更新都伴随着棋力的提高，更多的则是发现这种优化方法并不能带来棋力的提高，或者并不适合我们的原有结构，比如说策略优化，加入了策略之后，MC 模拟盘数从 10 万多降到了 3 万多，对棋力的影响是十分之大的，所以最后即使实现出来了，比赛版本中也没有用到。这种做了无用功的感觉，其实是很影响积极性的，因为总觉得一样都是 UCT，大家的效果就很不一样，但是毕竟一分耕耘，一分收获，别的组肯定也是通过更多的尝试，才找了适合的方法，抱着这种思想，我们也一直都没有放弃，通过不断的尝试来尽量提高棋力，比如 UCB 的公式，我们换了四个，开场步数的确定，我们换了又换，最后也终于得到了一个最佳的版本。

另外一点感悟，就是像这种竞技性大作业，也不仅仅是闭门造车那么简单，比如说，需要时时关注别的组的进展，并通过对战来相互提高。比如说跟”神之一手”组的交流中，在我们 version4.x 的时候，跟他们比，可以赢 60 子+，过了几天，他们得到了优化后比较强的版本要求跟我们一战的时候，就可以赢我们 30 子+了，当时心里虽然很难过，但是这种受挫也给了我们继续优化的动力，等到我们 version7.x 的时候，就可以先后手稳赢他们了。

不过竞技只是一部分，友谊第一，比赛第二，在这个过程我们更多的是相互帮助，每一次对战，我们看的不仅仅是最终的结果，也会看 AI 的表现来给对方提意见，你们的棋为什么填眼，你们的救子很有效果，快速判断怎么做，这些交流让我们共同进步。

在整个开发过程中，我们分工明确，团结合作，通过文献，前人经验，对战实测多方面信息寻找优化 AI 的方法并努力实现，虽然有些没能实现出来，有些实现出来了效果差强人意，但重要的是我们在这个过程中所锻炼出来的能力与素质，和我们之间在这种高强度的工作配合中加深的友谊。记得在比赛前一天，大部分同学都已经结束考试周回家了的时候，偌大的自习教室只剩我们和另一个组，还在辛苦的寻找进一步优化的方法，这种经历虽然有艰辛，但也有共同奋斗的温暖。

其实最后的竞赛中，我们组的表现并不突出，但是毕竟这种竞技比赛总有偶然性，所以我们对结果也没有什么抱怨，因为我们深信我们已经做了足够多，学习到了足够多。

最后，还要感谢张老师的谆谆教诲，助教的热心指导，和同学们的关心与帮助，如果没有这些师长，同学的帮助，我们肯定无法取得现在的成绩，谢谢你们！

参考文献

[1]Chaslot G, Chatriot L, Fiter C, et al. Combining expert, offline, transient and online knowledge in Monte-Carlo exploration. 2008.

[2]S. Gelly, D. Silver, Combining online and offline learning in UCT, in: 17th International Conference on Machine Learning, pp. 273–280.

[3]Gelly, S., Wang, Y., Munos, R., & Teytaud, O. (2006). Modification of UCT with patterns in Monte-Carlo Go (Technical Report 6062). INRIA.

[4] Gelly S, Silver D. Monte-Carlo tree search and rapid action value estimation in computer Go. Artificial Intelligence, 2011, 175(11): 1856-1875.

[5] Enzenberger M, Müller M. A lock-free multithreaded Monte-Carlo tree search algorithm Advances in Computer Games. Springer Berlin Heidelberg, 2010: 14-20.