棋谱实现

1. 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。

2. 实现过程

1) “学习”：读取并保存 SGF 棋谱文件上的信息

SGF 文件

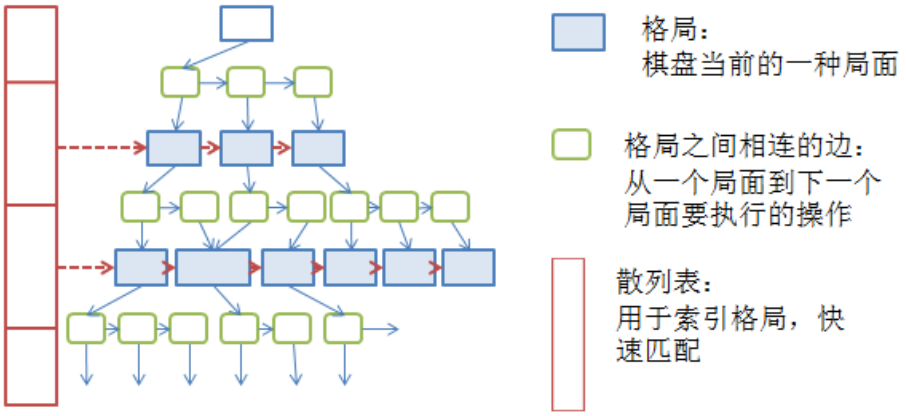
SGF 是为了存储双人棋类对局记录而设计的一种纯文本文件格式，由节点组成并构造成对局树，可以存储对局记录(一系列着子)和实战的变化图。

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。

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第另一个位置，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。



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：

① 当遇到“（”时，向栈 push 一个空指针；

② 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；
③ 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；
④ 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。

3. 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。

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

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。

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。

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。

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。

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：

① 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；

② 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；

③ 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。

	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	

策略

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，对完全随机作出一定的改进，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。