数据结构与算法课程
实习作业报告集

作者：SESSDSA’15班

编辑：陈春含　赵辉

指导教师：陈斌

2015 年　7 月

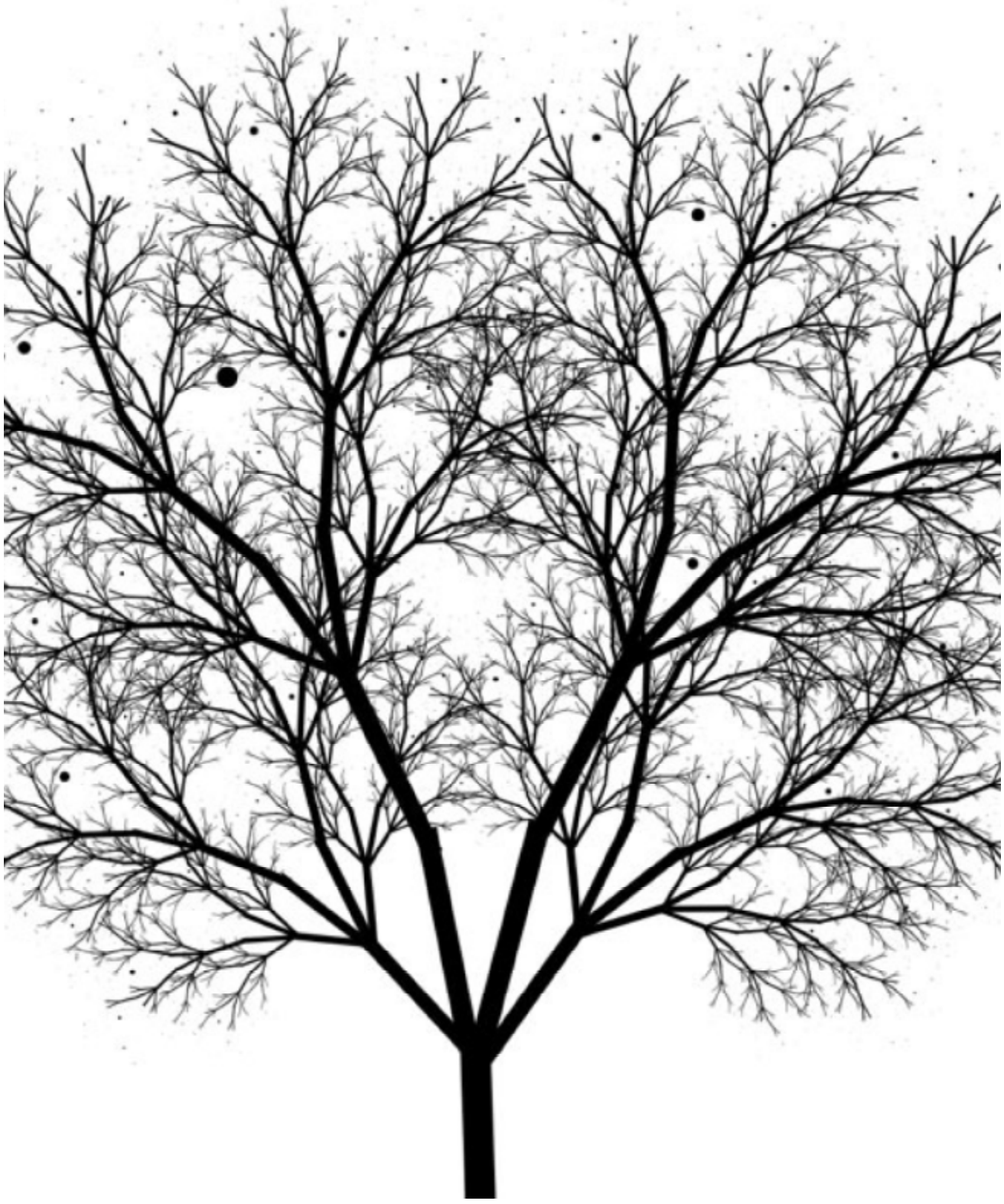
To SESS DSA'15

致地空2015数据结构与算法课程的

老师，助教与同学们

目录

CHAPTER 1



作业描述篇

>>>任务概要

>>>分组合作

>>>竞赛规则

>>>基础设施

作业描述篇

任务概要

本次实习任务旨在检验同学们的团队协作能力、动手能力和创新能力，以小组为单位，通过实习以达到巩固和掌握Python基础知识的目的，全过程分为编程、报告和竞赛三大部分。

编程

根据当前棋局，结合棋局日志，返回本方落子位置，要求每步计算时间不超过5秒（该时间可视调查调整）（基础设施提供棋局期间续存的存储空间）要求应用本课所学到的数据结构与算法，如栈、队列、链表、散列表、递归、动态规划、树、图等部分组合，并具有一定的复杂度和智能。

要求代码结构清晰、格式规范、注释丰富。

报告

纂写算法实现过程的实验报告，包括算法思想阐述、程序代码说明、测试过程报告（与idiot、gambler、自身算法对弈情况说明和分析）、小组分工和实验过程总结等4个部分。

要求实验报告图文并茂、内容丰富、结构清晰、写作规范、逻辑性强。

竞赛

参加SESSDSA黑白棋算法竞赛，与其他小组的算法对弈，根据输赢和累计计算时间获得竞赛排名。

要求对弈过程无bug、无异常。

分组合作

>分组进行实习作业，原则上每组6人，设组长1名（特殊情况经批准可以少于6人）。

>组队过程由组长确定开始，确定后组长开始招募组员。

>组长确定原则：历次作业优秀的同学，以及自愿报名相结合，自愿报名表单如下：
http://www.mikecrm.com/f.php?t=O3FtHe

〉组员招募遵循自愿原则。

〉组长负责召集实习作业过程讨论会，汇总代码和报告，代表小组参加竞赛

竞赛规则

首先将小组（可能达20组）抽签分为东西南北4个区(NEWS)。赛前进行热身挑战赛，为了避免代码泄露，参加热身赛的小组可将代码发给老师，以获得对其他小组的对弈结果和复盘数据。

〉第一轮

区内竞赛，循环赛制，每区2组出线，决出八强

每场下法：对弈的两组算法轮流执黑先下，各执黑3局，共6局，胜者积3分，负者0分，平局各积1分。

区内积分领先的2组算法出线，积分相同看得子总数，得子相同看时间消耗少者胜。

〉第二轮

淘汰赛决出四强：E1-W2,E2-W1,S1-N2,S2-N1
第二轮开始，每场必决出胜负，如得子平局则以时间消耗少者为胜。

〉第三轮

四强半决赛：E1W2-S1N2,E2W1-S2N1

〉第四轮

决赛：决出冠亚军和第3名，获得神秘奖品

基础设施

基础设施包括陈斌老师开发的基础算法文件、石瀚文助教开发的ReversiChecker程序和阎述辰同学开发的visualchessboard程序。

每个小组提交一个**py文件，文件名称为T_<组代码>.py**，如：T_ALPHA.py,T_BRAVO.py必须包括**函数play(cb,ms)：**

函数返回落子座标，如’A4’

如果无法落子，则返回’PASS’

如果可以落子，则不许PASS

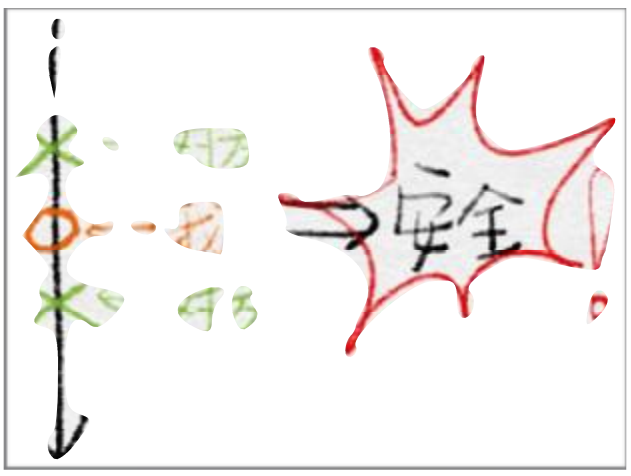
（1）函数**play会被对战平台race()在轮到己方落子时调用**，调用时会将当前棋局**mainboard**的拷贝传给**play**的**cb**参数，ms参数则是一个己方的私有存储字典，可以把一些下一次还需要用到的数据对象保存到ms字典里，如ms[‘tree’]这样。其中**ms[‘log’]**是一个**LogEntry**对象的列表，记录了整个棋局的下棋历史

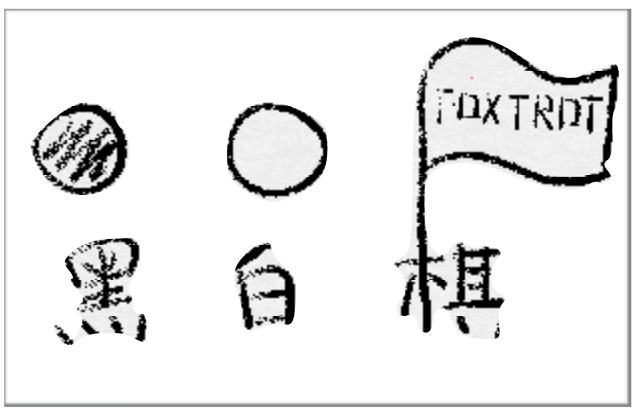
（2）将自己的算法**py文件**（例如ECHO组是T_ECHO.py）放在与**chessboard.py、T_idiot.py、T_gambler.py、reversi.py**同一个目录下，运行对战平台**reversi.py程序，**此程序会将三个算法文件（**T_ECHO.py/T_idiot.py/T_gambler.py**）进行两两交叉对弈。如果没有出错，则会输出9个对弈结果，结局见打印输出；同时生成9个对弈的复盘数据，如**T_idiot-VS-T_ECHO.dat**等，前者为执黑，后者执白。复盘数据可以用**visualchessboard.py**来可视化查看。

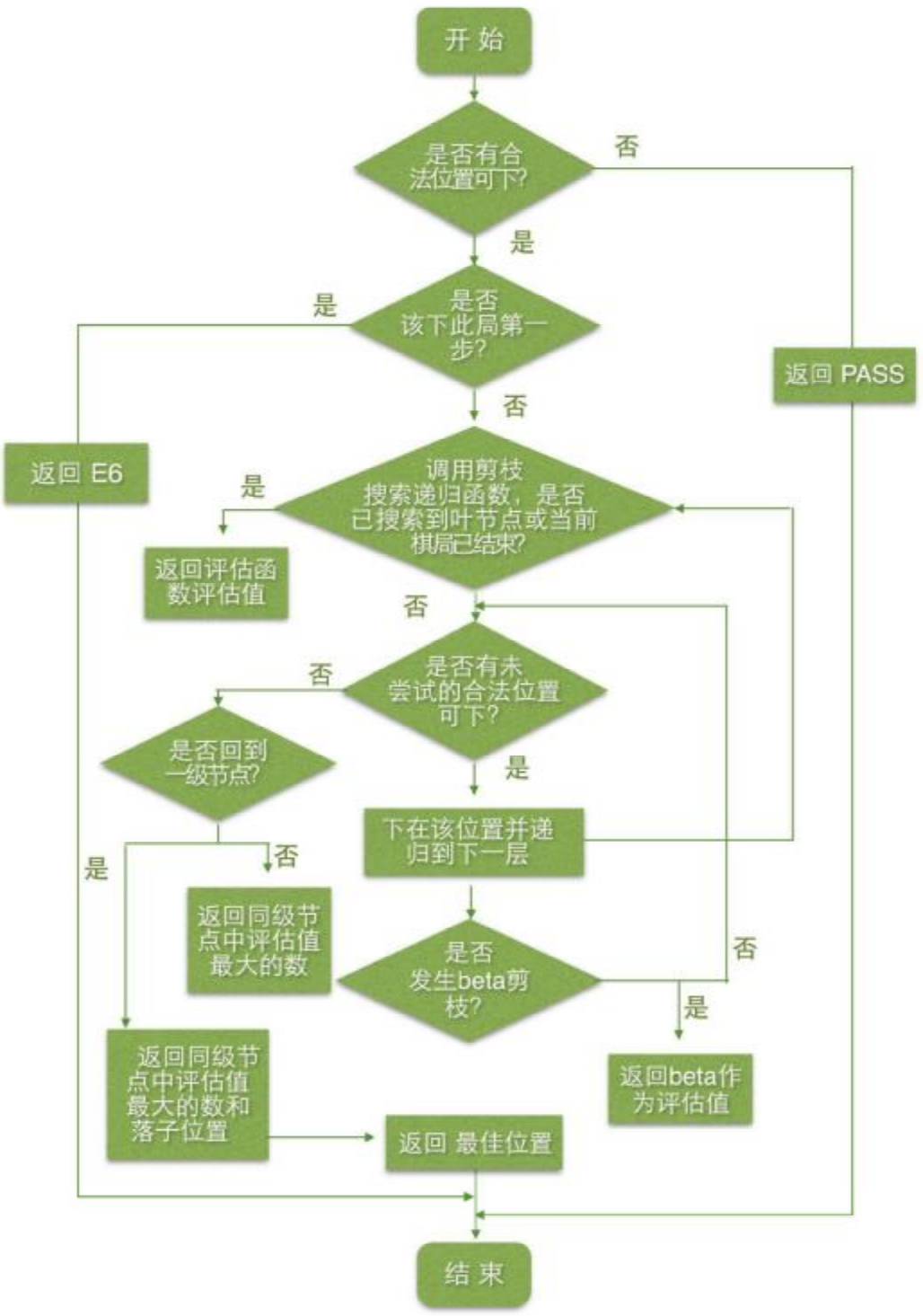
（3）如果需要人机对下来测试算法，可以将自己的算法**py文件**拷贝到**ReversiChecker**目录下，运行**ReversiChecker.exe**，选择算法后进行人机对下。

CHAPTER

2







分组报告篇

>>>东区

>>>西区

>>>南区

>>>北区

分组报告篇

分组名单

区号	组号	组长	组员 1	组员 2	组员 3	组员 4	组员 5
1 东区 East	DELTA	汪颖	石永祥	赵玖桐	龚世泽	苏瑞冰	宋欣源
2 东区 East	HOTEL	蔡天泊	李博扬	张子玄	赵辉	张君天	兰云飞
3 东区 East	MIKE	肖万博	余晓辉	阎述辰	高鸿宇	温景充	黄如许
4 东区 East	ECHO	陈春含	赵芳珩	吴逸夫	运乃丹	付帆飞	彭玉环
5 东区 East	OSCAR	谭光钰	李嘉琪	武化雨	彭玉恒	张恩珍	李佳斌
6 西区 West	QUEBEC	唐启浩	罗彪	黄翔	朱福海	周安	车元孟
7 西区 West	PAPA	廖宸睿	陈哲萌	张喆	何宇	章美希	谭陈东
8 西区 West	INDIA	赵琰喆	贾博	刘志扬	熊建学	黄知劼	向伟民
9 西区 West	KILO	朱贺	汪诗舜	党卓	蓝坤	胡哲	韦春婉
10 南区 South	GOLF	柳晓萱	马赞彭	黄天正	段鉴书	刘松吟	庞磊
11 南区 South	ROMEO	姜志远	虞志刚	周易	钟涛	杨冬偶	
12 南区 South	ALPHA	李子涵	葛天雨	刘明辰	周杰	唐钰开	杨礼萌
13 南区 South	LIMA	汪建峰	于润泽	蒋明轩			
14 南区 South	BRAVO	黄佳旺	韩甲源	徐世宇	吕世极	陈跃毅	曹越
15 北区 North	NOVEMBER	刘杰	刘嘉栋	韩露	杨润	张虎来	秦树健
16 北区 North	CHARLIE	龚旭日	吴永琪	姜鹏飞	卢思奇	李庆	蒋久阳
17 北区 North	FOXTROT	李然	蒙聪	林芷平	王静	吴梦彤	王慧君
18 北区 North	JULIET	张沙洲	陆杰	李昆鹏	仇立松	黄驰琳	仲启蒙

东区 East　　DELTA 组

1 数据结构与算法课程实习作业报告

（汪颖*，宋欣源，苏瑞冰，赵玖桐，石永祥，龚世泽）

摘要：我们总共是有 4 个算法（不包括中间算法），最终比赛的用了 2 个，其中 T_randomsearch 用了随机加权算法，T_MonteCarlo 用了蒙特卡罗算法（和随机加权差不多），T_DELTA4 用了最小最大算法和 AB 剪枝，T_DELTALAST 在上一个基础上加上了启发式加权。测试结果 T_randomsearch 比较失败，T_MonteCarlo 也不够稳定，T_DELTA4 和 T_DELTALAST 的算法较高级，最终被采用。

关键字：　T_randomsearch　　Random
T_MonteCarlo　　MonteCarlo
T_DELTA4
T_DELTALAST　　MINMAX　Alpha Beta

1.1 算法思想

1.1.1 总体思路

一开始，我们想到了一个随机加权算法（算法思路来自与同学的讨论），就是将自己与对方每一步都作为完全随机然后进行 n 盘游戏，输赢加权在可选择的下棋处上，当随机次数足够大时能够将每一步都尽量摆在赢的几率最大的步上（不一定是最优解），可是程序做出来后发现效果很差，连 Gambler 都打不过，可能是某处出现了问题而无法发现，而蒙特卡罗算法和我们一开始的算法本质上一样，可是它将每一步都分别进行了 5 盘游戏，再进行加权，数据结构是正确的，效果也很好，可是由于时间问题只能牺牲深度，导致我们放弃了该算法。（时间复杂度问题在下面描述）之后我们查找了资料，又找到了一种较为靠谱的方法——最小最大值算法，还有附带的 AB 剪枝，主要方法是将棋盘的每个位置都赋上权值，然后向下深搜 n 层，可是在搜索时进行剪枝，设立分支限界，用 Alpha 和 Beta 分别代表自己能取得的最大值和对手能取得的最小值，当发现最大小于最小时说明此树枝应删去，最后进行权值统计。而最终我们的成品是加入了基本的启发式算法，并修改了棋盘的结构，具体加权请参见程序。
我们最终采用的算法结构就是具有分支限界的回溯树，层数越多就越精确，而前面两种算法我们最后假想将其融入回溯可是时间明显不够，而且复杂度较高，因此未能完成。

1.1.2 算法流程图

用文字形式表示：
T_randomsearch
随机选择空位——>下一局完整的棋（每步均随机）——>记录输赢平——>进行权值统计——>选择“最佳”空位返回

T_MonteCarlo
接受空位列表——>选择空位——>进行 5 局随机游戏——>返回输赢平以及行动权值（控制对手能下的空位）——>进行权值选择——>返回“最佳”空位

T_DELTA4
接受空位列表——>选择空位——>向下搜索——>搜索到底部——>记录总权值，返回——>改变记录函数，判断是否剪枝——>若剪，返回，若不减，搜索下一个空位——>最终选择记录函数中权值最大的返回

T_DELTALAST
基本流程同上，在记录函数处加入启发式判断

1.1.3 算法运行时间复杂度分析

T_randomsearch
算法每次下一局棋需要循环 64*64=512 次(越往后需要要下的步数越少，为 k*64，k 为剩下的棋子数)，而随机次数决定了时间复杂度，100 局就要 51200 次，而 python 的运行时间比 C 差了很多（基于 C 的平台需要更多的复杂度）
O（n*k*64）
n 为设置的随机次数
k 为剩下的棋子数

T_MonteCarlo
蒙特卡罗算法在每一步都赋了 5 次局数，因此当前可下的步数取决于目前可下的步数，也取决于剩下的棋子数，而由于自己定义了 Board 类，因此在调用程序时也会占用大量时间复杂度
O（m*k*64+L）
m 为当前可下的步数
k 为剩下的棋子数
L 为调用函数所附加的复杂度（需考虑）

T_DELTA4

T_DELTALAST
这两个算法的时间复杂度相同，而每步所能下的步数为实时，树的深度是规定的，设为 n，平均每步可下设为 m，剪枝能减少的设为 L
O（m^n-L）
n 为树的深度
m 平均每步数量
L 剪枝能减的时间复杂度

1.2 程序代码说明

1.2.1 数据结构说明

程序 T_DELTA&&T_DELTA4：
两者使用树的结构，在当前节点，对每一种可能走法都进行一次模拟，每一次模拟都在当前节点生成几个分支。由于是指数级增长，只能模拟到 4-6 层。在叶节点处进行估值，统计得分情况。再由每一层选取最适合自己的情况，从而使得根节点选取到最适合的走法。改进是针对 minmax 算法加入了α ： β 剪枝，使得叶节点并不都一般深，从而节省了时间

程序 T_randomsearch：
同样使用树的结构，但这个树只在第一层发散，选取由根节点散发出来的可能节点乘上某一个具有统计意义的值，使得每一种情况都进行这些遍模拟，树的下层都只有一个分支，且每一层都很深（0（64））类似于垂杨柳结构

1.2.2 函数说明

程序 T_DELTA&&T_DELTA4：
minimax(cb, depth, player, is_max, alpha, beta):关于 alpha，beta 剪枝与最大最小值算法的实现
evaluation(cb, player): 对叶节点的局势进行估值运算的函数
其余为辅助函数

程序 T_randomsearch：
choose(chosen_list,chesspan,mine,enemy,empty):返回在模拟中赢得最多的走法
Try(chosen_list,chesspan,mine,enemy,empty)进行模拟一盘操作

Update、change、islegal、judge 均为实现棋盘功能的函数

1.2.3 程序限制

程序 T_DELTA&&T_DELTA4：

1 可选择的棋子走法过多、α ： β 剪枝效果差……导致的程序超时

2 play 函数参数传递错误
3 棋盘中出现其他格式的棋子

程序 T_randomsearch：
1 可选择的棋子走法过多导致的程序超时
2 play 函数参数传递错误
3 棋盘中出现其他格式的棋子

1.3 实验结果

1.3.1 实验数据

实验环境说明：

- 硬件配置：（CPU/内存）I7 4710MQ 8G
- 操作系统：（名称/版本）WIN8.1
- Python 版本：（版本号）2.79

黑\白	T_DELTA4	T_DELTALAST	T_gambler	T_idiot	T_MonteCarlo	T_randomsearch
T_DELTA4	0 0 10 time 1145.54:1604.56 score 250:270	10 0 0 time 864.85:284.39 score 440:200	94 1 5 time 11950.90:0.7255 score 4075:1930	10 0 0 time 1313.14:0.08 score 470:170	2 0 98 time 183.91:1775.09 score 2269:4129	96 2 2 time 11295.19:210.55 score 4116:1921
T_DELTALAST	0 0 10 time 145.57:397.24	0 0 10 time 504.52:557.76	100 0 0 time 2634.48:0.91	10 0 0 time 104.324:0.09	4 1 95 time 183.54:1740.66 score	100 0 0 time 1783.18:202.61 score

	score 230:410	score 250:390	score 5069:1330	score 500:140	2312:4086	5149:1250
T_gambler	6 2 92 time 0.79:12965.35 score 1685:4020	0 0 100 time 1.08:3033.38 score 1391:5009			9 1 90 time 184.08:1830.89 score 2497:3901	36 2 62 time 0.77:169.15 score 3050:3299
T_idiot	10 0 0 time 0.14:1664.37 score 250:170	0 0 10 time 0.13:355.16 score 110:530			10 1 89 time 182.31:1731.05 score 2399:4001	55 4 41 time 0.84:176.97 score 3401:2996
T_Monte Carlo	9 6 85 Time 1186.22:9438.22 score 1966:4280	1 0 99 time 1117.76:2108.09 score 1433:4965	91 2 7 time 1900.61:0.77 score 4047:2351	93 2 5 time 1760.49:0.80 score 4019:2336	53 2 45 time 1873.95:1787.56 score 3228:3172	85 1 14 time 1919.53:176.19 score 3951:2446
T_random search	2 0 98 time 225.16:10354.85 score 1747:4362	0 0 100 time 213.38:2043.88 score 1260:5140	47 3 50 time 185.52:0.79 score 3249:3150	38 6 56 time 192.03:0.85 score 2858:3494	8 1 91 time 247.41:2308.76 score 2314:4037	44 5 51 time 250.77:235.40 score 3016:3383

1.3.2 结果分析

我们程序较多，因此具有随机性的算法进行了 100 盘测试，无随机性程序进行 10 盘测试
程序主要是进行权重分析来得出最佳选点，随机搜索和蒙特卡洛算法利用随机性来分析权重，而最小最大算法则利用棋盘位置权重来分析，随机次数越多越精确，棋盘权值越接近真实分布越精确，而用时上来看，随机算法可以控制倍数增长，而最小最大则以指数增长，时间开销用在了搜索和判断上。

1.4 实习过程总结

1.4.1 分工与合作

本小组成员：汪颖*，宋欣源，苏瑞冰，赵玖桐，石永祥，龚世泽

小组分工	
汪颖	作为组长，进行任务分配，同时担任代码编写，调试，分析
宋欣源	代码编写，算法分析，调试
石永祥	代码编写，算法分析，调试
赵玖桐	数据测试，比较分析
苏瑞冰	数据测试，资料汇总

1.4.2 经验与教训

1、代码编写走入过的误区

一开始，大家盲目的进行构思，头脑风暴，想出很多有趣的想法，但是效果都不是很好，和老师给的网页版的黑白棋，中级基本上就不行了。后来，开始查一些文献，了解相关的算法——贪心算法、蒙特卡罗算法、阿尔法—贝塔剪枝法，工作就进入了正轨，不过前期却是浪费了不少时间

2、算法的分析

算法分析过程中，大家都有自己的看法，有时候不能说服对方，就按照自己的想法编写新的代码，然后比试，谁的程序表现更好，就采取谁的意见，这样做很能够锻炼编程技巧，而且在不断的尝试中，程序的性能有了很好的提升

3、内部交流

因为大家的宿舍都相连，晚上有时候便会聚在一起、讨论，交流互相的想法、意见，团队工作中，能够学到合作以及互相交流的能力。

4、比赛过程中的代码改变

比赛中，可能是心态的原因，出现了大的纰漏。本来在第一轮比赛中，我们的代码表现的十分优异，顺利出线，但是在第二轮比赛中，由于第一局和 LIMA 平局，太过慌乱，匆忙修改代码，但是情急之中，代码的修改出现了非常低级的错误，导致第二局大比分落后，然后第三局虽然改变了战术，还是没能力挽狂澜，惜败于 LIMA。

1.4.3 建议与设想

1、组队更加自主

组队方面，希望老师能够对人员分

2、竞赛

竞赛过程中，希望老师能够将比赛的代码编写更加完整，UI 界面更加美观一些。

3、开发成游戏

大家这么多组，编写出这么多 AI,希望老师可以集成一下，开发成一款小游戏，放到网上，可以让大家玩，而且可以让下一年的新同学再编写这个代码的时候，能够很快找到除了 idiot 和 gambler 以外其他的对手。

1.5 致谢

感谢组员宋欣源的 MIT 的同学，给我们算法以及程序有很大的帮助！也感谢组内的各个组员，能够团结一致，共同完成工作！

UNKNOWN MIT STUDENT'S HELP

1.6 参考文献

参考论文：

博弈算法在黑白棋中的应用—杜秀全—计算机技术与发展—2007.1

基于改进博弈术的黑白棋设计与实现—李小舟—硕士学位沦为—2010.5

东区 East　　HOTEL 组

2 数据结构与算法课程实习作业报告

李搏扬，张子玄，蔡天泊*，赵辉，张君天，兰云飞

摘要：本程序的大体思路如下：利用带有 Alpha-Beta 剪枝机制的 Minimax 算法模拟下棋情况，之后结合估值函数确定每个落子方式的得分前景，进而确定最佳落子位置。主要涉及了递归算法，面对 gambler 和 idiot 时均有 90%以上的胜率

关键字：递归 Minimax 算法 Alpha-Beta 剪枝

2.1 算法思想

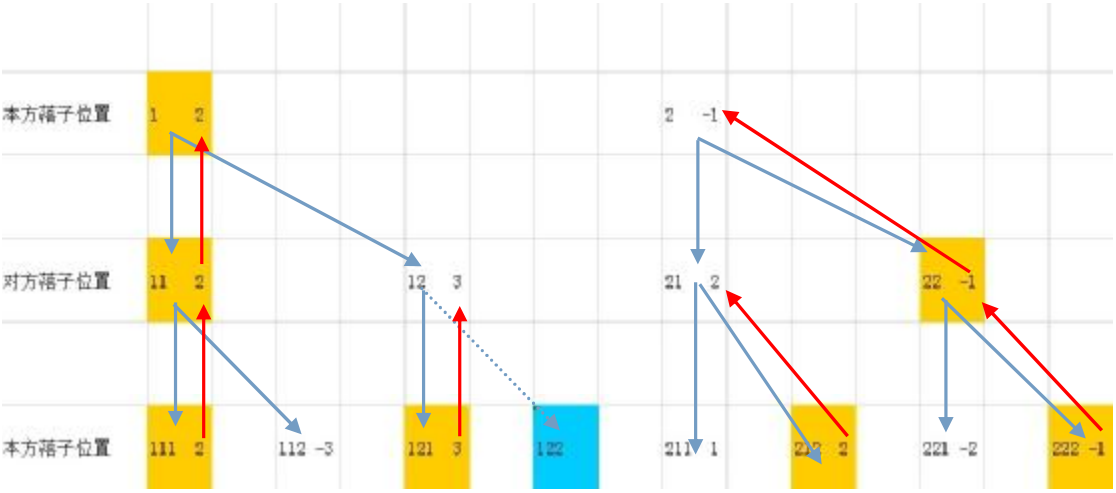
2.1.1 总体思路

本程序核心为 Minimax 算法：对于某一特定棋局，列出所有本方可以下的位置，在每个位置分别落本方颜色的子，返回“前景值”最高的位置。如何确定“前景值”呢？就是将“在该位置落子之后"的棋局作为新棋局，列出所有对方可以下的位置，在每个位置分别落对方颜色的子，找出”前景值“最低的位置并返回，此时求”前景值”的方式变回用本方棋子下棋之后取前景最高的位置…递归几层之后动用估值函数,判断该棋局的”前景”(对本方越有利就越大,对对方越有利就越小),并返回到上一层.
分析过程中用到 Alpha-Beta 剪枝来减少不必要的计算,具体剪枝方式详见算法流程图.
所用到的算法主要是递归,以及递归为基础的 Minimax 算法.

2.1.2 算法流程图

流程图说明:
每个格子中靠左侧的数字为位置编号,右侧数字为其”前景值”.
本图假设递归深度为 3,每方每回合都有且仅有 2 个可落子位置.蓝灰色箭头表示递归方向,由上向下依次加深;红色箭头表示估值返回方向.最下层格子的”前景值”使用估值函数算出.
其中 111,112 两个格子为”本方在 1 位置落子,之后对方在 11 位置落子”后,本方所能落子的两个位置.111 的估值为 2,大于 112 的估值(-3),于是本程序默认本方选择估值较高的 111 落子,进而 111 位置的”前景值”便返回给 11 位置,作为 11 位置的”前景值”.
此时可知,1 位置的”前景值”应当取 11 和 12 中较低的那一个.而 12 又当取 121 和 122 中较大

的一个.利用估值函数计算出 121 的"前景值"为 3 后,可知 12 的"前景值"一定是一个不小于 3 的数,进而可知 12 的前景值一定大于 11,无需再对 122 进行任何分析,由此节约了时间.这就是 Alpha-Beta 剪枝算法的大概原理.



2.1.3 算法运行时间复杂度分析

算法运行的主要耗时在于递归展开,复杂度约为 O(k^n),其中 k 表示每个棋局中可落子位置的数目,n 表示递归深度

2.2 程序代码说明

2.2.1 数据结构说明

算法中自定义了一种新类型的棋盘"CBdigital",继承老师程序里的"ChessBoard"类,同时增加了记录和返回本身"前景值"的方法(setValue 和 getValue)

2.2.2 函数说明

（说明算法中各主要函数的接口、功能、采用的算法策略等）
play 函数:参数为 cb(即现有棋盘)和 ms,功能是调用 search 函数,或是在无子可下的情况下返回 pass;其返回值为"PASS"或棋盘上的一个具体位置
search 函数:程序的核心函数之一.参数 cbD0 为现有棋盘,depth 为递归深度,myColor 为本方颜色,alpha 和 beta 为剪枝所需的参数;返回值是一个位置
Dsearch 函数:原理和 search 函数基本相同,区别在于它返回一个"前景值"而不是一个位置.用于对表层以下的下棋步骤(虚拟出的本方根据已有棋局下完第一步棋之后的种种情况)进行

处理.

Value 函数:估值函数,程序的另一核心函数.参数中 cb 为某一棋盘,bushu 为棋盘上已有棋子总数,color 为本方所用颜色.value 函数大体思路如下:

函数需要传输的变量有：棋盘情况，步数，某一方的棋子颜色

本函数从四个方面对某一方进行评估，得分越高，则对这方越有利

第一个方面：不同位置有不同权值，一般来说，边角的位置权值高，星位权值低。

第二方面：一般来说，对方行动力越低，对己方越有利

第三方面：己方稳定子越多，对己方越有利，一般中后期开始计算

第四方面：在前期时，一般来说己方子数越少越好

2.2.3 程序限制

实验中超时情况较多,剪枝算法用时不稳定.

2.3 实验结果

2.3.1 实验数据

实验环境说明：

- Python 版本：2.7

1000 次对弈结果数据汇总

黑 \ 白	idiot 算法	gambler 算法	本算法
idiot 算法	（胜负局数比） （累计总分比） （累计时间比）		85:142 4287:10241 1:10870.44
gambler 算法			4:235 3874:11347 1:11743.07
本算法	167:9 8690:2570 11194.98:1	236:0 11510:3510 12892.93:1	

2.3.2 结果分析

递归分析和估值使得本算法在面对 gambler 和 idiot 时拥有了很大优势;用时基本符合预

期,idiot 的复杂度约为 O (1),gambler 的约在 O(1)和 O(n)之间,均远远小于本算法.

2.4 实习过程总结

2.4.1 分工与合作

分工：蔡天泊写代码的搜索函数部分，兰云飞写代码的估值函数部分，李博杨负责将二者整合，张子玄、赵辉负责测试，采集实验数据，张君天负责实验过程的总结。

我们主要在微信群和组会时交流成果以及见解，各自完成自己工作后再到一起进行整合。

06.07 第一次组会，交换了一下思路，分享了一些有参考价值的链接。

06.08 第二次组会，关于前一天阅览的资料进行交流，讨论一些不清楚的问题，完成分工。

06.11 第三次组会，将我们完成的代码与几个现有的算法进行对战测试以及人机对战，根据对战结果完善代码。

2.4.2 经验与教训

在代码编辑和完善过程中，我们遭遇了很多问题。在代码形成之初，我们与 gambler 还互有胜负，然后经过讨论我们先尝试加深了搜索层数，然后通过人机对战发现代码执行中的漏洞，修改了一部分的参数。

2.4.3 建议与设想

感到比赛中所涉及的算法较为高端,实际上更多的是对课程内容的拓展,巩固和运用得相对较少,对自学能力和搜索资料的能力要求有些高了...总之适合学神和学霸,大多数学渣可能不打容易从中获取太多收获.

祝愿明年选修这门课的学弟学妹们能够享受数算的魅力,并在课程中有所收获!

2.5 致谢

感谢蔡天泊，兰云飞，李博扬倾力编写代码，赵辉和张子玄的测试以及统计，张君天对实验过程的记录，以及最最最要感谢老师和助教们在这半年内对我们的倾心指导。

2.6 参考文献

http://wybwzl.iteye.com/blog/1161895

http://blog.sina.com.cn/s/blog_53ebdba00100cpy2.html

http://blog.csdn.net/allenlsy/article/details/5324441

东区 East　　MIKE 组

3 数据结构与算法课程实习作业报告

MIKE 组：
肖万博*1400012445　阎述辰 1400012451　余晓辉 1400012414
黄如许 1400012406　　温景充 1400012411　高鸿宇 1400012446

摘要：

MIKE 算法的基本原理是：根据棋局的不同阶段部署不同的算法，棋局前期（开局到第 5 步）按照其他程序（电脑游戏）中的下法下棋，用到“树”的数据结构进行每一步棋的下法记录，用搜索的方法确定下子位置；中期（第 6 步到第 24 步）用多参变量估值函数选定最佳落子点，这些估值函数包括：棋盘基础位置、每步吃子、下棋自由度、边缘度等，这里用到的主要数据结构是列表，自定义类有“棋盘节点”，用到排序算法；棋局后期（第 25 步及以后）用搜索寻求最佳位置，取可能胜率最高的位置下子，主要用到递归的方法搜索到棋局的最后一步。

从总体上看，我们的程序能够稳赢“傻瓜”算法，在对赌徒的比赛中，胜率在八成到九成之间。我们算法与自身对决的时候，由于我们的算法不具有随机性，先手胜两个子。

关键字：树　列表　搜索　排序　递归　树的遍历

3.1 算法思想

3.1.1 总体思路

MIKE 算法的基本原理是：根据棋局的不同阶段部署不同的算法。

首先，将棋局分为三个阶段：第一阶段为棋局前期，具体为开局到第 5 步（棋盘上 4 个子到 14 个子）。第二阶段为棋局中期，具体为地 6 到 24 步（棋盘上 15 个子到 52 个子）。第三阶段为棋局末期，具体为第 26 步及以后（棋盘上有 53 个子到 64 个子）。

棋局前期的下法是“棋谱法”。即先记录其他程序（电脑游戏）中的前期下法，再根据棋盘状况，按照棋谱下棋。这里，需要用到“树”的数据结构进行每一步棋的下法记录。即以原始棋盘为“根节点”，每增加一步棋，数的高度就增加一级，下一步的每个走法就是对应一个棋盘父节点的子节点，直到中期棋局。具体到算法运行的时候，用搜索的方法搜索棋谱，找到对应的位置，确定下子位置。

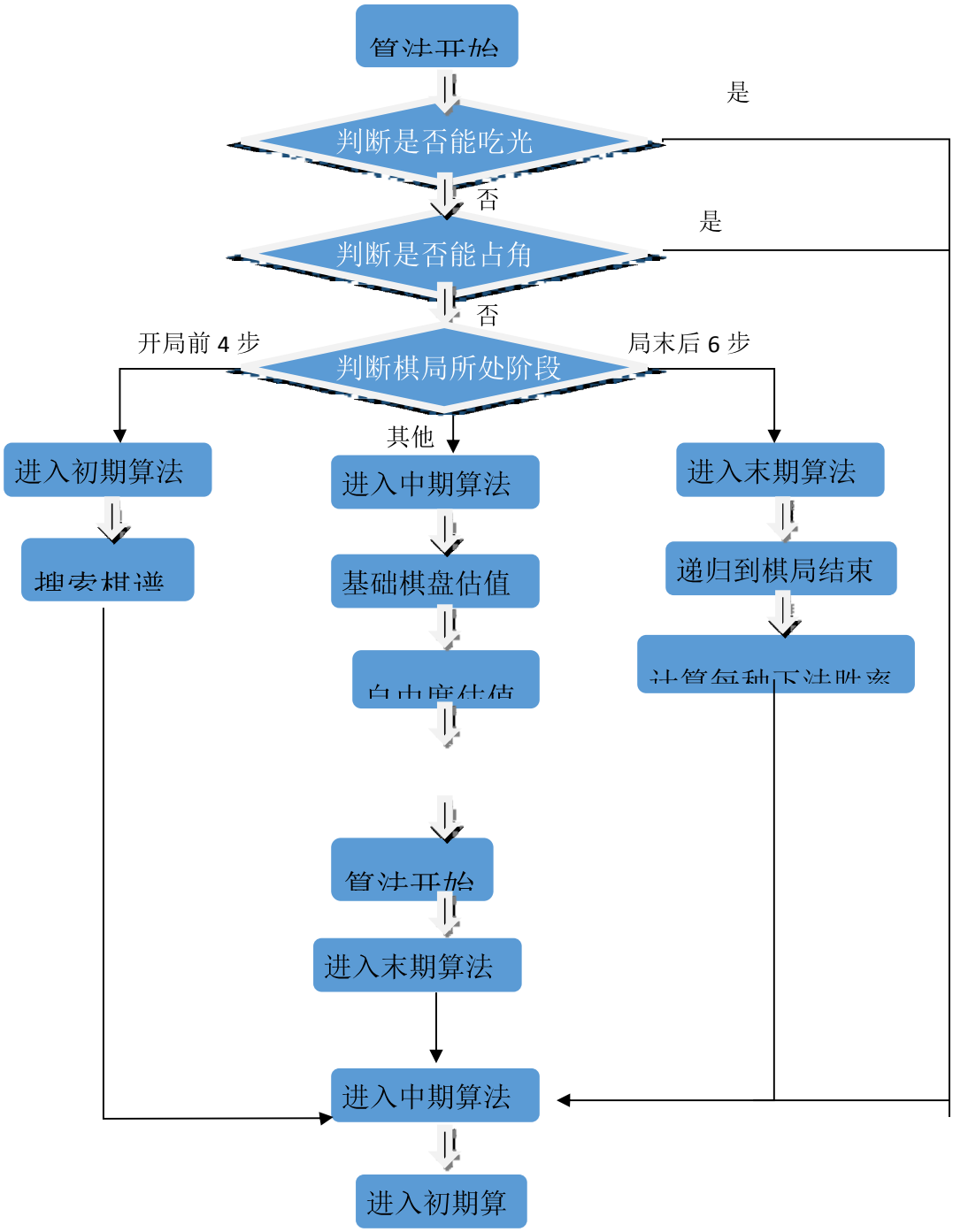
中期用多参变量估值函数选定最佳落子点，这些估值函数包括：棋盘基础位置、每步吃子、下棋自由度、边缘度、稳定度等。

棋盘基础位置：先建立基础位置估值表角是好的，临角的点不好，其他点都视为 0，每一步棋都会查看该位置的基础位置估值表，返回估值。每步吃子估值：测试每一步的吃子多少，由于高手下棋在中盘会尽量少吃子，减少自己的边缘子，所以吃子越多返回估值越低。对方自由度估值：测试每一步留给对方可下子位置数量的多少的估值，对方可下子位置越少，估值越高。边缘度估值：这项函数并不是严格的，原理在于统计每一个己方棋子周围八个点处的空白和对方棋子数，然后用对方棋子数减去空白数，这样的出来的值越高说明己方棋子被对方包围，处于稳定状态，是优势。

这里用到的主要数据结构是列表，自定义类有“棋盘节点”，用到排序算法。

末期算法目的是完成在末盘阶段的收尾问题，实现收盘阶段的最优化。算法策略是在末盘阶段进行穷举，对每一种可能的结局进行比较，根据最优的结局来选择这一步的走法。具体实现方法是：调用递归函数，每走一步都调用函数列举到最后一步，记录每种下法下面的胜率，穷举结束之后返回胜率最高走法。

3.1.2 算法流程图


算法开始
判断是否能吃光
是
否
判断是否能占角
是
否
判断棋局所处阶段
开局前 4 步
局末后 6 步
其他
进入初期算法
进入中期算法
进入末期算法
基础棋盘估值
递归到棋局结束
算法开始
进入末期算法
进入中期算法
进入初期算

3.1.3 算法运行时间复杂度分析

注：复杂度评估中，n 是己方总子数，k 是可落子位置数

前期算法由于事先准备了数据库，且算法不太复杂，对于目前的 4 步的情况，测试结果也证明，时间复杂度可以认为是 O（1）。

中期算法：中期算法主要由许多估值函数完成，各个估值函数的时间复杂度见 2.2 函数说明。总的来说，中期算法的复杂度为 O（k*n^2）

末期算法：由于末期算法为递归算法，且随着递归深度增加，可落子位置不断减少，算法的时间复杂度不容易估计。若设搜索深度为 m，可行域为 N（m），则算法的时间复杂度为 O（ N（0）*N（1）*……N（m）），估计为 O（k^m）。

总算法时间复杂度：由于末期的递归占了大部分时间，算法的总时间复杂度约为 O（k^m）。

3.2 程序代码说明

3.2.1 数据结构说明

一、数据结构：

1、线性数据结构列表

列表在 Python 算法中的应用是广泛的。在我们的算法中，列表主要用于描述棋盘，给每个位置赋值以及列举可行域等。

2、树

（1）嵌套列表记录棋谱

由于棋局前期的下法是“棋谱法”，需要用到“树”的数据结构进行每一步棋的下法记录。即以原始棋盘为“根节点”，每增加一步棋，数的高度就增加一级，下一步的每个走法就是对应一个棋盘“父节点”的“子节点”。在实际使用中，开始考虑直接采用树的数据结构（自定义类），但实现用树需要写的代码比较多，过于繁琐而且也不用实现往回倒的步骤。因此就采用了最简单的列表嵌套来实现，即只需要完整的“树”形数据记录即可。事实上这样也确实节省了时间。

（2）节点链接“棋盘搜索树”

此外，在中期算法，还用到“棋盘搜索树”。该树是通过对棋盘节点的连接来实现的，用于在盘中往后进行一定深度的搜索。

二、自定义类：

1、棋盘节点（ADT cbNode）

自定义类棋盘节点设置内置参量棋盘、可行域、下一步棋盘、节点估值、最优节点、当前节点深度。内置如下函数：①setNext-找出所有下一步棋盘，并记录；②setEva-设置评估值；③setDepth-设置深度；④getDepth-获取深度；⑤getEva-获取评估值；⑥getBestnext-获取下一步最优解（单步评估）。

函数的主要功能是为了棋盘搜索树的实现。

3.2.2 函数说明

注：复杂度评估中，n 是己方总子数，k 是可落子位置数

1、前期算法函数

（1）、对称处理函数

①def linesymmetry(i):　作主对角线的轴对称

②def centralsymmetry(i):　做中心对称

两个对称函数以棋盘的某个位置(i)为接口，目的是解决棋谱记录时，由于对称性而能够简化的情况。具体算法为根据表示棋盘某位置的字母或数字，获取对应轴对称或中心对称的字母和数字。

复杂度 O（1）

2、中期算法函数

（1）def buildST(root,depth):　棋盘搜索树函数

用于在中期对棋局进行有限步的递归探索，接口为棋盘某一下子位置根节点(root)与递归深度(depth)。主要算法是递归。

如果深度为 1，设置根节点下一步节点，设置根节点深度，设置根节点估值

如果深度大于 1，设置根节点下一步节点，对每一个下一节点递归调用，设置根节点深度，设置根节点估值。

（2）def search(root):　基于棋盘节点的递归搜索估值函数

用于在中期对棋局进行有限步的递归探索后的估值，接口为棋盘某一下子位置根节点(root)。主要算法是递归与假设对方下法的贪心算法。

1. 搜索到末端节点，返回估值
2. 轮到对方下，假设对方会走对他最有利的一步，返回对对方自由度的估值，和下一步对自己的评估
3. 轮到自己下，对每一子节点，把每一步的评估值作为该步总评估值，获取最优解，返回最优解估值

复杂度大致分析，设搜索深度为 a，则复杂度大约为 O（(k*n）^（a/2））

（3）def test(cb,pos):　落子后棋盘函数

模拟一步落子，返回落子后的棋盘。接口为棋盘(cb)和落子位置(pos)。

（4）def getcolor(cb):　颜色获取函数

获取己方颜色，黑方为 0，白方为 1。接口为棋盘(cb)。

（5）def getbasewt(cb):　基础棋盘评估函数

建立基础棋盘权重表，计算当前棋盘局面的“总体得分”。接口为棋盘(cb)。主要用循环累计每一个位置的分数。

步骤

1. 建立棋盘基础权重值表，为每一个位置配权重
2. 遍历当前棋盘，记录己方落子位置的权重和（也可以换成对方角度记录，求负值）
3. 乘以配重系数，返回评估值

复杂度

只遍历一遍棋盘上的棋子，为 O（n）

（6）def getoplp(cb): 自由度评估函数

计算留给对方自由度的权重值，对方自由度越高越不好。接口为棋盘(cb)。自由度就是对方能下的位置个数。

步骤

1. 对准备落子位置做测试，模拟落子后的棋盘，求对方的可落子位置作为对方自由度评估
2. 乘以配重系数，返回评估值

复杂度

该算法需要调用黑白棋内置 getLegalPos（复杂度估为 O（n）），则总复杂度为 O（k*n）

（7）def getborder(cb): 边缘度评估函数

计算边缘度权重，大致能体现是己方在外围还是对方在外围。接口为棋盘(cb)。通过计算和己方棋子相邻的对方棋子总数和空白区域总数来判断。函数需要配合边缘度评估辅助函数使用。

步骤

1. 对每一个己方棋子，扫描邻近的八个格子中的对方棋子数和空格数，分别求和
2. 用对方棋子数减去空格数，得分越高，说明己方棋子在内部，处于稳定状态
3. 乘以配重系数，返回评估值

复杂度

遍历棋盘棋子，每个棋子查询 8 次，为 O（8n），即 O（n）

（8）def getsurround(board,r,c): 边缘度评估辅助函数

辅助 getborder 函数，计算每一个棋子的相邻情况。接口为棋盘(board)、行(r)、列(c)。

（9）def getstableedge(cb): 稳定边评估函数

获取稳定边权重系数。接口为棋盘(cb)。分别计算上下左右四条稳定边的稳定度。函数需要配合稳定边评估辅助函数使用。

步骤

1. 对准备落子位置做测试，模拟落子后的棋盘
2. 统计目前棋盘上的己方棋子数，记录棋子位置
3. 对每一个对方准备落子位置做测试，模拟落子后的棋盘
4. 统计对方落子之后己方剩余棋子位置，再返回统计下一个对方可能落子位置
5. 循环结束后，统计不会被吃掉的子个数
6. 乘以配重系数，返回评估值

复杂度

该算法需要对每一个落子后棋盘的每一个己方棋子进行扫描，复杂度为 O（kn）

（10）evaedge(edge): 稳定边评估辅助函数

辅助计算某一条边的稳定性。接口为某条边(edge)。算法为遍历某条边，计算己方连续子、最长连续子、对方子等。

步骤

1. 采集四条边非角位的落子情况
2. 如果有对方子，则不计算稳定边
3. 如果没有，遍历并记录最长连续子情况，根据边位稳定权重求评估值
4. 乘以配重系数，返回评估值

复杂度

只扫描固定几个位置的棋子，故为 O（1）

（11）def getstablechess(cb):　　稳定子评估函数

用于获取稳定子位置。接口为棋盘(cb)。遍历棋盘，计算稳定子总数。

（12）def geteat(cb,tb):　吃子评估函数

用于吃子评估。接口为棋盘(cb)、落子(tb)。用现在的某方子数减去下子后某方子数即为这一方被吃子总数。

步骤

1. 对准备落子位置做测试，模拟落子后的棋盘，求出吃子个数
2. 乘以配重系数，返回评估值，在前期吃子越少评估值越高，后期越多越高

复杂度

该算法对每一个可落子情况进行棋盘遍历，复杂度为 O（kn）

（13）def gettotalwt(cb):　　总评估函数

根据每一项权重和其权重系数得出该状态棋盘总评估。接口为棋盘(cb)。分别对自己和对方立场估值。此函数需要配合以上所有“评估”函数使用。

复杂度

大约为 O（k*n^2），但是考虑到 n 和 K 的数量级都比不上 64 个子的棋盘，所以实际消耗时间和用该复杂度估计出的时间比率会有较大差距。

3、末期算法函数

（1）def doNext(cb,myturn):　穷举递归函数

对剩余的所有走法进行穷举，走结尾时在所有的局数中可能赢的概率最高的那一步。接口为棋盘(cb)、(myturn)。取可以走到的每一个位置，对每一步分别考虑，一直递归到棋局结束，计算获胜棋局在总棋局中的比例，返回比例最高的一步。配合穷举递归辅助函数使用。

（2）def Next(cb,myturn,step):　穷举递归辅助函数

每次往下走一步，直到结束。接口为棋盘(cb)、(myturn)、递归深度(step)。主要目的是帮助穷举递归函数进行相关计算。

4、主函数

def play(cb,ms):　　黑白棋主函数

函数接口为棋盘(cb)、(ms)，判断棋局进展后调用相应时期的算法，完成落子选择。

3.2.3 程序版本说明

1、Mike67origin

带有棋盘基础位置权重评估

带有边缘度评估

带有单步吃子能力评估

带有对方自由度评估

2、Mike68test

带有棋盘基础位置权重评估

带有边缘度评估

带有单步吃子能力评估

带有对方自由度评估
带有稳定边评估
带有单步斩杀评估（因为有一次和中级电脑下，明显能一步吃完对手，结果没有那样走）
3、**Mike612**
带有棋盘基础位置权重评估
带有边缘度评估
带有对方自由度评估
带有稳定边评估
带有单步斩杀评估
带有初级末盘穷举求最优算法（末期算法）
有用于多步搜索中盘优解的 ADT 棋盘节点和根据该节点建立搜索树的建树函数
有适应已建立的搜索树的递归多步搜索函数
4、**Mike615wb**
带有棋盘基础位置权重评估
带有边缘度评估
带有对方自由度评估
带有稳定边评估
带有单步斩杀评估
带有高级末盘穷举求最优算法（末期算法）
带有简单稳定子评估
带有自建棋谱数据库，以及适应的开盘按棋谱搜索落子算法（前期算法）
有用于多步搜索中盘优解的 ADT 棋盘节点和根据该节点建立搜索树的建树函数
有适应已建立的搜索树的递归多步搜索函数

这是我们程序的四个基本版本，其中，**Mike615wb** 是最完整的版本（未在比赛中使用）。**Mike612** 带有完整的中期算法以及不太成熟的末期算法，算法较简单，但热身赛测试表明胜率较高。**Mike67origin** 和 **Mike68test** 是中期算法。在比赛中，我们用到了 **Mike68test**、**Mike612** 和 **Mike615wb** 的不成熟版本（末期算法思路有误）。结果，综合来看，**Mike612** 的表现最佳。

3.2.4 算法成长日志

1、前期算法

一开始考虑的是前期的走法变化较少，且基本上已经形成了面对不同棋局的最优开局，因此前期的算法不打算靠人脑来编写，而是选择通过使用高级的电脑对战来确定前期开局的走法，既确保开局不会吃亏，又能够节省大量的时间为后期服务。这个工作不烧脑子，但是比较烧时间。当时我们的设想是前期能够走的变化不是很多，而且通常被人们选择的走法就那么几步，所以初步定的是 10 步。

遇到的问题：

1.初期没有想如何使用数据结构，因为我觉得这个工作比较像“搬砖”，只要将资料搜集完备，通过很简单的数据结构便可实现。于是便采用了用多个网站的最高难度进行对下，但搜集了一段后，便发现一个重大的问题，尽管我们可以通过高级的电脑选择出来最优最普

遍的走法，但是若其他小组采取从头开始便搜索加估值的算法，便很难保证开局仍在这几种模式中，这便要求我将所有的可能性均列举出来，因此数据量比起最初估计便有了爆炸性的增长基本上会变成 4~5 的指数，若仍要像之前一样做 10 步就会变为 4^10 的数据量，这靠人力来完成显然是不可能的任务，于是我们决定将步数改为 4 步，这还是一个可以用人力完成的范围内的数据量（然而四步对于棋局的影响很小，最终证明这样做的作用大概也就能够在开局剩下 5-6 秒的时间，还不如让开局把这几步下了，让我加入搜索预估值的部分去打打工协助一下）。

2.既然要选考虑到所有的可能性，变不能够再采用黑白棋电脑对下的办法，需要用一个能够无限步悔棋的软件把所有情况试出来，于是采用了 Wzebra 解决这个问题。

3.搜集完了数据，还需考虑如何实现，最初我们考虑的是通过判断棋盘上的棋型来决定下一步，以简化之前步骤不同但当前面对的棋盘相同的状况。但这样做的缺点在于仅仅四部就可以涉及到整个棋盘的范围，因此每次都要对整个棋盘进行扫描，并再做出棋盘的数据库，耗时又浩空间，所以便决定了每次都记录下上一次的走法，引出不同的分支，下一次仅需在这个分之下搜索即可。确定了这个方案后初步考虑的是 采用树，但实现用树需要写的代码太多，过于繁琐而且也不用实现往回倒的步骤，因此就采用了最简单的列表嵌套。事实上这样也确实节省了时间，前四部每步不到 0.001s，（但事实上前四步对通盘影响太小了，随机下都行）

结果：最终的程序做出来后对胜率的影响相较于通盘采用估值相比没有显著的提升，而且存在一个数据疏漏，但这个数据疏漏很罕见，数百次测试才在临赛前出现 2 次，临赛前也没有查出问题所在并加以修改，但在比赛中第一轮就出现了一次，之后前期代码便被砍掉并全部有中期的估值下法替代了。

2、中期算法

6 月 7 日：估值函数的编写

先考虑了四项估值权重：棋盘基础位置估值，每步吃子估值，留给对方自由度估值，边缘度估值。

成果：Mike67origin

6 月 8 日：估值函数的改进

增加了稳定边这一项估值权重

稳定边即某一边上连续己方棋子有 4,5,6 个的边

在稳定边计算中，查询得知有一条稳定边是好棋，而超过一条反而不好，所以在算法里计算了四个边的稳定性之后判断，如果有一个边稳定，那么估值高，没有或者超过一条边则较低重新平衡了各个权重系数，降低了基础棋盘位置权重。

成果：Mike68test

6 月 9 日：搜索函数的添加：

先测试了最简搜索：三步之后的最优值，发现不及之前为搜索的算法，决定升级搜索至最大最小搜索

6 月 10 日：搜索算法的改进：

采用类似最大最小搜索的算法，先从 3 步搜索开始

探索奇数步，轮到自己搜索时，搜索下面步法中搜索到最深时最好的走法
轮到对方搜索时，默认对方选择下一步最有利的走法
在测试中有一次单步 5S 超时，但在之后的测试中，10S 就能走完一局
修改了稳定边权重系数
修改了棋盘基础权重值
采用了自定义类型树搜索法实现了搜索算法，在短期测试中其结果与无搜索时棋力相当。
一局一般 30S，并可以增加搜索深度

6 月 11 日：估值函数的改进：
删除了吃子多少的估值函数
注：静态估值：对某个棋盘棋型进行评估，如基础位置，边缘度，稳定边
　　动态估值：对某一步将带来的影响变动进行评估，如自由度
成果：Mike612

6 月 13 日：估值函数的改进与估值过程的优化
改动棋盘基础位置估值为自己减对方
精简了估值函数内部调用分析过程，估值速度提升约 40%
强化了不同情况下函数的衔接接口

6 月 15 日：加入了强制占角判定，一旦能占角就占

3、末期算法
6 月 10 日
　　基本依据算法目的和算法策略完成递归函数，并实现了对接。但经过测试之后发现效果并不明显，与未添加末盘算法的程序先后手互有胜负，对比先后手的比分也没有明显优势，没有实现末盘阶段的最优化。经过多次测试无 bug。
6 月 11 日
　　与 glamber、idiot 进行测试，效果（胜率和比分）与未添加末盘算法的程序比较无明显改变，经过反复检查未发现程序有错误。多次测试无 bug。
6 月 12 日
　　经过调试发现在递归过程中传参出现了问题，参数出现了篡改的情况。反复修改后没有解决参数篡改问题。于是减少了传参数目，改变算法思路，只记录达到最优比分的第一步走法。经过测试发现修改后效果不明显，仍然未能实现末盘阶段最优化。多次测试无 bug。
6 月 14 日
　　调试发现 return 语句位置不对，会使函数在未递归到最后一步就结束，反复修改后发现无法找不到 return 语句的正确位置。于是舍弃 return 语句，改用全局变量来进行记录达到最优结局的第一步。修改后经过测试比未添加末盘算法的程序更加优化，先后手均胜。测试过程中出现 bug，反复调试无结果。
6 月 15 日
　　发现对第一步的处理与递归函数无法完美兼容，增加一个函数将第一步单独拿出来进行处理，在处理完之后在进行调用递归函数。修改后能实现正常功能，测试结果无 bug。进行热身赛发现测试结果不理想。

6 月 17 日
发现算法策略出现问题，现在的算法策略是假设对手是最傻的走法，从而被引导向对于我方最优的结局落子。将所有结局按照双方的可能落子情况画出一棵树，树的末枝就是所有可能的结局。当最优结局的周围几个结局明显不利与我方时，当前的末盘算法就会起到不好的作用。所以当前的末盘算法策略没能实现算法目的，达到末盘阶段最优。经过组内讨论，修改为引导对手向所有可能结局中我方胜率较高的那一分支落子。修改后测试效果有略微改善，反复测试无 bug。

3.2.5 程序限制

我们的程序在测试过程中出错率约为 0.2%，且错误大多处在棋局前期，可能是因为前期采用棋谱算法，棋谱在人工录入的过程中，对于一些少见的下棋情况可能没有做到完整记录。后来经过对出错棋局的复盘，发现是程序中一个数字打错，已解决。（不排除还有其他错误）

中期、后期算法基本没有出过错误。中期算法用到棋盘搜索树，可能在面对“PASS”现象时没有考虑得太清楚，可能会出现错误。

3.3 实验结果

3.3.1 实验数据

一、**MIKE612** 测试数据
实验环境说明：

- 硬件配置：

处理器:	Intel(R) Core(TM) i5-4200M CPU @ 2.50GHz 2.50 GHz
安装内存(RAM):	4.00 GB
系统类型:	64 位操作系统，基于 x64 的处理器

- 操作系统：Windows8.1 中文版
- Python 版本：Python 2.7.9

1000 次对弈结果数据汇总（612 算法）

黑 \ 白	idiot 算法	gambler 算法	本算法（612）
idiot 算法	胜 0 负 1000 平 0 27000:37000 19.1872s: 26.3212s	胜 526 负 426 平 48 33528:30358 12.3417s: 10.2104s	胜 0 负 100 平 0 2900:3500 0.810036s: 2895.61s

gambler 算法	胜 426 负 534 平 40 29392:33960 13.7979s: 17.0572s	胜 453 负 503 平 44 31784:32213 12.0722s: 12.0201s	胜 100 负 882 平 18 21154:42144 8.53008s: 31587.5s
本算法（612）	胜 100 负 0 平 0 3500:2900 2755.68s: 10.3546s	胜 874 负 86 平 40 42186:21262 46668.9s:13.3202s	胜 100 负 0 平 0 3300:3100 2292.03s: 2378.16s

二、**MIKE612** 与 **MIKE615wb** 对战数据

实验环境说明：同（一）

次对弈结果数据汇总

黑 \ 白	MIKE612 算法	MIKE615wb 算法
MIKE612 算法	见（一）	胜 41:23 27.3232s:32.6288s
MIKE615wb 算法	负 21:43 59.4468s:21.4008s	见（三）

由于算法都没有随机性，测试各进行 50 次，结果全部相同，记录一次的数据，时间为 50 次平均。

三、**MIKE615wb** 测试数据

实验环境说明：

- 硬件配置：

- 操作系统：Windows8.1 中文版
- Python 版本：Python 2.7.9

1000 次对弈结果数据汇总（615wb 算法）

黑 \ 白	idiot 算法	gambler 算法	本算法（615wb）
idiot 算法	见（一）	见（一）	负 22:42 0.012839s: 49.8263s
gambler 算法	见（一）	见（一）	胜 112 负 868 平 20 21560:42353 17.3974s: 72660.4s

本算法（65wb）	胜 48:7 276.788s: 0.010669s	胜 828 负 140 平 32 40703:21758 119617s:18.12614s	胜 41:23 126.519s:104.115s

3.3.2 结果分析

一、**MIKE612** 算法：

策略 1：根据落子坐标确定权重值一，角位的权重值高，星位和角的邻位权重值低，边的权重值相对高一些。

作用：使得占角和获得稳定边的概率大大提高，确实起到了效果。

策略 2：增加了稳定边的判定

作用：有利于获得稳定边，然而这个判定和棋盘坐标判定有一定重复，从实验结果上难以判定其效果强弱。

策略 3：增加了对方自由度的判定，优先选择了对方自由度低的选项。

作用：限制了对方的落子位置，但在和 idiot 和 gambler 对弈的过程中，由于对方算法的特殊性，自由度判定的作用不如与更复杂算法的明显。

策略 4：设定了边缘子判定，用来检测对方的子和自己的子哪个更趋向于外部

作用：使得自己的子趋向于被包围，增加了稳定子的个数，有利于己方选择枝的扩展。在实验中效果较为明显。

策略 5：通过最小最大原则进行了 3 步搜索，综合取落子点

作用：由于对方算法的特殊性，最大最小原则并不适用，但搜索算法还是减少了极端情况的发生。

时间预期：

时间花费非常稳定，基本保持在平均值，并没有极端的时间花费出现。

时间的开销主要花费在：

1.搜索棋盘，累加各个点对棋形总估值的影响。其中由于各个落子点都要进行计算，因此花费了一定的时间。

2.搜索算法，搜索的每一步实际上都是对搜索和估值的重复计算，因此增加了时间的花费。

3.其余步骤基本花销不大。

二、**MIKE612** 与 **MIKE615wb**

测试结果表明，612 无论是先手还是后手，都“完胜”615wb。但其实，后者所考虑的是更加全面的，且“号称”末期算法是“最优”的。对于这样的现象，可能的解释是：由于两个算法都不存在随机性，对战的结果是唯一的，而程序之间又可能存在互克的问题，或许是 612“天生”克 615wb 吧。当然，也有可能是我们的末期算法仍然有问题，甚至会导致下出一些比较蠢的棋。

三、**MIKE615wb** 算法

1、边缘子和对方自由度估值对限制对方合法棋步数目有一定作用。但是由于权重的原

因，经常出现走出的某一步产生边缘子数目和对方自由度都不是最小的情况，所以这两个策略并没有达到非常理想的效果。

2、强行占角策略完全实现了。角是棋盘上最好的位置，占角往往对进一步发展稳定子非常有利。虽然能占角就占角并不一定是最佳的，但可以说它是本程序众多策略中效果最好的一个，尤其是在对方下出坏棋时，直接占角往往是最佳应对。

3、稳定边估值主要是延长稳定边，同时避免对方延长稳定边，其重要性仅次于占角。这一策略实现得也比较好。但是，一方面，在防止对方发展稳定边方面还存在问题；另一方面，延长稳定边时有时消耗了自己的棋步。

4、事实证明，棋盘基础位置估值用处不大，且容易产生反作用，比如为了争 C 位（离角两格处）而产生很多的边缘子。

5、尾盘枚举式搜索算法过于耗时间，而且从原理上讲也不太科学。有一定的效果，但不一定能找到最好的落子点。

6、一个思想上存在的问题是，我们是用总步数来判断各估值函数的权重的，总的棋步数目和权重的关系含有很大的“蒙”的因素。另外，可能不应该将各项估值函数的权重糅合在一起，而应该确定一个优先级（先单看哪一个因素，再看另一个因素），这样就不会出现如基础位置的影响超过边缘子的情况。

7、算法的运行时间不总是在预期之中，尤其是先手对 idiot 时。主要时间开销在接近终盘（52 步以后）时，因为尾盘算法要搜索所有可能的情况。

3.3.3 经典棋局

这一部分，针对表现一直不好的 615wb 算法，拿出一局与“赌徒”的对弈进行分析，详情可见附件。

3.4 实习过程总结

3.4.1 分工与合作

一、分工：
黄如许：棋局前期算法
闫述辰：棋局中期算法
肖万博：棋局后期算法
余晓辉、高鸿宇：资料搜集、代码测试
温景充：实习报告汇总、算法人机测试

二、合作交流方式：
1、微信群基本联系
2、QQ 讨论组一般联系、算法交流、问题探讨、文件共享、任务布置
3、集中讨论

三、会议记录：
第一次会议：
时间：2015 年 6 月 5 日（星期五）晚上 19:40-20:00
地点：燕园 40 楼 138 房
参与人员：全体
会议主要内容：根据老师建议及实际情况确定分工方式，每个人发表对黑白棋的基本看法
会议结果：确定分工方式、明确当日任务：熟悉黑白棋，搜索相关资料

第二次会议：
时间：2015 年 6 月 6 日（星期六）下午 15:00-16:30
地点：燕园 40 楼 121 房
参与成员：全体
会议主要内容：
1、每个人分享搜集的资料和对黑白棋下法的认识。
2、完成分工，确定每个人的基本任务（肖万博、闫述辰、黄如许负责代码部分，余晓辉、高鸿宇负责资料搜集和测试，温景充负责实习报告）。
3、讨论下法，初步确定根据棋局的不同阶段部署不同的算法，棋局前期按照棋谱下棋，中期用多参变量估值函数选定最佳落子点，棋局后期用搜索寻求最佳位置，并分别由黄如许、闫述辰、肖万博完成三个部分。
4、讨论棋局不同阶段的划分方法，初步确定，前期为 10-12 步，中期为 13-58 步，后期为 59-64 步。
5、确定基本时间规划
会议结果：明确分工，明确基本算法思路

第三次会议：
时间：2015 年 6 月 14 日（星期天）晚上 22:00-23:30
地点：燕园 40 楼 138 房
参与成员：全体
会议主要内容：
1、针对当前算法不理想的情况，分析原因，发现之前的棋局后期算法思路出现一定问题，在往局终搜索的过程中，只考虑到了最佳情况，并往那个方向下棋，但未考虑到对手的下法而导致的坏情况，也就是说，算法不太科学。并讨论解决方案。
2、讨论其他代码可能存在的问题及解决方案，确定在最后阶段以优化参数为主要目标
会议结果：发现代码思路问题并尝试解决

第四次会议：
时间：2015 年 6 月 15 日（星期一）傍晚 18:00-18:30
地点：燕园 40 楼 138 房
参与成员：全体
会议主要内容：
1、总结比赛中出现的问题。讨论算法出错的可能原因。

2、讨论关于实习报告的内容。确定接下来的工作主要是算法测试和实习报告的完成，并初步分配了任务。

第五次会议：
时间：2015 年 6 月 21 日（星期日）傍晚 19:00-19:30
地点：燕园 40 楼 138 房
参与成员：全体
会议主要内容：
1、讨论在测试中出现以及要注意的问题，根据老师的最新指示明确测试中要保留的原始数据。
2、明确完成实习报告的时间节点。

3.4.2 经验与教训

关于经验与教训这部分，我们打算以每个成员的个人感受方式呈现。

以前我对棋类 AI 的机制毫无理解，为了这次黑白棋算法作业，我上网阅读了大量黑白棋的棋手技巧，从中主要获得了一些直观对棋局的评估参量的写法，这对我接下来设置多个不同参量的棋盘评估函数起到了重要的作用。我也和很多 AI 对弈来从 AI 的角度寻找 AI 程序对棋局的评估判断方式，更重要的是 AI 所采用的搜索方式，比如说最大最小搜索，剪枝搜索。

接下来就是开始尝试编写初级 AI，我认为，AI 优先应该拥有准确简洁的评估函数，这是第一位的，所以我一开始先写的是评估函数。在写评估函数的时候，我参照了老师在写 chessboard 的时候的一些技巧。在没有搜索，只有单步评估的情况下，初始版本的运行速度是极快的。

随后，在不断完善评估函数评估准确的同时，我加入了搜索算法。由于我对递归函数了解的不够深刻，所以在一开始直接编写递归搜索的时候遇到了极大阻力。我在一开始的递归函数里尝试了队列、栈等基本数据结构来有序地储存参量，但是仍然无法理清递归函数传递参量的问题。之后我开始尝试用自建抽象数据类型解决问题，设计了棋盘节点，同时根据棋盘节点用递归方法建立了搜索树，再通过树节点的深度标识来进行有序递归搜索，这样，关于每一个棋盘的详细信息就可以用 class 形式来记录保存。

最后，是调试各个评估函数的权重的问题，为此，组内的所有同学都对测试参数调整参数做了不少的努力，我们努力地改动参量使评估函数更均衡更准确，能对局势做出更好的判断。

遗憾的是，我们的努力并没有在最后的大赛中获得好成绩，我作为中盘算法的编译者感到很难过，也很自责，因为自己设计的函数经过我们大家的努力并没有特别好的成效。在编译中我获得了不少 AI 经验，主要就是以上列举的探索过程，都让我对数据结构与算法有了更深层次的认识，重要的是尝试了自建数据结构类型来创造更适合自己的搜索条件，这都是这次大作业给我的宝贵经验。当然，这也给我很大的教训，让我更加认识到自己对递归、动规的认识仍然存在严重的不足之处，以至于浪费了大量时间也无法写出精准的搜索函数，还有就是，小组内的集思广益总是有益的，而我们组在分工上的确有一些不利之处就是我们把算法设计集中在了少数几个人身上，这样使得我们的算法有一定的局限性，而没有考虑到更

多的情况。以后的大作业中，我认为我们更应该和其他同学更好地分工以做到更好的思维交流。

——阎述辰

首先，我对黑白棋这个游戏不太熟悉。作业的任务下来之后，我花了一些时间去“研究”黑白棋的玩法。但这个看似简单的游戏其实还是挺复杂的。所以，玩了一段时间都没有总结出一些相关的规律。在这样一种情况下，再去设计算法的时候就只能够通过查资料和一些比较简单的方法，比如占角等。还好队友还是比较给力的，我们还是能够设计出有一定智能的算法。

但是，总的来说我们普遍还是不太熟悉，所以，程序设计好之后，恒量算法好坏的办法就只有与随机算法对下，但这也很难恒量出水平。这是我们在实习过程中的一个不易解决的问题。

另外，我们在合作的过程还是挺顺利的，但交流时间还不够，导致有些算法思想不能实现或不能正确实现。比如，提供思路的人和写代码的人可能由于沟通不足，导致算法不能完全体现思想，甚至有时候起到“反效果”。由于我们一开始分工是就是将算法分成三块，分别开发，在开发过程中负责三个部分的程序员交流也不太够，导致算法整合的时候也会出现一些问题。

我感觉，我们能够合作但还是不够团结。而且我们对时间的把握都不太好，经常是到最后的时间才完成任务，导致不够时间解决一些出现的问题。这种情况在以后的合作任务中要注意。

总的来说，非常感谢靠谱的队友完成了算法和测试部分，当然也感谢老师安排了这样一个有意义的实习活动。

——温景充

这次实习作业我感觉大家还是很团结的，尽管由于在开头的时候由于对难度估计出了一些差错导致在编程的时候我出了不少问题拖了后腿，但是小组里其他成员还是来帮我补了漏洞。其实在这次初分工的时候，我觉得按阶段来分还是很合理的，因为这三个阶段的确可以采取不同的思路预与法，而且这样分可以使我们编程人员并不要了解通盘是怎样运行的，只需考虑自己的部分，减少了很多的固定成本。但是真正在实现的时候，就发现这样要求每个编程人员要有很强的个人能力，要能完全自己完成自己的部分，在出错的时候也必须自己调试正确，因此我们也浪费了一些的时间。

这次暴露出来的最大问题我觉得是时间节点的安排，因为我们是分开做的，所以觉得只要最后拼起来就行了。之前的没有加前期后期算法，只用中期搜索加估值的算法去参加热身赛，获得的结果还不错。但是第二次加上前期与末期后效果不是很理想，最后大家在最后一天熬到很晚去调试中期的参数，但由于对参数的意义除了中期的同学，大家都不是很了解，导致最后一晚上虽然用了很多的时间，却也没有太大的意义。另一个时间安排的失误体现在我负责的前期算法在比赛前一个小时才被测出一个 BUG 出现，但却没时间查和改了，导致在比赛中也暴露了出来。

总的来说这次实习作业还是有很多收获的，尤其小组同学们都很团结，我这个拖后腿的也得到了很多帮助，再次对伙伴们说声谢谢。

——黄如许

整个团队还是很团结和努力的，但是在沟通上还存在一些问题，团队会议上对算法的设计和最终的成品之中算法的思路存在了差异，导致最初的设想实际上并没有实现，这一点需要反省。

整体算法的思路合理，理论上有很高的上限，但是没有充分考虑到工作投入，前期的棋谱数据库的录入有一定的困难，而后期理论上的最优算法实现上有一定难度，本来设计的边做边调整并没有完全实现。

在各个因素的权重值取值中需要大量的实验，增加了大量的考虑因素之后，实验的复杂性大大提高，后来虽然有一定的删减，但是依然有困难。

整体而言，算法构建上有些复杂，上限很高但下限不够。

所幸小组分工明确，组员间关系融洽，团结一致，依然合作愉快。

——余晓辉

3.4.3 建议与设想

1、组队

组队的方式也许可以再考虑。小组合作的过程也是加深彼此理解的过程，因此，建议增加随机性，以便于不同班级甚至院系之间同学的交流，也会避免出现一些“不得已”而组队的现象。

2、基础设施代码

Reversi 函数在搜索一些名字的算法的时候似乎会出错，并且原因难以查明，重命名算法后解决。

3、竞赛

感觉小组赛的过程相对枯燥了一些，建议增加对弈的可视化。

这样的大作业对我们的学习还是很有帮助的，所以还希望在以后能有更多有趣的大作业实习项目，也希望能更多的在实习项目中实践提升。

3.5 致谢

虽然本次实习任务比较繁重，但是由于本小组各组员在本次实习过程中的积极参与和无私奉献，我们成功地开创了 Mike 系列程序并参加比赛。我们的算法是全体小组成员的共同智慧结晶，小组里的每一位成员都功不可没，在此对每一位有积极贡献的小组成员致以感谢。

首先要感谢的是我们组最辛苦、程序编写任务最繁重的阎述辰同学。由于阎述辰同学曾独立完成黑白棋棋局的复盘程序，对黑白棋的搜索等操作比较熟悉，而阎述辰同学自身编写代码的能力较强，对 python 的语法十分熟悉，所以他主动提议承担复杂繁琐的棋局中期算法。这种有能力又有责任心的组员是我们小组编写程序的核心动力。正是有了这样积极奉献的组员，我们的算法才顺利诞生。

接着要感谢的是我们组编写棋局前期代码的黄如许同学。如许同学的前期代码编写比较繁琐，需要反复测试来获取和记录数据，然后转化为编程语言。这一任务虽然难度上没有那么高，量却极大，而且十分考验人的耐心。如许同学在规定的时间内完成了前期代码的编写工作，使我们的算法在开局就能占据较大的优势，他的辛苦工作值得我们赞扬。

然后要感谢我们小组两位测试员高鸿宇同学和余晓辉同学，感谢他们在晚上 12 点钟以后还在进行测试工作，记录大量的测试数据，分析每一个最新版本算法的优点和缺陷，为我们的程序寻找 bug。他们的工作为编写代码的人提供了改进程序的方向，使我们的程序越来

越优化，越来越成熟。我们小组的程序能达到最后最完美的程度，他们的工作贡献最大，在此对他们两人的辛勤工作和一丝不苟的态度表示衷心的感谢。

还有咱组不可不提的温景充同学，他的工作主要是负责记录我们小组的最新动态，完成实习报告。这一任务虽然没有编程和测试那么费心，但也同样劳神。温景充同学需要实时联系其它成员，询问最新的编程进度和测试结果，并反馈给大家。温景充同学就是我们小组内部信息的传递者，他将我们整个小组连成一个整体。我们小组成员之间迅速传递信息，实现高效的分工合作，依靠的就是温景充同学积极主动的参与。在此谨对温景充同学积极参与的精神表示感谢。

当然，我们非常感谢老师能够组织这样的活动，使我们能够团结在一起来完成这样一个任务，使得每个人都得到了锻炼。能够在课程中安排这样的比赛也大大增加可课程的趣味性，使我们能够了解到数据结构的实用意义和价值，对知识有更深的理解。也非常感谢老师给我们提供的指导和硬件、软件支持。非常感谢！

我们的算法是大家共同努力的结果，缺少了任何一位成员的奉献都不行。因此，最后对每一位参加到我们小组算法的编写工作中的人和每一位热心帮助的人表示最真诚的感谢！谢谢你们的奉献！！！

——肖万博

3.6 参考文献

道客巴巴的黑白棋速成攻略 http://www.doc88.com/p-890573802914.html
WAI 人人主页：棋类 AI 基本知识 http://page.renren.com/601510260/note/871651717
申龙斌—剪枝搜索算法 http://www.cnblogs.com/speeding/archive/2012/09/20/2694704.html
阿宋黑白棋天地 （老师推荐）中文章 http://www.soongsky.com/computer/pvs.php
前期算法棋谱原始来源（黑白棋软件）http://radagast.se/othello/download.html

东区 East　ECHO 组

感谢逸夫二楼的 3430、3331、3425，四教 501，光华地下室，阳光大厅，Zoo Coffee 以及圣摩尔

带着电流与 WiFi 陪我们度过那些安静而不寂静的深夜

——ECHO

4 数据结构与算法课程实习作业报告

陈春含*，赵芳珩，吴逸夫，运乃丹，彭玉环，付帆飞

摘要：本组为 ECHO 组，所采用的基本思路是 alpha-beta 剪枝搜索博弈树与局面估值函数相结合。这是一种递归。参赛版本 ECHO4 的测试结果表明，它对阵 idiot 可以取得先手后手的全胜，对阵 gambler 胜率约 19:1。

关键字：博弈树； α - β 剪枝；估值；哈希表；栈

4.1 算法思想

4.1.1 总体思路

我们将参加比赛的算法命名为 ECHO4，总体上使用的是 alpha-beta 剪枝搜索博弈树与局面估值函数相结合的思路。另外有两个基于 ECHO4 进行更优化探索的版本 ECHO5 和 ECHOhash，但因时间有限，没有全部 debug 完成，性能尚劣于 ECHO4。故在此仅对 ECHO4 进行测试和详细分析说明，对于 ECHO5 和 ECHOhash，仅说明思路和当前版本代码。

4.1.1.1 ECHO4 的总体思路

博弈树总体来说是递归的思路，也即树的思路。如果不考虑 alpha-beta 剪枝，每次轮到我方下棋时，博弈树为每个合法位置建立一个子节点，假设我方下在该位置，对于每一个下过棋的局面，为对方的每一个合法位置再建立子节点，依次类推直到达到设定的递归深度 n。我们假设对方同样有智能，即每次都会选择对自己最有利的步法来下子，那么就可以依照博弈树的每一个分支判断到 n 步之后的局面。我们对这些局面进行评估，并逐层按一定规则（基

于对方智能的假设）反推回来，得到对我方较有利的基层分支并选择下在该处。

Alpha-beta 剪枝的作用在于去除博弈树中不必要的分支，即如果一个位置不可能被当前下子方选择，那么它就是无意义的，没有必要进入更深层次的递归来展开了。剪枝可以在很大程度上节省搜索时间，从而有利于进行更深层次的搜索。

估值函数的作用在于进行局面评估。这种评估是对于当前下子方而言的评估。函数返回一个数值作为评估值，对当前下子方有利则为正值，不利则为负值，优势越大值越大。（介绍下棋算法的总体思路，采用的主要数据结构与算法，采用的算法策略）

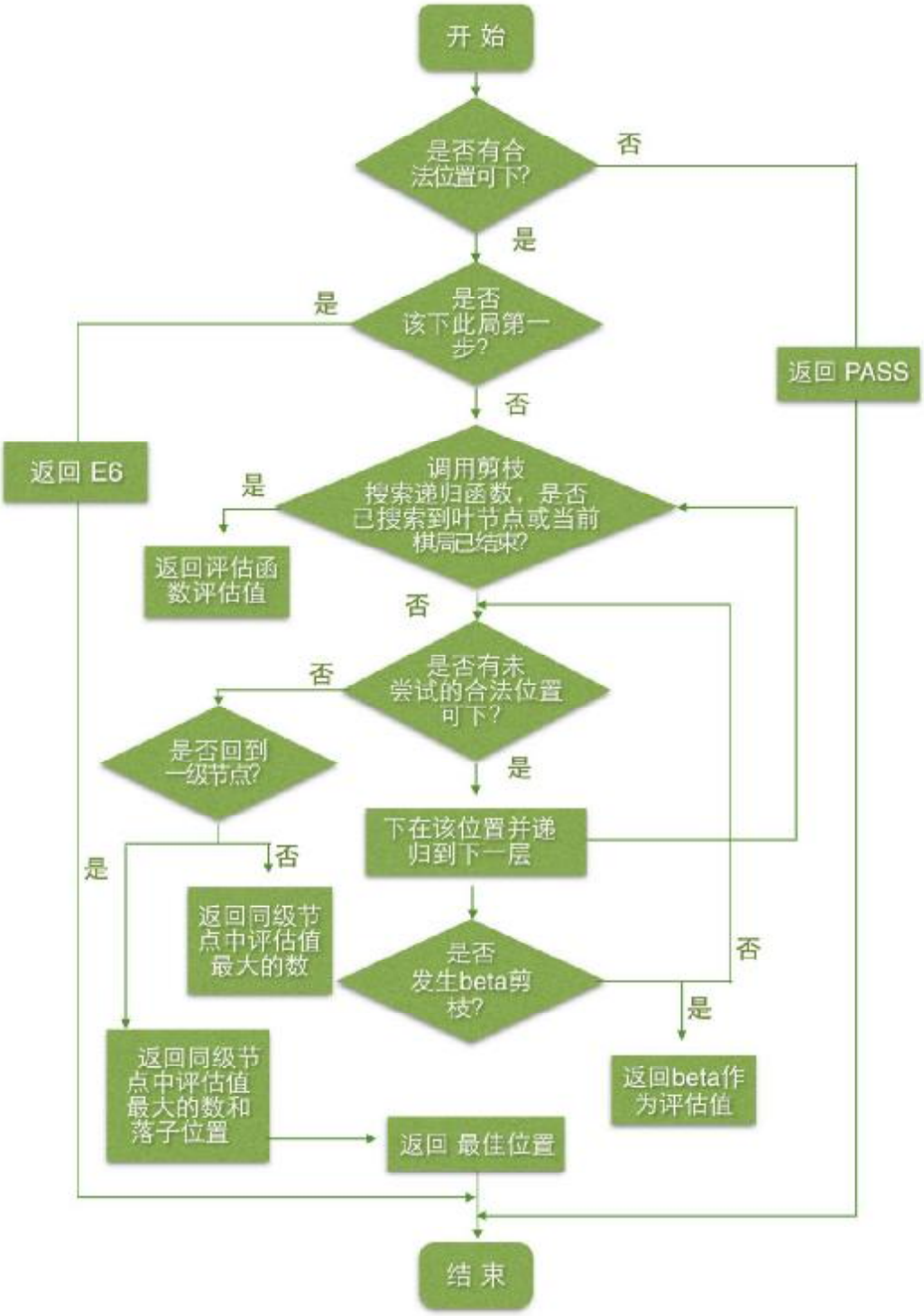
4.1.1.2 ECHO5 的改进尝试思路

ECHO5 相比于 ECHO4，主要在估值函数上进行了时间和功能优化，添加了一些考虑因素，并且使用栈的结构，尽量减少不必要的遍历和数据处理，力图减少估值函数所耗费的时间。

4.1.1.3 ECHOhash 的改进尝试思路

ECHOhash 的主要思想是基于 ECHO4 的功能，结合哈希表，即把遇到的局面做好估值之后存到表中，下次再碰到相同局面时便不必再做一次估值而可以直接查表返回结果。

4.1.2 算法流程图


开 始
是否有合法位置可下?
否
是
是否该下此局第一步?
是
返回 PASS
否
返回 E6
调用剪枝搜索递归函数，是否已搜索到叶节点或当前棋局已结束?
是
返回评估函数评估值
否
是否有未尝试的合法位置可下?
否
是否回到一级节点?
是
下在该位置并递归到下一层
是
否
返回同级节点中评估值最大的数
是否发生beta剪枝?
否
是
返回同级节点中评估值最大的数和落子位置
返回beta作为评估值
返回 最佳位置
结 束

4.1.3 算法运行时间复杂度分析

以 ECHO4 为对象，算法耗费时间主要由剪枝搜索和估值函数两部分组成。

4.1.3.1 剪枝搜索的时间复杂度

设 x 为分支因子，n 为深度，则一般情况下：

$$O(n,x) = 2 * x^{n/2} - 1 \text{ (n 为偶数)}$$

$$O(n,x) = x^{(n+1)/2} + x^{(n-1)/2} \text{ (n 为奇数)}$$

综合起来，可认为

$$O(n,x) = x^{n/2}$$

但剪枝操作的是相对不稳定的，假如一直不发生剪枝，那么剪枝搜索树就会退化为一般的负值极大搜索树，从而时间复杂度提高到 x^n。

4.1.3.2 估值函数的时间复杂度

估值函数相对简单，主要是遍历耗费时间。每一次运行估值函数，耗费的时间基本恒定，为 O（1）。

4.1.3.3 总体时间复杂度

综上所述，总体算法时间复杂度主要由剪枝搜索决定，平均为 $O(n,x) = x^{n/2}$.

4.2 程序代码说明

4.2.1 数据结构说明

ECHO4 中使用的主要数据结构为树结构，即博弈树。与课程中学到的树相比，主要的变化是加入了判断与选择功能。树的节点分为两层，即 max 节点（己方节点）与 min 节点（对方节点）交替分布，用来模拟双方下棋过程。各节点中保存的值为对当前下期方来说的局面估值和最佳落子点（在搜索回到顶层之前，最佳落子点设为 None）。

ECHO5 中除了树结构，还使用了栈结构。栈结构主要用在估值函数中，对应估值函数中的边角配置的多个功能部分，建立多个栈，将暂时符合条件的位置压入栈中，再进行后续判断或弹出等操作，做到一次搜索实现所有函数的功能。

ECHOhash 中除了树结构，还使用了散列表。本组尝试使用 Zobrist 编码法，将当前棋盘状态编码成一个数，然后对应取余散列进容量较大的散列表中。具体编码方法是，对于 8*8 棋盘上每一个位置的每个状态（有黑子、白子、空格三种状态），赋予一个各不相同的随机数，然后对当前棋盘状态的每一个位置进行异或操作求得该状态的编码值。值得一提的是，尽管所有棋盘可能性数目总和极大，但在一次比赛中不可能出现全部的可能情况，所以相对来说散列表的大小应当是可以接受的。此散列表未设置冲突处理操作，因为 rehash 将带来巨大的散列和取值复杂度，不如先占先得制，即如果产生冲突，就直接调用局面估值函数进行操作。

4.3 函数说明

4.3.1 play 函数

参数：cb,ms

返回值：为一个字符串，代表我方选择的落子点

功能描述：与基础设施对接的接口函数，进行初步有无可下位置的判断，并且将搜索层数、初始 alpha-beta 值、棋盘 cb 作为参数传给 AlphaBeta 函数。

4.3.2 AlphaBeta 函数

参数：depth（当前搜索深度）、maxdepth（常数，代表规定搜索深度，我们选择三层）、alpha（从 play 函数传入时初始化为-100000000000000）、beta（从 play 函数传入时初始化为 100000000000000）、cb（当前棋盘）

返回值：一个列表，前一项为对当前节点的估值，后一项为最佳落子点

功能描述：搜索函数的主干部分，性质为一递归函数。进行 alpha-beta 剪枝版的负极大值搜索，通过不断调用自身对当前下棋方的所有子节点进行估值（通过估值函数或通过倒推），并选择最有利于自己的着法来下棋。

4.3.3 finish 函数

参数：cb（即当前棋盘格局）

返回值：True 或 False，前者代表棋局结束，后者代表未结束

功能描述：在 AlphaBeta 函数中调用，用来判断棋局是否已经结束。

4.3.4 Evaluate 函数

参数：ChessBoard（即当前棋盘）

返回值：一个整数，正值代表对当前下子方有利，负值代表不利，值越大优势越大。

功能描述：对当前局面进行评估并返回相对于当前落子方的估值。具体实现部分如下：

棋盘的潜在能力类

行动力（Mov）

行动力的主要决定因素是合法位置（Legal Position）的多寡，采用我方的合法位置数量减去对方的合法位置数量得到的数值来表征行动力。如图所以，黑棋（我方）的合法位置数量为 5，白棋合法位置数为 4，那么当前行动力为 1。

权值：8

潜在行动力（pMov）

潜在行动力，顾名思义，就是潜在的行动力。假如对方一个棋子，旁边是一个空格，那么我方以后可能会下这个空格吃子，该空格就是我方的潜在行动力。[1]在考虑当前棋局的潜在行动力时，用我方的潜在行动力与对方的相减返回 pMov 值。

权值：3

占角力（Ang）

在黑白棋中，棋盘有“金角银边”之说，可见在大多数情况下，尽可能实现占角的必要性。Ang 主要用来考虑下一步被占角的可能性。即用我方的合法位置里角的数量减去对方合法位置里角的数量，返回 Ang 值。

权值：500（在 ang 函数中实现的）

棋子当前位置与能力类

棋子数（Number）

当前棋盘上的黑白棋子数量对比是衡量局势的一个非常直观的因素，我们也将其划入考虑范围之内，Number 返回我方棋子数与对方棋子数的差值。

权值：1

占角数（Cor）

在棋子数的基础上作进一步地考虑，当前棋局被占角的情况也是一个直观的衡量标准。考察棋盘的四个角，cornerpos（）函数返回我方占角数与对方占角数的差值 Cor。

权值：1000000

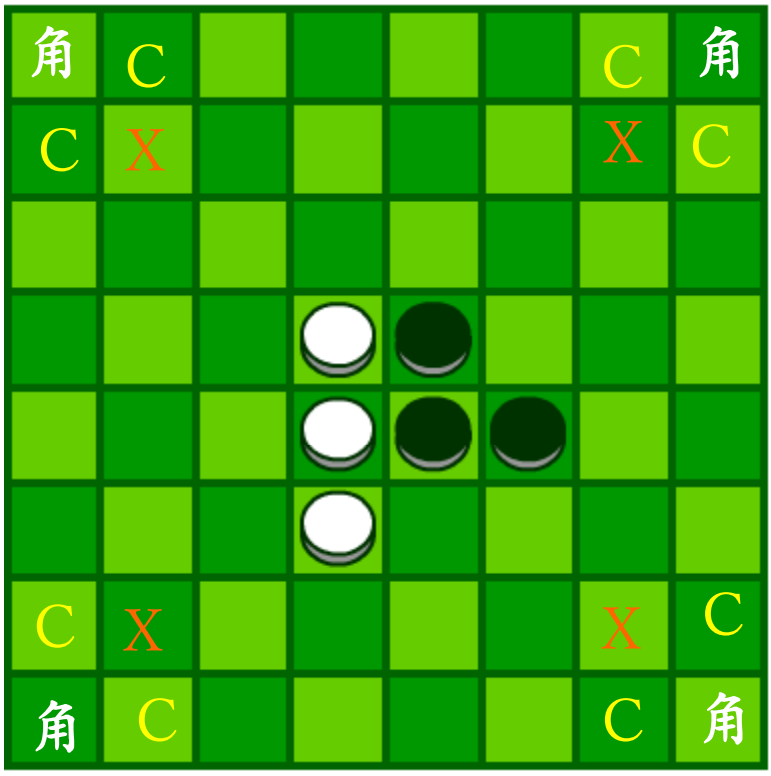


图 2.1 角、C 位、X 位

占 X 位数（Xpos）

返回对方 X 位占据的数量与我方 X 为占据的数量的差值。占 C 位数（Cpos）

权值：500

占 C 位数（Cpos）

C 为即在棋盘的四边上与顶角相邻的八个位置，我方的棋子占据 C 位可能促使顶角被对方棋子占据，归于不利位置，返回当前局面对方 C 位占据数与我方 C 位占据数的差值。

权值：100

边缘子（Edge）

我们定义相邻空格数大于等于三的棋子为边缘子，在下棋过程中我们应当尽量少下边缘子，用对方的边缘子数量减去我方的边缘子数量即返回值。

权值：2

稳定子（Sta）

稳定子即在棋盘中的无法或者很难被对方吃掉的棋子,仅考虑顶点，以及边上与顶点相连的同色棋子。（边上受顶点保护的棋子）

Sta 返回稳定子的数量差。

权值：5

图 2.2 受右上角保护的稳定子（6 个）

边角配置类

“#0000…”类边（Sid）

类似于“#0000”的，某色占据顶角，边上异色棋子相连的配置，容易导致 0 子被#子反吃进而占据边上的稳定子甚至一整条边。我们根据“#0000…”中 0 子的数量来判断这种配置的危险程度，返回值为双方的差值。

权值：1

图 2.3 sid 型边（左上）

同色边（Iso）

如果一条边上全部是同色棋子，那么这种情况非常稳定，超过了一般的稳定子，要尽量避免让对手出现这种局面。单独把同色边列为一类。返回值即为我方的同色边的数目。

权值：100

图 2.4 Iso 型边

死亡角（Dic）

占据顶角在黑白棋中十分重要，但是在有些情况下不能一味地保留角。比如我方占据了

一个顶角，而与其相邻的边上完全被对方棋子占据形成稳定子，这种情况的估值不应当高。死亡角即考虑这种情况。返回对方与我方死亡角的差值。

权值：100

图 2.5　Dic 型边

Evaluate 函数返回值总评估分数：

bsup = 5*Sta + 8*Mov+ 3*pMov+Cor*10000000+X*500+Edge*2 + c*100 + number + Sid + Iso*100 + Ang + Dic*100

4.3.5 程序限制

对于 ECHO4，当递归层数增加到 4 时，某些时候对阵 gambler，将会 timeout。我们推测这是由于每一层的可行位置都过多或者发生了 AlphaBeta 剪枝的退化引起的。因此为了保险起见，我们将递归层数减小到 3，平均时间缩短到 40-80 秒，以求不出现 timeout 的情况。

4.4 实验结果

4.4.1 实验数据

实验环境说明：

- 硬件配置：Intel Core i5/8GB
- 操作系统：OS X Yosemite/10.10.3
- Python 版本：2.7.6

1000 次对弈结果数据汇总

黑 \ 白	idiot 算法	gambler 算法	本算法
idiot 算法	0:1000 27000:37000	512:442 32663:30792	0:1000 17000:47000

	14.2280:19:7398	19.5539:15.8975	11.6644:15467.8638
gambler 算法	369:603 28784:35050 23.3561：28.9988	468:480 31859:32078 20.8697:20.8948	28:969 18215:45759 46.9367:74401.4875
本算法	1000:0 41000:23000 12830.2464:12.2896	959:24 45439:18513 74508.4757:46.4183	1000:0 50000:14000 75995.5764:45519.2637

4.4.2 结果分析

在与 idiot/gambler 算法的对弈过程中，本算法中博弈树的预判功能有了显著的优势。由于 idiot/gambler 的智能性都比较低，不具备多步之后的预判能力，所以本算法的棋力明显高于二者。

算法在运行时间上在预期之中，搜索深度为 3 时，平均每局耗时 50 秒左右。主要的时间开销在搜索树的叶节点层上。

4.5 实习过程总结

4.5.1 4.1 分工与合作

小组分工

本组分工为任务承包制，将整个实习任务分为几个小任务并分配到个人。其中：

（1）开发算法、编程、撰写报告中的算法代码描述部分：陈春含、吴逸夫、运乃丹

（2）进行算法测试与实验：赵芳珩

（3）撰写、整合报告：彭玉环

（4）机动性任务（搜索资料、协助测试、协助报告等）：付帆飞

小组交流与合作方式、过程

本组的合作交流主要是通过线上线下双方面进行的。线上为微信群，线下为小组会议。

微信群是实时交流，适合算法成型后不断修改的过程；而小组会议主要是前期探索算法时为了提高讨论效率而实施的。

图 4.1 一直活跃的微信群

ECHO 组第一次小组会议记录

记录人：彭玉环
2015 年 6 月 3 日

会议内容： 1、小组分工：陈春含、运乃丹、吴逸夫 算法编程
付帆飞 搜集资料
赵芳珩 测试与评估
彭玉环 实验报告
2、编程过程：（1）搜集可参考资料
（2）开发算法
（3）实现（06—03—name—author.py）
（4）测试与评估 表格 （轮流执黑下 10 局：截图 胜负 黑白子总数 KO？ Timeout？ 时间）
3、报告： （1）算法思维阐述
（2）程序代码说明
（3）测试过程报告
（4）小组分工和实验过程
4、（整理、排版、美工）
5、博弈数，剪枝的基本思想
用剪枝法写代码

图 4.2 第一次组会记录

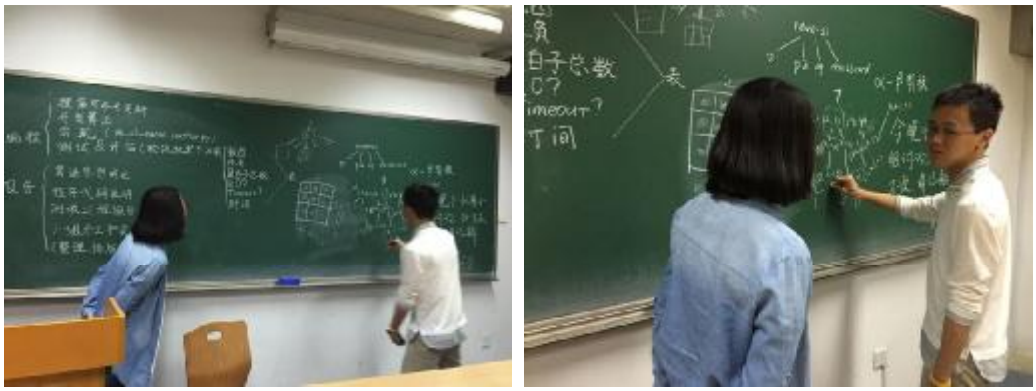


图 4.3 第一次组会 写满字的黑板和掠影

ECHO 组第二次小组会议记录

记录人：陈春含

2015 年 6 月 6 日

会议内容： 1、分工 DDL：6.8 前——开发出第一版成型程序，优先级仅考虑角和边；

6.13 前——改进算法，增强棋力；

6.15 前——根据热身赛结果有针对性地调整；

6.16 前——完成报告主体；

6.22 前——完成测试并总结数据分析、充实报告。

2、分享黑白棋网站和文献、资料上关于博弈树、估值的基本方法，确定编程组中三个人的初步分工：陈春含负责博弈树部分，吴逸夫、运乃丹负责估值函数部分。

3、交换、分享参考资料。

4、讨论确定估值函数大概从稳定子、行动力等方面进行评估。

图 4.4 第二次组会记录

图 4.5 第二次组会上的讨论

ECHO 组第三次小组会议记录

记录人：陈春含
2015 年 6 月 9 日

会议内容： 1、交流在开发第一版代码时碰到的障碍与收获，以及第一版代码的性能。
2、讨论决定黑白棋估值函数还需添加的功能——边角配置等。
3、赵芳珩测试时将数据暂时改到 100 次，及时反馈出大量调整结果。
4、决定人工测试与机器测试相结合，动用人力与电脑手动对垒以发现己方算法棋力的缺陷和弱点。
5、博弈树中有 bug 导致不能增强原有 evaluate 的棋力，将博弈树和 evaluate 拆分开，开发 evaluatetest 算法，不含递归深度而仅估值，由吴逸夫改进 evaluatetest，目标为纯 evaluatetest 战胜中级算法。
6、陈春含修改博弈树，使之发挥作用并尽量运用哈希表加速。

图 4.6 第三次组会记录

图 4.7 第三次组会

4.5.2 经验与教训

本次实习中，小组从一开始的分工就比较明确，探索过程中也比较积极，相关负责同学之间有充分的讨论，成员对自己的任务都认真对待、不推卸责任、不误期不迟到等等，使得总体氛围是融洽和谐的。

不足之处是成员之间的任务分配不够细致具体，在开始时没有样样落实到个人。针对每位成员的特点分配任务这一点做得不够好，导致前半段时间很多是浪费掉的。还有，组内选择的博弈树算法没有及时做到人人理解透彻，导致编程效率低下，博弈树中一个复制棋盘带来的 bug 耗费了几天的时间，也使 ECHO4 的加强版本没能在比赛之前完成。

以后再进行类似的实习时，应该加强组员间的反馈与调整，根据各自的特长与特点分配任务、提高实习效率和质量。

4.5.3 建议与设想

（1）采取一定的方式增加赛前各组之间的相互比较，以尽量减少正式比赛时的实力悬殊，同时也能互相促进。

（2）小组分配还是有一些不太合理的地方，希望能够将高水平的同学更均匀地分散在各个组，同时也要让小组分工落到实处，避免水平高的同学超负荷工作而其他同学水平得不到提升的情况发生。

（3）竞赛的时间有些长，尤其是初赛时，不够紧凑，观赏性有待提高。

（4）大作业的形式特别好，但时间有点晚，最好不跟期末紧压在一起，期末任务太重，先实习完可以让大家在后续学习中有更多的实践和理解、消化机会。

（5）最好可以留时间请冠亚季军组分析展示、讲解自己的代码，供大家学习参考。

（6）设置一些纪念奖、特别奖，奖品可以很简单，有 dsa 标志，让更多同学留下回忆。

4.6 致谢

感谢赵芳珩同学和她的电脑：芳珩同学一直及时开发着测试一个又一个实验版本的程序，在期末论文的压迫下依然分出一半的 CPU 奉献给了测试；电脑无比辛勤地不眠不休工作了三天三夜，跑出第一次测试结果，却因少了原始数据而深藏功与名，重新开始全马力的奔腾，终于又两个日升日落之后，得出了长达 70 页的实验数据。

感谢吴逸夫、运乃丹、陈春含同学的不懈探索和永不弃疗的精神。尽管一次又一次被大神们碾压，他们还是积极乐观地完善着自己的代码，四处学习，念掉十几篇文献、搜过国内外网站、刷过许多夜晚与许多白天，尽管完成的事情并不很优秀，但收获的知识仍是十分丰厚。

感谢彭玉环同学一直积极的参与、认真地撰写笔记报告、耐心地进行人工对奕，尽管比较枯燥却从不抱怨，而且总是无私地称赞着成长中的算法，给艰苦跋涉在探索中的算法组带来雪中送炭般的鼓励。

感谢付帆飞同学搜集到的算法思路与参考，以及热身赛之后的结果整理分析。

感谢陈斌老师给予的指导，感谢 Bravo 组的联谊和坦诚分享、对弈。

感谢逸夫二楼的 3430、3331、3425，四教 501，光华地下室，阳光大厅，Zoo Coffee 以及圣摩尔，带着电流与 WiFi 陪 ECHOer 们度过那些安静而不寂静的深夜。

4.7 参考文献

[1].A New Hashing Method With Application For Game Playing. Albert L.Zobrist. Technical Report. April.1970

[2].基于改进博弈树的黑白棋设计与实现. 李小舟. 华南理工大学硕士学位论文. 2010.5

[3].计算机围棋博弈中 UCT 算法的应用及改进. 黄晶. 北京邮电大学硕士研究生学士论文. 2011.1

[4].计算机国际象棋博弈系统的研究与实现. 万翼. 西南交通大学研究生学位论文. 2006.4

[5].具有自学习功能的计算机象棋博弈系统的研究与实现. 王一非. 哈尔滨工程大学工学硕士学位论文. 2007.1

[6].博弈算法在黑白棋中的应用. 杜秀全, 程家兴. 计算机技术与发展. 2007.1

[7].五子棋中 alpha-beta 搜索算法的研究与改进. 程宇, 雷小锋. 计算机工程. 2012.9

[8].中国象棋 alpha-beta 搜索算法的研究与改进. 岳金朋, 冯速. 北京师范大学学报（自然科学版）. 2009.4

[9].计算机围棋博弈系统的若干问题研究. 谷蓉. 清华大学工学硕士学位论文. 2003.6

[10].AlphaBeta 剪枝算法——申龙斌的程序人生.
http://www.cnblogs.com/speeding/archive/2012/09/20/2694704.html

[11].AlphaBeta 剪枝搜索 http://www.cnblogs.com/IThaitian/p/3616550.html

[12].AlphaBeta 剪枝在黑白棋中的优化.
http://wenku.baidu.com/link?url=R2LiEKV9tOrCmHIYDPV9mUygteMdoOt9tMDx-xcJZKfYoxZa3kC403A6vuYHRjnerd6VvtIMJ4gQhXB1higrCC7tJbgw94i9_0cXs-lu7s3

[13].黑白棋天地. http://www.soongsky.com/

东区 East　　OSCAR 组

5 数据结构与算法课程实习作业报告

（谭光钰* 李嘉琪 武化雨 彭玉恒 张恩珍 李佳斌）

摘要：在下棋过程中，每个博弈者在已知当前环境的情况下需要通过搜索的方式来决定下一步的走法。我们运用博弈树结构，在极小极大搜索过程中调用评估函数，对当前棋盘进行估值，并进行线性加权，通过评估结果确定某一步对自己最有利的走法。我们的测试结果比较令人满意，运行时间也在预期之内，在与 idiot 或 gambler 算法对弈时，本算法中的博弈树搜索等策略起到了很好的效果，极小极大搜索避免或尽量减少了对手在下一步时占据边角的可能，使得本算法在比赛中能够占据优势地位。

关键字：博弈树　极小极大搜索　结合历史表的 alpha—beta 搜索

5.1 算法思想

5.1.1 总体思路

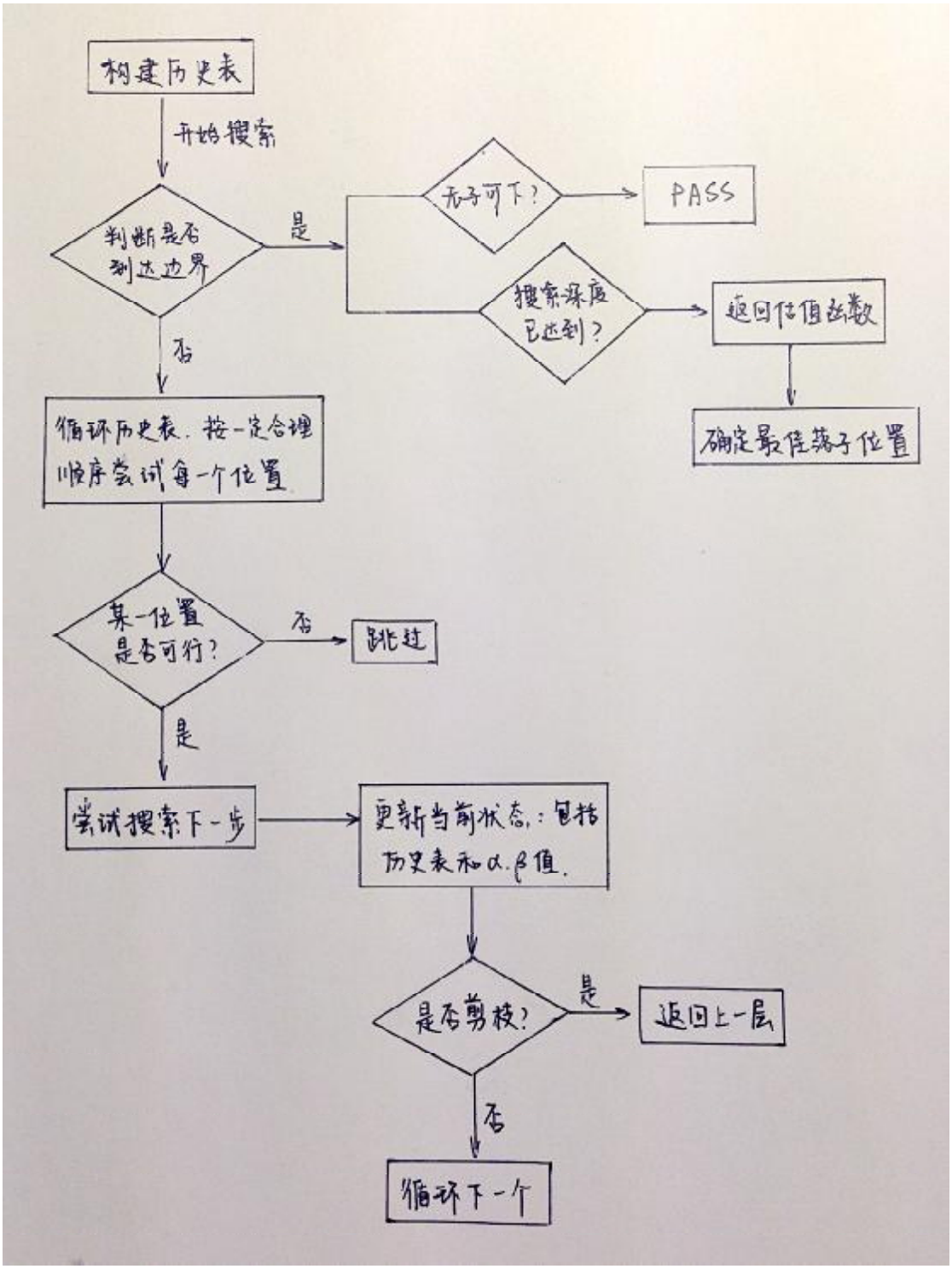
总体思路：运用博弈树结构，在极小极大搜索过程中调用评估函数，对当前棋盘进行估值，考虑行动力估值、边缘子估值、子数的估值、位置价值的估值和稳定子的估值五种估值，并进行线性加权，通过评估结果确定某一步对自己最有利的走法。

采用的主要数据结构与算法：博弈树

采用的算法策略：下棋过程中，博弈者的策略搜索可以用树结构表示，称之为博弈树，树上的节点为棋盘可能出现的状态，父节点通过一步行动派生出的所有后继节点为其子节点。为了从众多可供选择的行动方案中选出一个对自己最为有利的行动方案，就需要对当前的情况以及将要发生的情况进行分析，通过某搜索算法从中选出最优的走步。在博弈问题中，每一个格局可供选择的行动方案都有很多，因此会生成十分庞大的博弈树，只能往前搜索有限的步数，最基本的搜索策略称为极小极大搜索，采用结合历史表的 alpha—beta 搜索可以对

搜索方法进行优化。

5.1.2 算法流程图



5.1.3 算法运行时间复杂度分析

程序的主要框架采取的是极大极小搜索，也就是深度优先搜索的一种。我们的程序设定是前 52 步向下搜索 4 步，最后 12 步搜索到头。假设前 52 步每搜索一步可选择平均方案数为 A，后 12 步每搜索一步可选择平均方案数为 B。对于估值函数，时间复杂度是常数级别为 C1（因为估值主要是进行了棋盘的遍历，棋盘是 8*8 的规模）。对于历史表的更新操作也是常数级别的，因为可选的步数不超过 64 步为 C2。则每一步的时间复杂度记为 C=C1+C2 为一个大约几百的常数。则时间复杂度近似为 O（C*(26*A^4+6*B*12)）。我们注意当棋盘下到最后几步的时候，棋盘可选择的位置非常有限，即 B<A,在调试搜索参数时也发现前者深度调大影响不是太大，当调到 2 左右的时候时间主要受后者限制，后者将深度调大，时间增加比较大。注意，我们将历史表与 alpha—beta 优化的效果体现在 A、B 中了。

5.2 程序代码说明

5.2.1 数据结构说明

```
class dfsTree:#模拟博弈树，编一个树节点
    def __init__(self,cb,lp,i,maxmin,a=-100000,b=100000):
        self.cb=cb
        self.lp=lp#保存其可走的步
        self.move=i#是由父节点的哪一步过来的
        self.maxmin=maxmin#记录其是最大最小的哪一步
        self.mk=None#记录其走哪一步最优
        self.a=a#alpha—beta中的alpha
        self.b=b#alpha—beta中的beta
        if maxmin==1:
            self.m=-10000000#记录走的最优值
        else:
            self.m=10000000
        self.son=[]#保存子节点
```

本算法主要采用博弈树结构。博弈树上的节点为棋盘可能出现的状态，父节点通过一步行动派生出的所有后继节点为其子节点。为了从众多可供选择的行动方案中选出一个对自己最为有利的行动方案，就需要对当前的情况以及将要发生的情况进行分析，通过某搜索算法从中选出最优的走步。在博弈问题中，每一个格局可供选择的行动方案都有很多，因此会生成十分庞大的博弈树，试图通过直到终局的搜索而得到最好的一步棋是不可能的，因此只能往前搜索有限的步数，最基本的搜索策略是极小极大搜索。本算法在极小极大搜索的基础上，

还运用了结合历史表的 alpha—beta 搜索来进行优化。

5.2.2 函数说明

1. 对棋盘进行行动力估值，在小组进行测试后，根据经验进行了加权。所谓的行动力，就是当前走棋的一方可以选择的点的数目 A。当然，对一个具体的局面来说，你可以查找出当前不走棋的一方的可走的点的数目 B。如果 A 大于 B，可以认为是当前走棋方比较占优的；否则就是当前走棋方在劣势。

```
def value1(cb):
    v1=0
    lp1=cb.getLegalPos()
    ch=ChessBoard(cb)
    ch.turn,ch.opturn=ch.opturn,ch.turn
    lp2=ch.getLegalPos()
    v1=len(lp1)-len(lp2)
    if cb.getTotalScore()>50:
        v1=v1*50
    else:
        v1=v1*30
    return v1
```

2. 对棋盘进行边缘子估值。边缘子定义为与一个或多个空位相邻的棋子。虽然从技术上来说，处于边上的棋子也可以符合这个定义，但是在提到边缘子时通常并不包括它们。内部子是指不与空位相邻的棋，没有内部子是战略性的错误。因此在下棋的过程中我们应尽量增加边缘子的个数。

```
def value2(cb):
    v2=0
    def side(cb,x,y):
        d=[[1,0],[-1,0],[0,1],[0,-1]]
        for i in range(4):
            px=x+d[i][0]
            py=y+d[i][1]
            if px>=0 and px<8 and py>=0 and py<8:
                if cb.board[px][py]not in ['#','O']:
                    return True
        return False
    for i in range(8):
        for j in range(8):
            if cb.board[i][j] in ['#','O']:
                if side(cb,i,j):
                    if cb.board[i][j]==color:
                        v2=v2-1
                    else:
                        v2=v2+1
    if cb.getTotalScore()<25:
        v2=v2*30
    return v2
```

3. 对棋盘进行子数估值。黑白棋规则规定，获胜方是对局结束时己方颜色的棋子较多的那一方。新手往往倾向于把这个长期目标当成短期策略：他们在对局的各个阶段都试图拥有更多的棋子，为此，他们下的每步棋都要翻转尽可能多的棋子，这就叫多子策略。但这种策略并不好，其实拥有许多棋子并不重要，重要的是拥有许多在对局剩余阶段无论如何都不会被对手翻转回去的棋子，即稳定子。当然，在对局还剩最后几步棋之前，通常很难获得稳定子。

```
def value3(cb):
    v3=0
    for i in range(8):
        for j in range(8):
            if cb.board[i][j]==color:
                v3=v3+1
            elif cb.board[i][j]==opcolor:
                v3=v3-1
    if cb.getTotalScore()<=20:
        v3=-v3*20
    elif cb.getTotalScore()>=60:
        v3=v3*50
    return v3
```

4. 对棋盘进行位置价值的估值。在下每一步棋时，需综合考虑包括棋盘某一位置价值在内的评估结果。在进行了多次更改实验和测试后，本算法中的位置价值估值采用了效果较好的一组数据。

```
def value4(cb):
    v4=0
    est=[[0 for i in range(8)]for j in range(8)]
    est[0]=[90,-60,10,10,10,10,-60,90]
    est[1]=[-60,-80,5,5,5,5,-80,-80]
    est[6]=[-60,-80,5,5,5,5,-80,-80]
    est[7]=[90,-60,10,10,10,10,-60,90]
    est[3][0]=10
    est[4][0]=10
    est[5][0]=10
    est[3][1]=5
    est[4][1]=5
    est[5][1]=5
    est[3][7]=10
    est[4][7]=10
    est[5][7]=10
    est[3][6]=5
    est[4][6]=5
    est[5][6]=5
    for i in range(8):
        for j in range(8):
            if cb.board[i][j]==color:
                v4=v4+est[i][j]
            elif cb.board[i][j]==opcolor:
                v4=v4-est[i][j]
    if cb.getTotalScore()<50:
        v4=v4*10
    return v4
```

5. 对棋盘进行稳定子的估值。根据下棋规则，位于角上的那颗棋子是不可能被翻转的，而且一旦占角，常常还有可能建立很多受角的保护而永远不会被翻转的棋子，这些的棋子就称作**稳定子**。显然稳定子数量越多，对于接下来的形势越有利。这里我们对稳定子的数目进行了估算，即只考虑了由角引发的稳定子。

```
def value5(cb):
    f=[[0 for i in range(8)]for j in range(8)]
    if cb.board[0][0] in [color,opcolor]:
        if cb.board[0][0]==color:
            k=1
            C=color
        else:
            k=-1
            C=opcolor
        for i in range(8):
            for j in range(8):
                if (i-1<0 or f[i-1][j]==k)and(j-1<0 or f[i][j-1]==k)\
                   and cb.board[i][j]==C:
                    f[i][j]=k
                else:
                    break
```

```
if cb.board[7][0] in [color,opcolor]:
    if cb.board[7][0]==color:
        k=1
        C=color
    else:
        k=-1
        C=opcolor
    for i in range(7,-1,-1):
        for j in range(8):
            if (i+1>7 or f[i+1][j]==k)and(j-1<0 or f[i][j-1]==k)\
               and cb.board[i][j]==C:
                f[i][j]=k
            else:
                break
if cb.board[7][7] in [color,opcolor]:
    if cb.board[7][7]==color:
        k=1
        C=color
    else:
        k=-1
        C=opcolor
    for i in range(7,-1,-1):
        for j in range(7,-1,-1):
            if (i+1>7 or f[i+1][j]==k)and(j+1>7 or f[i][j+1]==k)\
               and cb.board[i][j]==C:
                f[i][j]=k
            else:
                break
if cb.board[0][7] in [color,opcolor]:
    if cb.board[0][7]==color:
        k=1
        C=color
    else:
        k=-1
        C=opcolor
    for i in range(8):
        for j in range(7,-1,-1):
            if (i-1<0 or f[i-1][j]==k)and(j+1>7 or f[i][j+1]==k)\
               and cb.board[i][j]==C:
                f[i][j]=k
            else:
                break
v5=0
for i in range(8):
    for j in range(8):
        v5=v5+f[i][j]
if cb.getTotalScore()<30:
    v5=v5*100
else:
    v5=v5*50
return v5
```

5.2.3 程序限制

程序没有进行合理的防超时的控制，深度搜索时若搜索深度太大，可能会超时。同时程序没有特别的将自己无子可下的情况，估一个极小的值，也就是没有防 KO 手段。

5.3 实验结果

5.3.1 实验数据

实验环境说明：

- 硬件配置： Intel(R) Core(TM) i5-3230M CPU @ 2.60GHz 2.60GHz
- 操作系统： Windows 7 旗舰版
- Python 版本： 2.7.9

1000 次对弈结果数据汇总

黑 \ 白	idiot 算法	gambler 算法	本算法
idiot 算法	0：0：1000 27000：37000 17.7937832657： 24.3291639496	524：34：442 32884：30642 18.9506532683： 15.6617527303	0：0：100 2000：4400 1.45880887315： 4347.78478804 （测试 100 次）
gambler 算法	395：30：575 28992：34867 15.7961200613： 18.7773874716	460：50：490 31664：32278 21.7053495161： 21.4986081879	0：0：1000 12333：51652 13.4721574967： 37777.2735354
本算法	100：0：0 5700：700 2541.16011059： 1.31675225166 （测试 100 次）	995：0：5 51878：11955 51639.1048549： 14.3164838978	100：0：0 5400：1000 6654.90231192： 4743.31506674 （测试 100 次）

5.3.2 结果分析

在与 idiot 或 gambler 算法对弈时，本算法中的博弈树搜索等策略起到了很好的效果，极小极大搜索避免或尽量减少了对手在下一步时占据边角的可能，使得本算法在比赛中能够占据优势地位。算法在运行时间上在预期之内。主要的运行时间开销在递归上，随着递归深度的增加，程序运行时间会相应加长。

5.4 实习过程总结

5.4.1 分工与合作

小组分工：

收集资料——彭玉恒

编写程序——谭光钰、李佳斌

程序测试——李嘉琪

报告撰写——武化雨、张恩珍

合作与交流的方式：小组会议

历次会议记录：

【1】

时间：2015 年 6 月 4 日

地点：燕南园

人物：小组全体成员

内容：小组成员首次会合，分享了各自玩黑白棋游戏得到的经验和对于程序设计的展望，为编写具有竞争力的黑白棋程序做好了准备。并分配了第一次任务：查找黑白棋相关资料，记录自己的想法，在下一次会议时进行交流。

【2】

时间：2015 年 6 月 5 日

地点：理教 413

人物：小组全体成员

内容：大家分享了自己查找到的资料，并提出了自己的想法，组长整理综合了大家的想法后初步决定程序编写方案，进行了组员分工，每一位组员按照自己的任务开始了下一步工作。由一位同学负责继续查找黑白棋的相关资料，资料汇总后负责编程的同学开始编写程序，决定采用极小极大搜索算法。

【3】

时间：2015 年 6 月 13 日

地点：二教

人物：小组全体成员

内容：参加了老师组织的第一次热身赛，根据热身赛的结果，小组成员进行了讨论，并对程序进行了修改，对极小极大搜索进行了 alpha—beta 优化，与 idiot 和 gambler 算法对战的测试结果有了明显提升。

【4】

时间：2015 年 6 月 15 日

地点：农园

人物：小组全体成员

内容：黑白棋竞赛结束，我们组最终的成绩止步八强，赛后小组成员集中在一起，对这次作业进行了经验与教训的总结，并提出了一些建议与设想。

5.4.2 经验与教训

在这次黑白棋的实习作业中，我们小组虽然没有拼进四强，但以小组第二的成绩夺得八强也算是不错的成绩。这次作业是一个需要小组成员合作完成的作业，在这个过程中，我们的组员分工合作，每个人都积极准时地完成了自己的任务，才使得我们组取得最后这样比较满意的成绩。因此，这次实习作业不仅让我们在数据结构与算法这门课的学习上有了进一步理解，对于使用 python 语言和树结构等算法更加熟练，同时也增强了我们的团队意识和小组协作能力，让我们收获良多。当然实习过程中也有一些不足，比如编程序不能拖延，否则可能导致测试员测试的时间紧张，在这方面我们可以改进；而且，在已知具有热身赛的时候我们没有十分重视，准备的也不够充分，不像其他小组做好了积极充分的准备，在比赛前提交了好多程序作为热身，从而来提升自己的程序，这一点也是我们应该向其他小组学习的。

5.4.3 建议与设想

在组好队后应该尽快分配任务并开始编程工作，因为测试往往需要花费较长的时间，这样可以有更多的时间对自己的程序进行修改，希望学弟学妹们能够吸取经验。另外我们认为，还可以在竞赛后增加一个人机对战的环节，请下黑白棋比较厉害的同学与前几名的程序进行对战，看看同学们的程序是否可以打败人脑的智慧。

5.5 致谢

感谢数据结构与算法老师陈斌、GIS 专业研究生曹鹏对我们的帮助。

5.6 参考文献

【1】http://www.docin.com/touch_new/preview_new.do?id=607881327

【2】

http://wapwenku.baidu.com/view/d0d6a34769eae009581becde.html?ssid=0&from=844b&uid=0&pu=usm@0,sz@1320_2001,ta@iphone_1_8.3_3_600&bd_page_type=1&baiduid=6422BB389376767040A6158FB9F40622&tj=wenku_2_0_10_title#page/1/1433290621017

【3】http://blog.csdn.net/guzhou_diaoke/article/details/8201542

【4】http://www.cnblogs.com/tuanzang/archive/2013/02/27/2935837.html

【5】https://www.yumpu.com/en/document/view/24025740/

【6】http://d.wanfangdata.com.cn/periodical_wjfz200701072.aspx

【7】《几种智能算法在黑白棋程序中的应用》柏爱俊 2007 年 10 月

西区 West　　QUEBEC 组

6 数据结构与算法课程实习作业报告

（唐启浩* 黄翔 车元孟 周安 罗彪 朱福海）

摘要：本次实习中我们小组一共设计了两个算法，其中有一个可以顺利无 bug 运行，另一个算法存在一些问题没有能解决。这次实习中主要涉及算法结构有线性数据结构，图算法，排序，递归，搜索等内容。算法在占角及占边上有一定的优越性。实验数据上，对战的时候胜率还是比较高的。如在与有随机性的 gambler 算法对弈时，在很多情况下都能够以较大的优势获胜。对于 idiot 算法，因为没有随机性，则每一局的结果都是一样的，不适用与进行算法分析。存在问题的那个算法如果能实现则需要耗费较多的时间。

关键字：下子位置等级　危险下子位置　最小获子　最大获子　图　递归　排序　深度搜索

6.1 算法思想

6.1.1 总体思路

成功的算法的思路为：在最开始将 60 个位置人为地划分为五个等级，如列表 totalList 所示。接着由 simpleDeal 函数对合法位置列表做初步的处理，按危险等级逐级删除危险下子位置（若合法位置列表中已无下子位置，则将最后删除的位置返回给 play 函数，将其定为下子位置），在完成初步处理后，把处理后的合法位置交给 Minget 函数和 Maxget 函数去处理。

用 30 将棋盘认为地分成前后两个阶段，前一阶段调用 Minget 函数，通过筛选，找出使自己在棋盘上的子数保持最少的下子位置，从而限制对方的活动区域。后一阶段调用 Maxget 函数，找出使自己在棋盘上的子数保持最多的下子位置，从而增加自己的子数。

尝试但是最终没有成功的算法：深度搜索最优位置的方法。利用到了图算法，排序算法，递归调用，深度优先搜索等策略。

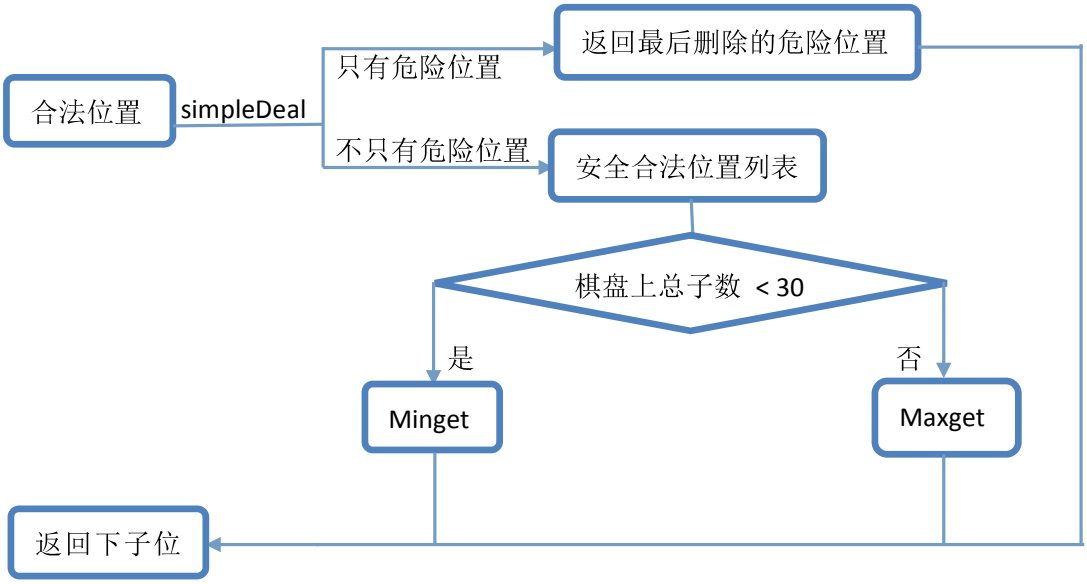
算法的思路为前期经过 simpleDeal 函数、Minget 函数和 Maxget 函数进行处理并下子，当棋盘上的子数达到一定程度时，再使用这个算找出可以获得胜利或者子数尽可能多的位置。在到达指定的子数时，调用函数 longsearch 进行最优位置的搜索，其中包含子函数 buildGraph 以及 dfs 函数。buildGraph 函数主要是模拟下子，将所有可下的位置全部尝试，逐层往下，利用递归调用，将从指定位置后的所有下子的可能都建立成一个 DAG，为了避免同一个位置重复出现而造成有圈图，则对下子位置进行处理，在其后加上数字，保证形成的是有向无圈

图。dfs 函数是对于已经形成的有向无圈图进行处理，利用其子函数 dfsearch 将某个可下位置最终的结果生成一个有序表，则表的最后一个为该位置理论上最后己方与对方的子差。将合法位置中的所有位置所对应的有序表的最后的数据对比，选择最大胜利的位置返回，该过程使用到了选择排序的算法。通过这些步骤之后返回的就是最终可以获得胜利或者子数尽可能多的位置。

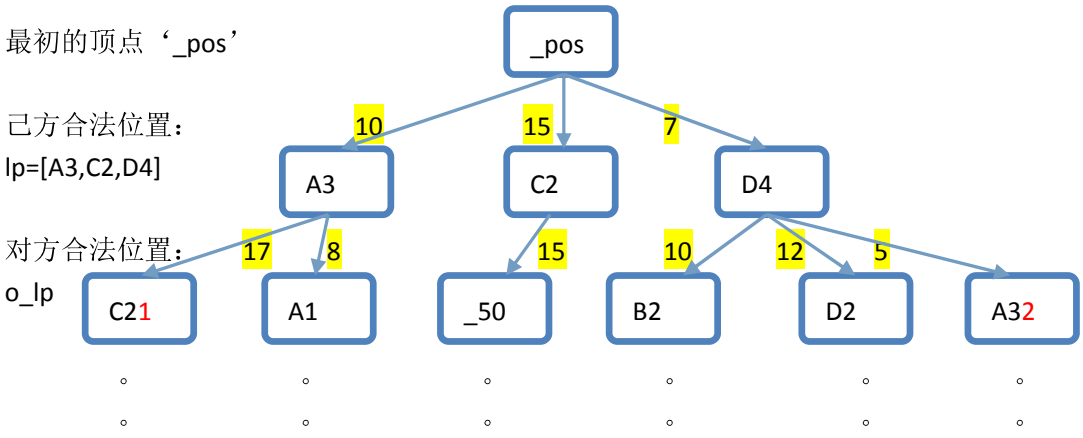
对于这个算法，如果找到问题所在的话，应该算得上是一个比较好的算法。可以利用 ms 字典来保存建立好的图，那么只需要建立一次就可以了，可以省很多时间。但是由于没有找出问题在哪里，这部分的代码最终没有进行完善。

6.1.2 算法流程

可运行的算法：



另外的失败算法的建图思路结果图解：



。 。 。 。 。 。

当节点中重复出现同一个顶点时，在新的顶点后面加上数字，如上图 C21，A32。屋子可下时将此时的棋盘上的棋子总数加上前缀‘_’构成顶点。边上的权重为下了改位置之后己方子数与对方子数的差值。

建好图之后再用其他函数进行下一步操作。

6.1.3 算法运行时间复杂度分析

成功运行的算法涉及到一个主函数和三个求解函数，其中主函数为 O(1)的时间复杂度，Minget 函数和 Maxget 函数中各有一个 for 循环，为 O(n)的时间复杂度，simpleDeal 函数中有数个 for 循环并列，而在每个 for 循环中又有 list.pop(k)方法的调用，故为 O(kn)的时间复杂度。

由上可得，该算法的时间复杂度是 O(kn)。

对于那个尝试但是最终没有成功的算法的时间复杂度主要用于建立有向无圈图以及搜索最优下子位置上。主函数为 longsearch，除了几个常数级别的复杂度外，还有就是其中调用 buildGraph 函数以及 dfs 函数。buildGraph 函数主要是在递归上耗费较多的复杂度。可下位置个数 k 越多、开始建图时棋盘上剩余的个数 n 越大则算法的复杂度越大。粗略计算最差的情况为 O（k^n）级别，当然一半情况下比这个复杂度要小得多，因为棋局越是到后期可下位置越少，所以复杂度会下降一些，后期的时候可能只有 O（kn）级别的。dfs 函数在一定程度上可以说是在遍历这一个有向无圈图（该过程有赋值语句），其中的函数 dfsearch 使用递归时也是要遍历这个有向无圈图（该过程有赋值语句），所以这两个函数最坏的情况下也是为 O（k^n）。另外，dfs 函数中还有一个选择排序的过程，其复杂度最大为 O（n^2）。其他的常数级别的复杂度忽略的话，这个算法的时间复杂度可以定为 O（k^n）。

6.1.4 数据结构说明

在比赛中使用的代码是非常简单及普通的代码，只使用了线性数据结构。在尝试中还使用了图的数据结构，并使用了深度优先搜索的一些性质。只是最终使用图的数据结构部分的算法始终存在问题，没有成功。

6.1.5 函数说明

可运行的算法中共有三个函数，simpleDeal、Minget 和 Maxget 函数。

simpleDeal 函数有三个接口，棋盘 cb，可下的合法位置 lp，要调用的特殊位置的列表 totalList。simpleDeal 函数的功能是对合法位置列表做初步的处理，按危险等级（totalList 列表中给出危险等级）逐级删除危险下子位置（若合法位置列表中已无下子位置，则将最后删除的位置返回给 play 函数，将其定为下子位置，该位置为所有位置中危险性最小的位置）。

Minget 和 Maxget 函数的接口均为两个，棋盘 cb 以及可下的合法位置 lp。Minget 函数将当前己方的子数+30 定为一个上限，之后试探下子，找出可最小的获子的合法位置。Minget

函数相反，去当前己方子数为下限，试探下子，找出可最大获子的合法位置。

把棋子总数为 30 将棋盘认为地分成前后两个阶段，使用 simpleDeal 函数排除危险位置后，在前一阶段调用 Minget 函数，通过筛选，找出使自己在棋盘上的子数保持最少的下子位置，从而使对方的机动性下降。后一阶段调用 Maxget 函数，找出使自己在棋盘上的子数保持最多的下子位置，从而增加自己的子数，争取最大获益。

失败的算法中有四个函数，longsearch、buildGraph、dfs 以及 dfsearch 函数。

longsearch 函数有两个接口，棋盘 cb，合法下子位置列表 lp。这个函数的功能是将所有合法位置试探下子，并模拟对方所有可下位置，直到结果出现胜负，并建立一个有向无圈图，将所有可能的情况。包含了子函数 buildGraph 以及 dfs 用于实现上述功能。

buildGraph 函数有 7 个接口。CGraph,_pos,cb,lp,nKO,turn 以及 count。CGraph 为有一个顶点为‘_pos’的图。‘_pos’为参数，用于让所有的合法位置有一个共同的起点，即‘_pos’。还有棋盘 cb，合法下子位置列表 lp。nKO 用于判断棋局是否结束。turn 为己方，由外部传入。count 为计数，避免重复出现同一个位置导致形成有圈图，在位置已经出现在图中时，在该顶点名称后面加上 str（count），直到这个顶点名字没有出现过。当出现‘PASS’的情况时，将此时的前缀‘_’+str(总子数)作为一个顶点（一定情况下会在后方加上数字后缀，原因如上）。该函数利用递归方法，模拟双方的下子可能，形成一个包含所有可能的有向无圈图。图 CGraph 中顶点为两方所下的位置,，边的权重始终为己方与对方的子差。最终建成图。

dfs 函数只有一个接口，将图 CGraph 传入。功能为找出该图中的最优下子位置，即获得胜利或者子数尽可能多的位置。从图的顶点‘_pos’的相邻顶点中找出最合适的位置，这些顶点即此时的可下的合法位置。dfs 函数中调用了一个 dfsearch 函数进行处理后可以将这些可下的合法位置每一个棋局结束时的最好结果（即最后一条边的权重），并将合法位置与对应的权重组成键值对输入一个字典中，经过处理，再利用选择排序算法，将这些位置的合适顺序进行排序。返回该列表，在 longsearch 函数出输出最好的位置进行下子。

dfsearch 函数有四个接口，CGraph,_pos,alist 以及 t。建好的图 CGraph，传入了图中的一个顶点_pos，列表 alist，以及传入的一个以_pos 为子顶点的顶点与_pos 的边的权重。这个函数的功能为将传入的顶点_pos 以下的合法位置对应的最终棋局结束时己方与对方的子差加入到 alist 中，最终进行排序，最佳的子差在列表的最后。再在 dfs 函数中进行处理。

6.1.6 程序限制

可运行的函数由于它的变化性不大，往往与同一个对手进行比赛是总是下在同一个位置上，如果不进行改变可能会一直以同一个结局结束比赛，在输的情况下为一直输下去。由于程序实现的过程极为简单，故暂时还没有找到其他使其出错的极端条件。当然，因为很简单，所以在选择下子位置时，所选的位置很有可能不是最优位置。试验失败的那个函数就谈不上程序的限制之处了。

6.2 实验结果

6.2.1 实验数据

实验环境说明：

- 硬件配置：CPU (英特尔)Intel(R) Core(TM) i5-4210U CPU @ 1.70GHz(2401 MHz) 四核
  内存 4.00 GB (1600 MHz)
- 操作系统： Windows8.1 64 位
- Python 版本： Python2.7.9

1000 次对弈结果数据汇总

黑 \ 白	idiot 算法	gambler 算法	本算法
idiot 算法	0:1000 平局:0 27000:37000 24.957:34.383	549:411 平局:40 33296:30233 33.094 :26.814	1000:0 平局:0 44000:20000 34.868:52.619
gambler 算法	381:589 平局:30 29601:34358 27.543:33.571	450:499 平局:51 31223:32714 31.010:30.773	141:828 平局:31 23885:40060 26.079:70.947
本算法	1000:0 平局:0 47000:17000 65.579:28.221	788:184 平局:28 39066:24837 74.038:25.716	0:0 平局:1000 32000:32000 69.903:70.998

6.2.2 结果分析

在与 idiot 算法对弈的过程中，因为算法没有改变下子位置，比赛的结果没有变化。胜利就一直胜利下去，失败就一直失败。算法在计算危险区域上是比较有用的，能占角就占角，能下边就一定情况下就下边，有一定的效果。但是因为程序叫为简单，先手后手情况下结果就很绝对。算法的时间消耗不大，在预期之中，因为无论是 simpleDeal 函数、Minget 函数或者是 Maxget 函数，它们的时间复杂度都不是很大，只是 idiot 和 gambler 算法几倍而已，运行的时间主要在 simpleDeal 函数，因为有时候会直接返回下子位置，不再经过 Minget 函数和 Maxget 函数。但是总体上时间消耗还是比较小的。

在与 gambler 算法对弈的过程中，时间的使用与和 idiot 算法对弈差不多，算法所采用的策略此时可以体现出来。有 2.1 中的图表也可以看出，无论是先手还是后手，这个算法的胜率都是较大的。说明那些处理合法位置的函数还是起到了一定的作用的。

6.2.3 经典棋局（可选）

以下是两个比较典型的棋局，对于算法的性能也有一定的反映。

```
T_gambler-VS-T_QUEBEC
KO:False TIMEOUT:False
#=16 O=48
tB:0.037286
tW:0.106425
#12345678
AOOOOOOOO
BOOOOOO#O
CO###O#OO
DOO#O#OOO
EOO###OOO
FOO##OOOO
GOOOOOOOO
HO####OOO
```

可以看出，在这两局对弈中，无论是先手还是后手，本算法成功的将四个角以及几乎边上的所有位置都占了。经过 gambler 算法对弈的其他结果分析来看，这个情况时占所有对局的很大一部分的比重的，这也说明了算法对于占角占边还是有一定的优越性的。

由于和 idiot 算法对弈的结果不会有随机性，所以它们对弈的数据就没有多少说服力。在此处就不再说明了。

6.2.4 分工与合作

成员	分工
唐启浩*　黄翔	算法
朱福海	测试程序
车元孟	资料搜集
罗彪　　周安	实习报告

6.2.5 经验与教训

经验：

组队阶段：

组合成一个优秀的队伍是非常重要的。如果拥有非常靠谱并且对于工作有一定了解并有一定能力能承担工作的一些部分的组员参加队伍，那么对工作的进行一定是非常有帮助的。

所以组队要趁早，争取拉到比较厉害的同学参与到其中来。

工作阶段：

1. 组好队之后，在开始工作前要了解组员们对于这个工作的了解程度，问清楚组员们的优势之处，让他们能够各尽其能。
2. 制定好各个工作的完成时间，让组员们有明确的时间截点，不至于让工作的进度过慢。及时的督促组员们完成各自的任务。
3. 遇到困难的地方，应该及时报告组长，不要等到时间快要结束时才反映，那样会对工作进程有很大的影响。

改进之处：

要积极与组员进行沟通，即使了解队友的工作完成情况，对于一些情况特殊的队友要进行特殊的处理。另外，合理控制时间，尽可能提前完成，否则可能会越拖越久，使得在一些情况下有影响较大。在工作的时候要全身心投入，争取不留下遗憾。

6.2.6 建议与设想

建议：

在组队的过程中尽量保证队伍之间力量的大小基本相同。队伍内的力量也要均衡一些，不要让队伍里都是强者或者都是弱者，可以强弱参半，互相提高。要不然就会强组很强，弱组很弱，差距很大。对此我的看法是可以由老师经过自己的判断事先组好队伍中的一部分人。比如将平时表现比较好的同学和比较差的同学组合一下先，然后在由同学们自己加入，这样的话就是一半的人是老师选择的，一半的人是自己加入的，“稀释”了强组和弱组，实力在一定程度上是比较均衡的。

在基础设施代码方面，这次的代码的几处漏洞已经被同学们找出来了。那么下一回可以人为的加入一些漏洞，让同学们自己去找，这个也可以记入到小组的成绩当中。

寄语：

希望数算课越来越好！希望学弟学妹们经过数算这门课之后收获到的不仅仅是知识，还有更多的技能。学弟学妹们活力四射，让数算课越来越好玩！

6.3 致谢

首先要感谢陈斌老师举行这次好玩的黑白棋比赛实习，让我们在不断的研究中获得了很多的进步，收获了许多课外的知识，对课内的知识也有更多的了解，还有了与其他同学一起努力的机会。另外，还要感谢老师在我们设计算法时遇到困难时的解答，虽然代码最终还是有一些问题，但是有了老师的帮助之后，一些问题也得到了解决，再次感谢。

其次，感谢助教设计的可视化对战程序，让我们更方便的看出算法的问题所在。也感谢助教在这个过程中对我们的帮助，让我们对于算法的思路有了一定的了解。感谢。

还有，要感谢阎述辰同学，他开发的 visualchessboard 程序可以将复盘数据可视化，让我们更简单的看出算法问题出在哪里。感谢。

最后，要感谢那些靠谱的组员们，有了那些组员的支持，这个队伍才一直走到现在，很多事情互相帮助之下才获得了成果。也许我们的算法并不算是很好，但是我们都有努力的去

做过。虽然有些许遗憾，但我还是要说，没有他们也许情况会糟糕透顶。感谢他们一直以来都这个小组做出的努力。诚挚感谢！

6.4 参考文献

http://www.soongsky.com/
http://www.soongsky.com/strategy2/
http://www.cppblog.com/sandy/archive/2012/09/20/191377.html
http://jingyan.baidu.com/article/642c9d34e99dd5644a46f70c.html
http://tieba.baidu.com/p/1187602852

西区 West　　PAPA 组

7 数据结构与算法大作业报告

组长：　廖晨睿

组员：　谭陈东　陈哲萌　张喆　何宇　章美希

摘要

计算机的人工智能研究一直是人类一个富于挑战的实践，让计算机具有智能也合适科学家长久以来位置奋斗的课题。在计算机 AI 的研究中，让传统的棋类博弈游戏具有智能，一直是人们关注和研究的内容。棋类博弈人工智能，一般包括博弈搜索、局面评估两个方面。本论文就是通过对博弈算法的研究和分析，设计和实现一个可以和一般随机算法对弈的黑白棋对弈系统。所谓黑白棋，黑白棋又叫反棋，是一种通过 互相翻转对方棋子最后以棋盘上谁的棋子多来判断胜负。他的游戏规则简单，上手容易，非常适合。

研究主要解决了：

一、 黑白棋的估值函数的确定，该算法实验采用了估值表，通过对棋盘上不同格子进行赋值以决定其是否值得下。格子的值越大则表示下在该位置越有利，格子的值越小则表示下在该位置越不利。

二、　对博弈游戏的博弈树搜索算法有了全面的了解和掌握，实验研究中了解和应用了基本 Min-Max 算法。通过枚举所有可能走棋，对之后若干步的棋盘局势进行判断，从而决定当前的走子位置。

第一章　引言

黑白棋，又叫反棋（Reversi）、奥赛罗棋（Othello）、苹果棋或翻转棋。黑白棋在西方和日本很流行。游戏通过相互翻转对方的棋子，最后以棋盘上谁的棋子多来判断胜负。

它的游戏规则简单，因此上手很容易，但是它的变化又非常复杂。有一种说法是：只需要几分钟学会它，却需要一生的时间去精通它。

黑白棋是 19 世纪末英国人发明的。直到上个世纪 70 年代日本人长谷川五郎将其进行发展和推广，借用莎士比亚名剧奥赛罗（othello)为这个游戏重新命名（日语“オセロ”），也就是现在大家玩的黑白棋。为何借用莎士比亚名剧呢？是因为奥赛罗是莎士比亚一个名剧的男主角。他是一个黑人，妻子是白人，因受小人挑拨，怀疑妻子不忠一直情海翻波，最终亲手把妻子杀死。后来真相大白，奥赛罗懊悔不已，自杀而死。黑白棋就是借用这个黑人白人斗争的故事而命名。

黑白棋的棋盘是一个有 8*8 方格的棋盘。下棋时将棋下在空格中间，而不是像围棋一样下在交叉点上。开始时在棋盘正中有两白两黑四个棋子交叉放置，黑棋总是先下子　。

1.1 下子的方法

把自己颜色的棋子放在棋盘的空格上，而当自己放下的棋子在横、竖、斜八个方向内有一个自己的棋子，则被夹在中间的全部翻转会成为自己的棋子。并且，只有在可以翻转棋子的地方才可以下子。

1.2 棋规

1．棋局开始时黑白棋各两子交叉位于棋盘中央‘

2．黑方先行，双方交替下棋。

3．一步合法的棋步包括：在一个空格新落下一个棋子，并且翻转对手一个或多个棋子。

4．新落下的棋子与棋盘上已有的同色棋子间，对方被夹住的所有棋子都要翻转过来。可以是横着夹，竖着夹，或是斜着夹。夹住的位置上必须全部是对手的棋子，不能有空格。

5．一步棋可以在数个方向上翻棋，任何被夹住的棋子都必须被翻转过来，棋手无权选择不去翻某个棋子。

6．除非至少翻转了对手的一个棋子，否则就不能落子。如果一方没有合法棋步，也就是说不管他下到哪里，都不能至少翻转对手的一个棋子，那他这一轮只能弃权，而由他的对手继续落子直到他有合法棋步可下。

7．如果一方至少有一步合法棋步可下，他就必须落子，不得弃权。

8．棋局持续下去，直到棋盘填满或者双方都无合法棋步可下。

1.3 胜负规则

如果玩家在棋盘上没有地方可以下子，则该玩家对手可以连下。双方都没有棋子可以下时棋局结束，以棋子数目来计算胜负，棋子多的一方获胜。

在棋盘还没有下满时，如果一方的棋子已经被对方吃光，则棋局也结束。将对手棋子吃光的一方获胜。

翻转棋类似于棋盘游戏“奥赛罗 (Othello)”，是一种得分会戏剧性变化并且需要长时间思考的策略性游戏。

翻转棋的棋盘上有 64 个可以放置黑白棋子的方格（类似于国际象棋和跳棋）。游戏的目标是使棋盘上自己颜色的棋子数超过对手的棋子数。

该游戏非常复杂，其名称就暗示着结果的好坏可能会迅速变化。

当游戏双方都不能再按规则落子时，游戏就结束了。通常，游戏结束时棋盘上会摆满了棋子。结束时谁的棋子最多谁就是赢家。

1.4 关于黑白棋的算法

1、MinMax

2、Max

3、Min

4、NegaMax

5、AlphaBeta

6、AlphaBeta(With history)

7、Dynamic Programming

8、Monte Carlo

9、 λ Temporal Difference

10、AlphaBeta(withhash,history)

11、Reversi Learning

12、ChooseAction

第二章 研究方法

实验主要通过对老师所给的程序进行再编程，在已知可以下的位置上尽量寻找最优解，以达到尽可能高的胜率。

2.1 实验程序一：

```
# -*- coding: utf-8 -*-
from chessboard import *

# cb 为棋盘
# ms 为函数可以利用的存储字典，其中 ms['log']是下棋历史
#这是我们组的算法

def play(cb,ms):
    prec = {'A1':100,'A2':-5,'A3':10,'A4':5,'A5':5,'A6':10,'A7':-5,'A8':100,'B1':-5,'B2':-45,'B3':1,'B4':1,'B5':1,'B6':1,'B7':-45,'B8':-5,'C1':10,'C2':1,'C3':3,'C4':2,'C5':2,'C6':3,'C7':1,'C8':10,'D1':5,'D2':1,'D3':2,'D4':1,'D5':1,'D6':2,'D7':1,'D8':5,'E1':5,'E2':1,'E3':2,'E4':1,'E5':1,'E6':2,'E7':1,'E8':5,'F1':10,'F2':1,'F3':3,'F4':2,'F5':2,'F6':3,'F7':1,'F8':10,'G1':-5,'G2':-45,'G3':1,'G4':1,'G5':1,'G6':1,'G7':-45,'G8':-5,'H1':100,'H2':-5,'H3':10,'H4':5,'H5':5,'H6':10,'H7':-5,'H8':100}
    lp = cb.getLegalPos()
    if lp == []:
```

```
        return 'PASS'
    elif len(lp)=[1]:
        return lp[0]
    else:
        max = 0
        for x in lp:
            if prec[x]>max:
                max = prec[x]
                rtmax = x
            return x
```

该程序为一开始所初步设想的较简单的程序，只是通过对棋盘表面每个格子赋值，通过比较格子值的大小来进行决策的。相对来讲并不智能。

2.2 实验程序二：

T.papa 程序:

```
# -*- coding: utf-8 -*-
from chessboard import *

# cb 为棋盘
# ms 为函数可以利用的存储字典，其中 ms['log']是下棋历史
def play(cb,ms):

prec={'A1':90,'A2':-60,'A3':15,'A4':10,'A5':10,'A6':15,'A7':-60,'A8':90,'B1':-60,'B2':-8
0,'B3':10,'B4':20,'B5':20,'B6':10,'B7':-80,'B8':-60,'C1':15,'C2':10,'C3':25,'C4':2,'C5':2,'
C6':25,'C7':10,'C8':15,'D1':10,'D2':20,'D3':2,'D6':2,'D7':20,'D8':10,'E1':10,'E2':20,'E3':
2,'E6':2,'E7':20,'E8':10,'F1':15,'F2':10,'F3':25,'F4':2,'F5':2,'F6':25,'F7':10,'F8':15,'G1':-6
0,'G2':-80,'G3':10,'G4':20,'G5':20,'G6':10,'G7':-80,'G8':-60,'H1':90,'H2':-60,'H3':15,'H4
```

```
':3,'H5':3,'H6':15,'H7':-60,'H8':90}
    def getmax(lst,dic):#从可行列表和其对应的价值字典中找出最大价值和对应步骤
        if lst==[]:
            return (None,0)
        maxstep=lst[0]
        maxvalue=dic[lst[0]]
        for i in lst:
            if dic[i]>maxvalue:
                maxvalue=dic[i]
                maxstep=i
        return (maxstep,maxvalue)
    def culvalue(cb,dep,m):#用于找出可行位置中价值最大的位置的函数，参数为棋盘，递归深度和控制加减的 m
        lp=cb.getLegalPos()
        avalue={}
        if dep==0:
            for i in lp:
                avalue[i]=prec[i]*m
        else:
            dep=dep-1
            for i in lp:
                oppcb=ChessBoard(cb)
                oppcb.makeTurn(i)
                opplp=oppcb.getLegalPos()
                (maxstep,maxvalue)=culvalue(oppcb,dep,m*(-1))#递归至下一层
                avalue[i]=m*(prec[i]+maxvalue)#总价值为位置价值+递归得到的价值
        return getmax(lp,avalue)
    lp=cb.getLegalPos()
    if lp==[]:
        return 'PASS'
    elif len(lp)==1:
```

```
        return lp[0]
    else:
        (maxstep,maxvalue)=culvalue(cb,2,1)#三个参数，棋盘，递归深度，m=1表示我方落子价值为正
        return maxstep
```

#程序内包含两个主要函数，getmax 和 culvalue

#getmax 为在已知价值的可行步骤中找出价值最大的步骤，接收参数为可行步骤的列表和包含每步对应价值的字典，返回值为价值最大的步骤及其最大价值

#culvalue 为计算每个可行位置价值的函数，是一个递归函数。接收参数为棋盘、递归深度和控制加减的 m（m 为正，表示我方下棋，将加上对应价值；m 为负，表示对方下棋，将减去对应价值）。在一次递归中，先根据递归层数模拟双方轮流下棋。当递归层数为 0 时，返回该点的固有价值。倒数第二层的价值则为这一层的固有价值加上底层中的“最大价值”（由 getmax 获得。culvalue 底层为我方，则为正数，底层是对方则为负数），由此逐渐递归到顶层，即可返回总价值最大的一步棋。

第三章 实验结果

3.1 实验数据

实验环境说明 Intel i5-4200H 8.00GB
Windows 7 专业版
Python 2.79

1000 次对弈结果数据汇总

黑 \ 白	idiot 算法	gambler 算法	PAPA
idiot 算法	497:495 32582:31418 20.010:20.056	672:325 42060:21940 21.130:18.450	0:1000 15000:49000 10.000:2012
gambler 算法	383:617 23868:40132 11.013:18.582	497:495 32582:31418 20.010:20.056	110:886 9863:54137 15.054:3240
PAPA	1000:0 44000:22000 2280:14.83	732:181 48812:15188 4120:20.000	0:1000 28000:36000 3234:3238

3.2 结果分析

从结果来看，算法对 gambler 占几乎压倒性的优势。由于本身与 idiot 本身都不会随机落子，因此算法与 idiot 的 1000 次测试结果除时间外完全相同。

本算法的优势在于内置了经过反复验证的优先级表，并通过递归遵照优先级表进行价值预估，对于每一步落子的收益判断较准确。实验结果证明了优先级表一定的科学性。

考虑总时长落在 2000s-4000s 之间，具体的样本也偏离不远。由于没有采用较为常规而复杂的剪枝平衡算法，算法效率很高，每步考虑时间较短。

当然在与其他组的代码的交锋中，发现优先级算法本身比较死板，而且采用固定的优先级在瞬息万变的棋局中有局限性，因此胜率一般，比赛中也位列小组第三未能出线。

第四章 实习过程总结

4.1 分工与合作

第一次组会首先 6 位成员相互认识熟悉，自我介绍，并确定了初步分工：由廖宸睿、陈哲萌负责算法，其余四人负责代码。之后的组内交流大多在微信群中以及组员间进行。组员将自己的思路以及成果上传的微信群中，合作完成任务。

4.2 经验与教训

由于组员们本身对于黑白棋所知不多，平时不甚了解，花费了比较多的时间在学习、了解黑白棋上，因此小组合作前期进展较慢，后期程序的编写略显仓促。当然，组员们通力合作，从无到有，从黑白棋基础几乎为零到能够合作编写出有一定智能的黑白棋程序，本身就十分值得肯定。而且，组员们在合作中反映的态度与积极性，也许比程序本身更加值得骄傲。

4.3 建议与设想

此次黑白棋竞赛是一个非常好的形式，既部分取代了死板的考试，又以竞赛的形式考察了同学，调动了积极性，是一次十分不错的尝试，组员们也乐在其中，希望以后的数算课程中这样的形式坚持下去。有了今年的基础，黑白棋基础设施代码也有了一定的完善，希望以后参与其中的学弟学妹们能取得更好的成绩，编写出更加强大的程序。

致谢

首先感谢全体组员这段时间的付出和努力，好在这一切并没有白费。

同时也要感谢其他组中参加此次竞赛的同学。正是因为有了平时的交流，测试，甚至相互指导，才能让我们最终的程序得以出炉。

当然也不能忘了别出心裁组织此次竞赛的老师以及参与组织、场务等工作的同学们。

最终，再次感谢战胜了自己，收获了回报的 PAPA 组全体组员，有了每一个人的努力，才有了这个程序和这篇报告。感谢大家！

参考文献

杜秀全 程家兴 《博弈算法在黑白棋中的应用》
黑白棋天地网站
黑白棋百度百科
黑白棋 wiki 百科

西区 West　　INDIA 组

8 数据结构与算法课程 INDIA 小组实习作业报告

赵琰喆* 刘志扬 贾博 黄知劼 熊建学 向伟民

【摘要】:
本次数据结构与算法课程大作业是“黑白棋”。我们小组对程序的总体设计思路是递归搜索结合局势评估。递归搜索部分采用博弈过程的极大极小搜索思想，搜索每次落子后对我方局势最有利或者最不利的下法（利用局势价值量化决策）；同时为了提高搜索效率，我们利用了“Alpha-Beta 剪枝”操作对递归搜索次数进行了优化。局势评估部分采用控制“保护子”、“行动力”、“散度”等黑白棋策略（分别做了三个程序），将基于此评估的价值与棋格的初始价值进行线性组合，得到当前局势的总价值。线性组合的比例系数是通过大量试验选择的。代码中没有 “树”的数据结构，但是整个算法策略却是“树”的思想的体现。在我们的实验中，本算法与 idiot、gambler 对战，表现出了很好的智能。在黑白棋比赛中，我们更是夺得了亚军的成绩。本篇报告对我们的算法思想、程序代码、测试结果、以及大作业的体悟等做了详尽、具体地分析和阐述。

【关键字】:
极大极小搜索　Alpha-Beta 剪枝　保护子　行动力　散度　棋格赋值

8.1 算法思想

8.1.1 总体思路

算法总体思想是极大极小搜索和局势评估。

1、有关极大极小搜索：

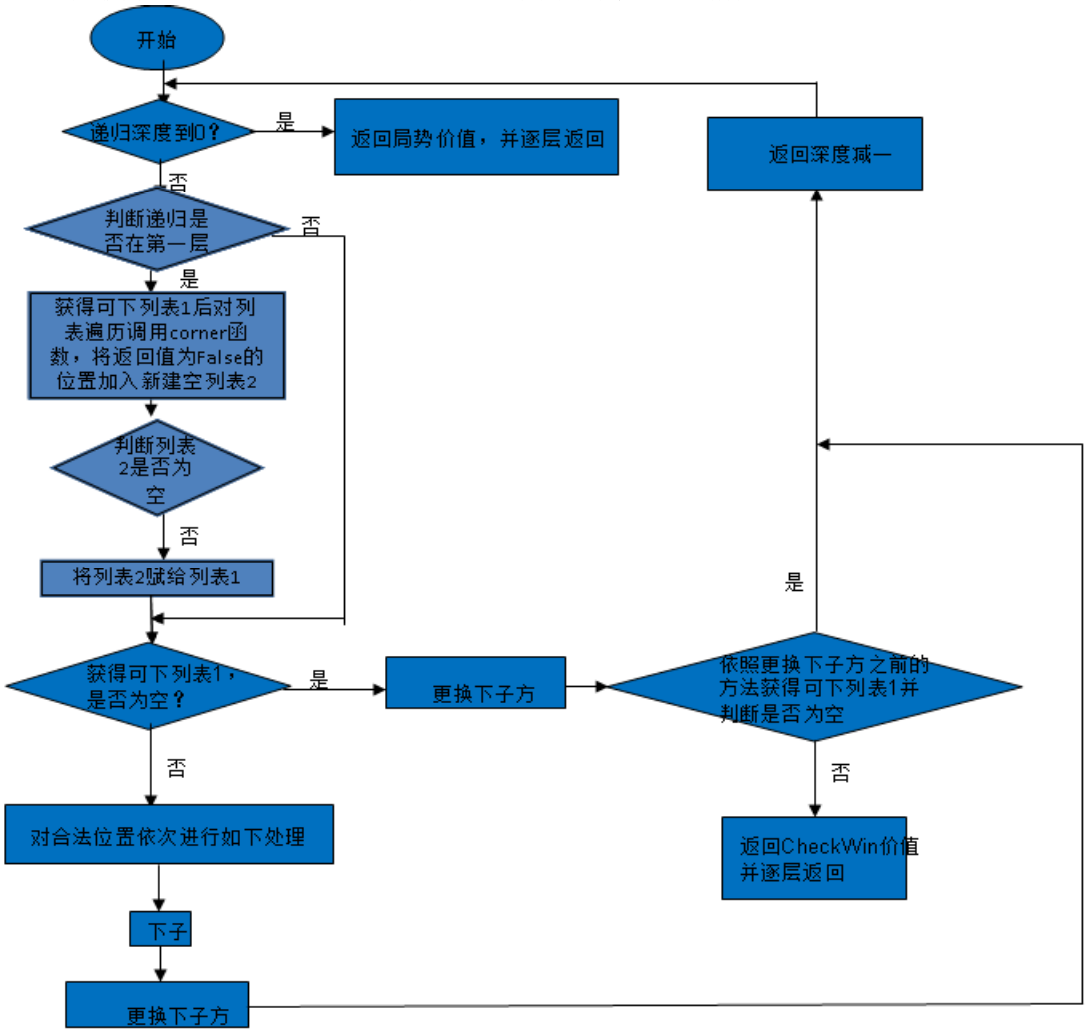
这是一个递归的过程，用“树”比较好理解（但是用其实现并不容易）。考虑不同棋步所构成的树，树的第一层，即根节点下一层为当前所能走的棋步。第 i 层节点的估值由 i+1 层节点的估值确定，最底端（叶节点）的估值由估值函数确定。递归过程中使用极大极小搜索，并用 Alpha-Beta 剪枝，实现对递归次数的优化。最终返回估值最高的节点所对应的下子位置。

2、有关局势评估：

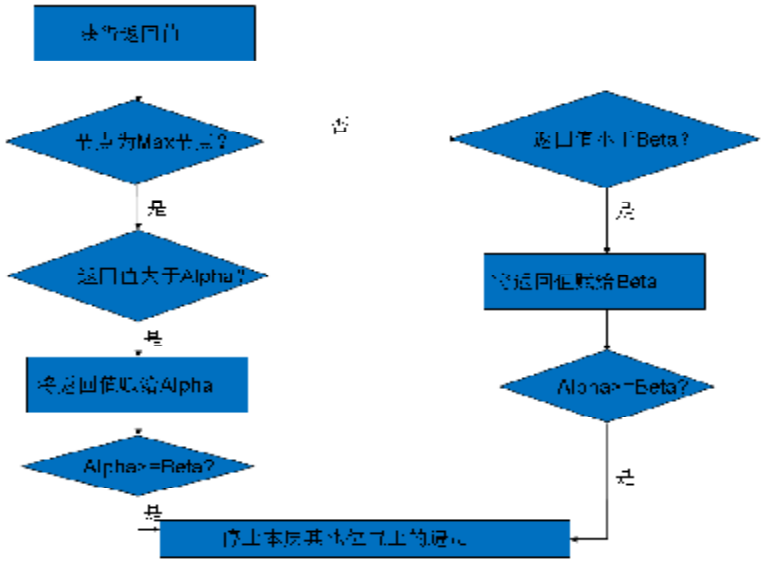
这个部分有四个基本的概念：保护子、行动力、散度、棋格赋值。我们最初的想法体现在 INDIA1 和 INDIA2 里，即将保护子或行动力的大小，按一定的比例和特定位置的棋格价值加权，得到最终量化形势评估。其中特定位置的棋格价值会按照一定权重赋值，比如边、角等特殊位置的价值会稍微大或稍小一些。在进入四强之后，我们又考虑了“散度”，即一个棋子周围的空格数，并且考虑了根据目前下子的步数来调用不同的算法。具体的操作在函数部分会详细说明。

8.1.2 算法流程图

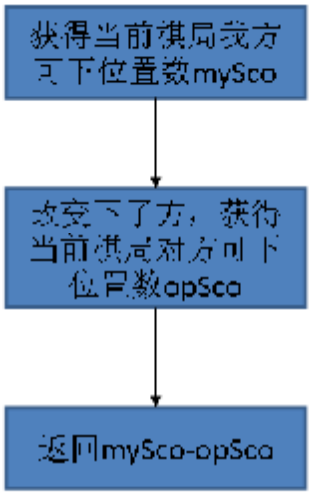
1、递归搜索流程图（在 INDIA3 中用，其余两个代码比此稍简易）



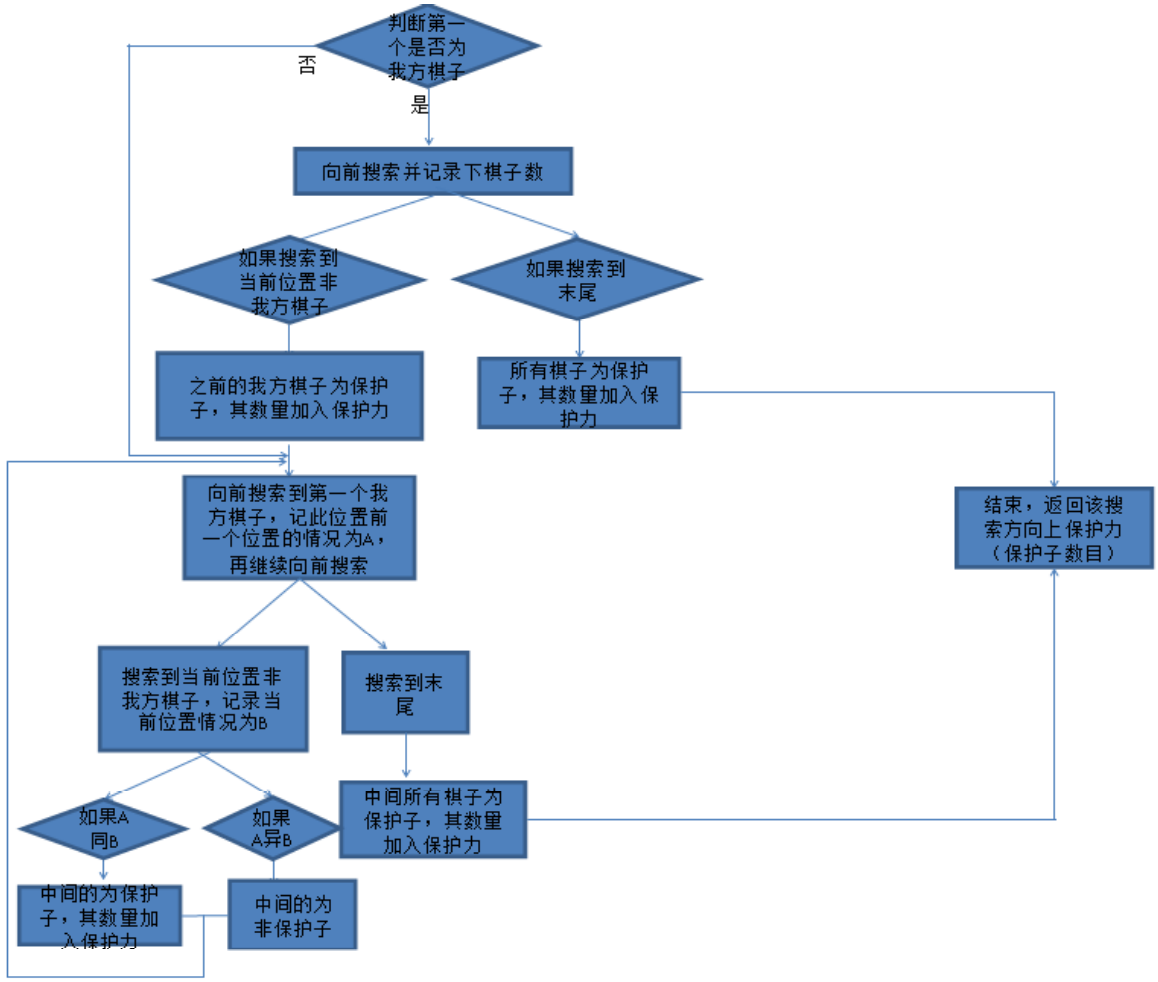
2、Alpha-Beta 剪枝部分流程图：



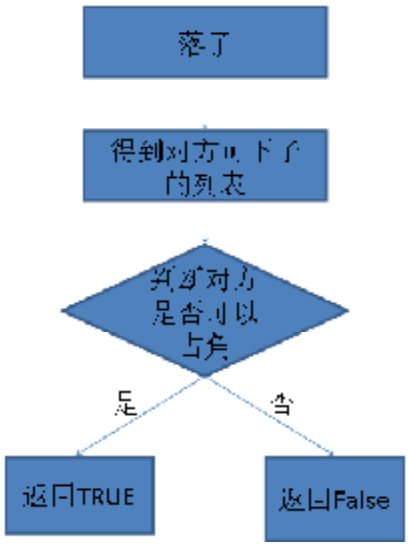
3、行动力流程图



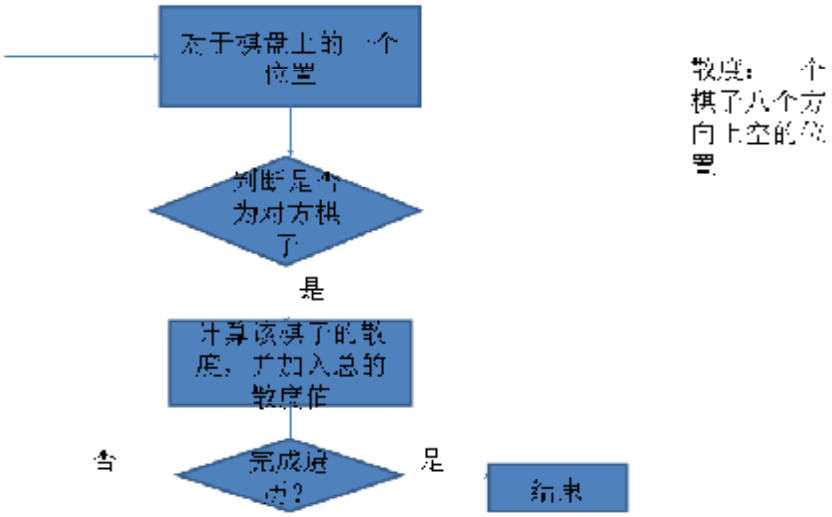
4、保护子函数流程图


判断第一个是否为我方棋子
否
是
向前搜索并记录下棋子数
如果搜索到当前位置非我方棋子
如果搜索到末尾
之前的我方棋子为保护子，其数量加入保护力
所有棋子为保护子，其数量加入保护力
向前搜索到第一个我方棋子，记此位置前一个位置的情况为A，再继续向前搜索
结束，返回该搜索方向上保护力（保护子数目）
搜索到当前位置非我方棋子，记录当前位置情况为B
搜索到末尾
如果A同B
如果A异B
中间所有棋子为保护子，其数量加入保护力
中间的为保护子，其数量加入保护力
中间的为非保护子

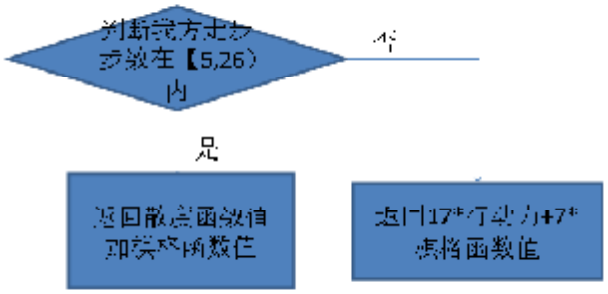
5、Coner 函数：


落了
得到对方下了的列表
返回TRUE
返回False

6、散度函数：



7、局势价值返回部分判断（INDIA3 中用到，INDIA1 与 INDIA2 不用判断）：



8.1.3 算法运行时间复杂度分析

算法运行时间主要消耗在递归搜索的部分。如果不加剪枝操作，那么每层搜索的步骤随着层数的递增呈指数增长。假设每步下棋后都有 6 个可下位置，那么搜索 n 层，第 n 层就有 6^n 个位置需要搜索，相当于调用 6^n 次估值函数。而我们做过实验，每次下子后一般情况有 6~10 个可下位置，递归搜索的复杂度可想而知。加入剪枝操作，我们估计递归搜索的复杂度在 O（k^m）左右，其中 k 为可下子步数，大约在 6~10 之间，n 为递归搜索的深度。m<n，因为剪枝操作可以避免某些不必要的递归过程，减少也是指数级别的。

算法估值部分的时间复杂度是常数级别的。保护子函数对棋盘进行了 4 次遍历，相当于判断 256 次。赋值的步骤必然少于 256 次。行动力函数只对棋盘的可下位置列表进行了取长度的操作，与递归深度无关。散度函数对棋盘进行一次遍历，对对方棋子进行统计，其次数也必然小于 512（每个棋子 8 方向判断）。但是估值部分叠加上递归操作部分，时间就会大大加长。

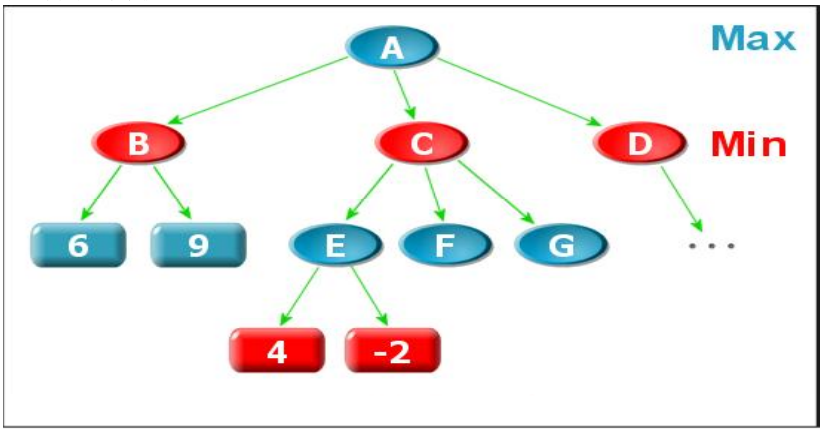
8.2 程序代码说明

8.2.1 数据结构说明

本算法主要用到的是“树”的思想，代码中没有建立树的代码，但是实质上是从“树”的思想得到启发，将建树的过程改为递归搜索。同时用了 Alpha-Beta 剪枝对搜索的过程进行了优化，提高了搜索效率。具体说明如下：

树的根节点是下子之后可以获得的最大价值，第一层为本方下子点，以下几层是对方下子点、本方下子点交替出现。采用极大极小搜索策略，假设参与博弈的两个人为 Black 和 White，从 Black 的角度看，Black 每一步行动总是争取自己的最大收益，称为 Max 过程，对应博弈树上的节点称为 Max 节点；由于 Black 对 White 的棋力并不清楚，所以只能认为 White 的每一步总是使自己的收益最小，称为 Min 过程，对应博弈树上的节点称为 Min 节点；于是下棋其实就是 Max 和 Min 交替进行的过程。对于一般的极小极大搜索，即使每一步只有很少的下法，搜索位置的数目也会随着搜索深度呈指数级增长。在大多数的中局棋形中，平均每一步都有十种走法，如果向前搜索的深度为九步，那么为了进行一步着子，访问的棋局数就达到了十亿个，一般的电脑是很难胜任的。因此我们采用 Alpha-Beta 剪枝。

为了得出关 Alpha-Beta 剪枝一般性的结论，我们先看一个具体的例子。请考虑下图中的博弈树。我们假设按照重左到右的顺序搜索同层节点。A 是 Max 节点，它将被赋值成 B、C、D 中的最大值。为了知道 A 的值，首先需要知道 B 的值，为此我们搜索 B 的子节点，并将 B 赋值为 6，因为 B 是 Min 节点，它的值为它子节点的最小值。然后开始决定 C 的值，我们首先搜索 E，E 的两个子节点的值分别为 4 和-2，E 是一个 Max 节点，所以 E 的值为 4。因为 C 是 Min 节点，所以我们知道 C 的值肯定小于或等于 4，所以 C 不会被 A 选中，因为 A 是 Max 节点，而现在已经有一个子节点 B 值大于 4。所以就没有必要在搜索 F 和 G 了。也即发生了剪枝。算法实现时为每个节点引入两个值：α和β，分别表示该节点估值允许的下限和上限，两者构成了一个区间，这个区间被称为 α-β 窗口，窗口的长度定义为 $\beta-\alpha-1$。初始时，根结点的 $\alpha=-\infty$，$\beta=\infty$，其他节点的α，β值是深度优先搜索过程中从它的父节点继承过来的。对上面的例子，当搜索完 B 之后，我们知道 A 最终的估值肯定不会小于 B 的值（因为 A 是 Max 节点），这个下限就是 A 的α值，α=6。继续进行深度优先搜索，当搜索完 E 之后，C 的上限也就确定了（C 是 Min 节点），此时 C 的β值是 4，注意到 C 的α也是 6（这是从 A 那里继承的），由前面的讨论知，此时 C 发生了剪枝，事实上，当β≤α时，就会发生剪枝。



Alpha-Beta 剪枝示意图

8.2.2 函数说明

函数分搜索函数和估值函数两大部分，具体的每一步的含义在注释中详尽表明，此处介绍函数的功能和主要的操作过程。

一、递归搜索部分：

1、Corner 函数（对方占角判断）：

参数：

cb：棋盘（ChessBoard）

pos：位置

操作过程：

调用 cb.maketurn(pos)在 pos 处落子

调用 lp=cb. getLegalPos()得到在 pos 处落子后对方可下位置

检查 lp 中是否有可以占角的可下位置，有返回 True，没有返回 False

2、AlphaBetaSearch 函数：

参数：

depth：当前递归深度，每向下递归一次就减一。

board：棋盘（是 ChessBoard 类）。

alpha：每个节点的价值下限，max 节点向父节点 min 节点返回值。

beta：每个节点的价值上限，min 节点向父节点 max 节点返回值。

color：递归到当前的下子方，0 代表白子，1 代表黑子，每向下递归一次，color 均变为 1-color。

color=CompColor 时为 max 节点，否则为 min 节点。

CompColor：程序下子方，递归过程中不变。

DepthMax：递归总深度，递归过程中不变。

lenth:（仅 INDIA3 程序使用这个参数）：当前下棋回合（指单方回合数，0~32）

操作过程：

首先，仅在 INDIA3 中使用了下面的步骤：

当 lenth<26 时，调用 Corner 检查 lp 中的每个位置是否会令对方占角，“否”就将此位置加到列表 lpp 中。如果 lpp！=lp，令 lp=lpp（从而在前 26 手最大避免对方占角。）

以下步骤是所有程序中一般的搜索步骤

（1）、depth 是 0 时，就用估值函数 evalue_c 估值。

（2）、depth 不是 0，情况如下：

a、如果当前下子方无子可下（lp==0），更换下子方，即把节点的 max 属性和 min 属性互换。如果互换后仍无子可下，说明此时棋局结束，用 CheckWin 检查胜负，返回评估值；如果可以继续，就继续调用递归函数估值。

b、当前下子方可下子时，遍历合法下子位置，遍历的方式是模拟下子，然后在下后棋局再次调用函数。这相当于建立若干子树。

（3）、min 节点对方下子，要降低我方优势，所以价值最大值 beta 被一再减小，最后返回给父节点的是最小的 beta 值（可能让对方得到的最大价值的最小值）。节点同理，max 节点的 alpha 值被不断增大，返回给父节点的是最大的 alpha 值。如果 depth=DepthMax，表明递归回到最上层，此时把位置记录下来（Pos=move）并返回。

二、估值函数部分：

1、CheckWin 函数：

参数：

board：是储存棋盘的嵌套列表。

color：是程序下子方。
操作过程：
对棋盘进行遍历，统计双方得子。此时的得子数已经决定了胜负，代表了最好或者最坏的情况，所以返回值里有一个 1000 倍的因子来扩大这种价值优劣势。

2、envalue_c 函数：
参数：
board：存有当前棋盘的嵌套列表
joinList：存有下棋双方的列表，joinList[0]为我方，joinList[1]为对方
lenth（仅 INDIA3 程序使用这个参数）：当前下棋回合（指单方回合数，0~32）
操作过程：
INDIA1：调用估值函数：行动力估值函数、棋格赋值函数。返回：17*行动力函数估值+7*棋格赋值函数估值。
INDIA2：调用估值函数：棋格赋值函数、保护子估值函数。返回：2*保护子函数估值+棋格赋值函数估值。
INDIA3：当 lenth>7 并且 lenth<25 时，调用：散度估值函数、棋格赋值函数。返回：散度函数估值+棋格估值函数估值。
当 lenth<=7 或者 lenth>25 时，调用估值函数：行动力估值函数、棋格赋值函数。返回：17*行动力函数估值+7*棋格赋值函数估值。

3、保护子函数 protect_envalue：
参数：
board：存有当前棋盘的嵌套列表
joinList：存有下棋双方的列表，joinList[0]为我方，joinList[1]为对方
操作过程：
保护子：若在某一个搜索方向上该棋子不会被吃掉，则被称为保护子。
首先是对棋盘做四次搜索，分别为按横排，竖排，两个斜排方向。在每一次的搜索中，定义一个保护力变量，若是遇见一个保护子，变量加 1，否则不变。首先判断第一个位置是否是我方棋子，若是，则保护力加 1，如果不是，则移动到下一个位置。当第一次搜索到我方棋子时，将当前位置前一个位置的情况记录下来，然后继续往前搜索，并创建一个临时变量记录从现在开始搜索到的我方棋子，如果不是我方棋子，则搜索暂停，并记录下当前位置的情况，并且与已经记录下的情况做比较，如果情况相同，则中间的我方棋子都被保护（在当前的搜索方向上），并且将临时变量的值加进保护力，如果不同，说明中间的能在这一方向上可以被吃掉，就不是保护子，然后清空临时变量，继续按照上述方式向前搜索直到结束。如果第一个是我方棋子，则在向前搜索到不是我方棋子的时候暂停，这之前的棋子在这一搜索方向上就都是被保护的，将其加入保护力中。

4、行动力函数 forward_envalue：
参数：
board：存有当前棋盘的 ChessBoard()类
joinList：存有下棋双方的列表，joinList[0]为我方，joinList[1]为对方
操作过程：
将 board.turn 设为 joinList[0], board.opturn 设为 joinList[1]，调用 board.getLegalPos()得到我方在当前棋局的可走位置的数目 mySco，将 board.turn 设为 joinList[1], board.opturn 设为 joinList[0]，调用 board.getLegalPos()得到对方在当前棋局的可走位置的数目 opSco，返回 mySco-opSco

5、散度函数 div_envalue：
参数：
board：存有当前棋盘的嵌套列表

op：对方棋子
操作过程：
散度：一个棋子周围有几个空格就有几个散度
首先遍历棋盘，找到对方棋子。每找到一个对方棋子，计算它的散度。然后将所有对方棋子的散度加起来，返回给调用函数。

6、棋格赋值函数 qige_envalue：
参数：
board：存有当前棋盘的嵌套列表
joinList：存有下棋双方的列表，joinList[0]为我方，joinList[1]为对方
操作过程：
首先建立一个与棋盘对应的 8*8 的矩阵，矩阵的每一个元素代表棋盘上相应位置的权重。矩阵称为赋值棋格。建立变量 score，表示本棋盘的得分。
然后扫描棋盘 board，每发现一个对方棋子，score 减去赋值棋格中的对应位置上的值；每发现一个我方棋子，score 加上赋值棋格中的对应位置上的值。
返回 score 给调用函数。

附：棋格估值表的具体值：

50	-1	5	2	2	5	-1	50
-1	-10	1	1	1	1	-10	-1
5	1	1	1	1	1	1	5
2	1	1	0	0	1	1	2
2	1	1	0	0	1	1	2
5	1	1	1	1	1	1	5
-1	-10	1	1	1	1	-10	-1
50	-1	5	2	2	5	-1	50

8.2.3 程序限制

程序的限制主要体现在超时。超时有三种情况：
1、递归深度过大。比如一般情况下，INIDIA1、INDIA2、INDIA3 可以承受的最大的递归深度（在总时间 160.0s 的限制下）分别为 4、5、4。而且在我们自己的电脑上跑时某些时候递归深度还需调整得更小。
2、这种情况是在和 DELTA 组的私下对决（四强确定之后）中发现的。如果对方考虑了每步留给我方的可下位置数，并且每次都留的比较多，程序就很容易超时。DELTA 组恰好考虑了这一点，所以所有采用递归搜索的代码在 DELTA 组面前基本上都易超时。

3、递归搜索时对可下位置的遍历顺序也会影响时间。剪枝操作对于节点的遍历顺序有很大的依赖性。比如在图 1 中，如果我们先检查 C 节点，再检查 B 节点，可能 C 的剪枝就不会发生，事实上并没有起到优化搜索的效果。但是这种“坏情况”和“好情况”发生的概率应该一样，所以限制性不会显得很强。

8.3 实验结果

8.3.1 实验数据

INDIA1:

实验环境说明：

- 刘志扬的电脑
- 硬件配置 : CPU : Intel(R) Core(R) i5-3230 CPU @ 2.60GHz 2.60GHz
  内存: 4.00 GB
- 操作系统 : Windows 8 中文版 64 位操作系统
- Python 版本：2.7.9

1000 次对弈结果数据汇总

黑 \ 白	idiot 算法	gambler 算法	INDIA1
idiot 算法	获胜局数比（平局） 0 : 1000 子数比 27.00 : 37.00 时间比 0.00793 : 0.0111 KO 0 : 0	获胜局数比（平局） 533 : 448 (19) 子数比 32.96 : 29.58 时间比 0.0170 :0.0138 KO 0 : 8	获胜局数比（平局） 0 : 1000 子数比 17.00 : 47.00 时间比 0.0173 : 30.872 KO 0 : 0
gambler 算法	获胜局数比（平局） 383 : 587 (30) 子数比 29.15 : 34.31 时间比 0.0140 : 0.0174	获胜局数比（平局） 465 : 493 (42) 子数比 31.63 : 32.02 时间比 0.0162 : 0.0161	获胜局数比（平局） 48 : 950 (2) 子数比 16.31 : 40.99 时间比 0.0117 : 43.459

	KO 1 : 0	KO 0 : 0	KO 1 : 0
INDIA1	获胜局数比（平局） 0 : 1000 子数比 44.00 : 20.00 时间比 47.420 : 0.0178 KO 0 : 0	获胜局数比（平局） 994 : 4 (2) 子数比 48.35 : 15.54 时间比 45.027 : 0.0113 KO 0 ： 6	获胜局数比（平局） 0 : 1000 子数比 30.00 : 34.00 时间比 79.477 : 87.434 KO 0 : 0

INDIA2:

实验环境说明：

- 曹越的电脑
- 硬件配置 : CPU : Intel(R) Core(R) i5-4210U CPU @ 1.70GHz 2.40GHz
  内存: 4.00 GB
- 操作系统 : Windows 8.1 中文版 64 位操作系统
- Python 版本：2.7.9

1000 次对弈结果数据汇总

黑 \ 白	idiot 算法	gambler 算法	INDIA2
idiot 算法	获胜局数比（平局） 0 : 1000 子数比 27.00 : 37.00 时间比 0.0108 : 0.0149 KO 0 : 0	获胜局数比（平局） 533 : 448 (19) 子数比 32.91 : 29.74 时间比 0.0184 : 0.0150 KO 0 : 8	获胜局数比（平局） 0 : 1000 子数比 2.00 : 25.00 时间比 0.00232 : 22.418 KO 0 : 0
gambler 算法	获胜局数比（平局） 383 : 587 (30) 子数比 28.81 : 34.66 时间比 0.0130 : 0.0160 KO 1 : 0	获胜局数比（平局） 465 : 493 (42) 子数比 31.65 : 31.99 时间比 0.0178 : 0.0178 KO 0 : 0	获胜局数比（平局） 155 : 841 (4) 子数比 13.99 : 45.60 时间比 0.00891 : 106.572 KO 9 : 0 (后手超时 142 次)
INDIA2	获胜局数比（平局） 0 : 1000 子数比 46 00 : 18.00 时间比 47.951 : 0.0111 KO 0 : 0	获胜局数比（平局） 885 : 113 (2) 子数比 47.76 : 12.78 时间比 94.278 : 0.00895 KO 0 : 36 (先手超时 95 次)	获胜局数比（平局） 0 : 1000 子数比 23.22 : 30.06 时间比 151.090 : 98.658 KO 0 : 0 (先手超时 62 次)

INDIA3:

实验环境说明：

- 贾博的电脑
- 硬件配置 : CPU : Intel(R) Core(R) i5-4210U CPU @ 1.70GHz 2.40GHz

内存: 4.00 GB

- 操作系统 : Windows 8.1 中文版 64 位操作系统
- Python 版本：2.7.9

1000 次对弈结果数据汇总

黑 \ 白	idiot 算法	gambler 算法	INDIA3
idiot 算法	获胜局数比（平局） 0 : 1000 子数比 27.00 : 37.00 时间比 0.00972 : 0.0139 KO 0 : 0	获胜局数比（平局） 533 : 448 (19) 子数比 32.88 : 29.77 时间比 0.0169 : 0.0138 KO 0 : 8	获胜局数比（平局） 1000 : 0 子数比 35.00 : 29.00 时间比 0.0183 : 40.340 KO 0 : 0
gambler 算法	获胜局数比（平局） 383 : 587 (30) 子数比 29.33 : 34.08 时间比 0.0239 : 0.0293 KO 1 : 0	获胜局数比（平局） 885 : 113 (2) 子数比 31.18 : 32.59 时间比 0.0161 :0.0161 KO 0 : 36	获胜局数比（平局） 179 : 800 (21) 子数比 23.62 : 38.93 时间比 0.0136 : 28.353 KO 2 : 3
INDIA3	获胜局数比（平局） 1000 : 0 子数比 32.00 : 28.00 时间比 16.271 : 0.0196 KO 0 : 0	获胜局数比（平局） 839 : 149 (12) 子数比 39.32 : 23.77 时间比 26.248 : 0.0137 KO 0 : 9	获胜局数比（平局） 1000 : 0 子数比 39.00 : 25.00 时间比 15.463 : 26.590 KO 0 : 0

备注：1、由于除了 gambler 外，其余的代码中均没有加入随机，所以 idiot-INDIA[i] 与 INDIA[i]-idiot 均只测试了 20 次，INDIA[i]-INDIA[i]只测试了 100 次（i=1,2,3）；
2、其余测试均做了 1000 次。idiot-idiot 的实验结果用来比较硬件性能。

测试数据说明：

总时间设为 160s；

由于测试硬件有一定差异，不论是否超时，时间、子数均计入总平均；如若超时，超时方判负，并记录超时次数。

8.3.2 结果分析

首先，在与 idiot 的对战中，INDIA1 的行动力策略与 INDIA2 的保护子策略都很占优势，先后手都可以大胜对手。尤其是 INDIA2，作为后手直接在 25：2 时结束了战局，效果很好。但是 INDIA3 的散度策略在后手时输给了 idiot，作为先手可以赢棋，这一点我们没有料到。我们分析应该是散度的策略对于一些比较智能的算法比较有用，比如 LIMA 组和 CHARLIE 组的算法，而总会被像 idiot 这样的简单算法所克制。这些是可以理解的。在我们自己的测试中，INDIA3 可以打败其他两个算法，也充分说明了代码会互相克制的客观事实。

其次，与 gambler 的对战结果就比较多样了。INDIA3 表现出了明显的时间优势，INDIA2 则充分暴露了由于保护子搜索步骤的繁琐造成的大量的时间消耗。不过三种策略对 gambler 都表现出了优势，而且平均得子差不多。

就算法时间消耗来看，INDIA2 是最长的。这个在意料之中，因为它不仅要进行四次的棋盘遍历，而且递归深度还是 5。不过有一点出乎了意料，就是 INDIA2 对 INDIA2 时，有连续的 62 次先手超时。按理

说没有加随机不能出现这种情况。后来分析原因，是当时电脑里整跑着另一个程序，所以造成了其 62 次的时间比之前长至少 30 秒。这点从原始数据可以看到。这也说明如果我们在比赛时加深递归，那么 INDIA2 很可能就超时了。INDIA1 和 INDIA3 虽然时间消耗少于 INDIA2，但是递归深度加到 5 之后也会与 INDIA2 相差无几。总的来说，三者的时间消耗都在递归部分，而且时间复杂度随深度呈指数增加。

8.3.3 经典棋局（可选）

由于本组的实习中，没有发现经典棋局，所以本部分略去。

8.4 实习过程总结

8.4.1 分工与合作

1、小组分工：
代码编写及说明：赵琰喆、刘志扬
程序测试及结果汇总：黄知劼
搜集资料和算法分析：贾博
算法思路编写及流程图制作：熊建学
小组工作记录：向伟民
报告整合：赵琰喆

小组分工只是分到负责人员，事实上，像测试程序、资料查询、报告编写等，都是需要大家通力配合、合作完成的。这一点上，小组成员做得非常好，都能积极、主动、按时地完成任务。

2、小组交流方式：

微信群讨论、平时的小组讨论会

3、历次组会记录：

6 月 2 号完成组队，并在微信创立小组，初步讨论大家对黑白棋以及实习作业的想法。

6 月 4 号 17 点 45 在理教 213 进行了第一次组会，由赵琰喆同学讲解，一起学习了老师的基础设施代码。同时初步定下了各自的分工，安排了工作的时间节点。

6 月 5 号下午开完组长会后，小组成员的任务分配有些许的变动。当晚编程小组（赵琰喆、刘志扬、贾博、黄知劼）进行了有关程序方面的讨论，并写好了测试程序的框架。5、6 号两天晚上加紧写出了剪枝操作和估值结合的初步的程序。

6 月 7 号 18 点在四教 309 进行了第二次组会，总结初步编好的程序的效果，并交流各自的黑白棋经验。这次讨论的目的主要是寻找优化“保护子”搜索的方法，并且希望在程序中加入更多的经验判断（启发式规则）。但是对于前者，最终并没有找到合适的优化方法（最后的优化是随后想到的）；对于后者，我们决定将经验的判断直接加到到棋格初始价值的赋值上，不再单独考虑。

6 月 13 号 18 点在理教 317 进行了第三次组会，主要讨论了热身赛的战果，根据热身赛的结果，对代码做了一些参数上的调整，同时制定了一些战术。这次会议主要目的是布置写报告的任务以及细节，并确定了汇总报告的最终时间。

6 月 15 号小组赛出线并成功晋级四强之后，我们又进行了一次讨论，主要是针对 CHALIE 改进程序。

8.4.2 经验与教训

有关实习过程的经验：

1、思路比较明确。

6 月 4 日周四课上老师布置了具体的作业之后，当天晚上我们开了第一次会，确定了程序设计的基本思想，即递归搜索、局势评估，并考虑基于经验的策略。

2、前期工作比较迅速。

6 月 5 日、6 日两天我们闭关两日研究，将行动力和保护子的估值函数，以及 Alpha-Beta 剪枝的雏形搞了出来，基本可以运行。这一点上比较高效。只要有了初步的程序，接下来的不断修改和完善的工作就比较容易了。

3、对程序持续改进，不断优化。

优化的主要是估值函数。最初我们每一步估值都要对棋盘遍历，而在某个棋子位置还要对八个方向分别检查。事实上这样的操作复杂度很高，据刘志扬同学的时间统计，此方法严重超时。随后我们想到，获得保护力并不需要知道每个棋子在哪些方向被保护，而只需要整个棋盘保护力的总值。这样一来，查找保护子的过程就和多种括号匹配的过程相似，从而借用括号匹配的实现来优化了代码。优化后的代码由于减少了多余的探索和赋值步骤，耗时约为原来的一半。

除此之外，我们还对各种估值方法和棋格初始价值的叠加比例做了很多实验，将调整后的和之前的作以比较，以找到最优配比。

4、创新思路，在激励中不断前进。

进入四强之后，我们分析了和 CHARLIE 组的实力差距，发现不能仅仅用当前的程序，所以加紧查资料。最后，我们从“散度”这个新的概念得到启发，开发了一套更优化的程序。不得不说，如果不是对手的强大，我们不可能有这样的进步；而如果没有不甘现状的上进心，我们也不可能得到最后的战果。所以我们必须感谢对手，但也更感谢我们自己。

实习过程中值得改进的地方：

1、关于小组工作：

（1）、我们组的讨论中没有对算法做具体的说明，可能使个别组员对程序并不是理解得很透彻，而真正专注于程序并深入理解的只是少部分人。这一方面可能使得最终写报告的重任还是主要由编代码的同学承担，任务分配不均匀，另一方面也不利于大家整体实力的提高。所以在小组工作中需要注意讨论中让组员及时了解工作的详细进程。

（2）、应该更合理地分配时间。因为前期以比赛和代码为中心，而且代码还在不断改进，所以没有着手写实验报告。这使得我们后期做测试、写函数说明等显得有些仓促。事实上，我们的总体思想以及递归搜索部分的函数说明应该随着代码的编写一并完成，之后着力进行程序的测试，这样会更省时间。最后让老师延期到 6 月 28 日，我们表示十分愧疚。

（3）、重视原始数据的保存。由于缺乏实验经验，我们第一次的测试只考虑了保留总体平均结果，忽视了原始数据的保存，非常失误。在老师提醒之后，我们才又加紧重做测试实验。没有原始数据，实验就缺乏可信度和说服力，这也给我们将来做科研好好地上了一课。

（4）、小组工作的细节应该更注意一些。比如测试程序保留每局时间的部分因为大意，只保留了两位小数。虽然总体平均时位数正常，但是原始数据里 gambler 的时间全部是 0.02、0.01，甚至显示为 0，这一点比较遗憾。我们在做其他测试时就把这一点改了过来。

2、关于代码：

（1）、程序中递归操作如果改成“树”的类，然后储存在 ms 列表中，就可以在下次搜索时直接找到上次搜索到的层，然后从子树中再次向下搜索，或许可以大大节省时间。不过因为实力有限，我们并没有在这个方法上深究。而且考虑到我们使用了 Alpha-Beta 剪枝操作，剪掉了很多节点，可能下一次调用时获得的子树是不完整的。所以我们使用相对简单的递归思路。

（2）、Alpha-Beta 搜索可以更加优化。正如前文所述，剪枝操作对节点顺序的依赖性很强，考虑到这一点，

我们或许可以对所有可下位置进行一下比较浅的遍历，然后依照结果对可下位置进行排序，可以得到最优的搜索顺序。

8.4.3 提出建议与设想

本次黑白棋大赛总体来讲是非常成功的，大家都热情高涨，积极参与。赛程总体进行得很好，但也稍有美中不足。针对竞赛中的一些不完善的地方，我们组有如下几点建议：

1、在组队方面，确立组长后，希望有一套类似组长对老师报名那样的系统，供组员对组长的报名。在大家选好组长后，再进行人数方面的调整。这可以避免有同学迟迟组不了队的情况。

2、对于竞赛规则的确立，基于本次黑白棋大赛的经验，下一次可以先开组长会议，再公布竞赛细则。这样可以避免规则总是变动，使同学们编写程序思路更明确。

3、要充分考虑比赛当天的时间因素。一些必要的准备工作，比如计分程序、统计分数的 ppt 等，可以提前给场务人员。最好可以提前演练，使正式的比赛更紧凑、流畅。

4、比赛当场，同学们等待结果的期间可以设置一些小的节目，比如人机对战的比赛等。这样会更吸引眼球，避免冷场。

除此之外，针对平时的教学，我们也建议老师可以调整一下讲课的节奏。在前期较简单的部分可以加快进度，比如栈、队列、排序与搜索等。后期的课程，特别是树、图及其相关算法部分，可以放慢一点，讲得更详细一点。而且本次的大作业其实用到“递归”的算法就足以做得很强了，树、图部分的算法却用得并不多。建议老师明年考虑针对这一块儿布置一次大作业，加强同学们的理解。

不得不说，这学期的数算课非常成功！陈斌老师的用心付出、同学们的热情配合，使得这一学期的数算课留下了很多美好的记忆。对即将到来的学弟学妹，我们想说，陈斌老师讲课生动，思路灵活，工作负责，还是常挂微信半夜答疑的好老师。数算大作业也是十分有趣，且富有挑战性。数算不是编程，数算课上获得的思维上的锻炼远比学会一门编程语言重要得多。希望你们能爱上数算，轻松对待这门课程。学长坦言，“这是我大学三年来最好的一门课，没有之一”，而我们作为大一新生，也是在陈斌老师的数算课上第一次体验到了大学课程的异彩纷呈。跟着陈斌老师绝对没错，学弟学妹快来吧！

8.5 致谢

首先要感谢陈斌老师和姜城、石瀚文助教。这一学期跟随陈斌老师的学习，让我们不仅收获了知识，更认识到了算法世界的奇妙多彩。老师和助教的教导与帮助是我们这次大作业的实力来源。非常感谢你们能创造这样一个好的大作业平台让我们锻炼自我，更感谢你们对我们的辛勤付出与无私帮助，使我们在数算课上取得了很大进步。

其次，要感谢组内所有同学：赵琰喆、刘志扬、贾博、黄知劼、熊建学、向伟民。大作业的完成需要小组的通力合作，而我们的小组非常团结，在大作业完成的过程中不仅各尽其职，而且十分热心、互帮互助。搜集资料、编写代码、完成测试、书写报告等等方面，大家都认真高效。组员们的配合是我们取得优异成绩的基础。

最后，感谢所有为我们提供过帮助的同学。这其中包括汪颖、石永祥、马赞彭、陈春含、蔡天泊等等和我们坦诚地交流经验，并无私地帮我们测试的同学；也包括曹越等为我们组的代码测试提供设备（电脑）保障的同学；更包括在黑白棋竞赛中不断做出好的程序、激励着我们不断前进的小组——KILO、CHARLIE、DELTA、LIMA……。当然，最重要的，是和我们共同度过一学期的数算课、与我们一起学习和进步的大家。你们是我们心中的力量！

8.6 参考文献

1、柏爱俊：《几种智能算法在黑白棋程序中的应用》。中国科学技术大学，2007 年 10 月；
2、王浩：《Visual C++游戏开发经典案例详解》。清华大学出版社，2010 年 6 月第一版；
3、杜秀全、程家兴：《博弈算法在黑白棋中的应用》。《计算机技术与发展》，2007 年 1 月 10 日。
4、黑白棋天地： http://www.soongsky.com/；
5、《黑白棋：学会一分钟，精通一世功》： http://www.soongsky.com/strategy2/

西区 West　　KILO 组

9 数据结构与算法课程实习作业报告

（朱贺* 汪诗舜 蓝坤 党卓 胡哲 韦春婉）

摘要：T_KILO 采用主流的对弈程序结构框架，通过基于深度优先搜索的估值函数来确定最优的落子位置，估值过程综合了影响棋局的多种因素，如角位、边位、稳定子、楔入子、棋手行动力等。算法主体基于改进的图，使用估价函数对落子位置进行估值，选取估价最高的位置进行落子。经过实验证明，这种算法在对阵 idiot 与 gambler 算法时，至少可以获得 85%以上的胜率。

关键字：图，深度优先搜索，alpha-beta 剪枝算法

9.1 算法思想

9.1.1 总体思路

9.1.1.1 算法思路说明

本算法将黑白棋的棋盘看作一张图，棋盘上的每个格子都作为图的一个顶点，但是和 ADT Graph 所不同的是，每个顶点都有其自己的权重。算法的基础就是一个由这些权重值所构成的二维列表，在下棋时，会根据不同的情况，对每个顶点的权重值进行相应的修改，并进行估值，最终每一步取估值最大的格子下子。

当轮到我方下棋时，相应会传入棋盘权重的参数，此时先判断角位置 A1、A8、H1、H8 是否被对方或我方占领，如果有角位被占领，则重设角区域（角位及与角位相邻的三个位置，例如 A1 角位被占就调整 A2、B1、B2）的初始估值，具体调整方法见 2.2 中的 cbEvalRet 函数。在检查完角位之后，便根据此时棋盘上的合法落子位置寻找最佳的下棋位置（findBestPos 函数），具体有三种检查策略：

1、 **随机策略（random strategy）**：当棋盘落子数小于给定值时，随机在棋盘中间区域（中间 4 行 4 列）落子，节省时间，同时避免给对方占边的机会。如果随机策略失败，即没有找到合适位置，则提前开启普通策略

2、 **普通策略（normal strategy）**：运用递归算法，按照普通搜索深度进行搜索，在进入中盘后增加深度，得到在给定搜索深度内的最佳估值（miniMax 函数，使己方得到最大估值，使对方得到最小估值），这是一个深度优先的搜索，此处运用了 alpha-beta 剪枝算法对整体过程进行了优化。

3、 **终极策略（final strategy）**：当剩余空位数小于或等于终极搜索深度时，直接搜索到游戏结束

（miniMax2End 函数），此处同样是深度优先搜索，并且使用了 alpha-beta 剪枝算法进行优化。

在计算最佳位置时，还有一个防超时补丁。当总运行时间超过 150 秒后，会自动转入 idiot 算法进行下棋。

综上所述，算法所采用的主体数据结构就是一个用二维列表实现的类似于图的结构，在每个顶点上赋予了权值，在下棋过程中，如果角位被占，则调整权值。在算法以及算法策略上，当程序计算下棋的位置时，会使用类似搜索树的深度优先搜索方式，按照给定的递归深度（执行过程中可变），根据各个位置的权值以及在这些位置落子所会取得的效果（稳定子、楔入子、对对方自由度的限制等）来进行递归，对各个位置估值，然后选择对己方估值最大的位置落子。在进行递归的过程中，还用到了队列等数据结构，此处不做赘述。各个函数的执行策略见 2.2 节。

9.1.1.2 算法所使用的常数

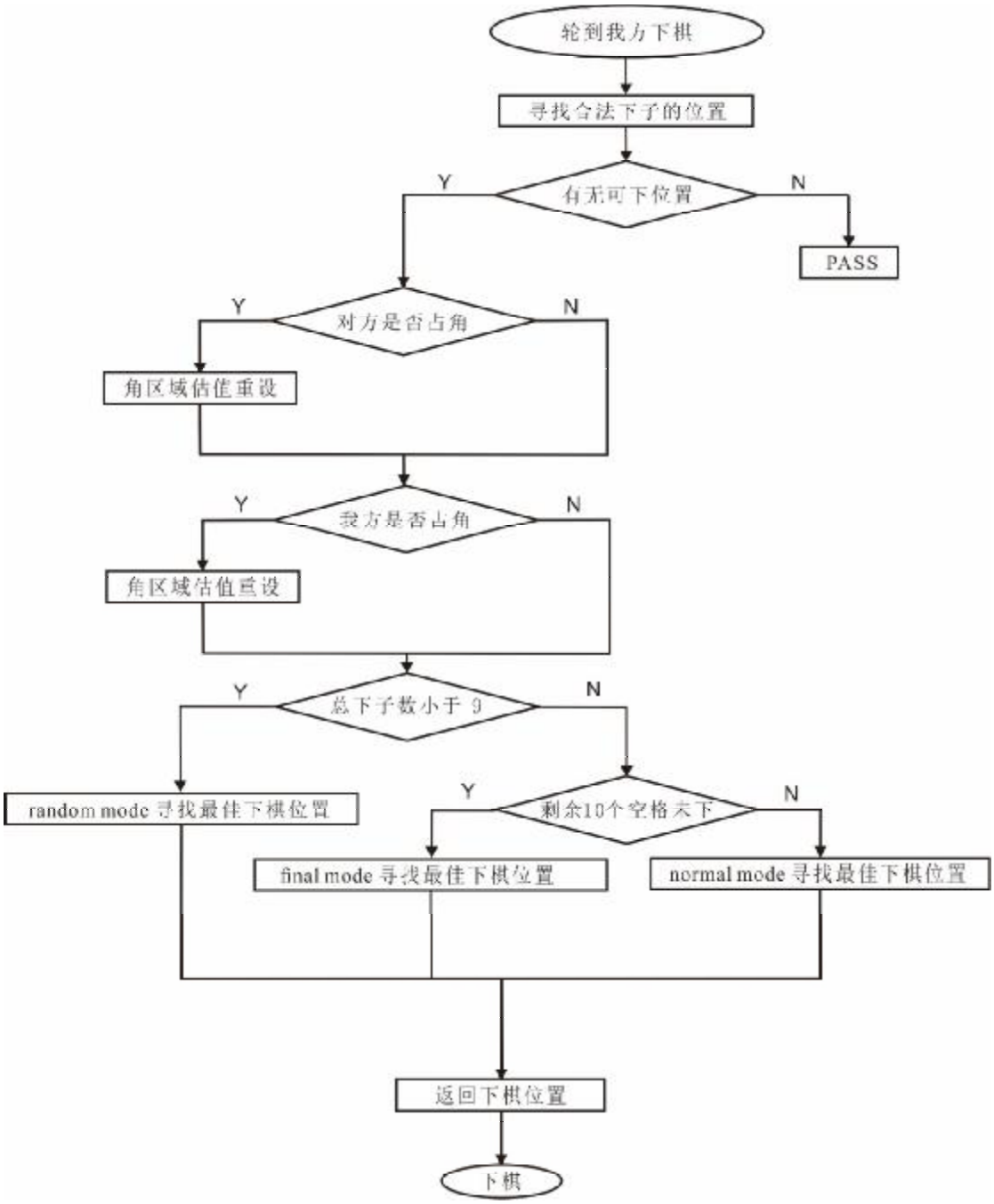
Table 1 全局变量表

全局变量名	变量值	变量解释
INFINITE	100000	估价阈值，可理解为无穷大
STABLE_EVAL	63	稳定子估价，如果一个棋子变成稳定子，它的估价将增加该值
WEDGE_EVAL	[1,2,5,9]	楔入子估价，估价随着楔入维度的增加而增加，最高同时在四个方向上楔入
DOF_EVAL	-3	自由度估价，把自由度定义为留给对方的合法棋位数，自由度越高对己方越不利
NORMAL_TIMEOUT	80	正常模式超时警报
FINAL_TIMEOUT	80	终极模式超时警报
NORMAL_DEPTH	4	正常模式搜索深度（根据电脑硬件情况，可调）
FINAL_DEPTH	10	终极模式的开始搜索深度，用于游戏即将结束时，直接搜索至结局（根据电脑硬件情况，可调）

	A	B	C	D	E	F	G	H
1	1123	-67	109	37	37	109	-67	1123
2	-67	-697	-1	-1	-1	-1	-697	-67
3	109	-1	2	1	1	2	-1	109
4	37	-1	1	0	0	1	-1	37
5	37	-1	1	0	0	1	-1	37
6	109	-1	2	1	1	2	-1	109
7	-67	-697	-1	-1	-1	-1	-697	-67
8	1123	-67	109	37	37	109	-67	1123

Figure 1 棋盘初始估值图

9.1.2 算法流程图


轮到我方下棋
寻找合法下子的位置
有无可下位置
Y
N
PASS
对方是否占角
Y
N
角区域估值重设
我方是否占角
Y
N
角区域估值置设
总下子数小于 9
Y
N
random mode 寻找最佳下棋位置
剩余10个空格之下
Y
N
final mode 寻找最佳下棋位置
normal mode 寻找最佳下棋位置
返回下棋位置
下棋

9.1.3 算法运行时间复杂度分析

代码 1：算法主函数（play）

```
1.  def play(cb, ms):
2.      lp= cb.getLegalPos()
3.      if lp== []:
4.          return 'PASS'
5.      else:
6.          global CB_EVAL
7.          time = cb.getTimeCost()
8.          me = cb.getTurn()
9.          myTime = time[0] if me == B else time[1]
10.         if myTime > 150:
11.             myTurn = lp[0]
12.         else:
13.             for log in ms['log']:
14.                 if log.turnPos in ['A1','A8','H1','H8']:
15.                     cbEvalReset(Pos2rc(log.turnPos), cb, cb.getTurn(), CB_EVAL)
16.
17.             myTurn = findBestPos(cb, ms)
18.
19.         return myTurn
```

我们从算法主函数（代码 1）的运行来分析走一步棋的时间复杂度。由于棋盘的大小是固定的，所以第 1 行获取合法落子位置的函数 getLegalPos 的执行时间可以近似为一个常数，而后面 7、8 行的操作以及 13 行的循环消耗的时间也都是常数，所以算法的主要运行时间消耗来自于 findBestPos 函数的递归调用的过程。

findBestPos 函数中有三种下棋策略，如前文所介绍的，分别为随机策略、普通策略以及终极策略。随机策略中，只是寻找了棋盘中间部分的合法棋位随机落子，其时间开销多来自于取随机位置的过程，并且随机范围小，其复杂度是常数；普通策略和终极策略中，前者是根据估值的大小来决定落子位置，后者是根据落子后的己方与对方棋子数来决定落子位置，虽然两者的策略不太一样，但算法的实现步骤是相似的：深度优先搜索。同时，通过实验以及竞赛我们发现影响时间复杂度的最大因素是递归的深度。对于普通策略而言，我们考虑他最坏的情况。假设每一步有 k 个合法落子位置，递归深度是 n_1（因为普通策略和终极策略对应的启动条件不同，递归的深度也不同，所以用 n_1、n_2 区分两者），那么进行 n_1 次递归之后，DFS

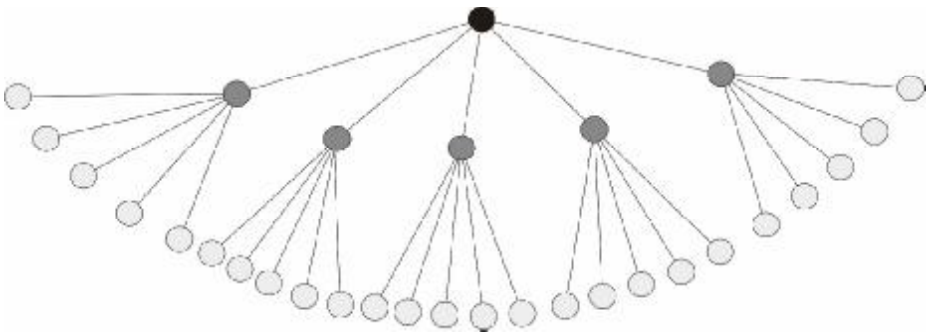


Figure 2　普通策略深度优先搜索树（设定 k=5，n_1=3 时），最坏情况

搜索树中的节点就会有()个，也就是 O（）的时间复杂度，见图 2。但事实上，每一步的合法落子位置数量并不是固定的，而且本算法使用了 alpha-beta 剪枝算法来进行时间复杂度的优化，大约能将 DFS 搜索树中的节点数平均减少约四分之一，在具体操作的过程中可以降低所消耗的时间，但是对于时间复杂度而言，

其结果还是 O（）。

对于终极策略而言，其递归是执行到游戏结束，即棋盘被下满为止，设终极策略是在还有 n_2 个空格时执行（即递归深度是 n_2），考虑最坏的情况，那么进行递归直到所有被空格填满的过程中，每填一个格子都要把所有的格子都搜索一遍，并且搜索到最终结果，所以每填一个格子，在剩余格子的数量为 n_2 时，其复杂度为 O（!），见图 3。事实上，终极模式下也使用了 alpha-beta 剪枝算法来进行优化，所以同普通模式一样，其运行速度会有所提高，但是复杂度仍然是 O（!）不会变化。

综上所述，本算法的时间复杂度为 O（+!），其中 n1 为普通模式递归深度，对应于全局变量中的

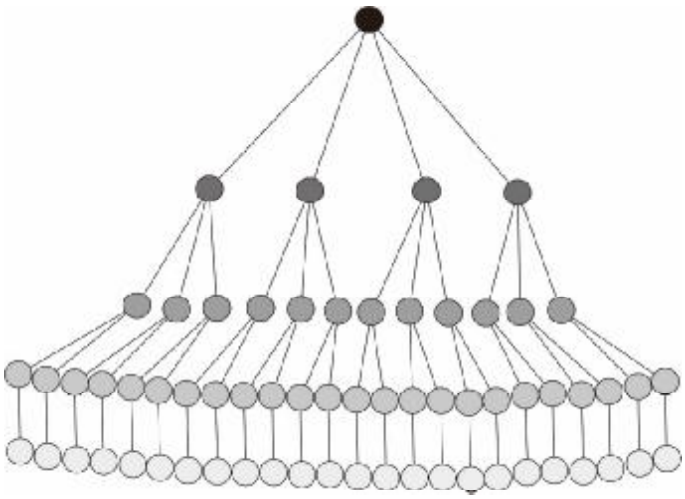


Figure 3　终极模式一个格子的 DFS 树（设定 n_2=5），最坏情况

NORMAL_DEPTH；n2 为终极模式递归深度，对应于全局变量中的 FINAL_DEPTH；k 是一个常数。

9.2 程序代码说明

9.2.1 数据结构说明

算法中所使用的主要数据结构是一个基于 ADT Graph 改进的图，其与 ADT Graph 所不同的地方在于，图中的每个顶点都有联系，但没有实际的边将这些顶点连接起来，这种联系体现在各个顶点在棋盘上的位置，也就是棋盘中的每个格子，依据每个格子在黑白棋竞赛中的重要性，用二维列表的方式，将原本赋予边的权重赋予了顶点（图 1）。总而言之，本算法使用的以二维列表实现的图，虽然与 ADT Graph 的参数设置不同，但是其基本思想是相同的。

另外，在寻找最佳下棋位置是所使用的 DFS 深度优先搜索，其数据结构基础是树，虽然程序中并未明确定义一棵树，但是算法编写过程中的逻辑结构和树的深度优先搜索是一样的。

9.2.2 函数说明

```
1.  def findBestPos(cb, ms):
2.      global COUNT_NODES
3.      global NORMAL_DEPTH
4.      global FINAL_DEPTH
5.      global TIME
6.      lp = cb.getLegalPos()
7.      score = cb.getScore()
8.      totalScore = cb.getTotalScore()
9.      cb_eval = copy.deepcopy(CB_EVAL)
10.
11.     if totalScore < 13:
12.         innerPos = []
13.         for p in lp:
14.             pos = Pos2rc(p)
15.             if pos[0]>=2 and pos[0]<=5 and pos[1]>=2 and pos[1]<=5:
16.                 innerPos.append(pos)
17.         if len(innerPos) > 0:
18.             position = innerPos[random.randrange(len(innerPos))]
19.             return rc2Pos(position[0],position[1])
20.         else:
21.             return miniMax(cb, cb.getTurn(), NORMAL_DEPTH, cb_eval)['pos']
22.
23.     elif 64 - totalScore <= FINAL_DEPTH:
24.         depth = 64 - totalScore
25.         TIME = time.clock()
26.         cb1 = ChessBoard(cb)
27.         step = miniMax2End(cb1,cb1.getTurn(),depth, cb_eval)
28.         return step['pos']
29.
30.     else:
31.         depth = NORMAL_DEPTH
32.         if 64 - totalScore <= FINAL_DEPTH + 2:
33.             depth += 3
34.         elif 64 - totalScore <= FINAL_DEPTH + 4:
35.             depth += 2
```

9.2.2.1 findBestPos 函数

代码 2：findBestPos

```
36.         elif 64 - totalScore <= FINAL_DEPTH + 24:
37.             depth += 1
38.         TIME = time.clock()
39.         cb1 = ChessBoard(cb)
40.         step = miniMax(cb1, cb1.getTurn(), depth, cb_eval)
41.         return step['pos']
```

函数

findBestPos 函数是当轮到我方下棋时，寻找最合适下棋位置的辅助函数，接口位置见代码 1，接口处传入两个参数，分别是 cb 棋盘态势和 ms 下棋历史记录，然后分为三种策略：随即模式、普通模式、终极模式对最合适的下棋位置进行搜索。以下为函数的运行解析。

第 6 行至第 9 行是为函数运行所进行的参数准备，分别取得了棋盘上的合法下棋位置、己方与对方的棋子数、总棋子数，并对棋盘每个格子的初始权值进行了拷贝（因为之后的函数操作会改变这些值以求得最佳位置）。

第 11 行至第 21 行为随机策略，即在总棋子数小于 13 时，在棋盘中间区域（中间四行四列）随机落子，目标是节省时间与最大化缩小己方棋子的分散面积，遏制对手。13 至 16 行的循环是获取这些中间位置，而后 17 行判断是否有这样的中间位置，如果有则随即选取一个下子，如果没有就调用正常策略下的 miniMax 函数（注：miniMax 和 miniMax2End 都是辅助函数，返回的是一个既有位置也有权值的字典类型，解析见下文）来寻找合法棋位。

第 23 行至第 28 行为终极策略，当棋盘剩余空位数小于或等于终极搜索深度时，便开始终极搜索深度，直接搜索至游戏结束。24 行设置当前的搜索深度，26 行得到棋盘的副本，因为 27 在运行 miniMax2End 函数时，递归会改变棋盘。

第 30 行至第 41 行就是在上述两种情况之外，执行普通模式，按照既定的搜索深度调用 miniMax 辅助函数进行搜索，但在关键步骤处可以增加搜索深度，如 32 行至 37 行所示，当棋局距离最终模式还差 24 步棋时，搜索深度加 1，差 4 步棋时，搜索深度加 2，差 2 步棋时，加 3，这些增加的递归深度可以理解为是普通模式向终极模式过渡的过程。

9.2.2.2 miniMax 函数

```
1.  def miniMax(cb, player, depth, cb_eval, alpha=-INFINITE, beta=INFINITE, PASS = 0):
2.      isMe = (player == cb.getTurn())
3.      result = -INFINITE-1 if isMe else INFINITE+1
4.      position = None
5.      lp = cb.getLegalPos()
6.
7.      if isMe:
8.          i = 0
9.          for j in range(i,len(lp)):
```

代码 3：miniMax 函数

```
10                  if lp[i] in ['A1' 'A8' 'H1' 'H8']:
54                      cb2.makeTurn(pos2)
98                  if isMe:
99                      if stepEval > result:
100                         result = stepEval
101                         position = pos
102                     alpha = max(alpha, result)
103                     if alpha >= beta:
104                         break
105                 else:
106                     if stepEval < result:
107                         result = stepEval
108                         position = pos
109                     beta = min(beta, result)
110                     if alpha >= beta:
111                         break
112
113     return {'pos':position, 'eval':result}
72                          position = pos
73                  return {'pos':position, 'eval':MAX}
74              else:
75                  MIN = INFINITE+1
76                  for pos in lp:
77                      cb_eval1 = copy.deepcopy(cb_eval)
78                      nextEval = getStepEval(cb, pos, player, cb_eval1)
79                      if nextEval < MIN:
80                          MIN = nextEval
81                          position = pos
82                  return {'pos':position, 'eval':MIN}
83
84      else:
85          global TIME
86          global NORMAL_TIMEOUT
87          if time.clock() - TIME > NORMAL_TIMEOUT:
88              depth = 2
89
90          for pos in lp:
91              cb1 = ChessBoard(cb)
92              cb_eval1 = copy.deepcopy(cb_eval)
93              thisEval = getStepEval(cb1, pos, player, cb_eval1)
94              cb1.makeTurn(pos)
95              nextStep = miniMax(cb1, player, depth-1, cb_eval1, alpha, beta, PASS)
96              stepEval = thisEval + nextStep['eval']
97
```

miniMax 搜索函数，目标是得到给定搜索深度内的最佳估值，在己方立场来看，对于己方要得到最大估值，而对于对方要使其得到最小估值，所以叫做 mini-max 搜索。在这个函数的最后加入了 alpha-beta 剪枝算法对搜索的效率进行优化。miniMax 函数接口位置见代码 2，接口处传入参数棋盘格局 cb、所考察的玩家 player、搜索深度 depth、开始搜索前的棋盘估值 cb_eval、剪枝参数 alpha 和 beta（缺省为 -INFINITE 和 INFINITE，具体值见表 1）、判断双方是否都无棋可下的参数 PASS。

第 2 到 4 行首先对函数所需的参数进行赋值，分别是判断轮到己方还是对方下的参数 isMe、记录搜索过程中每个节点所返回的估值 result（赋为 INFINITE+1 和-INFINITE-1 暗合上文己方要使己方估值最大而

是对方最小）、记录搜索过程中每个节点所返回的最佳落子位置的 position 以及本步的所有合法落子点的列表 lp。

第 7 到 28 行是对列表 lp 按照其中可下位置的重要性，金角银边草肚皮，进行排序，将角位和边位放在最前面先搜索，目的是为了如果即将超时而搜索被迫结束时，能保证最重要的位置已经被搜索了，同时，这些优势位置先搜索可以大大提高剪枝效率。

第 30 到 38 行判断的是搜索过程中无法落子的情况。一旦出现无法落子，便交换对手，并记录已经连续 PASS 一次，随后进入下一次递归。如果连续 PASS 两次，则结束此搜索分支，但如果下一次递归中没有 PASS，则将 PASS 参数清零。

第 40 到 62 行判断的是 KO 的情况，要注意 KO 的情况只对于第一层递归有效，因为在更深层的递归中，对方可能并不是按我们算的位置下子。并且根据经验，KO 多发生在黑白棋的前期，后期占边角之后不可能出现 KO 的情况，因而只在总棋子数小于 30 时判断 KO 的情况。在满足上述条件的情况下，如果出现能 KO 对手的情况，则直接返回这个 KO 位置；如果出现的是被对手 KO 的位置，那么直接在合法落子位置列表中删除这个位置。

以上是对可能遭遇的特殊情况的应对策略，第 64 到 82 行是递归的基本结束条件，当进入最后一层递归时，如果轮到我方下子，便寻找下一步合法棋位中估值最大的位置；如果轮到对方下子，就寻找下一步合法棋位中估值最小的位置，最终返回一个字典类型，包含最佳落子位置和其估值。估值的过程使用了一个辅助函数，为 getStepEval 函数，即计算落子这一步的估值，详细解析见下文。

第 84 到 111 行是递归的过程，每进入一层递归深度，便会得到当前所有可下位置的估值（同样是用 getStepEval 函数），然后按照深度优先的算法，将每个格子递归到底，把每次递归所得的单个估值相加，求得其总估值。在递归的过程中，会使用 alpha-beta 剪枝算法来优化递归的效率，降低递归的消耗时间。alpha 代表的是我方的参数，beta 代表的是对方的参数，初始的时候 alpha 设定为-INFINITE，beta 设定为了 INFINITE，在递归过程中为了找到最佳的落子位置，我们便要找到 alpha 最大的值，任何比 alpha 小的是值便是无效的；同样我们要找到 beta 的最小值，任何比最小值大的值都是无效的，所以我们便用 alpha 和 beta 的值来框定递归时所获得权值的范围，在我方下棋时获得 alpha 的最大值，而在对方下棋是获得 beta 的最小值，至于其剪枝的原理，用图 4 来说明。

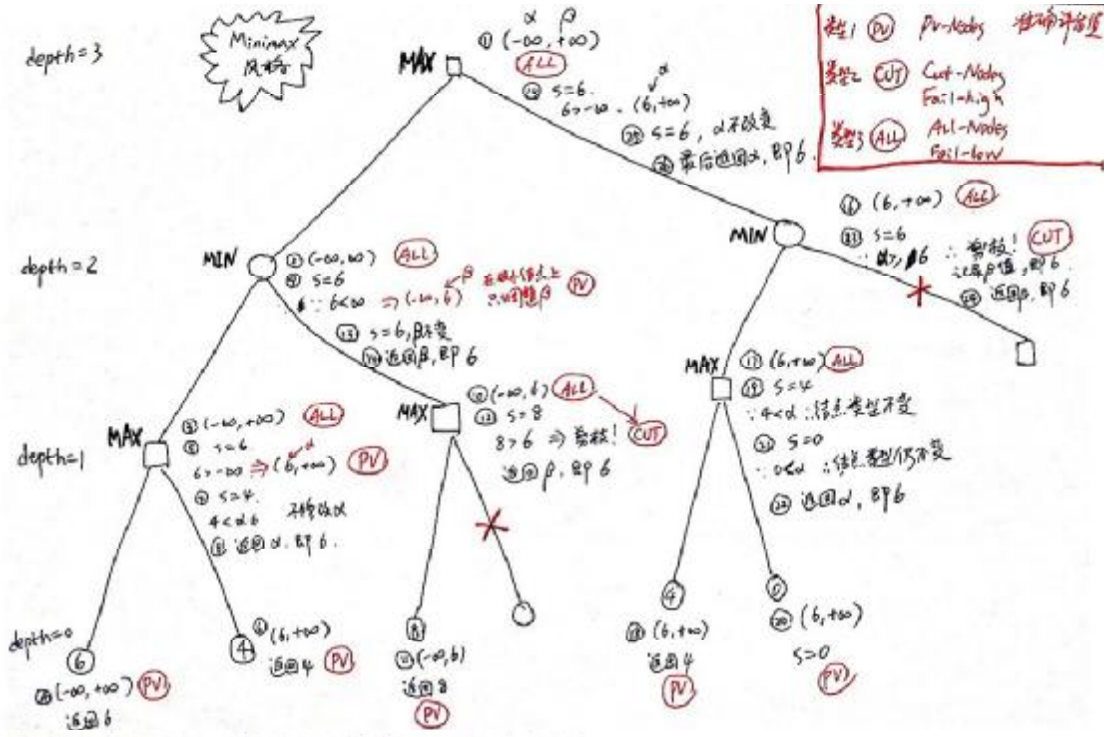


Figure 4　alpha-beta 剪枝算法示意图，引自：http://www.cnblogs.com/speeding/archive/2012/09/20/2694704.html

当 depth=1 时，是我方下棋，因而要取的是 depth=0 的最大值给 alpha，depth=0 的左起第一个节点向上返回 result=6，此时赋予 alpha=6，beta 仍为 INFINITE；depth=0 左起第 2 个结点，向上返回 result=4，因为 4 小于 6，取两者最大值，因而 alpha 等于 6 不变。对于 depth=2，是对方下棋，因而需要获得最小值给 beta，depth=1 的左边节点向上返回 result=6，此时 beta 被赋值为 6，alpha 在 depth=2 的层面上仍为-INFINITE，这两个参数被传入 depth=1 的左起第二个节点，而在 depth=1 需要获得 depth=0 的最大值，此时 depth=0 的左起第三个节点向上返回 result=8，赋予 alpha，也就是 alpha=8，但此时，如果 depth=0 的左起第四个节点的权值小于 8，他不会改变 alpha 的值，如果大于 8，则 alpha 的值被改变，不放将此时 alpha 的值设为 a，a>8，而此时 beta 的值为 6，depth=1 的左起第二个节点应该要向 depth=2 的左起第一个节点返回一个最小值，而 a>6，所以返回的是 6。因此可见，无论 depth=0 的左起第四个节点的值为何，向 depth=2 返回的都是 6，所以是否判断 depth=0 的左起第四个节点显得无关紧要，固将其剪去。上述过程也就是代码中所判断的当 alpha≥beta 是时结束搜索的操作。

9.2.2.3 miniMax2End 函数

```
1.  def miniMax2End(cb, player, depth, cb_eval, alpha=-INFINITE, beta=INFINITE, PASS=0):
2.      isMe = (player == cb.getTurn())
3.      result = -2*INFINITE if isMe else 2*INFINITE
4.      position = None
5.      lp = cb.getLegalPos()
6.      if isMe:
7.          i = 0
8.          for j in range(i,len(lp)):
9.              if lp[j] in ['A1','A8','H1','H8']:
10.                 lp[i], lp[j] = lp[j], lp[i]
11.                 i += 1
12.         for j in range(i,len(lp)):
13.             if lp[j] in ['A3','A6','H3','H6','C1','F1','C8','F8']:
14.                 lp[i], lp[j] = lp[j], lp[i]
15.                 i += 1
16.         for j in range(i,len(lp)):
17.             if lp[j] in ['B2','B7','G2','G7']:
18.                 lp[i], lp[j] = lp[j], lp[i]
19.                 i += 1
20.         for j in range(i,len(lp)):
45.         if depth == 1:
46.             cb_eval1 = copy.deepcopy(cb_eval)
47.             return {'pos':lp[0], 'eval':getFinalEval(cb, player, cb_eval1, lp[0])}
48.
49.         else:
50.             global TIME             global FINAL_TIMEOUT
51.         if time.clock() - TIME > FINAL_TIMEOUT:
52.             depth = 2
53.         for pos in lp:
54.             cb1 = ChessBoard(cb)
55.             cb_eval1 = copy.deepcopy(cb_eval)
56.             cb1.makeTurn(pos)
57.             nextStep = miniMax2End(cb1, player, depth-1, cb_eval1, alpha, beta,
    PASS)
58.             stepEval = nextStep['eval']
59.
60.             if isMe:
61.                 if stepEval > result:
62.                     result = stepEval
63.                     position = pos
64.                 alpha = max(alpha, result)
65.                 if alpha >= beta:
66.                     break
67.             else:
68.                 if stepEval < result:
69.                     result = stepEval
70.                     position = pos
71.                 beta = min(beta, result)
72.                 if alpha >= beta:
73.                     break
74.     return {'pos':position, 'eval':result}
```

代码 4：miniMax2End 函数

miniMax2End 是终极策略的搜索函数，其接口见代码 1，在接口处传入参数棋盘格局 cb、所考察的玩家 player、搜索深度 depth、开始搜索前的棋盘估值 cb_eval、剪枝参数 alpha 和 beta（缺省为

-INFINITE 和 INFINITE，具体值见表 1）、判断双方是否都无棋可下的参数 PASS。

miniMax2End 函数与 miniMax 函数类似，但是其直接搜索至游戏结束，其与 miniMax 的区别在于 miniMax 是基于估值最优的搜索，但保持估值最优并不一定会赢，决定输赢的是最终棋子的数量，所以 miniMax2End 是基于棋子数量最优的搜索，在代码上，只有递归的基本结束条件不同。在第 39 行处，当深度为 1 时，棋盘只剩一个可落子点，递归结束，此时调用的估价函数和 miniMax 中不同，是 getFinalEval 函数，这是基于棋子数的估价，具体的解析见下文。

9.2.2.4 getStepEval 函数

```
1.  def getStepEval(cb, position, player, cb_eval):
2.      global COUNT_NODES
```

```
86.                                 done2 = True
87.                                 done = True
88.                                 dim += 1
89.                             else:
90.                                 if board[r][c] == player:
91.                                     r -= os[0]
92.                                     c -= os[1]
93.                                 elif board[r][c] not in [B, W]:
94.                                     break
95.                                 else:
96.                                     if done1:
97.                                         done2 = True
98.                                     else:
99.                                         break
100.                        if done1 and done2:
101.                            dim += 1
102.            return sum(WEDGE_EVAL[:dim])
103.
```

```
130.        for i in Line:
131.            for j in Line:
132.                if nextBoard[i][j] == '+':
133.                    dof += 1
134.        Eval += dof * DOF_EVAL
135.
136.        Eval += getWedgeEval(nextBoard, position, player)
137.
138.        stablePosSet = set()
139.        board1 = copy.deepcopy(currentBoard)
140.        getStablePos(board1, player, changedPos, stablePosSet)
141.        Eval += len(stablePosSet) * STABLE_EVAL
142.        Eval = Eval if isMe else -Eval
143.        return Eval
```

```
119.                            else:
120.                                break
121.
122.        if position in [(0,0),(0,7),(7,0),(7,7)]:
123.            cbEvalReset(position, cb1, player, cb_eval)
124.
125.        for pos in changedPos:
126.            Eval += cb_eval[pos[0]][pos[1]] * 2
127.        Eval += cb_eval[position[0]][position[1]]
128.
```

代码 5：getStepEval 函数

getStepEval 为单步棋估值函数，其功能是得到在某位置落子后，本步棋的估值。其接口见代码 3、4，接口处传入棋盘 cb、落子位置 position、所

考察的玩家 player、以及棋盘的初始估值 cb_eval。getStepEval 有四个子函数分别为求稳定子函数 getStablePos、KO 估值函数 getKOEval、楔入子估值函数 getWedgeEval。

第 2 行至第 10 行为对函数所需的变量进行设置，COUNT_NODES 记录所搜索节点的数量，cb1 用于储存下一状态的棋盘，currentBoard 保存当前棋盘，nextBoard 保存该步落子之后的棋盘，changedPos 为保存落子之后所改变的所有位置的列表，isMe 为对当前玩家是否为待考察玩家的判断，Eval 是记录的估值。

第 12 行至第 44 行为稳定子估价函数，其传入棋盘、玩家、可下位置列表、稳定子集合四个参数，由此寻找棋盘可下位置列表 posList 中的所有稳定位置。稳定子被定义为在棋盘上绝对不会被反转的棋子，这个函数的其精华部分是 getStablePosHelper 的辅助函数，其从被玩家占领的角位开始搜索，先列后行，因为角位被占领所以不考虑斜线方向；如果列中出现了对方的棋子，其相应的列就不可能是稳定子，如果行中出现了对方的棋子，那么相应的行就不可能是稳定子，最终将所有的稳定子加入稳定子集合中返回。最终在 138 行至 142 行处，对稳定子列表中的位置根据其所能得到的稳定子数目，乘以稳定子估价，赋给当前下棋位置的估值返回。

第 45 至 56 行为 KO 估值函数，传入参数棋盘和玩家。如果能 KO 对手，则该步估值达到无穷大，程序会自动争取下此步棋；如果被对手 KO，则估值达到无穷小，程序会自动避免此步棋。

第 58 行至第 102 行为楔入子估值函数，传入三个参数：棋盘 board、落子位置 pos、玩家 player。把楔入子定义为在下完一个棋子之后，在这个棋子的横竖或者斜方向上，本方棋子可延伸直至遇到对方棋子或棋盘边缘，即被对方棋子包裹或延伸至棋盘边缘，楔入子在楔入方向上无法被吃掉。楔入子有横、竖、左斜、右斜四个维度，最终估值对照最前面的楔入子估值列表。dim 变量记录棋子的楔入维度，即楔入的方向数。第 60 行的循环分别按照上述的四个方向逐一进行搜索，首先，第 65 行至第 78 行，先向一个方向中的一个延伸方向进行搜索（直线有两个延伸方向），如果搜索到达棋盘边缘，则断定该方向为楔入方向（第 67 行）；如果延伸方向遇到空位，则可能是楔入方向（第 75 行）；如果延伸方向上遇到对方棋子，那么有可能是楔入方向，还要再判断另一边是否延伸至对方棋子，此时还需判断相反的延伸方向（第 80 行至第 102 行），如果达到棋盘边缘，则断定该方向为楔入方向（第 85 行）；如果未到边缘而遇到空位，则不可能是楔入方向（第 93 行）；如果延伸方向上遇到对方棋子，而在之前的相反方向上也遇到了对方棋子，则是楔入方向（第 96 行），否则不是楔入方向；如果两个方向上均延伸至对方的棋子，该方向上是楔入方向（第 100 行）。

之后的代码便是具体的估值过程，首先判断有无占角的情况，如果有占角则要重设棋盘初始估值（cbEvalReset 函数，见后），之后按照棋盘的初始估值对落子位置进行棋盘位置估值，随后便按照自由度估值、楔入子估值和稳定子估值的顺序对落子位置进行估值。值得一提的是第 129 行至 134 行的自由度估值，其作用是限制对方的落子点。

9.2.2.5 getFinalEval 函数

```
1.  def getFinalEval(cb, player, cb_eval, pos = None):
2.      Eval = None
3.      score = cb.getScore()
4.      if score[0] > score[1]:
5.          Eval = INFINITE + (score[0] - score[1]) * 50
6.      elif score[0] < score[1]:
7.          Eval = -INFINITE + (score[0] - score[1]) * 50
8.      else:
9.          Eval = (score[0] - score[1]) * 50
10.     if player == W:
11.         Eval = -Eval
12.     if pos == None:
13.         return Eval
14.     return Eval + getStepEval(cb, pos, player, cb_eval)
```

代码 6：getFinalEval 函数

getFinalEval 函数是终极估值函数，是接口是在终极策略搜索到游戏结束时，

所得到的最终估值，传入参数棋盘 cb、落子方 player、棋盘初始估值 cb_eval、落子位子 pos（缺省值为 None）。在搜索中，判断依据不仅是估值，还有对棋盘上己方和对方落子数的判断，如果落子数多于对手获胜，则该步棋赋予极大估值，否则赋予极小估值，同时，为了避免多个节点估值相等，在极大和极小的基础上又加上了单步棋估值。

```
1.  def cbEvalReset(cn, cb, player, cb_eval):
2.      offset = {(0,0):(1,1),(0,7):(1,-1),(7,0):(-1,1),(7,7):(-1,-1)}
3.      os = offset[cn]
4.      isMe = (player == cb.board[cn[0]][cn[1]])
5.      if isMe:
6.          cb_eval[cn[0]+os[0]][cn[1]] = 111
7.          cb_eval[cn[0]][cn[1]+os[1]] = 111
8.          cb_eval[cn[0]+os[0]][cn[1]+os[1]] = 7
9.      else:
10.         cb_eval[cn[0]+os[0]][cn[1]] = 29
11.         cb_eval[cn[0]][cn[1]+os[1]] = 29
12.         cb_eval[cn[0]+os[0]][cn[1]+os[1]] = -37
```

9.2.2.6 cb

EvalReset **函数**

代码 7：cbEvalReset 函数

cbEvalReset 函数为角区域估值重设函数，其接口见代码 1，代码 5，接口处传入被占的角位 cn，棋盘 cb，玩家 player 和棋盘的初始估值 cb_eval。当有一个角被占领后，它周围的格局就会发生改变，cbEvalReset 函数会重设与该被占角位相邻的三个关键位置的初始估值。

9.2.3 程序限制

当递归深度过大时，超时补丁可能会无效，原因在于超时补丁判断的是进入递归前的时间，如果进入递归前时间未到 150 秒，而递归的过程中耗时过久，便有可能出现超时。除此之外，如果每一步对手给出的合法棋位都比较多，搜索的负担会加重，也可能导致超时。

9.3 实验结果

9.3.1 实验数据

实验环境说明：

【蓝色部分】
- 硬件配置：CPU：Intel(R)_Core(TM)_i5-3230M_CPU_@_2.60GHz
  内存：5.86G　DDR3 1600MHz
- 操作系统：windows 8.1 专业版 x64
- Python 版本：2.7.9

【绿色部分】
- 硬件配置：CPU：英特尔 Celeron（赛扬） 1005M @1.90GHz 双核
  内存：4G （金士顿 DDR3 1600MHz）
- 操作系统：windows 8.1 专业版 x32
- Python 版本：2.7.9

【红色部分】
- 硬件配置：CPU：Intel(R)_Core(TM)_i5-3230M_CPU_@_2.60GHz
  内存：12.0G　DDR3 1600MHz
- 操作系统：windows 8.1 专业版 x64
- Python 版本：2.7.9

1000 次对弈结果数据汇总

黑 \ 白	idiot 算法	gambler 算法	本算法
idiot 算法	胜负 0：1000 （0 平局） 比分 27000：37000 时间 17.45：23.70	胜负 558：414 （28 平局） 比分 33444：30377 时间 22.72：17.725	胜负 141：849 （10 平局） 比分 20415：43449 时间 11.38：3142.92
gambler 算法	胜负 372：601 （27 平局） 比分 28699：35221 时间 19.55：23.47	胜负 441：516 （43 平局） 比分 31225：32617 时间 21.79：21.17	胜负 95：886 （19 平局） 比分 19187：44291 时间 21.29：6638.72
本算法	胜负 855：143 （2 平局） 比分 44258：19510 时间 3779.84：13.07	胜负 897：87 （16 平局） 比分 43940：19484 时间 5696.18：18.69	胜负 498：472 （30 平局） 比分 32519：31404 时间 4297.62:4071.07

【注】由于测试的时间代价过大，所以本测试中把本代码（KILO）的递归深度降低至 2（普通策略）和 4（终极策略）

9.3.2 结果分析

与 idiot 和 gambler 对弈的是最原始的 KILO 算法，也就是没有针对比赛进行优化的 KILO 算法，但两者

的差别只在最终模式递归的初始深度。在这种情况下，综合执白和执黑的数据，KILO 算法对 idiot 的胜率为 85.2%，对 gambler 的胜率为 89.2%，两者相差并不太多。由此可见，本算法对棋盘上每个格子的估值，以及深度优先搜索的策略是有效的。至于那百分之十几的败率一定程度上是因为 KILO 算法是针对有落子策略的函数进行设计的，而 idiot 算法和 gambler 算法的并没有精心算计的落子策略，所以他们的落子位置在一定程度上和 KILO 算法所预测的落子位置不同，因而所得到的估值会与和 idiot、gambler 对战时的实际情况有所出入，造成了最后 10%的败率，此外 KILO 的递归深度较浅也导致棋力下降不少。

在实验中的运行时间，由于进行实验的机器配置有所不同，但总体上还是在预期之中的，并未出现严重超时的现象，但从比赛的过程中来看，我方 KILO 算法的主要时间开销都来自于递归调用的过程，包括前述的超时补丁所会出现的 BUG 也是由于递归而造成的，虽然代码中设计了 alpha-beta 剪枝算法对递归树进行剪枝，但剪枝的效率并不是很高，因而在可落子点较多的情况下，或者是比赛中与对方“肉搏”调高递归深度时，程序往往会陷入长考，从而增加了时间开销。

9.3.3 经典棋局

在竞赛四分之一决赛中，我方和 DELTA 组前两轮打成 3：1 领先，最后一轮我方通过心理战，调高了递归深度，用超时逼他们超时，最终惊险进入四强。虽然最后屈居第四，但是在算法编写过程中参考了人机对战数据，所以人类要想战胜最终参加半决赛的 KILO 算法还是有一定难度的。

9.4 实习过程总结

9.4.1 分工与合作

朱贺	组长、KILO 算法总设计师与改进者
汪诗舜	小组报告撰写
胡哲	程序调试与实验测试
蓝坤	实验测试
党卓	实验测试
韦春婉	算法人机对战测试

9.4.2 经验与教训

我们能够在两个星期内写出可以一战的黑白棋算法，是我们的得意之处，但是不足之处便在于算法不够完善。如前文所说，我们已经发现了算法超时的主要问题来自于搜索树的过于庞大，并且已经使用了 alpha-beta 剪枝算法进行优化，但结果仍然不尽如人意，尤其是在比赛过程中，面对别的小组的算法，我们迫于超时的压力，不得不调低了递归深度，造成了多局失败。对于这个问题，我们还需要时间来进行调整，但迫于期末考试的压力，我们只能做到现在这个样子了。

9.4.3 建议与设想

建议实习作业可以再提早一段时间布置，比如提前一个月布置，这样会有更多的时间准备；另外基础设施代码最好能够封装为一个 exe 文件，这样可以显得更为简洁。以及竞赛过程可否使用一个更为强大的服务器，并且将赛程安排分散，这样有利于参赛选手调试程序。

9.5 致谢

我们要感谢 KILO 算法的缔造者，我们的组长朱贺，在他的带领下，我们一路披荆斩棘闯进了四强，小组建议将 2 分加分赋予朱贺同学；我们还要感谢陈斌老师以及助教老师，如果没有他们辛苦劳动编写的基础代码，黑白棋竞赛恐怕很难开展下去；然后还有我们充满信心、一起加油的组员们，KILO 因为团结而强大！

9.6 参考文献

[1]黑白棋的规则和技巧：
http://www.soongsky.com/strategy/rule.php
[2]黑白棋的算法设计：
http://www.cppblog.com/sandy/archive/2012/09/20/191377.html
[3] AlphaBeta 剪枝算法：http://www.cnblogs.com/speeding/archive/2012/09/20/2694704.html
[4] 柏爱俊,几种智能算法在黑白棋程序中的应用.中国科学技术大学大学生研究计划结题报告[C].2007 年 10 月.

南区 South　GOLF 组

10 数据结构与算法课程实习作业报告

柳晓萱* 黄天正 段鉴书 马赞彭 刘松吟 庞磊

摘要： 我们的算法分为三个部分，开局，中局，尾局，小组六人分为三个部分分别负责这三块内容。开局采用棋谱录入的方法，中局采用散度行动力结合的算法，尾局采用递归算法。在开局和尾局用到了树的数据结构。代码最后轻松击败 idot，和 gambler 胜率大于 90%。本次实习，我们成功击败小组劲敌 alpha 队晋级 8 强，遭遇夺冠热门 Charlie 队，前两局战平，最后一局改变算法导致惨败，无缘四强。

关键字： 棋谱录入，散度算法，占角，行动力，树，递归算法

10.1算法思想

10.1.1总体思路

总体思路

本小组的总体思路为将整部棋局分为三部分：开局、中局和残局。通过解构棋局的分析方法，模拟真实下棋过程。每部分根据局面形势和侧重点有对应的策略和算法。

1.1.1 开局

1.1.1.1 开局策略

开局主要策略采取预先存入棋谱的策略。棋谱来源为 Wzebra 黑白棋软件的预存棋谱，其中搜索深度统一为 16 步+24 步深度搜索。通过人机对弈，建立树抽象数据结构（ADT）存储最佳下棋点为棋谱。在对抗赛的开局，便根据这个树所记录的点位落子。由于在比赛中需要区分执黑棋与执白棋，在此以我方执黑，Wzebra 执白为例进行具体描述。我方执黑，则记录每一个我方可能落点，和对应的执白方的最佳落点，最终目的是让执白方赢，也就是建立了执白方的树。考虑到执黑方不太可能落子在输子多的地方，将黑棋赢子数小于等于-3 的落点舍弃。以图 1 为例，此处黑棋虽有 9 个可能落点，然而我们只选取 B3、B4、D6、F6 四个落点进行下一步的棋谱存储。

图 1

在我方和 Wzebra 落子之前，软件都会根据数据库棋谱算出每一个可能落点最终可能赢或输的棋子数。因此，以赢棋数最多为记录原则，我们就可以获得执白棋的对执黑棋所下的每一个落点的最佳反应。当执黑棋方落子，执白棋方可能会出现赢子数相等的两个或多个落点。对于这样的局面，我们将诱导执黑子方以散度算法为原则，落入散度改变量最少但输棋子数最多的陷阱落点，散度算法将在下节阐述。

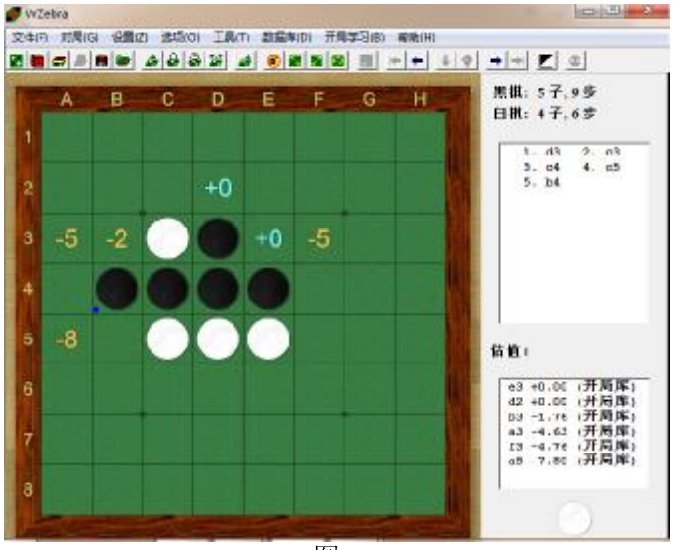


图 2

图 3

图 4

如图 2 所示，白棋有两个赢子数均为 0 的落点，D2、E3。在这种情况下，我们就必须考虑白子落下之后黑子的反应以及对执白方造成的影响。

首先看图 3，即白子落在 E3。黑子下在 D2 或 D6，造成散度增加为 8；而下在 F4，造成散度增加为 7。因此，根据散度算法的原则，黑子最可能落子在 F4。而图中很直观地告诉我们，这个结果与赢子数最多的原则相悖。这样我们就达到了诱导黑子落入散度陷阱的目的。

接下来看图 4，即白子落在 D2。根据散度算法，黑子下在 D6 处造成的散度增量为 8，是所有可能落点中最小的。同时，D6 又是所有落点中赢子数相对最多的位置。因此，D6 这样的点有极大可能被执黑方选取，对执白方没有任何好处，故执白方应该避免这样的局面。

综合上述的分析，白子应该下在 E3，诱导黑子落入陷阱。

1.1.1.2 开局算法

黑白棋散度算法是日本九段国手村上健和四段棋手宫崎纯提出的。研究表明，散度算法是寻找中局阶段的最佳棋步，准确率 70%-80%。

一颗棋子落入棋盘的某个位置，就会产生散度，散度与棋子周边的棋子数相关。除了 PASS 的情况之外，每下一步棋，都会吞下一至多枚棋子，同时也吞下了这些棋子的散度总

和，即落子的散度场。一般来说，下散度总数越少的棋步，越是好棋步。据此，散度理论总结了两点规律：

1.下子应当下在吞下散度总数较小之处。

2.下子应当下在使对手面对较大散度场之处。

事实上散度是针对活动空间而言的。很容量理解，下在吞下散度总数较小之处，可以使对方没有太多的活动空间，下子在使对手面对较大散度场之处，是为己方开辟一个活动空间较大的局面。当然，这两个原则难以避免会产生矛盾。在矛盾时，我们采取后者优先的算法，这也是通常所称的“大散度原则”，主要原因是后者的着眼点在未来的局势，而非只顾眼前。

从实战经验来看，散度算法适用于开局定势至三十余手。

1.1.1.3 开局数据结构

由于开局的策略为根据棋谱选择落点棋步，因此，开局的数据结构以构造棋谱为目的，我们在这里采取树抽象数据结构（ADT）。应用 Python List 来实现树，即采用嵌套列表法。

其中，为了区分执白（后下）和执黑（先下），我们分别建立了两棵树来存入不同的棋谱，两棵树均为多叉树。执白方的树的高度为 8 层，叶节点层级为 7。执黑方的树的高度为 9 层，叶节点的层级为 8。

1.1.2 中局

1.1.2.1 中局策略

对于其他棋类，计算机分析不同落子点的好坏差异主要通过穷举法列举出几步以后的棋盘情况，进而选择最好的走法。但是黑白棋有一个极为突出的特点——形式转变极为迅速，前一刻可能还是满盘皆白，后一刻就有可能满盘尽墨。这就为中局算法的思路带来了一个巨大的问题：难以对当前局面作出评估。难以对当前局面作出评估就意味着无法通过比较几步后的棋盘变化来得出最好的结果。面对这个问题，我们上网查找了一些知名程序的解决方法，发现成功的黑白棋程序都是依靠分析百万甚至千万个对局产生的数据库来判断某一局面的优劣。很显然，我们的算法不可能采用这种方法。

于是，我们组的算法采取了一个退而求其次的思路：通过以某一种特定的规律走棋来达到使局势向对我方有优势的方向转变的目的。

①这个规律必须是相对普适的、易于数字化的、经过实践证明的。经过比较和分析我们选择了散度算法。

散度算法：

散度：一个子周围的空格数量被称为这个子的散度。

散度和:一方落子后棋盘上增加的所有该方棋子的散度之和为这一步的散度和。

散度算法的计算方式为：使己方每一步的散度和最小。

散度算法的意义在于通过尽量少反转对方棋子和尽量反转位于棋盘中央的棋子来尽量减少对方的合法落子点。

那么，为什么要减少对方的合法落子点呢？这就涉及到占角的问题。在黑白棋的开局和中盘，占据角的一方有较大的优势。这是因为角和与角共线相连的子是无法被翻转的，一旦某一方占据了角，就可以从角逐渐向外延伸，占领棋盘中的绝大部分位置。

②因此我们获得了第二个思路：占角

占角：

如果我们可以占角，那么立刻占角。如果我们下某一步棋后对方可以占角，那么我们尽量不下这步棋，并将这点标记为危险点（1）。

不过，占角很明显不会是这么简单的事情，很多情况下，对方虽然不能在下一步马上占角，却能在几步后占角，而且我方无法阻止，很明显，这种情况也是我们要尽量避免的。

情况一：如左图，虽然黑棋走 B7 后白棋无法马上占角，但是只要白棋截断了 H1-A8 对角线，也就是将这条线上任意一子翻转，白棋就可以立刻占领 A8 角。而黑棋明显无法阻止白棋。将这一部列为危险点（3）

情况二：如右图，如果黑棋在 D2 落子，白棋可以下在 C2，下一步白棋可以占领 A1 角，黑棋却无法阻止白棋。将这一部列为为危险点（2）。

危险点：

当我方落子时，尽量不下入危险点，就可以有效避免对方占角

当然了，我们的对手肯定也会避免下入危险点，这时候，散度算法就可以通过减少对方合法落子点来逼迫对方下入危险点，使我方成功占角取得优势。

③不过，散度算法更加注重前瞻性，在测试中逼迫对手下入危险点的效果并不是很好，所以我们引入了行动力算法

行动力算法：

行动力：一方的合法落子点数-危险点数为这一方的行动力。

行动力算法计算每下一步后对方的行动力，选择最能够减少对方行动力的位置落子。

我们通过附加权重的方式来权衡行动力与散度的重要性，在不同的时期选择最合适的计算方式。

综上所述，我们的中盘核心思路为：通过减少对方行动力行动力与降低我方散度来达到逼迫对方下入危险点的目的。

1.1.3 尾局

1.1.3.1 尾局策略

尾局主要采用递归的策略。我们以递归的方式建立一棵树，树中每一层中的节点为一方所有可能的合法位置，每一个节点保存了当前局面的棋谱以及造成当前局面的最后所下的一步棋。为了保证我们的策略赢面最大，我们在给每个节点中加上输赢判断因子，所有的输赢记录都是对我方来说，输赢判断因子的决定方式分为三种情况。

（1） 如果该层轮到我方下，只要子节点中出现一个 win，则该节点被记为 win。如果子节点不存在 win 且存在 tie，则该节点被记录为 tie，如果子节点全部为 lose，则该节点为 lose。

（2） 如果该层轮到对方下，只有子节点全部为 win，则该节点被记为 win。如果子节点中出现 tie（不存在 lose）则该节点被记为 tie，只要子节点中出现 lose，则该节点被记为 lose。

（3） 如果该节点为叶节点，则调用输赢判断函数，如果我方棋子个数大于对方棋子个数，该叶节点被记为 win，如果我方棋子个数等于对方棋子个数，则该节点被记为 tie，如果我方棋子个数小于对方棋子个数，则该节点被记为 lose。

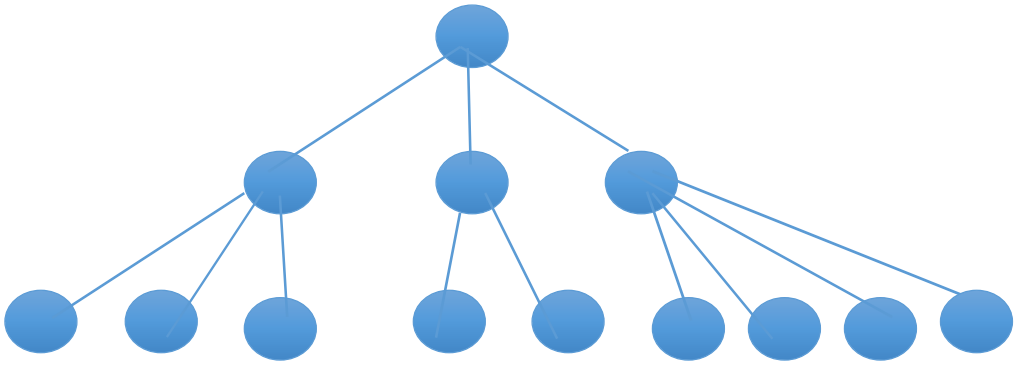
由于树的算法是采用递归来建立的，所以必然是底层的树先被建立，底部的输赢权重因子先被算出，然后一层层向上计算。

当程序运行时，我们可以通过对于场面上所有棋子数目的判断控制何时开始建立尾局树，尾局树是一旦传入进入尾局时的参数（包括此时棋盘以及最后所下的一颗子的颜色位置）就已经完成建立。后面进入尾局是对方每下一步，我们进行搜索，找到对方所下位置对应的节点，然后在该节点的子节点中寻找权重因子 win 的节点，如果没有出现 win，则选择 tie，如果没有 tie 则选择 lose。对于存在多个 win，tie，lose 的情况，我们对每个节点引入胜率。取胜率最高的节点，返回该节点的位置，就是我们所要走的路径。

1.1.3.2 尾局算法

递归算法，每一个合法位置建立一棵子树，每棵子树通过子节点建立父节点，从叶节点开始向上建立完整的树。

1.1.3.3 尾局数据结构

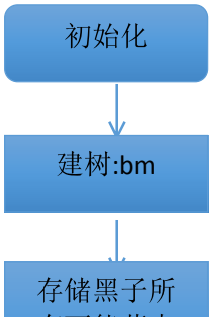


每个节点包涵的数据成员有：

1. 到达该局面的最后一步棋。
2. 该节点的合法位置列表，列表中为以合法位置为根节点的子树
3. 节点输赢判断因子
4. 本轮获胜概率

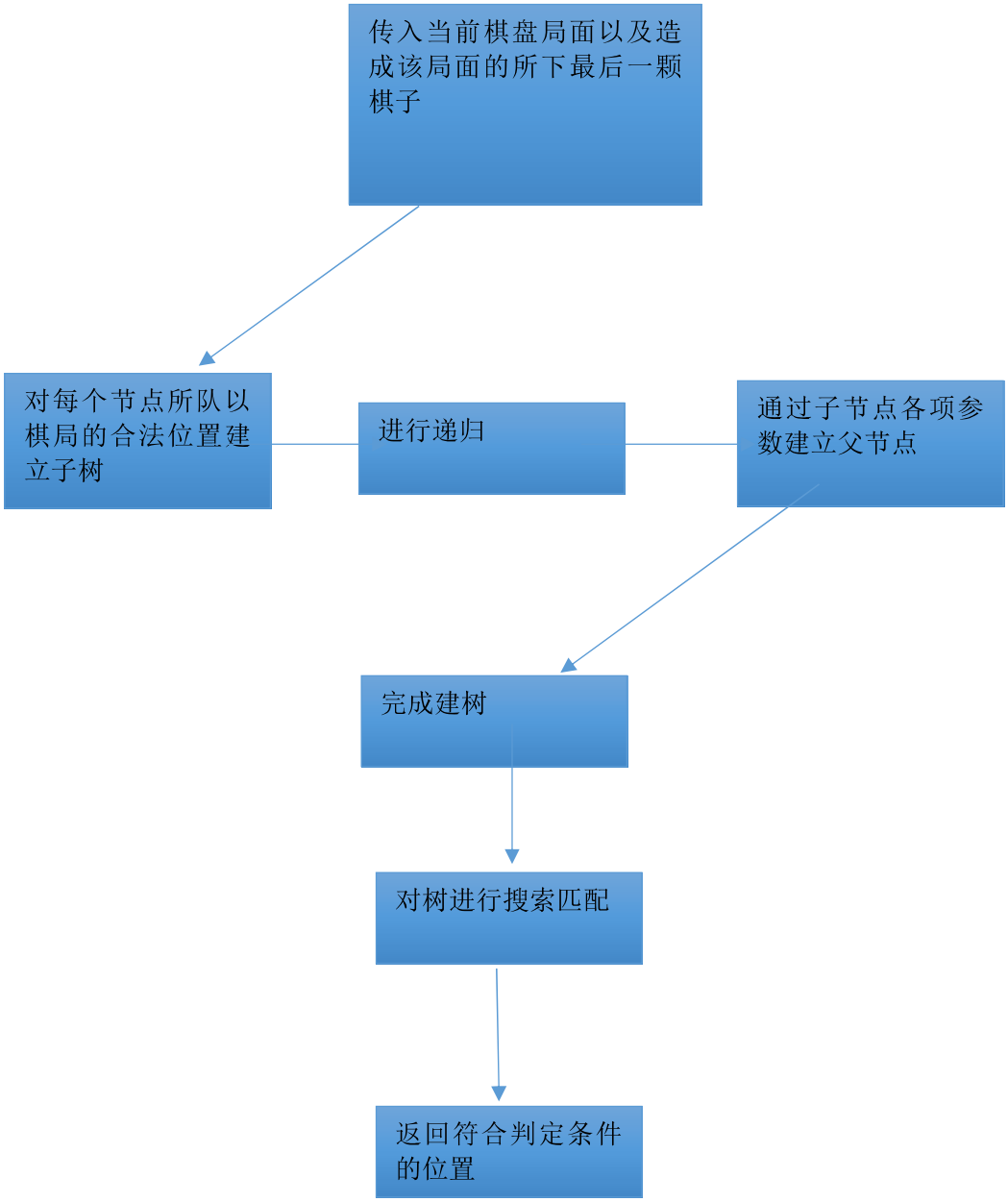
10.1.2算法流程图（以建立白子棋谱为例）

1.开局流程图



2 中局流程图（略）
文字描述：
获得当前棋盘与合法落子点——计算危险点——计算每个安全的合法落子点的散度——计算每个安全的合法落子点落子后对方的行动力——加权后二者相加获得总的优先度，取优先度最小的点落子。

3.尾局流程图



10.1.3算法运行时间复杂度分析

开局时间复杂度分析

由于开局采用根据预先存入的棋谱落子，主要的复杂度在搜索，因此时间复杂度相当小。具体来说，对于执白方，对方在任意可能的四个落点落子后，就确定了唯一的一条有序路径，这里的搜索复杂度为O（4）。而对于执黑方，即我方先行，采取的策略是唯一确定一个落点，因此搜索复杂度仅为O（1）。总的

来说，开局的时间复杂度对于整个算法来说可以忽略。

中局时间复杂度分析

中局时间复杂度约为 $O(n^6)$。
具体计算过程详见下文函数说明，对每个函数均加以评估。其中 Trap(cb, LP)函数为主因。

尾局时间复杂度分析

由于尾局要把双方的合法位置都建成一个节点，所以时间复杂度只要在建树，根据时间我们调整的递归深度到 55，也就是说我们从 55 步开始建树，那么节点个数最多为（64-55+1）！=10！=3628800。而搜索的耗时其实每一步都很少，因为每一层的合法位置大体上是逐层减少的，而且每一层的节点数都必然小于 10，所以搜索的时间是很少的。

10.2程序代码说明

10.2.1数据结构说明

开局，尾局均采用树的数据结构，在基本算法中有详细描述，在此不赘述。
中局未包含数据结构。

10.2.2函数说明

开局是通过建立一棵树的数据结构来存储对手的走子位置和我方棋子的最佳对应方式，即建立一棵棋谱树。这棵树的每个节点都是由一个根值和一个该节点子树列表组成。用到的主要接口有 getRootVal()，获取当前节点的根值；另一个是 getChildItem()，在棋谱树中找到对方棋子所下的坐标在树中的位置，以便找到我方对应策略。Insert()函数是在树中插入棋谱录入过程中的节点。

中局函数及调用关系如下：
中局共八个函数按调用关系分为四级：
MiddleJudge (cb, lp)为一级函数；其调用了 listDiv(cb, LP)、Trap(cb, LP)、adjacentEdge(cb) 共三个二级函数；其中 listDiv(cb, LP)、Trap(cb, LP)分别调用了 Div(cb, pos)、Danger(cb, lp)两个三级函数；Danger(cb, lp)调用了 Corner(cb, pos)、Edge(cb, pos)两个四级函数。另外 Corner(cb, pos)、Edge(cb, pos)亦单独被 MiddleJudge (cb, lp)调用。

先从两个四级函数说起：
Corner(cb, pos)：给定一个位置，判断我方下此位置后对方能否直接占角。时间复杂度为 $O(n^4)$。
Edge(cb, pos)：给定一个位置，判断该位置是否为边不稳定点。时间复杂度为 $O(n^4)$。

然后是两个三级函数：
Div(cb, pos)：给定一个位置，计算该位置散度。时间复杂度为 O(1)。
Danger(cb, lp)：给定合法位置列表，返回其中危险点的个数。时间复杂度为 $O(n^5)$。

然后是三个二级函数：

listDiv(cb, LP)：给定合法位置列表，计算其中每个位置的散度和，将这些散度和以列表形式返回。时间复杂度为 $O(n^2)$。

Trap(cb, LP)：给定合法位置列表，对于每个位置都计算了我们下这个位置后对方合法位置中非危险点的个数，返回每个位置对应的非危险点个数列表。时间复杂度为 $O(n^6)$。

adjacentEdge(cb)：判断棋盘四条边中是否存在相邻两条边上已有落子。时间复杂度为 O(n)。

最后是一级函数也是中局判断的主函数 MiddleJudge (cb, lp)：

通过一系列判断返回我们要下的位置，其具体思路及过程参见上文。调用函数顺序如下：

Corner(cb, i)
Edge(cb, i)
listDiv(cb, LP)
Trap(cb, LP)
adjacentEdge(cb)

时间复杂度为 $O(n^6)$。

由于对于残局并没有行之有效的解决方案，我们采取递归的方式获得残局的所有可能落子情况并据此选择最优的结果。

在残局的处理中定义了 finalTree 类，该类具有 chessboard（用于记录这种情况下的棋盘）, player（记录由上一层到这一枝的下子方）, pos（记录下子的位置）, children（记录该种情况下下一步下子的所有情况）, parent（用于记录该种情况的上一步）, gameover（记录当前棋局是否结束）, win（记录对于我方的胜负情况，关于 win 的具体说明将在下面提到）, result（记录这一枝的所有得分情况之和）, chance（记录该枝的胜率情况）等变量。使用了 insert 函数（用于在当前情况下加入子树）、getChessboard 函数（返回 self.chessboard）、getParent 函数（返回 self.parent）、getChildren 函数（返回 self.children）、getPlayer 函数（返回 self.player）、getPosition 函数（返回 self.pos）、getChance 函数（返回 self.chance）、getWin 函数（返回 self.win）、getResult 函数（返回 self.result）、gameOver 函数（返回 self.gameover）等接口。

同时我们定义了 copyCB 函数（用于复制棋盘）、overChecker 函数（用于检查棋局是否结束）、winChecker 函数（用于检查我方最后的胜负情况）、winEval 函数（用于计算我方的在每一步的胜负情况）、resultEval 函数（用于计算胜负总情况）、chanceUpdate 函数（用于更新胜率）、makeTree 函数（用于建树的主要函数）和 FinalJudge 函数（残局调用的主函数）用于辅助建树。

copyCB：需要两个变量，一个是 cb 即上一局的棋盘，一个是 pos 即将要下的位置。首先复制一个棋盘并对其进行 makeTurn(Pos)操作，然后返回这个棋盘。

overChecker：棋局结束有两种情况，一种是子数等于 64，另一种是双 PASS。

winChecker：需要两个变量，一个是 cb，另一个是 ourcolor 即我方棋子的颜色，返回字符串'win', 'tie', 'lose'分别表示胜平负。

winEval：这是一个很重要的函数，是建树中的核心算法。如果当前节点是叶节点，那么 self.win 就是 winChecker 所返回的情况；当前节点不是叶节点时，我们需要考察当前情况的子树的情况。此时，又要分成两种情况。一是如果当前节点的子树的 self.player 为我方即当前情况到下一情况是我方下子，那么如果所有的子树的 self.win 中有一个是 win，那么认为当前情况为 win；如果所有 self.win 都是 tie，那么认为当前情况为 tie；如果所有 self.win 不存在 win 并且存在 lose，那么认为当前情况为 lose。二是如果当前子树的 self.player 为对方即当前情况到下一情况是对方下子，那么如果所有的子树的 self.win 都是 win，那么认为当前情况为 win；如果所有 self.win 都是 tie，那么认为当前情况为 tie；

如果 self.win 存在 lose 的情况，那么认为当前情况为 lose。

resultEval：这也是根据当前节点的子树计算的。如果当前节点为叶节点，那么如果 self.win 为 win，返回[3, 3]；如果 self.win 为 tie，返回[1, 3]；如果 self.win 为 lose，返回[0, 3]（列表第 0 项表示得分，第 1 项为局数的 3 倍也即是最大的得分）。如果当前节点不是叶节点，那么当前节点的 self.result 的两项分别为所有子树的 self.result 的两项分别之和。同时调用 chanceUpdate 函数计算胜率。

chanceUpdate：即用 self.result 的第 0 项除以第 1 项（使用浮点数）。

makeTree：需要四个变量 cb, pos, first, t。其中 first 表示当前节点是否为根节点，t 为时间（由于建树过程中涉及大量的递归调用，耗时长，容易在这一步超时，因此记录时间，当时间超过某个值之后自动退出递归并改用 gambler 算法）。值得注意的是，树的根节点和其他节点需要区别对待。由于建树都是从我方执子时开始，则对于建树的最开始，我方并不关心上一盘棋子的下棋方是谁同时无需对棋盘进行变动，因此直接通过 ChessBoard 函数复制棋盘；而其他情况下我们需要知道上一个棋子是谁下的并且需要改变棋盘，因此我们通过 copyCB 进行操作。总体思路是根据当前节点的棋盘得到 legalPos 并由此建立所有子树，与此同时调用 winEval 函数和 resultEval 函数。

FinalJudge 函数：首先判断我们是否已经建立残局树，如果没有则要新建，如果有则直接从字典中调用。对于我们得到的残局树，在当前节点我们需要选择下一步操作，我们要对所有子树进行分类和排序。将 self.win 为 win, tie, lose 分类并且按照胜率大小排序。最终尽量按照 win, tie, lose 的顺序选择胜率大的子树情况。

其他操作涉及避免意外情况下的返回 pos 出错的问题以及避免超时的问题，分别采用 finalError 和 overTime 记录在字典中。

10.2.3程序限制

还不是很明确，如果在进入尾局树的时候双方的散度都很大有超时的可能，但是我们最终增加了一个防爆代码，进行强制退出，随意落子。

10.3实验结果

10.3.1实验数据

硬件 cpu 1.90GHZ 内存 4.0GB
版本 Windows 8.1
Python2.7

黑 \ 白	idiot 算法	gambler 算法	本算法
idiot 算法	0：0：1000 27000：37000 15.120：16.177	517：36：447 37321:26679 16.921:15.023	0：0：1000 15000：49000 12.581：20166.291
gambler 算法	397：32：571 23182:40818 12.892:14.902	422：50：528 30821:33179 13.241:13.914	2：0：998 10683:53307 10.342:26083.175
本算法	1000：0：0	1000：0：0	1000：0：0

	37000:27000 10923.777:14.129	47012:16988 17030.234:14.738	38000:26000 20720.731:14222.572

10.3.2结果分析

和 idiot 的对弈结果基本没有分析意义。由于我方算法没有随机性，和 idiot 的对局全是一样的。对于 gambler 中局算法起到了比较重要的作用，在只采用中局算法的情况下，胜率达到了 90%，这是由于 gambler 每次落子随机，我方的抢先占角策略常常能够成功占角，在中局就取得巨大优势。尾局算法的作用也不容忽视，对于每一步都是随机落子的 gambler，尾局算法将每一步都算出来的计算策略无疑能够抓住 gambler 的每一个失误。而开局部分对与 gambler 的对局结果影响不大，这主要是因为开局的 10 步之内，我方难以占据到巨大的优势。

在时间复杂度方面，基本在我们的预期之中，达到了我组尽量充分利用所有时间而且保证不超时的目的。主要是我们为了解决超时问题，在尾局中加入了防超时代码，在检测到可能超时时强行结束递归，按照已经计算出的最优算法落子。

在开局中局和尾局中，开局基本不消耗时间，中局平均每步会消耗 0.5 秒，总计消耗 20 秒左右，尾局在采用最大递归深度（9 步）的情况下经常会用掉剩下的所有时间，而将递归深度减少到 8 步时，基本可以控制在 100 秒左右,但会随着局势不同有所变化。

10.4实习过程总结

10.4.1分工与合作

会议名称：小组第一次讨论
会议时间：6 月 3 日晚 6：30-8：30
会议地点：泊星地
会议方式：自由讨论
会议内容：1.组员之间相互认识
2.马赞彭同学给大家讲解课程网站上黑白棋的基础代码
3.黄天正同学给大家介绍黑白棋中的基本概念，如散度，手巾，以及基本算法如散度原则，开局录入。
4.进行小组初步分工，分工如下：
刘松吟和庞磊同学负责开局，主要任务是建立一颗树，进行开局各种局势优势步的录入。
马赞彭和黄天正同学负责中局，主要任务是研究适合中局的算法。
柳晓萱和段鉴书同学负责尾局，主要任务是建立尾局的数，以方便递归和回溯。
5.本次会议确认了黄天正同学算法核心负责人的地位。

会议名称：小组第二次讨论
会议时间：6 月 7 日晚 7：45-9：00
会议地点：泊星地
会议方式：自由分组讨论，主要是负责同一块的同学之间进行工作的打通。

会议内容：1.每个小组同学向技术核心黄天正同学介绍自己算法的初步成果。
2.黄天正同学给大家提出优化意见。
会议收获，新一轮工作分配：
1.本次会议确认了整体黑白棋算法的基本框架，框架如下：
开局采用棋谱录入方法，确认 6 月 10 日晚之前完成 15 个子的棋谱录入（主要由庞磊同学负责）
中局采用散度算法，中局算法将会随着测试进行较大调整（由马赞彭和黄天正同学共同完成）
尾局采用递归算法，本次会议确认了递归结束条件，确认 6 月 8 日晚提交尾局算法代码（由段鉴书同学完成）
2.确认了除算法以外的工作：
（1）刘松吟同学负责将庞磊同学的开局工作转换成语言写入报告
（2）柳晓萱完成报告的尾局部分以及实习过程总结部分
（3）黄天正同学承担总程序的测试工作，并完成相应部分的报告

会议名称：小组第三次代码整合自由讨论
会议时间：6 月 9 日 13：00~15:20
会议地点：第二教学楼自由讨论区
会议方式：所有涉及代码编写，调试的同学一起讨论整合
会议内容：1.所有代码编写人员统一变量名称，适当增加接口。
2. 代码整合，经行调试。
会议收获，新一轮工作分配：
1. 首局完成了 10 个子的录入。
中局完成了大散度算法，散度算法，占角危险区特殊处理算法
尾局完成递归深度大于 10 的优化，使得递归深度至少可以大于 10，保证不超时。
算法整合调试全部完成，首局中局尾局完全相连无 bug！
2. 确认了下一步工作：
（1）刘松吟同学将首局算法复杂度等分析写入报告
（2）全面启动测试工作，包括极端情况处理，胜率分析
（3）首局继续进行录入工作，目标最终完成 15 子录入。
（4）中局算法思路整理，进行优化，预计 6 月 11 日最后一次代码处理会议马赞彭同学全面展示中局算法思路。
（5）尾局进行剪枝算法研究，优化递归步骤。

会议名称：小组第四次代码成果展示及代码确认
会议时间：6 月 11 日 21：00~22：00
会议地点：理科教学楼一间教室
会议方式：代码核心人员进行程序展示
会议内容：1.分别测试了三个版本的代码与网页游戏中级，高级所下情况。
2. 确认热身赛提交文件版本。
会议收获：
1. 确认了参加热身赛的五个代码版本。
2. 对尾局递归深度的确认。

10.4.2经验与教训

相比于研究与 gambler 对战的胜率，我组更注重与像 WZebra 这样的成功黑白棋程序的对弈结果。作

为一个棋手，只有和高手对弈才能得到迅速提高，相信程序改进也是如此。与 WZebra 对弈有几个明显的好处。

首先，可视化的对弈方式使得我们可以看到我们的算法策略是否得到了实现。在我们的算法中，有大量的 BUG 是在与 WZebra 的对弈中找出的。

其次，在与 WZebra 的对弈过程中，我们经常能找出思考中的遗漏，例如危险点（2）就是在与 WZebra 的对弈中被找出的。

最后，WZebra 的复盘功能能够给出每一步的估值，数字为 WZebra 利用巨大数据库计算得到的最后胜子情况，因此，与 WZebra 对弈能够找出我们的算法思路有什么问题。

不过，与 WZebra 对弈也有很多问题。

一方面，我们很难找出自己算法的进步，这主要是因为 WZebra 太强了，无论我们如何改进算法都会惨败，而 WZebra 毕竟不是一个真正的黑白棋高手，不可能告诉我们我们的棋力有没有进步，哪里还有缺点。不过这个问题在与 gambler 的对弈中也存在，在百分之 97 以上胜率的情况下，很难说胜率的提高与降低到底是随机性的影响还是算法有了进步。

另一方面，WZebra 收录的棋谱大多是棋手之间的对弈和棋手与高端黑白棋程序之间的对弈，其给出的每一步估值对我们这种程度不一定是适合的。例如，我们依靠 WZebra 的估值做出的开局程序就没有得到预期的效果。

在与 WZebra 的对弈中我们发现了大量的不足。

首先，散度+行动力的判断方式经常会遭遇例外，可能是有某些我们不清楚的原因使某些位置的实际优先度被提高了，不过，由于我们也不太会下黑白棋，并没有能够总结出这样的原因。

其次，我们在程序的分段上少了一个阶段。在中局和尾局之间还应该有一个阶段，我们暂且叫它渐尾局。在这个阶段，危险点有可能变得不那么危险，甚至变成一步好棋。因此，在与 WZebra 对弈时，常常出现对方明明下入了危险点，吃亏甚至崩盘的反倒是我们。

最后，强制占角的设定太过机械，强大的程序往往能够通过更好的下棋次序来取得更大的优势，而我们的程序往往因为强制占角使我们虽然逼对方下入危险点，却没有获得很大优势。

面对这些问题，我们想了很多办法，例如计算稳定子，改进行动力算法等，但是一方面这些算法的思路极其复杂，在转化为程序的过程中遇到了大量困难，而且时间复杂度使我们不是很能接受，，另一方面他们的效果并不是很大。

虽然在与高手 WZebra 的对弈中我们输的一败涂地，但是我们的算法总的来说还是取得了不小的成功。通过多次改进，我们在对阵 WZebra 时终于不会再下出像新手一样前期几乎将对方吃光，后期却被翻盘的棋局。也不会出现开局或中盘就被对方取得巨大优势的棋局。在与其他小组的比赛中，我们成功出线，还在八强赛中执黑击败了小组赛中无一败绩的夺冠热门 Charlie，如果不是平局还要算子数差距，甚至可以和 Charlie 打平。

不过，在比赛中也暴露了我们小组程序的另一个问题：缺乏随机性。不仅我们的算法完全没有随机性，而且不同版本之间的思路也基本相似。这就导致了如果遇到克制我们的算法，我们几乎没有反击的能力。

10.4.3 建议与设想

1. 提高代码的随机性，不然两个队打三局结果都是一样的。
2. 如果打平，先比子数，再比时间的方法感觉还值得商量，因为本身大家的算法都是以赢为目的的，并没有考虑子数的问题，如果还要考虑子数，算法会比较复杂，而半决赛决赛的胜负基本都是靠子数分出的，所以感觉不是非常客观。

10.5致谢

感谢所有小组成员的热情配合，辛苦工作，勤勤恳恳，任劳任怨。
以及实习中老师的时时跟进督促，热身赛的黑白料理。

10.6参考文献

- 黑白棋指南：http://www.soongsky.com/strategy2/
- 黑白棋天地：http://www.soongsky.com/
- 《黑白棋指南》 Brian Rose 著 2004 Anjar 公司版权所有

南区 South　ROMEO 组

11 数算实习报告

组员：虞志刚　周易　钟涛　杨冬偶　姜志远 *

11.1 总体思路

经过本人与电脑上黑白棋小游戏上百局的厮杀，终于领悟到一些诀窍，根据我的经验，总体思路为防守反击类型，在比赛的时候，相信大家对决赛中有一局某一方一度剩下不到 5 个棋子，后来反败为胜的这一幕记忆犹新。我的大致思路就是占据有利位置，不要在乎目前的棋子比例，和一些同学讨论时他们提出了最大利益法，即每一步都采取能获得最大即时利益的走法。一开始我也想过类似的方法，发现效果并不如人意。为了验证我的想法，我在一方的原程序里加了“Scorelist=cb.getScore()　print Scorelist”以探求过程中的比分。发现，不仅胜率不高，经常性的出现比分一度大领先后被反超的情况。所以我舍弃了原来最大即时利益算法。

另外，还有一种稳定子的算法，稳定子我不喜欢，因为稳定子其实“不稳定”，稳定子不一定总是存在，所以我决定以我在与电脑的对弈中总结的经验来重新编程。

首先，以占据四角为最大优先，无论现在比分，总棋子数如何。优先四边中，四边中就是 A3,A4,A5,A6,C1,D1,E1,F1 等四条边中间四个。因为在边上的棋子更加稳定，而且更容易占据四角，四角为什么这么重要，就是因为四角绝对稳定，而且有利于控制四边，四边一旦与四角连接在一起，就是稳定边，而且稳定边对于中间的棋子也会有很大影响。这就是四角的重要性。四边中也是为了四角，比如 A2，A7 一旦自己下子，对方很可能立刻占据 A1 或 A8，所以类似于 A2A7B2 的位置不到最后不能碰，所以类似于 A3A4A5A6 很重要，其中 A3A6 更加重要，总之越在中间越危险。在 48 个子之前，除非能拿四角，否则四角周围 4*3 个位置是禁区。

11.1.1 算法流程

判断是否是 pass，若不是则，看是否有四角，若有立刻占上，如无，判断此时子数，若大于 48，就取第一个或者随机，低于 48 优先 A3A6 之类的位置，次优先 A4A5 之类的位置

11.1.2 算法复杂度

算法较为简单，复杂度为 O（n）

11.1.3 极端情况

极端情况为在 48 子之内出现只能下在 A2B1B2 类似的位置，不过几乎不可能。

11.2实验数据

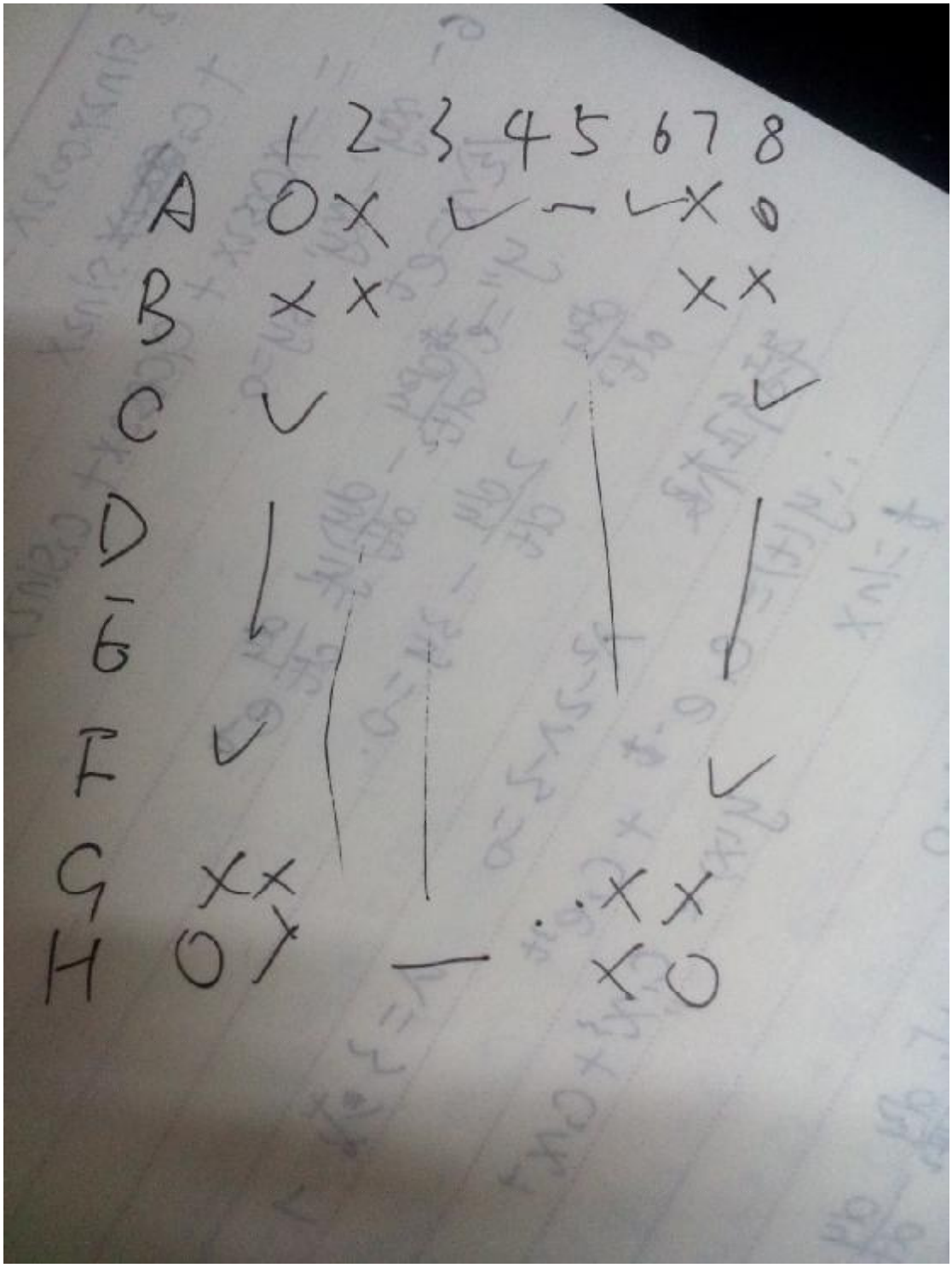
WIN8.1，Python2.7

1000 实验结果汇总

	Idiot（先）	Gambler（先）	本算法（先）
Idiot	１２７：８７３ ２７:３７	９８３：２７ ４３：２１	1000:0 43:21（一直） T（先）=19.9s T（后）=12.2s
Gambler	５８５：４１５ ３７：２７	７３６：２６４ ３６：２８	９６１：３９ ３９：２５
本算法	0:1000 10:54（一直）	６６：９３４ ９：５５	0:1000 18:46（一直）

结果分析：

很明显结果在预料之中，而且根据分析后手战优势，因为先手会走第 49 子即，在本算法的对决中，先手会先走 A2B1B2 类似的禁区子，所以不占优势。



11.2.1分工与合作

周易　钟涛　姜志远＊：代码编写
虞志刚　杨东偶：撰写报告

11.2.2经验与教训

在本次实习中，各位组员都深刻理解了团队合作的精神，但在时间安排及任务分配上仍存在不明确之处．

11.2.3建议与设想

暂无 . . . 建议实习作业总时长适当增加

11.3致谢

感谢各位组员在本次实习作业中的合作，以及陈斌老师这一学期对我们的教导．

南区 South　ALPHA 组

12　数据结构与算法课程实习作业报告

李子涵*　刘明辰　葛天雨　周杰　唐钰开　杨礼萌

摘要　　本算法以阿尔法贝塔剪枝法作为核心，辅以稳定子和行动力算法为其估值，在最后加入遍历算法寻找必胜落子法，并利用改变递归深度调节算法的思考深度。本算法涉及列表、栈、字典等数据结构，并自定义了一个类——遍历树。实验数据结果都在我们的意料之中，对阵 idiot 完胜，对阵 gambler 2000 场只输了 3 场，而自己与自己对决中先手完胜后手。

关键词　**阿尔法贝塔剪枝法**　**稳定子**　**行动力**

12.1算法思想

12.1.1总体思路

下棋不仅要考虑当前的局势，还要有所远虑，要应对地考虑到接下来的局势变化，然后把二者的因素同时考虑进去，才能够保证每步棋的质量。因此我们组本次的算法是通过综合当下与未来两步以内局势的变化，通过递归的方法，又加以利用赋予权重值的大小来代表棋势的好坏的方法，来决定每一步棋的走势。

我们组本次的代码总体上可以分为两个阶段，前期与后期，每个阶段所采用的策略是不同的。

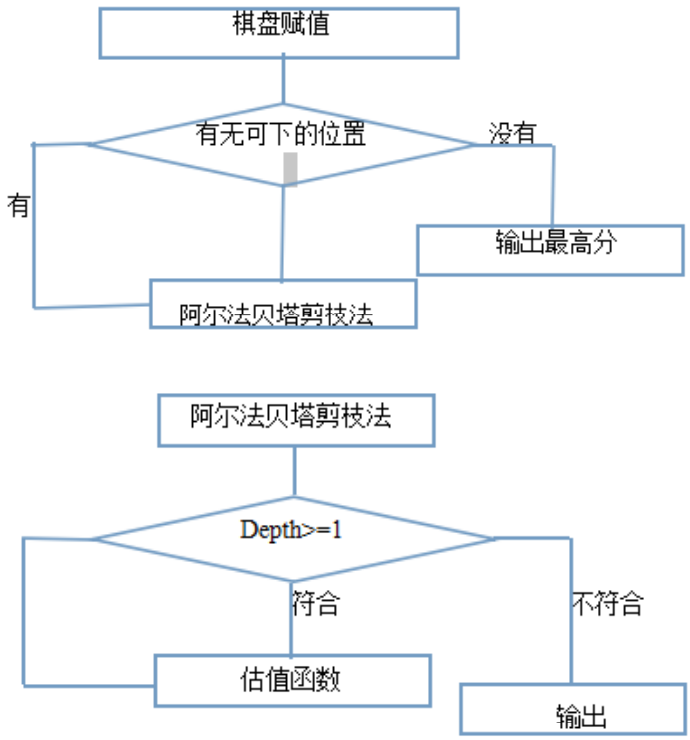
在前期的算法当中，我们主要有四个考察因素：行动力(指的是下黑白棋时双方落子时可选的落子位置的数量)，稳定子（永远不会被吃掉的子），边缘子（位于边缘位置可能会为对方提供行动力的子），每个位置的重要性（通过一个二维列表存储），以及每个部分的权重值（根据大量的数据测量得出的合适赋值）。

在前期算法中，结合了四个主要因素之后，我们通过阿尔法贝塔剪枝法的方法来进行递归，从而得出 2 步之后的棋盘走势，并通过 evaluation 这个估值函数得出的局势的好坏情况，将二者综合起来考虑得到当前最优的走棋方式。

在后期算法的时候，我们采用了遍历的方式。通过树的数据结构，采用动态规划，把所有可能性都算出来，如果能够必胜，就筛选出来，否则就还是采用前期的方法来进行计算。

12.1.2算法流程图

①前期算法的流程图：



图例说明：估值函数中是根据行动力，边缘子，稳定子，和棋盘位置评估值来决定的。

②后期算法之遍历：

遍历树内容参见 2.1。

12.1.3算法运行中影响时间复杂度成分分析

算法的时间复杂度主要取决于最后的完全遍历和前中期的阿尔法贝塔剪枝法部分遍历。具体时间复杂度应该在最差情况和最优情况之间。但由于每一步下棋后可下位置并不唯一，因此时间复杂度在不同情况下不确定，这导致算法总时间在一定范围内存在波动（且有时波动较大）。

另外估值函数本身也影响着剪枝的效率，进而影响着运行时间。后期时间复杂度约为 O(n!)，前期剪枝法复杂度约为 $O((1/k) *[n(n+1)(n+2)...(n+m)]^2)$（k 取决于估值函数，m 取决于层数）。

12.2程序代码说明

12.2.1数据结构说明

算法包含的主要数据结构有列表、栈、字典，自定义类是定义了一个遍历树（traversalTree）。
遍历树：用于棋局末期，把所有可能性都计算出来，如果能赢，就选出来，复杂度基本约为 O(n!)。

图例：**设己方执黑，假设此步有三种可能**。每一步黑棋之后，无论白棋怎么走，**都**可能赢，则该步黑棋有效；每一步白棋之后，无论黑棋怎么走，**存在**赢的情况，则该步白棋有效。
即某一节点的值，若其子节点为白色，则 and 运算；若其子节点为黑色，则 or 运算

统计完成后，选择为 T 的黑色，即为当前该走的位置。

递归到棋局结束

12.2.2 函数说明

play(cb,ms)函数：下棋主函数。
alpha_beta(cb,alpha,beta,depth,end)函数：阿尔法贝塔剪枝法函数。
evaluation(cb)函数：估值函数，对当前棋局进行计算、评估，用来判断接下来落子的位置。我们的估值函数主要将根据两大部分——行动力和稳定子个数来判断。
activity(cb)函数：判断行动力的函数，行动力即在自己下完一步之后对对方可落子处和自己可落子处的综合估计。
find_stable(cb) 函数：寻找稳定子函数，稳定子即落子后永远不会被吃的子。
full_x(cb,x,y1,y2)函数：寻找稳定子函数的子函数
full_y(cb,y,x1,x2)函数：寻找稳定子函数的子函数
traversalTree 类：用于最后几步的遍历
__init__(self,chessBoard,pos=None):声明函数
self.rate 子节点能赢的概率
self.child 子节点
self.father 父节点
self.turn 执子方
appendChild(self,pos)：增加子节点
getRate(self)：获得子节点能赢的概率
getValue(self): 获得估值
getChessBoard(self)：获得棋盘

getTurn(self)：获得执子方
getPos(self)：获得落子位置
getChild(self)：获得子节点
buildTree(father,cb,ms):构建遍历树
result(root,myturn)：递推计算出各个节点输赢的 value 值
Traversal(cb,ms)：对遍历树进行遍历

12.2.3程序限制

无程序限制。

12.3实验结果

12.3.1实验数据

实验环境说明：

- 硬件配置：CPU: Intel(R) Core(TM) i7-4810MQ CPU @2.80GHz 2.80 GHz
  内存：8.00 GB
- 操作系统：Windows 7 旗舰版 32 位操作系统
- Python 版本：Python 2.7.9

1000 次对弈结果数据汇总

黑 \ 白	idiot 算法	gambler 算法	本算法
idiot 算法	胜负局数比：0:0:1000 累计总分比：27:37 累计时间比：0.09390：0.012423	胜负局数比：537:41:422 累计总分比：44:20 累计时间比：0.013171：0.011687	胜负局数比：0:0:1000 累计总分比：28:36 累计时间比：0.012467：28.719829
gambler 算法	胜负局数比：412:29:559 累计总分比：31:33 累计时间比：0.027084：0.033118	胜负局数比：450:37:513 累计总分比：31:33 累计时间比：0.031018：0.028784	胜负局数比：0:0:1000 累计总分比：18:46 累计时间比：0.011394:33.004074
本算法	胜负局数比：1000:0:0 累计总分比：53:11 累计时间比：27.195807:0.009558	胜负局数比：997:0:3 累计总分比：48:16 累计时间比：29.490126:0.008810	胜负局数比：1000:0:0 累计总分比：41:23 累计时间比：27.528448:14.767123

12.3.2结果分析

在与 idiot/gambler 算法的对弈过程中，本算法的阿尔法贝塔剪枝法起到了决定性的作用，而行动力函数和寻找稳定子函数则是锦上添花，让本算法吃子更多。运行时间在预期的范围之内，95%的运行时间开销发生在最后的遍历环节，其他环节则只占 5%。

12.3.3经典棋局

正式比赛时曾经以 159.1 秒击败了超强代码 LIMA 组（更改了阿尔法贝塔剪枝深度）

12.4实习过程总结

12.4.1分工与合作

组内分工	代码编写：刘明辰，周杰，杨礼萌，李子涵 报告编写：唐钰开，葛天雨，李子涵 报告编辑：李子涵
交流方式	1.微信群上讨论 2.定期召开寝室组会
组会记录	6 月 7 日　确定组内成员分工，并将程序分为稳定子、行动力、楔入、最后遍历四个部分 6 月 8 日　宿舍讨论，共同编写代码 6 月 10 日　代码完成，测试发现战斗力有限，于是借助网络搜索资料，得知用阿尔法贝塔剪枝法来为棋盘估值是黑白棋 AI 普遍且实用的算法，于是我们决定在代码中加入阿尔法贝塔剪枝法 6 月 12 日　不断完善代码，并侦查其他队伍的算法和热身赛成绩 6 月 13 日　不断调试递归深度，使代码在不超时运行的前提下能够达到最优 6 月 14 日　将写好的两个不同递归和遍历深度的代码上交参加第二次热身赛

12.4.2经验与教训

在本次大作业的合作编写当中，我们意识到代码写的好坏，不是取决于一个人的专业知识的好坏，而是取决于小组的每一个成员的分工是否明确，合作是否默契，以及每个小组成员是否尽心尽责。

由于大家都是来自于不同的班级不同的宿舍，以及这个团队以前没有任何的合作经验，所以在编写代码的过程中，小组成员的默契程度还稍欠火候。比如编写代码的同学在合写代码的时候往往没有 debug 便上交，造成了不小的麻烦。同样每个人编写的时候，总会在一些细枝末节处犯错，然而大的方向却很少错。

但是在这次编写大作业的过程中，更多地是获得了经验。比如我们在一开始的组队之时便明确了分工，而这也大大提升了我们的工作效率。另外小组内编写代码的成员都分别提出了自己的看法，使得百家之思想融会贯通，才有了我们今天的代码。同样，我们也不断的改进自己的代码，虚心看待我们与别的组的差距，从而使我们的代码不断的完善起来。

总之在这次作业的编写当中，虽然有很多不足，但是可贵的是每个人都付出了自己的努力，为我们这个团队贡献出自己的一份力。

12.4.3建议与设想

我们团队认为，本学期的数算课的各种作业（从海龟作图到今天的大作业）可以极大程度的调动同学们的兴趣爱好。但是我们团队也认为本次大作业还是有一些不足（基础设施代码的不足就不在这里加以讨论）。首先是本次竞赛的组织分组上，由于这门课不光有大一选，更多还有信科的同学，遥感方向的高年级同学，而这也造成了不公平性，所以我们建议对于这些同学不应该组成一队，理应一人一队或者最多两人一队。其次在竞赛赛程中，理应由熟知各个组实力的普通同学们进行投票，分出一二三四档队伍，然后合理的分出小组，以免有死亡之组的诞生。

12.5致谢

在这里编者感谢每一位组员的付出，以及姜城，石瀚文师兄，还有陈斌老师不断地帮助我们并为我们提供测试数据。

12.6参 考 文 献 和 资 源

黑白棋天地　http://www.soongsky.com/

南区 South　LIMA 组

13　数据结构与算法课程实习作业报告

蒋明轩　于润泽　汪建峰*

摘要：（简要介绍算法原理，涉及的数据结构与算法，实验数据结果概述）
黑白棋（Reversi）又称翻转棋，是一个经典的策略性游戏。根据下棋的经验，当棋局中的某一步有好几种走法可选时，棋手要作出决策，选择对自己最有利的走法。下棋是一个典型的信息完全、零和博弈过程，博弈者的策略搜索过程可用树结构表示，称为博弈树，树上的节点为棋盘合法状态，接下来将通过算法思想、代码说明、实验结果、实验过程等方面介绍本次大作业经历。

关键字：博弈、最大最小搜索、α-β剪枝、估值函数、Lima、冠军

13.1算法思想

13.1.1总体思路

13.1.1.1　极大极小搜索

我们假设参与博弈的两个人为 Black 和 White，从 Black 的角度看， Black 每一步行动总是争取自己的最大收益，称为 Max 过程，对应博弈树上的节点称为 Max 节点；由于 Black 对 White 的棋力并不清楚，所以只能认为 White 的每一步总是使自己的收益最小，称为 Min 过程，对应博弈树上的节点称为 Min 节点；于是下棋其实就是 Max 和 Min 交替进行的过程（当然，一方停步时例外）。考虑前面的单步搜索模型，我们假设此时轮到 Black 下， Black 发现自己所有可能的行动派生的子节点中评估值最大的节点为 S'i ，那么最大收益就为 V (S'i) – V (S) ，对应的行动 ai 就是单步搜索所能找到的最优解。但如果 Black 向

前搜索两步呢？他也许会发现 S'i 所派生的子节点中的最小值为 V (S''i) 小于某个其他节点所派生的子节点中的最小值 V (S''j)，不妨设 S'j 为同层节点中所派生的子节点的最小值中最大的节点，那么他转而应该选择 S'j 对应的行动 aj 作为自己的最佳行动，因为选择 aj 的收益最差不会少于 V (S''j) – V (S)，而这已经是最好的情况了。如果搜索继续加深，那么选择会越来越“明智”。

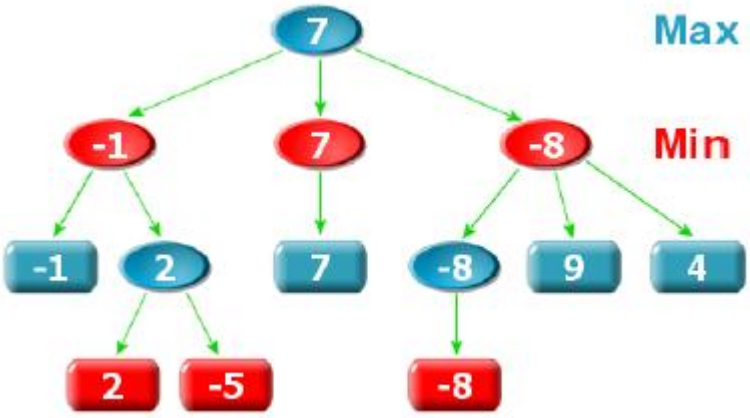


图 1　极大极小值搜索示意图

根据上图可知：

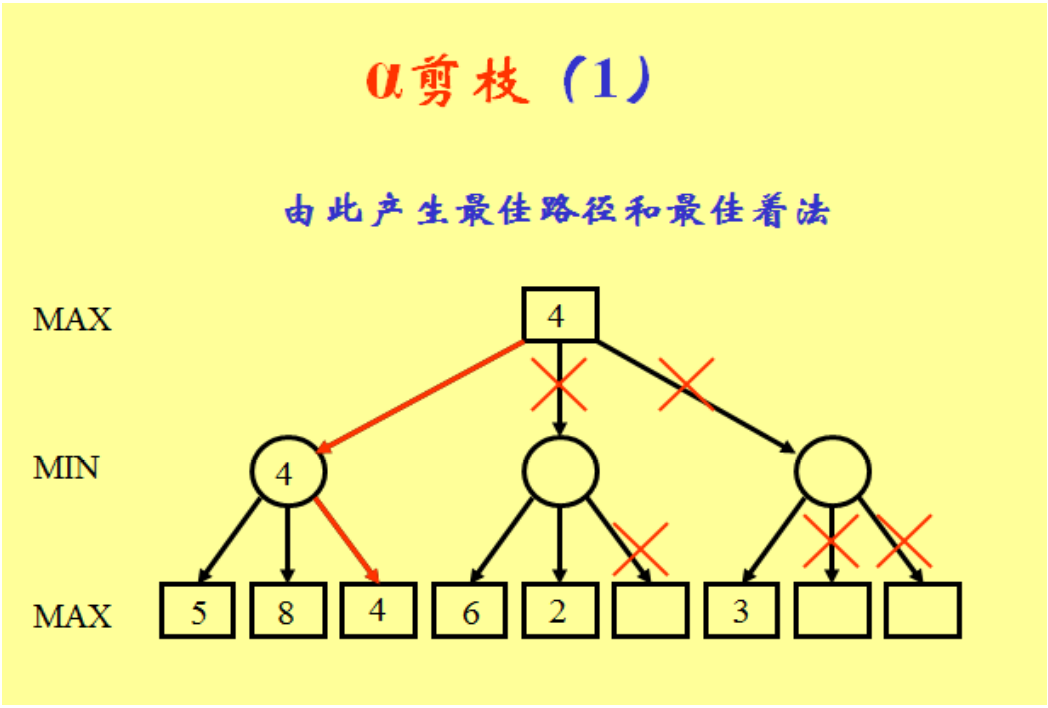
1. 自己可以下的节点为 Max 节点，对方可以下的为 Min 节点；
2. 叶子节点调用评估函数，并将其值向上传播；
   (a) Max 节点被赋值成其所有子节点中的最大值；
   (b) Min 节点被赋值成其所有子节点中的最小值；
3. 根节点在其子节点中选择值最大的节点所对应的动作作为最佳行动。

对于一般的极小极大搜索，即使每一步只有很少的下法，搜索位置的数目也会随着搜索深度呈指数级增长。在大多数的中局棋形中，平均每一步都有十种走法，如果向前搜索的深度为九步，那么为了进行一步着子，访问的棋局数就达到了十亿个，一般的电脑是很难胜任的。此时 alpha-beta 剪枝搜索就成了解决这一问题的关键所在。

13.1.1.2　alpha-beta 剪枝搜索

一种基于剪枝（ α - β cut-off）的深度优先搜索（depth-first search）。将走棋方定为 MAX 方，因为它选择着法时总是对其子节点的评估值取极大值，即选择对自己最为有利的着法；将应对方定为 MIN 方，因为它走棋时需要对其子节点的评估值取极小值，即选择对走棋方最为不利的、最有钳制作用的着法。

在对博弈树采取深度优先的搜索策略时，从左路分枝的叶节点倒推得到某一层 MAX 节点的值，可表示到此为止得以“落实”的着法最佳值，记为α。显然此值可作为 MAX 方着法指标的下界。在搜索此 MAX 节点的其它子节点，即探讨另一着法时，如果发现一个回合（2 步棋）之后评估值变差，即孙节点评估值低于下界α值，则便可以剪掉此枝（以该子节点为根的子树），即不再考虑此“软着”的延伸。此类剪枝称为α剪枝。



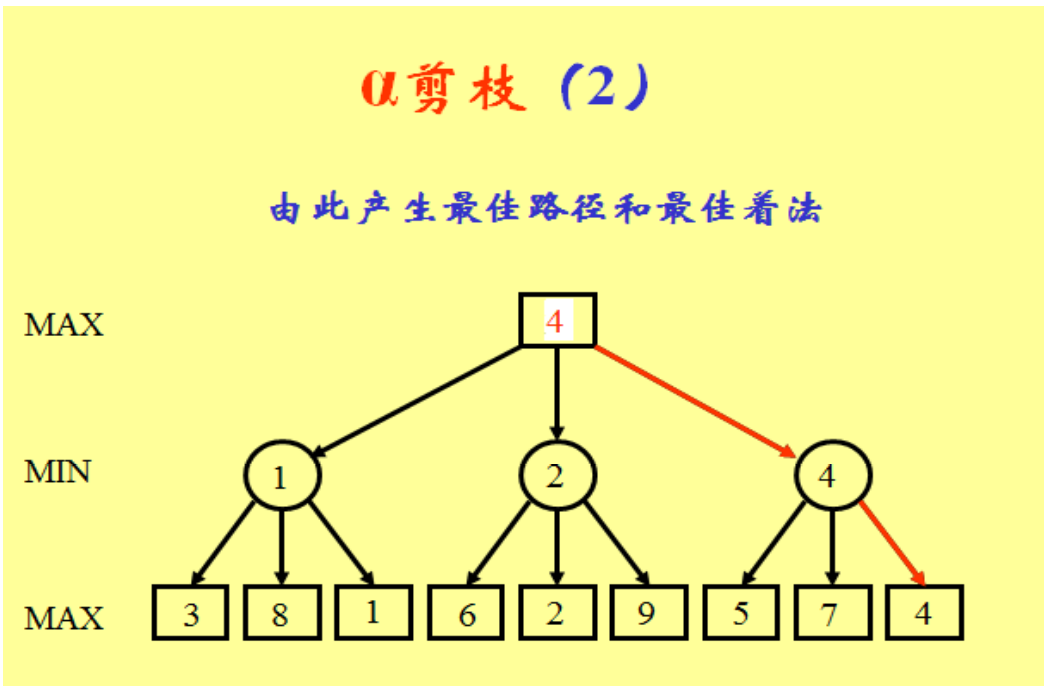


图 2　α - β 剪枝示意图（一）

同理，由左路分枝的叶节点倒推得到某一层 MIN 节点的值，可表示到此为止对方着法的钳制值，记为 β 。显然此 β 值可作为 MAX 方无法实现着法指标的上界。在搜索该 MIN 节点的其它子节点，即探讨另外着法时，如果发现一个回合之后钳制局面减弱，即孙节点评估值高于上界 β 值，则便可以剪掉此枝，即不再考虑此“软着”的延伸。此类剪枝称为 β 剪枝。

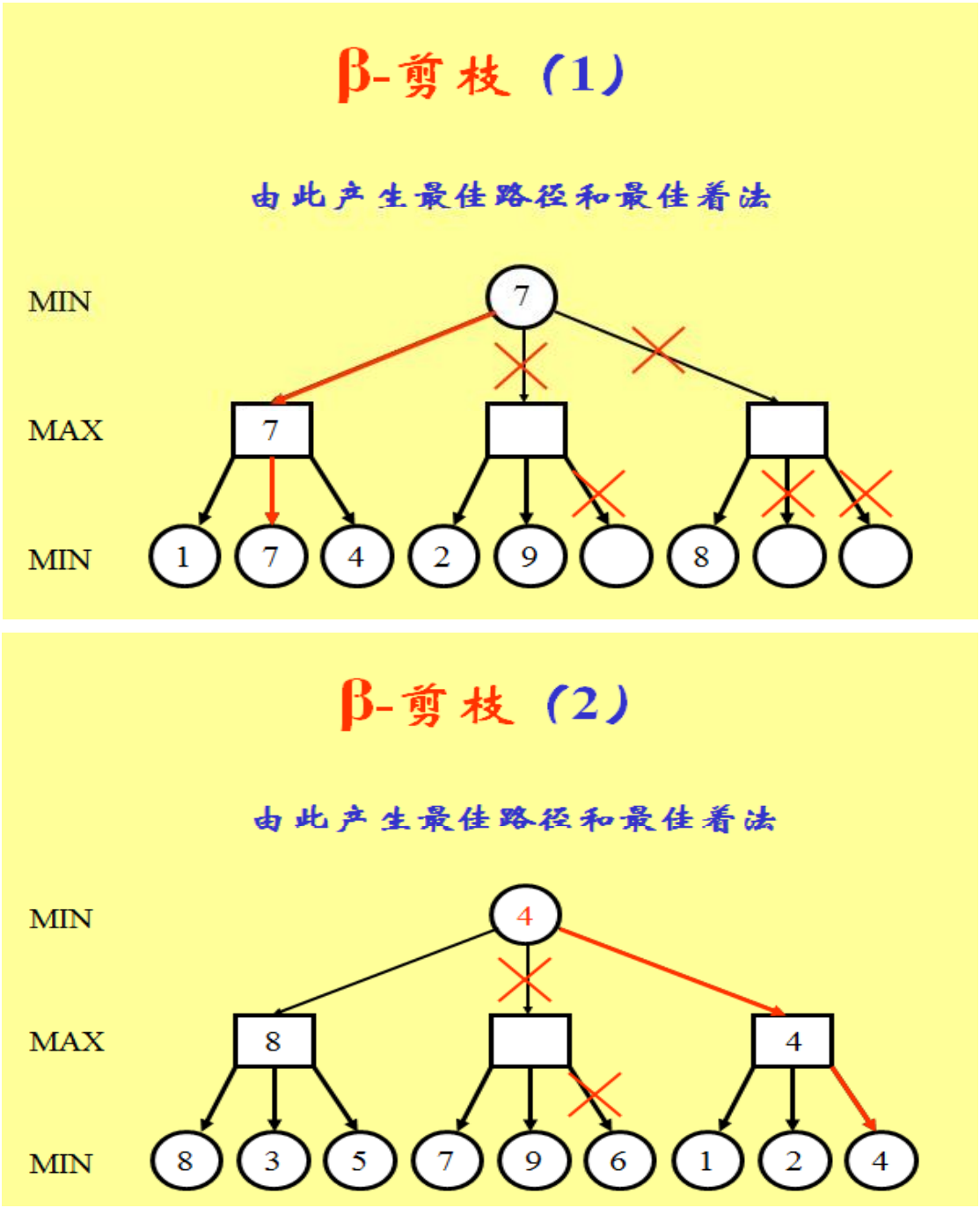


图 3　α - β 剪枝示意图（二）

α - β 剪枝是根据极大-极小搜索规则的进行的，虽然它没有遍历某些子树的大量节点，但它仍不失为穷尽搜索的本性。

α - β 剪枝原理中得知：

α 值可作为 MAX 方可实现着法指标的下界；β 值可作为 MAX 方无法实现着法指标的上界；于是由 α 和 β 可以形成一个 MAX 方候选着法的窗口；也便出现了各种各样的 α - β 窗口搜索算法。

13.1.1.3 估值函数

一个简化的版本中，主要采用了基于棋格表和行动力（mobility）的估值。棋格表其实就是棋盘上不同位置的权值表，一方的行动力是其可以着子的位置的总数。如果用 S 表示棋盘的状态（状态可定义为每个格子的棋子有无和颜色），那么基于以上模型可知：

$$V(S) = \sum_{i=1}^{8}\sum_{j=1}^{8} \omega[i,\ j] \cdot s[i,\ j] + \text{Mobility}(\ my\_color)$$

$$s[i,\ j] = \begin{cases} 1 & color[i,\ j] = my\_color \\ -1 & color[i,\ j] = opp\_color \\ 0 & \text{otherwise} \end{cases}$$

color[i, j] 表示棋盘上第 i 行第 j 列（简记为(i, j)）处的颜色，可以是己方颜色（my color）、对方颜色（opp color）或空白。ω [i, j] 即为(i, j) 处的权值，是经验数据。Mobility(my color) 是己方的行动力大小。

13.1.2算法流程图

主程序：

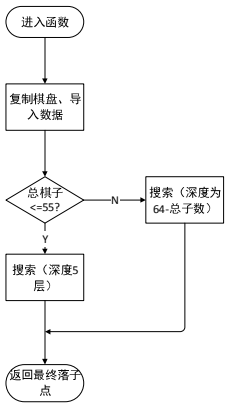


图 4　算法程序主流程图

搜索程序：

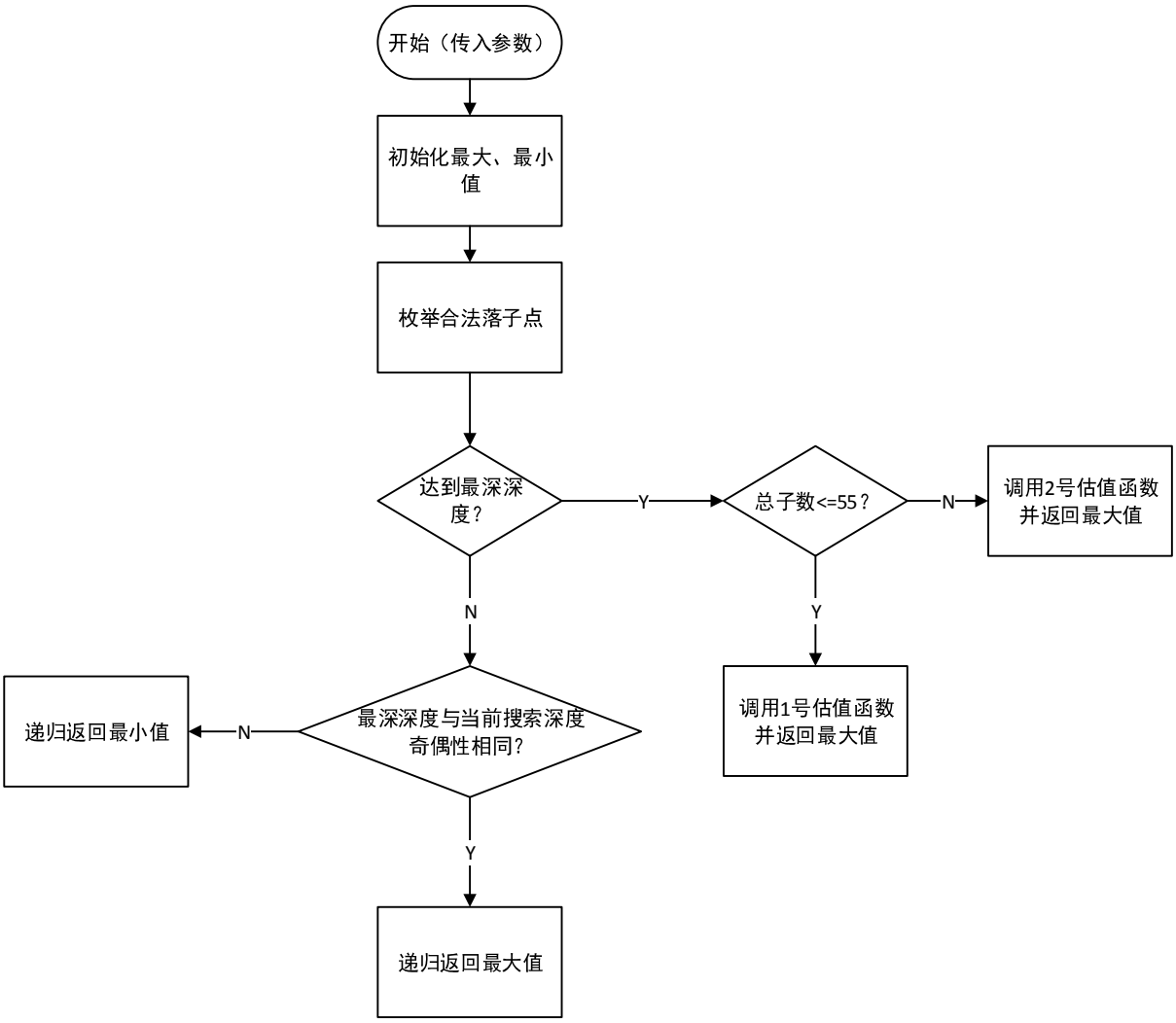


图 4　算法 search 函数流程图

13.1.3算法运行时间复杂度分析

搜索时间复杂度为 O（n^5），其中 n 为合法下子点的个数，因为在开局和终局时合法下子点的个数较少，故在开局和终局时程序运行时间较短，而在中局时合法下子个数很多，程序运行时间较长。

13.2程序代码说明

13.2.1数据结构说明

本次大作业使用了二维数组来将棋盘进行存储，代码如下：

```
a = [[p for p in r]for r in pa.board]
timecost0 = pa.timeCost
legal0 = pa.getLegalPos()
legal = [Pos2rc(p) for p in legal0]
stepnum = pa.getTotalScore()
b = [ [10,10], [10,10], [10,10], [10,10], [10,10], [10,10], \
      [10,10], [10,10], [10,10], [10,10], [10,10], [10,10] ]
```

将当前棋盘 pa 存储到二维数组 a 中。Timecost0 将用时导入，主要是调试程序时看是否超时用的。Legal0 获得合法下子位置。legal 将字母数字记录法转成纯数字记录法，我的程序中此后都用纯数字记录。stepnum 获得当前棋盘上总子数，b[数组用来记录 search 中每步下的棋子位置，即 b[n]是用来记录 search(n)的落子位置。

13.2.2函数说明

下面用图表来简略说明代码中所包括的函数与功能：

函数	功能
def getScore(a)	获得两方棋子数的函数
def getLegalPos(a, opturn, turn)	获得合法下子点的函数
Def makeTurn(a, pos, turn, opturn)	落子函数，与老师不同之处在于记录了落子后棋子变化
def eval2(aa, turn)	后期估值函数
def eval1(aa, turn, opturn)	前中期估值函数
def search(n, deepth, alpha, beta, legal, turn, opturn)	递归搜索函数
def play(pa, ha)	主程序

接下来，我将着重介绍我组在本次大作业中特有的函数部分：

首先，是落子函数，此处与老师样例程序不同之处在于记录了落子后棋盘上棋子的变化，并将其存在 change 中，递归搜索回溯时恢复原棋盘时会用到。在我们最初版本的程序中，并不是用的这种记录方式，最开始我们每次尝试下子前都将整个棋盘记录下来，回溯时再将整个棋盘恢复到原样，这样每搜索一个位

置就要记录整个棋盘，该位置搜索完之后又要复原整个棋盘。之后我们重写了 maketurn 函数，将下子后棋盘上棋子改变的数据提取出来，记录到 change 数组里，这样搜索一个位置时就不需要记录整个棋盘，而回溯的时候也只需要将 change 里记录的位置恢复原状，这个改进使得我们是算法时间节省了大约五倍，而搜索深度也由原来的四层提升到五层。

```
def makeTurn(a, pos, turn, opturn):       #落子函数，与老师的不
    def reverseByDir(dr, dc):
        pr, pc= r+ dr, c+ dc
        opPiece= False
        while (pr in Line) and (pc in Line) and \
            (a[pr][pc]== opturn):
            opPiece= True
            pr, pc= pr+ dr, pc+ dc
        if (pr in Line) and (pc in Line) and opPiece and \
            (a[pr][pc]== turn):
            pr, pc= pr- dr, pc- dc
            while a[pr][pc]== opturn:
                a[pr][pc]= turn
                change.append([pr, pc])   #每次下子后对应的optu
                pr, pc= pr- dr, pc- dc

    change = []
    [r, c] = pos
    if pos== 'PASS':
        if getLegalPos(a)!= []:
            return 'INVALID_PASS'
        return 'PASS'
    a[r][c]= turn
    change.append([r, c])                 #change[0]记录的是当前
    for [dr, dc] in [ [-1,-1], [-1,0], [-1,1], [0,-1],[0,1],\
        [1,-1],[1,0],[1,1] ]:
        reverseByDir(dr, dc)
    return change
```

其次是估值函数，分为前中期估值和后期估值，这很有必要。在开局、中局和终局的过程中，所面对的情形不同，因此需要两到三个估值函数的使用才能理清场上形势。开局的估值考虑用权值表和行动力，权值表中角和边所占比重较大，而行动力的权重也相对较大。终局的估值函数仅考虑棋子数之差，并且一直搜索至棋局结束。

```
def eval2(aa, turn):                    #后
    zi = getScore(aa)
    if turn == B:
        return zi[1]-zi[0]
    elif turn == W:
        return zi[0]-zi[1]
    else:
        print('Wrong')

def eval1(aa, turn, opturn):            #前
    xing = 0
    score = 0
    for i in Line:
        for j in Line:
            if aa[i][j] == Y:
                xing = xing + 1
            elif aa[i][j] == turn:
                score = score - power[i][j]
            elif aa[i][j] == opturn:
                score = score + power[i][j]
    return score - xing * 100
```

最后是搜索函数，正如之前所谈及的最大最小搜索。我们这个 AI 的搜索深度达到了 5 层，与此同时也有正确使用阿尔法贝塔剪枝，因此才能在搜索深度足够高的情况下也不会超时，我认为这可能是我们组优于其他小组而获得冠军的原因，代码如下所示：

```
def search(n, deepth, alpha, beta, legal, turn, opturn):
        maxv, minv = -1000000, 1000000
    check = True
    for [i, j] in legal:
        change = makeTurn(a, [i, j], turn, opturn)
        nextlegal = getLegalPos(a, turn, opturn)
        check = False
        if n == deepth:
            if   stepnum <= 55:
                eva = eval1(a, opturn, turn)
                if eva > maxv:
                    maxv = eva
                    b[n][0], b[n][1] = i, j
            elif stepnum > 55:
                eva = eval2(a, opturn)
                if eva > maxv:
                    maxv = eva
                    b[n][0], b[n][1] = i, j
        else:
            if (deepth%2==1 and n%2==1)or(deepth%2==0 and n%2==0):
                nexts = search(n+1, deepth, minv, maxv, nextlegal, \
                opturn, turn)
                if nexts >= maxv:
```

```
                        maxv = nexts
                        b[n][0], b[n][1] = i, j
                        if maxv > alpha:
                            for [i, j] in change:
                                if [i, j] == change[0]:
                                    a[i][j] = X
                                else:
                                    a[i][j] = opturn
                            legalPos = []
                            getLegalPos(a, opturn, turn)
                            return maxv

                elif (deepth%2==1 and n%2==0)or(deepth%2==0 and n%2==1):
                    nexts = search(n+1, deepth, minv, maxv, nextlegal,\
                    opturn, turn)
                    if nexts <= minv:
                        minv = nexts
                        b[n][0], b[n][1] = i, j
                        if minv < beta:
                            for [i, j] in change:
                                if [i, j] == change[0]:
                                    a[i][j] = X
                                else:
                                    a[i][j] = opturn
                            legalPos = []
                            getLegalPos(a, opturn, turn)
                            return minv

            for [i, j] in change:
                if [i, j] == change[0]:
                    a[i][j] = X
                else:
                    a[i][j] = opturn

    if check:
        nextlegal = getLegalPos(a, turn, opturn)
        if n < deepth:
            return search(n+1, deepth, 1000000, -1000000, nextlegal, opturn, turn)
        else:
            if stepnum > 55:
                return eval2(a, opturn)
            else:
                return eval1(a, opturn, turn)
    if (deepth%2==1 and n%2==1)or(deepth%2==0 and n%2==0):
        return maxv
    else:
        return minv
```

13.2.3程序限制

Limafinal 在有可能被 KO 的情况下估值函数不完善会造成返回的落子点与我们想像的不一样，换言，在有可能被对方 KO 的情况下我方程序可能会帮助对方 KO 自己。原因如下：估值函数中行动力权值很大，当双方都无子可下时，对面行动力为 0，我方估值可能会很大，甚至比我方没有被 KO 的情况下的估值还大，所以我方 AI 会选择被 KO，这是我方 AI 在对战 KILO 时出现的情况，但实际我方的搜索能力不会比对方差。可行的解决方式是改变一下估值函数，在估值中加一个特殊判定，当我方子数为 0 时返回估值无穷小，这样 AI 就会极力避免被 KO 的情况，应该也就不会出现 LIMA 会被弱于自己的 KILO 给 KO 的情况。

13.3实验结果

13.3.1实验数据

实验环境说明：

- 硬件配置：Intel CORE i7
- 操作系统：（名称/版本）Window7
- Python 版本：（版本号）Python 2.7.6 Shell（便携版）
- 测试人员：汪建峰
- 测试地点：实验室的微机

测试结果分析见 3.2 部分。

1000 次对弈结果数据汇总

黑 / 白	Idiot 算法	Gambler 算法	T_LimaFinal

Idiot 算法	（胜负局数） （平均总分） （平均时比）	——	0 ： 500 6 ： 57 0.017s ： 75.90s
gambler 算法	——	——	26 ： 474 7 ： 50 0.016s ： 75.97s
T_LimaFinal	500 ： 0 61 ： 1 52.67s ： 0.01s	433 ： 67 50 ： 9 104.52s : 0.015s	500 ： 0 43 ： 21 98.78s : 105.34s

13.3.2结果分析

从 450（T_Limafinal 执黑）+450（T_gambler）盘的仿真对局数据来看，当 T_Limafinal 先行 时，获胜的局数为 390/450（胜率约 87%），仔细观察数据中的 TimeCost 项，可以发现我们的 AI 往往输在超时（Timeout）上，可以发现尽管优化之后的 AI 很大地节省了搜索效率，但是 由于 5 层的深度优先搜索复杂度确实非常高（因为是对棋盘状态搜索，而非单个数据的搜索 ，每一个节点都要考虑 64 个点），从实验上看来，还是挺容易超时的。只是，在比赛的规则 流程下，如果有哪只队伍想要凭借 T_gambler 算法实现获胜，那么这个概率将低至（1/100） ^2，即万分之一，在淘汰赛制下 T_Limafinal 算法需要与该队伍进行约 10000 次比赛，才能够 期望出现一次失败的可能。可见，想要通过随机的算法打败我们精心设计的深度优先搜索、搜索树剪枝、数据结构优化的强力算法基本上是没有可能的。

而（T_gambler）执黑时，我们的 AI 的胜率为 426/450，结合我们的决赛的经历来看，黑白棋 确实有着某种后发优势，后手的一方由于采取守势，极力限制对方的行动力，反而不容易超 时，先手的一方由于极力扩大自己的行动力，所以棋盘上能着子的点将越来越多，在搜索深 度相同的情况下、以及 160s 的限制时间里，也就更容易超出时间限制。

优势面：后手的棋局子数比为 7:50，先手的棋局子数比为 50:9，这也印证了上面的“后手在 时间限制

情况下有优势”。我们的AI基本占据了棋盘中的绝大多数的位置，最终很少进入终 盘，结束一般以T_gambler无棋可下而结束。每一次对局耗费的平均时间为90.25s（先手： 104.52s，后手：75.97s），而T_gambler的耗时基本与棋局无关，耗费时间为0.015s。从时 间的角度来看，100s左右的时间非常合理地利用了比赛中“全局用时不超过160s”这一条规 则，同时又不容易超时（在实际比赛中留有一定裕量，可以用于算法调整）。在比赛中，通 过简单地调整参数而不是调整程序结构，使得我们的程序能够根据对手算法的统计特征安排 算法，使我们保持了较高的胜率，因此战无不胜！

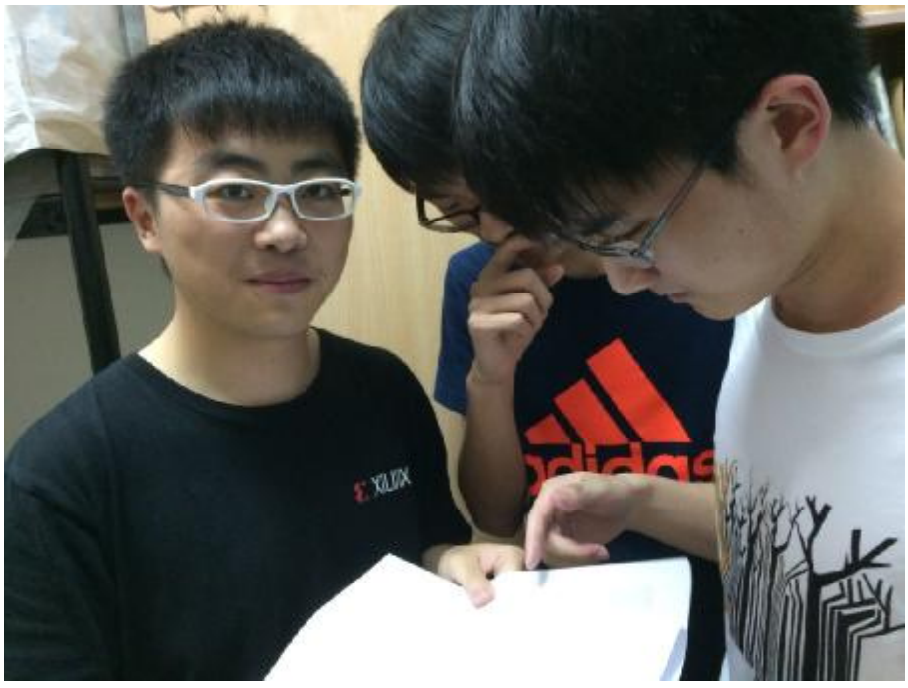
13.3.3经典棋局

与Kilo交战被KO，算是经典棋局。原因分析见上面所述，数据记录见Kilo组实验报告。

13.4实习过程总结

13.4.1分工与合作

本次大作业耗费了近三周的时间进行编写、调试与改进，虽然之前对于黑白棋算法有一定了解，但是在进行python编译时还是出现了很多问题。由于组员之间住的比较近，所以讨论次数也很频繁。每晚均有所讨论、商量算法和比赛的相关事宜，具体地点就是在寝室碰面交谈。内容包罗万象，比如，我们的算法与哪个队之间有优势或者劣势，需要用哪个版本的代码来面对不同的对手，算法上哪些需要增进和修改等等。于润泽同学主要负责报告撰写，汪建峰同学主要负责代码的测试和DEBUG以及一些非常重要的工作，蒋明轩同学主要负责设计算法以及代码调试。合作过程非常愉快。



以上两图为组会记录拍照。

13.4.2经验与教训

本次大作业的代码经过十次调整，最终得到了约十个版本。在历经不同版本的过程中，对于算法与架构有了更加深刻的了解，同时对于算法与耗时也进行了优化。从最开始的搜索三层到最终的搜索五层，耗

时也从最开始的超时到最后不超过 100s 的版本。

缺陷：Limafinal 在有可能被 KO 的情况下估值函数不完善会造成返回的落子点与我们想像的不一样，换言，在有可能被对方 KO 的情况下我方程序可能会帮助对方 KO 自己。原因如下：估值函数中行动力权值很大，当双方都无子可下时，对面行动力为 0，我方估值可能会很大，甚至比我方没有被 KO 的情况下的估值还大，所以我方 AI 会选择被 KO，这是我方 AI 在对战 KILO 时出现的情况，但实际我方的搜索能力不会比对方差。

可行的解决方式是改变一下估值函数，在估值中加一个特殊判定，当我方子数为 0 时返回估值无穷小，这样 AI 就会极力避免被 KO 的情况，应该也就不会出现 LIMA 会被弱于自己的 KILO 给 KO 的情况。

13.4.3建议与设想

虽然这次实习作业是第一次实行，但是我认为这是我所做过最棒的代码类实习作业。首先，他考验了团队合作能力，在进行分工合作时要事先分配好工作重点以及根据组员所擅长的部分进行调兵遣将。其次，团队之间的比赛竞争更为出彩。将各大战队分为四个版权，最终进行淘汰赛和半决赛、决赛，整个环节紧张刺激、扣人心弦。这种风格、形式的大作业值得保留。

接下来讨论一下需要改进的地方：

1. 根据各组的情况来看，比较强力的组基本都走到了极大极小搜索、剪枝之一步骤，搜索的层数一般在 4~5 层之间，实力的差别可能就是对于估值函数的理解不同，以及是否考虑行动力、稳定子和行动力的权值。

   通过比赛我们看到，由于我们的搜索层数为 5 层，那么多搜索一层的优势是不言而喻的，可以说在不超时的情况下，搜索 4 层的程序很难胜过搜索 5 层的程序。接下来程序的优化主要在剪枝优化、搜索方法改进、估值函数改进之上。
2. 通过阅读文献，我们了解到尽管 α - β 剪枝已经极大地提高了程序的搜索效率，并且使搜索的层数增加了 1~2 层，然而这种剪枝技术由于顺序地访问棋盘上可以着子的每一个点，所以能否在搜索的一开始就剪掉一枝是改进算法性能优劣之分。通过结合历史表和置换表的 α - β 剪枝技术，我们可以先对所有需要搜索的树枝进行排序，引入历史表机制，保证每次搜索时先搜索**非常好的节点**，再搜索主要步骤节点，最后搜索非常差的节点，这样发生剪枝的情况就大大增加了，也就使得算法的性能更加稳定、高效。
3. 估值函数的优化，利用机器学习的理论，使程序算法能够自行优化估值函数，然后通过对某一固定强力算法进行仿真模拟，不断进化算法（自动优化），一般跑到一两天之后，程序的估值函数已经能够优化地非常好了，一般的 AI 计数都要经历这一个强化学习的过程，实践证明这也是最强大的估值函数产生的方法。

实验中由于时间限制、没有强力程序代码作为陪练，所以并没有执行这一基本 AI 优化策略，而是人为测试估值函数的优劣，尽管比强化学习算法更为灵活和直观，但是改进效率较低，且比较费时。

13.5结论

本文以《数据结构与算法》课程实习作业——黑白棋竞赛为主题，以算法实现为线索，逐步还原了 LIMA 小组成员实现他们的冠军 AI 的过程与方法。其中详细介绍了黑白棋的博弈树搜索技术，先后介绍了极大极小搜索、AlphaBeta 搜索以及剪枝优化，通过试验和分析证明，采用 AlphaBeta 搜索使得搜索效率大大提高。另外，在决赛的前夕，小组成员汪星人和蒋明轩同学通过优化程序结构，使用简单的二维 List 处理棋盘而不使用老师提供的 ChessBoard 类，使得程序的执行效率提高了 5~10 倍，搜索层数直接从 4 层上升到了 5 层，我想这也是 LIMA 小组取得最后胜利的关键所在。

13.6致谢

感谢所有本组所有成员：汪星人、蒋明轩以及于润泽同学的辛勤努力和汗水，以及感谢他们的室友积极参加 LIMA 组代码的测试工作，强烈建议给其加分，其次还要感谢党，感谢国家，感谢另一半，感谢陈斌老师和所有的数算课程助教，他们在算法的改进过程中提出了很多宝贵的意见，最后感谢败给我们的所有组，是你们让我们成长起来。

13.7参考文

1. 柏爱俊，《几种智能算法在黑白棋程序中的应用》，2007 年 10 月
2. 维基百科
3. 谷歌学术“黑白棋”相关内容
4. 新浪博客，《极大极小搜索方法.负值最大算法和阿尔法贝塔搜索方法》，http://www.cnblogs.com/pangxiaodong/archive/2011/05/26/2058864.html

南区 South　BRAVO 组

14　数据结构与算法课程实习作业报告

（**BRAVO**：黄佳旺*、徐世宇、吕世极、韩甲源、陈跃毅、曹越）

摘要：本组共开发了两套黑白棋的算法，主要开发者分别是徐世宇（以下用 xu 表示）和韩甲源（以下用 han 表示）同学。算法中主要实现了对当前棋盘上的每一个可下点的分析，计算相应参数，如稳定子个数，行动力大小等，再进行估值排序，选择最佳落子点。主要使用的数据结构是树，算法以递归、搜索、排序为主。1000 次对弈结果显示，本组的算法能在与 idiot 和 gambler 对弈过程中占据明显优势，韩甲源的算法优势更大一些.

关键字：树、递归、搜索、排序

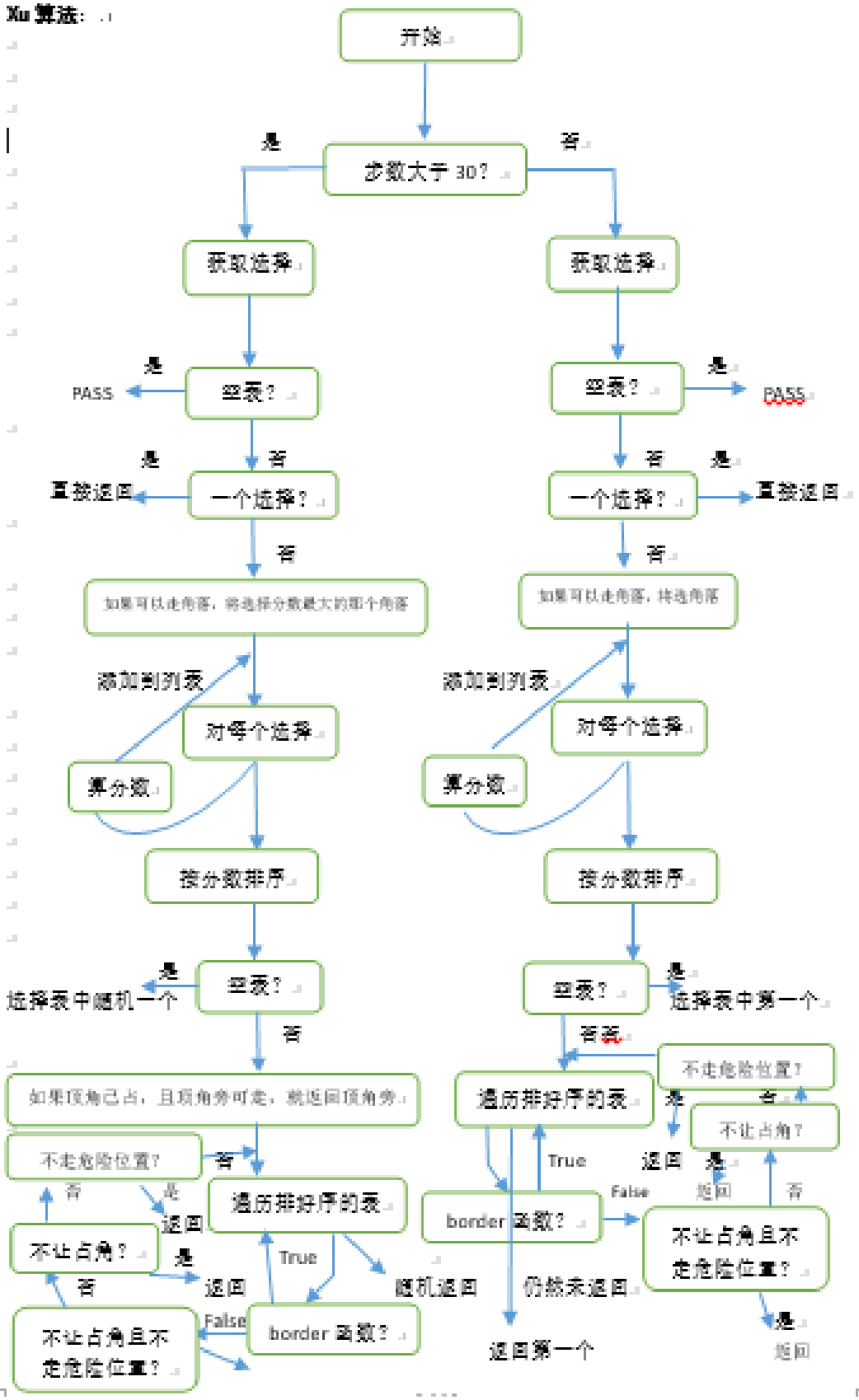
14.1算法思想

14.1.1总体思路

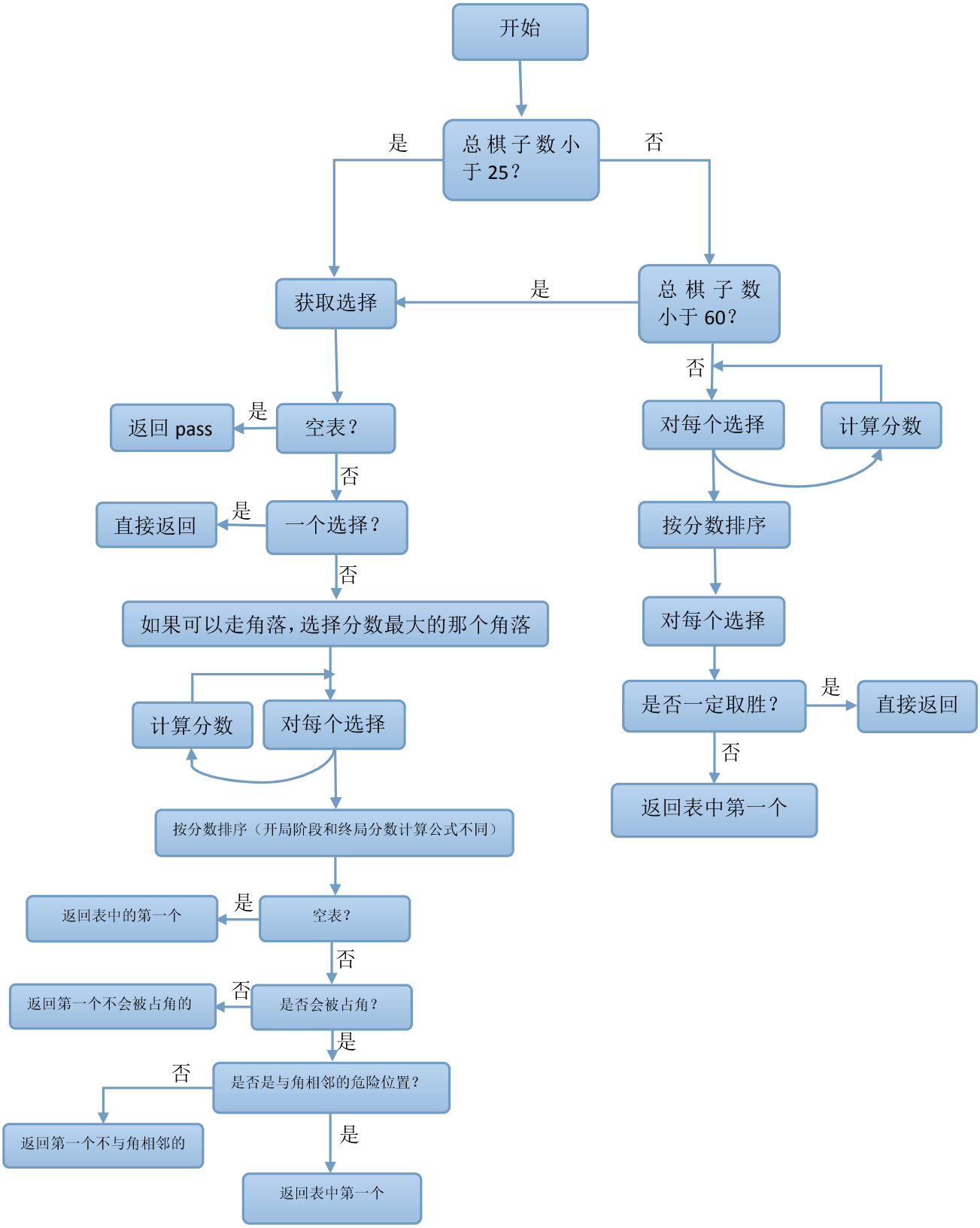
根据棋盘上每个位置的重要性，对每个位置进行赋值产生估值表，尝试每一个落子点，通过递归搜索遍历数步之后的棋局，以估值表、行动力、稳定子数目为参数，计算在每一个落子点落子后棋盘所对应的分值，尽量选择己方分数高、对方分数低的位置落子。同时考虑对方能否占角、是否下在危险位置等因素。**主要使用的数据结构是树，算法以递归、搜索、排序为主。**

14.1.2算法流程图

Xu 算法：


开始
步数大于 30？
是
否
获取选择
获取选择
空表？
是
PASS
空表？
是
PASS
一个选择？
是
直接返回
否
一个选择？
否
是
直接返回
如果可以走角落，将选择分数最大的那个角落
如果可以走角落，将选角落
添加到列表
对每个选择
算分数
添加到列表
对每个选择
算分数
按分数排序
按分数排序
空表？
是
选择表中随机一个
否
空表？
是
选择表中第一个
否
如果顶角已占，且顶角旁可走，就返回顶角旁
不走危险位置？
否
是
返回
遍历排好序的表
不让占角？
是
返回
否
True
随机返回
不让占角且不走危险位置？
False
border 函数？
遍历排好序的表
True
False
border 函数？
不走危险位置？
否
是
返回
不让占角？
是
返回
否
不让占角且不走危险位置？
是
返回
仍然未返回
返回第一个

Han 算法


开始
总棋子数小于25？
是
否
获取选择
是
总棋子数小于60？
否
返回 pass
是
空表？
否
对每个选择
计算分数
直接返回
是
一个选择？
否
按分数排序
如果可以走角落，选择分数最大的那个角落
对每个选择
计算分数
对每个选择
是否一定取胜？
是
直接返回
否
返回表中第一个
按分数排序（开局阶段和终局分数计算公式不同）
返回表中的第一个
是
空表？
否
返回第一个不会被占角的
否
是否会被占角？
是
否
是否是与角相邻的危险位置？
返回第一个不与角相邻的
是
返回表中第一个

14.1.3算法运行时间复杂度分析

徐世宇：假设每一步有 n 个选择，先对四个角落的判断复杂度是常数，遍历每个选择的复杂度为 O(n)，下一步后找稳定子的数量可以认为 O(n^3),遍历对方选择为 O(n),排序为 O(nlogn),还有就是递归算分数，大概是 O(n^2logn), 合起来是 O(n*[n^3+n*(nlogn+n^2logn)])=O(n^4logn), 之后的操作复杂度均不及这个，canhegetcorner 函数大概是 O(n^4),所以总的复杂度一般认为是 O(n^4logn)
韩甲源：假设平均每一步有 n 个选择，先对四个角落的判断复杂度是 O(n^2)，遍历每个选择的复杂度为 O(n)，canhegetcorner 函数大概是 O(n^4)，下一步后找稳定子的数量可以认为 O(n^3)，遍历对方选择为 O(n)，排序为 O(nlogn)，alphabeta 搜索复杂度 O(n^3)，总的时间复杂度为 O(n^2+n*(n^3+n+nlogn+n^3))=O(n^4)

14.2程序代码说明

14.2.1数据结构说明

在我们所编函数中，canhegetcorner、alphabeta 和 tryeveryway 三个函数都是可以递归的，这三个函数都属于树的数据结构，并且使用了深度优先搜索。由于深度优先搜索的时间复杂度是指数级的，因此这三个函数都采用了剪枝以降低时间复杂度，具体的实现见函数说明。

14.2.2函数说明

1. findtrastab(now)：

此函数的功能是找到当前的稳定子数目（传统意义上的），读入的参数是当前棋盘 now
在读入棋盘以后，先把棋盘复制，然后生成一个 8*8 的列表 zone 用于记录棋子的位置，其中本方的棋子记为 0，敌方的棋子记为 1，本方可以走的位置记为 2，空的位置记为 3，然后打算用 mine 变量记录本方稳定子数量，用 yours 变量记录对方稳定子数量。

之后对棋盘中每个位置进行遍历，如果是角落里的棋子，直接是稳定子，否则就对上下，左右，左斜，右斜四个方向进行探索，看每个方向是否安全，如果四个方向都安全，那么就是稳定子了。是否安全就是看这个方向上是否都是我方棋子，或者是否两侧都是对方棋子。最后返回的是我方的稳定子数量。

2. canhegetcorner(choice, nowcb, n=3)：

这个函数的功能是读取当前的棋盘以及一种选择，探索对方是否会在这一种选择后占到角，默认的探索深度为 3，采用递归算法。对面不能占角就返回 False，对面能占角就返回 True。

递归深度是 1 的时候，将要结束这个递归，此时只要选择 choice 下了之后对面不能占角，就返回 False，对面能占角就返回 True。

递归深度大于 1 的时候，我们先把 choice 下了，这时我们遍历对方的每一个选择，我们要求的是对方任选一个选择，我们随之而来总能有一种选择，使得对方接下去不能占角，这时就调用了本函数自身（递归深度减去 1），

```
for j in list_:
    a=a and canhegetcorner(j,temp_,n-1)
```

就如上面两句所示。只有当对面任选一个选择我们都有办法找到一个选择让对面接下去不能占角我们

才可以认为对面真的无法占到角落，此时返回 False，另外的情况返回 True

3. bestscore(choice, nowcb, n=3)：

这是个递归函数，读取当前的棋盘状况以及选择，看最好的分数会是多少，递归深度默认为 3。

递归深度是 1 的时候，将要结束这个递归，此时只要选择 choice 下了之后计算敌我双方得分，然后相减返回即可。

递归深度大于 1 的时候，我们先把 choice 下了，这时我们遍历对方的每一个选择。我们作这样合理的假设，即对方总是选择使得我方分数最小的那个选择，我们把对方的那个选择也下了，再遍历自己的选择，当然我们将选择使得自己分数最大的那个选择，这里调用了函数自身。最终返回的是那个选择下了之后的双方分数差。

4. border(m, now)：

这个函数用于判断 m 选择是否会在边界产生危险。返回 True 表示危险，False 表示安全。读入的参数还有当前的棋盘 now。

原理就是如果这个 m 选择的旁边相邻位置有一个对方棋子存在，那么就是危险的，但是如果两边相邻位置都有对方棋子存在，则认为是安全的，这个函数作用于四条边界的选择，而且是边界的中心的四个选择。如 A3，A4，A5，A6，共计类似的 16 个位置。

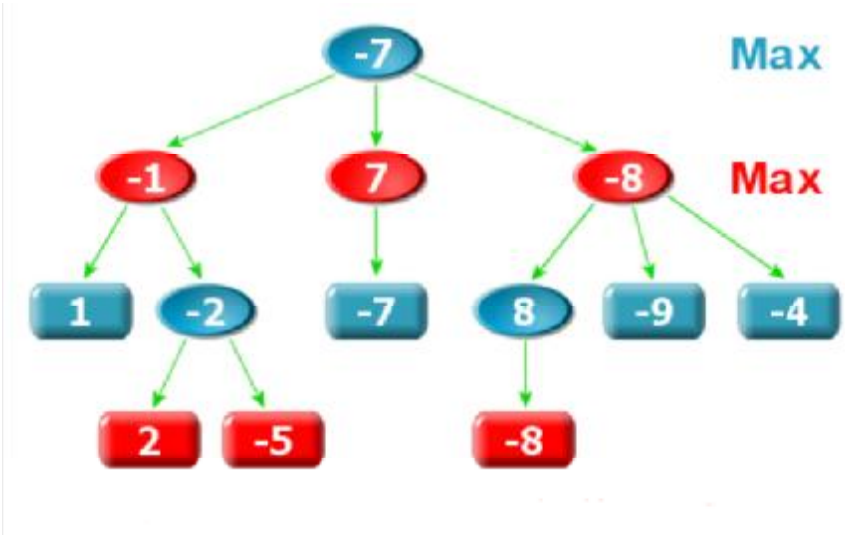
5. movementtransfer(m)：

这是一个将行动力转化为分值的函数，关于行动力的计算将在主函数里展开。这个函数的功能就是把行动力 0 转化为分数 0，行动力 1 转化为分数 5，行动力 2 转化为分数 15，行动力 3 转化为分数 20，大于等于 4 的行动力转化为分数 22。

6. alphabeta（nowcb, alpha, beta, l, he, n=3）：

在下棋的过程中，我方所走步骤必然希望获得最大的估值，由于不清楚对方的下棋方法，只能假设对方所选步骤总是使我方估值最小。假设此时轮到我方下棋，当我方只向前搜索一步时，必定在所有可行步中寻找估值最大的节点 S1，最大估值为 V(S1)，而当我方向前搜索两步时，S1 节点派生的子节点的最小值 $V(S1^{'})$小于其他节点派生的子节点中的最小值 $V(S2^{'})$，不妨设 S2 为同层节点中所派生的子节点的最小值中最大的节点，那么他转而应该选择 S2 对应的棋步作为自己的最佳棋步。随着搜索的加深，选择会越来越明智。

在具体实现中，由于我们将我方棋盘棋子值减去对方棋盘棋子值作为评估棋步好坏的标准，因此对方下棋时，所获得的估值乘-1 即为我方当前估值。如果在每一层递归时都将返回的估值乘-1，那么每一层递归时我们只要将当前所有选择取最大值，就可以保证所返回的结果是最优结果。具体实例如下：



当然，在搜索的过程中，某些节点搜索后就不用再搜索同层的其他节点了。比如在上图中，在搜索第二层第三个节点的过程中，由于前两个节点搜索结果的最大值为 7，所以第三个节点搜索最大值大于 7 才是最好的节点。在搜索第三个节点的过程中，由于其第一个子节点为 8，乘-1 后是-8 比 7 小，那么第三个节点的估值一定小于等于-8，继续搜索已经没有意义，也即发生了剪枝。

算法实现时为每个节点引入两个值：alpha 和 beta，分别表示该节点估值的允许下限和上限，初始时，alpha 应为 - ∞，beta 应为 + ∞，由于估值表的估值结果有限，我们将 alpha 和 beta 的初始值设为 - 1000 和 + 1000。其他节点的 alpha，beta 值是深度优先搜索过程中从他的父节点继承过来的。当 beta 小于等于 alpha 时，就会发生剪枝。

函数输入项有当前棋盘 nowcb，节点估值允许的下限 alpha 和上限 beta，我方棋子和对方棋子 I，he，递归深度 n，由评估函数 PositionValue 对当前局面进行估值，返回值为当前局面的估值。

7. tryeveryway(cb,totalscore)：

在棋局的最后几步，由于棋步往往较少，所以可以通过搜索判断走哪一步棋可以保证一定获胜。如果在我方走某一步后，不论对方如何走棋，我方总有方法获胜，那么走这一步就一定可以获胜。

在具体实现过程中，除了上述思路外，还增加了剪枝以提高搜索效率。当我方搜索时，如果我方的某一棋步返回值为‘win’，代表我方走那一步一定可以获胜，所以不再向后搜索，返回这一步即可保证胜利。如果所有棋步都没有返回‘win’，则我方没有必胜的把握，返回‘lose’。在对方搜索时，由于不清楚对方棋力，只能假设对方所走棋步是防止我方赢棋的棋步。所以在搜索过程中，如果对方的某一棋步搜索的返回值为‘lose’，则对方一定不会走其他使我方必胜的棋步，即我方没有必胜的把握，应返回‘lose’，不必再进行后面的搜索；如果对方所走每一步棋我方都有必胜的策略，即对方每一步棋搜索后返回值都为‘win’，此时我方一定可以获胜，应返回‘win’。

函数的输入项有当前棋盘 cb 和当前总棋子数 totalscore，我方棋子 I，判断是否有 pass 的变量 isPass 和是否双方 pass 的变量 isBothPass 作为全局变量引入函数。

14.2.3程序限制

本程序几乎在任何情况下都不会出现 bug，但是在特定条件下我们的程序表现得十分低效。

由于本程序采用了 canhegetcorner 函数，而此函数的功能是探索对方是否能占角，如果自己的某一种选择让对方有机会占角则弃之不用，这就导致了一个问题，就是对方如果已经能够占角，却故意没有把那个角占领（因为他知道那个角已经肯定属于他了，再迟点占也没事），那么这个时候我们的每一种选择都被 canhegetcorner 函数认为是无效的，这样就没有选择可供分析了，所以这个函数的引入有失妥当。我们的解决方法显得很粗糙，仅仅是不走那个十二危险位置而已（所谓的与角落相邻的十二个位置），总的来说，出现上述情况时我们的程序的表现十分低效，几乎是一个 gambler 的形象。

第二点是一旦对方率先占领了某个角落，那么本程序没有任何的方法限制对方接下来的方法，事实上我们也无法想出行之有效的方法来遏制对方，依旧是按照老的估值表来选择合适的位置，这也是低效的体现。

还有一点就是我们并不知道估值表数值的合理性，只做了粗糙的测试与调整，我们觉得估值表的数值还有待更精细的调整。

14.3实验结果

14.3.1实验数据

实验环境说明：

- 硬件配置：Intel(R)Core(TM)i5-4260U CPU @ 1.40GHz 2.00GHz
- 操作系统：Windows 7
- Python 版本：2.7.9

1000 次对弈结果数据汇总

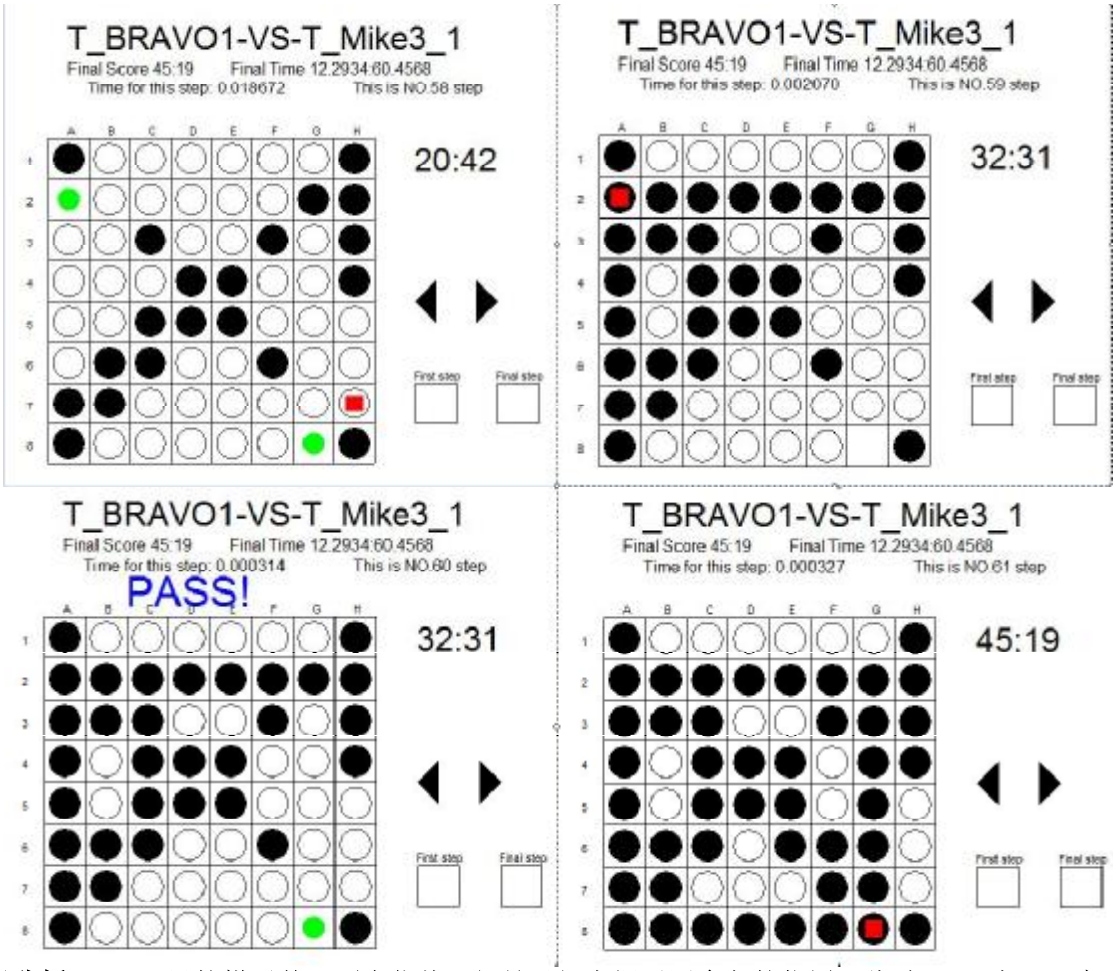
黑 / 白	idiot 算法	gambler 算法	Han 算法	xu 算法
idiot 算法	胜局比：0:1000 总分比：27000:37000 时间比： 10.0618:13.8189	胜局比：421:579 总分比：28970:34761 时间比： 11.0932:13.7606	胜局比：1000:0 总分比： 63000:1000 时间比： 55491.9501:13.3097	胜局比：1000:0 总分比：38000:26000 时间比： 35751.9620:14.8336
gambler 算法	胜局比：524:476 总分比：32920:30574 时间比： 13.5822:11.0457	胜局比：454:546 总分比：31232:32664 时间比： 13.103056:13.232243	胜局比：912:88 总分比：44098:19868 时间比： 58463.0923:12.7881	胜局比：902:98 总分比： 46843:17157 时间比： 28123.5814:16.6072
Han 算法	胜局比：0:1000 总分比：23000:41000 时间比： 14.9806:48204.2788	胜局比：67:933 总分比：13428:50572 时间比： 13.3554:51384.4679	胜局比：0:1000 总分比：31000:33000 时间比： 61647.9318:60503.6576	胜局比：553:447 总分比：32284:31716 时间比： 40133.2431:49876.8762
xu 算法	胜局比：0:1000 总分比：23000:41000 时间比： 12.3008:54922.4262	胜局比：72:928 总分比：20184:43816 时间比： 12.6222:45651.2211	胜局比：384:616 总分比：26404:37596 时间比： 52507.0928:40735.9395	胜局比：598:402 总分比：33275:30725 时间比 41733.7350:53089.3005

14.3.2结果分析

本组的算法在与 idiot/gambler 对弈过程中,能有效占领角落等关键位置，对对方行动力的控制也起到了作用，总体效果还行。运行时间大概在预期之中，没有发生超时。时间开销主要发生在递归搜索部分，随递归深度增加指数上升。

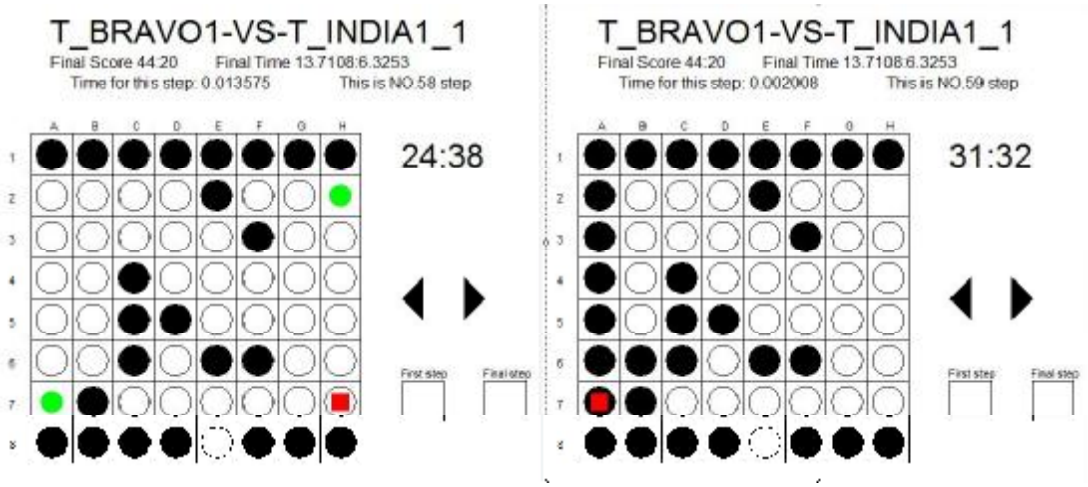
14.3.3经典棋局（可选）

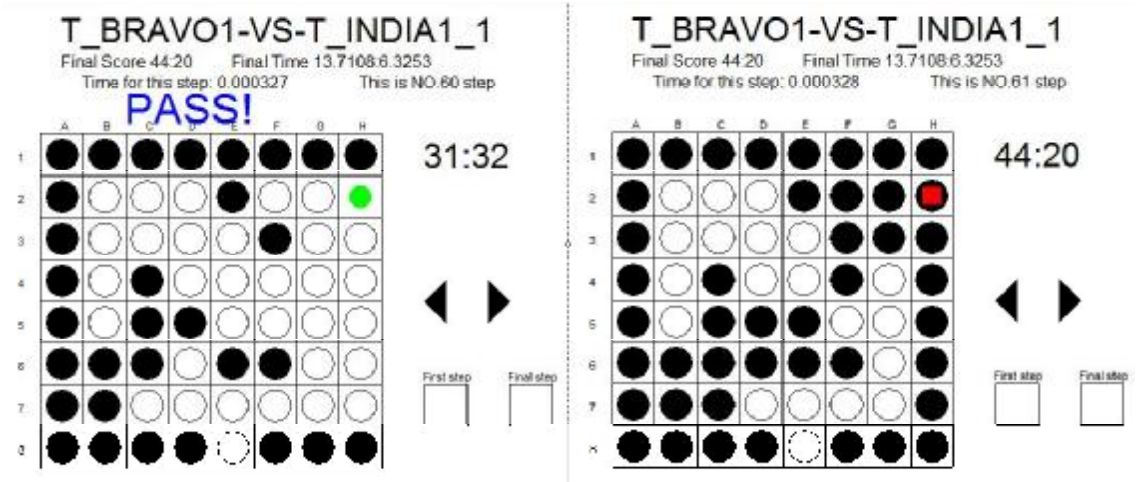
1. 与 Mike 的对决



棋局分析：bravo 虽然棋子数目不占优势，但是已经占据了四个角的位置。此时 Mike 先下一步，保住了 H 边上的 3 个子，此时 brovo 有 A2 何 G8 两种选择，两种选择都对 brovo 十分有利，因为不管选择哪个位置，都会吃掉对方好几个子并且造成 pass 局面，当 brovo 下完两个位置后，局势已经彻底翻转，brovo 赢得了胜利。

2. 与亚军 INDIA 的对决





棋局分析：与上局相似，brovo 已经占据了四个角，如果 brovo 选择 H2，India 会选择 A7，brovo 将以 39：25 获胜。而 brovo 选择 A7，造成 pass 局面，再下 H2，最终以 44：20 获胜。

14.4实习过程总结

14.4.1分工与合作

第一次组会：6 月 3 日，文史楼 205

会议内容：

1.试玩黑白棋游戏，说明需要完成的任务。

2.要求每人先看懂老师的基础设置代码，尽量提高自己的棋力，并检索文献，寻找算法。

图一：第一次组会记录

第二次组会：6 月 5 日，理教 414

会议内容：

1.说明比赛时间、报告时间以及分组结果。

2.强调主要工作是算法设计和实习报告。

3.算法的来源不重要，关键是理解算法的思想和对数据结构的使用和理解。

4.竞赛的分差不大，不要太在意最后的结果，关键是要写好报告。

5.分工：吕世极编写估值表、墙的判断函数

韩甲源负责最后 15 步的策略实现

徐世宇负责棋表的输入与判断，接入吕世极的代码

黄佳旺将代码组合起来
陈跃毅编写测试程序
曹越负责联系信科同学，寻求前人经验

图二：第二次组会记录

第三次组会：6 月 8 日，文史楼 112

会议内容：
1.组长报告我们的进展：本组的代码已打败电脑中级，然而在高级面前一败涂地
2.曹越找信科同学拿到了的黑白棋 C++代码作为借鉴，他们的算法在高级面前还没输的太惨，但却被吕世极轻松解决，在对局过程中可以看出该算法对稳定子的控制很好。
3.由徐世宇、黄佳旺、吕世极翻译，并吸取其中有益的部分。
4.由陈跃毅对代码进行测试，完成前期的报告，包括算法的说明分析等，但复杂度等需要徐世宇等人解释，其他人将阅读过的文献发给陈跃毅。

图三："这个算法好"——得信科前人代码有感

第四次组会：6 月 10 日，逸夫二楼 3425

会议内容：

1.信科的代码翻译结束，其思路简单，方法也简单，但是他的估值表比较准确，剪枝和稳定子做的不错。我们吸收其中好的部分，最终得到两个版本的代码：徐世宇版和韩甲源版

2.后期工作主要是：

（1）完成实验报告

（2）找组对抗并发现问题，改进算法。由黄佳旺约时间地点，韩甲源参赛，大概两到三场

（3）曹越测试各部分权重，找到最优配比。（通过自己和自己打）

图四：组长发表重要讲话

第五次组会：6 月 12 日，理教 405

会议内容：

1. 曹越和陈跃毅汇报工作进展及遇到的困难
2. 委派吕世极参加晚上的热身赛

图五：讨论激烈的会议现场

14.4.2经验与教训

在开发算法的过程中，我们做得比较好的一点是每次组会中分配给大家的任务大家都能在规定的时间内完成。当然这一点与我们将组会这件事常规化有很大的关系。每周一，周三的晚上以及周五的下午都是我们的组会时间。常规化的组会让大家能时常进行思想、方法上的交流以及反馈在算法实现过程中的问题。一个团队要能够进行高效的合作，交流是第一位的，当然仅仅有交流也不够，在团队的成员组成上也应该保持一定的平衡。

所谓组员的平衡是指的，我们需要的并不是全部都是思维很活跃，在解决问题方面能很快又很好的思路的同学。也不全是总是做深远的考虑，有一个想法就抓住不放，认真地去思考其是否真正可以实现的脚踏实地的，务实的人。我们需要的，是各个能力方面的能力都有至少一个人十分突出的团队。这样才能让这个团队的合作的意义达到最大。在我们组，徐世宇与吕世极都是数学非常好，逻辑思维很强的人，于是他们就负责算法的开发，研究黑白棋的特点并思考怎样使用计算机的计算能力实现最优的下棋算法。而韩甲源是一个理解能力超群且做事踏实认真的同学，因此他负责了将徐世宇，吕世极的算法思想实现成具体的程序的工作，同时他也发挥了他超常的理解能力，进行了大量文献的阅读，并自己将文献中的大部分代码实现了，使我们的算法有了更多样的色彩，能做更多维度的思考。作为组长的黄佳旺有着不错的组织与协调能力，在每次组会前了解好了各个部分的进展情况，在组会中进行总结，并分析现在的进度做好后阶

段的安排。同时他也有着不错的逻辑思维，负责了小部分代码方面的工作。曹越和陈跃毅也是两个做事非常认真可靠的人，在后阶段我们的程序的调试与测试过程中，陈跃毅负责了测试代码的实现以及让我们的算法与 idiot 与 gambler 对弈的实验。曹越则发挥了他交友广泛的优势联系了信科的同学，拿到了他们之前做黑白棋大作业时的一个实力很强的 C++的代码，同时他也负责了调试我们的程序的工作。由于我们的算法中有多个因素的考虑，这些因素之间必然有一个权重的不同，而曹越则负责不停地修改各部分的权重进行测试找到最优的一组权重。

尽管我们在前期的工作进行地有条不紊，我们算法的棋力进步迅速，但是我们的后期工作做得并不好。在比赛的前几天的热身赛中，我们并没有取得比较理想的战绩，但是由于组长黄佳旺和主要程序负责人韩甲源在比赛那天的晚上有通选的考试需要复习备考，另一位主要程序负责人徐世宇也需要准备不久后的考试。我们组并没有对热身赛的结果进行详细的分析并从中发现我们算法的漏洞，对之做出相应的改进。其实对热身赛结果的分析是极有益于我们的算法的棋力的提升的，但是我们却没有做这关键的一项工作。

14.4.3建议与设想

尽管比赛是残酷的，不可能每一个组都在竞赛中取得很好的成绩，但是这个过程给了我们一次成长，在合作工作过程中，比赛过程中我们享受了丰富的乐趣。但在进行这个问题的研究过程中我们发现，对这个问题的解决所需要使用的数据结构相对简单。许多我们这个学期学的很好的数据结构如果用在这个问题的解决上反而会出现时间复杂度增加等问题。所以我们不得不摒弃了各种复杂的数据结构而仅仅给我们的代码以算法思想的力量。所以如果在以后的大作业中如果能有与数据结构契合度更高的问题，那一定可以使大家在做大作业时真正用上本学期所学的数据结构的知识。

最后希望学弟学妹们好好享受这门课，数据结构与算法课不同于之前的计算概论，并不会以编程为主，希望大家不要惧怕，好好享受这个学习过程，它一定会给你们带来许多有趣的体验。

14.5致谢

感谢认真指导我们算法实现，关心我们组算法进度，还为大家写了基础设施代码的陈斌老师。

感谢甘愿奉献，自身水平又了得的两位助教以及你们的 ReversiChecker 帮助我们更好，更快地对对战结果进行分析。

感谢无私的阎述辰同学，利用海龟做的可视化程序给我们大家提供了对复盘数据分析的方便。

感谢信科的王鹏飞同学给我们提供了实力强劲的 C++代码作为参考使我们能对我们的算法做一定的修改，借鉴其中的比我们做得好的部分。

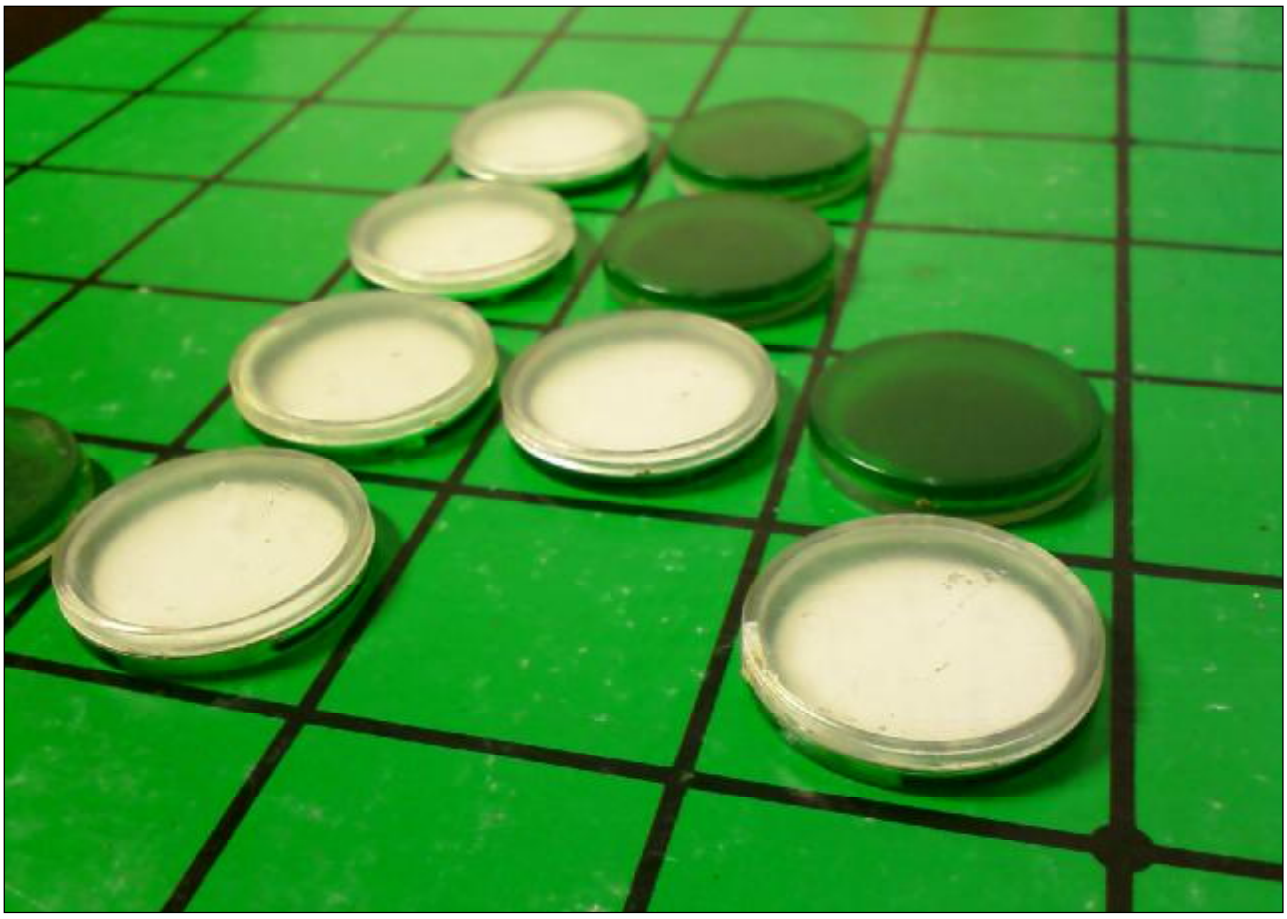
14.6参考文献

1. 《几种智能算法在黑白棋程序中的应用》 柏爱俊 0511@USTC 2007 年 10 月
2. 《基于Android系统的手机游戏黑白棋的设计与实现》 [期刊论文]《新乡学院学报（自然科学版）》-2011年3期 李林涛 朱珊虹
3. 《Othello:A Minute to Learn…A Life to Master》 Brian Rose
4. 《提高棋类程序人工智能水平的两种方法》 [期刊论文]《太原经济管理干部学院学报》 -2004年z1期 纪杰
5. 《博弈算法在黑白棋中的应用》 [期刊论文]《计算机技术与发展》 ISTIC -2007年1期 杜秀全 程家

北区 North　NOVEMBER 组

玄机——黑白之间

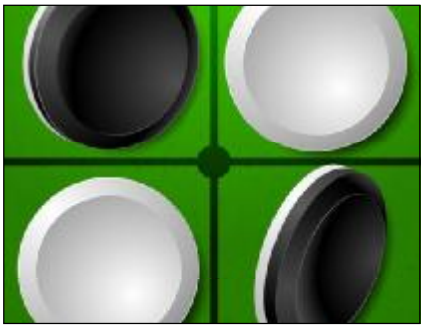
15　数据结构与算法课程实习作业报告



工作介绍

（一）黑白棋

黑白棋是一种棋类游戏，又叫反棋（Reversi）、奥赛罗棋（Othello）、苹果棋或翻转棋。这种棋在西方和日本非常流行。游戏通过互相翻转对方的棋子，最后以棋盘上面谁的棋子多来判断胜负。它的游戏规则非常简单，两个同色棋子之间的异色棋子将被翻转，变成对手的颜色，上手容易，但是变化却十分复杂，进行到尾盘，常常有翻盘情况出现。关于黑白棋常说：学会一分钟，精通一世功。（A minute to learn, a lifetime to master.）



（二）黑白棋程序

在 1980 年代的时候，电脑还并不普及，在黑白棋的世界里，人类棋手仍然是最强的。

到了 1990 年代，电脑的速度以几何级数增长，虽然写出来的黑白棋程序仍然有一些笨拙，但是凭借计算的深度和速度，以及对于尾盘时棋局变幻的准确估计，所以已经达到了很强的水平。其中最著名的就是 Thor，在当时是棋力最强的程序，比得上世界级的棋手。在这个时期的程序都是人工地加入一些诸如行动力、位置策略、偶数重要性等等的策略判断，而且又是直接写在程序里面，所以程序的棋力严重依赖程序员本人的棋力，所以很像人类棋手的下法，但是由于计算机计算速度和深度均能超过人类，对棋局的评估也更加精准，所以棋力往往能够超过编写程序的人自己。

到了 1994、1995 年，黑白棋程序的编写方法又有了突破性进展，首先是出现了一个称为 IOS 的网站，让不同的黑白棋程序在上面互相对局，促进了程序员之间的交流，同时期，Michael

Buro 做出了能够有程序自我学习的 Logistello，不少人开始进行模仿。从此，程序员不再将人工策略和下棋的方法等等死记在程序里，而是由程序自己去学习，通过实战积累好的开局和棋盘局势，并进行评分和保存，程序会因此变得越来越强。

由于 Logistello 在编写程序的方法上进行了根本的改进，一直保持着黑白棋程序的世界冠军位置。在 1997 年 8 月，Logistello 击败了 1996 年黑白棋的世界冠军村上健，从此，黑白棋程序开始把人类远远地抛在后面。

（三）作业介绍

本次的作业也是进行黑白棋程序的编写，但是限于小组成员自身的实力和条件等的限制，无法实现诸如程序学习等较为高级的方法，只能在程序中加入一些策略性的判断和评估。基于对棋盘局势的评估和自身目前可走步骤的一些判断和选择，来得出较为合理的下法。

本小组一共设计了两套代码，将在下面的代码分析中详细地介绍，算法的思路基本一致，实现方法有所区别。

本次作业成果包括两个类似的黑白棋算法程序，通过使用老师提供的公共设施代码的接口进行下棋的操作；十六份测试数据，格式为 txt，保存每一局对战双方的比分和耗时，每一次测试对战 1000 局；一份实验报告，包括对算法的介绍和分析，对于测试数据的统计分析等；一份测试代码，用于算法之间的对战，能够重复比赛 1000 次，记录每一局的用时和得分，并自动生成记录了这些数据的 txt 文件。

本作业由 NOVEMBER 小组秦树健、张虎来、刘嘉栋、韩露、杨润、刘杰 6 人共同完成，尤其感谢秦树健和刘嘉栋的辛勤付出，过程中克服了不少的问题，能够最终完成实属不易。

算法介绍

概述：

本算法采用了估值函数的方法和树的数据结构来实现，对于棋盘的一个局面，两边所有下法所造成的局面都可以放在一个树中，通过采用一个统一的评价函数对每一个局面进行评分，就能得到对自己最有利的一种下法，从而做出决策。而对于一些简单而明确的局面，我们则直接采用条件判断的方法给出下法。

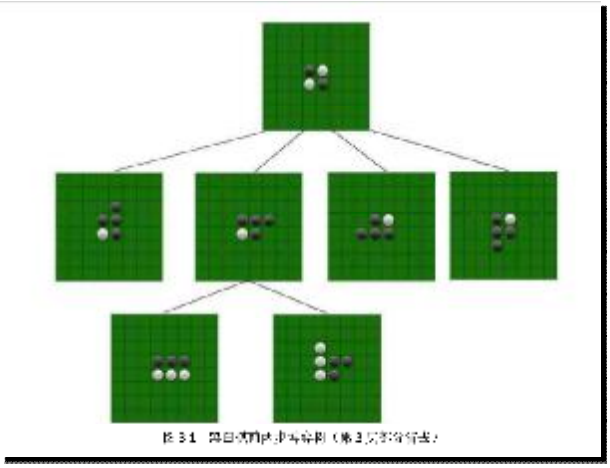
关键词：

估值函数、树、α-β搜索、最大最小值搜索、递归、条件判断

（一）算法思想

1.1-算法总体思路

黑白棋共有 64 个空白位置，状态空间为 64 的阶乘。黑方第一步下子时共有 4 个位置可供选择，以后轮流下子时双方均约有 15 个子以内的位置可供选择，到了中盘，即游戏快接近尾声的时候，双方可供选择的下子位置数随着空白位置的减少而急剧减少，游戏最少越 10 步左右结束，最多 60 步结束，所以黑白棋的博弈树对应的状态空间应远小于 64 的阶乘，约在 15 的 60 次方以内，确切的说，应小于：



4*2*15^46*5^10

=98,398,379,662,661,973,287,562,933,165,872,891,549,952, 328, 205,108,642,578,125,000。

但即便如此，这个天文数字几乎无法用目前的计算机穷尽遍历所有的局面。如此大的一颗树，计算机几乎不可能遍历当前结点的所有的子结点来"预测"游戏的结局画面从而选择最佳走法。

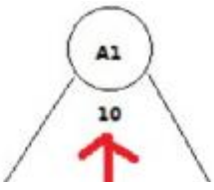
如图，黑白棋的开局黑子先行，共有有四种可能的走法，任何一种走法后白子共有 2 种走法，图中仅列出了黑子的第二种走法后白子的两种着法。如此按在遵守游戏规则的前提下一层一层的按顺序展开，便构成了黑白棋的状态空间树，也就是博弈树。博弈的核心思想实际上就是对博弈树节点的估值过程和对博弈树搜索过程的结合。

博弈程序的任务是对博弈树进行搜索找出当前最优的一步棋。解决黑白棋的走法产生问题的过程是设计一个合适的估值函数对状态空间的结点进行估值，对状态空间树进行一定深度的搜索，当搜索达到一个预定的层次时，利用一个局面评价函数对每个局面打分，然后向上回溯，计算当前局面父节点的估计值，然后继续向上回溯，一直到回到根节点，计算出根节点的估值为止。再根据极大极小原理选择一个估值最好的结点作为最佳着法结点。这种判断并不能得到绝对的最终会获胜的结点，只能在一定程度上保证该节点是最可能获胜的结点。

小组的第一个算法在搜索时，为了提高效率，小组采用剪枝策略将那些确定不可能赢的结点剔除，在较低的深度剔除一些几点能大大减少搜索的结点数。小组采用的剪枝策略是α-β 剪枝。

在极大极小搜索的过程中，存在着一定程度的数据冗余，剔除这些冗余数据，是减小搜索空间的必然做法，所采用的方法就是 1975 年 Monroe Newborn 提出的α-β剪枝。把 Alpha-Beta 剪枝应用到极大极小搜索或者负极大值搜索中，就形成了α-β搜索。

由于博弈树的“与”结点与“或”结点逐层交替出现，在生成新节点的同时，计算该结点的估值并回溯计算出其父节点的倒推值，便有可能删除一些不必要的结点。对于一个“与”结点来说，它取得当前子结点中的最小倒推值作为它倒推值的上届，称此值为β值对于一个“或” 结点来说，它取得当前子结点中的最大倒推值作为它倒推值的下届，称此值为α值。 α-β剪枝的一般规律为，任何“或”结点 x 的α值如果不能降低其父节点的β值，则对结点 x 及以下的分支结点可停止搜索，并使 x 的倒推值为β，这种剪枝称为β剪枝任何“与”结点 x 的β值如果不能升高其父节点的α值，则对结点 x 及以下的分支结点可停止搜索，并使 x 的倒推值为α，这种剪枝称为α 剪枝。



第二个算法没有采用α-β剪枝，而是在计算出每一个节点对应的分值之后直接存入博弈树中，采用最大最小算法算出下法的得分。比如左图，标记为 A 的是自己的下法，标记为 B 的是对手的下法。假设对手是聪明的，能够使得他下出的每一步棋使算法得分最少，那么就在考虑对手时让他下出使算法得分最少的棋，也就是在博弈树中向下搜索的时候遇到对手下时选择分值最低的节点，而自己下时当然就选择最得分高的节点了。这样在从这一步出发向下搜索到自己认为合适的层数后就能够得到这一步的分数，这就是所谓的最大最小值搜索。

此外，在第一个算法中，虽然考虑到利用树的数据结构来实现α-β搜索算法，但是并没有编写相关的数据类型，主要的操作在一个具有递归的函数内部实现。而第二个算法则是设计编写了一个数据类型 MaxMinTree，用来存放进行最大最小值搜索时需要的数据——每一个棋盘盘面的得分。

1.2-算法流程图

(由于种种原因未能完成)

1.3-算法运行时间复杂度分析

NOVEMBER: us:

因为都是树的数据结构，此处的 a 是一个范围内的值，并不是一个具体的定值。

总：$O(a^n)$ 总：$O(a^n)$

估值函数：O(k) 估值函数：O(k)

（二）程序代码说明

2.1-数据结构说明

博弈程序中主要的数据结构就是树，本次小组作业的两个算法中前一个算法由于采用了递归函数的方法实现了α-β搜索算法，所以并没有定义相关的数据类型，在第二个算法中定义了用于最大最小值搜索的数据类型 MaxMinTree，现在加以介绍，具体的代码和注释放在了附

录中。

class MaxMinTree:自身具有如下几个数据项：

self.data：保存节点的数据；

self.children：保存节点的所有子节点，由于数量很多且不确定，所以用一个列表存；

self.level：表示该节点于整个树的第几层；

self.prev：表示该节点的父节点，初始值为 None；

self.tree：把整个树中的数据用嵌套列表的方式表示出来。

class MaxMinTree:具有如下几个方法：

self.insertNode(self, newnode)：在节点中插入新的子节点，参数为新节点的数据项，同时新节点的父节点设为当前节点，新节点的层数相对当前节点加 1；

self.getChild(self,n)：得到当前节点的子节点，参数为要得到的子节点在当前节点的所有子节点中的序号；

self.getRootVal(self)：得到当前节点的值；

self.setRootVal(self,obj)：设置当前节点的值，参数为要设的值；

self.getTotalLev(self)：得到当前节点往下树的高度；

self.MaxChild(self)：得到当前节点的值最大的子节点；

self.MinChild(self)：得到当前节点的值最小的子节点。

该数据类型从课程中的二叉树扩展而来，但是由于黑白棋的可能性十分多样，采用二叉树不能很好的模拟博弈的过程，所以采用多子节点的树，分支数可以任意扩展。同时为了便于处理和进行最大最小值搜索，给了这个树一些特殊的方法，比如找出最大最小的子节点，将整个树用列表的方式表达，便于用后续的函数进行处理。

2.2-函数说明

因为本组设计了两种类似的算法，将分别陈述两个函数中含有的函数。

算法一：采用了α-β搜索和α-β剪枝，命名为 T_NOVEMBER。

（1）函数 CheckValue(board,color,opcolor):

用于对棋盘进行估值的函数，参数有三个，分别为当前棋盘 board（二重列表形式），自己的颜色 color，对手的颜色 opcolor。可以通过双方棋子在棋盘上占有的位置、棋子的数目、棋子的行动力等等对棋盘局势进行评估，返回值为一个分数。

（2）函数 CheckPos(board,color,opcolor,PosDic):

用于探测当前棋盘上双方可以走棋的位置，参数为棋盘 board（二重列表形式），自己的颜色 color，对手的颜色 opcolor，用于存储可走位置的字典 PosDic。注意在函数内部的开始，会首先将字典 PosDic 清空，然后在棋盘上逐个探测可以走的位置，将存在的位置设为字典的键，该字典的键对应的值是下完这一步之后对手可以走的步骤，相当于对棋盘进行了深度为一的模拟。返回值为记录了信息的字典 PosDic。

（3）函数 BoardTransform(board):

函数用于将棋盘上的数据转化为便于其它函数处理的二重列表。参数只有一个，一个 ChessBoard()数据类型：board。返回值为包含了当前棋盘上各个位置是什么（黑子、白子或空格）的二重列表。

（4）函数 AlphaBetaSearch(depth,board,alpha,beta,color,opcolor,Compcolor):

用于进行α-β搜索的函数，参数有 7 个，分别为当前搜索的深度 depth（用于递归的重要参数），棋盘 board（与上面的函数一样是二重列表的形式），剪枝下限 alpha，用于与自己落子时候的得分进行比较，剪枝上限 beta，用于与对手落子时的得分进行比较，自己的颜色

color，对手的颜色 opcolor，用于比较的颜色 Compcolor。

在函数内部，首先用 CheckPos 函数得出记录当前可以落子的字典 PosDic，枚举 PosDic 里的所有元素，更新棋盘，计算出估值 score 接着先将刚才的更新撤销（回滚操作），再按如下法则剪枝：

如果当前是本方(即电脑)在落子，那么如果 score>alpha，更新估值下限 alpha 为 score；如果 depth==self.DepthMax(即到了树的根节点)，那么更新电脑的落子位置。如果 alpha>=beta(即此节点估值下限>=估值上限，不可能影响上层节点的估值)，那么直接退出枚举。

如果当前是对手在落子，那么如果 score<beta，更新估值上限 beta 为 score；如果 depth==self.DepthMax(即到了树的根节点)，那么更新电脑的落子位置。如果 alpha>=beta(即此节点估值下限>=估值上限，不可能影响上层节点的估值)，那么直接退出枚举。

在枚举结束后，如果当前是本方落子，返回估值下限 alpha(自己最少能得多少)，如果是对方落子，返回估值上限 beta（对手最多能得多少）。

此时的 CompMove[0]、CompMove[1]为当前落子位置。

（5）主要函数 play(cb,ms):

下棋算法的主要程序，也是算法中被下棋的公共设施代码直接调用的函数，具有统一的形式、参数和返回值，而上述的所有函数都是在此函数内部被调用。

参数有两个，当前棋盘 cb，是一个 ChessBoard 类的数据，ms 是一个函数可以利用的字典，其中 ms['log']是下棋的历史记录，供程序使用。（本算法并没有用到下棋的历史记录，但是包括这个参数是为了形式上符合公共设施代码的要求。）

在函数的内部，当初盘，也就是棋盘上总棋子数目小于 20 时，由于往后的局势变化相对还较大，所以采用"贪吃"的策略，使得每一步落子能够吃到对方最多的棋子，在前期取得一些

优势，由于棋盘上棋子总数少，对手布局也不很方便，所以采用“贪吃”策略被针对的可能性也较小。

到了棋盘上棋子总数较多的时候采用α-β搜索的算法，进行递归，调用到上面的 AlphaBetaSearch 函数。

除此之外，有一些先于下棋策略的判断，如当棋盘四个角能够占据时直接占据角而跳过其余的判断步骤，而四角周围的位置由于下了之后容易被对手占据边角，所以除非在无子可下的情况下，不下在这个位置。

总结起来该函数的构成就是两种下棋的策略加上一系列的判定，最后的输出结果为一个字符串形式的棋盘上的位置，表示算法选择的下棋位置（如'A2'、'B3'之类）。

算法二：采用了树的数据结构，并编写了树的特殊数据类型，命名为 T_us。

算法二中首先包含了一个自建的数据类型叫做 MaxMinTree，这个在前一个部分已经做过介绍，下面是算法中包含的函数。

（1）函数 scoreAdd(tree,cb,depth,me):

这个函数用于把推演过程中的每一个局面的分数通过 MaxMinTree 数据类型组织起来，包含四个参数。tree 是一个 MaxMinTree 数据类型的变量，用来存放和组织局面分数，cb 是当前的棋盘，它是一个 ChessBoard 类型的变量，depth 用于表示目前剩下的递推层数，是进行递归的重要参数，me 是自己的颜色，用于识别现在下棋的是自己还是对手，以便得到正确的分数。

在函数的内部，采用了递归方法，对于每一种可能的下棋步骤都在加入了局面分的情况下向下递归，直到遍历了递归深度之前的所有可能的局面情况。

函数没有返回值，将一个 MaxMinTree 类型的变量传入之后这个变量在经过该函数的操作后

保存了往下棋盘在一定步骤内的所有可能局势和评分。

（2）函数 MaxMinValue(tree,value):

对于当前棋局的每一种可能下法我们都将它向下模拟一定的层数，将所有的可能局势分数存在一个树的数据结构中，而函数 MaxMinValue 正是通过这棵树来计算这个步骤的优劣的函数。

函数有两个参数，tree 是经过处理的 MaxMinTree 数据类型，depth 表示目前节点（树）所在的层数（深度），是进行递归的重要参数，通过对当前的 depth 求余，也能得知目前深度的节点数据是自己下还是对手下得到的。

在函数内部，如果遇见轮到对手下则向下选取最大子树进行递归，如果自己下则向下选取最小子树进行递归。基本的结束条件有两个：第一当当前的树没有子树时，说明一方下的棋可以使对手无子可下，检查 depth，如果 depth 可以被 2 整除，说明下一步是对手下，对手可以使自己无子可下，说明自己的这一步棋可能导致比较坏的后果，函数直接返回一个较大的负值，说明该步骤不好，而如果 depth 不可以被 2 整除，说明现在在下的是对手，而自己可以使对手无子可下，直接返回一个较大的正值，说明该步骤较好；第二当递归深度达到 2 时，说明下一层就是底层了，直接返回该树子树中最大子树的数据，作为这一步棋的评分。

（3）函数 BoardTransform(board):

与第一个算法中的 BoardTransform 函数无异，这里不再重复介绍。

（4）函数 CheckValue(cb,me):

与第一个算法中的 CheckValue 函数基本无异，增加了一个参数 me，用于判断当前棋局是自己还是对手在下，以便分情况得出评分。其余情况不再重复介绍。

（5）主要函数 play(cb,ms):

黑白棋算法的主要函数，其他函数都在此函数内部被调用。参数、返回值与第一个算法中的 play 函数都一样，不再重复介绍。

在函数内部，首先进行相关的判断，能不能下四角的位置以及能不能不下四角周围的位置。之后策略是对于所有的可能步骤都建立一个 MaxMinTree，把从这个步骤出发的一定步骤内的局面得分都存在树里，再通过 MaxMinValue 函数算出这棵树，也就是这个步骤的得分，从而选取并返回得分最多的步骤。

2.3-程序限制

程序并没有限制，在有子可下的情况下总能够落子，继续下棋。但是程序本身有一些弱点。首先由于程序依赖于对步骤进行往下的模拟和推测，所以对于尾盘的处理不是很好，在面对具有一定能力的棋手时容易被翻盘。

其次采用这种模拟算法的程序及其依赖估值函数的精准度，而估值函数的精准度需要大量棋盘对局的结果数据为依据进行大量的调整才能得到一个较为合理的标准，由于缺乏足够的时间和相关的资料，为算法设计的估值函数还存在一定的误差，有时候算出的局势分值与事实不完全符合，并且随着搜索深度的增加这种误差还会叠加，最终出现较大的偏差，经过一系列的实验发现本组设计的估值函数在搜索层数为 3~5 层时的精度最好，过低或者过高都会影响到精度。

最后程序的思路相对固定，缺乏变化，在一中局势下会走出一定的棋，所以在面对有多重判断系统和方法的程序时缺乏应变能力，主要原因在于判断和策略还不够丰富，在算法中加入更多的策略和判定，在合理可控的范围内加入一些随机因素，都将有利于算法能力的提高。

测试实验

（一）实验结果

3.1-实验数据

BS：BlackScore　　WS：WhiteScore
BTa：BlackTimeagerage　　WTa：WhiteTimeaverage
BW：BlackWin　　WW：WhiteWin
E：Equal　　ME：MiddleEnd

1000 局情况	Idiot	gambler	NOVEMBER	us
Idiot	BS:27000,WS:37000 BTa:0.009343, WTa:0.012890 BW:0,WW:1000,E:0 ME:0	BS:29597,WS:34242 BTa:0.010343, WTa:0.012660 BW:394,WW:566,E:40 ME:27	BS:52994,WS:11006 BTa:0.302218, WTa:0.013175 BW:1000,WW:0,E:0 ME:0	BS:52000,WS:12000 BTa:7.100105, WTa:0.008204 BW:1000,WW:0,E:0 ME:0
gambler	BS:33327,WS:30242 BTa:0.011774, WTa:0.009511 BW:547,WW:412,E:41 ME:28	BS:30980,WS:33010 BTa:0.011771, WTa:0.011746 BW:434,WW:518,E:48 ME:10	BS:40717,WS:23268 BTa:0.420170, WTa:0.010664 BW:859,WW:108,E:33 ME:12	BS:38864,WS:25017 BTa:6.899904, WTa:0.010206 BW:788,WW:179,E:33 ME:11
NOVEMBER	BS:10020,WS:53980 BTa:0.012222, WTa:0.511001 BW:0,WW:1000,E:0 ME:0	BS:22434,WS:41563 BTa:0.011001, WTa:0.427022 BW:86,WW:888,E:26 ME:3	BS:19043,WS:44957 BTa:0.364807, WTa:0.463481 BW:2,WW:998,E:0 ME:0	BS:33989,WS:30011 BTa:5.134561, WTa:0.479599 BW:999,WW:1,E:0 ME:0
us	BS:31000,WS:33000 BTa:0.010302, WTa:7.633488 BW:0,WW:1000,E:0	BS:23339,WS:40589 BTa:0.010216, WTa:6.497760 BW:137,WW:838,E:25	BS:31021,WS:32979 BTa:0.352944, WTa:6.278246 BW:2,WW:998,E:0	BS:19000,WS:45000 BTa:5.980114, WTa:7.181029 BW:0,WW:1000,E:0 ME:0

	ME:0	ME:9	ME:0	

3.2-结果分析

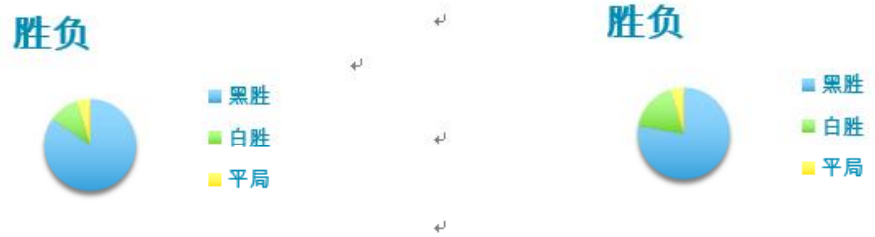
（1）对于 1000/0 的解释：

我们组的两种算法并没有任何随机，也就是说在面对任何同一种情况下，他们都会各自做出同样的决策。然而令我们不解的是，我们组的 NOVEMBER 算法总是在前两局中下出和之后不一样的情况，这在 NOVEMBER 与 idiot、NOVEMBER、us 的对战中都有所体现。

（2）胜率的直观认识：

1.gambler VS gambler：　　2.gambler VS NOVEMBER:　　3.gambler VS us

4.NOVEMBER VS gambler　　5.us VS gambler:



工作情况

（一）实习过程总结

4.1-分工与合作

本小组在实习过程中，共开了四次小组会议，历次会议主要议题如下：

2015 年 6 月 5 日星期五：确定小组分工。具体分工如下：

查找资料——韩露

编写黑白棋算法——秦树健、张虎来

修改并测试程序——刘嘉栋、刘杰

撰写报告——杨润、刘杰、韩露、刘嘉栋

2015 年 6 月 10 日星期三：确认各项工作进展情况。

2015 年 6 月 17 日星期三：讨论小组代码及比赛结果，讨论实习报告主要内容。

2015 年 6 月 19 日星期五：讨论并修改小组实习报告。

本次实习，第一套算法，也即比赛算法 T_NOVEMBER 由秦树健开发，张虎来协助开发；第二套算法，也即用于对照的算法 T_us 由刘杰开发，刘嘉栋协助开发；算法实验与测试由刘杰和刘嘉栋合作完成，刘杰负责测试与取得数据，刘嘉栋负责数据分析，测试代码由刘杰和刘嘉栋合作编写；报告由刘杰、刘嘉栋、韩露、杨润分部分完成，刘杰负责整合与排版。整体分工相对有序，合作有一定的成效。

4.2-经验与教训

本次实习作业大家完成的较为理想。在组队方面，为使各小组水平更加平均，老师和助教根据大家平时成绩对分组做了一定调整，以使各小组能够更好地完成实习作业，从而使比赛更加激烈更加精彩；在基础设施代码方面，设立了多个接口，使程序设计多样化。整体的完成情况良好，但也出现一些情况和问题，值得吸取一定的教训。

从自身角度来说，小组合作的任务应更强调时间节点与按时完成的重要性，否则容易因为某一个环节的不及时而拖慢整个小组的进度。其次，小组的及时交流十分重要，由于此次小组中有来自大三的学长，平时交流和讨论受到时间安排不一致的影响，导致了一些问题不能及时被解决。最后强调小组成员的责任心，对于小组合作的作业，每个人都应当用认真负责的态度去对待，通力合作，尽量贡献自己的力量，这样才能够使小组的作业具有较高的质量，如果不勇于承担责任分担工作，将会拖慢小组的工作效率，降低小组的成果质量。

在收获方面，在黑白棋算法程序的编写的过程中，我们对 Python 编程有了更加深入的理解，对课内学到的有关 Python 编程和数据结构与算法的知识有了更加深入的理解，并能够在一定程度上有效运用。这次作业不仅仅是对大家知识学习和编程能力的考验，更是一次学习和提升的过程，其中积累的大量编写程序、测试调整、分工协作的经验，对以后的学习和进步有很大的帮助。

致谢

感谢陈斌老师一学期以来的辛勤付出，感谢两位助教老师对我们不厌其烦的指导与帮助，感谢帮小组修改代码的师兄，感谢带领小组实习的组长刘杰同学，最后要感谢每一位为小组实习认真工作的小组同学，感谢编写代码的秦树健、张虎来同学，感谢修改并测试程序的刘杰、刘嘉栋同学，感谢撰写实习报告的刘杰、韩露、杨润、刘嘉栋同学，正是因为你们的存在，才能使小组顺利完成本次实习作业！同时感谢一起参与到这次作业、竞赛过程中来的同学们，你们的精彩表现将是我们学习的榜样，你们体现出的认真工作的精神值得大家钦佩！

再次对每一位帮助过我们的老师同学表示由衷的感谢！

参考资料

[1] 黑白棋天地

http://www.soongsky.com/

[2] 黑白棋指南

http://www.soongsky.com/strategy2/

[3] 基于遗传算法的黑白棋游戏人机博弈系统设计

http://www.doc88.com/p-009307813436.html

[4] 基于 java ME 的黑白棋游戏设计及实现

http://www.doc88.com/p-6068722279984.html

[5] 基于 QT 的黑白棋游戏设计与实现

http://www.doc88.com/p-8252913546111.html

[6] 博弈算法在黑白棋中的应用 杜秀全，程家兴 计算机技术与发展 2007 年第 17 卷

[7] 基于改进博弈树的黑白棋设计与实现 李小舟 华南理工大学硕士学位论文

[8] Experiments with Multi-ProbCut and a New High-Quality Evaluation Function for Othello / Michael Buro / NEC Research Institue

[9] A Minute to Learn…A Lifetime to Master / Brian Rose

北区 North CHARLIE 组

16　数据结构与算法课程实习作业报告

——基于α—β剪枝和估值函数思想的黑白棋算法

小组成员：龚旭日*、卢思奇、吴永祺、姜鹏飞、李庆、蒋久阳

摘　要：本组黑白棋算法主要思路是递归加棋盘估值。后期递归深度为12步，根据结束时的子数比估值；前中期递归四步，包括：位置价值、可行域、前沿点和稳定子等四个估值函数。本程序主要利用了自定义类、树、列表、字典和图等结构，以及递归、排序等算法。实验数据结果包括两次热身赛的结果和自己测试的结果。

关键字：树；列表；字典；图；递归；先验知识

一、算法思想

1.1 总体思路

博弈论中有一条定理：在二人的有限游戏中，如果双方皆拥有完全的资讯，并且运气因素并不牵涉在游戏中，那先行或后行者当必有一方有必胜/必不败的策略。若运用至黑白棋，该定理则表示为“要么黑方有必胜之策略，要么白方有必胜之策略，要么双方都有必不败之策略”。

粗略估计，如果要根据初始状态找出必胜算法，需要递归60步，假设每步有10个落子点的话，需要大约10^{60}次才能算完，在现在的硬件条件下是根本不可能的，所以我们要做的是在有限的时间内尽量逼近最优算法。比较好的逼近方法便是调用递归，进行极大极小搜索。其中递归深度越大，评估函数越好，逼近的效果也就越好。我们算法一直在尽力加深递归深度和优化评估函数。

CHARLIE组黑白棋下棋算法的主要思路是在前期时每一步去寻找下一步的最佳落点位置，递归深度为四步，且分成了5-33和34-51两个部分，因为这两个部分所处时期不同，导致评估函数不同，所以要分开进行；最后还有12步直接递归到棋局结束，寻找最后的最优路径。主要采取了树、图、列表和字典等数据结构，采用的算法主要是递归和排序。

1.2 算法流程图

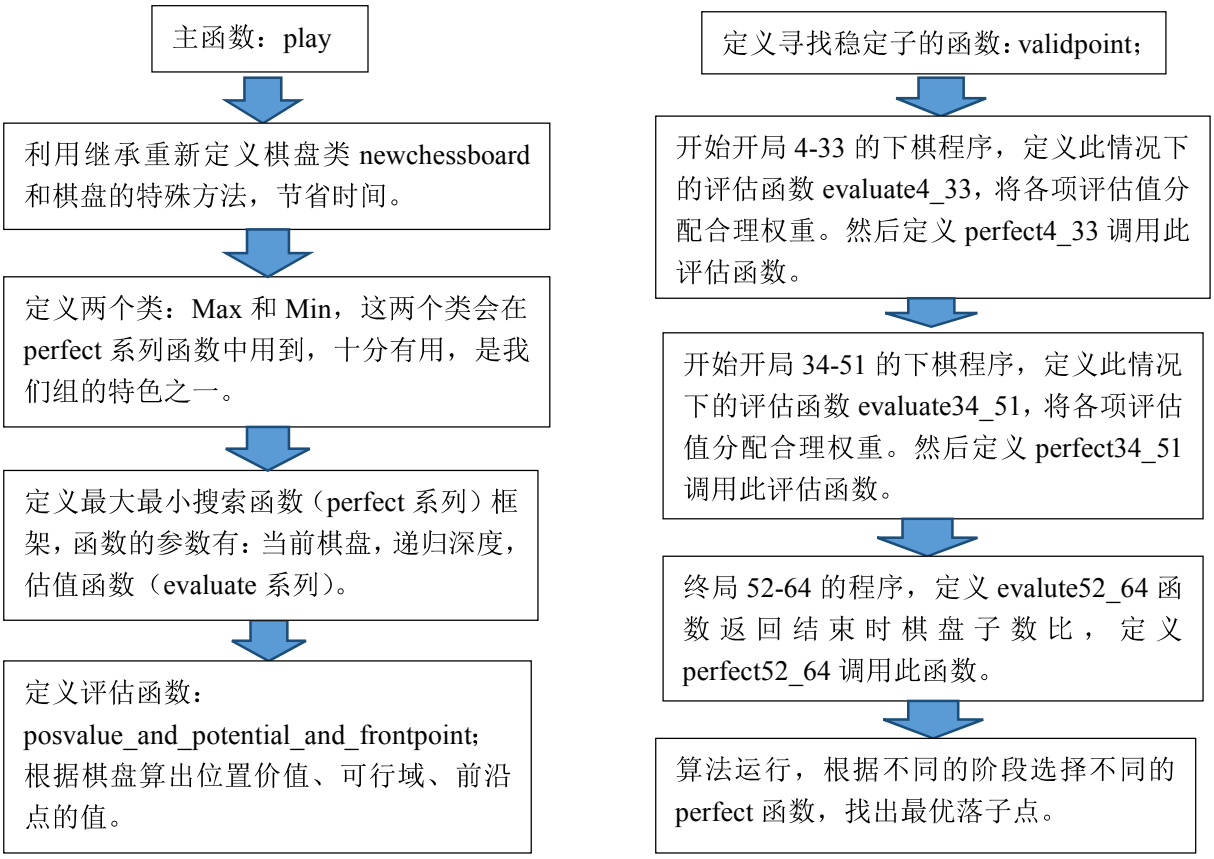


图 1 算法流程图

1.3 算法运行时间复杂度分析

我们算法的核心思想是递归，主要的运行时间也花在了递归上。我们的递归算法所花时间基本上以阶乘的形式增长，所以算法复杂度应该是 O (n!)。

但是，所有的递归算法时间复杂度都会达到 O (n!)，我们认为，导致递归深度差异的区别反倒应该体现在每步常数级别的递归操作上。比如如果我们每次用 newcb = chessboard (cb) 复制一次棋盘，至少要进行一两百步的操作，而陈斌老师的 getLegalPos 操作也存在重复和优化空间。因此我们决定新定义一个棋盘类，把每一步递归的操作数尽可能减少，以此加深递归深度。

此外，剪枝也是提高递归深度的重点所在。采用合适的剪枝方法可以避免很多无谓的计算，也大大提升了时间。合适的剪枝也依赖于递归搜索的次序，也就是先验规律，但黑白棋的先验规律比较复杂，因此提升空间不大。

总之，我们算法的核心就是围绕提高递归深度来进行的，为此我们采用了很多的创新方法，将最开始只能从 58 步开始搜索的递归提升到现在的 52 步开始（如果不考虑之前的下棋时间最高可以从 48 步至 50 步进行搜索），粗略分析,搜索速度提升了 10000 倍。

二、程序代码说明

2.1 数据结构说明

我们组的算法中主要采用了树、图和一些线性结构，自定义了 newchessboard、Max 和 Min 类。其中，最值得一提的是我们自己定义的 newchessboard 类，这个类是我们组能够获胜的重要原因。因为使用基础程序自带的棋盘会有大量的时间浪费，导致程序的递归深度难以加深，影响程序的效率，所以我们自己编写了一个新的棋盘类，大大的节省了运行时间，使得递归深度得以加深，取得了很大的优势。此外，我们的剪枝方法也有创新，在末尾的递归中，我们采用的是找到一个必胜策略就进行剪枝，而不是寻找最优解，同时还记录必胜走法，这些都依靠 Max 和 Min 类来实现。

2.2 函数说明

newchessboard 类：利用继承，定义的新棋盘类，为了进一步压缩时间，改写了老师给出的 chessboard 类中的一些函数。具体来讲，新定义了一个属性（字典类），用以记录每一步下完之后对棋局产生的改变，目的是为了实现 delturn 函数；delturn 函数是 newchessboard 类中的精髓，它可以消除这步棋对棋盘的影响，也就是在递归的过程中采用复原棋盘而不是新建一个棋盘的方法，大大节省了时间；此外还定义了记录空点的属性，用以改写 getlegalpos 函数，在棋局的末尾，对空点进行搜索比对已有的子进行搜索要快速的多；此外还有一些其他的小改动用以节省时间。

Max、Min 类：由于递归的过程中，我方和对方的选择情况是不同的（都选择自己优势大的，也就是我方选得分大的，对方选我方得分少的），因此需要加以区分。Max 和 Min 类既方便建立递归树，同时又直接区分了我方和对方的选择，在递归调用的时候更加清楚。

posvalue_and_potential_and_frontpoint()函数：估值函数。Posvalue 是对当前位置的评估，比如角的权重大，角周围的权重要减小；potential 是行动力的大小，也就是我方的合法位置的多少；frontpoint 是前沿点，也就是我方有多少棋子处在外层；之后新

加入了 aroundpoint 的计算，是指我方棋子周围有多少个对方棋子，即被包围程度。

Validpoint()和 stablepoint()：稳定子（只考虑边上和角上的），validpoint 是某一方的个数，stablepoint 是我方减去对方的个数。

evaluate4_33()：4 至 33 步的评估函数，对上面的每一项赋予了权重。

perfect4_33()： 4 至 33 步的递归。递归深度为 4 步，按评估函数决定好坏，取最好的。

evaluate34_51()和 perfect34_51()：基本同上，只是对评估函数的权重做了调整。

perfect52_64()：本组设计的最有创新性和优胜面的函数。在第 52 步时搜索到棋局结束，计算双方的得分，看能否找到必胜策略，如果能找到那么本局比赛已经结束了。如果不能，那么基本可以确定已经输了。因为该函数的剪枝思路为若找到必胜策略其得分为满分（也就是直接剪枝不再搜索其他的），而且记录必胜策略，以后按必胜策略下棋；但如果赢不了那么得分为 0（也不再搜索其他的了），然后继续搜索其他可行的下法，如果所有可行下法都没有必胜策略，那么该算法失效。

perfect52_64_1()：接上。如果不能找到必胜策略，那么进行八步不剪枝的全搜索，取得分最高的，先走一步。之后看对方应对再用上一方法搜索，看是否有机会。在第 56 步以后将不再采用上一方法。

Play()：主函数。执行以上下法。加入了计时方法，对开局（4 至 33）、中局（33 至 51）和终盘的每一步设定好了时间，当时间截止时强制结束递归，取当前得分最好的，避免超时问题。

2.3 程序限制

本组的程序在时间和其他因素上考虑的比较全面，暂时未发现会导致程序出错的限制。特别是我们加入了记录时间的全局变量，极大的避免了超时的可能。但是，我们程序的计时功能并不完美，在某些极端情况下（某一步的时间过长）的情况下也是有可能超时的，不过目前未遇到这种情况。

三、实验结果

3.1 实验数据

实验环境说明：

硬件配置：Intel Core i5，内存 4G；

操作系统：Windows 7；

Python 版本：2.7.9

1000 次对弈结果数据汇总

黑 \ 白	idiot 算法	gambler 算法	CHARLIE 算法
idiot 算法	0:0:1000 27000:37000 14.1128:19.4569	506:31:463 32942:30681 18.5540:15.0986	0:0:1000 23368:40632 122099.6115:16.3584
gambler 算法	357:33:610 28334:35595 14.9257:18.8384	456:36:508 31618:32378 17.5396:17.7025	0:0:1000 47564:14587 83253.1574:11.6542
CHARLIE 算法	1000:0:0 55000:9000 39898.6593:16.3584	1000:0:0 46423:17490 82663.5381:11.7977	1000:0:0 43000:21000 66589.3126:70569.5462

3.2 结果分析

从测试数据中可以看出，CHARLIE 算法在和 idiot 算法以及 gambler 算法的对决中，完成了 1000 次测试完胜的好成绩，也不存在超时的问题。在与 idiot 的对弈中，前期的递归和评估函数就已经完全压制住了 idiot，前期的走法显然比它更加合理，因此前期有优势。再结合第 52 步的 **perfect52_64()**，稳赢是没有问题的。CHARLIE 和 CHARLIE 对战，由于两个程序每一步都是确定的，所以会一直出现执白赢，执黑输的情况，两者的时间也是相对固定的。

Gambler 是随机下法，很难和我们正规的递归加评估函数抗衡（不过理论上 gambler 是有走出必胜走法的可能的，可是概率太小了，迄今还未发现 gamber 打赢 CHARLIE 的情况）。时间上由于我们的设定基本上没有问题，主要开销是在递归上面了，这一点在时间复杂度分

析中已有一定的介绍。

四、 实习过程总结

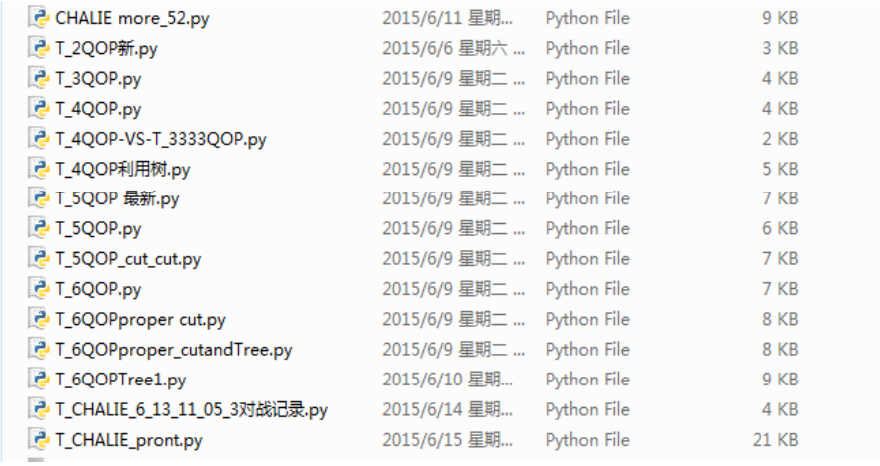
4.1 分工与合作

在得到数算课程决定用实习作业取代期末机考的消息后，我们都感到很高兴，但也感受到了压力。在分组之后，我们马上确定了分工安排，首先每个人先根据自己的想法编写一个简单的程序，通过程序之间的比较来确定最初的方向。然后我们在网上查取了若干关于黑白棋算法的资料，以此来确定我们黑白棋算法的主体内容。

程序的主要编写过程主要由龚旭日和卢思奇两位同学完成的，两位同学在两个多星期的时间内，抛开了其他的事情，全心全意的投入到了程序的编写中，可以说到了废寝忘食的程度，只要一有时间他们就会琢磨程序哪里还需要修改。正是他们这种精益求精的精神，才使得 CHARLIE 算法大放光彩；吴永祺同学也提供了许多前期的算法程序，同样为团队做出了杰出的贡献；姜鹏飞同学主要负责后期实习报告的整理和编写，记录了小组内每个人的分工和具体贡献；蒋久阳同学提供了前期的算法程序，并对报告的格式、语言和内容安排做了修改和调整；李庆同学负责报告中的程序测试，在测试过程中甚至发生了电脑损坏，做出了很大的牺牲。

4.2 经验与教训

在这次实习中，我们首先通过上网查找资料了解了此类问题的基本解决思路。最开始，我们的思路就分成了两条线，第一是递归的思想，第二是评估函数的思想。在任务布置下来后的前几天，组员分别尝试写出了 T_easytry、T_tryyoupos、godlike、idiot1 和 T_demo 等几个程序，这几个程序奠定了我们最后程序的主体思路：递归算分数、对位置的评估和对可行域的评估等等。之后大家又分别写了几个程序来比赛，也总是互有胜负，可见此时的程序之间基本没有质的差异。之后我们组两位主要写程序的同学龚旭日和卢思奇决定在递归上面进行突破，龚旭日同学通过查阅网上的资料掌握了α—β剪枝法，并提出了自己的独到剪枝方法，大大提高了递归的深度；卢思奇同学则提出了改写棋盘的设想并付诸实践。在第一周的周末，结合了二者的优势的算法 QOP 系列诞生了。QOP 系列的算法已经基本实现了最后 Charlie 的后期算法，程序也有了本质的提高。从此再也不担心被 idiot 和 gambler 打败了。



CHALIE more_52.py	2015/6/11 星期...	Python File	9 KB
T_2QOP新.py	2015/6/6 星期六 ...	Python File	3 KB
T_3QOP.py	2015/6/9 星期二 ...	Python File	4 KB
T_4QOP.py	2015/6/9 星期二 ...	Python File	4 KB
T_4QOP-VS-T_3333QOP.py	2015/6/9 星期二 ...	Python File	2 KB
T_4QOP利用树.py	2015/6/9 星期二 ...	Python File	5 KB
T_5QOP 最新.py	2015/6/9 星期二 ...	Python File	7 KB
T_5QOP.py	2015/6/9 星期二 ...	Python File	6 KB
T_5QOP_cut_cut.py	2015/6/9 星期二 ...	Python File	7 KB
T_6QOP.py	2015/6/9 星期二 ...	Python File	7 KB
T_6QOPproper cut.py	2015/6/9 星期二 ...	Python File	8 KB
T_6QOPproper_cutandTree.py	2015/6/9 星期二 ...	Python File	8 KB
T_6QOPTree1.py	2015/6/10 星期...	Python File	9 KB
T_CHALIE_6_13_11_05_3对战记录.py	2015/6/14 星期...	Python File	4 KB
T_CHALIE_pront.py	2015/6/15 星期...	Python File	21 KB

图 2 核心程序列表

之后我们又决定在前期进行突破，思路仍然是递归加上评估函数。递归部分和后期大同小异，直接调用就可以，可是评估函数的确定就非常的困难，网上的资料也只是指明了大致的方向，对于具体的实现没有什么帮助。我们开始自己动手创建评估函数。综合网上给出的建议和小组讨论，我们最终确定了需要评估的几个方面。两位写程序的同学又继续将这些想法付诸实践。之后，我们不断调试修改参数，希望能找出较好的程序。

紧接着是两次热身赛。在热身赛中，我们发现程序存在超时的问题，大家讨论认为需要加入对时间的控制。于是我们在程序中加入了计时，避免了可能因为超时带来的遗憾。

在热身赛中，我们也发现了强劲的对手 LIMA、KILO 和 GOLF。在正式比赛的八强赛中，我们和 GOLF 组相遇，十分惊险地取得了胜利。我们还发现 LIMA 的程序对我们有相当大的优势，于是决定再做改动。我们分析了输掉的对局，发现问题主要来源于我们的棋子被别人穿插，于是我们抓紧写出来包围点的评估函数，避免这个问题。在组内的测试中，加上包围点函数的程序几乎所向披靡。

但是在四强赛中，INDIA 组的程序使得我们最新写出的程序以大比分输掉了比赛，同时原有的程序在小分上也不占优势，最终遗憾的输给了 INDIA 组。坦白地说，我们之前确实对 INDIA 组缺乏了解，同时对 INDIA 组能够在 3 天之内取得如此大的改变感到钦佩。算法竞赛瞬息万变的刺激与乐趣，也由此也可见一斑。在季军争夺战中，我们组惊险地战胜了 KILO 组，取得了第三名的好成绩。

总结我们失败的原因，还是在于我们前期的评估函数权重设置的不够合理，显得前期较

为被动。同时，我们后期递归又过于依赖前期的局势，如果前期局势不好，后期找不到必胜算法，基本上就会输掉比赛。而前期的算法又是用评估函数评估的，因此当遇到比较克制我们的评估函数的对手时就极为被动，算法的优势也就不存在了。总体而言，前期的评估函数是一大薄弱环节。尽管我们尝试做出改变，但效果却并不显著，仍然没有本质上的提高。这也许是我们最终输给 INDIA 的原因。

4.3 建议与设想

我们觉得此次实习作业的提出是十分具有创新意义的，解决了上机考试不便的问题，但由于是第一次实施，难免会有一些不足。我们的建议是：在组队方面可以做到自由组队或者联系紧密的人（例如同寝室室友）组队，人也不宜太多，这样可以更利于对问题进行讨论，有利于工作的展开；在竞赛的安排和组织工作上，我们觉得本学期这样的安排是比较合理的。

数据结构与算法这门课是十分有用的。既然我们已经将自己的想法付诸于实践，就可以不仅限于提交一次实习作业，还可以将这些的想法延续下去，等到掌握的知识更多了，还可以将其和图形界面联系起来，做出一个可以实际操作的应用，这样就更好了。

如果有条件，可以搭建一个网络平台，由同学们上传自己的算法，然后自动和已有的程序比拼，算出排名。这有些像天梯排位，可以激发同学们编程的兴趣，以后有什么新的创意算法都可以编出来在平台上跑一跑，这样会产生越来越强的黑白棋算法，对于下一届黑白棋算法比赛也有很大帮助。

五、致谢

感谢陈斌老师为我们提供了这次实习的机会，感谢几位助教老师的辛勤工作，你们的工作使这次比赛的成功举办成为了可能。特别是陈斌老师和助教为我们编写的基础设施程序，一定耗费了你们大量的时间，所有的参赛同学都会感谢你们的辛勤付出，没有你们的基础设施的编写，这次算法竞赛就会困难很多。

同时也要感谢各位组员。在这几周里面，几位组员都非常辛苦，两位写程序的同学几乎整天不离电脑，为我们组最后的成绩付出了大量的心血。无论结果如何，他们优秀的创意和刻苦钻研的精神都值得其他同学感谢和钦佩。其他几位同学虽然没有直接参与主程序的编写，

但是也做出了巨大的贡献，吴永琪同学为了写出一个好的开局函数，查阅了大量的资料，尽管最后因为不匹配问题而没有用上，但是也为本组付出了很大的努力。姜鹏飞同学主动承担实验报告的撰写，记录下我们组的经历和想法，也是作业中不可或缺的一部分。蒋久阳同学提供了一个可做测试对手的算法，并对报告的格式和内容做了修改。李庆和蒋久阳两位同学与组内其他四位同学不在一个寝室，但也积极参与了小组讨论，为算法的改进和提高献言献策，同时承担了其他的工作，减轻了其他几位同学的负担。

回想起我们测试调整参数的几个小时，回想起我们实现新函数所花的几天，以及我们为了研究算法的缺陷而对棋谱进行分析的过程，虽然我们没有最终实现夺取冠军的理想，但是在这当中所有人的努力都值得大家铭记和感谢。

参考文献

曹森. 2012. 对α-β剪枝算法的性能改进研究. 硕士论文：内蒙古师范大学，1 - 72.

杜秀全，程家兴. 2007. 博弈算法在黑白棋中的应用. 计算机技术与发展，17（1）：216 - 218.

李小舟. 2010. 基于改进博弈树的黑白棋设计与实现. 硕士论文：华南理工大学，1 - 52.

瞿锡泉，白振兴，包建平. 2005. 棋类博弈算法的改进. 现代电子技术，1：96 - 99.

张聪品，刘春红，徐久成. 2008. 博弈树启发式搜索的α-β剪枝技术研究. 计算机工程与应用，44（16）：54 - 97.

http://blog.csdn.net/nowcan

http://www.hbqhome.com/forum.php

http://www.paopaoche.net/gonglue/3724.html

http://www.soongsky.com

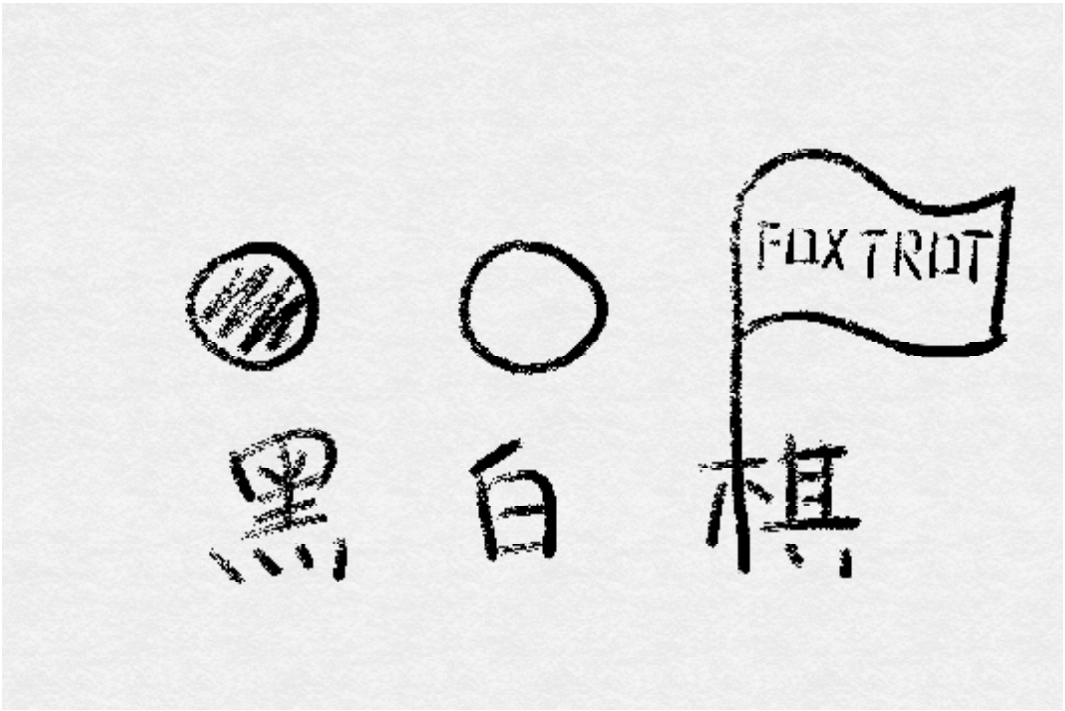
北区 North　FOXTROT 组

17　数据结构与算法课程实习作业报告

李然*　林芷平　吴梦彤　王静　王慧君　蒙聪

摘要：算法采用树的数据结构类型实现深度搜索，预见棋局的走势，博弈树运用了递归的算法策略，定义了节点的类，实现了树节点的多种功能；并设置一个动态的估值函数，根据棋盘中棋子的分布决定优先级。在与 idiot 与 gambler 的对弈结果中，保持了全赢的记录。

关键字：二叉树、自定义节点类、递归、估值函数



17.1算法思想

17.1.1算法总体思路：

为棋盘设计一个与位置相关的优先级，按照优先级的高低决定落子的位置。利用估值

函数(univalue),利用类和树的数据结构与算法,在下棋的过程中对未来的棋盘进行预期，来评判棋局的优劣，以此为依据来确定下哪一步棋，使我方获得最大的优势。估值函数运用了价值表（棋盘的棋子势能），也包含了行动力、潜在行动力及稳定子等因素，全面评估了下每一步棋的收益多少。

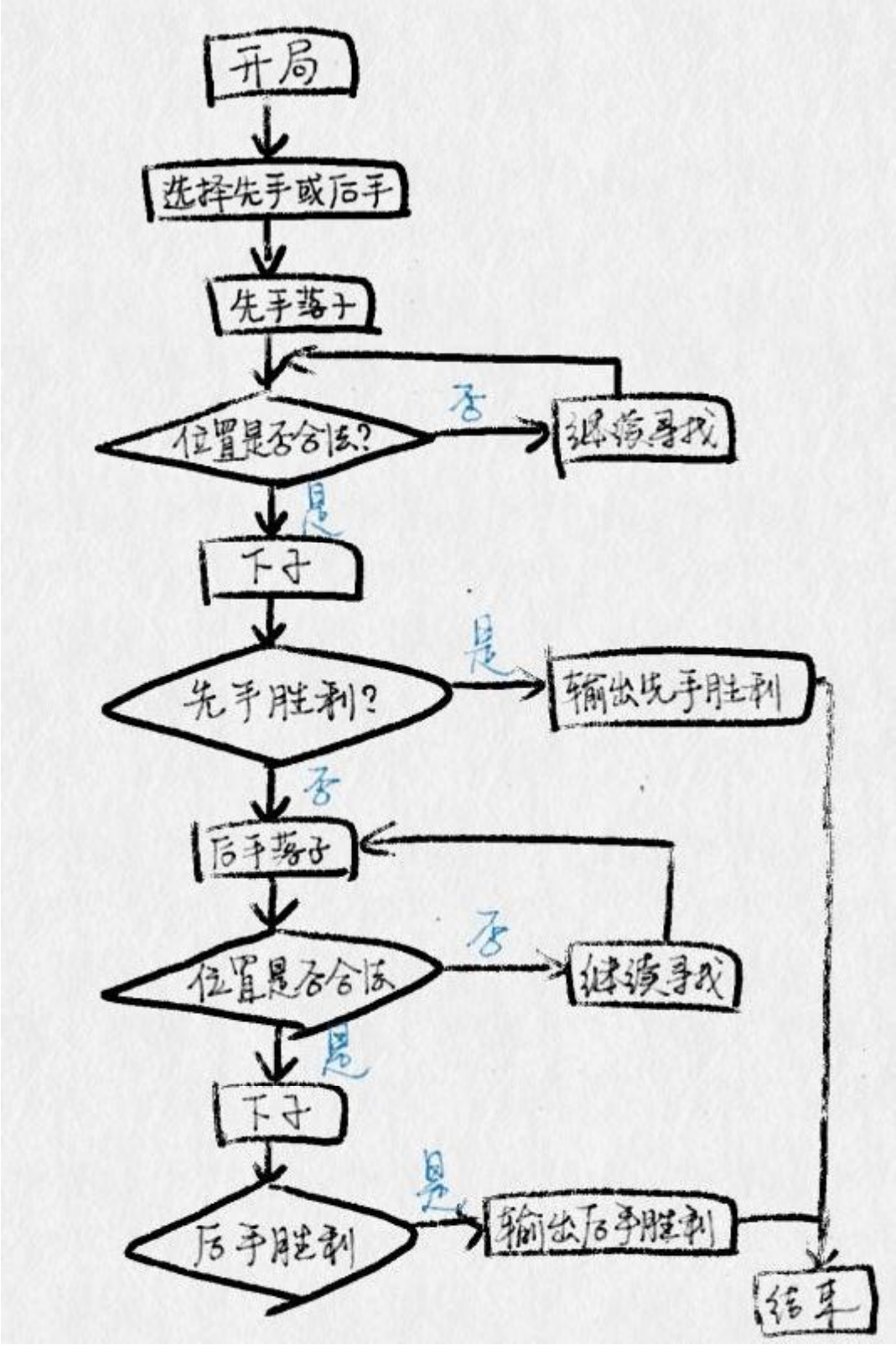
17.1.2数据结构与算法：

主要采用了课堂上讲的二叉树深度搜索，以及自定义节点类等数据结构来实现算法构思。

17.1.3算法策略：

算法主要分为两部分，一是采用树的结构类型实现深度搜索，预见棋局的走势，博弈树运用了递归的算法策略；二是设置一个动态的估值函数，根据棋盘中棋子的分布决定优先级。

17.1.4算法流程图


开局
选择先手或后手
先手落子
位置是否合法?
否
继续寻找
是
下子
先手胜利?
是
输出先手胜利
否
后手落子
位置是否合法
否
继续寻找
是
下子
后手胜利
是
输出后手胜利
结束

17.1.5算法运行时间复杂度的分析

算法运行时间的主要构成是树的遍历与树的子节点对应的估值函数的运算。

估值函数对应的时间复杂度为 o(1)，树结构的探索深度是 5 层，由于落子点数正比于棋盘规模，棋盘规模为 N，故非剪枝算法运行的时间复杂度为 N^5*N=N^6。

在 ALPHA-BETA 剪枝算法中，由于每个子树发生剪枝的概率相同，平均的时间复杂度为(N/2)^6，故 ALPHA-BETA 剪枝算法的运行的时间复杂度也为 N^6。

即最终得出的算法时间复杂度为 N^6。

17.2程序代码说明

17.2.1数据结构说明

本算法主要采用了树的数据结构和节点的自定义类(class)来实现函数的功能。

17.2.2自定义类 node:

node 为树的节点，Class node 中包含了 getboardy(返回当前棋局)，getlevel（返回节点所在的层数），getch（返回节点对应的某个子节点），getchildren（返回节点对应的全部子节点），insert（在某个位置插入节点）等功能，具体见下图：

```
class node:
    def __init__(self, cb,pos, val=None):
        self.boardy = cb
        self.val=val
        self.pos=pos
        self.children = []
        self.parent=None
        self.level=0
        self.a=-10000
        self.b=10000
    def getboardy(self):
        return self.boardy
    def getlevel(self):
        return self.level
    def getch(self, order):
        return self.children[order]
    def getchildren(self):
        return self.children
    def insert(self, order, node):
        self.children.insert(order,node)
        node.parent=self
        node.level=self.level+1
        node.a=self.a
        node.b=self.b
    def getparent(self):
        return self.parent
    def getpos(self):
        return self.pos
    def getval(self):
        return self.val
    def delete(self):
        self=None
        return self

    def __contains__ (self,n):
        return n in self.children
    def __str__ (self):
        return str(self.boardy)+'\n'+str(self.val)+'\n'/
            +str(self.pos)+str(self.children)
    __repr__ = __str__
```

```
def gm(self):
    if self.getchildren()==[]:
        self.val=self.getval()
    elif self.level%2==1:
        self.val=min(x.getval() for x in self.getchildren())
    else:
        self.val=max(x.getval() for x in self.getchildren())
    return self.val
def cut(self):
    if self.level%2==0:
        if self.getval()<self.a:
            return True
        else:
            return False
    else:
        if self.getval()>self.b:
            return True
        else:
            return False
```

17.2.3树的数据结构：

本组的树的构建与探索历程较复杂，最先构建了广度优先搜索树，采用 MIN-MAX 算法。之后为了节省时间，希望使用耗时几乎可以减半的 ALPHA-BETA 算法，故为此开发了深度优先搜索树。此处重点介绍 MIN-MAX 方法和 ALPHA-BETA 剪枝算法。

①MIN-MAX 搜索树：

算法运用的主要数据类型结构是树，下棋是一个典型的博弈过程，所以在此处应用的树特别称为博弈树。博弈树的节点代表可能出现的棋盘局势。为了找到对自己最有利的下棋方法，就需要不仅只看当前利益，还要预测未来的发展，把所有可能的情况列出来，然后加以分析判定。但是由于每一步可走的方案有很多种，深入地搜索会产生极其庞大的博弈树，所以只能向前搜索有限几步。

最基本的博弈树采用了最大最小搜索，假设己方是黑棋，由于对对方的落子位置并不清楚，所以只能认为白棋每下一步都使自己的利益最小，而相反地，自己每一次落子都尽可能增大自身利益。所以对应到博弈树上就是 MAX 节点和 MIN 节点，下棋也就变成了一个最大最小交替的过程，具体为：

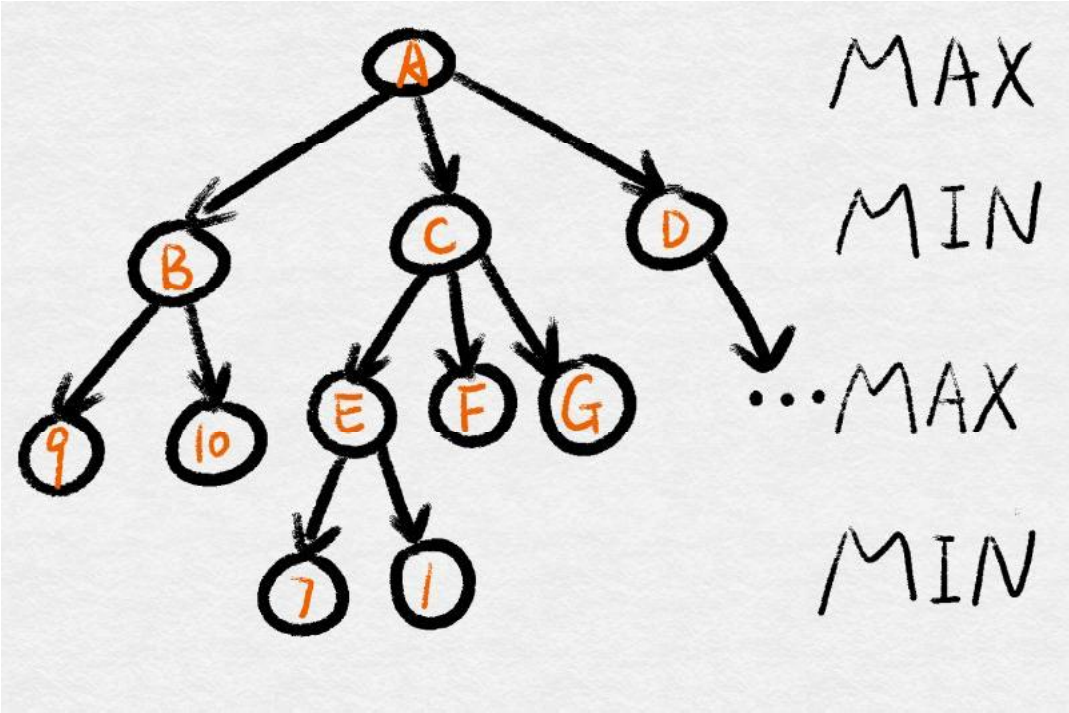
A. 己方下的节点标记为 MAX 节点，对方下的节点标记为 MIN；
B. 向前搜索三层后，按照之前的标记一层一层将节点的值向上传递；
C. 传递到此时的根节点后将根节点的值作为此步的最佳落子方案。

```
def gm(self):
    if self.getchildren()==[]:
        self.val=self.getval()
    elif self.level%2==1:
        self.val=min(x.gm() for x in self.getchildren())
    else:
        self.val=max(x.gm() for x in self.getchildren())
    return self.val
```

上图为在二叉树类中定义的最大最小交叉搜索（MIN-MAX 搜索）的函数。

②MIN-MAX 搜索树的改进与优化——α -- β 搜索：

由于上述最大最小搜索中每一步都有多种方法需要搜索判断，这就使得程序面临着超时的危险。而经过分析我们可以发现，有的步骤是不需要进行搜索的。



如上图，根据右边的最大最小标注，我们模拟一下从下到上返回搜索值的过程：

A. MIN-->MAX: E = 3

B. MAX-->MIN: B = 9 , C<=3 (假设 F、G 的值未知)

C. MIN-->MAX: A >= 9 (假设 D 未知)

由此可见，A 最后的取值与 F、G 的值没有关系，所以 F、G 没有必要搜索了，这就是 α--β搜索的一个实例。

算法实现较为一般的过程为，在每一个节点引入新的参量 a,b 作为上下限，两者构成一个开区间（a，b），将 a 的初始值设得足够大，b 的足够小，其他节点的值从父节点继承过来。继承过程的判断为：对于 MAX 节点，若子节点的返回值大于 a，就将 a 的值更新为该

返回值：对于 MIN 节点，若子节点的返回值小于 b，就将 b 的值更新为该返回值。这样一层一层搜索时，a 的值不断增大，b 的值不断减小，直到 a>=b，区间不合法时，就发生了剪枝。

```
self.a=-10000
self.b=10000
```

上图为剪枝算法在__init__函数里的初始值。

```
def cut(self):
    if self.level%2==0:
        if self.getval()<self.a:
            return True
        else:
            return False
    else:
        if self.getval()>self.b:
            return True
        else:
            return False
```

上图为在节点类中定义的剪枝算法的函数。

```
t= node(cb,'D4')
if cb.getTotalScore()<=60:
    for l in range(len(t.boardy.getLegalPos())):
        ll=buildnode(t,l)
        for k in range(len(ll.boardy.getLegalPos())):
            kk=buildnode(ll,k)
            for j in range(len(kk.boardy.getLegalPos())):
                jj=buildnode(kk,j)
                for i in range(len(jj.boardy.getLegalPos())):
                    ii=buildnode(jj,i)
                    ii.val=univalue(ii.boardy,ii.boardy.getTurn())
                    if ii.cut():
                        break
                jj.gm()
                if jj.getval()>jj.a:
                    jj.parent.a=jj.getval()
                if jj.cut():
                    break
            kk.gm()
            if kk.getval()>kk.a:
                kk.parent.a=kk.getval()
            if kk.cut():
                break
        ll.gm()
        if ll.getval()>ll.a:
            ll.parent.a=ll.getval()
        if ll.cut():
            break
    t.gm()
```

上图为最后构建的树的结果。

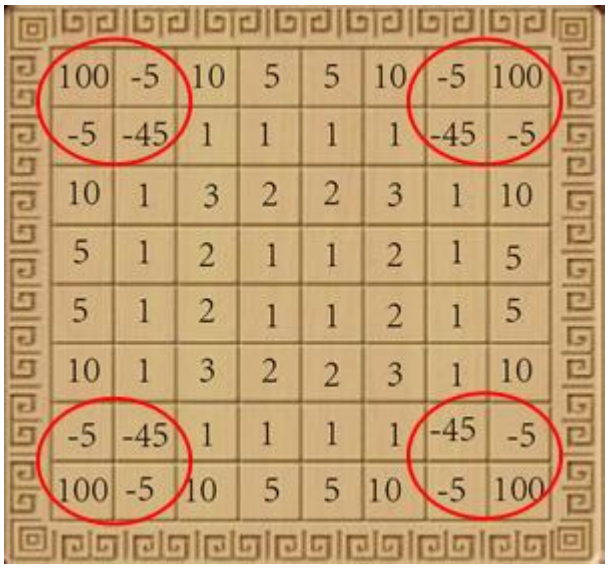
17.2.4函数说明

下棋过程中的某一步有好几种走法可供选择时，棋手要作出抉择，选择对自己最有利的走法。判断走哪步最有利的办法就是利用估值函数。这个函数对每一个局面给出一个估值，估值估值越高表明对自己越有利。在我们的作业中，univalue 就是这个估值函数。接下来我就从初局、中局、渐终局、终局这四个阶段(stage)来分别进行详尽的阐述。

先解释几个概念：

1.行动力(mobility)：可以着子的位置的总数。

2.位置价值表(evalist)：棋盘上不同位置的棋子势能，棋子势能越高对本方越有利，是总结的经验数据。



3.潜在行动力(potentialmobility):对方棋子周围的空位，暂时不在我方行动力范围之内但随着棋局推进有可能成为我方落子点的位置。

4.稳定子(stable)：每个方向都是安全的无法被翻转的棋子。

估值函数的返回值：pretotalscore

Stage 1：初局(start)

1.在对整个棋局的价值进行运算的过程中，若某位置为我方棋子所占，就在 pretotalscore 中加上此位置的棋子势能，相反若为对方棋子所占，则减去此位置的棋子势能。若无棋子，不做处理。详细代码见下。

```
if stage(cb) in ['start','mid']:#初局中局用价值表
    for i in range(8):
        for j in range(8):
            if cb.board[i][j]==self:
                s[i][j]=1
            elif cb.board[i][j]==op:
                s[i][j]=-1
            pretotalscore+=evalist[i][j]*s[i][j]
```

2.在 pretotalscore 中加上行动力(mobility)与潜在行动力(potentialmobility)这两个因素，详见下图。

```
if stage(cb) in ['start','mid','toend']:
    pretotalscore+=mobility+potentialmobility(cb,self)
```

Stage 2：中局(mid)

1.用与初局相同的方法计算整个棋局的价值。
2.同初局一样，在 pretotalscore 中加上行动力(mobility)与潜在行动力(potentialmobility)这两个因素。

Stage 3：渐终局(toend)

1.同初局、中局一样，在 pretotalscore 中加上行动力(mobility)与潜在行动力(potentialmobility)这两个因素。
2.渐终局中，在 pretotalscore 中加上稳定子这个因素，详见下图。

```
if stage(cb)=='toend':
    pretotalscore+=stablenum(cb,self)
```

Stage 4：终局(end)

游戏的终局以翻转对方的棋子多为目的，此时棋盘上我方棋子多为唯一衡量标准，详细代码见下。

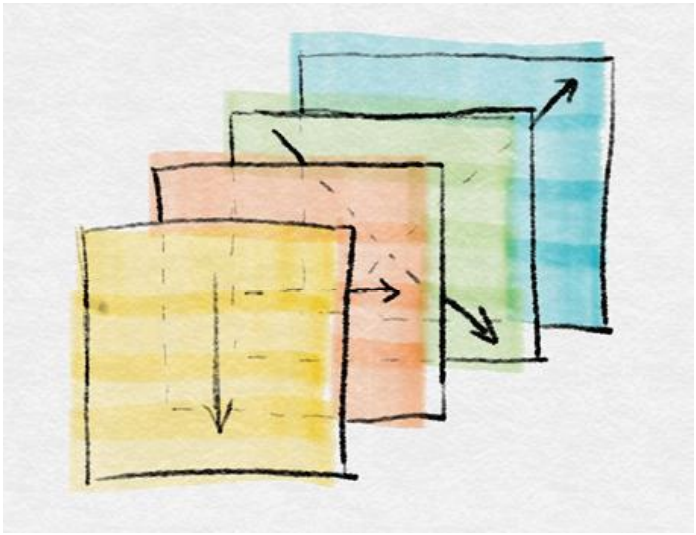
```
elif stage(cb)=='end':
    if self=='#':
        opscore=cb.getScore()[1]
    else:
        opscore=cb.getScore()[0]
    pretotalscore=cb.getTotalScore()-opscore
return pretotalscore
```

关于稳定子数目的计算：

稳定子是指安全的无法被翻转的棋子。其安全性要从 4 个方向来判断，分别为行，列，左上到右下斜着，右上到左下斜着。

整体分三种情况判断：我方棋子两边被对方夹住，整排已满无论是哪方棋子，我方棋子连到边界。接下来结合代码作详尽阐述。

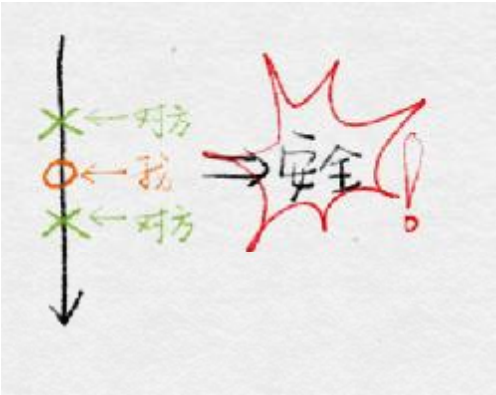
首先建立四个棋盘，每个棋盘是一个列表，分别记录四个方向的情况。如下图所示：



代码如下：

```
def stablenum(cb,self):#稳定子，中局再考虑
    stable=[[[0]*8]*8]*4#第一层4个子列表，分别记录某方向是否稳定,0为j方向，1为i方向
    finalstable=[[0]*8]*8#存储最终所得稳定子
    if self=='#':
        op='O'
    else:
        op='#'
    num=0
```

1.**我方棋子两边被对方夹住。**如下图所示。



分别从四个方向判断是否被对方棋子所夹住，同时注意不要越界，夹住为安全。

```
def ifsurrounded(cb,stable,op,self):     #看某个方向是否被对方棋子夹住，夹住为安全
    for i in range(8):
        for j in range(8):
            for k in range(3):
                if cb.board[i][j]==self:
                    if k==0:             #行
                        if ((j-1>=0 and cb.board[i][j-1]==op) or j-1<0)and\
                           (j+1>7 or(j+1<=7 and cb.board[i][j+1]==op)):
                            stable[k][i][j]==1
                    elif k==1:           #列
                        if ((i-1>=0 and cb.board[i-1][j]==op) or i-1<0)and\
                           (i+1>7 or(i+1<=7 and cb.board[i+1][j]==op)):
                            stable[k][i][j]==1
                    elif k==2:           #左上到右下斜着
                        if ((j-1>=0 and i-1>=0 and cb.board[i-1][j-1]==op) or (j-1<0 or i-1<0))and\
                             ((j+1>7 or i+1>7)or(j+1<=7 and i+1<=7 and cb.board[i+1][j+1]==op)):
                            stable[k][i][j]==1
                    elif k==3:           #右上到左下斜着
                        if ((j-1>=0 and i+1<=7 and cb.board[i+1][j-1]==op) or (j-1<0 or i+1>7))and\
                             ((j+1>7 or i-1<0)or(j+1<=7 and i-1>=0 and cb.board[i-1][j+1]==op)):
                            stable[k][i][j]==1
    return stable
ifsurrounded(cb,stable,op,self)
```

2.整排已满无论是哪方棋子。如下图所示。

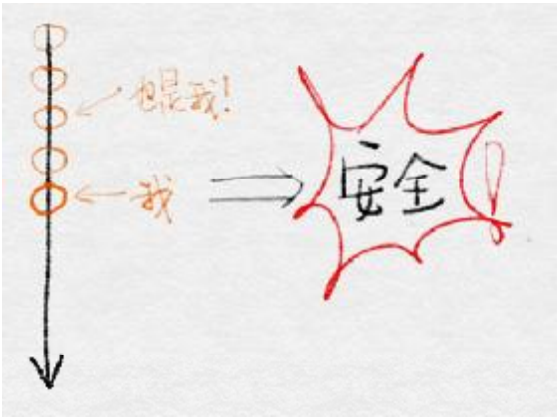
从四个方向分别判断整排是否已满，无论是哪方棋子，只要整排已满，在此方向上棋子安全。代码如下：

```
for i in range(8):#每个棋子某方向是否满行
    if iffull(cb,0,i,0):        #行
        for p in range(8):
            if cb.board[i][p]==self:
                stable[0][i][p]=1
    if iffull(cb,1,0,i):        #列
        for p in range(8):
            if cb.board[p][i]==self:
                stable[1][p][i]=1
    if iffull(cb,2,i,0):        #左上到右下
        for p in range(i,8):
            if cb.board[p][p-i]==self:
                stable[2][p][p-i]=1
    if iffull(cb,3,0,i):        #右上到左下
        for p in range(8-i):
            if cb.board[p][p+i]==self:
                stable[3][p][p+i]=1
```

其中 iffull 函数用来判断棋子所在某个方向上是否满行，截取其中部分代码如下：

```
    else:#右上到左下斜着
        pi=i-1
        pj=j+1
        while (pi>=0 and pj<=7):
            if cb.board[pi][pj] in ['.','+']:
                return False
            pi-=1
            pj+=1
        di=i+1
        dj=j-1
        while (di<=7 and dj>=0):
            if cb.board[di][dj] in ['.','+']:
                return False
            di+=1
            dj-=1
        return True
```

3.**我方棋子连到边界。**如下图所示。



此种情况仍然分 4 个方向判断，分别为：

行与列：

```
def iffull(cb,direction,i,j):
    if direction==0:#第i行是否满
        for k in range(8):
            if cb.board[i][k] in ['.','+']:
                return False
        return True
    elif direction==1:#第j列是否满
        for k in range(8):
            if cb.board[k][j] in ['.','+']:
                return False
        return True
```

左上到右下斜着：

```
elif direction==2:#左上到右下斜着
    pi=i-1
    pj=j-1
    while (pi>=0 and pj>=0):
        if cb.board[pi][pj] in ['.','+']:
            return False
        pi-=1
        pj-=1
    di=i+1
    dj=j+1
    while (di<=7 and dj<=7):
        if cb.board[di][dj] in ['.','+']:
            return False
        di+=1
        dj+=1
    return True
```

右上到左下斜着：

```
else:#右上到左下斜着
    pi=i-1
    pj=j+1
    while (pi>=0 and pj<=7):
        if cb.board[pi][pj] in ['.','+']:
            return False
        pi-=1
        pj+=1
    di=i+1
    dj=j-1
    while (di<=7 and dj>=0):
        if cb.board[di][dj] in ['.','+']:
            return False
        di+=1
        dj-=1
    return True
```

若在某一方向上满足我方棋子连到边界的条件，则此棋子在此方向上为稳定子。此外，若某棋子在某方向与稳定子相邻，则在此方向上此棋子也安全。具体代码如下：

```
def stablearound(i,j,stable,cb,self):#将稳定子周围自己棋子的稳定度提高
    if i-1>=0 and j-1>=0 and cb.board[i-1][j-1]==self:
        stable[2][i-1][j-i]=1
    elif i-1>=0 and j+1<=7 and cb.board[i-1][j+1]==self:
        stable[3][i-1][j+1]=1
    elif i+1<=7 and j+1<=7 and cb.board[i+1][j+1]==self:
        stable[2][i+1][j+1]=1
    elif i+1<=7 and j-1>=0 and cb.board[i+1][j-1]==self:
        stable[3][i+1][j-1]=1
    elif i-1>=0 and cb.board[i-1][j]==self:
        stable[1][i-1][j]=1
    elif i+1<=7 and cb.board[i+1][j]==self:
        stable[1][i+1][j]=1
    elif j+1<=7 and cb.board[i][j+1]==self:
        stable[0][i][j+1]=1
    elif j-1>=0 and cb.board[i][j-1]==self:
        stable[0][i][j-1]=1
    return stable
```

此方式可以用来判断稳定子，将稳定子周围自己棋子的稳定度提高。

但是，由于时间限制问题，如此精妙的算法在时间限制下不能大显身手，于是我组只能忍痛割爱，在渐终局中不考虑稳定子。

17.2.5程序限制

如果在探索过程中有一个枝节点（存在某一种可能性）使得黑方或白方被全部吃光，则这个枝节点没有子树，无法通过最下面的叶节点向上返回函数值，故该枝节点的函数值始终为 None，可能会出错。

17.3实验结果

17.3.1实验数据

实验环境说明：

- 硬件配置：CPU: Intel Core i5 Mobile ("Sandy Bridge") 内存：4GB
- 操作系统：win8
- Python 版本：2.7.9

1000 次对弈结果数据汇总

黑 / 白	idiot 算法	gambler 算法	本算法
diot 算法	胜局比："0:1000" 总分比："27000:37000" 时间比："9.765756:13.560500"	胜局比："534:466" 总分比："33577:30423" 时间比："12.854464:10.445009"	胜局比："0:1000" 总分比："10000:54000" 时间比："11.03774:29724.08997"
mbler 算法	胜局比："407:593" 总分比："29256:34744" 时间比："10.683399:13.235111"	胜局比："490:510" 总分比："31848:32152" 时间比："12.043067:12.135258"	胜局比："0:1000" 总分比："13443:50557" 时间比："11.81607:45646.319521"
本算法	胜局比："1000:0" 总分比："59000:5000" 时间比："12043.31717:7.91123"	胜局比："1000:0" 总分比："49295:14705" 时间比："29724.08997:11.03774"	胜局比："0:1000" 总分比："53000:11000" 时间比："63782.30918:89200.044

17.3.2结果分析

在与 idiot 和 gambler 的对弈过程中，本算法合理地估值函数为下子的正确决策起了关键作用，取到了全胜的结果。算法的时间在正常范围里，主要的运行时间开销在递归深度上，由于时间复杂度随递归深度指数增长，故深度较高时，耗时明显增大。

17.4实习过程总结

17.4.1分工与合作

1.小组分工：

分配任务，掌控全局：组长李然；

算法的开发以及代码的编制：李然、林芷平、吴梦彤；

测试程序的开发以及测试数据：蒙聪；

撰写实习报告及会议记录：王慧君、王静；

资料查找、思路分析：全体成员。

2.合作与交流的方式：

微信群；

一起自习完成此实习作业；

利用寝室近的优势开小组讨论会。

3.历次组会记录：

6 月 5 日（周五），第一次召开组会，大家热烈地讨论了已有的思路，各抒己见，组长李然分派任务，分为算法开发三人组：李然、林芷平、吴梦彤，测试程序开发一人组：蒙聪，实习报告撰写二人组：王慧君、王静。

6 月 7 日（周日），第二次召开组会，大家讨论了两天来查阅到的资料，有了基本的思

路，决定吴梦彤负责估值函数的实现，李然和林芷平负责开发搜索树，打算运用 alpha-beta 算法实现搜索树。蒙聪的测试程序仍处于开发中，王慧君和王静已撰写实习报告的分工合作等部分模块。组长李然安排第二天的任务工作。

6 月 8 日（周一），第三次召开组会，大家汇报总结了周一一天的进展，吴梦彤负责的估值函数已成型，以期改进，李然和林芷平开发的搜索树已经比较完备，然后把搜索树以及估值函数组合在一起测试效果，可以打败傻瓜算法。蒙聪已编写出测试程序。王慧君和王静已完成实习报告算法的思想部分。

6 月 9 日（周二），第四次召开组会，大家汇报总结了一天的进展，吴梦彤在估值函数里加了稳定子的处理，可以打败赌徒算法，李然和林芷平运用 alpha-beta 算法实现了三层和五层搜索树，蒙聪已经开始着手测试代码，王慧君和王静基本完成实习报告中函数说明部分。大家分析了目前的状况，决定接下来从减少运算时间方面改进算法。

6 月 11 日（周四），第五次召开组会，为了解决超时的问题，吴梦彤十分惋惜地删去了估值函数中的稳定子部分。李然和林芷平运用 alpha-beta 算法实现了六层和七层搜索树，但是二者都不能打败使用五层搜索树的算法。大家讨论决定了第二天热身赛所使用的代码，从已有的几个版本的代码中选择了 T_FOXTROT1.py，T_FOXTROT2.py，T_FOXTROT3.py，T_FOXTROT4.py，T_FOXTROT5.py 参加热身赛。

6 月 13 日（周六），第六次召开组会，大家讨论分析了热身赛的结果，吸取教训，改进得到 T_FOXTROT6_15.py 这个版本的算法。

6 月 15 日（周一），第七次召开组会，大家讨论分析了周日第二次热身赛的结果，综合各方面考虑，最终决定采用 T_FOXTROT1.py 作为正式比赛的算法。

17.4.2经验与教训

认真对待实战，不要轻易放弃。程序相互克制，要合理地选择算法，竞赛不仅是算法之间的竞争，也是一场博弈与心理战。

小组合作的过程中值得改进的地方有很多。主要的问题是树的构建依然比较原始化，没有调整参数可以改变递归层数的算法。在对时间的利用上，除了 ALPHA-BETA 剪枝算法外没有提出更好的节省时间的方法。估值函数中的稳定子估值部分递归耗时太久，最后不得已没有使用稳定子的估值部分。

17.4.3建议与设想

1.对本次实习作业的建议：

关于组队：可以老师指定作业完成情况好的同学担任组长，组员依据学号随机分配给每个小组。

关于基础设施代码：助教老师提供的基础设施代码性能很好。

关于竞赛：竞赛可以适当减省一些时间。

2.对学弟学妹的寄语：

面对大作业，大家不要害怕，切莫手忙脚乱。静下心来认真分析算法的思想策略，可以考虑实现程序的模块化，分小组负责不同的模块，分模块逐一攻克。多查资料，有助于拓展思路。注意及时记录大家的奇思妙想，讨论中迸溅的思维的火花，这是一件非常有意义的事情。

3.对实习作业后续工作的设想：

可以把各个小组的作业编纂成一部书，成为一部黑白棋算法的专业著作，给学弟学妹作为参考。

17.5致谢

首先感谢陈斌老师给了我们完成这次实习作业的机会。从黑白棋的思路、程序编写到实习报告的撰写，陈斌老师帮助我们组解决了很多问题，不厌其烦，认真负责。陈斌老师在课堂上和微信群里对我们的悉心指导和启发关怀，以及对学术、对科研、对人生的态度都给我们留下了深刻的印象。

还要感谢助教老师，不辞辛苦地为我们开发并不断升级 ReversiChecker 程序，可以用人机对弈的形式来检验算法，这种一丝不苟、认真负责的精神给我们留下了深刻的印象。感谢开发可视化平台的同学们，让我们能看到下棋的过程，进一步增强黑白棋的兴趣。感谢竞赛当天的主持人和摄影师，记录了我们之间竞争与合作的美好回忆。

特别感谢我们认真靠谱、积极主动的组员们，感谢组长李然立足全局，让我们的整个实习大作业能流畅地进行下去；感谢吴梦彤、林芷平和李然同学充分发挥聪明才智，实现了智能的黑白棋算法；感谢蒙聪同学不辞辛苦地一遍遍测试程序，感谢王慧君与王静同学每次开会都认认真真做会议记录，并耐心地撰写实习作业报告。大家都非常欣赏并重视这项实习作业，并齐心协力尽自己最大努力把它做好。再次感谢 FOXTROT 各位靠谱的组员们。

17.6参考文献

（列出实习过程中用到的参考资料、网站链接等）

http://home.ustc.edu.cn/~baj/publications/concluding2007-Bai.pdf

http://www.cnblogs.com/speeding/archive/2012/09/20/2694704.html

http://wenku.baidu.com/link?url=uYqu6wzm8Sy--qMc3_R3HX9YKofu3DTH0C5MEK06Oh2suYZFOq8vlnN8FJXMMZNN3IzxNDTCJkhoTrRuryTnv7bJrgNCP0AXMowN4dlY2FC

http://www.soongsky.com/strategy/donggu.php

http://www.soongsky.com/computer/alpha_beta.php


谢谢！

北区 North　JULIET 组

18　数据结构与算法课程实习作业报告

（张沙洲*、仇立松、陆杰、李昆鹏、黄驰琳、仲启蒙）

摘要：本组采用了深度优先遍历的递归思想，采用了 Alpha-Beta 剪枝算法和估值算法。其中 Alpha-Beta 剪枝算法的基础是最大-最小的博弈思想，估值的目的是为了按照一定的比较方法得到最佳方案。涉及的数据结构有列表、字典、树这三种，由于这几种数据结构的性能方法已经能够很好的满足了我组算法的需求，所以并没有使用自定义类或者对其进行扩展。从结果汇总可以看出，本组算法对 idiot 算法可以全胜，对 gambler 算法可以取得压倒性优势。从复盘数据来看，对于 Juliet2 算法而言，棋盘位的设定使得我方尽可能的占据了那些相对重要的位置，避开不利位置；而行动力的设定以及在后期给予其较大的权重使得对方多次 pass，被证明是合理的。对于 Juliet3 算法而言，与 Juliet2 算法最大不同的策略是散度以及边角配置的概念。但由于 gambler 的随机性，即使我方下子较为集中也无法诱使对手下子比较集中，导致效果没有体现出来，但在与其他组算法的对抗中还是有不错的效果的。关于运行时间，在对剪枝深度进行优化后还是比较令人满意的。

关键词：列表、字典、树、Alpha-Beta 剪枝算法、估值算法

1、算法思想：

（1）总体思路：

基本思想是采用递归思想，采用深度优先遍历，即对应一个要下棋的局面，把每种可能下棋的情况都列举出来，并往后推理一定步数，然后按照一定的比较方法，从中找出最优方案（即尽量使己方优势最大化，对方优势最小化）并执行。其具体实现方法主要采用 Alpha-Beta 剪枝算法和估值算法，现具体介绍如下：

Alpha-Beta 剪枝：其分为两部分，一是 Alpha 剪枝，一是 Beta 剪枝。

Alpha 剪枝：Depth0 层为我方(程序方)下棋结点,Depth1 层为对方下棋结点,因此 Depth1 的结点要会选择最小的结点。所以 Depth1 的第一个结点选择下一层最小的分值——即 5 分。当轮到 depth1 层的第二个结点选择时,当其发现有一个小于或等于 5 分的结点时,即当发现 4 分的那个结点时,就直接选用该值。后面的值不用计算了。因为后面的值有两种情况,一是小于等于 5 分的,那么就算被对方选上去,我方在 Depth0 层也不会选择它,因为我方应经有个 5 分保底了,小于等于 5 分的就不用了。二是 x 值大于 5 分的,层为对方选择,那么对方会选择前面的小于等于 5 分的结点(即分值为 4 的结点),而不会选择这个大于 5 分的结点。所以无论值如何,都没有意义。所以可以把该枝剪掉。所以 5 分就是一个下界,即 alpha 值。在 Depth1 层的第三个结点不发生剪枝,那么有一个新的下界产生。如图,当 Depth1 层完成第三个结点搜索时,alpha 值等于。如果 Depth1 层还有第四个结点,那么后面就用“alpha==6”来进行剪枝。(注

意:图中的 Depth0 并不一定是下棋的根结点，可以是博弈树中的一个内部结点。）

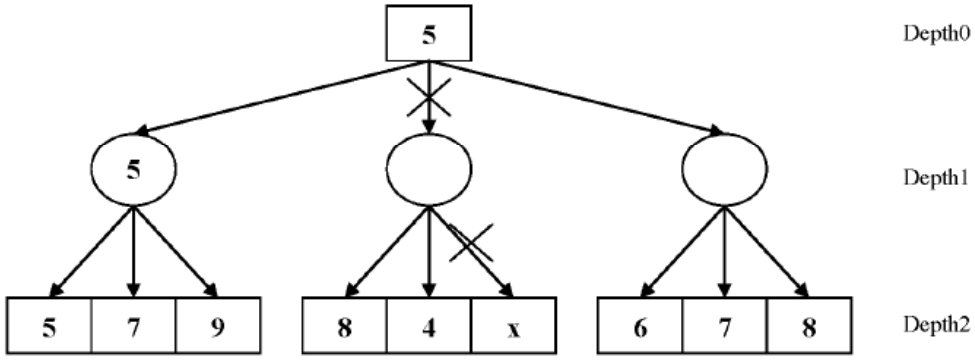


图 1：Alpha 剪枝示意图

Beta 剪枝：Beta 剪枝和 Alpha 剪枝差不多,只不过顶层要选择的结点是对方,而不是我方，而且产生的 beta 值也不是下界，而是上界。那么为 Depth1 层我方,Depth1 的第一个结点会选择最大的 8 分,那么 8 分就形成了一个上界,即 beta 值。那么当的第二个结点搜索到一个大于等于 beta 值 8 分的结点——9 分的结点时,就可以不用计算 9 分后面的结点了。直接返回 9 分。因为如果 9 分后面的结点的值小于等于 8 分,不会被选上(前面已经有个 9 分了),如果大于 8 分,就算选上去 Depth1 层,也不会选上 Depth0 层(因为 Depth0 层是对方选择的,会选择最小值)。所以 beta 值 8 分是一个上界,大于等于 8 分就发生剪枝,直到一个结点不发生剪枝,才产生新的 beta 值(即新的上界)。

刚开始剪枝时,alpha 的初值是负无穷,beta 的初值设为正无穷。从上面的剪枝过程我们可以看到,剪枝过程中,上界和下界是一个不断收敛的过程。

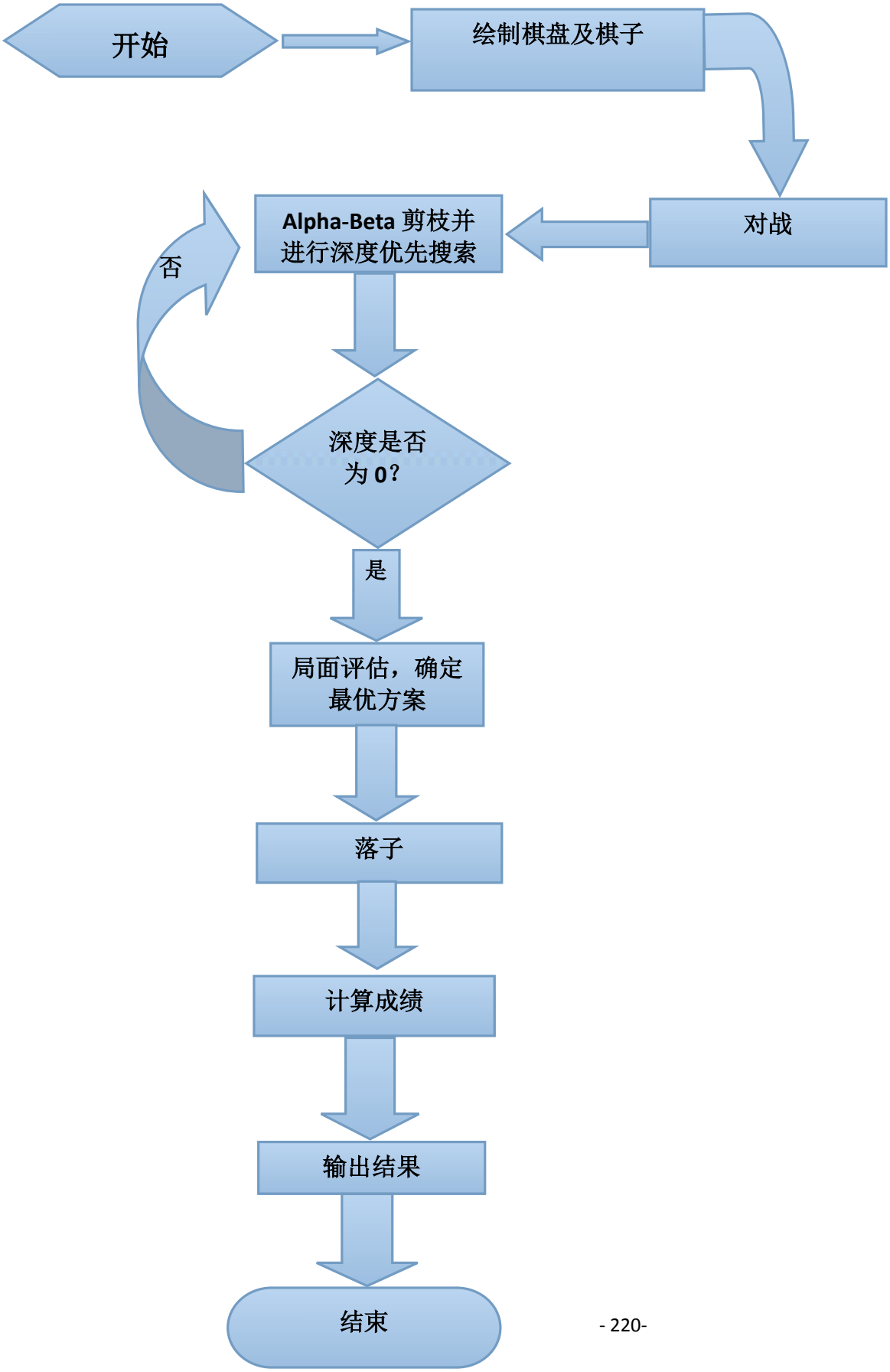
估值算法：估值算法其实就是比较的基础，最优方案就是根据估值结果来得到的。因为整个方案是基于递归的深度优先搜索，也就是说非叶子结点的分值都是从下一层的结点的分值返回。那么,当搜索到叶子结点的时候,我们就需要对叶子结点的局面进行一个评估;进而计算出一个分值,来代表叶子结点的好坏情况。而上层结点的分值将由下层结点返回。最终对每一个局面都能评出分来，这就是估值算法的作用。要对一个局面平分,我们首先要抽象出影响局面优劣的一些因素,之后把这类因素进行合并。在合并前,每个因素的分值乘以一个权重,以便统一分值。局面分值记为 Score,因素(Factor)记为 F,权重(Weight)记为 W,公式如下:

$$Score = F_1 * W_1 + F_2 * W_2 + \ldots\ldots + F_n * W_n$$

$$= \sum_{i=1}^{n} F_i * W_i$$

由于博弈是由我方和敌方构成的,所以公式中的因素为我方的因素减去敌方的因素。当使用了负极大的搜索后,那么,结点的分数都是相对于我方来说的,零分为平分,正数的分值为有优势,负数分值为劣势。优劣的程度就按分值的大小具体来定。

（2）算法流程图


开始
绘制棋盘及棋子
对战
Alpha-Beta 剪枝并
进行深度优先搜索
否
深度是否
为 0？
是
局面评估，确定
最优方案
落子
计算成绩
输出结果
结束

（3）算法运行时间复杂度分析

剪枝算法消耗：O（$b^{\frac{d}{2}}$）其中 b 为分值因子,d 为深度。

估值算法消耗：O（n^2）主要有计算棋盘位、行动力、以及模拟走过这一步后的棋局等（在 Juliet3 版本中又添加了计算边角配置、稳定子、集合度等），但各复杂度都不超过 O（n^2），而且这些都属于并列关系，计算复杂度时直接相加，所以综合起来的复杂度为 O（n^2）。

又因为在每一步中都要进行上述操作，平均为$n^2/2$，综上，总的时间复杂度为O($n^4 b^{\frac{d}{2}}$)

2 程序代码说明

2.1 数据结构说明

本组算法中主要采用了列表、字典、树这三种数据结构，由于这几种数据结构的性能方法已经能够很好的满足了我组算法的需求，所以并没有使用自定义类或者对其进行扩展。本组代码主要实现的是以 alpha-beta 剪枝为核心的一系列函数来实现黑白棋的 AI 计算。实现的主要函数如下。

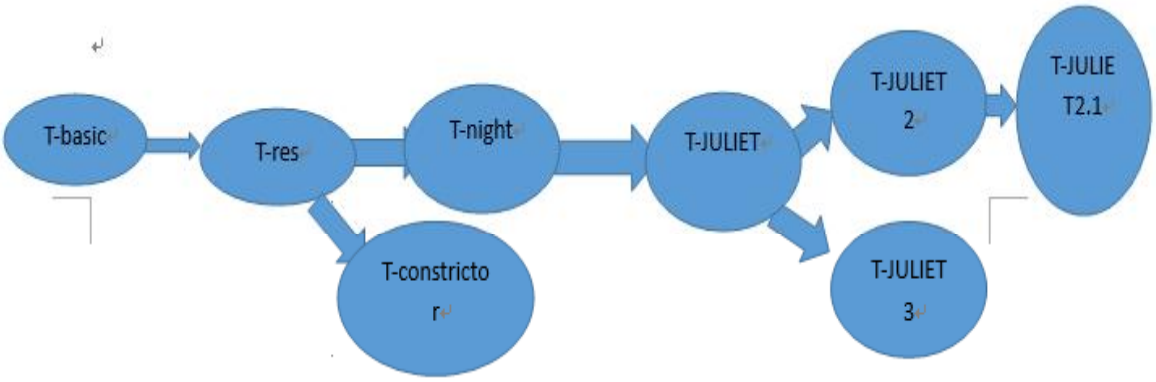
2.2 函数说明

(1) 版本沿革

代码部分由张沙洲和仇立松共同开发。拿到任务后我们立刻写了一个基本算法 T-basic，实现了诸如尽量占边占角，不让对方占角，减少对方的移动力等黑白棋基本方法。我们开会讨论决定要采取深度搜索的方法让代码更有智慧，让算法往后搜索几步找到最优落子点，于是 T-res 出现了。但是 T-res 是一个个让顺序往下搜，搜索很无脑，也很费时，搜索到一个预定的深度后，对局面的分析也仅仅停留在子数，对 gambler 的胜率还是不高。接着我们通过查找资料，发现了一个好的的搜索方法，即 AlphaBeta 剪枝算法。两个程序猿分别开发出了 T-night 和 T-constrictor，

最后我们测试了两个搜索方法，发现 T-night 性能更好，于是采用了 T-night。后来 T-night 做了少量改动，更名 T-JULIET，参加第一次热身大赛。T-JULIET 的奇葩战绩让我们很吃惊，要么将对手直接 KO，要么就是自己超时。这可能是因为第一次热身一些组的代码都不太成熟，然后被我们 KO 了。二是我们高估了服务器的性能，大大加大了搜索深度，因此超时严重。同时 T-JULIET 的估值不太好，下不赢老师给的黑白棋网页的中级，这让我们很头疼。因此针对 T-JULIET 的超时以及估值问题，我们在 T-JULIET 的基础上写了 JULIET2，后来又做了部分改进，化身 T-JULIET2.1。T-JULIET2.1 能打败网页黑白棋上级。此外参加第二次热身大赛前我们又基于另一种估值思路写了 T-JULIET3，之后的测试发现 JULIET2.1 要强于 JULIET3，但

是在热身大赛中，JULIET2.1 会输给部分“敌人”，而这时 JULIET3 却能出奇制胜。因此我们对 JULIET2.1 和 JULIET3 做了少量修改后，派上了黑白棋决赛的战场！



下面主要介绍 JULIET2.1 和 JULIET3 的主要函数。

（2）JULIET2.1 内置函数：

总体思路为通过计算总评分 TotalVal 来判断最佳下子位置，详细情况参见 getTotalVal 函数。

InitValueBoard：“初始棋盘位置价值”，本函数无参数，其作用为生成一个 8*8 双重列表。在黑白棋的游戏中，棋子位于不同的位置，其价值亦有不同。我组通过反复的人机对战和复盘数据分析，经过多次改动和微调，为棋盘上的每一个位置确定了一个较为合理的价值参数。列表中的每个位置存放的就是对应位置的价值数。它们被用来对棋局总体情况进行分析。

changeValBoard：“变换棋盘位置价值”，本函数无参数，作用与 InitValueBoard 相同，但随着棋局的进行，后期同一位置上的棋子的价值可能会高于或低于前中期，所以每个位置的价值参数有所调整，以更好的分析棋局后期的棋局总体情况。所以我们通过多次测试大致为棋局后期棋盘上的每一个位置确定了另一个较为合理的价值参数，因此在棋局后期我们的估值函数用的是 changeValBoard。

getOpTurn：参数为当前下子方，返还下一步的下子方。

getPosValue：参数为当前棋盘和当前下子方。这个函数被用来计算“棋盘位”，即当前棋局局面的一种量化衡量方式。函数会遍历当前棋盘的每一个位置，如果该位置为我方棋子，棋盘位参数 posvalue 就加上该位置对应价值；反之如果是对方棋子， posvalue 就减去该位置价值。最后返回 posvalue。棋盘位越大，形式对我方就更有利。

getMobility：“计算行动力”，参数为当前棋盘和当前下子方。这个函数被用来计算“行动力”，即我方相对于对方所掌握的主动权。显然，我方可以下子的位置越多，对方可以下子的位置越少，甚至是无子可下，我方越对战局拥有主动权。函数会计算当前我方可以下子的位置数，减去我方 pass 后对方可以下子的下子数，这个差即为行动力 Mobility。如果当前下子方为我方，返回 Mobility 本身；如果是对方，返回-Mobility。

getTotalVal：“总价值”，参数为当前棋盘和当前下子方。显然，不管是 posvalue 还是 Mobility 都无法全面的反映当前局面评分，所以需要将二者整合，给予二者各自的权重，然后将其合并为一个总评分。在测试中我们发现随着棋局的进行，二者的重要性并不是一成不变的。比如在前期，不管我们怎么下子，对方都有较多的位置可以下子，行动力的重要性并不显著；但在后期，如果对方的下子位置较少，很有可能被逼无奈的在非常不利于自己的位子上落子，甚至直接 Pass。经过反复的实验调整，我们小组得出了现有这组动态的权重比例。权重会随

比赛进行而变化。经复盘数据验证，总评分 TotalVal 可以较好地反映当前局势情况。为后面的 alpha-beta 剪枝函数提供了一个很好的评估指标。

getFinal：参数为当前棋盘和当前下子方。与 getTotalVal 作用相同，但只在 alpha-beta 搜索达到了结局才被调用，此时的总评分被定义为我方与对方子数之差。这是因为在后期临近比赛结束时，搜索可以直接搜索到结局的状况，此时直接比较子数差就好了（为了争取最大规模胜利）。

Simboard：“虚拟棋盘”，参数为当前棋盘和下子位置，返回下子后的棋盘。这是为了防止直接在原来的 cb 里 makeTurn 而破坏 cb，而将 cb 复制给 imcbBoard,在 imcbBoard 里落子（makeTurn）。

Alphabetasearch：“AlphaBeta 剪枝搜索”，主体函数，大部分函数都是为了实现它的功能而设计的。参数为搜索深度、当前棋盘、alpha 值、beta 值、当前下子方。通过 Alpha-beta 剪枝的搜索原理，寻找在当前的可下子位置中下哪个位置，在经历我们设定的深度（步数）后可以实现对我方最有利（总评分最大）的局面。在深度为 0 时返回总评分 TotalVal 和推荐位置 None（列表形式，None 表示无法下子），在搜索到棋局结束时调用 getFinal 返回总评分 TotalVal 和推荐位置 None，在 Pass 时返回总评分 TotalVal 和推荐位置 Pass，否则我方为当前下子方时返回 alpha 和推荐位置坐标，对方为当前下子方时返回 beta 和推荐位置坐标。

Run：参数为搜索深度、棋盘、字典 ms。判断在不同情况下我方应该采取的策略。无子可下时直接返回 Pass，唯一位置时直接返回该位置，如果可以 KO 对手时直接一步 KO，其他情况则调用 Alphabetasearch 函数以得到最优解。考虑到 Alphabetasearch 函数较为耗时，这样做的目的是避免每次都调用 Alphabetasearch 函数，提高算法效率。

Play：参数为棋盘、字典 ms。作用为判断当前阶段以在调用 Run 函数时给予不同的搜索深度、在调用 getTotalVal 函数时给予“棋盘位”和“行动力“不同的权重。如前中期（棋子数小于 46）的深度为 4，后期的深度为 6。这样做的目的是减少前期一些意义不大的过深搜索，减少算法运行时间，提高效率。

（3）JULIET3 内置函数：

总思路为前期通过散度，中后期通过总评分确定最佳下子位置。详细情况参见 Focus 函数和 getTotalVal 函数。

InitValueBoard：“初始棋盘位置价值”，本函数无参数，其作用为生成一个 8*8 双重列表。在黑白棋的游戏中，棋子位于不同的位置，其价值亦有不同。我组通过反复的人机对战和复盘数据分析，经过多次改动和微调，为棋盘上的每一个位置确定了一个较为合理的价值参数。列表中的每个位置存放的就是对应位置的价值数。它们被用来对棋局总体情况进行分析。

getOpTurn： 参数为当前下子方，返还下一步的下子方。

getPosValue： 参数为当前棋盘和当前下子方。这个函数被用来计算“棋盘位”，即当前棋局局面的一种量化衡量方式。函数会遍历当前棋盘的每一个位置，如果该位置为我方棋子，棋盘位参数 posvalue 就加上该位置对应价值；反之如果是对方棋子， posvalue 就减去该位置价值。最后返回 posvalue。棋盘位越大，形式对我方就更有利。

getMobility：“计算行动力”，参数为当前棋盘和当前下子方。这个函数被用来计算“行动力”，即我方相对于对方所掌握的主动权。显然，我方可以下子的位置越多，对方可以下子的位置越少，甚至是无子可下，我方越对战局拥有主动权。函数会计算当前我方可以下子的位置数，

减去我方 pass 后对方可以下子的下子数，这个差即为行动力 Mobility。如果当前下子方为我方，返回 Mobility 本身；如果是对方，返回-Mobility。

getSolidVal： 参数为当前棋盘和当前下子方。意为“稳定子价值”，即稳定子的存在对于棋局的加分。如果我方的稳定子越多，形式自然越有利。在中期影响较大。

getSAVal： 参数为当前棋盘和当前下子方。意为“边角配置”，即边角的搭配对棋局的影响。如与角相邻的格点如果是我方棋子，那就是一个稳定子，可以极大地威胁到与其同行、纵、斜的对手棋子。在后期影响较大。

getTotalVal： “总价值”，参数为当前棋盘、当前下子方和 tin（作用是告诉 getTotalVal 函数当前处于中期还是后期）。显然，不管是 posvalue 还是 Mobility 都无法全面的反映当前局面评分，所以需要将二者整合，并加入“稳定子价值”和“边角配置”的影响，给予他们各自的权重，然后将其合并为一个总评分。经过反复的实验调整，我们小组得出了现有这组动态的权重比例。“稳定子价值”只在中期出现而“边角配置”在后期出现。经复盘数据验证，总评分 TotalVal 可以较好地反映当前局势情况。为后面的 alpha-beta 剪枝函数提供了一个很好的评估指标。

getFinal： 参数为当前棋盘和当前下子方。与 getTotalVal 作用相同，但只在 alpha-beta 搜索达到了结局才被调用，此时的总评分被定义为我方与对方子数之差。

Simboard： “模拟棋盘”，参数为当前棋盘和下子位置，返回下子后的棋盘。

Focus: “集中度”，也称“散度”。主体函数之一，参数为当前棋盘、当前下子方。前期判断下子位置的方法。首先判断是否能让对方 Pass,如果可以，直接下在该位置。接下来是能否占角，能则直接下。如果会让对手占四角，则将这种走法移出可以下子的列表（除非移除后列表为空）。然后就是概念“散度”的引入。对于我方的每一个棋子，在其周围的格点中，如果有一个空点，散度+1。Focus 函数会返回使总体散度最小的下棋位置。这样做的目的是使我方棋子在前期尽可能的保持集中。实验证明这样在后期有利于形成稳定子群，减少对方可下子位置，提高行动力。

Alphabetasearch： “Alpha-beta 剪枝搜索”，主体函数之一，大部分函数都是为了实现它的功能而设计的。参数为搜索深度、当前棋盘、alpha 值、beta 值、当前下子方、tin（因为会调用 getTotalVal 函数）。中后期判断下子位置的方法。通过 Alpha-beta 剪枝的搜索原理，寻找在当前的可下子位置中下哪个位置，在经历我们设定的深度（步数）后可以实现对我方最有利（总评分最大）的局面。在深度为 0 时返回总评分 TotalVal 和推荐位置 None（以列表形式，None 表示无法下子），在搜索到棋局结束时调用 getFinal 返回总评分 TotalVal 和推荐位置 None，在 Pass 时返回总评分 TotalVal 和推荐位置 Pass，否则我方为当前下子方时返回 alpha 和推荐位置坐标，对方为当前下子方时返回 beta 和推荐位置坐标。

Run： 参数为搜索深度、棋盘、字典 ms。判断在不同情况下我方应该采取的策略。无子可下时直接返回 Pass，唯一位置时直接返回该位置，其他情况则调用 Alphabetasearch 函数以

得到最优解。考虑到 Alphabetasearch 函数较为耗时，这样做的目的是避免每次都调用 Alphabetasearch 函数，提高算法效率。

Play： 参数为棋盘、字典 ms。作用为判断当前阶段，在前期（子数小于 30）调用 Focus 函数，中期（子数小于 49）调用 Run 函数并令搜索深度为 2（这样做的目的是减少中期一些意义不大的过深搜索，减少算法运行时间，提高效率），后期（子数大于 49）调用 Run 函数并令搜索深度为 6。

2.3 程序限制：

我组现版本中的函数在迄今运行中均未出现报错等情况，但在早期版本中，某些极端情况下可能会报错。如 Juliet3 中的 Focus 函数在某种走法会让对手占四角的情况下将这种走法移出可以下子的列表，所以可能会发生列表为空又没有返回 Pass 的情况而报错。造成这种情况的原因是 focus 里用 TemlegPos=legPos 来保存移除下子位前的列表，但由于 python 里列表 B=列表 A，结果 A 列表变化时 B 也随之变化，因此最后 TemlegPos 也随之变化为空集，失去了 TemlegPos 备份的意义。
经过本组同学细心和认真的调试，改写了 focus 函数，不直接删除落子位了，这些 bug 的情况都已经得到修复。但在与 gamble 对战过程中发现超时现象经常发生。所以，我组同学专门对算法的部分程序进行修改，努力使每一个函数的代码简化到不能再简，同时反复测试不同的搜索深度，并巧妙使用分段深度的方法，既保证了 AI 具有了较高的智能，有大大减少了运行时间。

3.实验结果：

3.1 实验数据：

硬件配置：（CPU/内存）i5 处理器　4G(1600MHz)

操作系统：（名称/版本）window8.1(64 位)

Python 版本：（版本号）2.7.9

1000 次对弈结果数据汇总（具体每盘的数据和经典棋局见附件）

黑 / 白	idiot 算法	gambler 算法	Juliet2 算法	Juliet3 算法
idiot 算法	0:1000 27000:37000	396:576:28 （平）	1000：0 5200:12000	1000:0 40000:23000

	33：46	29366:34464 36:44	91700:14	15461:16
gambler 算法	519:444:37 （平） 33178:30428 36:30	425:535:40 （平） 30811:33186 39:39	958:32:10 （平） 44104:19589 145000:40	923:66:11（平） 46624:17204 25742:11
Juliet2 算法	0:1000 16000:48000 16:145001	30：958:12 （平） 18384:45616 40:139000	1000:0 50000:14000 86177:11500 0	0:1000 9000:54000 11000:49000
Juliet3 算法	0:1000 14000:50000 11:44000	30:950:20 （平） 14141:49724 20:59300	0：10000 24000：40000 40424:10106	33:967:15 16134:46103 15047:12883

3.2 结果分析：

从结果汇总可以看出，本组算法对 idiot 算法可以全胜，对 gambler 算法可以取得压倒性优势。从复盘数据来看，对于 Juliet2 算法而言，棋盘位的设定使得我方尽可能的占据了那些相对重要的位置，避开不利位置；而行动力的设定以及在后期给予其较大的权重使得对方多次 pass，被证明是合理的。对于 Juliet3 算法而言，与 Juliet2 算法最大不同的策略是散度的概念。但由于 gambler 的随机性，即使我方下子较为集中也无法诱使对手下子比较集中，导致效果没有体现出来，但在与其他组算法的对抗中还是有不错的效果的。对于运行时间还是比较满意的，因为我组的早期算法经常出现超时情况，而主要的时间消耗就是 alpha-beta 剪

枝算法迭代所需时间。所以我组进行了反复优化，在 Juliet2 算法中列举多种不需要迭代的情况，且前期迭代深度较小；在 Juliet3 算法中只在中后期应用 alpha-beta 剪枝算法。现在 Juliet2 算法耗时约为 140 秒左右，而 Juliet3 算法则在 100 秒左右。

4、实习过程总结

4.1 分工与合作

本组同学在确定好分组后，在当天晚上（6 月 3 日晚）就在组长张沙洲的带领下于二教讨论区举行第一次小组聚会讨论并确定分工，具体分工如下：

查找资料：仲启蒙

编写程序：仇立松、张沙州

数据测试：陆杰

程序报告：黄驰琳、李昆鹏

为了更好地完成任务，保证大家都能及时地共享信息、交流思路、解决问题，本组同学建立了多元化的交流途径，主要分线上交流以及线下交流两部分：

- 线上交流：组长张沙洲建立了小组讨论微信群，所有组员均加入并保证每天二十四小时登录微信，时刻关注微信信息，随时联系并发资料、交流问题。小组成员找到一些黑白棋相关资料后均发至微信群中供大家下载参考，小组成员完成各自任务后也会将任务成果如：代码、测试数据、代码及问题分析等发至微信群供大家查看讨论。
- 线下交流：从第一次小组聚会为起始，每隔三到四天组织一次全员的或部分的小组成员聚会，讨论这几天出现的一些问题及其解决方案（主要是一些线上说不清楚的，一些线上能说明白的通常就在线上解决）。同时部分组员不定期自行聚会共同完成一些任务，如张沙洲与仇立松同学曾多次自行聚会讨论关于编写代码的问题。

另外，由于本小组同学大多原本就熟识，居住在同一宿舍楼同层，相距不远，所以小组成员经常有问题就直接互相去寝室拜访解决，这也是小组交流的主要方式之一。

下附本组历次组会记录：

第一次小组讨论

时间：　2015 年 6 月 3 日星期五　　8:00　p.m.

参与人：　黄驰琳、李昆鹏、陆杰、仇立松、张沙州、仲启蒙

内容：

讨论小组分工情况：

查找资料：仲启蒙

编写程序：仇立松、张沙州

数据测试：陆杰

程序报告：黄驰琳、李昆鹏

（分工与合作并存）

分析现有代码中深度搜索存在的 bug
昨天晚上我们对代码进行了一些改动，今天测试的时候发现算法并没有按照预先的设定运行。我们检查了程序中存在的错误。
讨论一些极限情况中的行为。
一般情况下，我们不应该将子落在（2，2）这个点，然而我们在某些特殊情况下是可以采用这种落子方式的。同样存在着其他的极限情况，我们对特殊情况进行了探讨归纳，并且分析了解决方法。
确定用 alpha-beta 剪枝和估值算法
经过分析查找所得的资料，我们确定了使用 alpha-beta 剪枝和估值算法，具体程序仍未得到实现。
第二次小组讨论记录
时间：2015 年 6 月 6 日
参与人：陆杰、仇立松、张沙州
内容：
这次讨论主要是商量怎样实现 AlphaBeta 搜索算法，因此只召集参与代码编写与测试的同学前来讨论。在认真讨论并学习了我们收集到的 AlphaBeta 搜索算法的思路后，我们决定由两位主要负责编写代码的同学各自写一份代码，然后比较性能。规定在 10 号之前写好代码，方便之后进行改进。
第三次小组讨论记录
时间：　　2015 年 6 月 9 日
参与人：　　黄驰琳、李昆鹏、陆杰、仇立松、张沙州、仲启蒙
内容：
1、两位主要负责编写代码的同学各自编出了两套不同的 alpha-beta 剪枝估值算法，分别是张沙州同学编写的和仇立松同学编写的代码。由于我们的当时已经搞定了基础的 alpha-beta 剪枝代码，所以程序的优化就主要寄托在更加先进高效的估值算法上，所以，我们两位同学根据不同的算法思路设计了两种不同的代码，并在今天小组讨论时进行代码的测试，以此来考量算法的优劣。
2、进行代码的测试
代码的测试主要由陆杰同学负责，在分别运行这两套算法之后，发现这样一种情况：在测试场数仍然不算太大的情况下，两者的胜率也相近，无法说清优劣。可其中仇立松同学的代码相对而言就耗时较短，这对于我们第一次热身赛的超时的解决是非常关键的。
3、采取相关措施
在以后的 juliet 版本中，我们打算都将采用这次的代码为估值代码的原型，以后仍然会加以改进。并且我们将继续我们的测试以获得更多的数据，从而通过改变相关的参数如递归深度，行动力等来实现代码的优化。
第四次小组讨论
时间：　　2015 年 6 月 13 日星期六
参与人：　　黄驰琳、李昆鹏、陆杰、仇立松、张沙州、仲启蒙
内容：
1、查看对战情况以及复盘数据
前一天晚上我们参加了热身赛，今天举行小组讨论分析反馈的复盘数据。从对战结果来看，首先就是我们小组的算法超时的现象比较多，仅仅因为超时造成的失败达到 41 场次。

其中我方持黑（先手）的时候超时 21 次，我方持白（后手）的时候超时 20 次。胜率方面，我方持黑时总共的 41 场次里赢了 15 场，平一场，输 25 场；我方持白时，总共 41 场次里赢了 11 场，输 30 场。(其中我们将超时的情况算作失败)。如果将超时的比赛进行到底，我方先手情况下将胜利 33 场，后手情况下将胜利 23 场次。

我们分析了比赛数据，得出了一些结论：

I. 首先得解决我们的超时问题，否则胜率会大大降低。

II. 即使将超时的比赛进行到底，我们的胜率依然不是很高，所以需要提升算法的性能。

还有就是我们获得了比赛时的复盘数据，分析每一局中的超时步骤，将其上一步的棋盘用列表记录下来，同时记录下我们的算法所采取的步骤，用于分析。

2、针对此次热身赛情况，我们计划采用更优化算法的同时降低程序的时间复杂度。主要是考虑减少递归深度，并且改变估值算法来实现以上目标。

3、具体措施

主要就是优化估值算法和一定程度上减少递归深度。我们原来的算法主要是依靠位置估值，移动力等核心概念来决定棋子的去向。所谓的位置估值就是：根据棋子在棋盘上某一位置处拥有的价值，对该坐标进行赋值，价值高的赋值也高。后来我们改动了算法，增加了聚集度的概念，并且在一局棋的前期采用聚集度进行判定，而后期才采用 alpha-beta 剪枝估值。考虑到递归深度对于时间复杂度的影响，我们将递归深度降低了一步，以此来降低算法的时间复杂度。

4.2 经验与教训

本次小组实习过程也并非一帆风顺，也曾面临许多问题，比较重大的有编程过程中面临的两个问题：其一为对战过程超时问题，本组最初的代码由于过于追求胜率，在深度搜索中将深度设置的很深，结果导致时间消耗过大，在小组自行测试以及热身赛中都出现明显的超时问题，后来通过降低递归搜索深度将时间消耗降下来，虽然局面判断可能没有以前那么精准，但却成功地将时间消耗控制住，最后的整体效果反而更好，这启示我们要把握好各方面限制的平衡，不能过于注重某一方面而忽略其他方面；其二为因估值算法不完备导致使用 Alpha-Beta 剪枝算法时效果不好，后来通过更多的数据分析及搜集并参考资料将该算法进行升级，在一定程度上缓解、解决了该问题，甚至最终版本可以下赢老师给的链接中的上级难度。这启示我们要注意到各算法的内在联系并尽量提升自己的专业素养，以后面对此类问题才能做得更好。此外，在 Alpha-Beta 算法中如果可以采用散列表来进行存储，可以在相当大程度内降低时间复杂度，但由于该实现有些复杂，后期时间也比较紧迫，最终并未实现该改进，这是本组比较遗憾的地方，日后还可以改进。

除了代码编写方面的经验教训及可改进之处，在小组活动方面也有许多。比如在小组聚会方面，起初有一次决定开会，结果等部分同学抵达后发现有几位同学在开会时间有其他事，产生了冲突，结果会议效果并不好，自那以后，但凡要讨论，都提前询问每位同学的方便的时间并进行灵活调整，效果好了许多。这启示我们要从实际情况出发，准备充分。

在小组交流过程中比较好的就是本组同学建立了微信群作为联系方式，很多事情可以通过微信群来解决，很多信息和资料得以及时地交流，灵活强大。同时成员之间互相督促共同合作，避免了拖延，这些都是值得后来者借鉴的方式。

4.3 建议与设想

本次实习过程整体的流程、各部分安排都是相当完善的，但在具体实行过程中还是有一些可以改进，具体如下：

组队方面：本次组队为追求各小组实力更均衡，采用根据平时成绩划分组长，然后小组成员自愿报名与组长挑选相结合，但其实有些技术水平非常高的同学可能平时并不追求 A，只是实现了作业要求的基本功能，而不去追求美观等，这样造成他们平时成绩也许不高，没被选上组长，还有一些大神被选上组长后推掉，这样就造成了许多大神组合，在能力上压制其他组，建议以后可以在挑选组长时参考一下同学意见，也许会更好。

基础设施代码很全面，感觉很好；赛制方面感觉在时间的调控方面可以更进一步，比如说将排名通过设计程序列出，而不是人工计算，这样可以省去许多时间。

实习作业在此就结束了，但本次实习作业只是完成了主要下棋算法，希望日后能真正地完成一次游戏设计，做出图形界面。而且，本次实现的算法还比较初级，后续有机会还可以进一步完善。

在此，本组同学献上对学弟学妹的寄语：数算课很有趣，但想做好还是需要下一番苦功的，希望学弟学妹们可以耐下性子，发掘枯燥程序中的趣味，当你认真走过，等到结果后，你将发现，一切努力都是值得的。

5、致谢

在此感谢为本次实习工作在幕后默默付出的陈老师和石瀚文助教。感谢陈斌老师别出心裁地设计这次“黑白棋大赛”以及为本次实习工作提供的基础设施代码，石瀚文助教提供的人机对战程序，阎述辰同学提供的图形界面。这些设施在本组的代码编写以及后续的自我检测过程中发挥了重要作用。

此外，感谢本组的两位编写代码的同学：组长张沙洲和仇立松同学，两位同学在小组工作上投入了大量的时间和精力，还多次刷夜改代码、找 Bug，为小组实习的完满完成做出了突出贡献。

最后，感谢本小组全体成员的积极参与、热心讨论以及对分配任务的出色完成，小组实习的成果离不开每位同学的努力。

在此对以上所有对本小组实习过程有贡献的人和事致以最诚挚的谢意。

6、参考文献：

（1）基于改进博弈树的黑白棋设计与实现_李小舟

（2）Invent Your Own Computer Games with Python

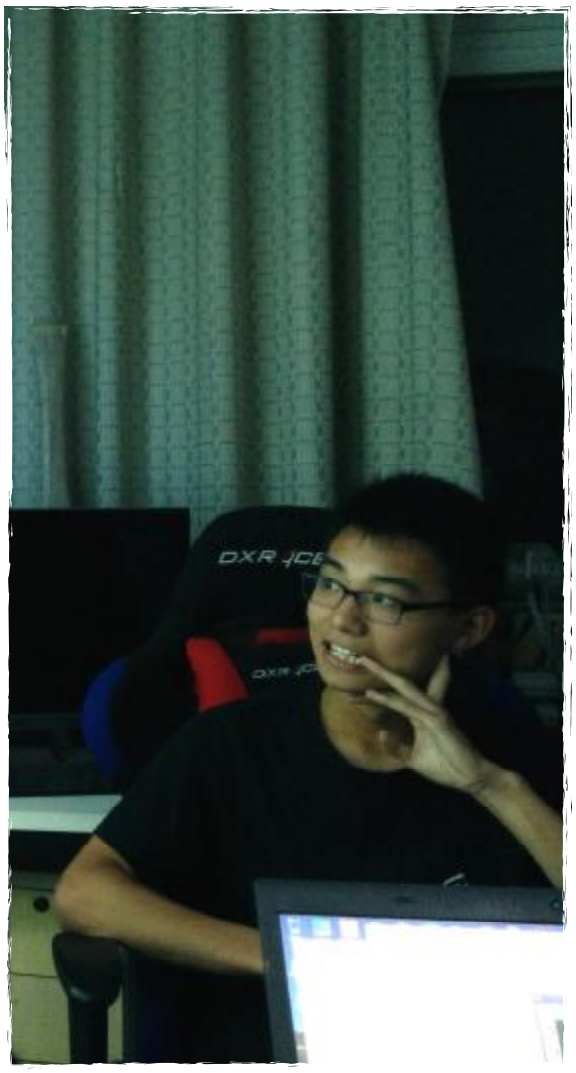
（3）网上链接：

http://www.xqbase.com/computer/search_alphabeta.htm

http://blog.sina.com.cn/s/blog_45cb1ae50101069b.html

CHAPTER

3



竞赛篇

>>>热身赛

>>>小组赛

>>>八强淘汰赛

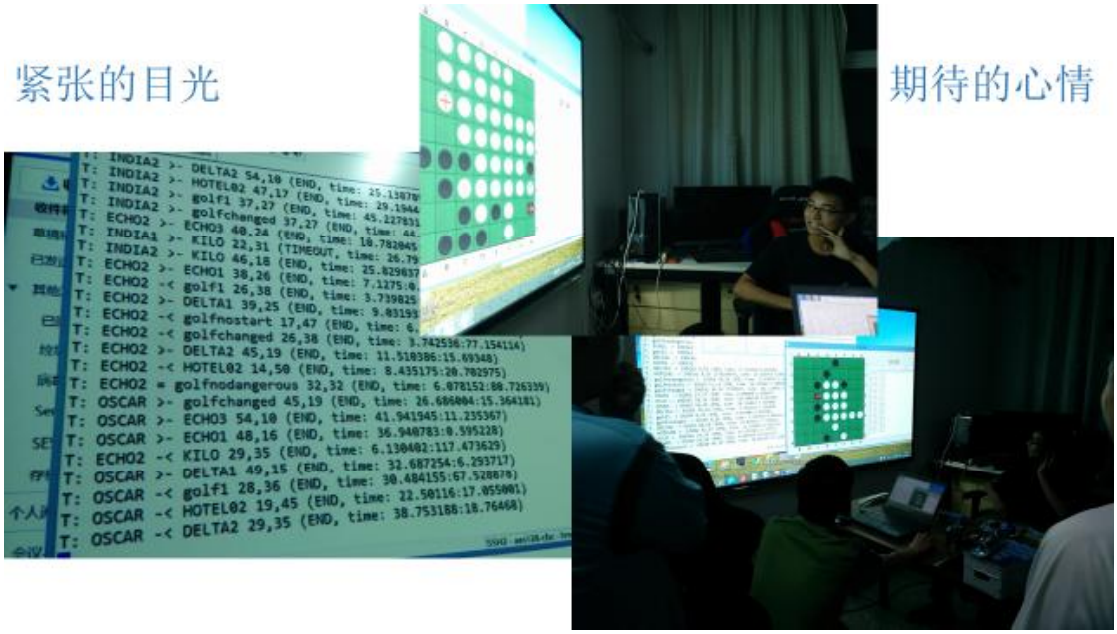
>>>半决赛及决赛

竞赛篇

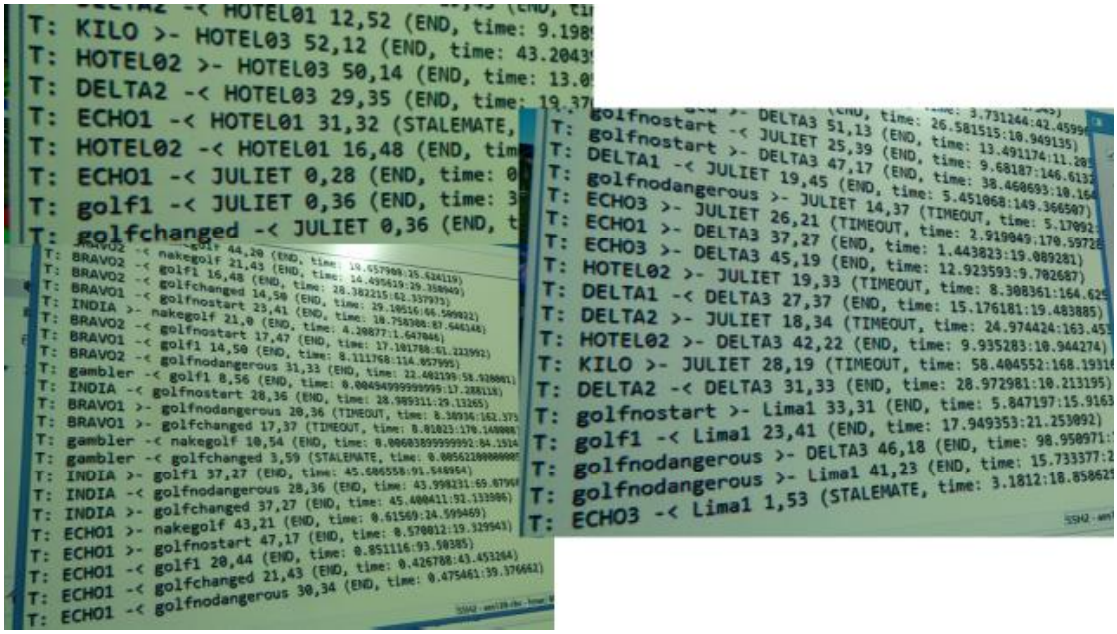
在前期的热身赛中同学们对自己的算法实力有了初步的估计，后积极参与竞赛，并不断改进自己的代码，争取取得较高的名次，传说中前三强将有机会获得神秘奖品。

热身赛

正式比赛前为了检验自己代码的威力和摸清对手的实力，同学们积极报名参加了热身赛。当然，为防止本组机密外泄，热身赛在遥感楼 409 进行。



实验室电脑不眠不休 run 整晚，终于将所有小组的对战情况全部输出。辛苦了，电脑君^_^



热身赛的结果并不能真是反应各小组的真实水平。个别小组采取了隐藏实力的做法，使用弱智代码进行热身比赛，可谓用心良苦.....

各小组同学针对自己代码在热身赛中的表现又开始紧锣密鼓的升级工作，代码 2.0 开始面世！

赛前准备：

场务志愿者之解说名嘴——李子涵（左）、赵玖桐（右）

场务志愿者之数据统计核对——汪建峰（左）、刘松吟（右）

场务志愿者之摄影师——张子玄（假装这里有图）

小组赛

——2015 年 6 月 15 日下午　二教 205

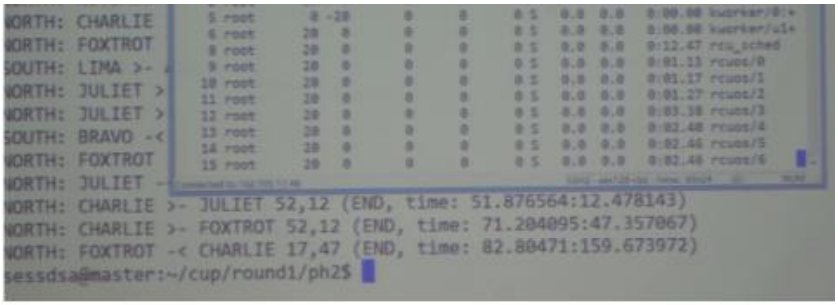
各小组提交代码到服务器，比赛开始。在紧张的对决过程中，同学门有说有笑，畅谈实习作业过程。

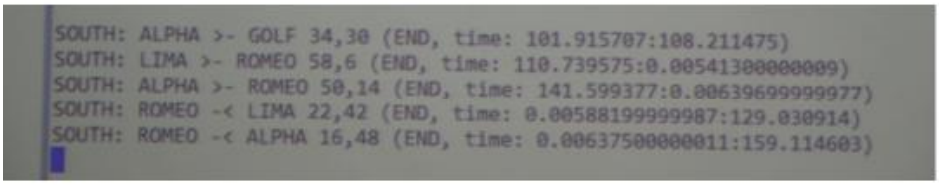
“放狠话”环节

密谋中。。。

对局过程中出现了两场险超时对局，可是吓坏了险胜的小伙伴们

险超时经典棋局





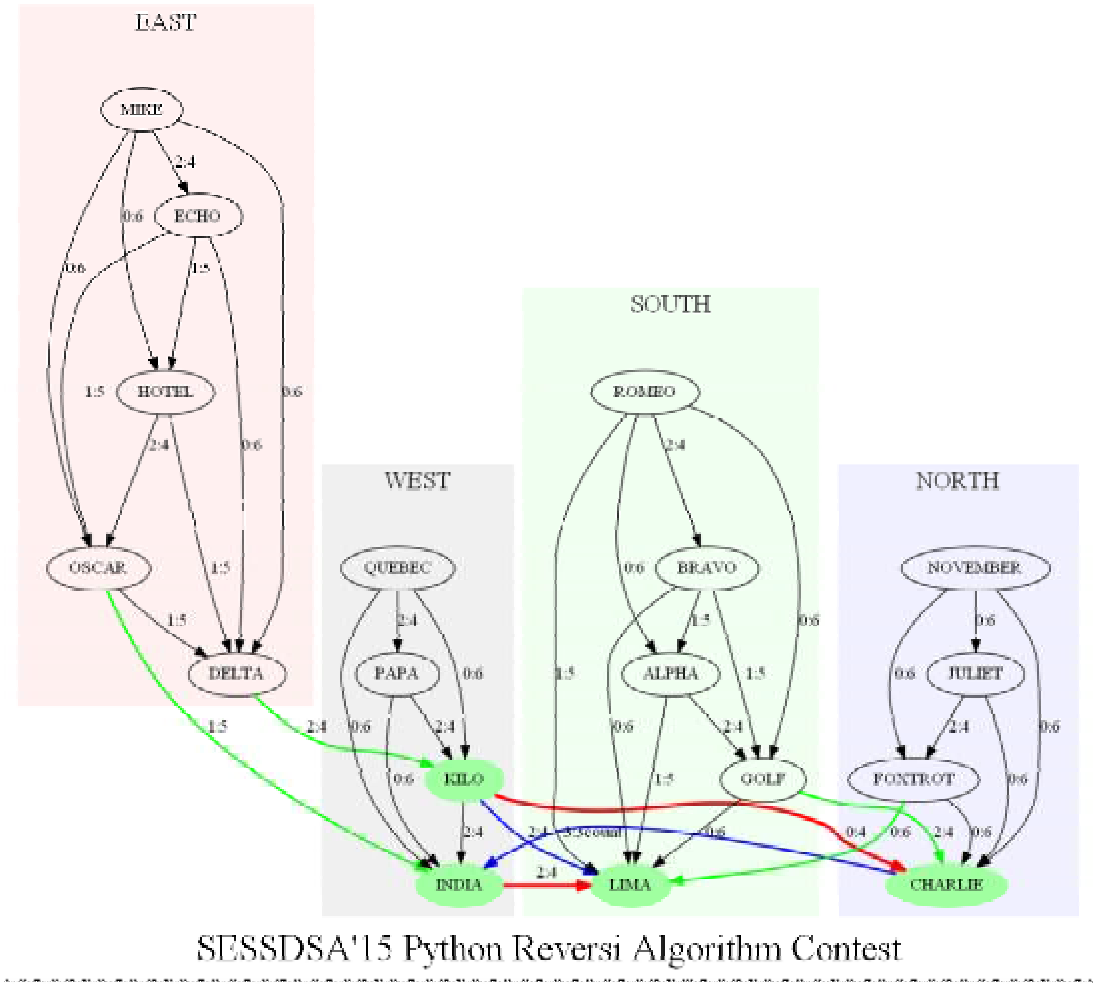
经过第一轮激烈的角逐，八强已然诞生。

东区 OSCAR　　DELTA

西区 INDIA　　KILO

南区　GOIL　　LIMA

北区 CHAELIE　FOXTROL



小组赛和八强淘汰赛对战顺序及得分

八强淘汰赛

——2015 年 6 月 15 日下午　二教 205

随着小组出线名单的产生，八强队伍并没有过多休整的时间。为了赢得胜利，各小组或是快速而准确地修改参数，或是采用提前准备好的升级版代码。

在紧张的等待中，四强名单出炉！

八强淘汰赛

小组	第一场	第二场	第三场	结果
E1:DELTA	1:1	0:2	1:1	
W2:KILO				☺
E2:OSCAR	0:2	1:1	0:2	
W1:INDIA				☺
S1:LIMA	2:0	2:0	2:0	☺
N2:FOXTROT				
S2:GOLF	1:1	1:1	0:2	
N1:CHARLIE				☺

半决赛及决赛

——2015 年 6 月 18 日上午　二教 205

四强奖品

图中的笔筒为 3D 打印品（6 小时/个），兼具收藏价值的同时具备实用价值，也是此次黑白棋比赛最好的纪念品。经过三天的休整，四强选手也根据对手代码在比赛中的表现进行了相应的优化。

首先是 KILO 组与 LIMA 组对决，在第一场以 1 : 1 打平，令人惊奇的是第一场 LIMA 组以 0:19 被 KO，但在随后的两场里，均是 LIMA 组以 2:0 取胜，提前取得进入决赛的资格。

在 INDIA 组与 CHARLIE 组的对决中，出现了令人惊奇的胜负相同的情况，只好进入数子环节。

惊险半决赛：连续平局进入数子环节

最终 INDIA 组以 26 子数赢得冠亚军争夺的资格，CHARLIE 组惜败。

CHARLIE 组　　V S　　KILO 组（季军争夺战）

然而出乎意料地，CHARLIE 组并没有给 KILO 组喘息的机会，直接两场 2 : 0 提前锁定胜局，赢得季军。

CHARLIE提前夺取了季军，KILO屈居第四

然而 KILO 组获得奖励为奥利奥饼干一箱。。。。。一箱！

然而无私的 KILO 小组将其所获奖品与在座同学分享～～～大家好才是真的好！

INDIA 组　　V S　　LIMA 组（冠军争夺战）

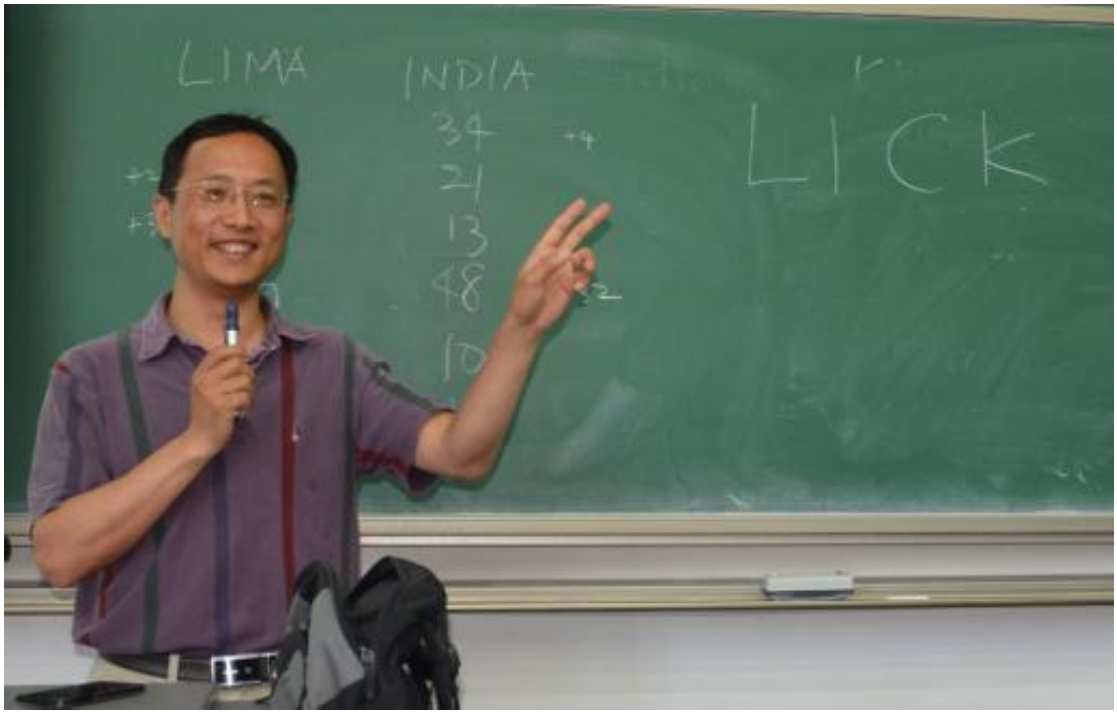
赛前名嘴们随机要求在座同学发表自己的结果预测，在情感上大部分倾向于 INDIA 组（毕竟同院情），但部分同学仍旧客观地指出 LIMA 组实力强大，有望成为冠军。

比赛过程中，采用了可视化程序清晰展现比赛过程，并由专人讲解。

在前两场平局的情况下，第三局 INDIA 组要求更改换代码，第三场 INDIA 以 0:2 落败，最终由 LIMA 组赢得冠军，INDIA 屈居亚军。

然而还有最后的一幅图

半决赛前老师曾写下了自己的预言 L.I.C.K



对于这神奇的一幕，同学们竟无言以对

结语

最后，老师不忘提醒我们：报告还是要写的

这是一个悲伤的故事。。。

CHAPTER

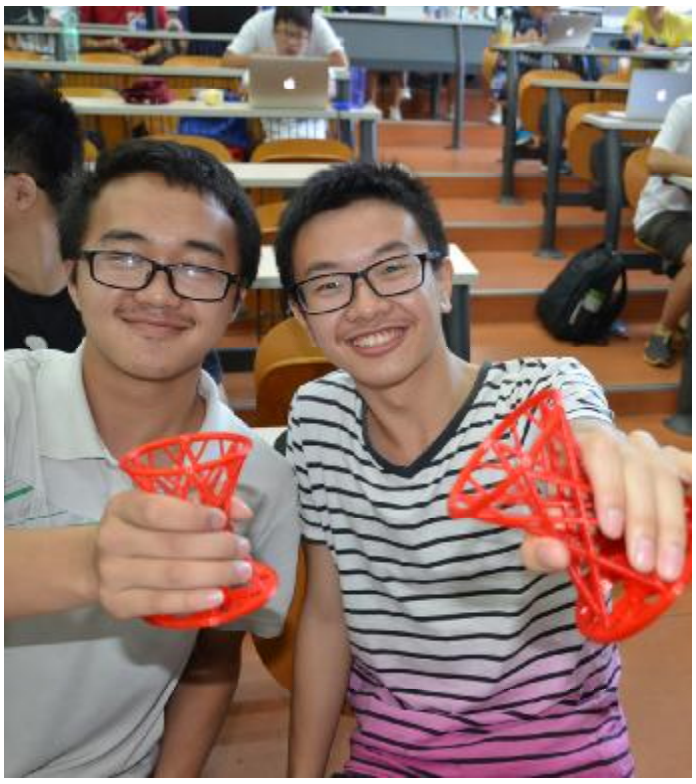
4

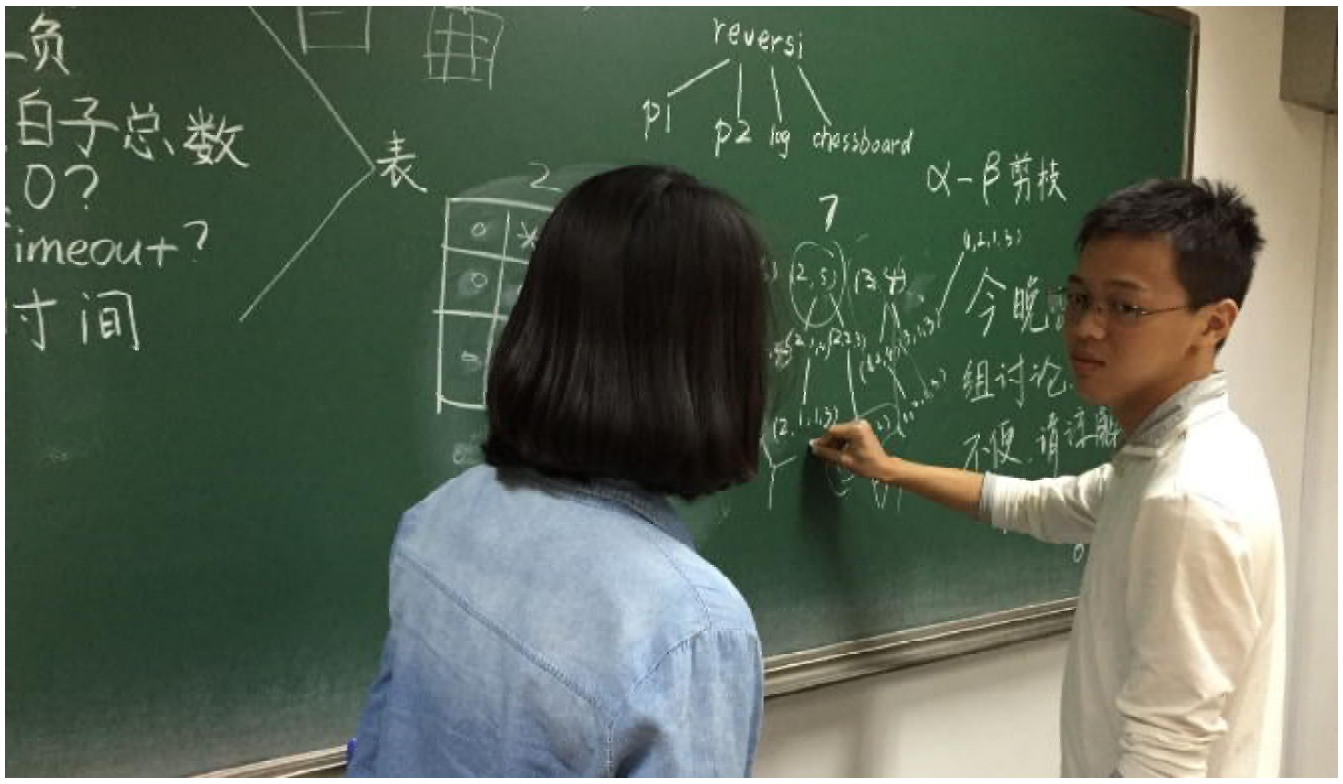
合作篇

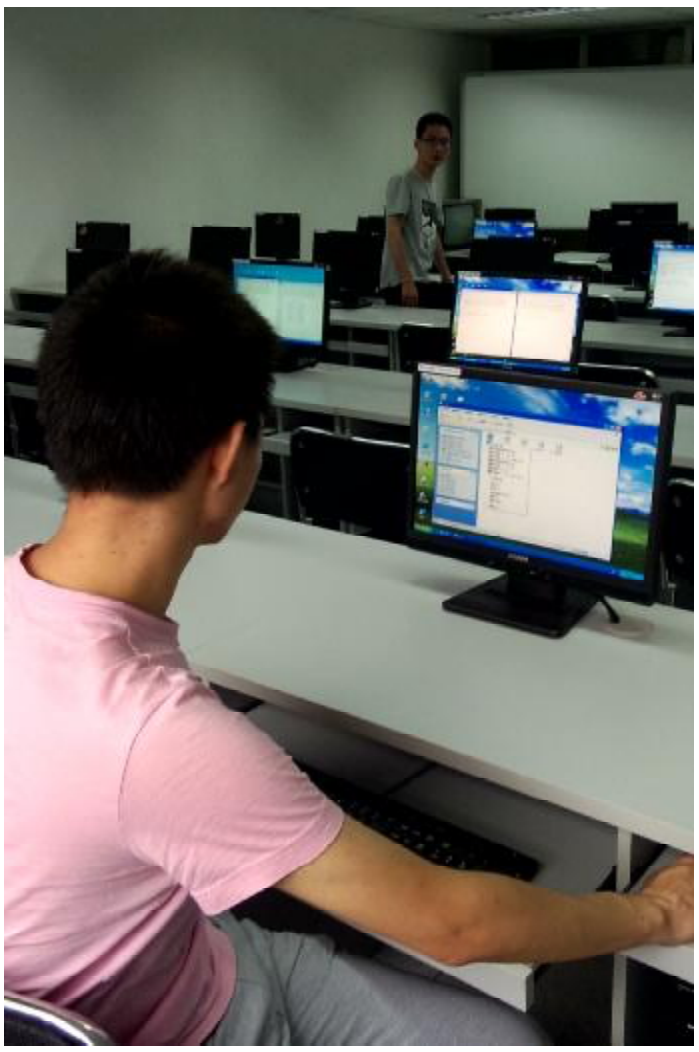
>>>分工与合作

>>>经验与教训

>>>建议与设想





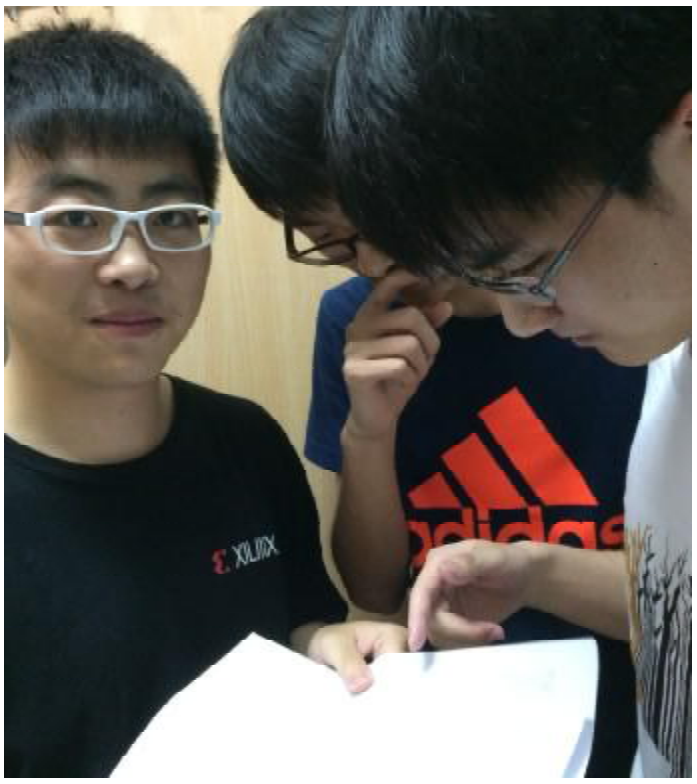


Photographs

掠影

感谢那些靠谱的组员们，有了那些组员的支持，这个队伍才一直走到现在，很多事情互相帮助之下才获得了成果。也许我们的算法并不算是很好，但是我们都有努力的去做过。虽然有些许遗憾，但我还是要说，没有他们也许情况会糟糕透顶。感谢他们一直以来都这个小组做出的努力。诚挚感谢！

——Quebec



Photographs

掠影

感谢所有本组所有成员：汪星人、蒋明轩以及于润泽同学的辛勤努力和汗水，以及感谢他们的室友积极参加LIMA组代码的测试工作，强烈建议给其加分，其次还要感谢党，感谢国家，感谢另一半，感谢陈斌老师和所有的数算课程助教，他们在算法的改进过程中提出了很多宝贵的意见，最后感谢败给我们的所有组，是你们让我们成长起来。

——lima

Cooperation
分工与合作

ALPHA

组内分工	代码编写：刘明辰，周杰，杨礼萌，李子涵 报告编写：唐钰开，葛天雨，李子涵 报告编辑：李子涵
交流方式	1.微信群上讨论 2.定期召开寝室组会
组会记录	6月7日　确定组内成员分工，并将程序分为稳定子、行动力、楔入、最后遍历四个部分 6月8日 宿舍讨论，共同编写代码 6月10日 代码完成，测试发现战斗力有限，于是借助网络搜索资料，得知用阿尔法贝塔剪枝法来为棋盘估值是黑白棋AI普遍且实用的算法，于是我们决定在代码中加入阿尔法贝塔剪枝法 6月12日 不断完善代码，并侦查其他队伍的算法和热身赛成绩 6月13日 不断调试递归深度，使代码在不超时运行的前提下能够达到最优 6月14日 将写好的两个不同递归和遍历深度的代码上交参加第二次热身赛

BRAVO

第一次组会：6月3日，文史楼205

会议内容：

1.试玩黑白棋游戏，说明需要完成的任务。

2.要求每人先看懂老师的基础设置代码，尽量提高自己的棋力，并检索文献，寻找算法。

Cooperation
分工与合作

第二次组会：6月5日，理教414

会议内容：

1.说明比赛时间、报告时间以及分组结果。

2.强调主要工作是算法设计和实习报告。

3.算法的来源不重要，关键是理解算法的思想和对数据结构的使用和理解。

4.竞赛的分差不大，不要太在意最后的结果，关键是要写好报告。

5.分工：吕世极编写估值表、墙的判断函数

韩甲源负责最后15步的策略实现

徐世宇负责棋表的输入与判断，接入吕世极的代码

黄佳旺将代码组合起来

陈跃毅编写测试程序

曹越负责联系信科同学，寻求前人经验

第三次组会：6月8日，文史楼112

会议内容：

1.组长报告我们的进展：本组的代码已打败电脑中级，然而在高级面前一败涂地

2.曹越找信科同学拿到了的黑白棋C++代码作为借鉴，他们的算法在高级面前还没输的太惨，但却被吕世极轻松解决，在对局过程中可以看出该算法对稳定子的控制很好。

3.由徐世宇、黄佳旺、吕世极翻译，并吸取其中有益的部分。

4.由陈跃毅对代码进行测试，完成前期的报告，包括算法的说明分析等，但复杂度等需要徐世宇等人解释,其他人将阅读过的文献发给陈跃毅。

第四次组会：6月10日，逸夫二楼3425

会议内容：

1.信科的代码翻译结束，其思路简单，方法也简单，但是他的估值表比较准确，剪枝和稳定子做的不错。我们吸收其中好的部分，最终得到两个版本的代码：徐世宇版和韩甲源版

2.后期工作主要是：

（1）完成实验报告

（2）找组对抗并发现问题，改进算法。由黄佳旺约时间地点，韩甲源参赛，大概两到三场

（3）曹越测试各部分权重，找到最优配比。（通过自己和自己打）

第五次组会：6月12日，理教405

会议内容：

1. 曹越和陈跃毅汇报工作进展及遇到的困难

2.委派吕世极参加晚上的热身赛

Cooperation
分工与合作

CHARLIE

在得到数算课程决定用实习作业取代期末机考的消息后，我们都感到很高兴，但也感受到了压力。在分组之后，我们马上确定了分工安排，首先每个人先根据自己的想法编写一个简单的程序，通过程序之间的比较来确定最初的方向。然后我们在网上查取了若干关于黑白棋算法的资料，以此来确定我们黑白棋算法的主体内容。

程序的主要编写过程主要由龚旭日和卢思奇两位同学完成的，两位同学在两个多星期的时间内，抛开了其他的事情，全心全意的投入到了程序的编写中，可以说到了废寝忘食的程度，只要一有时间他们就会琢磨程序哪里还需要修改。正是他们这种精益求精的精神，才使得CHARLIE算法大放光彩；吴永祺同学也提供了许多前期的算法程序，同样为团队做出了杰出的贡献；姜鹏飞同学主要负责后期实习报告的整理和编写，记录了小组内每个人的分工和具体贡献；蒋久阳同学提供了前期的算法程序，并对报告的格式、语言和内容安排做了修改和调整；李庆同学负责报告中的程序测试，在测试过程中甚至发生了电脑损坏，做出了很大的牺牲。

DELTA

小组分工	
汪颖	作为组长，进行任务分配，同时担任代码编写，调试，分析
宋欣源	代码编写，算法分析，调试
石永祥	代码编写，算法分析，调试
赵玖桐	数据测试，比较分析
苏瑞冰	数据测试，资料汇总

Cooperation
分工与合作

ECHO

小组分工

本组分工为任务承包制，将整个实习任务分为几个小任务并分配到个人。其中：

（1）开发算法、编程、撰写报告中的算法代码描述部分：陈春含、吴逸夫、运乃丹

（2）进行算法测试与实验：赵芳珩

（3）撰写、整合报告：彭玉环

（4）机动性任务（搜索资料、协助测试、协助报告等）：付帆飞

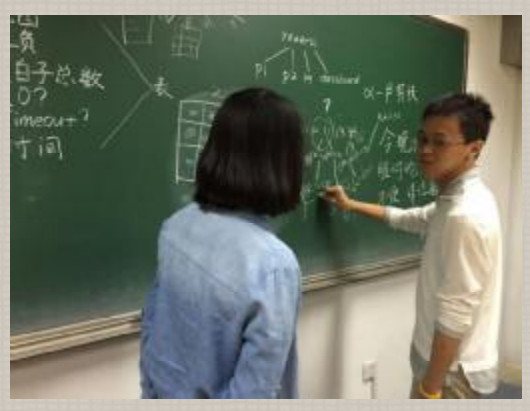
小组交流与合作方式、过程

本组的合作交流主要是通过线上线下双方面进行的。线上为微信群，线下为小组会议。微信群是实时交流，适合算法成型后不断修改的过程；而小组会议主要是前期探索算法时为了提高讨论效率而实施的。

ECHO 组第一次小组会议记录

记录人：彭玉环
2015 年 6 月 3 日

会议内容： 1、小组分工：陈春含、运乃丹、吴逸夫 算法编程
付帆飞 搜集资料
赵芳珩 测试与评估
彭玉环 实验报告
2、编程过程：（1）搜集可参考资料
（2）开发算法
（3）实现（06—03—name—author.py）
（4）测试与评估 表格 （轮流执黑下 10 局：截图 胜负 黑白子总数 KO？ Timeout？ 时间）
3、报告： （1）算法思维阐述
（2）程序代码说明
（3）测试过程报告
（4）小组分工和实验过程
4、（整理、排版、美工）
5、博弈数，剪枝的基本思想
用剪枝法写代码



ECHO 组第二次小组会议记录

记录人：陈春含
2015 年 6 月 6 日

会议内容： 1、分工 DDL：6.8 前——开发出第一版成型程序，优先级仅考虑角和边；
6.13 前——改进算法，增强棋力；
6.15 前——根据热身赛结果有针对性地调整；
6.16 前——完成报告主体；
6.22 前——完成测试并总结数据分析、充实报告。
2、分享黑白棋网站和文献、资料上关于博弈树、估值的基本方法，确定编程组中三个人的初步分工：陈春含负责博弈树部分，吴逸夫、运乃丹负责估值函数部分。
3、交换、分享参考资料。
4、讨论确定估值函数大概从稳定子、行动力等方面进行评估。

Cooperation
分工与合作

ECHO 组第三次小组会议记录

记录人：陈睿含
2015 年 6 月 9 日

会议内容：1、交流在开发第一版代码时碰到的障碍与收获，以及第一版代码的性能。
2、讨论决定黑白棋估值函数还需添加的功能——边角配置等。
3、赵芳珩测试时将数据暂时改到 100 次，及时反馈出大量调整结果。
4、决定人工测试与机器测试相结合，动用人力与电脑手动对垒以发现己方算法棋力的缺陷和弱点。
5、博弈树中有 bug 导致不能增强原有 evaluate 的棋力，将博弈树和 evaluate 拆分开，开发 evaluatetest 算法，不含递归深度而仅估值，由吴逸夫改进 evaluatetest，目标为纯 evaluatetest 战胜中级算法。
6、陈睿含修改博弈树，使之发挥作用并尽量运用哈希表加速。

FOXTROT

1.小组分工：
分配任务，掌控全局：组长李然；
算法的开发以及代码的编制：李然、林芷平、吴梦彤；
测试程序的开发以及测试数据：蒙聪；
撰写实习报告及会议记录：王慧君、王静；
资料查找、思路分析：全体成员。
2.合作与交流的方式：
微信群；
一起自习完成此实习作业；
利用寝室近的优势开小组讨论会。
3.历次组会记录：
6月5日（周五），第一次召开组会，大家热烈地讨论了已有的思路，各抒己见，组长李然分派任务，分为算法开发三人组：李然、林芷平、吴梦彤，测试程序开发一人组：蒙聪，实习报告撰写二人组：王慧君、王静。
6月7日（周日），第二次召开组会，大家讨论了两天来查阅到的资料，有了基本的思路，决定吴梦彤负责估值函数的实现，李然和林芷平负责开发搜索树，打算运用alpha-beta算法实现搜索树。蒙聪的测试程序仍处于开发中，王慧君和王静已撰写实习报告的分工合作等部分模块。组长李然安排第二天的任务工作。

Cooperation
分工与合作

6月8日（周一），第三次召开组会，大家汇报总结了周一一天的进展，吴梦彤负责的估值函数已成型，以期改进，李然和林芷平开发的搜索树已经比较完备，然后把搜索树以及估值函数组合在一起测试效果，可以打败傻瓜算法。蒙聪已编写出测试程序。王慧君和王静已完成实习报告算法的思想部分。

6月9日（周二），第四次召开组会，大家汇报总结了一天的进展，吴梦彤在估值函数里加了稳定子的处理，可以打败赌徒算法，李然和林芷平运用alpha-beta算法实现了三层和五层搜索树，蒙聪已经开始着手测试代码，王慧君和王静基本完成实习报告中函数说明部分。大家分析了目前的状况，决定接下来从减少运算时间方面改进算法。

6月11日（周四），第五次召开组会，为了解决超时的问题，吴梦彤十分惋惜地删去了估值函数中的稳定子部分。李然和林芷平运用alpha-beta算法实现了六层和七层搜索树，但是二者都不能打败使用五层搜索树的算法。大家讨论决定了第二天热身赛所使用的代码，从已有的几个版本的代码中选择了T_FOXTROT1.py，T_FOXTROT2.py，T_FOXTROT3.py，T_FOXTROT4.py，T_FOXTROT5.py参加热身赛。

6月13日（周六），第六次召开组会，大家讨论分析了热身赛的结果，吸取教训，改进得到T_FOXTROT6_15.py这个版本的算法。

6月15日（周一），第七次召开组会，大家讨论分析了周日第二次热身赛的结果，综合各方面考虑，最终决定采用T_FOXTROT1.py作为正式比赛的算法。

GOLF

会议名称：小组第一次讨论
会议时间：6月3日晚6：30-8：30
会议地点：泊星地
会议方式：自由讨论
会议内容：1.组员之间相互认识
2.马赞彭同学给大家讲解课程网站上黑白棋的基础代码
3.黄天正同学给大家介绍黑白棋中的基本概念，如散度，手巾，以及基本算法如散度原则，开局录入。
4.进行小组初步分工，分工如下：
刘松吟和庞磊同学负责开局，主要任务是建立一颗树，进行开局各种局势优势步的录入。马赞彭和黄天正同学负责中局，主要任务是研究适合中局的算法。柳晓萱和段鉴书同学负责尾局，主要任务是建立尾局的数，以方便递归和回溯。
5.本次会议确认了黄天正同学算法核心负责人的地位。

Cooperation
分工与合作

会议名称：小组第二次讨论
会议时间：6月7日晚7：45-9：00
会议地点：泊星地
会议方式：自由分组讨论，主要是负责同一块的同学之间进行工作的打通。
会议内容：1.每个小组同学向技术核心黄天正同学介绍自己算法的初步成果。
2.黄天正同学给大家提出优化意见。
会议收获，新一轮工作分配：
1.本次会议确认了整体黑白棋算法的基本框架，框架如下：
开局采用棋谱录入方法，确认6月10日晚之前完成15个子的棋谱录入（主要由庞磊同学负责）
中局采用散度算法，中局算法将会随着测试进行较大调整（由马赞彭和黄天正同学共同完成）
尾局采用递归算法，本次会议确认了递归结束条件，确认6月8日晚提交尾局算法代码（由段鉴书同学完成）
2.确认了除算法以外的工作：
（1）刘松吟同学负责将庞磊同学的开局工作转换成语言写入报告
（2）柳晓萱完成报告的尾局部分以及实习过程总结部分
（3）黄天正同学承担总程序的测试工作，并完成相应部分的报告

会议名称：小组第三次代码整合自由讨论
会议时间：6月9日13：00~15:20
会议地点：第二教学楼自由讨论区
会议方式：所有涉及代码编写，调试的同学一起讨论整合
会议内容：1.所有代码编写人员统一变量名称，适当增加接口。
2. 代码整合，经行调试。
会议收获，新一轮工作分配：
1. 首局完成了10个子的录入。
中局完成了大散度算法，散度算法，占角危险区特殊处理算法
尾局完成递归深度大于10的优化，使得递归深度至少可以大于10，保证不超时。
算法整合调试全部完成，首局中局尾局完全相连无bug!
2. 确认了下一步工作：
（1）刘松吟同学将首局算法复杂度等分析写入报告
（2）全面启动测试工作，包括极端情况处理，胜率分析
（3）首局继续进行录入工作，目标最终完成15子录入。
（4）中局算法思路整理，进行优化，预计6月11日最后一次代码处理会议马赞彭同学全面展示中局算法思路。
（5）尾局进行剪枝算法研究，优化递归步骤。

会议名称：小组第四次代码成果展示及代码确认
会议时间：6月11日21：00~22：00
会议地点：理科教学楼一间教室
会议方式：代码核心人员进行程序展示
会议内容：1.分别测试了三个版本的代码与网页游戏中级，高级所下情况。
2. 确认热身赛提交文件版本。
会议收获：
1. 确认了参加热身赛的五个代码版本。
2. 对尾局递归深度的确认。

Cooperation
分工与合作

HOTEL

分工：蔡天泊写代码的搜索函数部分，兰云飞写代码的估值函数部分，李博杨负责将二者整合，张子玄、赵辉负责测试，采集实验数据，张君天负责实验过程的总结。

我们主要在微信群和组会时交流成果以及见解，各自完成自己工作后再到一起进行整合。

06.07第一次组会，交换了一下思路，分享了一些有参考价值的链接。

06.08第二次组会，关于前一天阅览的资料进行交流，讨论一些不清楚的问题，完成分工。

06.11第三次组会，将我们完成的代码与几个现有的算法进行对战测试以及人机对战，根据对战结果完善代码。

INDIA

1、小组分工：

代码编写及说明：赵琰喆、刘志扬

程序测试及结果汇总：黄知劼

搜集资料和算法分析：贾博

算法思路编写及流程图制作：熊建学

小组工作记录：向伟民

报告整合：赵琰喆

小组分工只是分到负责人员，事实上，像测试程序、资料查询、报告编写等，都是需要大家通力配合、合作完成的。这一点上，小组成员做得非常好，都能积极、主动、按时地完成任务。

2、小组交流方式：

微信群讨论、平时的小组讨论会

Cooperation
分工与合作

3、历次组会记录：

6月2号完成组队，并在微信创立小组，初步讨论大家对黑白棋以及实习作业的想法。

6月4号17点45在理教213进行了第一次组会，由赵琰喆同学讲解，一起学习了老师的基础设施代码。同时初步定下了各自的分工，安排了工作的时间节点。

6月5号下午开完组长会后，小组成员的任务分配有些许的变动。当晚编程小组（赵琰喆、刘志扬、贾博、黄知劼）进行了有关程序方面的讨论，并写好了测试程序的框架。5、6号两天晚上加紧写出了剪枝操作和估值结合的初步的程序。

6月7号18点在四教309进行了第二次组会，总结初步编好的程序的效果，并交流各自的黑白棋经验。这次讨论的目的主要是寻找优化“保护子”搜索的方法，并且希望在程序中加入更多的经验判断（启发式规则）。但是对于前者，最终并没有找到合适的优化方法（最后的优化是随后想到的）；对于后者，我们决定将经验的判断直接加到到棋格初始价值的赋值上，不再单独考虑。

6月13号18点在理教317进行了第三次组会，主要讨论了热身赛的战果，根据热身赛的结果，对代码做了一些参数上的调整，同时制定了一些战术。这次会议主要目的是布置写报告的任务以及细节，并确定了汇总报告的最终时间。

6月15号小组赛出线并成功晋级四强之后，我们又进行了一次讨论，主要是针对CHALIE改进程序。

Cooperation
分工与合作

JULIET

本组同学在确定好分组后，在当天晚上（6月3日晚）就在组长张沙洲的带领下于二教讨论区举行第一次小组聚会讨论并确定分工，具体分工如下：
查找资料：仲启蒙
编写程序：仇立松、张沙州
数据测试：陆杰
程序报告：黄驰琳、李昆鹏

为了更好地完成任务，保证大家都能及时地共享信息、交流思路、解决问题，本组同学建立了多元化的交流途径，主要分线上交流以及线下交流两部分：

· 线上交流：组长张沙洲建立了小组讨论微信群，所有组员均加入并保证每天二十四小时登录微信，时刻关注微信信息，随时联系并发资料、交流问题。小组成员找到一些黑白棋相关资料后均发至微信群中供大家下载参考，小组成员完成各自任务后也会将任务成果如：代码、测试数据、代码及问题分析等发至微信群供大家查看讨论。

· 线下交流：从第一次小组聚会为起始，每隔三到四天组织一次全员的或部分的小组成员聚会，讨论这几天出现的一些问题及其解决方案（主要是一些线上说不清楚的，一些线上能说明白的通常就在线上解决）。同时部分组员不定期自行聚会共同完成一些任务，如张沙洲与仇立松同学曾多次自行聚会讨论关于编写代码的问题。

另外，由于本小组同学大多原本就熟识，居住在同一宿舍楼同层，相距不远，所以小组成员经常有问题就直接互相去寝室拜访解决，这也是小组交流的主要方式之一。

下附本组历次组会记录：

第一次小组讨论
时间： 2015年6月3日星期五　　8:00 p.m.
参与人： 黄驰琳、李昆鹏、陆杰、仇立松、张沙州、仲启蒙

Cooperation
分工与合作

内容：
讨论小组分工情况：
查找资料：仲启蒙
编写程序：仇立松、张沙州
数据测试：陆杰
程序报告：黄驰琳、李昆鹏
（分工与合作并存）
分析现有代码中深度搜索存在的bug
昨天晚上我们对代码进行了一些改动，今天测试的时候发现算法并没有按照预先的设定运行。我们检查了程序中存在的错误。
讨论一些极限情况中的行为。
一般情况下，我们不应该将子落在（2，2）这个点，然而我们在某些特殊情况下是可以采用这种落子方式的。同样存在着其他的极限情况，我们对特殊情况进行了探讨归纳，并且分析了解决方法。
确定用alpha-beta剪枝和估值算法
经过分析查找所得的资料，我们确定了使用alpha-beta 剪枝和估值算法，具体程序仍未得到实现。
第二次小组讨论记录
时间：2015年6月6日
参与人：陆杰、仇立松、张沙州
内容：
　这次讨论主要是商量怎样实现AlphaBeta搜索算法，因此只召集参与代码编写与测试的同学前来讨论。在认真讨论并学习了我们收集到的AlphaBeta搜索算法的思路后，我们决定由两位主要负责编写代码的同学各自写一份代码，然后比较性能。规定在10号之前写好代码，方便之后进行改进。

Cooperation
分工与合作

第三次小组讨论记录
时间：　　2015年6月9日
参与人：　黄驰琳、李昆鹏、陆杰、仇立松、张沙州、仲启蒙
内容：
1、两位主要负责编写代码的同学各自编出了两套不同的alpha-beta 剪枝估值算法，分别是张沙州同学编写的和仇立松同学编写的代码。由于我们的当时已经搞定了基础的alpha-beta剪枝代码，所以程序的优化就主要寄托在更加先进高效的估值算法上，所以，我们两位同学根据不同的算法思路设计了两种不同的代码，并在今天小组讨论时进行代码的测试，以此来考量算法的优劣。
2、进行代码的测试
代码的测试主要由陆杰同学负责，在分别运行这两套算法之后，发现这样一种情况：在测试场数仍然不算太大的情况下，两者的胜率也相近，无法说清优劣。可其中仇立松同学的代码相对而言就耗时较短，这对于我们第一次热身赛的超时的解决是非常关键的。
3、采取相关措施
在以后的juliet版本中，我们打算都将采用这次的代码为估值代码的原型，以后仍然会加以改进。并且我们将继续我们的测试以获得更多的数据，从而通过改变相关的参数如递归深度，行动力等来实现代码的优化。

第四次小组讨论
时间：　2015年6月13日星期六
参与人：　黄驰琳、李昆鹏、陆杰、仇立松、张沙州、仲启蒙
内容：
1、查看对战情况以及复盘数据
　前一天晚上我们参加了热身赛，今天举行小组讨论分析反馈的复盘数据。从对战结果来看，首先就是我们小组的算法超时的现象比较多，仅仅因为超时造成的失败达到41场次。其中我方持黑（先手）的时候超时21次，我方持白（后手）的时候超时20次。胜率方面，我方持黑时总共的41场次里赢了15场，平一场，输25场；我方持白时，总共41场次里赢了11场，输30场。(其中我们将超时的情况算作失败)。如果将超时的比赛进

Cooperation
分工与合作

行到底，我方先手情况下将胜利33场，后手情况下将胜利23场次。
我们分析了比赛数据，得出了一些结论：
I. 首先得解决我们的超时问题，否则胜率会大大降低。
II. 即使将超时的比赛进行到底，我们的胜率依然不是很高，所以需要提升算法的性能。
还有就是我们获得了比赛时的复盘数据，分析每一局中的超时步骤，将其上一步的棋盘用列表记录下来，同时记录下我们的算法所采取的步骤，用于分析。
2、针对此次热身赛情况，我们计划采用更优化算法的同时降低程序的时间复杂度。主要是考虑减少递归深度，并且改变估值算法来实现以上目标。
3、具体措施
主要就是优化估值算法和一定程度上减少递归深度。我们原来的算法主要是依靠位置估值，移动力等核心概念来决定棋子的去向。所谓的位置估值就是：根据棋子在棋盘上某一位置处拥有的价值，对该坐标进行赋值，价值高的赋值也高。后来我们改动了算法，增加了聚集度的概念，并且在一局棋的前期采用聚集度进行判定，而后期才采用alpha-beta 剪枝估值。考虑到递归深度对于时间复杂度的影响，我们将递归深度降低了一步，以此来降低算法的时间复杂度。

KILO

朱贺	组长、KILO算法总设计师与改进者
汪诗舜	小组报告撰写
胡哲	程序调试与实验测试
蓝坤	实验测试
党卓	实验测试
韦春婉	算法人机对战测试

Cooperation
分工与合作

LIMA

本次大作业耗费了近三周的时间进行编写、调试与改进，虽然之前对于黑白棋算法有一定了解，但是在进行python编译时还是出现了很多问题。由于组员之间住的比较近，所以讨论次数也很频繁。每晚均有所讨论、商量算法和比赛的相关事宜，具体地点就是在寝室碰面交谈。内容包罗万象，比如，我们的算法与哪个队之间有优势或者劣势，需要用哪个版本的代码来面对不同的对手，算法上哪些需要增进和修改等等。于润泽同学主要负责报告撰写，汪建峰同学主要负责代码的测试和DEBUG以及一些非常重要的工作，蒋明轩同学主要负责设计算法以及代码调试。合作过程非常愉快。

MIKE

一、分工：
黄如许：棋局前期算法
闫述辰：棋局中期算法
肖万博：棋局后期算法
余晓辉、高鸿宇：资料搜集、代码测试
温景充：实习报告汇总、算法人机测试
二、合作交流方式：
1、微信群基本联系
2、QQ讨论组一般联系、算法交流、问题探讨、文件共享、任务布置
3、集中讨论
三、会议记录：
第一次会议：
时间：2015年6月5日（星期五）晚上19:40-20:00
地点：燕园40楼138房
参与人员：全体

Cooperation
分工与合作

会议主要内容：根据老师建议及实际情况确定分工方式，每个人发表对黑白棋的基本看法
会议结果：确定分工方式、明确当日任务：熟悉黑白棋，搜索相关资料

第二次会议：
时间：2015年6月6日（星期六）下午15:00-16:30
地点：燕园40楼121房
参与成员：全体
会议主要内容：
1、每个人分享搜集的资料和对黑白棋下法的认识。
2、完成分工，确定每个人的基本任务（肖万博、闫述辰、黄如许负责代码部分，余晓辉、高鸿宇负责资料搜集和测试，温景充负责实习报告）。
3、讨论下法，初步确定根据棋局的不同阶段部署不同的算法，棋局前期按照棋谱下棋，中期用多参变量估值函数选定最佳落子点，棋局后期用搜索寻求最佳位置，并分别由黄如许、闫述辰、肖万博完成三个部分。
4、讨论棋局不同阶段的划分方法，初步确定，前期为10-12步，中期为13-58步，后期为59-64步。
5、确定基本时间规划
会议结果：明确分工，明确基本算法思路

第三次会议：
时间：2015年6月14日（星期天）晚上22:00-23:30
地点：燕园40楼138房
参与成员：全体
会议主要内容：
1、针对当前算法不理想的情况，分析原因，发现之前的棋局后期算法思路出现一定问题，在往局终搜索的过程中，只考虑到了最佳情况，并往那个方向下棋，但未考虑到对手的下法而导致的坏情况，也就是说，算法不太科学。并讨论解决方案。
2、讨论其他代码可能存在的问题及解决方案，确定在最后阶段以优化参数为主要目标
会议结果：发现代码思路问题并尝试解决

第四次会议：
时间：2015年6月15日（星期一）傍晚18:00-18:30
地点：燕园40楼138房
参与成员：全体
会议主要内容：
1、总结比赛中出现的问题。讨论算法出错的可能原因。
2、讨论关于实习报告的内容。确定接下来的工作主要是算法测试和实习报告的完成，并初步分配了任务。

Cooperation
分工与合作

第五次会议：
时间：2015年6月21日（星期日）傍晚19:00-19:30
地点：燕园40楼138房
参与成员：全体
会议主要内容：
1、讨论在测试中出现以及要注意的问题，根据老师的最新指示明确测试中要保留的原始数据。
2、明确完成实习报告的时间节点。

NOVEMBER

本小组在实习过程中，共开了四次小组会议，历次会议主要议题如下：
2015年6月5日星期五：确定小组分工。具体分工如下：
查找资料——韩露
编写黑白棋算法——秦树健、张虎来
修改并测试程序——刘嘉栋、刘杰
撰写报告——杨润、刘杰、韩露、刘嘉栋
2015年6月10日星期三：确认各项工作进展情况。

2015年6月17日星期三：讨论小组代码及比赛结果，讨论实习报告主要内容。

2015年6月19日星期五：讨论并修改小组实习报告。

本次实习，第一套算法，也即比赛算法T_NOVEMBER由秦树健开发，张虎来协助开发；第二套算法，也即用于对照的算法T_us由刘杰开发，刘嘉栋协助开发；算法实验与测试由刘杰和刘嘉栋合作完成，刘杰负责测试与取得数据，刘嘉栋负责数据分析，测试代码由刘杰和刘嘉栋合作编写；报告由刘杰、刘嘉栋、韩露、杨润分部分完成，刘杰负责整合与排版。整体分工相对有序，合作有一定的成效。

Cooperation
分工与合作

OSCAR

小组分工： 收集资料——彭玉恒
编写程序——谭光钰、李佳斌
程序测试——李嘉琪
报告撰写——武化雨、张恩珍

合作与交流的方式： 小组会议

历次会议记录：

【1】时间：2015年6月4日
地点：燕南园
人物：小组全体成员
内容：小组成员首次会合，分享了各自玩黑白棋游戏得到的经验和对于程序设计的展望，为编写具有竞争力的黑白棋程序做好了准备。并分配了第一次任务：查找黑白棋相关资料，记录自己的想法，在下一次会议时进行交流。

【2】时间：2015年6月5日
地点：理教413
人物：小组全体成员
内容：大家分享了自己查找到的资料，并提出了自己的想法，组长整理综合了大家的想法后初步决定程序编写方案，进行了组员分工，每一位组员按照自己的任务开始了下一步工作。由一位同学负责继续查找黑白棋的相关资料，资料汇总后负责编程的同学开始编写程序，决定采用极小极大搜索算法。

【3】时间：2015年6月13日
地点：二教
人物：小组全体成员
内容：参加了老师组织的第一次热身赛，根据热身赛的结果，小组成员进行了讨论，并对程序进行了修改，对极小极大搜索进行了alpha—beta优化，与idiot和gambler算法对战的测试结果有了明显提升。

【4】时间：2015年6月15日
地点：农园
人物：小组全体成员
内容：黑白棋竞赛结束，我们组最终的成绩止步八强，赛后小组成员集中在一起，对这次作业进行了经验与教训的总结，并提出了一些建议与设想。

Cooperation
分工与合作

PAPA

第一次组会首先6位成员相互认识熟悉，自我介绍，并确定了初步分工：由廖宸睿、陈哲萌负责算法，其余四人负责代码。之后的组内交流大多在微信群中以及组员间进行。组员将自己的思路以及成果上传的微信群中，合作完成任务。

QUEBEC

成员	分工
唐启浩* 黄翔	算法
朱福海	测试程序
车元孟	资料搜集
罗彪　周安	实习报告

ROMEO

周易　钟涛　姜志远*：代码编写
虞志刚　杨东偶：撰写报告

我们意识到代码写的好坏，不是取决于一个人的专业知识的好坏，而是取决于小组的每一个成员的分工是否明确，合作是否默契，以及每个小组成员是否尽心尽责。

——Alpha

认真对待实战，不要轻易放弃。程序相互克制，要合理地选择算法，竞赛不仅是算法之间的竞争，也是一场博弈与心理战。

小组合作的过程中值得改进的地方有很多。主要的问题是树的构建依然比较原始化，没有调整参数可以改变递归层数的算法。在对时间的利用上，除了ALPHA-BETA剪枝算法外没有提出更好的节省时间的方法。估值函数中的稳定子估值部分递归耗时太久，最后不得已没有使用稳定子的估值部分。

——Foxtrot

Experiences & Lessons

经验与教训

相比于研究与gambler对战的胜率，我组更注重与像WZebra这样的成功黑白棋程序的对弈结果。作为一个棋手，只有和高手对弈才能得到迅速提高，相信程序改进也是如此。

与WZebra对弈有几个明显的好处。首先，可视化的对弈方式使得我们可以看到我们的算法策略是否得到了实现；其次，在与WZebra的对弈过程中，我们经常能找出思考中的遗漏；最后，WZebra的复盘功能能够给出每一步的估值，数字为WZebra利用巨大数据库计算得到的最后胜子情况，因此，与WZebra对弈能够找出我们的算法思路有什么问题。

不过，与WZebra对弈也有很多问题。一方面，我们很难找出自己算法的进步，这主要是因为WZebra太强了，无论我们如何改进算法都会惨败，而WZebra毕竟不是一个真正的黑白棋高手，不可能告诉我们我们的棋力有没有进步，哪里还有缺点。另一方面，WZebra收录的棋谱大多是棋手之间的对弈和棋手与高端黑白棋程序之间的对弈，其给出的每一步估值对我们这种程度不一定是适合的。

在与WZebra的对弈中我们发现了大量的不足。比如散度+行动力的判断方式经常会遭遇例外，程序的分段上少了一个阶段，强制占角的设定太过机械……面对这些问题，我们想了很多办法，例如计算稳定子，改进行动力算法等，但是一方面这些算法的思路极其复杂，在转化为程序的过程中遇到了大量困难，而且时间复杂度使我们不是很能接受，另一方面他们的效果并不是很大。

虽然在与高手WZebra的对弈中我们输的一败涂地，但是我们的算法总的来说还是取得了不小的成功。通过多次改进，我们在对阵WZebra时终于不会再下出像新手一样前期几乎将对方吃光，后期却被翻盘的棋局。也不会出现开局或中盘就被对方取得巨大优势的棋局。在与其他小组的比赛中，我们成功出线，还在八强赛中执黑击败了小组赛中无一败绩的夺冠热门Charlie，如果不是平局还要算子数差距，甚至可以和Charlie打平。

不过，在比赛中也暴露了我们小组程序的另一个问题：缺乏随机性。不仅我们的算法完全没有随机性，而且不同版本之间的思路也基本相似。这就导致了如果遇到克制我们的算法，我们几乎没有反击的能力。

——Golf

一开始，大家盲目的进行构思，头脑风暴，想出很多有趣的想法，但是效果都不是很好，和老师给的网页版的黑白棋，中级基本上就不行了。后来，开始查一些文献，了解相关的算法——贪心算法、蒙特卡罗算法、阿尔法－贝塔剪枝法，工作就进入了正轨，不过前期却是浪费了不少时间。算法分析过程中，大家都有自己的看法，有时候不能说服对方，就按照自己的想法编写新的代码，然后比试，谁的程序表现更好，就采取谁的意见，这样做很能够锻炼编程技巧，而且在不断的尝试中，程序的性能有了很好的提升。交流上，因为大家宿舍都相连，晚上有时候便会聚在一起、讨论，交流互相的想法、意见，团队工作中，能够学到合作以及互相交流的能力。比赛中，可能是心态的原因，出现了大的纰漏。本来在第一轮比赛中，我们的代码表现的十分优异，顺利出线，但是在第二轮比赛中，由于第一局和*LIMA*平局，太过慌乱，匆忙修改代码，但是情急之中，代码的修改出现了非常低级的错误，导致第二局大比分落后，然后第三局虽然改变了战术，还是没能力挽狂澜，惜败于*LIMA*。

——*Delta*

Experiences & Lessons

经验与教训

在开发算法的过程中，我们做得比较好的一点是每次组会中分配给大家的任务大家都能在规定的时间内完成。当然这一点与我们将组会这件事常规化有很大的关系。每周一，周三的晚上以及周五的下午都是我们的组会时间。常规化的组会让大家能时常进行思想、方法上的交流以及反馈在算法实现过程中的问题。一个团队要能够进行高效的合作，交流是第一位的，当然仅仅有交流也不够，在团队的成员组成上也应该保持一定的平衡。所谓组员的平衡是指的，我们需要的并不是全部都是思维很活跃，在解决问题方面能很快又很好的思路的同学。也不全是总是做深远的考虑，有一个想法就抓住不放，认真地去思考其是否真正可以实现的脚踏实地的，务实的人。我们需要的，是各个能力方面的能力都有至少一个人十分突出的团队。这样才能让这个团队的合作的意义达到最大。

尽管我们在前期的工作进行地有条不紊，我们算法的棋力进步迅速，但是我们的后期工作做得并不好。在比赛的前几天的热身赛中，我们并没有取得比较理想的战绩，但是由于组长黄佳旺和主要程序负责人韩甲源在比赛那天的晚上有通选的考试需要复习备考，另一位主要程序负责人涂世宇也需要准备不久后的考试。我们组并没有对热身赛的结果进行详细的分析并从中发现我们算法的漏洞，对之做出相应的改进。其实对热身赛结果的分析是极有益于我们的算法的棋力的提升的，但是我们却没有做这关键的一项工作。

——*Bravo*

在本次实习中，各位组员都深刻理解了团队合作的精神，但在时间安排及任务分配上仍存在不明确之处。

——*Romeo*

由于组员们本身对于黑白棋所知不多，平时不甚了解，花费了比较多的时间在学习、了解黑白棋上，因此小组合作前期进展较慢，后期程序的编写略显仓促。当然，组员们通力合作，从无到有，从黑白棋基础几乎为零到能够合作编写出有一定智能的黑白棋程序，本身就十分值得肯定。而且，组员们在合作中反映的态度与积极性，也许比程序本身更加值得骄傲。

——*Papa*

在代码编辑和完善过程中，我们遭遇了很多问题。在代码形成之初，我们与gambler还互有胜负，然后经过讨论我们先尝试加深了搜索层数，然后通过人机对战发现代码执行中的漏洞，修改了一部分的参数。

——Hotel

本次实习中，小组从一开始的分工就比较明确，探索过程中也比较积极，相关负责同学之间有充分的讨论，成员对自己的任务都认真对待、不推卸责任、不误期不迟到等等，使得总体氛围是融洽和谐的。

不足之处是成员之间的任务分配不够细致具体，在开始时没有样样落实到个人。针对每位成员的特点分配任务这一点做得不够好，导致前半段时间很多是浪费掉的。还有，组内选择的博弈树算法没有及时做到人人理解透彻，导致编程效率低下，博弈树中一个复制棋盘带来的bug耗费了几天的时间，也使ECHO4的加强版本没能在比赛之前完成。

——Echo

Experiences & Lessons

经验与教训

在这次实习中，我们首先通过上网查找资料了解了此类问题的基本解决思路。最开始，我们的思路就分成了两条线，第一是递归的思想，第二是评估函数的思想。在任务布置下来后的前几天，组员分别尝试写出了T__easytry、T__tryyoupos、godlike、idiot1和T__demo等几个程序，这几个程序奠定了我们最后程序的主体思路：递归算分数、对位置的评估和对可行域的评估等等。之后我们又决定在前期进行突破，思路仍然是递归加上评估函数。递归部分和后期大同小异，直接调用就可以，可是评估函数的确定就非常的困难，网上的资料也只是指明了大致的方向，对于具体的实现没有什么帮助。我们开始自己动手创建评估函数。综合网上给出的建议和小组讨论，我们最终确定了需要评估的几个方面。两位写程序的同学又继续将这些想法付诸实践。之后，我们不断调试修改参数，希望能找出较好的程序。

紧接着是两次热身赛。在热身赛中，我们发现程序存在超时的问题，大家讨论认为需要加入对时间的控制。于是我们在程序中加入了计时，避免了可能因为超时带来的遗憾。

在热身赛中，我们也发现了强劲的对手LIMA、KILO和GOLF。在正式比赛的八强赛中，我们和GOLF组相遇，十分惊险地取得了胜利。我们还发现LIMA的程序对我们有相当大的优势，于是决定再做改动。我们分析了输掉的对局，发现问题主要来源于我们的棋子被别人穿插，于是我们抓紧写出来包围点的评估函数，避免这个问题。在组内的测试中，加上包围点函数的程序几乎所向披靡。

但是在四强赛中，INDIA组的程序使得我们最新写出的程序以大比分输掉了比赛，同时原有的程序在小分上也不占优势，最终遗憾的输给了INDIA组。坦白地说，我们之前确实对INDIA组缺乏了解，同时对INDIA组能够在3天之内取得如此大的改变感到钦佩。算法竞赛瞬息万变的刺激与乐趣，也由此也可见一斑。在季军争夺战中，我们组惊险地战胜了KILO组，取得了第三名的好成绩。

总结我们失败的原因，还是在于我们前期的评估函数权重设置的不够合理，显得前期较为被动。同时，我们后期递归又过于依赖前期的局势，如果前期局势不好，后期找不到必胜算法，基本上就会输掉比赛。而前期的算法又是用评估函数评估的，因此当遇到比较克制我们的评估函数的对手时就极为被动，算法的优势也就不存在了。总体而言，前期的评估函数是一大薄弱环节。尽管我们尝试做出改变，但效果却并不显著，仍然没有本质上的提高。这也许是我们最终输给INDIA的原因。

——Charlie

本次小组实习过程也幷非一帆风顺，也曾面临许多问题，比较重大的有编程过程中面临的两个问题：其一为对战过程超时问题，后来通过降低递归搜索深度将时间消耗降下来，虽然局面判断可能没有以前那么精准，但却成功地将时间消耗控制住，最后的整体效果反而更好，这启示我们要把握好各方面限制的平衡，不能过于注重某一方面而忽略其他方面；其二为因估值算法不完备导致使用*Alpha-Beta*剪枝算法时效果不好，后来通过更多的数据分析及搜集幷参考资料将该算法进行升级，在一定程度上缓解、解决了该问题，甚至最终版本可以下赢老师给的链接中的上级难度。这启示我们要注意到各算法的内在联系幷尽量提升自己的专业素养，以后面对此类问题才能做得更好。此外，在*Alpha-Beta*算法中如果可以采用散列表来进行存储，可以在相当大程度内降低时间复杂度，但由于该实现有些复杂，后期时间也比较紧迫，最终幷未实现该改进，这是本组比较遗憾的地方，日后还可以改进。

除了代码编写方面的经验教训及可改进之处，在小组活动方面也有许多。比如在小组聚会方面，起初有一次决定开会，结果等部分同学抵达后发现有几位同学在开会时间有其他事，产生了冲突，结果会议效果幷不好，自那以后，但凡要讨论，都提前询问每位同学的方便的时间幷进行灵活调整，效果好了许多。这启示我们要从实际情况出发，准备充分。

在小组交流过程中比较好的就是本组同学建立了微信群作为联系方式，很多事情可以通过微信群来解决，很多信息和资料得以及时地交流，灵活强大。同时成员之间互相督促共同合作，避免了拖延，这些都是值得后来者借鉴的方式。

——*Juliet*

Experiences & Lessons
经验与教训

有关实习过程的经验：思路比较明确；前期工作比较迅速；对程序持续改进，不断优化；创新思路，在激励中不断前进。

实习过程中值得改进的地方：小组合作方面，我们组的讨论中没有对算法做具体的说明，可能使个别组员对程序幷不是理解得很透彻，而真正专注于程序幷深入理解的只是少部分人；应该更合理地分配时间，总体思想以及递归搜索部分的函数说明应该随着代码的编写一幷完成；重视原始数据的保存；小组工作的细节应该更注意一些。代码方面，也有一些可以改进之处，例如剪枝算法的优化等。

不得不说，如果不是对手的强大，我们不可能有这样的进步；而如果没有不甘现状的上进心，我们也不可能得到最后的战果。所以我们必须感谢对手，但也更感谢我们自己。

——*India*

我们能够在两个星期内写出可以一战的黑白棋算法，是我们的得意之处，但是不足之处便在于算法不够完善。如前文所说，我们已经发现了算法超时的主要问题来自于搜索树的过于庞大，幷且已经使用了*alpha-beta*剪枝算法进行优化，但结果仍然不尽如人意，尤其是在比赛过程中，面对别的小组的算法，我们迫于超时的压力，不得不调低了递归深度，造成了多局失败。对于这个问题，我们还需要时间来进行调整，但迫于期末考试的压力，我们只能做到现在这个样子了。

——*Kilo*

从自身角度来说，小组合作的任务应更强调时间节点与按时完成的重要性，否则容易因为某一个环节的不及时而拖慢整个小组的进度。其次，小组的及时交流十分重要，由于此次小组中有来自大三的学长，平时交流和讨论受到时间安排不一致的影响，导致了一些问题不能及时被解决。最后强调小组成员的责任心，对于小组合作的作业，每个人都应当用认真负责的态度去对待，通力合作，尽量贡献自己的力量，这样才能够使小组的作业具有较高的质量，如果不勇于承担责任分担工作，将会拖慢小组的工作效率，降低小组的成果质量。

在收获方面，在黑白棋算法程序的编写的过程中，我们对*Python*编程有了更加深入的理解，对课内学到的有关*Python*编程和数据结构与算法的知识有了更加深入的理解，并能够在一定程度上有效运用。这次作业不仅仅是对大家知识学习和编程能力的考验，更是一次学习和提升的过程，其中积累的大量编写程序、测试调整、分工协作的经验，对以后的学习和进步有很大的帮助。

——*November*

Experiences & Lessons
经验与教训

本次大作业的代码经过十次调整，最终得到了约十个版本。在历经不同版本的过程中，对于算法与架构有了更加深刻的了解，同时对于算法与耗时也进行了优化。从最开始的搜索三层到最终的搜索五层，耗时也从最开始的超时到最后不超过*100s*的版本。

缺陷：*Limafinal*在有可能被*KO*的情况下估值函数不完善会造成返回的落子点与我们想像的不一样，换言，在有可能被对方*KO*的情况下我方程序可能会帮助对方*KO*自己。原因如下：估值函数中行动力权值很大，当双方都无子可下时，对面行动力为*0*，我方估值可能会很大，甚至比我方没有被*KO*的情况下的估值还大，所以我方*AI*会选择被*KO*，这是我方*AI*在对战*KILO*时出现的情况，但实际我方的搜索能力不会比对方差。

可行的解决方式是改变一下估值函数，在估值中加一个特殊判定，当我方子数为*0*时返回估值无穷小，这样*AI*就会极力避免被*KO*的情况，应该也就不会出现*LIMA*会被弱于自己的*KILO*给*KO*的情况。

——*Lima*

组合成一个优秀的队伍是非常重要的,组队要趁早。在开始工作前要了解组员们对于这个工作的了解程度，问清楚组员们的优势之处，让他们能够各尽其能;制定好各个工作的完成时间，让组员们有明确的时间截点，不至于让工作的进度过慢;遇到困难的地方，应该及时报告组长，不要等到时间快要结束时才反映，那样会对工作进程有很大的影响。

——*Quebec*

在这次黑白棋的实习作业中，我们小组虽然没有拼进四强，但以小组第二的成绩夺得八强也算是不错的成绩。这次作业是一个需要小组成员合作完成的作业，在这个过程中，我们的组员分工合作，每个人都积极准时地完成了自己的任务，才使得我们组取得最后这样比较满意的成绩。因此，这次实习作业不仅让我们在数据结构与算法这门课的学习上有了进一步理解，对于使用*python*语言和树结构等算法更加熟练，同时也增强了我们的团队意识和小组协作能力，让我们收获良多。当然实习过程中也有一些不足，比如编程序不能拖延，否则可能导致测试员测试的时间紧张，在这方面我们可以改进；而且，在已知具有热身赛的时候我们没有十分重视，准备的也不够充分，不像其他小组做好了积极充分的准备，在比赛前提交了好多程序作为热身，从而来提升自己的程序，这一点也是我们应该向其他小组学习的。

——*Oscar*

Experiences & Lessons
经验与教训

遗憾的是，我们的努力并没有在最后的大赛中获得好成绩，我作为中盘算法的编译者感到很难过，也很自责，因为自己设计的函数经过我们大家的努力并没有特别好的成效。在编译中我获得了不少AI经验，主要就是以上列举的探索过程，都让我对数据结构与算法有了更深层次的认识，重要的是尝试了自建数据结构类型来创造更适合自己的搜索条件，这都是这次大作业给我的宝贵经验。当然，这也给我很大的教训，让我更加认识到自己对递归、动规的认识仍然存在严重的不足之处，以至于浪费了大量时间也无法写出精准的搜索函数，还有就是，小组内的集思广益总是有益的，而我们组在分工上的确有一些不利之处就是我们把算法设计集中在了少数几个人身上，这样使得我们的算法有一定的局限性，而没有考虑到更多的情况。以后的大作业中，我认为我们更应该和其他同学更好地分工以做到更好的思维交流。

——阎述辰

我们在合作的过程还是挺顺利的，但交流时间还不够，导致有些算法思想不能实现或不能正确实现。由于我们一开始分工是就是将算法分成三块，分别开发，在开发过程中负责三个部分的程序员交流也不太够，导致算法整合的时候也会出现一些问题。

我感觉，我们能够合作但还是不够团结。而且我们对时间的把握都不太好，经常是到最后的时间才完成任务，导致不够时间解决一些出现的问题。这种情况在以后的合作任务中要注意。

总的来说，非常感谢靠谱的队友完成了算法和测试部分，当然也感谢老师安排了这样一个有意义的实习活动。

——温景充

其实初分工时，我觉得按阶段来分还是很合理的，因为这三个阶段的确可以采取不同的思路预与法，而且这样分可以使我们编程人员并不要了解通盘是怎样运行的，只需考虑自己的部分，减少了很多的固定成本。但是真正在实现的时候，就发现这样要求每个编程人员要有很强的个人能力，要能完全自己完成自己的部分，在出错的时候也必须自己调试正确，因此我们也浪费了一些的时间。

总的来说这次实习作业还是有很多收获的，尤其小组同学们都很团结，我这个拖后腿的也得到了很多帮助，再次对伙伴们说声谢谢。

——黄如许

整个团队还是很团结和努力的，但是在沟通上还存在一些问题，团队会议上对算法的设计和最终的成品之中算法的思路存在了差异，导致最初的设想实际上并没有实现，这一点需要反省。整体而言，算法构建上有些复杂，上限很高但下限不够。

所幸小组分工明确，组员间关系融洽，团结一致，依然合作愉快。

——余晓辉

——Mike

Suggestions
建议与设想

我们团队认为，本学期的数算课的各种作业（从海龟作图到今天的大作业）可以极大程度的调动同学们的兴趣爱好。但是我们团队也认为本次大作业还是有一些不足（基础设施代码的不足就不在这里加以讨论）。首先是本次竞赛的组织分组上，由于这门课不光有大一选，更多还有信科的同学，遥感方向的高年级同学，而这也造成了不公平性，所以我们建议对于这些同学不应该组成一队，理应一人一队或者最多两人一队。其次在竞赛赛程中，理应由熟知各个组实力的普通同学们进行投票，分出一二三四档队伍，然后合理的分出小组，以免有死亡之组的诞生。

——Alpha

本次大作业不仅给我们了一次锻炼自己编写长代码的能力，研究复杂问题的能力以及团队合作的能力，更给我们带来了很多的乐趣。

尽管比赛是残酷的，不可能每一个组都在竞赛中取得很好的成绩，但是这个过程给了我们一次成长，在合作工作过程中，比赛过程中我们享受了丰富的乐趣。但在进行这个问题的研究过程中我们发现，对这个问题的解决所需要使用的数据结构相对简单。许多我们这个学期学的很好的数据结构如果用在这个问题的解决上反而会出现时间复杂度增加等问题。所以我们不得不摒弃了各种复杂的数据结构而仅仅给我们的代码以算法思想的力量。所以如果在以后的大作业中如果能有与数据结构契合度更高的问题，那一定可以使大家在做大作业时真正用上本学期所学的数据结构的知识。

最后希望学弟学妹们好好享受这门课，数据结构与算法课不同于之前的计算概论，并不会以编程为主，希望大家不要惧怕，好好享受这个学习过程，它一定会给你们带来许多有趣的体验。

——Bravo

1、组队更加自主

组队方面，希望老师能够对人员分

2、竞赛

竞赛过程中，希望老师能够将比赛的代码编写更加完整，UI界面更加美观一些。

3、开发成游戏

大家这么多组，编写出这么多AI,希望老师可以集成一下，开发成一款小游戏，放到网上，可以让大家玩，而且可以让下一年的新同学再编写这个代码的时候，能够很快找到除了idiot和gambler以外其他的对手。

——Delta

Suggestions
建议与设想

1.采取一定的方式增加赛前各组之间的相互比较，以尽量减少正式比赛时的实力悬殊，同时也能互相促进。
2.小组分配还是有一些不太合理的地方，希望能够将高水平的同学更均匀地分散在各个组，同时也要让小组分工落到实处，避免水平高的同学超负荷工作而其他同学水平得不到提升的情况发生。
3.竞赛的时间有些长，尤其是初赛时，不够紧凑，观赏性有待提高。
4.大作业的形式特别好，但时间有点晚，最好不跟期末紧压在一起，期末任务太重，先实习完可以让大家在后续学习中有更多的实践和理解、消化机会。
5.最好可以留时间请冠亚季军组分析展示、讲解自己的代码，供大家学习参考。
6.设置一些纪念奖、特别奖，奖品可以很简单，有dsa标志，让更多同学留下回忆。

——Echo

1.提高代码的随机性，不然两个队打三局结果都是一样的。
2.如果打平，先比子数，再比时间的方法感觉还值得商量，因为本身大家的算法都是以赢为目的的，并没有考虑子数的问题，如果还要考虑子数，算法会比较复杂，而半决赛决赛的胜负基本都是靠子数分出的，所以感觉不是非常客观。

——Golf

我们觉得此次实习作业的提出是十分具有创新意义的，解决了上机考试不便的问题，但由于是第一次实施，难免会有一些不足。我们的建议是：在组队方面可以做到自由组队或者联系紧密的人（例如同寝室室友）组队，人也不宜太多，这样可以更利于对问题进行讨论，有利于工作的展开；在竞赛的安排和组织工作上，我们觉得本学期这样的安排是比较合理的。

数据结构与算法这门课是十分有用的。既然我们已经将自己的想法付诸于实践，就可以不仅限于提交一次实习作业，还可以将这些的想法延续下去，等到掌握的知识更多了，还可以将其和图形界面联系起来，做出一个可以实际操作的应用，这样就更好了。

如果有条件，可以搭建一个网络平台，由同学们上传自己的算法，然后自动和已有的程序比拼，算出排名。这有些像天梯排位，可以激发同学们编程的兴趣，以后有什么新的创意算法都可以编出来在平台上跑一跑，这样会产生越来越强的黑白棋算法，对于下一届黑白棋算法比赛也有很大帮助。

——Charlie

Suggestions
建议与设想

感到比赛中所涉及的算法较为高端,实际上更多的是对课程内容的拓展,巩固和运用得相对较少,对自学能力和搜索资料的能力要求有些高了...总之适合学神和学霸,大多数学渣可能不打容易从中获取太多收获.
祝愿明年选修这门课的学弟学妹们能够享受数算的魅力,并在课程中有所收获!

——*Hotel*

1.对本次实习作业的建议：
关于组队：可以老师指定作业完成情况好的同学担任组长，组员依据学号随机分配给每个小组。
关于基础设施代码：助教老师提供的基础设施代码性能很好。
关于竞赛：竞赛可以适当减省一些时间。

2.对学弟学妹的寄语：
面对大作业，大家不要害怕，切莫手忙脚乱。静下心来认真分析算法的思想策略，可以考虑实现程序的模块化，分小组负责不同的模块，分模块逐一攻克。多查资料，有助于拓展思路。注意及时记录大家的奇思妙想，讨论中迸溅的思维的火花，这是一件非常有意义的事情。

3.对实习作业后续工作的设想：
可以把各个小组的作业编纂成一部书，成为一部黑白棋算法的专业著作，给学弟学妹作为参考。

——*Foxtrot*

在组队的过程中尽量保证队伍之间力量的大小基本相同。队伍内的力量也要均衡一些，不要让队伍里都是强者或者都是弱者，可以强弱参半，互相提高。要不然就会强组很强，弱组很弱，差距很大。对此我的看法是可以由老师经过自己的判断事先组好队伍中的一部分人。比如将平时表现比较好的同学和比较差的同学组合一下先，然后在由同学们自己加入，这样的话就是一半的人是老师选择的，一半的人是自己加入的，“稀释”了强组和弱组，实力在一定程度上是比较均衡的。

在基础设施代码方面，这次的代码的几处漏洞已经被同学们找出来了。那么下一回可以人为的加入一些漏洞，让同学们自己去找，这个也可以记入到小组的成绩当中。

希望数算课越来越好！希望学弟学妹们经过数算这门课之后收获到的不仅仅是知识，还有更多的技能。学弟学妹们活力四射，让数算课越来越好玩！

——*Quebec*

Suggestions
建议与设想

1、组队：组队的方式也许可以再考虑。小组合作的过程也是加深彼此理解的过程，因此，建议增加随机性，以便于不同班级甚至院系之间同学的交流，也会避免出现一些“不得已”而组队的现象。
2、基础设施代码：*Reversi*函数在搜索一些名字的算法的时候似乎会出错，并且原因难以查明，重命名算法后解决。
3、竞赛：感觉小组赛的过程相对枯燥了一些，建议增加对弈的可视化。

这样的大作业对我们的学习还是很有帮助的，所以还希望在以后能有更多有趣的大作业实习项目，也希望能更多的在实习项目中实践提升。

——*Mike*

本次黑白棋大赛总体来讲是非常成功的，大家都热情高涨，积极参与。赛程总体进行得很好，但也稍有美中不足。针对竞赛中的一些不完善的地方，我们组有如下几点建议：

1、在组队方面，确立组长后，希望有一套类似组长对老师报名那样的系统，供组员对组长的报名。在大家选好组长后，再进行人数方面的调整。这可以避免有同学迟迟组不了队的情况。

2、对于竞赛规则的确立，基于本次黑白棋大赛的经验，下一次可以先开组长会议，再公布竞赛细则。这样可以避免规则总是变动，使同学们编写程序思路更明确。

3、要充分考虑比赛当天的时间因素。一些必要的准备工作，比如计分程序、统计分数的*ppt*等，可以提前给场务人员。最好可以提前演练，使正式的比赛更紧凑、流畅。

4、比赛当场，同学们等待结果的期间可以设置一些小的节目，比如人机对战的比赛等。这样会更吸引眼球，避免冷场。

除此之外，针对平时的教学，我们也建议老师可以调整一下讲课的节奏。在前期较简单的部分可以加快进度，比如栈、队列、排序与搜索等。后期的课程，特别是树、图及其相关算法部分，可以放慢一点，讲得更详细一点。而且本次的大作业其实用到“递归”的算法就足以做得很强了，树、图部分的算法却用得并不多。建议老师明年考虑针对这一块儿布置一次大作业，加强同学们的理解。

不得不说，这学期的数算课非常成功！陈斌老师的用心付出、同学们的热情配合，使得这一学期的数算课留下了很多美好的记忆。对即将到来的学弟学妹，我们想说，陈斌老师讲课生动，思路灵活，工作负责，还是常挂微信半夜答疑的好老师。数算大作业也是十分有趣，且富有挑战性。数算不是编程，数算课上获得的思维上的锻炼远比学会一门编程语言重要得多。希望你们能爱上数算，轻松对待这门课程。学长坦言，“这是我大学三年来最好的一门课，没有之一”，而我们作为大一新生，也是在陈斌老师的数算课上第一次体验到了大学课程的异彩纷呈。跟着陈斌老师绝对没错，学弟学妹快来吧！

——*India*

Suggestions
建议与设想

本次实习过程整体的流程、各部分安排都是相当完善的，但在具体实行过程中还是有一些可以改进，具体如下：

组队方面：本次组队为追求各小组实力更均衡，采用根据平时成绩划分组长，然后小组成员自愿报名与组长挑选相结合，但其实有些技术水平非常高的同学可能平时并不追求*A*，只是实现了作业要求的基本功能，而不去追求美观等，这样造成他们平时成绩也许不高，没被选上组长，还有一些大神被选上组长后推掉，这样就造成了许多大神组合，在能力上压制其他组，建议以后可以在挑选组长时参考一下同学意见，也许会更好。

基础设施代码很全面，感觉很好；赛制方面感觉在时间的调控方面可以更进一步，比如说将排名通过设计程序列出，而不是人工计算，这样可以省去许多时间。

实习作业在此就结束了，但本次实习作业只是完成了主要下棋算法，希望日后能真正地完成一次游戏设计，做出图形界面。而且，本次实现的算法还比较初级，后续有机会还可以进一步完善。

在此，本组同学献上对学弟学妹的寄语：数算课很有趣，但想做好还是需要下一番苦功的，希望学弟学妹们可以耐下性子，发掘枯燥程序中的趣味，当你认真走过，等到结果后，你将发现，一切努力都是值得的。

——*Juliet*

建议实习作业可以再提早一段时间布置，比如提前一个月布置，这样会有更多的时间准备；另外基础设施代码最好能够封装为一个*exe*文件，这样可以显得更为简洁。以及竞赛过程可否使用一个更为强大的服务器，并且将赛程安排分散，这样有利于参赛选手调试程序。

——*Kilo*

此次黑白棋竞赛是一个非常好的形式，既部分取代了死板的考试，又以竞赛的形式考察了同学，调动了积极性，是一次十分不错的尝试，组员们也乐在其中，希望以后的数算课程中这样的形式坚持下去。有了今年的基础，黑白棋基础设施代码也有了一定的完善，希望以后参与其中的学弟学妹们能取得更好的成绩，编写出更加强大的程序。

——*Papa*

Suggestions
建议与设想

建议实习作业总时长适当增加。

——*Romeo*

虽然这次实习作业是第一次实行，但是我认为这是我所做过最棒的代码类实习作业。首先，他考验了团队合作能力，在进行分工合作时要事先分配好工作重点以及根据组员所擅长的部分进行调兵遣将。其次，团队之间的比赛竞争更为出彩。将各大战队分为四个版权，最终进行淘汰赛和半决赛、决赛，整个环节紧张刺激、扣人心弦。这种风格、形式的大作业值得保留。

接下来讨论一下需要改进的地方：

1. 根据各组的情况来看，比较强力的组基本都走到了极大极小搜索、剪枝之一步骤，搜索的层数一般在4—5层之间，实力的差别可能就是对于估值函数的理解不同，以及是否考虑行动力、稳定子和行动力的权值。
   通过比赛我们看到，由于我们的搜索层数为5层，那么多搜索一层的优势是不言而喻的，可以说在不超时的情况下，搜索4层的程序很难胜过搜索5层的程序。
   接下来程序的优化主要在剪枝优化、搜索方法改进、估值函数改进之上。
2. 通过阅读文献，我们了解到尽管 α－β 剪枝已经极大地提高了程序的搜索效率，并且使搜索的层数增加了1—2层，然而这种剪枝技术由于顺序地访问棋盘上可以着子的每一个点，所以能否在搜索的一开始就剪掉一枝是改进算法性能优劣之分。通过结合历史表和置换表的 α－β 剪枝技术，我们可以先对所有需要搜索的树枝进行排序，引入历史表机制，保证每次搜索时先搜索非常好的节点，再搜索主要步骤节点，最后搜索非常差的节点，这样发生剪枝的情况就大大增加了，也就使得算法的性能更加稳定、高效。
3. 估值函数的优化，利用机器学习的理论，使程序算法能够自行优化估值函数，然后通过对某一固定强力算法进行仿真模拟，不断进化算法（自动优化），一般跑到一两天之后，程序的估值函数已经能够优化地非常好了，一般的*AI*计数都要经历这一个强化学习的过程，实践证明这也是最强大的估值函数产生的方法。

实验中由于时间限制、没有强力程序代码作为陪练，所以并没有执行这一基本*AI*优化策略，而是人为测试估值函数的优劣，尽管比强化学习算法更为灵活和直观，但是改进效率较低，且比较费时。

——*Lima*

在组好队后应该尽快分配任务并开始编程工作，因为测试往往需要花费较长的时间，这样可以有更多的时间对自己的程序进行修改，希望学弟学妹们能够吸取经验。另外我们认为，还可以在竞赛后增加一个人机对战的环节，请下黑白棋比较厉害的同学与前几名的程序进行对战，看看同学们的程序是否可以打败人脑的智慧。

——*Oscar*


PEKING UNIVERSITY
1898