SESSDSA2020 实习作业

sessdsa.2048 报告合集

本课程报告合集为

北京大学 2020 年春季学期课程

《数据结构与算法(B)》

期末作业课程报告

仅作个人学习用途

禁止未授权使用

课程主讲教师：陈斌

gischen@pku.edu.cn

报告合集整理人：张赖和

2021.2.1

目录

第一部分　SESSDSA 2020 实习作业 sessdsa.2048 介绍……1
第一章 本游戏基础设置……4
第二章 游戏平台实现说明……14
第三章 算法竞赛安排及结果……31
第四章 结语……38

第二部分　N19 联盟各参赛队报告……39
第一章 N19- 2333 组报告……39
第二章 N19- 404 组报告……56
第三章 N19- 666 组报告……73
第四章 N19- 777 组报告……81
第五章 N19- 8848 组报告……93
第六章 N19- 996 组报告……115
第七章 N19- Alpha 组报告……130
第八章 N19- Bravo 组报告……142
第九章 N19- Charlie 组报告……151
第十章 N19- Delta 组报告……168
第十一章 N19- Echo 组报告……186
第十二章 N19- Golf 组报告……206
第十三章 N19- November 组报告……218
第十四章 N19- Papa 组报告……236
第十五章 N19- Romeo 组报告……245
第十六章 N19- Sierra 组报告……267
第十七章 N19- Tango 组报告……279
第十八章 N19- Victor 组报告……288
第十九章 N19- X-ray 组报告……308

第三部分　F19 联盟各参赛队报告……345
第一章 F19-007 组报告……345
第二章 F19-119 组报告……364
第三章 F19-2333 组报告……382
第四章 F19-3721 组报告……394

第五章 F19-404 组报告 411
第六章 F19-520 组报告 426
第七章 F19-666 组报告 444
第八章 F19-777 组报告 463
第九章 F19-8848 组报告 478
第十章 F19-996 组报告 501
第十一章 F19-Alpha 组报告 511
第十二章 F19-Bravo 组报告 529
第十三章 F19-Charlie 组报告 542
第十四章 F19-Delta 组报告 557
第十五章 F19-Echo 组报告 567
第十六章 F19-Foxtrot 组报告 593
第十七章 F19-Golf 组报告 606
第十八章 F19-Hotel 组报告 618
第十九章 F19-India 组报告 637
第二十章 F19-Juliet 组报告 648
第二十一章 F19-Kilo 组报告 680
第二十二章 F19-November 组报告 692
第二十三章 F19-Oscar 组报告 708
第二十四章 F19-Papa 组报告 721
第二十五章 F19-Quebec 组报告 732
第二十六章 F19-Romeo 组报告 749
第二十七章 F19-Sierra 组报告 767
第二十八章 F19-Tango 组报告 782
第二十九章 F19-Uniform 组报告 803
第三十章 F19-Victor 组报告 822
第三十一章 F19-Whiskey 组报告 835
第三十二章 F19-X-Ray 组报告 852
第三十三章 F19-Yankee 组报告 867
第三十四章 F19-Zulu 组报告 881

SESSDSA 2020 实习作业 sessdsa.2048 报告

陈斌[†]、戴天翔[‡]、高云浩[§]、
田晨霄[¶]、王晨冰[‡]、魏一鸣[*]、张宇昊[‡]

† 北京大学地球与空间科学学院，北京大学遥感与地理信息系统研究所

‡ 北京大学物理学院

§ 北京大学元培学院

¶ 北京大学数学科学学院

* 北京大学地球与空间科学学院

摘要

本文档为关于 sessdsa.2048 游戏（由 2048 为基础进行改进的游戏; 以下称本游戏）的实习作业报告，主要讨论游戏系统的相关设置。

本项目存储于以下 GitHub 仓库：

`https://github.com/pkulab409/sessdsa.2048`

本项目的成立目的乃根据相应实习作业要求，建构合适的场地配置以供本游戏使用; 本项目被拆解成多个模块，透过引用关系以完成整体的架构设置。

关键词: 2048, python, C++

实习作业简要介绍

本部分将简要介绍本学期（2020 春季学期）本课程（北京大学地球与空间科学学院数据结构与算法课程）的实习作业（以下称本作业）;

本作业要求本课程的选课同学分成若干小组，其中有一组为技术组，负责对战平台的架构、相关运行脚本、相关可视化程序与简易数据分析程序; 其余小组中，每个小组成立一参赛队，综合本课程内容，采用 Python 语言编写算法，使得提交的算法能进行本游戏;

于技术组提供的对战平台进行对抗性竞赛上，算法应能顺利游玩游戏，并根据与其他算法的对局结果取得竞赛排名;

本游戏简要说明

本游戏的游戏原型为 2048; 原版游戏由 19 岁的意大利人 Gabriele Cirulli 于 2014 年 3 月开发，本游戏改编后加入对战属性。

本游戏为回合制游戏; 每局游戏由两个玩家参与，两个玩家分别下棋和选择方向合并;

玩家可以通过选择一个方向合并; 遇到数值相等的棋子，将吃掉遇到的棋子，占据其位置。

游戏目的乃透过合法操作扩大自己的领地，增大自己的棋子数值，以最终获得更高的数值;

目录

实习作业简要介绍 2

第一章 本游戏基础设置 **4**

1.1 基础定义 4

1.2 游戏进行 5

1.3 游戏结束 11

第二章 游戏平台实现说明 **14**

2.1 对战平台 14

2.2 单循环赛工具 19

第三章 算法竞赛安排及结果 **31**

3.1 赛程赛制安排 31

3.2 分区天梯赛结果展示 33

3.3 2020 地空数算大作业现场竞赛当日时间节点 34

3.4 淘汰赛结果 36

第四章 结语 **38**

4.1 致谢 38

4.2 设想与展望 38

4.3 主要参考资料 38

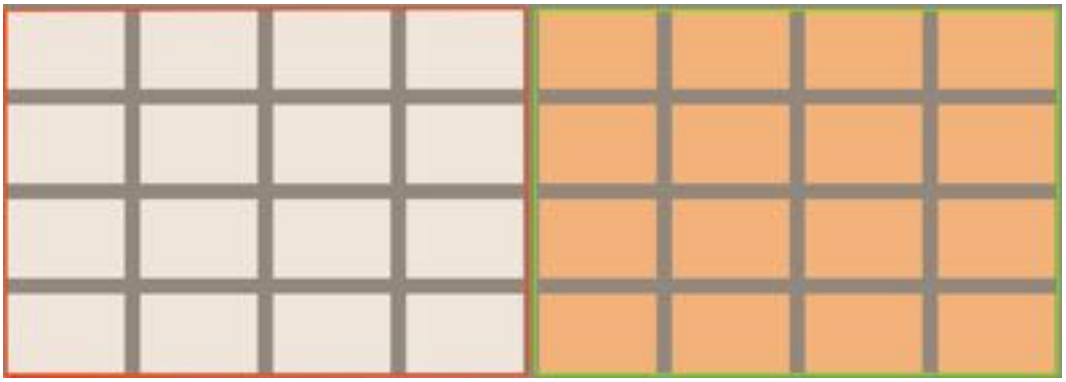
第一章　本游戏基础设置

1.1　基础定义

1.1.1　棋盘和领地

棋盘是一个 4 行 8 列共 32 个方格组成的游戏场地，左右两半分别称为先手、后手方的领地。在图1.1中红色框和绿色框分别为先手、后手方的领地。

图 1.1: 棋盘和领地



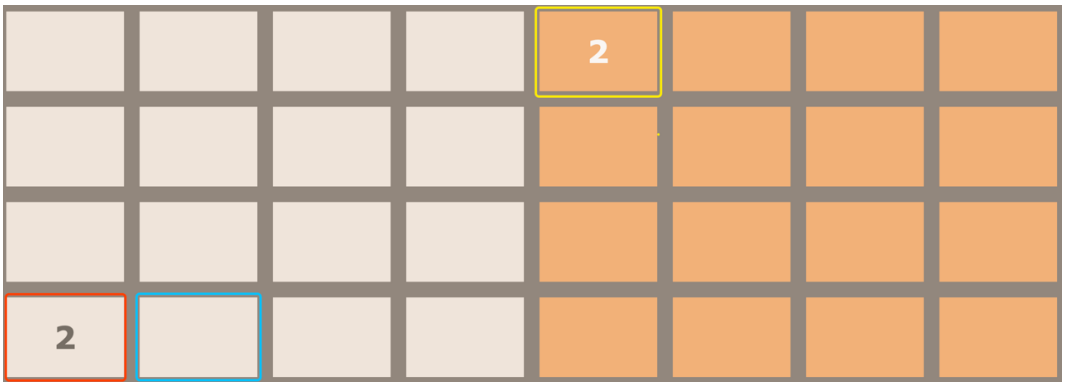
inclu

1.1.2　方格、空位和棋子

棋盘上的方格如果为空，则称此方格为一个空位；否则称此方格处有一个棋子，棋子具有数值和归属。在图1.2中红色框、蓝色框和黄色框分别是先手方的 2 棋子、先手方的空位和后手方的 2 棋子。

在某方的领地内的空位归该方所有。

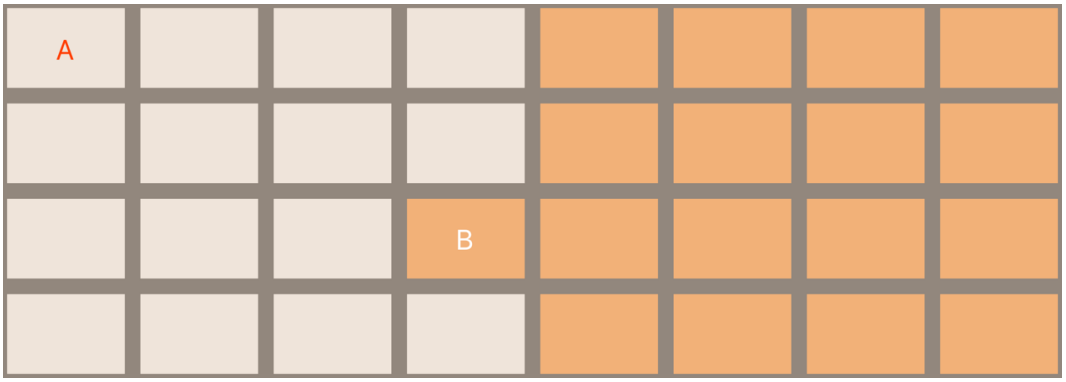
图 1.2: 空位和棋子



1.1.3　方向和靠前

在棋盘上定义有上、下、左、右四个方向。A 相对于 B 沿某方向靠前指 A 在 B 的该方向，即 B 到 A 的向量与该方向向量夹角小于 90 度。在图1.3中，A 相对于 B 沿向左靠前，A 相对于 B 沿向上靠前，B 相对于 A 沿向右靠前，B 相对于 A 沿向下靠前。

图 1.3: 方向和靠前



1.2　游戏进行

1.2.1　基本流程

游戏开始后，每个回合按照四个阶段顺序执行。

1. 先手方下一个 2 棋子

2. 后手方下一个 2 棋子

3. 先手方进行一次合并

4. 后手方进行一次合并

如果某一阶段某方没有位置可下棋或没有合法的合并方向，那么跳过该阶段。

1.2.2　下棋

某方在下棋时需选择其一

1. 在对方的空位中下棋，位置由己方决定，棋子归对方所有

2. 在己方的空位中下棋，位置被随机决定，棋子归己方所有

需要注意的是，在第二种情况中的随机是由游戏开始时公开的随机序列和计算位置的算法计算得到的，即某方在下棋之前已经知道如果选择在己方下棋，棋子将出现的位置。在图1.4中，对于先手方而言，在所有空位中，绿色框内为可下棋位置，所下棋子归对方所有；蓝色框内为随机决定的可下棋位置，所下棋子归己方所有；红色框内为不可下棋位置。

图 1.4: 下棋

1.2.3　合并

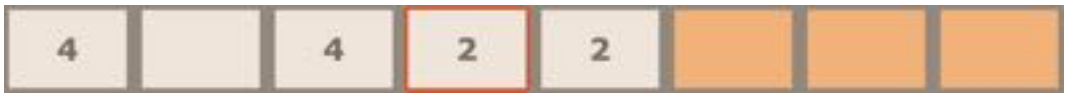
某方选择某方向进行一次合并描述的是这样的过程：本方棋子按照沿该方向靠前者先行的次序向该方向运动。如果遇到己方空位，将占领空位并继续运动；如果遇到对方空位，不占领空位并停下；如果遇到数值相等的棋子，将吃掉遇到的棋子，占据其位置，并将自身

数值修改为原数值的两倍，停下；如果遇到数值不等的棋子，不吃棋并停下。一个棋子如果吃掉了另一个棋子，在本次合并内不能被吃。

以先手方选择向右合并为例，分析某一行的合并动态过程如下，这里用红色框框出正在运动的棋子。

首先先手方的沿右最靠前的 2 棋子运动。

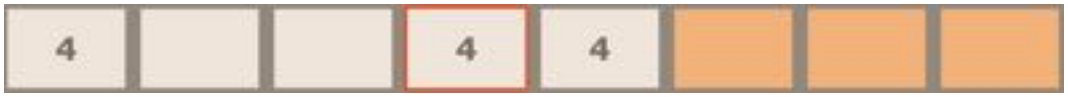
遇到对方空位，停下。接下来是沿右第二靠前的 2 棋子运动。



遇到数值相同的棋子，吃掉，占据位置并数值翻倍，停下。

接下来是第三靠前的 4 棋子运动。

遇到己方空位，占领空位并继续运动。



遇到本次合并中吃过棋的棋子，所以直接停下。第四靠前的 4 棋子运动

遇到己方空位，占领空位并继续运动。

遇到己方空位，占领空位并继续运动。

遇到数值相同的棋子，吃掉，占据位置并数值翻倍，停下。

至此，这一行中先手方的所有棋子都已经停下，合并结束。其他行同理。

再分析一个例子。同样先手方选择向右合并。

首先先手方向右最靠前的 2 棋子运动。

遇到数值相同的棋子，吃掉，占据位置并数值翻倍，停下。

接下来是第二靠前的 2 棋子运动。

遇到对方空位，停下。合并结束。

这个例子里需要注意的是，当第一个 2 棋子吃棋之后，其原先占据的位置变成了空位，而空位的归属依据的是领地，所以该空位归后手方所有。

这里还提供了一些典型情况以加深对合并规则的理解。

1. 先手方选择向右合并

图 1.5: 合并前

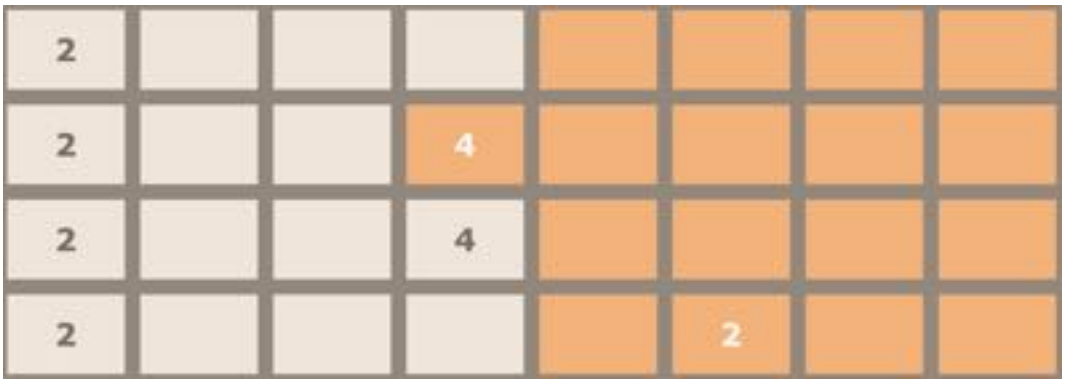
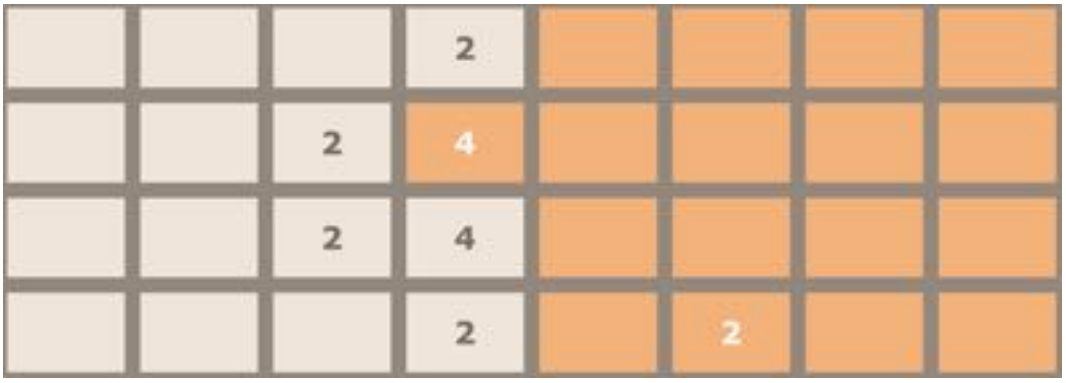


图 1.6: 合并后



2. 先手方选择向右合并

图 1.7: 合并前

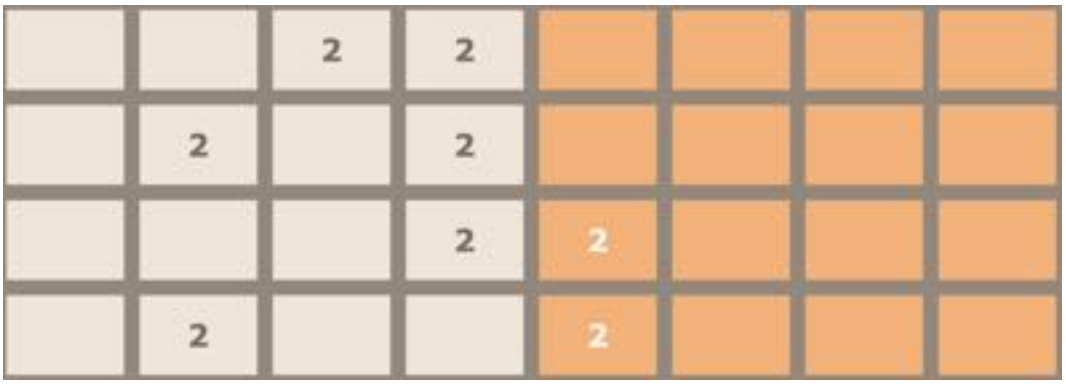
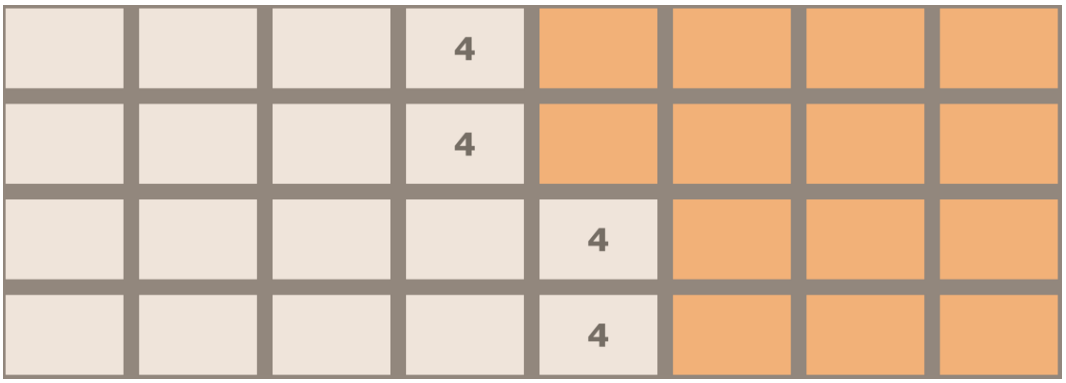


图 1.8: 合并后



3. 先手方选择向右合并

图 1.9: 合并前

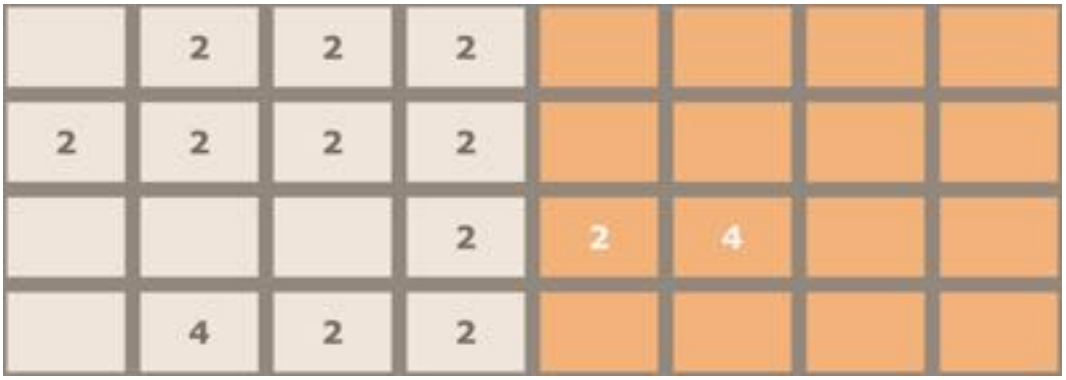
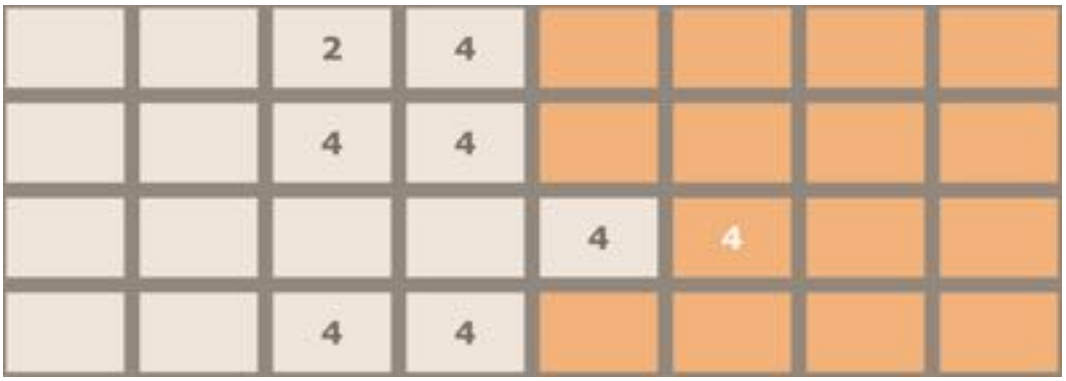


图 1.10: 合并后



4. 先手方选择向左合并

图 1.11: 合并前

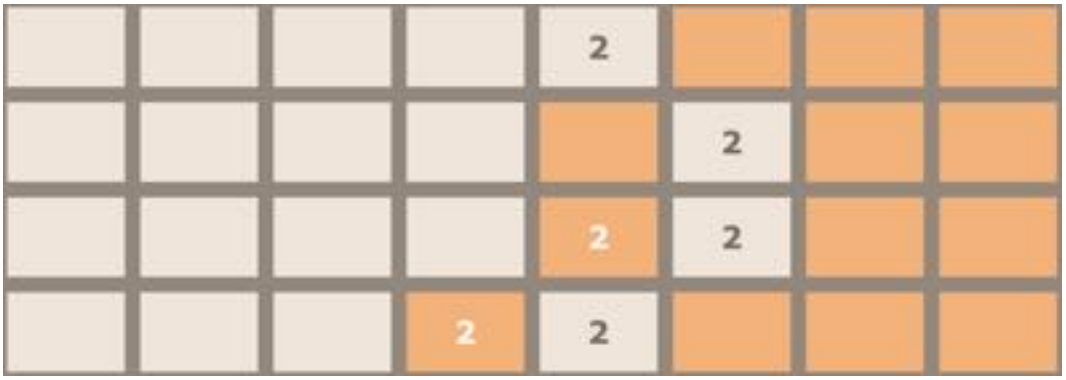
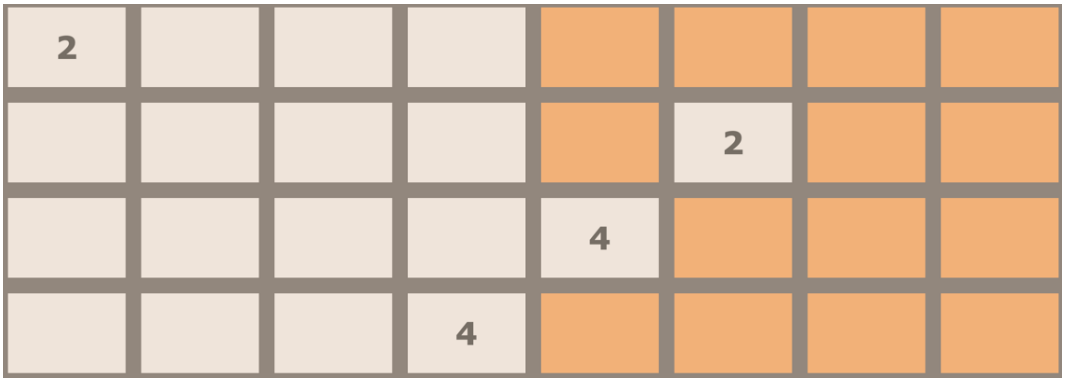


图 1.12: 合并后



1.3　游戏结束

1.3.1　正常结束

当已完成的回合数达到最大回合数时，游戏正常结束，进入计分环节。

1.3.2　意外结束

当出现以下情况时，游戏意外结束。

- 某方累计运行时间超出最大时间限制

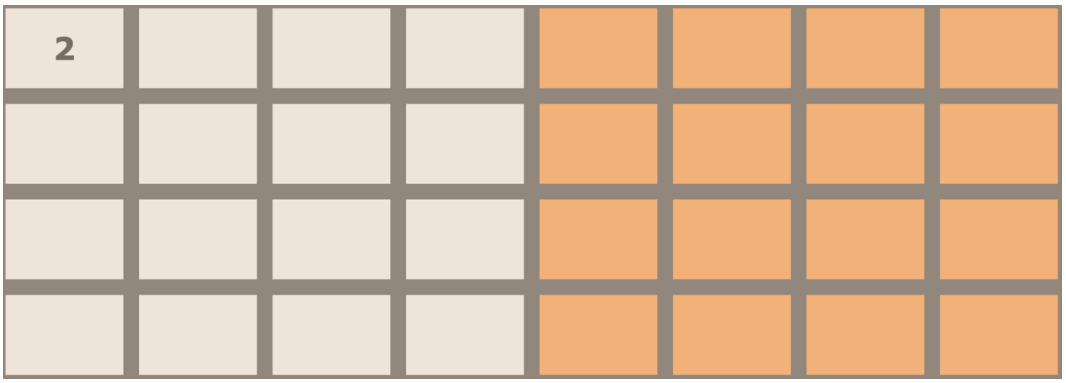
- 某方运行时发生错误
- 某方产生非法输出

非法输出有以下几类。

- 输出结果不合接口要求
- 在不被允许的位置下棋
- 进行无效合并，即合并前后整个棋盘未发生变化

例如，在图1.13中，先手方选择在 2 棋子处下棋，为第二种非法输出；先手方选择向左合并，为第三种非法输出。

图 1.13: 非法输出



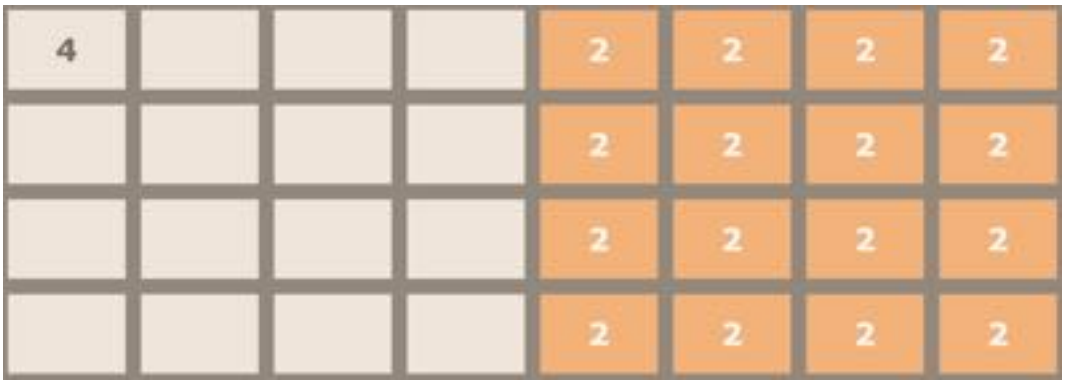
使游戏意外结束的一方将直接判负，进入计分环节。

1.3.3　计分与胜负

计分

将双方的棋子按数值高到低的顺序比较个数，一旦某方多即认定该方分数更高。例如，在图1.14中，先手方分数更高。

图 1.14: 计分



胜负

如果某方已经在意外结束时判负，则对方胜出；否则分数高者胜出。

第二章　游戏平台实现说明

2.1　对战平台

2.1.1　说明对象

plat.py 的 Platform 类。

2.1.2　类变量

1. self.states: 参赛 ai 的运行状态，为 dict 类型，遵循 states 的数据协议。
2. self.match: 比赛名称（保存路径），为有效 str 的格式路径。
3. self.livequeue: 用于直播线程通信的队列，为 queue.Queue 类型。
4. self.toSave: 是否保存记录文件，为 bool 类型。
5. self.maxtime: 最大时间限制，为 int 类型。
6. self.rounds: 总回合数，为 int 类型。
7. self.winner: 胜利者，为 bool 类型或者 None；为 True 代表先手，为 False 代表后手，为 None 代表无。
8. self.violator: 违规者，为 bool 类型或者 None；为 True 代表先手，为 False 代表后手，为 None 代表无。
9. self.timeout: 超时者，为 bool 类型或者 None 或者字符串'both'；为 True 代表先手，为 False 代表后手，为 None 代表无，为'both' 代表先后手。
10. self.error: 报错者，为 bool 类型或者 None 或者字符串'both'；为 True 代表先手，为 False 代表后手，为 None 代表无，为'both' 代表先后手。
11. self.currentRound: 当前轮数，为 int 类型，值为 0 代表第一轮。

12. `self.change`: 某次操作中棋盘是否发生改变，为 `bool` 类型。
13. `self.next`: 根据随机序列得到的下一个位置，为 tuple 类型，遵循 position 的数据协议。在极特殊的情况下，self.next=(`None`, `None`)，代表在先后手方初始化的过程中抛出了异常。
14. `self.board`: 棋盘，为 `Chessboard` 类的实例。
15. `self.log`: 日志，为平台内定义的 `Log` 类的实例。`Log` 类继承 `list` 类，可以依据 `list` 类的接口操作逐条存储的日志记录。单条日志记录遵循 `log` 的数据协议。
16. `self.name`: 本局比赛的名称，为 `repr` 格式的哈希字符串。

2.1.3　类方法

1. `__init__(self, states, match, livequeue, toSave, MAXTIME, ROUNDS)` 初始化平台对象，为参赛 ai 重载方法使其具有计时与捕获异常的功能。参数表如下
   - `states` 与 `self.states` 遵循相同约定，但是参赛 ai 对象应当只实例化而未经初始化。(即使用 `object.__new__(cls)` 实例化对象)
   - `match` 与 `self.match` 遵循相同约定。
   - `livequeue` 与 `self.livequeue` 遵循相同约定。
   - `toSave` 与 `self.toSave` 遵循相同约定。
   - `MAXTIME` 与 `self.maxtime` 遵循相同约定。
   - `ROUNDS` 与 `self.rounds` 遵循相同约定。
2. `play(self)` 初始化参赛 ai 并进行单局比赛，返回比赛报告。返回值遵循 result 的数据协议。
3. `start(self)` 进行比赛，**不支持外部调用**。
4. `checkState(self, isFirst)` 检查超时和报错，**不支持外部调用**。
5. `checkViolate(self, isFirst, mode, value)` 检查非法输出，**不支持外部调用**。
6. `save(self)` 保存比赛记录文件，遵循记录保存数据协议，**不支持外部调用**。

2.1.4　数据协议

1. states

```
states = {
        True: {
                'player': player,
                'path': path,
                'time': time,
                'time0': time0,
                'error': error,
                'exception' : exception,
                'index': index
        },
        False: {
                'player': player,
                'path': path,
                'time': time,
                'time0': time0,
                'error': error,
                'exception' : exception,
                'index': index
        }
}
```

键 True 与 False 分别代表先后手，其对应的值是其运行状态，说明如下

- `player` 参赛 ai 对象，为参赛队伍提供的类的实例。
- `path` 参赛队伍提供的类的路径或者类本身 (视读取的时候是按路径读取还是按类读取)，为有效的 `str` 格式路径或类。
- `time` 参赛 ai 的当前时间，为 `float` 类型。
- `time0` 参赛 ai 的时间零点，为 `int` 或 `float` 类型，通过调节此变量可以实现对特定参赛 ai 的总时间限制进行修改。
- `error` 参赛 ai 是否抛出异常，为 `bool` 类型。

- `exception` 参赛 ai 运行时抛出的异常信息，为标准 traceback 字符串或者 (当未抛出异常时)`None`。
- `index` 参赛 ai 的编号，为 `tuple` 类型，遵循 index 的数据协议。

2. log

   单条 log 记录由 & 起始。分为三类

   - &d? : … 某方作出决策

     这里的? 与…分别为当前轮数与决策说明。作为例子，在第一轮先手方选择向右合并，将会生成记录

     ```
     &d0: player 0 set direction right
     ```

     而在第五轮后手方选择在 (1, 4) 下棋，将会生成记录

     ```
     &d4: player 1 set position (1, 4)
     ```

   - &p? : … 当前棋盘

     这里的? 与…分别为当前轮数与棋盘的 repr 格式。作为例子，第 43 轮的某个棋盘记录为

     ```
     &p42:
     +00 +00 +00 +03 +02 -00 -00 -00
     +00 +00 +01 +02 -03 -01 -00 -00
     +00 +05 +03 +01 -05 -03 -01 -00
     +00 +00 +01 +05 -02 -02 -04 -01
     ```

   - &e: … 游戏事件触发

     这里的…为事件说明。游戏事件有：违规、超时、报错、胜利、计分、平局。作为例子，列举一些记录如下

     ```
     &e: player 1 violate by illegal output of direction
     &e: player 0 win
     ```

3. result

```
result= {
        True: {
                'index' : index,
                'win': win,
                'lose': lose,
                'violate': violate,
                'timeout': timeout,
                'error': error,
                'time': time,
                'exception': exception
        },
        False: {
                'index' : index,
                'win': win,
                'lose': lose,
                'violate': violate,
                'timeout': timeout,
                'error': error,
                'time': time,
                'exception': exception
        },
        'name' : name,
        'rounds' : rounds
}
```

键 name 与 rounds 对应的值分别为本局比赛名称与比赛实际进行总轮数，分别遵循 self.name 和 self.rounds 的约定。键 True 与 False 分别代表先后手，其对应的值是其表现报告，说明如下

- index 参赛 ai 的编号，为 tuple 类型，遵循 index 的数据协议。
- win 是否胜利，为 bool 类型。

- lose 是否失败，为 bool 类型。
- violate 是否违规，为 bool 类型。
- timeout 是否超时，为 bool 类型。
- error 是否报错，为 bool 类型。
- time 参赛 ai 总运行时间，为 int 或 float 类型。
- exception 参赛 ai 运行时抛出的异常信息，为标准 traceback 字符串或者 (当未抛出异常时)None。

4. position

   position = (row, column)

   row, column 为 int 类型，代表位置的行、列坐标

5. index

   index = (count, isFirst)

   其中 count 为 int 类型，是按顺序编定的参赛模块编号；isFirst 为 bool 类型，代表是否先手。

6. 比赛记录保存

   比赛记录将保存为标准 txt 文件，包含该局比赛的元信息和全部日志记录。比赛记录从上而下分别记录了模块的路径 (或模块本身)、比赛时间、比赛结果和全部比赛日志。此比赛记录可以被复盘工具解析并实现可视化，利于开发者进行调试。

2.2　单循环赛工具

此工具可以实现数个参赛队伍之间进行指定重复次数的先后手交替单循环比赛，并在指定文件夹中将比赛记录归档整理，同时具有对全部比赛的统计功能。

2.2.1　说明对象

`round_match.py` 的 `main` 函数

2.2.2　参数表

```
main(
        playerList,
        savepath = None,
        livequeue = None,
        toSave = True,
        toReport = True,
        toGet = False,
        debug = False,
        REPEAT = c.REPEAT,
        MAXTIME = c.MAXTIME,
        ROUNDS = c.ROUNDS
)
```

1. 必填参数 playerList: 参赛队伍的模块列表

   playerList = [module, module, ...] module 支持绝对路径、相对路径和已读取的类。以下都是正确调用的例子。

   ```
   >>> main(['some/absolute/path/to/player.py',
   ... 'another/absolute/path/to/player.py'])
   >>> main(['../player.py'] * 3)
   >>> from player import Player as p
   >>> main(['player.py', p])
   ```

   如果将某个 module 改为元组 (module, time0)，将指定该模块计时零点为 time0(此值默认为 0)，通过调整计时零点可以延长或缩短对特定模块的时间限制。使用的例子如下。

   ```
   >>> # 给第二个模块的时间限制+1s
   ```

```
>>> main(['player.py', ('player.py', -1)])
```

2. 可选参数 savepath: 比赛文件夹的保存路径

   savepath 支持绝对路径、相对路径和函数返回值；当参数缺省时，在当前文件夹下生成以当前时间命名的比赛文件夹。以下都是正确调用的例子。

```
>>> # 缺省调用
>>> main(['player.py', 'player.py'])
>>> # 路径调用
>>> main(['../player.py'] * 2, savepath='../match')
>>> # 函数返回值调用
>>> import random
>>> main(['player.py'] * 2,
... savepath = lambda : str(random.random()))
```

   如果该路径下已经存在同名文件夹，将创建文件夹副本以存放比赛记录。例如，match 文件夹名被两次占用后，产生副本 match_2，match_3。

3. 可选参数 livequeue: 直播中线程通信用的队列

   模块 queue 中的 Queue 对象，缺省值为 None，表示直播模式关闭。此队列需要在直播工具中用到，**对抗算法开发者无需手动设置**；赛制工具开发者可以在设定通信队列之后接入即可。

```
>>> q = __import__('queue').Queue()
>>> main(['player.py'] * 2, livequeue = q)
```

   为了保证直播观看体验，在比赛中设置了等待时间，所以运行总时间将会很长。

4. 可选参数 toSave: 是否保存单局详细对局记录文件

   bool 类型，缺省值为 True。

5. 可选参数 `toReport`: 是否生成全部比赛统计报告

   bool 类型，缺省值为 True。

6. 可选参数 `toGet`: 是否返回全部对战平台对象

   bool 类型，缺省值为 False。

7. 可选参数 `debug`: 是否生成报错信息报告

   bool 类型，缺省值为 False。

8. 可选参数 `REPEAT`: 单循环轮数

   int 类型，缺省值为默认单循环轮数

9. 可选参数 `MAXTIME`: 最大时间限制

   int 类型，缺省值为默认最大时间限制

10. 可选参数 `ROUNDS`: 总回合数

    int 类型，缺省值为默认总回合数。

一点提示，如果需要保存某些特定的参数组合，可以使用字典传入参数。

2.2.3　返回值

一般情况下，函数没有返回值。若 toGet 参数被设置为 True，函数将返回全部比赛的平台对象，遵循 result 的数据协议。

2.2.4　数据协议

1. result

   `result = {(i, j): [platform1, platform2, ... ], ... }` 有序数对 (i, j) 表示 playerList 中下标为 i, j 的模块分别作为先后手。一个数对对应的值为多轮比赛的平台对象列表 (长度为 REPEAT)。

2. 全部比赛统计报告

   在比赛文件夹中保存为 _.txt 文件，包含全部比赛的统计信息。

   作为例子，某次统计报告 (部分) 如下。

```
========================
total matches: 20
name: 5340670467737 -> player0 to player1 -> 114 rounds
...
...
...
name: 7981237979477 -> player0 to player1 -> 514 rounds
player0 from path player.py

offensive cases:
    average time: 0.006
        win rate: 80.00%
        win 8 at
            457295662957825
            ...
            ...
            ...
            835793467976796
        lose 2 at
            427956796709276
            821917974987957
        violate 1 at
            573946794857609
        timeout 0 at

        error 0 at

defensive cases:
    average time: 0.001
        win rate: 0.00%
        win 0 at
```

```
        lost 10 at
            479275973469798
            ...
            ...
            ...
            629145694567296
        violate: 10 at
            579346290609267
            ...
            ...
            ...
            346856926828972
========================
```

首先是按每局比赛的统计报告。从上而下分别记录了比赛总局数，每局比赛的名称、比赛队伍与实际回合数。

之后是按参赛模块的表现统计报告。包括其路径 (或类本身)，在先手方的平均耗时、胜率、胜利场次、失败场次、违规场次、超时场次、报错场次、对应比赛名称，以及在后手方的相应信息。

根据此报告，再在比赛文件夹中找到对应的详细比赛记录，可以分析参赛模块的表现，有利于开发。

3. 报错信息报告

   在比赛文件夹中保存为 _Exceptions.txt 文件，包含每局比赛的报错信息。如果没有报错，将输出 pass。某次报错信息报告 (部分) 如下所示。

```
-> 37890357726976


offensive:
pass
```

```
defensive:
Traceback (most recent call last):
    File "some/file", line 114, in wrappedFunc
        result = func(*args, **kwargs)
    File "some/file", line 514, in __init__
        raise SomeException
Exception
====================================
-> 34729579347695

offensive:
Traceback (most recent call last):
    File "some/file", line 114, in wrappedFunc
        result = func(*args, **kwargs)
    File "some/file", line 514, in __init__
        raise SomeException
Exception

defensive:
pass
```

4. 本地直播工具

   此工具可以实现比赛的实时本地直播，使比赛更有观赏性，对于开发和调试用处并不很大。

   - 说明对象

     livetool.py 的 Live 类

   - 简介

     此类唯一提供的外部调用接口为 ___init___，所以接下来只介绍 ___init___ 的参数表；运行时，只需如示实例化即可观看直播。

     ```
     >>> Live()
     ```

- 参数表

```
__init__(
    self,
    mode = 0,
    func = __import__('round_match').main,
    playerList = ['player. py' for _ in range(2)]
)
```

  - 可选参数 mode: 可视化版本
    为 0 或 1 分别对应 2048 原版可视化和**柚子社版**可视化，缺省值为 0。
  - 可选参数 func: 赛制工具
    目前开发的只有单循环赛工具可用，缺省值为单循环赛工具。
  - 可选参数 playerList: 参赛队伍的模块列表
    遵循与单循环赛工具相同的约定,缺省值为 `['player. py' for _ in range(2)]`，即两个样例随机 ai 比赛。

需要提醒的是，由于 tkinter 对不同系统的兼容性并不好，而编写者使用的是 macOS，所以此 UI 可能在 win 界面显示异常。

5. 对抗算法开发者工具

为了方便对抗算法开发，我们编制了面向对抗算法开发者的工具。

(a) 逻辑约定

- 坐标
  坐标为 tuple 变量，具有结构 (row, column)，row 行坐标，从上到下为 0 到 3 的 int 变量；column 列，从左到右为 0 到 7 的 int 变量。
- 方向
  方向为 int 变量，0、1、2、3 分别代表上、下、左、右。
- 归属
  归属为 bool 变量，True 为先手方，False 为后手方。

(b) ai 接口规范化

参赛队伍的 ai 须以以下方式实现。

实现一个 Player 类，并实现两个方法。

i. `__init__(self, isFirst, array)` 初始化

A. 参数:isFirst 是否先手，为 bool 变量。

B. 参数:array 随机序列，为 tuple 变量。

ii. `output(self, currentRound, board, mode)` 给出己方的决策

A. 参数:currentRound 当前轮数，与对战平台类变量 self.currentRound 遵循相同约定。

B. 参数:board 棋盘对象的拷贝，是 Chessboard 类的实例。

C. 参数:mode 模式，mode = position 对应位置模式，mode = direction 对应方向模式，如果 mode 为 '*position*' 和'direction' 表示在对应模式下己方无法给出合法输出。

D. 返回: 位置模式下返回下棋的坐标，方向模式下返回合并的方向，在己方无法给出合法输出时, 对返回值不作要求。

(c) 棋盘对象 Chessboard

棋盘对象提供了对棋盘的基本操作接口，参赛 ai 可以直接调用。接口如下

i. `__init__(self, array)` 初始化棋盘对象

参数:array 随机序列

ii. `add(self, belong, position, value = 1)` 在指定位置下棋

参数:belong 操作者，position 位置坐标，value 下棋的级别，缺省值为 1

注: 棋子的数值取以 2 为底的对数即为其级别，为 int 变量。

根据规则，实际上 belong 参数会被忽略，不应当对其意义做出假定。

iii. `move(self, belong, direction)` 向指定方向合并，并且返回棋盘是否变化

参数:belong 操作者，direction 合并方向

返回: 棋盘是否变化，为 bool 类型

iv. `getBelong(self, position)` 根据位置得到归属

参数:position 位置坐标

返回: 该位置上的棋子的归属，若为空位，则返回空位的归属

v. `getValue(self, position)` 根据位置得到级别

参数:position 位置坐标

返回: 该位置上的棋子的级别，若为空位，则返回 0

vi. `getScore(self, belong)` 获取升序排列的某方的全部棋子的级别列表

参数:belong 某方

返回: 由该方全部棋子的级别构成的 list 类型变量

vii. `getNone(self, belong)` 获取某方的全部空位的位置列表

参数:belong 某方

返回: 由该方全部空位的位置构成的 list 类型变量

viii. `getNext(self, belong, currentRound)` 获取某方在本方领域允许下棋的位置

参数:belong 某方，currentRound 当前轮数

返回: 该方在本方领域允许下棋的位置，若不可下棋，返回空元组

ix. `copy(self)` 返回一个对象拷贝

返回: 深拷贝的棋盘对象

x. `__repr__(self)` 或 `__str__(self)` 棋盘的可打印字符串

打印结果如下所示。

+01 +01 +04 +00 -05 -01 -04 -00

+01 +01 +04 +00 -05 -01 -04 -00

+01 +01 +04 +00 -05 -01 -04 -00

+01 +01 +04 +00 -05 -01 -04 -00

xi. `getTime(self, belong)` 返回用户 AI 剩余思考时间, 单位为秒.

参数:belong 某方

返回:float 类型变量

xii. `updateTime(self, belong, time)` 更新用户 AI 的剩余思考时间. 用户无需调用.

参数:belong 某方，time 剩余时间

xiii. `getDecision(self, belong)` 获取某方上一次决策时给出的结果

参数:belong 某方

返回: 若无决策, 返回 `tuple()`. 若为决定合并方向, 为 `tuple(direction)`. 若为决定下棋位置, 为 `(int, int)`.

xiv. `updateDecision(self, belong, decision)` 更新用户 AI 的决策信息. 用户无需调用.

参数:belong 某方，decision 决策

xv. `getAnime(self)` 给出上一次棋盘合并时棋子的移动信息. 用户无需调用.

undocumented

xvi. `getRaw(self)` 给出二维列表形式的棋盘

返回: 一个 4*8 的二维 list 类型, 为 `[[(value, belong) for column in range(8)] for row in range(4)]`, 这里 value 和 belong 分别为该位置的级别和归属。

(d) 复盘可视化 analyser

（你一定很好奇为什么整个技术组都把这个词拼错了）

i. tkinter 版复盘程序

解析对局记录文件并可视化复盘，为调试提供方便。

采用交互式设计，直接运行即可，根据终端提示依次输入对局文件名和可视化版本，即可进入可视化界面。

按键盘的 [和] 分别对应前进、后退操作。

需要提醒的是，由于 tkinter 对不同系统的兼容性并不好，而编写者使用的是 macOS，所以此 UI 可能在 win 界面显示异常。

而从 neattool.py 可以直接实现单循环赛的自动可视化，此时按键盘的 n、p 对应观看下一局、上一局操作（next、previous）。

ii. 网页版可视化程序

考虑到跨平台时 UI 可能出现异常，因此产生了网页版复盘程序。

浏览器打开 `src/analyser.html`，并按提示上传单场比赛记录即可。操作方式与前者一致。

此外，`src/analyser_dir.html` 是处理一整局循环赛的工具。由于浏览器的跨域限制，使用方法稍微麻烦一些。

设某个文件夹 `dir` 既包含 `src/analyser_dir.html` 和 `src/analyser_dir.js`，也包含你的比赛记录文件夹,那么在目录 `dir` 下执行 `python -m http.server`，并用浏览器打开 http://localhost:8000，你应当可以看到网页 (记为标签页 a) 上显示了 `dir` 文件夹的内容。

```
D:\sessdsa.2048\dir> python -m http.server
```

随后在其中选中 src/analyser_dir.html 并新标签 (记为标签页 b) 打开。再在网页 a 上进入你的比赛记录文件夹，也就是包含 _.txt 的那个文件夹，在地址栏复制其链接，按提示填入标签页 b 的输入框中。

此时点击一条记录即可查看，可以按 [] 或左右方向键调整比赛进程。

(e) 网页版人机对战工具

在单循环赛中将一方设为 src/human.py，即可自动打开浏览器页面。位置模式下鼠标点击方块可选取下棋的方块，方向模式下按键盘的方向键选取方向。目前错误处理尚不完善，例如没有断线重连功能。

```
>>> import round_match
>>> round_match.main(["player",("human", -30)], MAXTIME
    =114514)
```

此时浏览器应该会打开一个页面。在方向模式下用方向键给出方向，在位置模式下点击想放置棋子的方块给出位置。

第三章　算法竞赛安排及结果

3.1　赛程赛制安排

- F19 联盟：35 组 12 组出线，产生冠军 1，亚军 1，季军 2
- N19 联盟：20 组 8 组出线，产生冠军 1，亚军 1，季军 1
- 分为 F19 和 N19 两个联盟进行比赛
- 所有单败淘汰赛对战双方先后手各对战 10 局，共 20 局
- 比赛必须分出胜负，如果 20 局打平的话，则双方抽签决定先后手，用可视化对战一局分胜负。
- F19 联盟前 12 名出线，其中前 4 名第一轮轮空，如图分配直接进入 8 强赛上下半区；5～12 名随机分配进入第一轮单败淘汰赛，胜者进入 8 强赛上下半区开展对战。

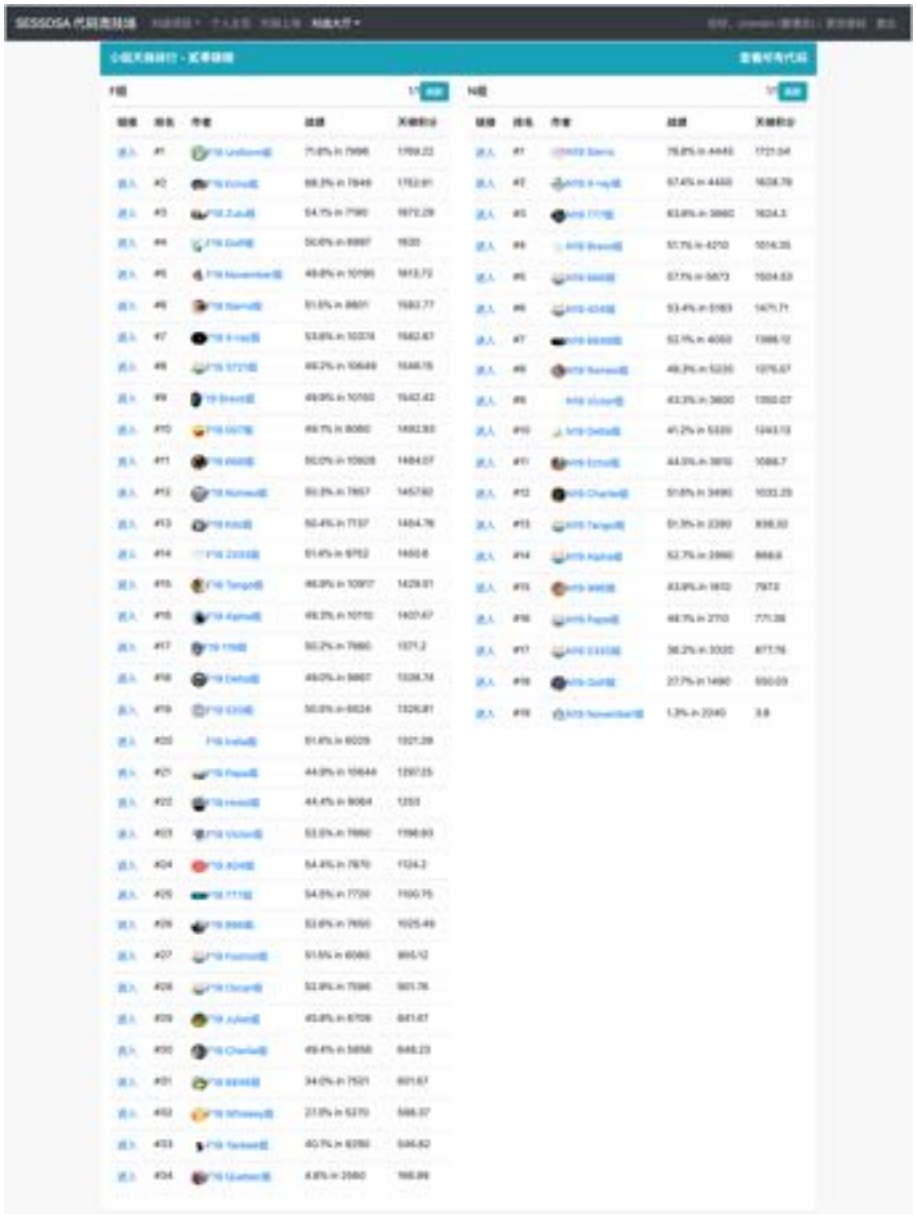
图 3.1: F19 赛制

- N19 联盟前 8 名出线，其中前 4 名按如图分配直接进入 8 强赛上下半区；5～8 名随机分配进入 8 强赛上下半区开展对战。

图 3.2: N19 赛制

3.2　分区天梯赛结果展示

图 3.3: 天梯赛结果



出线名单

F19 赛区：

第一名 F19 Uniform 组

第二名 F19 Echo 组

第三名 F19 Zulu 组

第四名 F19 Golf 组

第五名 F19 November 组

第六名 F19 Sierra 组

第七名 F19 X-ray 组

第八名 F19 3721 组

第九名 F19 Bravo 组

第十名 F19 007 组

第十一名 F19 666 组

第十二名 F19 Romeo 组

N19 赛区：

第一名 N19 Sierra 组

第二名 N19 X-ray 组

第三名 N19 777 组

第四名 N19 Bravo 组

第五名 N19 666 组

第六名 N19 404 组

第七名 N19 8848 组

第八名 N19 Romeo 组

3.3　2020 地空数算大作业现场竞赛当日时间节点

日期：2020 年 6 月 2 日

- 15:00 - 15:10
  - 【竞猜 1】19 级同学填写 F19 晋级赛竞猜表单
- 15:10 - 15:25

  - 陈老师开场
  - 现场竞赛正式开始
- 15:25 - 15:30
  - 【SSH】F19 晋级赛，产生 F19 全部八强（12 进 8）
  - （共同解说）
- 15:30 - 15:40
  - 更新记分板
  - 【竞猜 2】所有同学分联盟填写八进四竞猜表单
  - 【网页】解说回顾晋级赛精彩对局
- 15:40 - 15:50
  - 【SSH】F/N19 八进四同时展开，产生四强
- 15:50 - 16:00
  - 更新记分板
  - 【竞猜 3】所有同学分联盟填写四强名次竞猜表单
  - 【网页】解说回顾八进四精彩对局
- 16:00 - 16:15
  - 【SSH】F/N19 半决赛同时展开，产生决赛对阵双方
  - 更新记分板
  - 【网页】解说回顾半决赛精彩对局
  -

  16:15 - 16:30
  - 【SSH】N19 季军争夺赛
  - 【SSH】F/N19 决赛同时展开，产生分联盟冠军
  - 更新记分板
  - 【网页】解说回顾决赛精彩对局
  -

16:30 - 16:45

- 【SSH】友谊赛产生总冠军
- 更新记分板
- 【网页】解说回顾友谊赛精彩对局

- 16:45 - 17:00
  - 公布菠菜王和卷心菜王
  - 【PPT】布置期末作业
  - 陈老师总结

3.4　淘汰赛结果

图 3.4: F19 淘汰赛结果

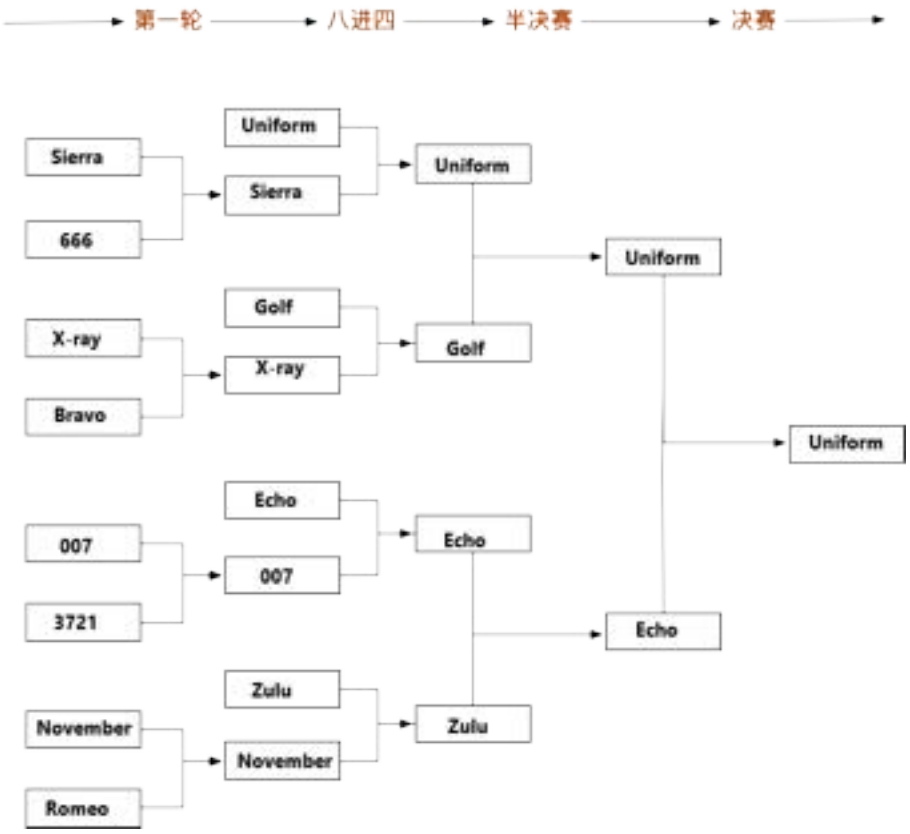
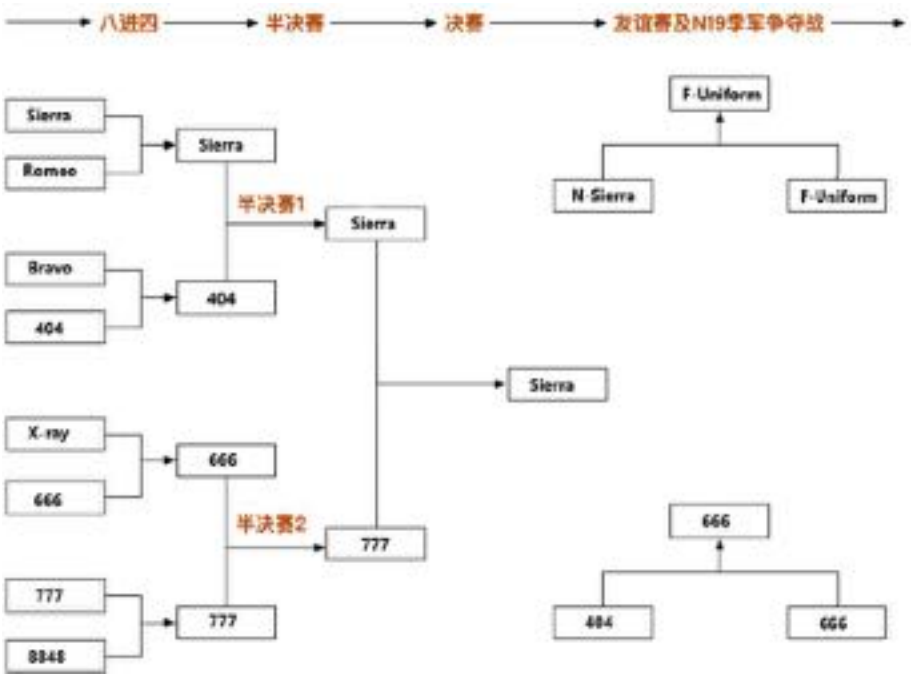


图 3.5: N19 淘汰赛结果



第四章　结语

4.1　致谢

感谢本课程授课老师陈斌老师进行技术指导; 感谢北大地空数算课程的代码编写者提供参赛代码; 感谢本课程助教陈天翔、张赖和、袁泽、冀锐、魏江南、杨佑铭等提供协助。

4.2　设想与展望

本项目可能具有部分的代码冗余或可进一步优化的部分; 本项目可能存在其他数据分析手段; 本项目可能发展为校际或以上规模竞赛; 此二部分可供接续的项目实施。

4.3　主要参考资料

1. 数据结构与算法:Python 语言描述, 裘宗燕著, 北京: 机械工业出版社,2015 年 12 月

2. 2048 - Wikipedia, `https://en.wikipedia.org/wiki/2048_(video_game)`, 2020-06-06

数据结构与算法课程实习作业报告

秋昱翀*，王贝铭，王新铭，徐秋月，张子铉（以字母顺序）

摘要： 本算法采用的主要是Minmax算法，即考虑对方和本方同样优越性的前提下，依次考虑对于本方最优和对于本方最差的情形依次交错到一定深度，根据该深度下的最优解得出操作。具体涉及到的结构主要包括课上所学到的树，列表，字典等相关结构。
关键字： $Minmax$方法，$Alppha-Beta$剪枝，树，字典，列表，矩阵

1 算法思想.....1
1.1 总体思路.....1
1.2 算法流程图.....2
1.3 算法运行时间复杂度分析.....3
2 程序代码说明.....3
2.1 数据结构说明.....3
2.2 函数说明.....3
2.3 程序限制.....13
3 实验结果.....13
3.1 测试数据.....13
3.2 结果分析.....14
3.3 经典战局（可选）.....15
4 实习过程总结.....15
4.1 分工与合作.....15
4.2 经验与教训.....16
4.3 建议与设想.....17
5 致谢.....17
6 参考文献.....17

1 算法思想

1.1 总体思路

最初思想是根据原版本的 2048 的合成策略来进行考虑，在没有对抗的情况下，一种比较好的策略就是使得合成的最大的那个数值尽量始终保持在一个角落当中，这样可以最大可能地避免由于新加入的小数值数据掺杂使得大数合成过程受到阻碍。但是在观看了最初的天梯赛中的数场比赛之后，发现这种方法在对抗赛中可能并不是最优的，因为有了对方的加入，这种策略可能会因为地盘被争夺的原因而导致无法继续合成，但是这个策略仍然是需要的一个参考因素，只是需要更多的加入其他判断。

关于移动方向函数的判断，我们最初的预想是大概包含以下几个方面的因素：（1）向某

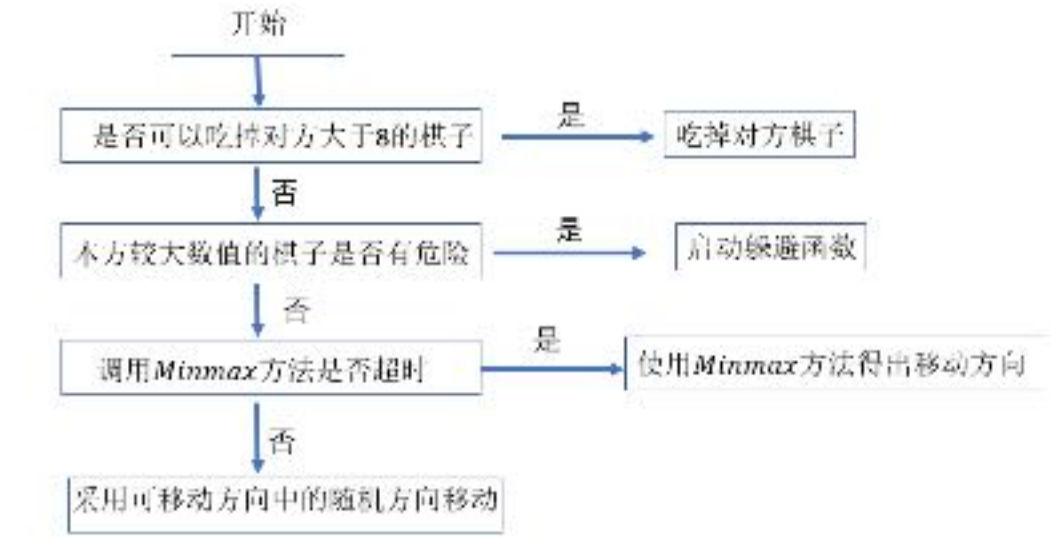
个方向合成之后的相同数相邻的数目；（2）向某个方向合成之后能够合成的最大数值；（3）移动之后最大数的位置；（4）移动之后对方局势以及对于下一次移动的利弊判断等。这些是在判断移动方向的过程中最主要因素。

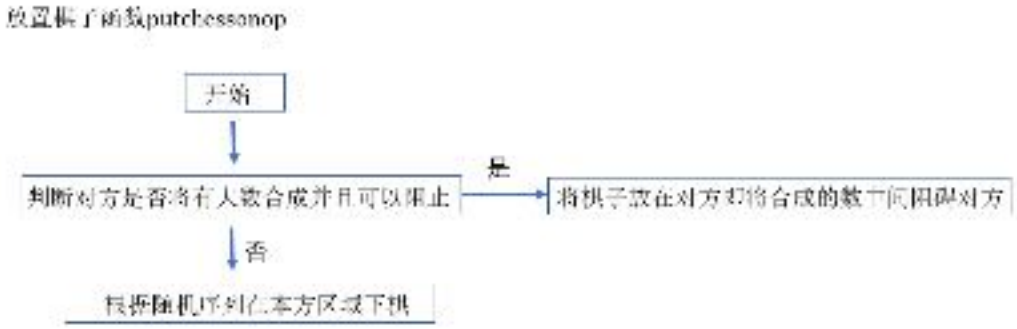
整体来说我们的代码经历了以下的发展过程：最初经过网络相关算法的调研参考找出大概思路，写出了最初版本的整体框架，但是整体来说实现功能比较少，比给出的白痴 AI 版本相比胜率不是很高，之后经过老师的相关讲解以及网络文献代码等参考，对代码进行优化，得到后来的使用Minmax方法的版本，并在实践过程中对其进行相关优化。

总的来说，最初的时候主要是借鉴了原版本 2048 的策略，但是加入对方的干扰之后，必然仅仅考虑本方就是不可取的方法，最终采用了深度为 4 的Minmax方法。

1.2 算法流程图

最终代码中所采取的流程为：





1.3 算法运行时间复杂度分析

时间复杂度分析

程序的时间复杂度我们考虑每一轮次下棋的过程所消耗的时间，问题规模和棋盘的大小相关，这里我们选择棋盘边长 N=4 作为问题规模，时间复杂度则的分析则是分析程序所耗费的时间和 N 的关系。程序中计算过程可以分为两部分：选择位置以及选择方向，在类 Player 中的 output 函数中实现。

位置选择中主要调用函数$putchessonop$来决定是否下棋到对方区域，该函数中主要是两次对棋盘的遍历，即两个双重 for 循环，因此时间复杂度为$2N^2$，即为$O(N^2)$。方向选择的过程的时间消耗主要包括两部分，函数eat_other用来决定是否吃掉对方棋子，以及若不吃掉对方，需要选择移动方向。对于eat_other函数，其中包括使用$getmax$函数对自己和对方棋盘的扫描，找到其中大于 16 的棋子，这包括两个对 N 的双重 for 循环，时间复杂度为$2N^2$，随后eat_other函数又对找到的较大数进行遍历，大约为N^2的时间复杂度。如不吃掉对方棋子，则包含对所有可能方向遍历的循环，这里为二维棋盘，最多可能方向为 4，每次循环中主要的时间消耗为 Decision 函数。Decision 函数调用了 minimax 算法的两个函数，在程序中我们设置最大深度为 4，并且设置了最大时间为 0.04，当时间大于 0.04 或超出深度时，调用经验函数，经验函数包含三重循环，时间复杂度为N^3，因此这种情况时间复杂度为N^3加一个常数；当时间小于 0.04 时调用 minimax 算法，其时间复杂度是指数增加的，我们设置的最大深度为 4，而每个 minimax 函数中包括对可能方向的遍历，因此时间复杂度为N^4 。所以当 N 较小时，会选用 minimax 算法，时间复杂度最多为N^4，但是其上限为 0.04。

将以上过程归纳起来，时间消耗为$2N^2 + 3N^2 + N^4$（时间小于 0.04 且深度小于 4）或$2N^2 + 3N^2 + N^3 + const$，即当 N 较小时时间复杂度为$O(N^4)$，否则时间复杂度为$O$（$N^3$)。

2 程序代码说明

2.1 数据结构说明

(a)算法中主要用到的数据结构包括树，列表，字典，矩阵。其中关于列表，字典和矩阵的使用更多的都是课程中所用到的基本操作，对于树结构的使用参考了老师课程中教授的决策树相关知识，使用Minmax方法并使用Alpha − Beta剪枝的方式对其进行优化。
(b)算法中涉及到的类包括Grid类，用于棋盘信息的判断；还有要求中需要的Player类，用于具体的对战过程下棋及合并使用。
(c)其中最主要的是运用了Minmax方法和Alpha − Beta剪枝，在实际对战过程中，应当考虑对方运转性能比较良好，以最有利于己方和最不利于己方这样的顺序交错下去进行判断，以期达到更好的结果，但是这种方法毕竟还是局部最优，无法做到全局最优解，对于最终结果还是有一定的局限性。

2.2 函数说明

Part 1：Minmax方法的实现

用于时间控制，如果时间在符合要求的范围内，则调用Minmax方法，得到具体的返回结果，如果时间超时，那么则不使用该种方法，以期在实际的对战中避免由于时间问题导致失败

Decision(grid, max = True/False)：对于某一移动方向移动之后所得结果的评估，评估的标准就是运用Minmax方法之后得到结果，本代码中考虑的深度为 4

Maximize(grid, alpha, beta, depth, start):当max = True的时候调用，得到一个效益最高的分支作为本方下一步的结果

Minimize(grid, alpha, beta, depth, start):当max = Flase的时候调用，得到一个效益最低的分支作为对方的策略

```
def Decision(grid, max=True):
    limit = 4
    start = time.clock()

    if max:
        return Maximize(grid=grid, alpha=-np.inf, beta=np.inf, depth=limit, start=start)
    else:
        return Minimize(grid=grid, alpha=-np.inf, beta=np.inf, depth=limit, start=start)


def Maximize(grid, alpha, beta, depth, start):
    if terminal(grid) or depth == 0 or (time.clock() - start) > 0.01:
        return Eval(grid)

    maxUtility = -np.inf

    for child in children(grid):
        maxUtility = max(maxUtility, Minimize(grid=child, alpha=alpha, beta=beta, depth=depth - 1, start=start))

        if maxUtility >= beta:
            break

        alpha = max(maxUtility, alpha)

    return maxUtility
```

```
def Minimize(grid, alpha, beta, depth, start):
    if terminal(grid) or depth == 0 or (time.clock() - start) > 0.01:
        return Eval(grid)

    minUtility = np.inf

    empty = grid.getAvailableCells();

    children = []

    for pos in empty:
        current_grid = grid.clone()

        current_grid.insertTile(pos, 2)

        children.append(current_grid)

    for child in children:
        minUtility = min(minUtility, Maximize(grid=child, alpha=alpha, beta=beta, depth=depth - 1, start=start))

        if minUtility <= alpha:
            break

        beta = min(minUtility, beta)

    return minUtility
```

Part 2：一些实现的辅助函数定义

```
66  def getChild(grid, dir):  #返回下一步（下一个相应移动方向后棋局）
67      temp = grid.clone()
68      temp.move(dir)
69      return temp
70
71  def children(grid):            #返回所有可移动后的局势
72      children = []
73      for move in grid.getAvailableMoves():
74          children.append(getChild(grid, move))
75      return children
76
77  def terminal(grid):       #叶节点，四个方向都无法移动，结束
78      return not grid.canMove()
```

```
def Eval(grid):
    import math
    import numpy as np

    if terminal(grid):
        return -np.inf

    gradients = [
        [[3, 2, 1, 0], [2, 1, 0, -1], [1, 0, -1, -2], [0, -1, -2, -3]],
        [[0, 1, 2, 3], [-1, 0, 1, 2], [-2, -1, 0, 1], [-3, -2, -1, 0]],
        [[0, -1, -2, -3], [1, 0, -1, -2], [2, 1, 0, -1], [3, 2, 1, 0]],
        [[-3, -2, -1, 0], [-2, -1, 0, 1], [-1, 0, 1, 2], [0, 1, 2, 3]]
    ]

    values = [0, 0, 0, 0]

    for i in range(4):
        for x in range(4):
            for y in range(4):
                values[i] += gradients[i][x][y] * grid.map[x][y]

    return max(values)


directionVectors = (UP_VEC, DOWN_VEC, LEFT_VEC, RIGHT_VEC) = ((-1, 0), (1, 0), (0, -1), (0, 1))
vecIndex = [UP, DOWN, LEFT, RIGHT] = range(4)
```

getChild(grid, dir):得到操作之后下一步的棋局
children(grid):得到所有可以移动的方向移动后的局势
terminal(grid):结束功能，当四个方向都没有办法移动的时候，截止操作
Eval(grid):评估效应函数，以此来判断移动后代价最小的方向

Part 3：Grid类及相关接口

```python
class Grid:    #用于棋盘判断
    def __init__(self, size=4, map=[[0] * 4 for i in range(4)]):
        self.size = size
        self.map = map

    def clone(self):   #复制棋盘信息
        gridCopy = Grid()
        gridCopy.map = deepcopy(self.map)
        gridCopy.size = self.size

        return gridCopy

    def set_map(self,map1): #修改棋盘
        self.map = map1

    def insertTile(self, pos, value):
        self.setCellValue(pos, value)

    def setCellValue(self, pos, value):   #给棋盘上的位置赋值
        self.map[pos[0]][pos[1]] = value

    def getAvailableCells(self):    #返回空位
        cells = []

        for x in range(self.size):
            for y in range(self.size):
                if self.map[x][y] == 0:
                    cells.append((x, y))

        return cells
```

```python
    def getMaxTile(self):      #找到最大值
        maxTile = 0

        for x in range(self.size):
            for y in range(self.size):
                maxTile = max(maxTile, self.map[x][y])

        return maxTile
```

Grid类，代码中的一个重要类，用于棋盘信息的判断
该段代码中对棋盘进行了初始化，以及一些在合并上的基础函数

```python
    def move(self, dir):  #棋盘移动
        dir = int(dir)

        if dir == UP:
            return self.moveUD(False)
        if dir == DOWN:
            return self.moveUD(True)
        if dir == LEFT:
            return self.moveLR(False)
        if dir == RIGHT:
            return self.moveLR(True)
```

```
    def moveUD(self, down): #往上或往下移动
        r = range(self.size - 1, -1, -1) if down else range(self.size)

        moved = False

        for j in range(self.size):
            cells = []

            for i in r:
                cell = self.map[i][j]

                if cell != 0:
                    cells.append(cell)

            self.merge(cells)

            for i in r:
                value = cells.pop(0) if cells else 0

                if self.map[i][j] != value:
                    moved = True

                self.map[i][j] = value

        return moved
```

```
    def moveLR(self, right): #往左或往右移动，判断能否移动并移动
        r = range(self.size - 1, -1, -1) if right else range(self.size)

        moved = False

        for i in range(self.size):
            cells = []

            for j in r:
                cell = self.map[i][j]

                if cell != 0:
                    cells.append(cell)

            self.merge(cells)

            for j in r:
                value = cells.pop(0) if cells else 0

                if self.map[i][j] != value:
                    moved = True

                self.map[i][j] = value

        return moved
```

这段代码中通过move, moveUD, moveLR三个函数来判断当前形势下能否移动并进行左右或者上下方向的移动。

```
    def merge(self, cells):    #若果遇到了两个能合并的数就merge
        if len(cells) <= 1:
            return cells

        i = 0

        while i < len(cells) - 1:
            if cells[i] == cells[i + 1]:
                cells[i] *= 2

                del cells[i + 1]

            i += 1
```

```
    def canMove(self, dirs=vecIndex): #判断能否移动

        checkingMoves = set(dirs)

        for x in range(self.size):
            for y in range(self.size):

                if self.map[x][y]:

                    for i in checkingMoves:
                        move = directionVectors[i]

                        adjCellValue = self.getCellValue((x + move[0], y + move[1]))

                        if adjCellValue == self.map[x][y] or adjCellValue == 0:
                            return True


                elif self.map[x][y] == 0:
                    return True

        return False
```

该段代码中

merge(self, cells)：该函数是对于相同数的合并判断

canMove(self, dir = vecIndex):该函数是对于在当前局势下，己方能否进行方向移动进行判断，如果不能进行方向移动，则会有在对方地盘下棋的操作调用

Part 4：Player类及其接口

对于Player类的定义，这也是代码中最重要的一个类

其中包括以下几个比较重要的函数：

```
class Player:
    def __init__(self, isFirst, array):
        self.xianhou = isFirst  # 先后手储存
        self.xiaqi = array  # 自方下棋位置储存
        self.decision = None  # 上次移动方向的决定

    def getChild(self, board, dir): #得到某一方向移动后的棋盘
        temp = board.copy()
        temp.move(self.xianhou, dir)
```

```python
    def get_boardMatrix(self, board):  #得到棋盘矩阵信息
        board_info = board.getRaw()
        matrix = [[0] * 4 for i in range(4)]
        if self.xianhou:
            for i in range(4):
                for j in range(4):
                    subtuple = board_info[i][j]
                    if  subtuple[0]>0:
                        matrix[i][j]=2**subtuple[0]
        else:
            for i in range(4):
                for j in range(4):
                    subtuple = board_info[i][j+4]

                    if  subtuple[0]>0:
                        matrix[i][j]=2**subtuple[0]
        return matrix
```

```python
    def getelseboardMatrix(self, board): #得到对方棋盘的矩阵
        board_info = board.getRaw()
        matrix = [[0] * 4 for i in range(4)]
        if not self.xianhou:
            for i in range(4):
                for j in range(4):
                    subtuple = board_info[i][j]
                    if  subtuple[0]>0:
                        matrix[i][j]=2**subtuple[0]
        else:
            for i in range(4):
                for j in range(4):
                    subtuple = board_info[i][j+4]

                    if  subtuple[0]>0:
                        matrix[i][j]=2**subtuple[0]
        return matrix
```

duobi(self, alist):躲避函数，即判断局势，当对方下一步可以将我方临近对方的较大的数值吃掉的话，就调用躲避函数，以此来尽量使得我方合成的大数不被吃掉（后手代码与先手原理一样，故：此处并没有放出后手情况代码的截图）

get_boardMatrix(self, board)：得到己方的棋盘矩阵信息，

getelseboardMatrix(self, board)：得到对方的棋盘矩阵信息

```
#如果整个棋局刚开始最大值小于8不再考虑，返回NONE
if xianshoumax[0]<3 or houshoumax[0]<3:
    return None
else:
    #第二次扫描得到先后手所有的最大值和次最大值数据
    for i in range(len(board_info)):
        sublist = board_info[i].split(" ")
        for j in range(len(sublist)):
            #获得当前棋子值
            num = int(sublist[j])
            #先手
            if num>0:
                if num>=2:
                    xianshouList.append([num,i,j])
            else:
                num = -num
                if num>=2:
                    houshouList.append([num,i,j])
    sorted(xianshouList,key=(lambda x:x[0]))
    sorted(houshouList,key=(lambda x: x[0]))
    return [xianshouList,houshouList]
```

getmax(self, board):找到自己和对方棋盘中的最大值和次大值，并找出相应的位置

```
418     def eat_other(self,board,isFirst):    #判断对方棋局中最大(次大)的数是否可以吃掉
419         #得到先手和后手的最大值次大值数据
420         allMax = self.getmax(board.copy())
421         #刚开始的话是无返回值的
422         if not allMax:
423             return None
424         #先手
425         if isFirst:
426             me_maxList = allMax[0]
427             rival_maxList = allMax[1]
428         #后手
429         else:
430             me_maxList = allMax[1]
431             rival_maxList = allMax[0]
432         #对这些最大值次大值进行扫描判断，只要有一个符合就返回
433         for i in me_maxList:
434             for j in rival_maxList:
435                 #判断是否相等
436                 if i[0]==j[0]:
437                     #判断方向
438                     #在同一行
439                     if i[1]==j[1]:
440                         #首先如果相邻就直接吃掉:
441                         if i[2]==j[2]+1:
442                             if board.move(self.xianhou, 2): return 2
443                         elif i[2]==j[2]-1:
444                             if board.move(self.xianhou, 3): return 3
445                         #如果说他们之间有空位可合并也可以写，但是还没写
446                     #在同一列
447                     elif i[2]==j[2]:
448                         #判断相邻就吃掉
449                         if i[1]==j[1]+1:
450                             if board.move(self.xianhou, 0): return 0
451                         elif i[1] == j[1]-1:
452                             if board.move(self.xianhou, 1): return 1
453         return None
```

eat_other(self, board, isFirst)：进攻函数，即在比较己方和对方局势之后，如果发现本方下一步可以吃到对方棋盘中最大或次大数值的棋子的话，那么就进行向对方方向的合并，吃掉对方棋子

（后手情况原理一样，故未截图显示）

putchessonop(self, matrix)：放置棋子函数，即在对己方和对方局势进行判断之后，如果防线对方的较大数值的数目即将合成并且中间有可以加入棋子的地方的话，就将新出现的棋子

2 加入对对方局势不利的位置上，干扰对方大数的合成过程；如果放置在己方，返回值为 0，若放置在对方，就返回到放置的位置

```
536     def output(self, currentRound, board, mode):  #决策输出
537         if mode == "_position" or mode == "_direction":
538             pass
539 
540         elif mode == "position":
541             self.decision = board.getDecision(self.xianhou)
542             if self.putchessonop(self.getelseboardMatrix(board.copy())) is not None:
543                 return self.putchessonop(self.getelseboardMatrix(board.copy()))
544             another = board.getNext(self.xianhou, currentRound)  # 己方的允许落子点
545             if another != ():
546                 return another
547 
548             available = board.getNone(not self.xianhou)  # 对方的允许落子点
549             if not available:  # 整个棋盘已满
550                 return None
551             else:
552                 from random import choice
553                 return choice(available)
```

```
555         elif mode == "direction":
556 
557             board1 = board.copy()
558             board2 = board.copy()
559 
560             direction = self.eat_other(board.copy(),self.xianhou)
561             if direction!=None:
562                 return direction
563             matrix = self.get_boardMatrix(board.copy())
564             grid_ob = Grid(size=4)
565             grid_ob.set_map(matrix)
566             moves = grid_ob.getAvailableMoves()
567             maxUtility = -np.inf
568             nexDir = -1
569             alist=[]
570             for move in moves:
571                 if board.copy().move(self.xianhou, move):
572                     child = getChild(grid_ob, move)
573                     utility = Decision(grid=child, max=False)
574                     if utility >= maxUtility:
575                         maxUtility = utility
576                         nexDir = move
577                         alist.insert(0,move)
578                     elif utility<maxUtility:
579                         alist.append(move)
580             if nexDir >= 0:
581                 return nexDir
582 
583             else:
584 
585                 directionList = [0, 1, 2, 3]
586                 from random import shuffle
587                 shuffle(directionList)
588                 for direction in directionList:
589                     if board.copy().move(self.xianhou, direction):
590                         return direction
591                 return
```

output(self, currentRound, board, mode)：输出函数，包含在本方处于各种模式（比如放置棋子模式和合并移动方向模式等）的时候相应操作

如果处于posiion模式的话，那么就对棋子放置位置进行调用，若下在对方位置，调用putchessonop(self, matrix)函数进行干扰，如若下在本方，则根据已知随机序列进行下棋

如果处于direction模式的话，则是对于移动方向进行判断，移动方向的顺序是：能够吃对方作为优先级最高的方向，能够吃掉就吃掉，如果不能吃掉的话，则调用Minmax方法对各个方向进行评估，然后选择效益最高的一个方向进行合并。如果这种方法超时的话，那么就随

机选择一个方向进行合并。

2.3 程序限制

在整个比赛过程中，并没有出现过因为程序崩溃而导致战败的例子。但本程序只是简单地将启发式规则的优先级提高，使其优先于 minmax 决策树；并且 minmax 决策树中用作为经验函数的梯度函数也仅仅只是能保证最大数可以被堆积在角上与堆积时同行（或同列）的单调性。这在单人游戏的 2048 中当然是足够的；然而，在对战版的 2048 中，由于敌方吞并我方棋子抢占地盘或是给我方地盘下棋的干扰，这一梯度函数就显得远远不足。至少应该将如局面上我方的空格数、棋子堆积的平滑性等因素纳入估值函数中；并且也不应将启发规则的优先级提升到这么高。没能建立一个考虑因素较为全面的数学模型无疑是很遗憾的。

3 实验结果

3.1 测试数据

实验环境说明：

- 硬件配置：Intel® Core™ i5-8250U CPU @ 1.60GHz 内存：8.00 GB
- 操作系统：Windows 10 专业版
- Python 版本：3.7

我们没有编写专门的测试程序进行代码测试。在每完成一版新的代码之后，我们都会先在本地与前一版的代码进行大量对战（第一版代码则和老师提供的测试代码进行对战），观察代码是否会发生报错以及新版代码是否比上一版代码表现得更好。

在排除报错问题确保代码可以使比赛顺利进行且新版代码性能比旧版更好的情况下，将代码上传至天梯上以观察我们代码的竞争力。

本着“再菜也要拿到 3 分”的原则，我们第一版代码仅仅是 minmax 树的实现与调用。由于该代码采取的是最为保守的策略（即：从不抢占敌方地盘），不出所料，在与其他小组的对决时遭到了惨败，胜率约为 20%。

确保了能拿到基础分的前提下，我们继续对代码进行了改进。显而易见，保守战略最大的问题就在于敌方会很快侵占我方的地盘，使我方很快陷入卡死的僵局中，之后，比赛的后半程就成为敌方的“个人游戏”时间。显然，这对我方是致命的打击。于是，在第二版代码中我们加入了侵吞对方地盘这一启发式规则，即：当对方边界上有大于等于 8 的数时优先进行合并以抢占对方的地盘。我们认为这一操作的优先级应高于 minmax 决策树。虽然这版代码还是很菜，但是在实战中有时可以通过吞并对方的一个大子而逆转局势，最终赢得比赛。

更进一步的，我们在代码中加入了防御机制，即：当发现在对方下一次选择移动方向时有可能抢占我方的地盘，我方在本回合选择移动方向时就选择可以避免“被吃”的方向（如果有多个方向可以供选择，则选择 minmax 估值函数中最大的方向）。这一版算法也是我们最终版的代码。

各版本代码的胜率如下表所示。

代码版本	胜率
1.0	21.1%
2.0	30.8%
3.0	36.2%

其中，代码 3.0 的胜率数据来自于最终天梯小组赛的数据。由于天梯小组赛的比赛安排原因，各个小组只会和与其排名相近的小组进行对战，由此得到的胜率其实是虚高的。可以看出，我们代码的对战表现并不理想，具体原因将在下一节中给出。

3.2 结果分析

从上一节可以看出，我们小组代码的对战表现很不理想，甚至最终版的代码在对战 official bot D 时仍然维持着类似于 2:8、3:7 这样的惨淡比分。这样的结果着实让我们感到奇怪。因为毕竟我们也应用了带有减枝的 minmax 决策树和启发式规则，如此还是败给 bot D 我们是没有料到的。

然而，在天梯赛代码提交 ddl 的那个早上，小组成员之一提出了一个大胆而可怕的想法：是不是自始至终 minmax 决策树都没有被正常调用过？经过观察，当其它的小组都在担心自己的算法在比赛中超时，我们的算法在比赛中的总耗时永远都只有零点零几秒——这显然不是因为我们的算法很优秀。并且，多次改变估值函数后，改变后的版本和原版本对战时的表现旗鼓相当，基本都是五五开。由此，我们不得不承认并且接受这个可怕的事实：我们最终版代码中的 minmax 决策树没有被正常调用。于是，整个代码实际上仅仅通过几个启发式规则支撑——最终的惨败由此也是必然的。由于发现问题的时间距离 ddl 已经只有不到两小时了，因此，直至 ddl 前的最后一刻我们都在努力寻找代码中哪一部分出错可能导致这样的结果；然而，事与愿违，我们最终只能带着遗憾、眼睁睁看着这个只是看似“用到了”minmax 决策树的代码在竞技场中被残酷“虐杀”。

由此，在本代码中，仅有 eat_other 函数这一“看到对方有较大数可以吞并就吞并”的启发式规则得到充分运用（duobi 函数由于需要在可以躲避的方向内选择 minmax 决策树中估值最大的那个方向，故其实际也不能按照计划执行）。而 eat_other 函数确实起到了抢占对方地盘的作用，在一些局面下可以通过合并对方的大数扭转局面、反败为胜。它在运行时间上也符合预期。

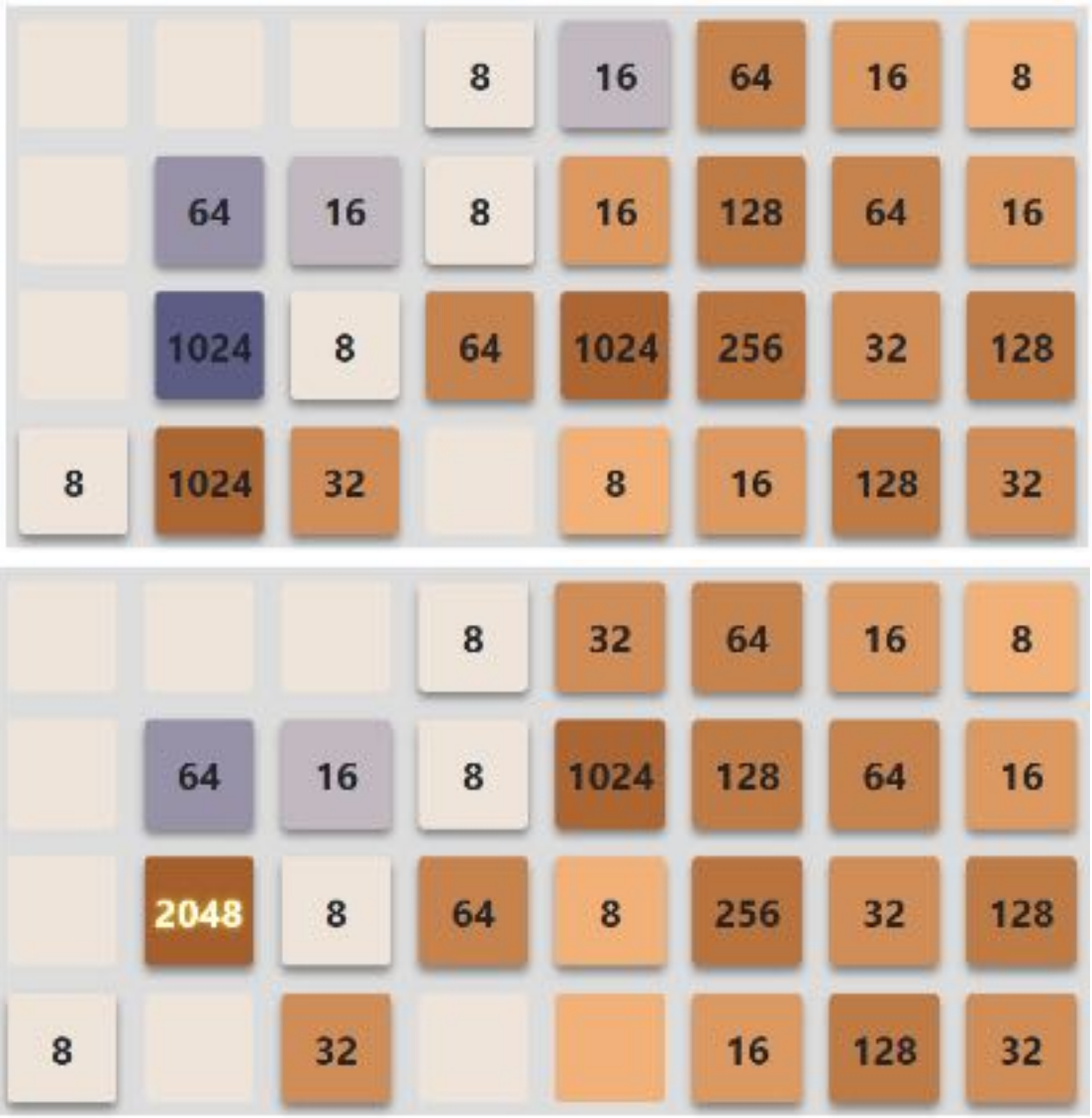
虽然比赛结束了，但是大家找问题、解决问题的决心并没有消散。经过大家的努力，终于找出了 minmax 树没有被调用的原因。

```
    elif mode == "direction":

        board1 = board.copy()
        board2 = board.copy()

        direction = self.eat_other(board.copy(),self.xianhou)
```

原来罪魁祸首是少了中间两行代码，在用 board 之前忘了 copy！一失足成千古恨，如果可以早一点发现问题，相信结果会更好。

3.3 经典战局（可选）

以下两张图便展示了 eat_other 函数是如何通过合并对方大数而提前锁定胜局的。（图中我们小组是反手方）



4 实习过程总结

4.1 分工与合作

秋昱翀是小组组长，负责组织组员进行讨论、对小组成员进行分工、报告书写；徐秋月同学负责基本策略的提出与报告书写；王贝铭、王新铭、张子铉同学负责代码各部分的编写。所有同学都参与到了代码测试、代码改进方案建议等讨论与工作中。

由于疫情的原因，我们所有的交流均是在线上进行，以线上会议+微信群聊的方式进行。我们一共进行了三次线上组会，均在腾讯会议上召开，小组成员均积极参与其中。此外，我们也时刻在微信群里交流自己的想法。相较于线上会议，微信群交流更为灵活方便，也不失为一种好的合作交流方式。

（第一次线上会议大家都忘了截屏；大家也都比较羞涩，没有打开摄像头）

4.2 经验与教训

整体来说，大家的合作还是比较愉快的，分工也比较合理明确，大家都在做着自己擅长的事情。所有小组成员都积极参与到这项大作业中，没有发生在线上交流中容易出现的“人间蒸发”的事情。虽然竞赛结果不是很理想，但是快乐就完事了！

对于这次大作业，我们当然也存在许多不足需要改进。首先就是代码的注释很重要的，尤其对于小组作业来说。注释一来可以大大提高代码的可读性，二来也可以使队友迅速了解你的代码思路帮助 debug、继续往下写和写报告，避免在代码解释上浪费过多的时间。并且，要善于发现一些隐藏的 bug，不能只把目光局限于报错上。事实上，如果我们这一次可以早一些注意到我们代码的运行时间过短，我们就能早一些发现 minmax 树在实际中没有被调用，也就有可能在算法竞赛 ddl 前找出 bug 并修改，结果也会更好。而我们这次仅仅因为代码在比赛过程中都能正常运行而认为与其它小组混战时惨败是由于其它小组的策略都太强了，丝毫没有考虑到也许是写的代码本身在运行时有问题。这一点对我们来说无疑是致命打击，在以后写代码的时候应该注意要时刻关注算法的运行（如：运行时间、效果等等）是否符合预期，这样能更容易找出问题。最后，在天梯赛 ddl 的前一天，负责写代码的主力军之一病倒了、最后只能带病工作，非常不容易。这也提醒了我们健康是做一切事情的根本保障，有一个好的身体是很重要的，多肝伤身。

4.3 建议与设想

本次大作业总体来说设计的很好，在原版 2048 的基础上进行的规则的改变使游戏的对战性变得更强，观赏性也变得更佳。技术组也非常给力，“菠菜王”的设置也让整个比赛的体验更加刺激、更加具有趣味性。

有一些关于大作业时间安排的小建议。我们认为可以将大作业的整体时间稍稍提前，这样在一定程度上可以避免“期末火葬场”。比如，可以考虑在五一前发布大作业的内容，这样大家就可以利用五一假期进行讨论编程，所拥有的空闲时间也会越多。大作业虽然会用到学期后半部分所讲的树的相关知识，但其实大部分时间，大家还是会通过自学来完成大作业（捂脸）。并且，在现场比赛环节，希望可以在决赛环节进行可视化的比赛，而不是大多数时间只能看着比分，相信这样的观赏性会更强。而为了节省时间，之前的 8 进 4 和半决赛可以不进行比赛的复盘。

对学弟学妹的寄语：虽然明年可能上课的形式和今年有很大的不同，大作业、创意作业等等都可能被削减，但还是要好好上陈老师的课呀！陈老师讲课思路很清晰，在课堂上可以收获很多；H 系列作业虽然看上去任务量比较大，但是咬咬牙一定能撑住，而且还能得到不小的提升。最后，小黑真的很可爱！

对于实习作业后续工作的设想：后续可以将优秀代码之间的对决的可视化过程做成一个视频，相信可以成为一个很好的宣传材料。

5 致谢

首先感谢陈老师一学期以来的指导，使这门课如此有趣，也使我们收获了很多。
感谢助教们的辛苦付出，在微信群里的答疑解惑，以及技术组同学为大作业基础设施付出的努力；没有你们就没有今天体验感极佳的大作业。
感谢奋斗到最后一刻的队友们，虽然竞赛结果不理想，但没有一个人放弃努力。
感谢腾讯会议软件，让我们的线上会议可以顺利进行。

6 参考文献

1. https://github.com/SrinidhiRaghavan/AI-2048-Puzzle

数据结构与算法课程实习作业报告

林旭辰* 白康博 龚德炜 郭洋帆 王依婷[1]

摘要 北京大学地球与空间学院 2020 年数据结构与算法课程实习项目为可进行对战版 2048 游戏的 AI。本组（N19_404）使用 minimax 搜索树配合 alpha–beta 剪枝算法进行 AI 实现，并通过实验确定了一个较好的估值函数。最终，本组代码以 N19 组第 6 名出线，成为八强，并在淘汰赛中取得第四名。本组项目可在此处在线查看。

关键字 极大极小值定理; alpha–beta 剪枝; 2048 游戏; 估值函数

目录

1 算法思想 2
1.1 总体思路 2
1.2 算法流程图 3
1.3 算法运行时间复杂度分析 3
2 程序代码说明 4
2.1 数据结构说明 4
2.2 函数说明 4
2.3 程序限制 9
3 实验结果 9
3.1 估值函数形式的确定 9
3.2 确定估值函数参数 12
3.3 经典战局 13
4 实习过程总结 14
4.1 分工与合作 14
4.2 建议与设想 16
5 致谢 16
6 引用 16

[1] 除组长外，其他组员按拼音音序排序。

1 算法思想

1.1 总体思路

1.1.1 整体搜索框架与数据结构

Minimax 搜索树基于冯·诺伊曼 (von Neumann, 1928)论证的极大极小值定理。该定理是博弈论的基本定理，声称对一个双参量函数$f(z,w)$，有

$$\max_{z\in Z}\min_{w\in W} f(z,w) = \min_{w\in W}\max_{z\in Z} f(z,w). \tag{1}$$

可利用此进行模拟决策。Alpha–beta 剪枝为对 minimax 搜索树进行的优化，其可大幅减少搜索树的节点数。 (Wikipedia contributors, 2020)

在整体结构上，本小组采用了 minimax 搜索树对可能的局面进行搜索，利用估值函数进行评估，筛选出可以产生最优局面的策略。并利用树结构实现以上过程。为了优化效率，采用了 alpha–beta 剪枝算法。

Minimax 搜索算法是一种找出失败的最大可能性中的最小值的算法，一般以递归方式实现。其假定博弈双方都尽可能获得最好结果。即一方要选择将其优势最大化的走法，另一方则选择令对手优势最小化的走法。以搜索两步为例，在第一步中，我方都会选择局面估值最大的策略。而在第二步中，对方会选择使局面估值最小的策略。为了使最后我方收益最大，应该在第一步选择使下一步估值最小值最大的策略。当搜索层数加深时同理。

为了实现 minimax 算法，我们构建了 minimax 搜索树的数据结构。其每个节点代表一个局面，包含对这个局面的价值信息。并将一步操作可以到达的局面关联为其子节点。在构建完指定深度的搜索树后，从叶节点开始估值，并向上按照最大-最小的规则递推得到各节点价值，即可得到第一层各节点的价值，筛选出合适策略。

然而预测的步数越多，生成的 minimax 树越庞大，搜索的消耗会很大。为了提高搜索的效率，引入了通过对评估值的上下限进行估计，从而减少需要进行评估的节点的范围的优化，即 alpha–beta 剪枝。它能够快速裁剪掉不必要的搜索分支，从而提高搜索的效率。

通过 minimax 算法和 alpha–beta 剪枝的结合，我们便可以对未来局面进行搜索，选出最优策略。但受制于时间，必须限制一定的搜索深度。我们根据当前决策模式（合并/落子）和剩余时间动态决定搜索深度，以最大化利用时间。详见算法流程图。

1.1.2 局部启发式算法

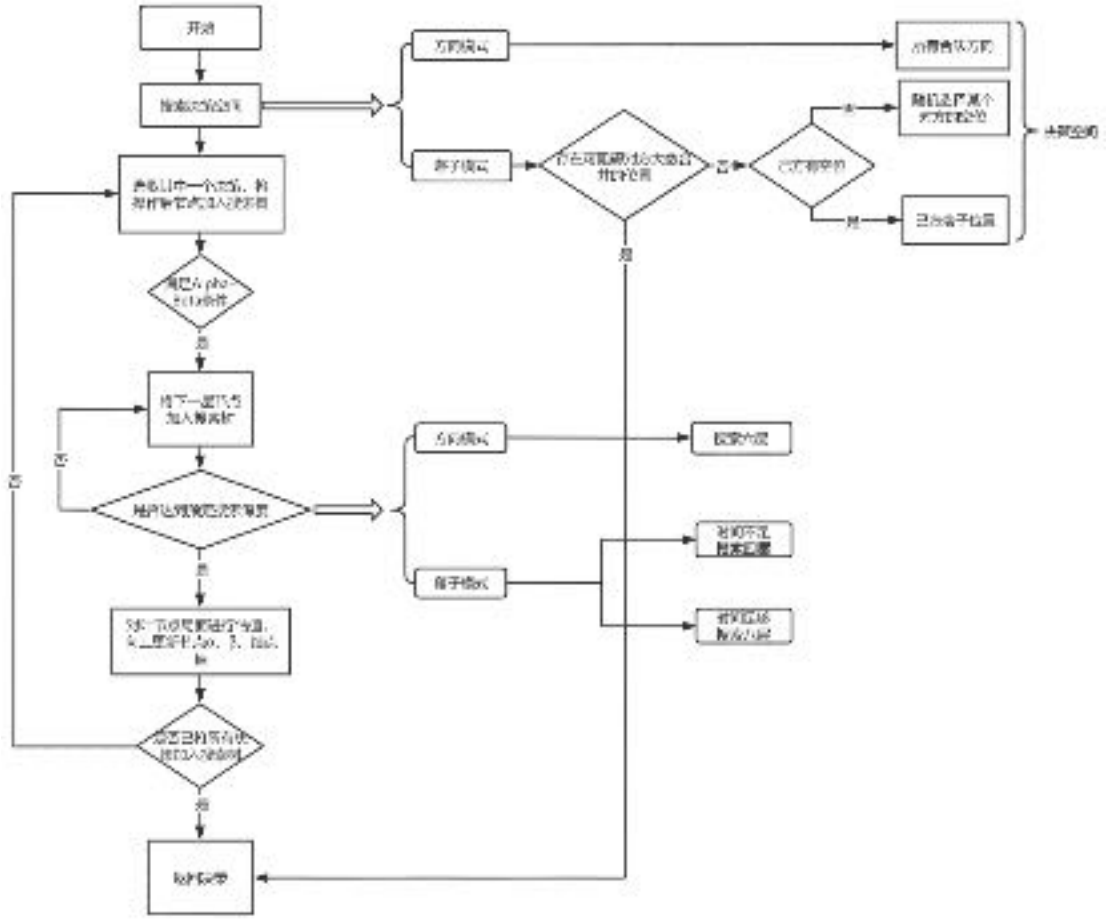
除了上述通用的 minimax 搜索和 alpha–beta 剪枝结构，根据 2048 问题的特殊性，还必须有特殊的启发式算法以对局部进行处理并体现我们的策略。在这方面，我们组主要设计了估值函数和寻找对方落子位置两个模块。

对于估值函数，首先要求其能够根据规则反映当前局面下的胜负关系。为此，我们设定了参数 `BASE_POWER`。对于一个值为 `2**n` 的格子，将 `BASE_POWER**n` 作为其价值，并通过求和得到我方总分。其次，我们鼓励向对方领地进攻，以获得更大的空间。对于每个进入对方领地的棋子，我们将它的数值乘以 `BONUS_POWER` 再加入我方总分中。最后，我们曾试图让数值相近、大小关系连续的棋子尽量聚集，根据数值间的平滑性、单调性对分数进行修正。但经实验发现，此举需要大量调用棋盘信息，耗时过大，故最终废弃。在获得我方分数后，利用类似的算法计算对方分数，二者相减即为局面总分。估值函数的具体介绍详见报告第三部分。

在搜索中，因为在对方领地落子的可选决策数量过多不可能全部加入搜索树。因

此，我们需要函数寻找在对方哪个位置落子最优，仅将此位置加入搜索树。我们经过实验发现，在对方领地落子的主要意义是阻止合并。因此，在最初我们对于对方领地的每个空位，考虑它十字形邻域内的四个数字，根据这四个数字有无相等进行赋分。但如此遍历过于耗时。最终，我们根据进行合并前后的估值变化，判断有无合并发生。进而找到可能阻止合并的位置，直接将其作为最优解加入搜索树。如果不存在以上情况，则默认在己方领地落子。这样既实现了策略要求，又压缩了搜索树以节约时间。

1.2 算法流程图



1.3 算法运行时间复杂度分析

为估计本算法的时间复杂度，需明确何为基本操作与问题规模。

估价函数的输入为某个节点的局面，每次运算需要访问双方的分数，盘面上 16 个位置的大小与归属，每次调用的时间消耗相对稳定，因此可以选为基本操作。而问题的规模可以用决策树的层数n来代表。

当搜索$2n$层（本算法中选为偶数）时，决策树中将会有$4^n \times 1^n$个节点（4 个方向+1 个位置选择）。如果不考虑剪枝，需要调用估价函数的次数将为$4^n \times 1^n$，考虑剪枝后，节点个数将会有一定减少。无论如何，本算法的大 O 数量级为$O(2^n)$，即指数量级。

由于本算法中n仅为个位数，因此难以看出指数趋势，但是在实验过程中，超时轮数随搜索层数有显著变化。

经过精细的调整，针对不同的情况本算法将搜索不同层数。最终天梯运行 500 轮

的耗时将稳定在 4 至 4.5 秒。同时，为避免超时，还设计了时间管理模式。在剩余思考时间少于 0.5 s 时，减少搜索层数。

2 程序代码说明

2.1 数据结构说明

本组使用的算法为 minimax 搜索树配合 alpha–beta 剪枝，因此需要实现一个搜索树。该搜索树定义为类 `Node`，为搜索树的节点。其实例拥有的属性见下方 doc string。

```
class Node:
    '''搜索树的节点.
    使用 self.node: dict 作为数据结构, 每个节点存储
        self.isFirst: 此树是否在先手的视角下生成
        self.mode: 此节点决策模式, 可为 'POSITION' 或 'DIRECTION'
        self.board: 待决策棋盘
        self.currentRound: 本节点轮数
        self.evaluate: 估值函数
        self.minimax: bool, 是否为 min 节点
        self.alpha, self.beta: ab剪枝形参数
        self.nodes: 节点
    '''
```

树结构的主要承载为属性 `self.nodes`，为一个 `dict` 字典。其键为可能的决策，值为该决策对应的子树的根节点。此外，每个节点还存储了待决策的棋盘 `self.board`，以及当前轮数 `self.currentRound`。`self.minimax`、`self.alpha`、`self.beta` 为在 alpha–beta 剪枝中要使用的参量，将在 2.2 中详述。

该树的结构较简单，最为复杂的是搜索加深函数 `deepen()`。

```
    def deepen(self, nodes: list):
        '''加深并搜索树. nodes 为主新的层的 (node, round) 元组构成的 list
        返回本层新的 alpha 或 beta
        '''
```

具体函数实现将在 2.2 中详述。

```
# 对对方会并方向的计数
direction_count = {True: {RIGHT: 3, UP: 1, DOWN: 1, LEFT: 0},
                   False: {LEFT: 3, UP: 1, DOWN: 1, RIGHT: 0}}
```

为一个字典，记录对手先后手的方向决策，以为搜索树的搜索顺序提供参考。

对于任务要求的类 `Player`，实现了要求的 `__init__()`, `output()`。另外实现了估值函数 `evaluate()`，将在 2.2 中详述。每一回合都会重新建设搜索树。

2.2 函数说明

2.2.1 Node 的接口

初始化函数为存储所有必要的属性。参数以及 doc string 见下

```
    def __init__(self, isFirst: bool, mode: str, board, currentRound: int,
                 evaluate, minimax: bool, alpha=-INF, beta: float = INF):
        '''生成节点.
        player: 玩家
        isFirst: 这棵 xxx树xxx 是 !!!以谁的视角!!! 生成的
        mode: 'position' or 'direction'
        board: 棋盘. 期待传入一个副本
        currentRound: 此层轮数

        evaluate: 估值函数, evaluate(board, isFirst) 返回 isFirst 方与 not isFirst 方的局面之差
        minimax: min 'True', max 'False'
        alpha, beta: alpha, beta值
        '''
```

除了将参数保存外，还会根据 minimax 参数决定 `self.ab_sttr` 以及 `self.comp`。

```
# 'ab_attr' 当本层的名称
# 'comp' 当于本层的比较函数
self.ab_attr = 'beta' if self.minimax else 'alpha'
self.comp = operator.lt if self.minimax else operator.gt
```

注意到，在 alpha–beta 剪枝算法中，min 节点仅修改估值的上限 beta，而 max 节点仅修改估值的下限 alpha。使用 `self.ab_attr` 存储属性名。同时，min 节点目标是使 beta 最小化，max 节点则是使 alpha 最大化。故使用 `self.comp` 存储相应的比较函数（小于与大于）。

函数 `count_direction()` 会更新对对方的操作频数的记录。

```
def count_direction(self, operation):
    '''记录对同对方向的操作情况
    '''
    if len(operation) == 1:  # 我也不知道为啥方向会被拿出元组来
        operation = operation[0]
        self.direction_count[not self.isFirst][operation] += 1
```

首先判断保证传入的是方向操作，然后记录对手的操作。由于己方的先后手由 `self.isFirst` 记录，故 `not self.isFirst` 即为对方先后手信息。

函数 `deepen()` 会根据参数 `modes` 的指示建设搜索树。该函数返回本层的新的 `alpha` 或 `beta`（即 `getattr(self, self.ab_sttr)`）。

```
def deepen(self, modes: list):
    '''加深并搜索树. modes 为由新的层的 (mode, round) 元组构成的 list
    返回本层新的 alpha 或 beta
    '''
```

函数的参数 `modes` 的示例为

```
modes = [(POSITION_MODE, currentRound), (DIRECTION_MODE, currentRound)]
```

表示希望建设两层，第一层为位置模式，`round` 为 `currentRound`；第二层为方向模式，`round` 也为 `currentRound`。

函数 `deepen()`根据 `mode` 是否为空分为两个分支。若 `modes` 非空，那么需要在本层建设树。

```
if modes:
    # 本层没被剪过，而且还要往更深层搜
    opers = self.operations()
    if opers:
        # 本层有路可走
```

首先获得本层可以做出的决策（`self.operations()`见下），如果有决策，那么尝试按此决策进行下棋。

```
for op_func in opers:
    # 这个 op_func 会更新棋盘，返回棋盘有没有变
    board_copy = self.board.copy()
    if getattr(board_copy, op_func[0])(op_func[1], op_func[2]) == False:
        # 棋盘在此操作下不动，非法
        continue
```

如果决策导致棋盘不动，那么是非法操作，直接跳过，否则创建新节点，并更新其的 alpha 或 beta 属性。

```
    # 创建新节点，给下一层
    self.nodes[op_func[2]] = Node(self.isFirst, modes[0][0], board_copy,
                                  modes[0][1], self.evaluate,
                                  not self.minimax, -INF, INF)
    # 更新子节点属性
    setattr(self.nodes[op_func[2]],
            self.ab_attr, getattr(self, self.ab_attr))
```

然后，对新节点进行 `deepen`，并获取其返回值，观察返回值是否比现有的上下限

更好。

```
        # 继续往深层搜
        new_value = self.nodes[op_func[2]].deepen(nodes[1:])
        if self.comp(new_value, getattr(self, self.ab_attr)):
            # 获得了更好的上下限
            setattr(self, self.ab_attr, new_value)
```

最后，在必要时剪枝。

```
        if self.alpha >= self.beta:  # 剪枝条件
            break
```

如果前面这些操作没有任何结果，那么将空操作作为新节点，列入考虑范围。

```
    if not self.nodes:
        # 如果前面没探出什么东西，那么把空操作弄进去
        self.nodes[()] = Node(self.isFirst, nodes[0][0], self.board.copy(),
                              nodes[0][1], self.evaluate, not self.minimax,
                              -INF, INF)
        setattr(self.nodes[()], self.ab_attr,
                getattr(self, self.ab_attr))
        new_value = self.nodes[()].deepen(nodes[1:])
        setattr(self, self.ab_attr, new_value)
```

而如果 `modes` 为空，则不需要建设树，其他则几乎一样。

```
    else:
        opers = self.operations()
        # 需保证层数为0的时候一定在搜索direction
        for op_func in opers:
            # 这个 op_func 会更新棋盘，返回棋盘有没有变
            board_copy = self.board.copy()
            if getattr(board_copy, op_func[0])(op_func[1], op_func[2]) == False:
                continue
            new_value = self.evaluate(board_copy, self.isFirst)
            if self.comp(new_value, getattr(self, self.ab_attr)):
                setattr(self, self.ab_attr, new_value)
            if self.alpha >= self.beta:
                break
    return getattr(self, self.ab_attr)
```

最后返回值。

`decision()` 函数会根据已经建设完毕的树进行决策，返回决策与新的根节点。

```
def decision(self):
    '''决策，需要先调用 deepen()
    返回 (决策, 新根节点)
    '''
    for op, node in self.nodes.items():
        if node.beta >= self.alpha:
            return op, node
    raise RuntimeError('no decision')
```

其中，判定一个决策可返回的条件是：该子节点的估值上界不小于根节点的估值下界。若无决策，会抛出异常。

函数 `operations()` 会返回本节点可行的决策。

```
def operations(self) -> list:
    '''返回可用操作的 [(化学, 物理传参), ...]
    '''
```

其假设双方都优先在己方落子（在对方区域落子的情况另行考虑，见 2.2.2），除非无法在己方落子。此时，将随机落子。

方向方面，其也给出搜索顺序。对于敌方，将根据其历史决策给出搜索顺序；对于己方，则优先向对方方向合并。

2.2.2 在对方落子的寻找: `find_pos()`

函数接口如下

```
def find_pos(board_raw: list, to_belong: bool, direction: int) -> tuple:
    '''寻找在 `to_belong` 方落子的位置，若无法阻止合并，返回 None
    board_raw: 棋盘原始数据，为嵌套列表
    direction: 为己和对方占优的合并方向
    仅处理对方至少是两个2^x倍的合并，其中 x 为 `MERGE_INTERFERENCE_THRESHOLD + 1`
    '''
```

调用该函数时，需要已知 to_belong 方在 direction 方向合并会出现大子合并，本函数会返回一个可能的限制对方合并的落子地点。

函数的流程即为在 direction 方向上搜索，观察是否有连续的等大且足够大的棋子。如有，返回一个可供插入的位置。否则返回 None。

2.2.3 Player 的接口

初始化函数十分简单，存储先后手信息以及随机序列，并新建了一个空的树。

```
class Player:
    def __init__(self, isFirst: bool, array: list) -> None:
        self._isFirst = isFirst
        self._array = array
        self.tree = Node(isFirst, POSITION_MODE, None, 0,
                         self.evaluate, False, depth=0)
```

output()函数首先会获得对手上步操作，并记入 Node 的类变量

```
    def output(self, currentRound: int, board, mode: str):
        # 获得上部操作，记录小本本
        prev_decision = board.getDecision(not self._isFirst)
        self.tree.count_direction(prev_decision)
```

如果为不需要返回决策的以 '_' 开头的模式，直接结束函数。否则建设搜索树。

```
        # 直接返回，不浪费算力
        if mode[0] == '_':
            return

        # 正经搜索树
        decision = None
        self.tree = Node(self._isFirst, mode, board.copy(), currentRound,
                         self.evaluate, False, depth=0)
```

如果是位置模式，首先查看对面有没有向某个方向的合并会导致大子合并。这里使用对棋盘副本进行 move 操作，并计算场上棋子的级别和来查看。如果有大子合并，那么所有棋子的级别和会减少一个较大的数值。挑选减少得最多的合并方向给 find_pos() 函数寻找好的落子位置。

```
        if mode == POSITION_MODE:
            # 尝试看看有没有高收益的阻挡对方合并的地方
            # 寻找对面的大子合并方向
            old_score = sum(board.getScore(self._isFirst)) + \
                sum(board.getScore(not self._isFirst))
            dir_merge = {}  # 方向: 合并后的sum(getScore)减少了多少
            direction = 0
            while direction < 4:
                board_copy = board.copy()
                if board_copy.move(not self._isFirst, direction) == False:
                    direction += 1
                    continue
                new_score = sum(board_copy.getScore(self._isFirst)) + \
                    sum(board_copy.getScore(not self._isFirst))
                delta = old_score - new_score
                if delta >= MERGE_INTERFERENCE_THRESHOLD:
                    dir_merge[direction] = delta
                direction += 1

            if dir_merge:
                max_dir = max(dir_merge, key=lambda k: dir_merge[k])
                decision = find_pos(board.getRaw(), not self._isFirst, max_dir)
```

如果找到好的落子位置，那么 decision 会变成非 None。否则，进行搜索树决

策。建设一个六层的搜索树，直到决策方第二次 direction 模式为止。

```
if decision == None:
    # 搜六层
    if self._isFirst:
        self.tree.deepen(
            [(POSITION_MODE, currentRound), (DIRECTION_MODE, currentRound),
             (DIRECTION_MODE, currentRound), (POSITION_MODE, currentRound + 1),
             (POSITION_MODE, currentRound + 1), (DIRECTION_MODE, currentRound + 1)])
    else:
        self.tree.deepen(
            [(DIRECTION_MODE, currentRound), (DIRECTION_MODE, currentRound),
             (POSITION_MODE, currentRound + 1), (POSITION_MODE, currentRound + 1),
             (DIRECTION_MODE, currentRound + 1), (DIRECTION_MODE, currentRound + 1)])
```

对于合并模式，则会根据剩余时间是否少于 0.5 秒，选择搜索 4 层或 8 层。最后则会进行搜索树或对方落子的决策。

```
elif board.getTime(self._isFirst) < .5:
    # @Mr.Luo Zhixiang
    # 搜四层
    if self._isFirst:
        self.tree.deepen(
            [(DIRECTION_MODE, currentRound), (POSITION_MODE, currentRound + 1),
             (POSITION_MODE, currentRound + 1), (DIRECTION_MODE, currentRound + 1)])
    else:
        self.tree.deepen(
            [(POSITION_MODE, currentRound + 1), (POSITION_MODE, currentRound + 1),
             (DIRECTION_MODE, currentRound + 1), (DIRECTION_MODE, currentRound + 1)])
else:
    # 搜八层
    if self._isFirst:
        self.tree.deepen(
            [(DIRECTION_MODE, currentRound), (POSITION_MODE, currentRound + 1),
             (POSITION_MODE, currentRound + 1), (DIRECTION_MODE, currentRound + 1),
             (DIRECTION_MODE, currentRound + 1), (POSITION_MODE, currentRound + 2),
             (POSITION_MODE, currentRound + 2), (DIRECTION_MODE, currentRound + 2)])
    else:
        self.tree.deepen(
            [(POSITION_MODE, currentRound + 1), (POSITION_MODE, currentRound + 1),
             (DIRECTION_MODE, currentRound + 1), (DIRECTION_MODE, currentRound + 1),
             (POSITION_MODE, currentRound + 2), (POSITION_MODE, currentRound + 2),
             (DIRECTION_MODE, currentRound + 2), (DIRECTION_MODE, currentRound + 2)])

# 决策
if decision == None:
    decision, self.tree = self.tree.decision()
return decision
```

2.2.4 估值函数

估值函数分为基本分和附加分两部分。

```
def evaluate(self, board, isFirst: bool):
    '''估值函数 TODO 返回 isFirst 方与 not isFirst 方的局面之差
    '''
```

基本分为将己方的所有棋子的级别做映射并求和，映射为

```
def func(x): return 2 ** (BASE_POWER * x)
```

只要 BASE_POWER > 1，就可以使合并后的棋子比合并前分数更高。同时，对对方的所有棋子也做相应映射并求和，乘上比例系数 ENEMY_WEIGHT 后减去。

```
res = sum(map(func, board.getScore(isFirst))) - \
    ENEMY_WEIGHT * sum(map(func, board.getScore(not isFirst)))
```

附加分是为了鼓励己方棋子向对方进攻所设立，对进入对方领域的棋子有指数形式的加分。

```
# 在对面的棋子的附加分
boardList = board.getRaw()
for col in range(4, 6) if isFirst else range(2, 4):
    for row in range(4):
        if boardList[row][col][1] == isFirst:
            res += 2 ** (BONUS_POWER * boardList[row][col][0])
```

```
        for col in range(5, 8) if isFirst else range(2):
            for row in range(4):
                if boardList[row][col][1] == isFirst:
                    res += 2 ** (ENEMY_BONUS_POWER + boardList[row][col][0])
        return res
```

其中的参量为

```
MERGE_INTERFERENCE_THRESHOLD = 4  # 仅触发 2^5 及以上的合并
BASE_POWER = 2  # 基础分值为 2**(BASE_POWER * 级别)
ENEMY_WEIGHT = 0.75  # 估值为己方分值减去 ENEMY_WEIGHT*对方分值
BONUS_POWER = BASE_POWER  # 己方棋子在对方领域领先己方的同列时，有额外加分 2**(BONUS_POWER * 级别)
FAR_BONUS_POWER = BASE_POWER - 1 # 己方棋子在对方领域远离己方的同列时，有额外加分2**(FAR_BONUS_POWER*级别)
```

2.3 程序限制

程序的最终版本在本地及代码天梯上均进行了多场比赛，未发现任何出错情况。

3 实验结果

3.1 估值函数形式的确定

3.1.1 实验环境说明

处理器 `Intel(R) Core(TM) i5-1035G7 CPU @ 1.20GHz`，内存 `8.00 GB`

操作系统 `Microsoft Windows 10` 家庭版，版本 1909

Python 版本 `3.8.1 (tags/v3.8.1:1b293b6, Dec 18 2019, 23:11:46)`

3.1.2 实验思路

估值函数是对棋盘价值进行评估的函数。经过多轮人机对战积累经验后，我们对此进行建模，并参考现有对 2048 单机版估值函数进行设计。设计完成后则是在运行时间限制下，不断调整形式平衡用时、效果以及与搜索层数，以达到最佳效果。

3.1.3 原始估值函数与分析

最初步的的估值函数如下

```
def evaluate(self, board, isFirst: bool):
    '''估值函数 TODO 返回 isFirst 方与 not isFirst 方的局面之差
    TODO 先后手记录可以两个版本？
    '''
    def func(x): return 1 << (3 * x)
    res = sum(map(func, board.getScore(isFirst))) - .125 * sum(map(func, board.getScore(not isFirst)))
    return res
```

思路为，每次移动后，对己方将所有的每个棋子数值乘 3 后作为 2 的指数再逐个求和后减去 0.125 乘以对方经相同操作得到的值。

此估值函数优势在于乘 3 并作为 2 的指数使得**大数对棋盘的价值权重增大**，由于此对局属于完全信息的零和博弈，减去对方棋盘价值有利于判断操作是否“损人利己”；缺陷在于其**不能有效反映棋盘中各棋子的关联**，在判断棋盘价值方面过于naïve，因而不能很好地完成己方合并的长远规划与在对方棋盘下子进行干扰的任务。

因此，需要一个新的估值函数，在继承现有估值函数优势前提下，体现棋盘棋子关联性带来的价值。

3.1.4 一些单机版 2048 已有的估值函数分析

Stackoverflow 用户@Khaled.K[2] 给出了如下的估值函数

[2] https://stackoverflow.com/a/22362818

```
Evaluation =
    128 (Constant)
    + (Number of Spaces x 128)
    + Sum of faces adjacent to a space { (1/face) x 4096 }
    + Sum of other faces { log(face) x 4 }
    + (Number of possible next moves x 256)
    + (Number of aligned values x 2)
```

此估值函数分为六部分

1. 常数 128，作为基线以及用于测试等其他用途。
2. 空位置数量×128，空格越多对局面越有利
3. 与空格相邻的棋子值取倒数×4096 后求和，向后评估 (by evaluating them backwardly) 可能合并的棋子
4. 堆叠棋子值取对数乘 4，我们取对数减小堆叠棋子对灵活性参数的影响。
5. 可能移动操作数乘 256。如果移动方式多，说明系统更为灵活
6. 值相同且相邻棋子数乘 2，表明可合并可能性

此估值函数可取之处为其通过考虑可能移动方向数来评价系统灵活度，与空格数相结合应该可较好地反映系统的灵活度。

简书用户@练不好 Wave 不改网名[3]给出了如下的估值函数

```
// static evaluation function
AI.prototype.eval = function() {
  var emptyCells = this.grid.availableCells().length;
  var smoothWeight = 0.1,
      mono2Weight  = 1.0,
      emptyWeight  = 2.7,
      maxWeight    = 1.0;

  return this.grid.smoothness() * smoothWeight
       + this.grid.monotonicity2() * mono2Weight
       + Math.log(emptyCells) * emptyWeight
       + this.grid.maxValue() * maxWeight;
};
```

此估值函数分为四个部分：

1. 平滑性：指每个方块与其直接相邻方块数值的差越小越平滑
2. 单调性：指方块从左到右、从上到下均遵从递增或递减
3. 空格数：局面的空格总数
4. 最大数：当前局面的最大数字, 该特征为积极因子

此算法可行度较高，但涉及矩阵分析需要对平滑性与单调性进行额外函数撰写求值，且单纯以最大数为积极因子略显粗糙。

3.1.5 估值函数撰写初步方案

综合以上，初步方案为，估值函数采用线性函数, 并添加权重系数。具体从将棋盘价值分为四部分进行评判，采用 `NumPy` 库进行矩阵运算以提高计算效率：

1. 基本分值: 通过类似现有方案形式判断当下对己是否有利
2. 自由角度：利用局面己方空格数评估
3. 平滑角度：指每个方块与其直接相邻方块数值的差越小越平滑
4. 单调角度：指方块从左到右、从上到下均遵从递增或递减.

具体实现首先将棋盘4×8棋盘转换为 `numpy.ndarray` 格式

[3] https://www.jianshu.com/p/ac01fe928118

```python
def repr_to_npboard_str: str) -> np.ndarray:
    '''将棋盘的字符串转变为 numpy 数组
    返回: numpy 的 int8 数组
    '''
    return np.array(board_str.split(), dtype=np.int8).reshape((4, 8))
```

接着则分四部分分别进行估值函数的撰写。基本分值 (basicValue)

```python
def basicValue(npboard, belong):  # 基本分值:己方所留棋盘分值，强化了处于劣势对方所留棋盘价值
    if belong:
        myBoard = np.maximum(npBoard, 0)
        return np.sum(8**myBoard[...,:4]+8**myBoard[...,4:]*7)
    else:
        myBoard = np.abs(np.minimum(npBoard, 0))
        return np.sum(8**myBoard[...,:4]*2+8**myBoard[...,4:])
```

可移动性 (movability)

```python
def movability(board, belong):  # 棋盘可移动性：由移动后空格数决定
    length = len(board.getNone(belong))
    if length <= 6:
        return 2**(2+length)
    else:
        return 0
```

平滑性 (smoothness)

```python
def smoothness(board,belong):
    #1.得到belong方的棋盘并创立辅助棋盘
    if belong:
        myBoard = np.maximum(npBoard,0)
    else:
        myBoard = np.minimum(npBoard,0)
    rowZeros = np.zeros((1,8),dtype=np.int8)
    colZeros = np.zeros((4,),dtype=np.int8)
    #2.将己方棋盘进行移位以便进行矩阵运算得当棋盘平滑性价值
    npLeft = np.concatenate((myBoard[...,1:],colZeros),axis = 1)
    npRight = np.concatenate((colZeros,myBoard[...,:-1]), axis=1)
    npUp = np.concatenate([myBoard[1:],rowZeros])
    npDown = np.concatenate((rowZeros,myBoard[:-1]))
    #3.计算每个棋子与周围棋子的差值并计算结果值: -1为最佳，平滑性越差值越低
    result = np.zeros((4,8),dtype=np.int8)
    npList = [npLeft,npRight,npUp,npDown]
    for i in npList:
        result -= 2**np.abs(i-myBoard)
    return np.sum(result)
```

单调性 (monotonicity)

```python
def monotonicity(npBoard,belong):#方块从左到右，从上到下均遵从逐渐缩减单调性分值
    #1.得到belong方4*4棋盘
    if belong:
        myBoard = np.abs(npBoard[...,:4])
    else:
        myBoard = np.abs(npBoard[...,4:8])
    monotonicities=0
    #2.对左右方向求单调性
    for i in range(4):
        row = myBoard[:,i]#得到每行非零项
        a = row[row>0]
        if len(a) <= 1:
            continue
        delta = np.abs(a[:-1] - a[1:])#得到每行相邻项等级差
        for j in range(len(delta)):
            if delta[j] <= 1:#若等级差<=1则强化两相邻项的价值
                monotonicities += 4**a[j] + 4**a[j+1]
            else:
                monotonicities -= 4**delta[j]
    #3.对上下方向求单调性
    for i in range(4):
        col = myBoard[i]
        a = col[col>0]#得到每列非零项
        if len(a) <= 1:
            continue
        delta = np.abs(a[:-1] - a[1:])#得到每列相邻项等级差
        for j in range(len(delta)):
            if delta[j] <= 1:#若等级差<=1则强化两相邻项的价值
                monotonicities += 4**a[j] + 4**a[j+1]
            else:
                monotonicities -= 4**delta[j]
    #4.得到单调性值
    return monotonicities
```

对以上四个选取合理的参数进行线性运算，得到估值函数最终返回值

```
smoothweight = 2.8
monoweight = 2.0
moveweight = 3.0
basicweight = 1.8

return basicweight*basicValue(npBoard, isFirst) \
+ monoweight*monotonicity(npBoard,isFirst) \
+ moveweight*movability(board, isFirst) \
+ smoothweight*smoothness(npBoard,isFirst) \
- .25*(basicweight*basicValue(npBoard, not isFirst) \
+ monoweight*monotonicity(npBoard, not isFirst) \
+ moveweight*movability(board, not isFirst) \
+ smoothweight*smoothness(npBoard, not isFirst))
```

但该估值函数是在早期搜索树还不是很深的时候设计的，在搜索层数加深后，超时现象较为频繁发生。故，我们对估值函数形式做了一些简化，改为最终使用的形式（见 2.2.4，注重己方合成同时，借助增强敌方区域己方棋子的分值加强了进攻性），并对参数的具体选择进行了实验。

3.2 确定估值函数参数

3.2.1 实验环境说明

处理器 AMD Ryzen 7 3700X 8-Core @ 16x 4.05GHz，内存 64.00 GB

操作系统 Manjaro 20.0.2 Lysia, x86_64 Linux 5.4.43-1-MANJARO

Python 版本 3.8.3 (default, May 17 2020, 18:15:42)

3.2.2 实验思路

确定了估值函数的形式后，我们需要确定估值函数的最佳参数。

我们的估值函数共涉及四个参数。我们规定，外生参数为 BASE_POWER 和 ENEMY_WEIGHT，内生参数为 BONUS_POWER 和 ENEMY_BONUS_POWER。我们希望通过使用不同参数、在本地运行大量次数的对战来确定最佳的外生参数。

3.2.3 实验步骤

通过自动生成不同参数的 Player 文件，我们利用技术组提供的 round_match 工具在本地发起大量对战。将不同参数下我方的胜率总结于表格中，通过综合排序比较得出最佳参数。

3.2.4 实验数据

以下数据为我方胜率。原始比赛记录可在 https://github.com/CastleStar14654/2048-Bad-Gateway/tree/master/experiment 在线查看。

假定敌方参数为：BASE_POWER, ENEMY_WEIGHT = 3, 1，我方的参数如下表显示。先后手各进行 100 场比赛。

我方为先手

E_W / B_P	0.75	1	1.25
2	61	56	75
2.5	53	58	50
3	51	/	64
3.5	48	53	56
4	59	50	56

我方为后手

E_W / B_P	0.75	1	1.25
2	63	53	61
2.5	34	39	42
3	43	/	43
3.5	31	38	41
4	31	37	36

可以发现，在 BASE_POWER, ENEMY_WEIGHT = 2, 1.25 时，先手胜率最高，后

手胜率也表现亮眼。因此，又假定敌方参数为：`BASE_POWER, ENEMY_WEIGHT = 2, 1.25`，我方的参数如下表显示。先后手各进行 40 场比赛。数据如下

我方为先手

E_W B_P	1	1.25	1.5
1.5	45	68	70
1.75	70	60	75
2	45	55	68
2.25	63	53	55
2.5	33	43	45

我方为后手

E_W B_P	1	1.25	1.5
1.5	52	65	68
1.75	53	60	53
2	35	45	40
2.25	40	33	40
2.5	38	30	33

可以发现，在 `BASE_POWER, ENEMY_WEIGHT = 1.5, 1.5` 时表现较好。

但在将代码上传到天梯上与其他代码进行对战时，我们发现参数 1.5, 1.5 效果不佳。因此，考虑到己方需要获得更大的权重，我们最终选择了第一轮实验中表现也较好的 2, 0.75。

3.3 经典战局

在天梯赛的一场比赛中，出现了如下状况：比赛初期与对方僵持，中后期我方大棋子被对方合并，从此一蹶不振，比赛失利。具体战况如下。

对方进攻性较强，在下图情形中，我方（后手）只能选择向左移动，将 1024 送入对方的合并范围内。

对方由此合成了 2048，相较我方拥有了压倒性优势。

通过对这场失利比赛的分析，我们对代码进行了进一步的改进：改进了估值函数，增加我方棋子在对方棋盘中的权重，增加对方棋子在我方棋盘中的权重，以增加代码的进攻性，减少该类失利的发生概率；建树时更改假设，更多考虑对方会进攻的情况。

4 实习过程总结

4.1 分工与合作

根据自愿、发挥各人长处的原则，小组成员分工如下：

- 林旭辰：组织讨论；代码总体框架编写；报告第二部分
- 白康博：向对面落子部分代码编写；报告第一部分
- 龚德炜：向对面落子部分代码编写；报告第一部分
- 郭洋帆：估值函数代码编写；报告三、四部分
- 王依婷：估值函数代码编写；报告三、四部分

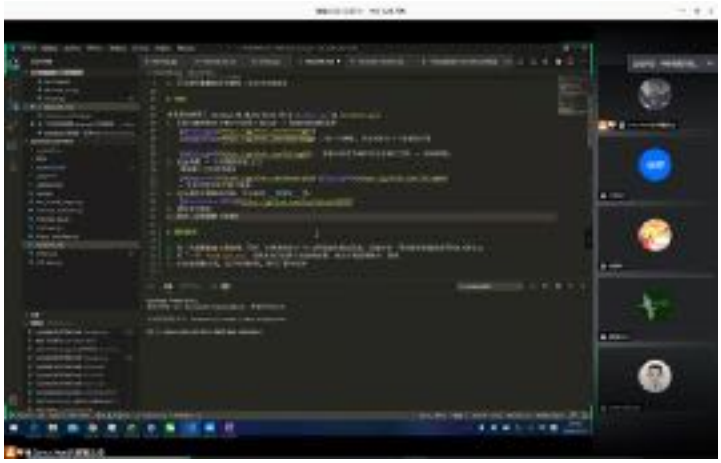
小组共进行了九次讨论，受线上条件的限制，讨论均在腾讯会议平台进行。下图为九次会议的会议发起记录。

每一次会议中，小组成员均能积极的参与到讨论中来，主动承担各项任务。会议记录摘要如下：

- 2020.5.6 第一次会议：小组参加大作业内测，小组成员第一次线上见面，了解了 git 的使用，并初步了解了大作业的要求
- 2020.5.6 第二次会议：组长林旭辰直播人机对战 2048，小组成员对该游戏有了更深的了解
- 2020.5.15 第三次会议：讨论代码编写思路，确定代码总体框架为 minimax + alpha–beta 剪枝。下图为部分讨论思路（可参见项目 `README`）。

- 2020.5.18 第四次会议：在代码总体框架完成后，进行剩余代码的讨论和分工。下图为该次会议记录（部分，可参见项目 `README`）。

- 2020.5.24 第五次会议：代码汇总，确定修改方向
- 2020.5.25 第六次会议：根据修改情况进行进一步的讨论
- 2020.5.27 第七次会议：预计小组将顺利出线，根据进行的估值函数参数实验进行最后的参数讨论。右图为会议截图
- 2020.5.31 第八次会议：比赛结束，讨论报告编写思路并初步



分工

- 2020.6.6 第九次会议：完成报告前三部分后，理顺已有报告逻辑，讨论报告剩余部分编写思路并分工

在本次实习作业中，小组成员不仅完成了自己部分的任务，更参与到了作业整体的讨论中来。在一次次高效率的讨论中，我们不断摸索和前进，发挥了合作的力量。

此次实习过程中，我们于各方面收获了不少经验与教训。

协作方式上，通过此次实习，我们可以较为熟练地利用 git 以及 GitHub 平台进行远程项目协作。之前进行需要合作的大作业时，由于同学们都在校内，很多情况下就是面对面讨论协作，通过微信等进行各自部分代码的汇总，但经常会出现版本错乱，文件复杂难以管理的情况。此次疫情居家通过 GitHub 进行项目协作，用较小的学习成本得到了很好的协作体验，为今后项目协作积累了宝贵经验。

合作层面上，我们体会到合作过程中时刻交流沟通的重要性。在完成 minimax 搜索树配合 alpha–beta 剪枝算法进行 AI 实现的主体部分后，我们分工完成了于对方下棋子干扰的函数与评估局面好坏的估值函数，但由于缺乏交流沟通，导致虽然各部分完成度都很高，但结合到一起后出现了严重超时的情况，之后不得不反复修改。协作过程中缺乏沟通很可能会出现$1 + 1 < 2$的情况。

时间管理上，我们较好地规划了各任务的时间得以较早地完成了项目，为后续的进一步完善与修改留下了充足时间~~，也感受到了不做 ddl 战士的快乐~~。在项目进行的两周内，我们共进行了九次会议进行任务的分配与完成度反馈以及问题交流，这些都为今后小组协作的时间管理留下了宝贵经验。

4.2 建议与设想

在最后比赛中，我们小组与 N19_666 组进行季军的角逐，最终以 24-26 止步四强，但在平时对战以及之前 4 进 2 比赛由于淘汰赛设施错误而匹配到 666 组的对战中，我们小组大多数情况下对 666 可以保持一定的优势。因此希望就最终决赛赛制提出相关建议与设想。

运气固然是实力的一部分，但旨在比较算法优劣的比赛不应使与算法无关的随机序列对胜负有较大影响。希望可以在决赛中对胶着场面进行适当加赛，以更好地确保公平性。

5 致谢

通过此次实习，我们更为深刻地理解掌握了树这一数据结构以及 minimax 配合 alpha–beta 剪枝算法的内涵，同时提高了小组协作与编程的能力，收获极大。

实习现已结束，首先特别感谢陈斌老师与助教们的二零四八大作业创意，丰富课程内容、增强课程趣味性同时也为同学们进一步理解掌握课堂内容提供了平台；同时也特别感谢老师、助教以及技术组同学们的辛苦付出，使同学们可以仅凭借课程内容进行大作业实习，同时确保了大作业平台的平稳、顺利运行；最后也十分感谢其他各小组同学，特别高兴可以与你们在竞争、比赛过程中共同进步！

6 引用

Khaled.K. (2014, 3 12). *What is the optimal algorithm for the game 2048?* Retrieved from stackoverflow: https://stackoverflow.com/a/22362818

von Neumann, J. (1928). Zur Theorie der Gesellschaftsspiele. *Math. Ann., 100*, 295-320. doi:10.1007/BF01448847

Wikipedia contributors. (2020, 6 6). *Alpha–beta pruning.* Retrieved 6 9, 2020, from Wikipedia, The Free Encyclopedia.: https://en.wikipedia.org/w/index.php?title=Alpha%E2%80%93beta_pruning&oldid=961089584

练不好 Wave 不改网名. (2020, 2 4). *MinMax-极小极大算法——2048.* Retrieved from 简书: https://www.jianshu.com/p/ac01fe928118

数据结构与算法课程实习作业报告

李木子*，白冰玉，成昱璇，段宇轩，李雨欣

摘要：本小组主要通过部分加深的贪心算法，结合启发式算法，编写了能够完成“二〇四八”游戏对局，并具有棋盘结构分析、行动收益与风险评估、主动攻击与防御等功能，具备一定实战能力的对战 AI。
关键字：贪心算法，启发式算法，递归，映射

1 算法思想.....1
1.1 总体思路.....1
1.2 算法流程图.....2
1.3 算法运行时间复杂度分析.....2
2 程序代码说明.....3
2.1 数据结构说明.....3
2.2 函数说明.....3
2.3 程序限制.....4
3 实验结果.....5
3.1 测试数据.....5
3.2 结果分析.....6
4 实习过程总结.....6
4.1 分工与合作.....6
4.2 经验与教训.....7
4.3 建议与设想.....8
5 致谢.....8
6 参考文献.....8

1 算法思想

1.1 总体思路

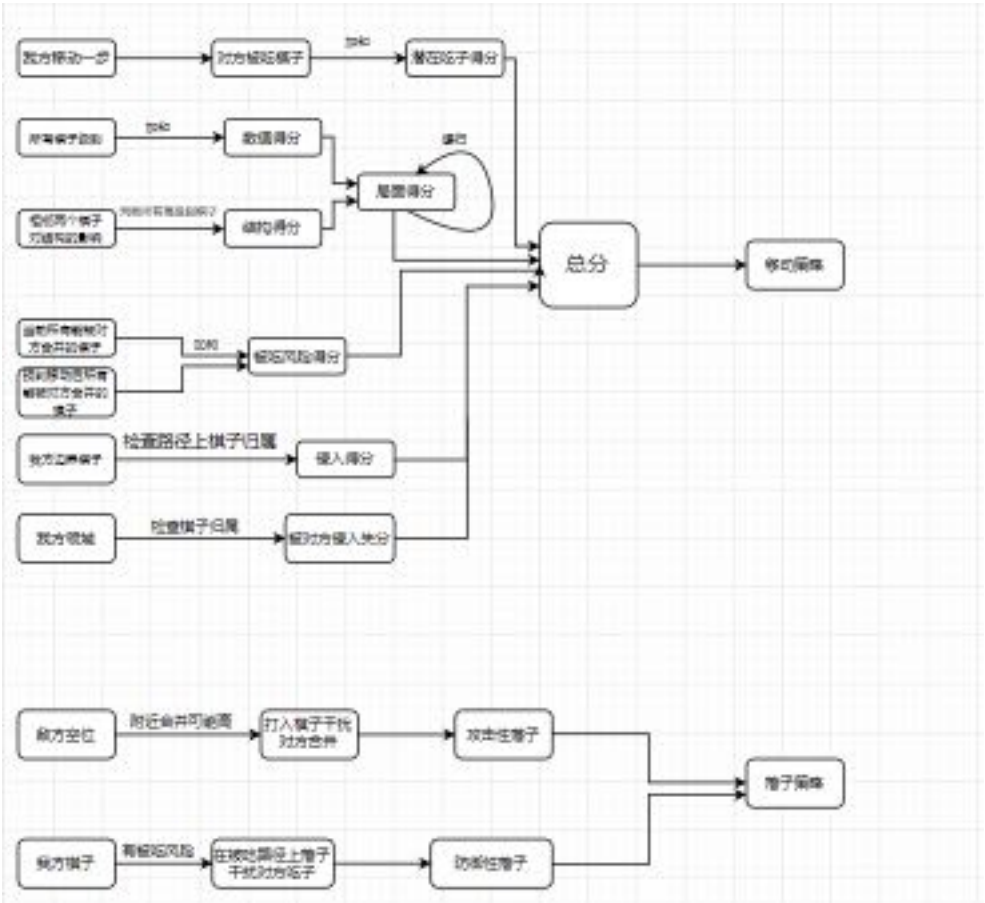
本小组开发的“二〇四八”程序主要决策方式为：结合对局经验，设计落子策略，达到尽可能干扰敌方、建立己方防御的目的。同时，通过对当前棋盘局面及不同行动后的棋盘局面的分析，结合被吃风险、侵入对方棋盘的收益，对不同行棋方式进行估值打分，进而选择对己方最有利的移动策略，即经过一定优化与部分加深，结合启发式算法的贪心算法。

结合人类玩家的对局经验与对已有经典对局的分析（即“启发式算法”），在落子策略的选择上，我们设计了主动攻击与防御的两项函数，达到帮助我方形成并保护优势结构、保护我方高级别棋子、破坏敌方优势结构的目的。

对棋盘局面的打分包括数值得分与结构得分等。数值得分是一个仅与己方棋子级别和个数有关的函数；结构得分则是通过己方棋子的分布模式，判断当前己方所有棋子分布结构的

有利程度，给出打分。在实现与调试过程中，我们又通过递归和循环，在结构得分的判断中加入了对之后多个回合局面的预判，进一步增加了程序的预判深度，提高了对当前结构的评估能力。除此之外，为提高对当前局面判断的精准程度，我们将我们将被吃风险、侵入收益、被侵入损失、潜在吃子等分数计入总分。

1.2 算法流程图



1.3 算法运行时间复杂度分析

本程序中，可以认为，通过棋盘类实现的操作的时间复杂度均为 O(1)，则影响时间复杂度的操作主要为：对棋盘的遍历与对局面变化的预判。

在 cal_invade 函数中，对对方领地的棋盘进行了遍历，并且对入侵棋子的四个方向进行检查打分，具体的赋值或比较次数因不同的入侵局面及附近空格数目而有所不同，时间复杂度为 O(n)。

在 get_chess，find_hotspot 等函数中，均进行了对棋盘的遍历操作。假设棋盘大小为 n，则遍历一次棋盘，时间复杂度为 O(n)。

对局面变化的预判是指，通过模拟己方与对方的操作，判断下一回合乃至多个回合局面的变化，在 get_point_atrisk，search_trap，recur_struc 等函数中应用。

在 get_point_atrisk 中，每预判一次，就对当前棋盘进行一次遍历，同时对移动前后己方棋子集合做差。遍历与集合相减的时间复杂度均为 O(n)，故该方法的时间复杂度为

$O(n)$。

在 search_trap 中，每进行一次搜索，需要预判两层，故时间复杂度为 $O(n^2)$。

在 recur_struc 中，实际操作中，考虑到时间开销，我们仅采用了三层递归，故总的时间复杂度为 $O(n \cdot 4^3)$

综上所述，本程序的时间复杂度为 $O(n \cdot 4^m)$，其中 n 为棋盘的尺寸，m 为迭代递归的层数。而在实际对战中，我方最长运行时间为 4 秒左右，未超过最大运行时间限制 5 秒。

2 程序代码说明

2.1 数据结构说明

我们主要使用的数据结构包括集合、列表、映射等。这些数据结构的主要功能为存储我方棋子、计算局面得分，因此这些简单的数据结构及其内置的排序、求和、增减等方法就已经可以满足需要。

值得注意的是，在 get_point_atrisk 函数中，为实现对我方损失的棋子的查找，我们使用了集合（set）这一数据结构来存储移动前后我方棋子，有利于直接进行“相减”操作。

2.2 函数说明

2.2.1 局面得分函数

recur_struct 是一个判断当前棋盘下某一方局面得分的打分函数，包括数值得分与结构得分两方面，并可以进行多层迭代计算分数。

数值得分由 calc_scores 函数计算给出。数值得分仅仅与棋子多少、棋子级别高低有关，是一个简单的加权求和。

结构得分由 calc_structures 函数给出。该函数通过遍历己方所有棋子，并根据该棋子的级别和紧邻的四个棋子（邻居棋子）的级别，计算其结构得分。结构得分由中心棋子的级别和紧邻四个棋子的级别共同决定，且仅计入邻居棋子级别小于中心棋子的情况。中心棋子级别越高，结构分越高；中心棋子的级别与邻居棋子相差越小，结构得分越高。

这两部分得分最终在 recur_struct 函数中进行汇总，并多层迭代，预判一定局数之后，我方的最终局面得分。

2.2.2 被吃风险估值函数

calc_risk 是一个对某一方被吃风险进行估值的函数。函数通过模拟一次对方任意方向的移动，查找出移动前后己方被吃棋子，计算出己方被吃棋子的总价值。其中，模拟移动及查找己方被吃棋子的功能由 get_point_atrisk 函数实现。

2.2.3 侵入收益打分函数

在实际对局中，侵入对方的领地、扩大我方领地范围的收益非常可观。calc_invade 是一个评估己方侵入对方领地棋子的函数，计算了在中线的另一侧己方棋子占据空间的大小，并且依次进行打分；此外，它针对每一个我方侵入对方领地的棋子，分别向四个方向进行检查，从而评估出该入侵棋子与附近棋子级数的大小差异，最终实现量化侵入的受益。在参数设置上，级数差异越大，则打分越高；如果级数相等，即存在被侵吞风险，则得分极低。

2.2.4 陷阱搜索及打分函数

在对局中，有一类特殊结构的贡献十分明显：陷阱，即通过把自己的棋子送到能被对方侵吞的位置，来让对方吃，但其实对方吃后，我方能够立刻实现反吃（即我方在被吃子的上下左右连通的位置有 2 倍大的棋子）。

我们通过 search_trap 函数实现了寻找我方可以利用的陷阱的功能。该函数通过调用 get_point_atrisk 函数，模拟两次移动，找到两次移动中己方和对方被吃的棋子，重叠位置即为陷阱位置。

得到陷阱位置之后，通过 calc_trap 函数实现对陷阱的估值。陷阱数目越多，级别越高，则陷阱价值越高。

2.2.5 吃子打分函数

calc_eat 函数用于计算我方移动后，对方受到威胁的棋子得分。我方移动一步之后，对方棋子如果在任一方向可能被吃，则计入得分，被吃棋子级别越高，数目越多，得分越高。这一函数既可以提高我方的进攻性，也可能逼走对方的侵入。

2.2.6 防御性落子函数

当我方较高级别棋子有被吃的风险时，则通过 defence 函数在中间落子，以保护我方棋子。函数首先搜索出所有有风险的棋子及敌方所在的方向，根据其本身级别、我方剩余空格，对保护该棋子的价值进行评估，并通过排序，选择出最值得保护的棋子进行落子保护。

2.2.7 攻击性落子函数

attack 函数是一个通过在对方高级棋子附近落子进行干扰，迫使对方形成不利结构的函数。通过 find_hotspot 函数可以寻找到对方边角容易被卡死的位置及卡死价值，并返回卡死价值最高的位置（热点位置），进而，可以在热点位置附近落子，卡死对方高级棋子。

2.2.8 主函数（Player 类）

在落子回合中，我方操作的优先级为：攻击性落子最高，防御性次之。如果不存在攻击和防御需要，则判断按照随机序列在己方落子之后，己方可移动方向是否小于等于一个，如果是，则在对方随机落子，如果不是，则在己方根据随机序列落子。

在移动回合中，我方则根据 2.2.1-2.2.5 的打分函数，对所有移动结果进行打分，选择分数之和最高的方向进行移动，即贪心算法。

2.3 程序限制

在回合数大于 497 时，由于 recur_struct 函数进行了 3 层递归，因而会对大于 500 回合的棋盘进行操作，这是无效的，我们在天梯对战中发现了这一问题并最终进行了修正。

在 recur_struct 函数中，我们在递归过程中只考虑了双方按照默认落子的情况进行下棋，也假设对方棋子不会吃到我方领地。这种假设在绝大多数情况下都是正确的，节约了大量预测对手方行动的算力，但是在面对看家小萝莉和 777 等进攻性较强的 AI 会导致计算错误，从而难以取胜。

在 defence 函数中，我们简略地认为，一个有被吃风险的棋子仅能从一个方向被对方吃掉，如果存在多个方向，则我们做出的防御很可能是无效的。

在搜索过程中，为了简化起见，我们只对结构得分进行了三层搜索，而对于其他函数没有进行多层搜索，这导致了我们的 AI 无法识别多层陷阱的情况（大于两层），不会向最优的情况进行移动。但这种情况在实战中遇到的比例并不高，所以没有作为优化的主要方向。

3 实验结果

3.1 测试数据

实验环境说明：

- 硬件配置： i5-8250U 8GB
- 操作系统： Windows 10. 1909
- Python 版本： Python 3.6.7

我们进行了两个阶段的测试。第一阶段是不断调整参数，增加应对策略，利用 userinterface 工具在本地进行 AI 之间的对战，选择出胜率最高的一版 AI；第二阶段是在天梯上与所有参赛 AI 反复对战，回放对局，找到失误点，改进参数、增加应对策略。

热身赛结果：

版本号	总场数	胜率	天梯分	排名
Version 3.0	211	42.2%	966	-
Version 4.0	587	58.2%	1340	14
Version 4.1	359	55.3%	1208	-

天梯赛结果：

版本号	总场数	胜率	天梯分	排名
Version 4.0.1	3953	57.5%	1422	5

淘汰赛结果：

版本号	对手	比分
Version 4.0.1	N19_X-ray	26:24
Version 4.0.1	N19_777	12:29
Version 4.0.1	N19_404	26:24

最终排名：N19 组第 3 名

图 1

图 1 显示了在 AI 调试的最初阶段，我方作为先手时一次移动方向的错误决策，根据当时已有程序我方选择了向上移动而非向下移动。我们推测是结构函数的计算上出现了问题，所以检查和完善了结构函数。

此后，我们的程序改进进入了瓶颈期，虽然在进攻性方面表现很好，但是结构函数还存在较大的问题，主要表现在不能进行可持续地运营，我们认为这是由于结构函数只预测了一次移动后的局面，而成功的棋子合并往往需要较多次的“谋划”。因此，我们在此基础上开始考虑运用多层迭代的方式来计算局面得分，即添加 recur_struct 函数。

图 2

在后来的调试中，我们还遇到了因我方随机落子在棋盘内而导致自己被卡住不能移动的问题：如图 2 所示，我方随机落子在左下角即（3，0），使得自己不能通过移动将两个 8 合并为 16。为了解决这一问题，我们添加了 bad_draw 函数，通过 Array 对随机落子位置进行预测，从而规避不利落子。

3.2 结果分析

与课程中其他小组相比，我组编写的 ai 在搜索深度方面较弱，仅在 search_trap 中有两次预判、在 recur_struct 中有三次递归迭代。但在对局中，我方 ai 一直保持了较高的对战水平，以 N19 组天梯赛第四名出线，并最终夺得了季军。这可能得益于，我们根据游戏经验，结合启发式算法，总结出了一些经验性的策略，例如陷阱、攻击性和防御性落子等，使 AI 直接搜索特殊结构，完成快速决策。

主要消耗时间的步骤为 recur_struct 中的三次递归迭代，我们发现，增加递归次数到四层时，很容易发生超时，因而最终迭代次数保留为 3。

4 实习过程总结

4.1 分工与合作

我们小组分工如下：

李木子（组长）：主要策略设计、主函数、attack 与 defence 函数、recur_struct 函

数、测试与策略调整；

白冰玉：侵入收益函数、被吃风险函数；

成昱璇：策略调整、代码规范化

段宇轩：测试与调参、策略调整；

李雨欣：陷阱搜索及打分函数、局面得分、结构得分、报告主要部分撰写。

进行一定的资料收集，各自进行游戏、熟悉游戏规则之后，我们在腾讯会议上进行了第一次小组讨论，对使用的主要策略有了统一认识，分工编写了各个决策打分函数。

将 AI 各部分组装之后，我们主要通过微信群讨论，进行各部分 debug 和调参工作，并在对战后，在群聊中及时反馈对战结果，调整参数、增减策略。

图 3 微信群讨论（A:编写函数中；B:调参；C:获得季军的庆贺）

4.2 经验与教训

在最初的讨论中，得益于组长合理的分工部署，我们对完成整个项目的操作步骤做了合理的安排，从起初的设计子函数到后来的调参及策略调整，以及在 AI 较为成熟后不断根据对战中的表现来优化 AI 和增加新功能，整个过程中我组思路清晰、效率较高，没有因缺乏沟通等原因做不必要的重复操作，工作得以有条不紊地进行。

如前文所述，我组 AI 在搜索深度、决策树建立和剪枝方面较弱，依赖“策略”和启发式算法的程度较高。这一方面降低了我们编写 AI 的难度，而另一方面也决定了，我组 AI 的上限并不能和许多搜索层数较深、剪枝策略较完善的 AI 相比，而热身赛和小组赛中的对局结果也说明了这一点。

我们的代码分享和交流主要依赖微信群文件，这可能会导致组员之间在代码版本上存在一定的混乱，但由于项目较小、学习使用 GitHub 需要一定时间，我们最终还是选择了微信群作为主要的共享平台。

本地优化和天梯优化都是非常重要的调参环节，通过多次复盘、寻找失误，我们可以更方便地发现调参方向、改进现有算法。

4.3 建议与设想

可以发现，我们在大作业的编写过程中，主要使用的数据结构和算法较为简单，与课程内容相距较远，在浏览往年实习报告的过程中，也发现了类似的现象。之后的大作业或许可以增加栈、队列、树等数据结构与排序、动态规划等算法的实际应用，与课程内容更好结合。

5 致谢

感谢陈老师的精心准备和耐心授课，在这个特殊的学期，我们都收获了很多；
感谢助教、技术组同学的帮助，对战平台相当完善、大作业非常有趣；
感谢所有的选课同学和旁听同学，微信群中围观大佬发言总能提醒我们还有多少要学习；
感谢 bilibili、中国大学 mooc，在这特殊的一学期为我们提供了方便的学习平台！
感谢小黑！在这个学期里提供了无数梗、表情包和快乐的时光~

谢谢小组的五位同学，合作愉快~

6 参考文献

[1] https://wiki.python.org/moin/TimeComplexity
[2] 知乎-游戏 2048 的基本游戏策略是什么？

数据结构与算法课程实习作业报告

王逸飞* 郁桂花 刘鸿骏 陈董泽 林翔宇

摘 要：本组使用 minimax 算法，配合 alpha-beta 剪枝实现了 2048 游戏的 AI 程序。此外还使用了经过多次完善的估值函数、用于避免重复搜索的储存树、用于充分利用决策时间的简单-复杂转换模式和搜索深度浮动机制。本组最终在比赛中取得了优异结果：N19 组天梯赛第 3 名出线，并在决赛中取得亚军的成绩。

关键字：决策树，Minimax，Alpha-beta 剪枝，启发式规则，存储表，递归

1 算法思想.....2
1.1 总体思路.....2
1.2 算法流程图.....5
1.3 算法运行时间复杂度分析.....5
2 程序代码说明.....6
2.1 数据结构说明.....6
2.2 函数说明.....7
2.3 程序限制.....7
3 实验结果.....7
3.1 测试数据.....7
3.2 结果分析.....8
3.3 经典战局（可选）.....10
4 实习过程总结.....10
4.1 分工与合作.....10
4.2 经验与教训.....10
4.3 建议与设想.....11
5 致谢.....11
6 参考文献.....11

1 算法思想

1.1 总体思路

这里介绍决赛中使用的 AI。

此 AI 采用 minimax 算法，配合 alpha-beta 剪枝，使用树结构存储搜索过的局面。Minimax 算法和 alpha-beta 剪枝不再介绍，下面主要说明我们采用的估值函数、落子位置选择的启发式规则、搜索深度调节策略、存储结构与算法的交互等方面的细节。

估值函数分四部分。第一部分是棋子面值（注意，不是等级）的 1.1 次幂求和，双方做差；第二部分是双方占领格数差；第三部分是双方未填充空格数的差；第四部分是统计双方相邻且相等的棋子对的数目，并做差。另外，若搜索到无法再往下搜索的局面，例如无合法合并方向，即使搜索深度不够，也直接返回估值，但为了惩罚这种情况，我方此时会在原估值函数基础上减 33，对方会加 33。

落子位置如果全部搜索，将大大增加搜索时间，但其中很多位置其实是不可能被选择的，为此我们采用启发式规则限制我们考虑的落子位置。规则如下：

- 如果下一步在我方有位置下，根据深度调节策略决定是否使用主动落子模式；否则使用被动落子模式。
- 主动落子模式：
    - 包含对方棋盘上所有满足如下规则的空位：在下一次合并时可以碰到的上、下、左、右四个棋子中，有两个面值相同且等级不小于三。
    - 当处于**“复杂”模式**[1]且对方空格数不超过 3 个时，包含对方所有空格。
- 被动落子模式：
    - 第一备选序列与主动落子模式第一条规则决定的空位相同。（尽量捣乱）
    - 第二备选序列包含对方棋盘上所有在下一次合并时不会和等级为 1 的棋子相遇的空位。（次之不喂到嘴边）
    - 优先采用第一备选序列，第一备选序列为空则采用第二备选序列，第二备选序列也为空则选择 getNone 返回的第一个元素。（实在不行就躺平平）

[1] 标黑体的是随后会介绍到的重要概念。下同。

深度调节策略的意义在于，在游戏的开始阶段，形势比较简单，较难拉开差距，此时应该注意节能；在中间阶段，形势复杂，应该尽量加深搜索；在时间快用完时，应该避免超时。需要注意的是，最后阶段也是很重要的，因此中间阶段也不应过度占用资源。具体的策略如下：

- 分成简单模式和复杂模式。
    - 简单模式，合并和落子搜索深度均为 2 层。（搜索深度的层数，是指搜索中涉及的合并次数，下面涉及这一概念将在括号中注出考虑到落子的层数。如果算上落子，这一深度对于先手合并为 2，落子为 4，对于后手合并为 4，落子为 3。这样定义搜索深度，是出于我们组第一版代码没有考虑落子，而后续代码是在其上扩展的这一历史原因。）
    - 复杂模式，先手合并搜索深度根据**时间敏感深度浮动机制**为 4 层（6 层）或 6 层（10 层），后手合并搜索深度相应为 3 层（5 层）或 5 层（9 层）。落子则根据上一次合并搜索的搜索树直接选择最有利的局面，如果对方将局面引到了上一次合并搜索时被剪枝的分支，那么则另行搜索，深度与合并搜索的深度相同。
- 前 4.3 秒采用复杂模式，剩 0.7 秒时切换到简单模式。
- 复杂模式中引入时间敏感浮动机制，当$\frac{\text{time left}-0.7}{5} > \frac{500-\text{currentRound}}{500}$时，搜索深度为 6，否则为 4。每 20 秒更新一次。
- 前 50 回合不启用主动搜索模式。（因此一般而言只有一个落子位置，只有一个落子位置时不搜索直接输出结果。）

搜索过的局面的存储直接由搜索树实现，搜索树的每一个节点，存储当前局面对应的棋盘、达到这个局面所做的决策、决策者、决策模式、父节点、子节点的 list、minimax 搜索中选择的子节点等七项内容。整个游戏中决策的过程，其实是一个“爬树”的过程。叙述如下：

- 每次决策前，读取对方上一次决策内容，如果该决策已被搜索过，将树根平移到搜索过的那个节点；如果未搜索过，则重新创建树根节点。
- 决策过程中，若搜索的节点已被构造，则不重新构造，直接往过“爬”；否则构造新节点。注意，如果是落子的决策且备选序列只有一个位置，则不搜索，直接返回这个位置。
- 做出决策后，将树根移到这个决策对应的节点。

下面介绍算法的发展历程：

在组里第一次讨论后，我们分别写了两个算法：陈董泽编写了基于贪心算法的 AI，刘鸿骏编写了基于 minimax 算法的 AI，在进行对比并且发现 minimax 占绝对优势后，我们决定使用 minimax 为 AI 主要算法。

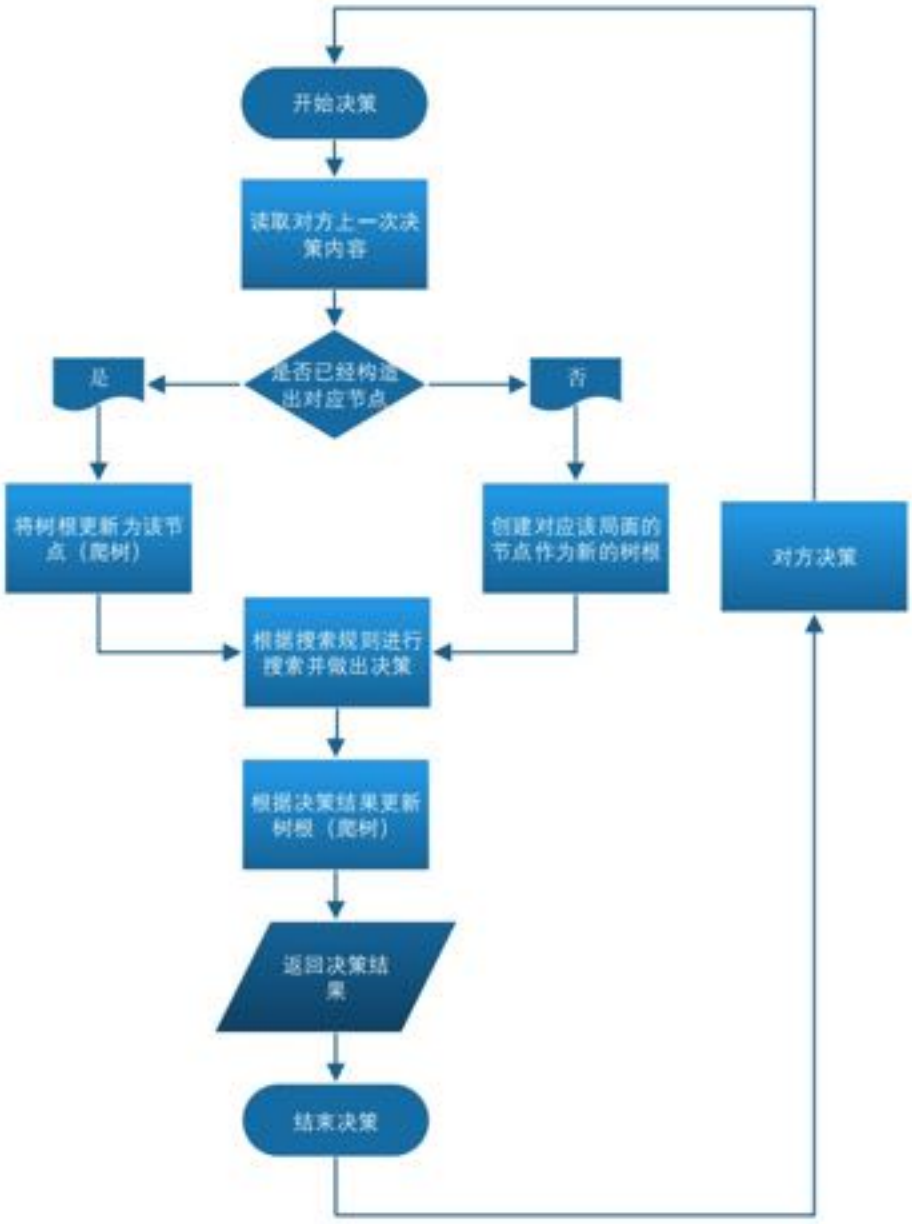
AI 的总体构想以及估值函数由刘鸿骏给出，同时使用了王逸飞的储存树结构以及部分细节调整。

估值函数的改进是刘鸿骏在做，从最开始的只是让 AI 会玩正常的 2048（只考虑在决策移动方向时，模拟双方各移动一次，使用的是 minimax 算法，局面评分函数是棋子面值和加空格数）。到后面不断尝试进行完善：增加搜索深度、占领对方领地限制对方移动、防止我方大子被对方吞并、为防止超时在 5s 的决策时间内切换简单-复杂模式，鸿骏不断地完善着我们 AI 的估值函数。后来尝试使用 RootStrap 和 TreeStrap 算法优化系数，但由于算力限制无法收敛。

王逸飞拿到刘鸿骏的代码之后开始魔改，尝试加上树来存储搜索过的局面，以减少复制次数。这样增加了存储结构（于是代码瞬间增加到了四百多行，debug 用了一天半）。最初的代码没有落子的搜索，王逸飞加上了上述启发式规则来限制搜索范围，尝试加入落子的搜索。最开始的启发式规则，只考虑堵某一方自己的合并，因此在 get_left 这些函数里面遇到另一方棋子直接返回 0，后来郁桂花在研究战局的过程中发现这种方式很容易使得自己的大子被对方吃掉，于是改进为另一方边界上的棋子也视为即将合并的棋子——本来就该这样，但最开始没想到，后来被发现，也体现了团队分工的必要性。

此外为了充分地利用决策时间，逸飞引入了搜索深度的浮动机制并将其加到复杂模式中来。本地测试的结果不理想，因为平台没有使用加速，性能较差，开始阶段的深度浮动会挤占后面的时间，使得游戏过早进入简单模式；在天梯上测试效果还不错，最终决定在决赛中使用。

1.2 算法流程图



1.3 算法运行时间复杂度分析

如果没有剪枝和存储，则构建高度为d的搜索树的时间复杂度为$O(m^d)$，m是一个节点子节点的平均个数。容易证明理想的 alpha-beta 剪枝，即返回的决策始终是第一个搜索的，会使得时间降低到$O\left(m^{d/2}\right)$。我们通过实验发现，最容易选

择的方向是进攻，其次是上下，最次是退回，由此设计了搜索顺序，可以说是一个最优的顺序。但这一顺序并不总是对应理想的剪枝，因此时间可以记为$O(m^{d/s}), 1 < s < 2$。（s的取值在 3.1 和 3.2 节中会进一步讨论。）

由于我们引入了存储机制，设有比例为α的节点被重复搜索，那么构建决策树的时间应再乘上一个系数$(1-\alpha)$，这并不影响时间复杂度的渐进表示，但对于我们的实际应用来说，系数也是非常重要的。

另一部分时间是计算估值函数的时间，我们只对叶节点计算估值函数。不剪枝，这一数值显然是$O(m^d)$，考虑到剪枝，理想状况这一数值变成$O(m^{d/2})$。实际用时也是介于两者之间。

2 程序代码说明

2.1 数据结构说明

_Node 类：决策树节点，成员变量：

- 决策的模式
- 决策的内容
- 决策者
- 父节点
- 子节点（列表）
- 棋盘（做出决策后的）
- 选择的子节点（只在确实需要记录的时候不为 None）

_GameTree 类：决策树，成员变量：

- 树根
- 是否为先手
- 复杂模式搜索深度
- 简单模式搜索深度
- 是否为复杂模式（bool）

另外，我们用线性表（python 内置 list）来存储子节点，在 minimax 搜索中使用递归，隐式地使用了栈的结构。

2.2 函数说明

_Node:

- get_child_decision：以列表形式返回子节点的决策
- get_avaiable_pos：返回落子的启发式规则允许的落子位置列表
- point：返回当前局面的估值函数
- __iter__：迭代器，按从叶到根，从左到右的顺序迭代。在竞赛中用不上，调试的时候用过。

_GameTree：

- cut_to_current：在对方决策后己方决策前，将博弈树树根更新为当前局面。己方决策后这一更新在决策的函数中实现。mode 为前一次决策的内容。board 为当前棋盘。
- shift：己方做出决策后，爬树（更新树根）。
- get_depth(self)：根据当前决策模式和内容、先后手情况等返回搜索深度。
- decide：使用 minimax + alpha beta 剪枝做决策。
- modify_depth：根据剩余时间来确定搜索深度和搜索模式。

2.3 程序限制

使用索引返回棋子的赋分，但索引列表长度有限，如果合成出的棋子等级过高，就无法获得索引值。但由于只有 500 回合，棋子面值和最多为 2000，即使一方吃掉了另一方的所有棋子并且合成了一个最大子（这种极端情况显然不可能发生），等级也不会超过 10，因此真实游戏中并不会出问题。

3 实验结果

3.1 测试数据

实验环境说明：

- 硬件配置：Intel® Core™ i5-6200U CPU @ 2.30GHz 2.40 GHz, RAM: 8.00GB
- 操作系统：Ubuntu 18.04.4 LTS
- Python 版本：3.7.6

开发过程中，主要通过本地测试确保程序的正确性，通过本地内战来测试性

能是否改进，通过热身赛和天梯赛来观察算法性能、发现行为的问题。

我们通过在原始代码上增加一些变量来统计某些数据，来定量研究算法的性能。其中包括：

- 测试主动落子可选择位置的平均数。40 局实验给出落子的平均搜索规模大约为 1.6。
- 测试剪枝性能。取消深度浮动机制，统一搜索 4/3 层，将时间限制扩展到 50s，让剪枝的代码和不剪枝的对战 20 回合，记录用时的平均值（单位：s）。

	先手（4 层）	后手（3 层）
剪枝	14.813	3.507
不剪枝	31.578	11.482

- 测试存储局面的重复利用率。我们仍然通过实验来观察：我们给决策树设置两个类变量，一个是 REUSE，一个是 CREATE。每新建一个节点，CREATE 加一；每“爬树”一格，即重复利用一个节点，REUSE 加一。仍然做 40 局实验，观察 REUSE 和 CREATE 的比值，平均约为 1:10，即大约节省出 9%的时间。
- 测试估值函数用时。由于本地未使用加速库，先手全部超时，故只考虑后手的用时统计。和不同的对手做两次 20 局的实验，数据如下（单位：s）：

	平均一次估值的时间	平均一局估值的时间	平均一局的总时间
第一次	1.45e-4	1.29	3.84
第二次	1.82e-4	1.21	4.35

3.2 结果分析

我们小组的战术策略是先完成，再优化。通过天梯对局，研究我们 AI 落败原因再进行算法优化。如图 2 所示，在该局对战中我们并未考虑非相邻合并情况，导致我方（红色）的 1024 被对方截获，导致战局大败。而解决这一问题的办法之一是：我方可以填棋子到空位处，从而避免这一情况的发生。通过战局反馈，我们一直在完善我们的 AI。使得在天梯赛阶段我们小组的积分已在 N19 区稳定在前三，奠定了小组出线的基础。在完成天梯赛后，我们对于总决赛下半战区的战队，在天梯过程中部分战局的胜率进行统计，对于代码间的克制情况进行了解，同时进一步完善了我们的代码。

图 2 某局对战问题反馈图

天梯赛 N-19 下半战区对战胜率表

组名	X-ray	777	666	404	8848	Romeo
X-ray	0	0.60	0.4	0.63	0.9	0.5
777	0.4	0	0.47	0.66	0.6	0.75

下面结合上一节的测试结果稍做分析。

对落子位置选择的启发式规则将平均选择规模降到了 1.6，使用该规则之前的规模约为 8（取 0 到 16 的平均值，这只是一个粗糙的估计，但数量级应该是对的。）这样，对于未剪枝的情况，搜索d层，时间节约到了原来的$(1/5)^{d/2}$，指数除以 2 是考虑到平均只有一半的层是和落子有关的。

剪枝的明显效果可以从上一节对应的数据中明显地看出，不剪枝与剪枝的时间的**对数比**为 1.918（后手）和 1.281（先手），这与我们在 1.3 节中分析的$1 < s < 2$是吻合的。

因为重复利用的比例并不高（接近 10%），存储带来的效率提升和动态规划中的情况相比并不明显，但在实战中也节省了一些时间，使得我们可以更多地使用更深的深度，更晚地进入简单模式。

估值函数的开支约占整体的$1/3$，这是合适的。因为完善的估值函数有利于做正确决策，而这些时间并不能用来增加搜索深度。（见上一节研究剪枝性能的表，先手搜索 4 层和后手搜索 3 层的时间对比提示了增加一层搜索所需的时间。）

3.3 经典战局（可选）

我们的算法在某些情况下有压倒性的优势，能够快速将对方区域占据，从而限制对方移动，进而拉开分数差距。在决赛八进四，对战 8848 战队比赛中，我们约有 90%控制对方移动，其中有一局控制了 229 回合，进而使得我们以 42:8 的成绩取胜。

4 实习过程总结

4.1 分工与合作

由于疫情无法当面沟通，我们小组采取微信为主，配以关键时间节点下的腾讯会议进行事项沟通。具体分工如下：

王逸飞：算法设计与 AI 编写（基于 minimax）

郁桂花：算法设计（基于 minimax）、天梯战局分析

刘鸿骏：算法设计与 AI 编写（基于 minimax）

陈董泽：算法设计与 AI 编写（基于贪心算法）

林翔宇：实验报告撰写

我们利用腾讯会议一共进行过三次全组讨论。第一次 5 月 12 日确定了小组分工；第二次 5 月 16 日对比了基于贪心算法的 AI 和基于 minimax 与 alpha-beta 剪枝的 AI，并决定以性能有显著优势的 minimax 为我组 AI 的主要算法；第三次 5 月 26 日讨论了一些细节并决定使用刘鸿骏的总体结构、估值函数，配合王逸飞的储存树等一些细节修改，以此作为进入正式天梯赛的 AI。会议和微信截图见照片文件。

4.2 经验与教训

本次 2048 游戏，我们小组采取的分工策略是第一周由刘鸿骏与陈董泽两位同学在基线 AI 的基础上实现一些功能改进。由王逸飞和郁桂花两位同学进行算法调研和算法分析。这样的好处在于在较为早期，我们就有了一个至少可以战胜基线 AI 的算法可以进行对战。之后在彼此交流中不断完善算法功能，主要体现在完善评估函数和增加程序考虑范畴。我们采用全员尝试的方式，在第二周决出了组内优秀 AI，王逸飞、刘鸿骏两位同学的代码展现优异的表现，之后他们进一步完善该代码。我们的算法初期未考虑时间、对局长度、边界战略等方面，通

过进行天梯对战，我们积极分析战局情况，发现问题后进行截图示意，利于算法实现同学进行代码改进。如此我们各司其职，平行推动工作进行，提高效率。

同时也有以下可以改进之处，比如对于时间管理上还可以进一步加强。虽然我们进行了时间管理，但是部分战局仍会因为超时问题落败，之后可以进一步加强。另外，在决赛提交的代码中，由于没有仔细检查，估值函数弄错了一个正负号，也对 AI 的战斗力产生了一些负面影响。尽管我们夺取了 N19 组的亚军，但是我们算法本身还有一定的上升空间。

4.3 建议与设想

感觉最后大家用的都是 Mininax 算法和 Alpha-Beta 剪枝，实习作业的设计应该让不同的算法性能相当，这样更加多元灵活，而不是仅仅在比拼启发函数和优化深度。

分年级进行组队能否较早告知，方便大家尽早成组。关于天梯热身赛时，能否先放开组别限定，便于和不同对手对战，继而发现问题完善代码，也利于赛区头部队伍不断优化自己的代码表现。

寄语：大家可以利用线下的机会，一起钻研算法，那样体验会更好。

5 致谢

感谢陈老师和助教们对于学习的指导，感谢技术组同学们设计并保证 2048 游戏的成功进行。感谢天梯平台给了我们测试代码的机会，为我们改进代码提供思路。感谢组长王逸飞有序组织小组讨论及人员分工事宜，使得我们小组每个同学都可以在合适的时间做自己擅长的事情。我们能成为 N19 的亚军，很大程度归功于王逸飞同学和刘鸿骏同学，他们不断改进我们的代码，使我们的表现越来越好。最后感谢整个 N19-777 小组，大家团结一致完成了本次 2048 大作业！

6 参考文献

[1] https://github.com/pkulab409/sessdsa.2048

[2] http://gis4g.pku.edu.cn/course/pythonds/

[3] https://stackoverflow.com/questions/22342854/what-is-the-optimal-algorithm-for-the-game-2048

[4] https://en.wikipedia.org/wiki/Minimax

[5] https://en.wikipedia.org/wiki/Alpha%E2%80%93beta_pruning#:~:text=Alpha%E2%80%93beta%20pruning%20is%20a,%2C%20Go%2C%20etc.).

[6] Chao Ma, Tuning Chess Evaluation Function Using Reinforcement Learning, 2012.

[7] https://stackoverflow.com/questions/16328690/how-do-you-derive-the-time-complexity-of-alpha-beta-pruning

数据结构与算法课程实习作业报告

蔡奕腾*，杨淳皓，齐浩宇，詹钊煌，李哲

摘要　本组代码将传统棋类游戏的决策树思路应用于对战型二〇四八游戏。首先对当前局面递归生成决策树，使用估值函数对决策树叶节点的状况进行打分，使用极小极大算法和深度优先搜索确定行动策略，并使用 α-β 剪枝算法减小搜索时间，以便尽可能增加搜索深度。代码整体策略偏向保守，能够有效合成大棋子和保持己方优势，最终以小组第七的名次出线。

关键字　决策树，极小极大算法，深度优先搜索，α-β 剪枝，递归

1　算法思想 2
1.1　总体思路 2
1.2　算法流程图 4
1.3　算法运行时间复杂度分析 5
2　程序代码说明 6
2.1　数据结构说明 6
2.2　函数说明 6
2.2.1　寻找对方可插入位置：position 方法 6
2.2.2　查找所有合法行动方式：getLegalMoves 方法 6
2.2.3　决策树的构建：populateNodeChildren 方法 7
2.2.4　估值函数：score 方法 8
2.2.5　特殊情况的额外判断：isEaten 和 weWin 方法 12
2.3　程序限制 13
3　实验结果 14
3.1　测试数据 14
3.1.1　本地测试 14
3.1.2　天梯赛测试 14
3.2　结果分析 15
3.3　经典战局（可选） 15
3.3.1　经典战局 1：weWin 的实战应用 15
3.3.2　经典战局 2：吞并对方大棋子 16
4　实习过程总结 18
4.1　分工与合作 18
4.2　经验与教训 20
4.3　建议与设想 21
5　致谢 22
6　参考文献 22

1 算法思想

1.1 总体思路

本组代码经历了六次大的版本迭代，共计产生代码 79 版。算法思路形成过程及重大版本更新如下所述：

对战型二〇四八是完全信息的零和博弈游戏。最初，分析游戏规则后，我们就确定了对当前局面展开决策树，并对每一局面使用估值函数进行打分，选择得分最高移动策略的思路。同时，我们提炼出游戏中需要注意的一些要点：最终判断胜负时由大到小比较，因而大棋子的重要性远高于小棋子；双方棋子可以相互合并而扩大双方差距，因而需要尽量避免己方大棋子被吃；游戏意外结束直接判负，因而需要竭力避免错误输出或运行超时。

我们的第一版代码受搜索思路和算法限制，只能进行两层搜索，这远不足以实现游戏要求，只能通过复杂化估值函数使之达到差强人意的效果。因此，后续剪枝优化显得更为重要。我们通过 α-β 剪枝等方式对极小极大（Minimax）算法进行优化，有效减少了运行时间，搜索深度得以增加，经过多次估值函数的调整（详见 2.2.4 中估值函数部分），代码对战能力得到大幅提升。

但是，在己方与天梯 bot 的数十场对战中，我们发现当前代码存在容易被对方吃子的现象，从而给对方造成很大优势。于是，我们在程序中简单粗暴地加入了防被吃模块，即当一种移动可能被吃大于 4 的子时，这种移动方向无论产生多大价值，都将被舍弃。从理论上讲，这一逻辑并不能保证己方收益最大化，但在实战检验中发挥了良好效果，此时程序已经可以近乎百分百打败天梯 botC。

为了进一步提高战力，我们还针对若干特殊情况增加了判断：当对方不能移动（即被“塞爆”）且同时我方占据相对优势时，只要保证对方仍被“塞爆”即可保持己方的优势，因此我们设置此时不采取任何可能使对方“起死回生”即解除“塞爆”状态的移动方式。在搜索层数方面，我们发现程序在前期表现上佳，但中后期往往易被翻盘，于是我们将层数改为分段函数，前 150 回合只进行 6 层搜索，而之后当剩余时间超过 0.2s 时均使用 10 层搜索（比起之前稳定的 9 层有所提升），而如果时间即将耗尽，则回到 6 层搜索避免超时。

在天梯赛的对战中，我们还注意到许多局面下己方被“塞爆”带来了极大的劣势。对方可以在此期间进行大规模合成，而己方几乎无法作出回应，只能在对方领地下子，进一步为其增加合成材料。因此，在最终版本中，我们在估值函数内增加了对己方移动能力的判断：如果出现己方被“塞爆”而没有合法移动方式的情况，则大幅调低当前局面的得分，从而避免选择会使得己方被“塞爆”的策略。

总体而言，本组代码采用建立决策树并对局面进行估值的策略。在每次决策时，都以当前局面为根节点，采用深度优先的方式递归搜索 getLegalMoves 函数得到的可能走法，递归结束条件为最大回合数限制、达到最大搜索深度（用 depth 函数控制）或当前节点无任何合法操作。搜索过程中，采用极大极小搜索，仅对叶节点局面进行估值，其余节点的估值由其子节点决定，己方节点为 Max 节点（即取子节点中分值最大的节点作为自己的分值，下相

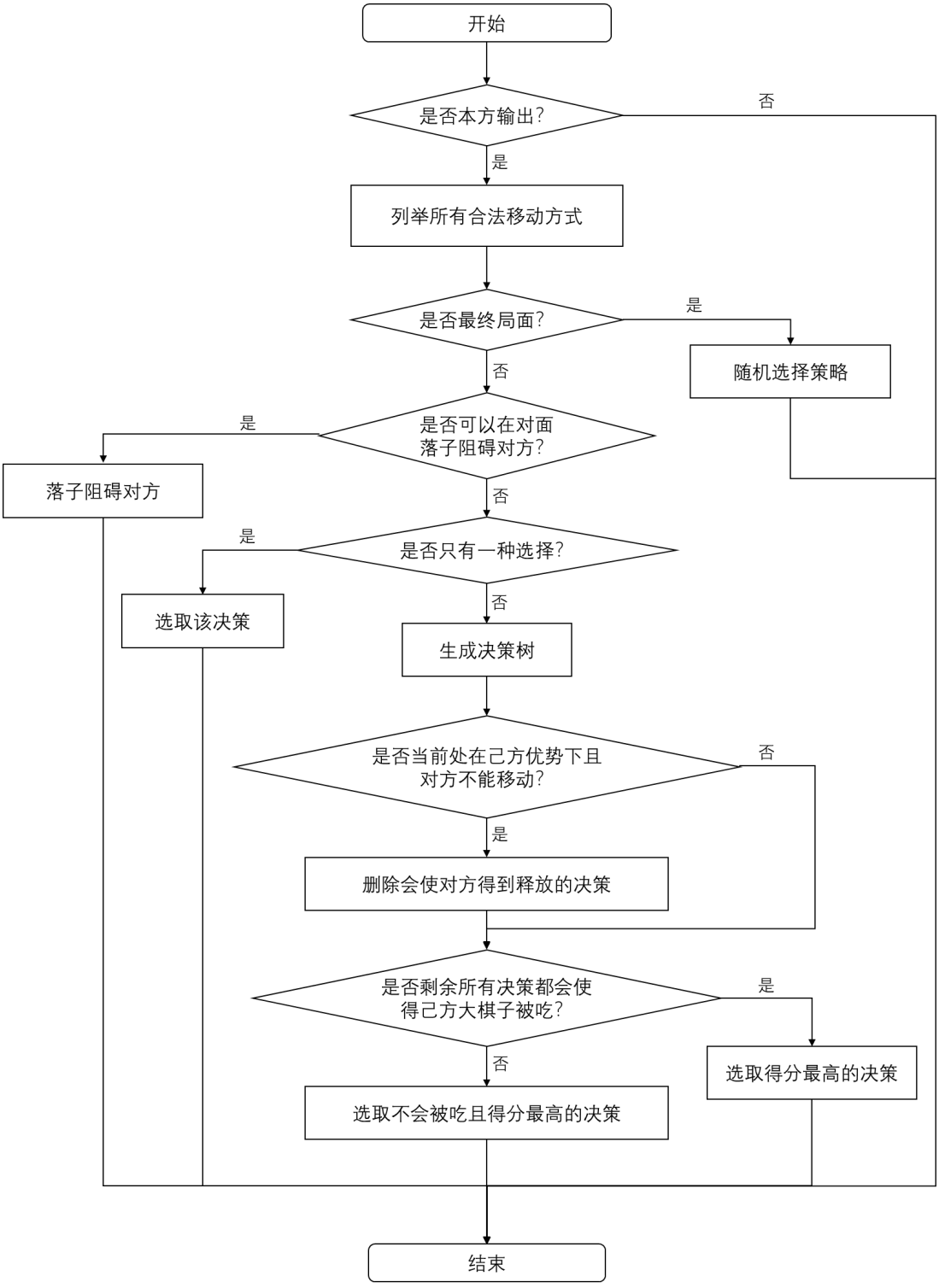
反），对方节点为 Min 节点。采用 α-β 剪枝策略减少搜索。值得说明的是，这里采用的 α-β 剪枝可能并不能做到完全剪枝，因为这里定义的 alpha 和 beta 变量仅在节点而不是整个搜索中的变量。

在开始搜索前，进行了多步判断。首先，若无法操作，直接返回 None。其次，寻找对方区域中适合落子的位置（调用 position 函数），若存在适合落子的位置，则选择落子该处。再次，先调用 getLegalMoves 函数寻找当前局面的合法操作，若仅有一个合法操作，则直接进行该操作而不再搜索。最后，判断是否为最后局面（回合数达到 499，我方后手且轮到进行合并，由于树的建立并不能对该情况进行搜索），是则随机选择一种操作（根据经验对局势影响不大而再判断增加时间成本）。若上述条件均不成立，才进行搜索。

结束搜索后，再进行两次判断。若发现操作后有胜局条件（分数超越对方且对方无合法操作），则除去其他操作。其次，若我方操作后有可能被对方捕获的子，则考虑其他操作；若任何操作都将导致被吃，则仍选择对我方最有利的操作。

具体算法流程图如 1.2 所示。

1.2 算法流程图


开始
是否本方输出?
否
是
列举所有合法移动方式
是否最终局面?
是
随机选择策略
否
是否可以在对面
落子阻碍对方?
是
落子阻碍对方
否
是否只有一种选择?
是
选取该决策
否
生成决策树
是否当前处在己方优势下且
对方不能移动?
否
是
删除会使对方得到释放的决策
是否剩余所有决策都会使
得己方大棋子被吃?
是
选取得分最高的决策
否
选取不会被吃且得分最高的决策
结束

1.3 算法运行时间复杂度分析

在一次 output 方法的完整运行中，首先调用的是 position 方法。本质上，position 方法是对整个棋盘进行遍历，由于棋盘大小在游戏中给定，尽管方法中存在三重 for 循环的嵌套，其时间复杂度仍为 O(1)。对于 getLegalMoves 方法，其优先考虑己方落子位置，通常不必列举对方领地空位，时间复杂度也可视为 O(1)。（即使列举对方领地空位，getNone 方法的实现仍是遍历棋盘，给定棋盘大小下时间复杂度也是 O(1)。）

估值函数 score 中调用了 Chessboard 类的 getScore 方法，而查阅源代码发现，getScore 方法的实现使用了 Python 的 sorted 函数，进一步查阅资料可知 sorted 函数使用的是 Timsort 排序算法，时间复杂度在 O(m)与 O(m logm)之间（m 为场上一方棋子个数）。计分过程使用 for 循环遍历所有棋子，将得分累加，时间复杂度为 O(m)。综合考虑，估值函数的时间复杂度不超过 O(m logm)。

决策树的构建是算法时间消耗的主要部分。在不剪枝的情况下，需要递归展开整个决策树，建立决策树的时间复杂度是 O(n^k)（n 为每步的合法移动数目，k 为搜索深度，记根节点深度为 0，总的节点数目可以看作是 n^0 + n^1 + … + n^k）。此外，还要对叶节点调用估值函数进行打分，全部打分的时间复杂度为 O(m logm * n^k)。采用 α-β 剪枝后，剪枝效率依赖于搜索顺序。不妨认为每层剪枝可以节省一半计算（假设搜索顺序随机），则剪枝后时间复杂度变为 O(m logm * (n/2)^k)。

注意到，算法的时间复杂度指数级地依赖搜索深度，因而搜索深度增加会显著增大运算时间，这也是增加搜索深度的最大障碍。幂函数项的底数即 n/2，分别表征合法行动数目和剪枝效率，在搜索深度较大时会对运算时间产生很大影响。为了控制运行时间，我们在位置模式中设定 getLegalMoves 优先只返回己方落子位置，相当于将 n 限定为 1（此时剪枝对该层节点无贡献），时间复杂度退化为 O(m logm * 2^(k/2))（位置模式平均有 k/2 层，每个节点只有一个子节点，为 1^(k/2)；方向模式有 k/2 层，每个节点有不超过 4 个子节点，认为剪枝可以剪去一半的子节点，为(4/2)^(k/2)），相比完整搜索的时间复杂度 O(m logm * (n/2)^k)有了很大提升，也因此可以进行更大深度的搜索。进行该步简化时尚未进行详尽的时间复杂度分析，但实战效果检验了本部分结论的正确性。

2 程序代码说明

2.1 数据结构说明

算法中采用的主要数据结构是决策树。本代码中决策树的实现结合了嵌套列表法与链表的优点，通过建立 MoveNode 节点类，用列表存储指向所有子节点的链接。为便于调用，其他方法均在 Player 类中实现，不再单独建立树类。

一个 MoveNode 类变量代表决策树的一个节点，存储了当前对战局面的所有属性。其中，player 属性是布尔变量，存储该步行动的玩家信息；move 属性是元组或整数变量，分别对应'position'和'direction'模式，代表该步动作即相对父节点的变化，与 mode 属性保持一致；parent 属性是 MoveNode 变量，指向该节点的父节点；children 属性是 MoveNode 变量构成的列表，包含该节点的子节点；depth 属性是整数变量，记录该节点所处的搜索深度。此外，MoveNode 变量还根据父节点 parent 和该步动作 move 计算出该节点对应的棋盘状态，存储在 board 属性中；alpha 和 beta 属性储存节点分值，用于后续进行 α-β 剪枝。

2.2 函数说明

2.2.1 寻找对方可插入位置：position 方法

在 position 模式下，我们可以选择在己方领地的给定位置放置 2 或在对方领地的任意位置放置 2。前者有利于己方自身合成，而后者如果放置位置适当，可以阻断对方的合成步伐（如将 2 放在对方两个 512 之间的空位置上）。因此，我们定义 Player 类的 position 方法，用于查找对方领地可以插入从而阻碍其进一步合成的位置。

该 position 方法只有一个输入参数 board，board 是一个 Chessboard 对象，包含当前待判断局面的棋盘信息。函数内部的 minValue 决定了可以考虑插入的对方棋子最小级别（如 minValue = 3 时，对方两个 8 棋子之间有空位时会选择插入，而两个 4 之间有空位时不会插入）。

该方法的实现思路较为直接：先将棋盘 board 转化为分行和分列的两套嵌套列表，并按行和按列遍历查找可插入位置。满足以下三个条件即可插入：两个对方棋子大小相同，相同棋子之间没有阻碍，棋子大小超过设定的最小值。函数的输出是可插入的位置（元组类型），当没有可插入位置时，函数返回 False。

2.2.2 查找所有合法行动方式：getLegalMoves 方法

根据游戏规则，对战中产生非法输出会直接报错并判负，因此行动策略只能在合法方式中选择。列举所有合法行动方式也是构建决策树的基础。因此，在 Player 类下我们定义 getLegalMoves 方法，用于查找当前局面、当前模式下所有合法的行动方式，这也是代码中调用频率最高的函数之一。

为了扩大 getLegalMoves 方法的适用范围，我们设置了四个输入参数：currentRound 标记当前回合数，用于根据回合数和随机序列判断己方棋子的放置位置；board 是 Chessboard 对象，记录当前棋盘状态；mode 标记当前行动模式，在落子模式'position'和方向模式'direction'之间选择；player 标记当前行动的玩家。通过 getLegalMoves 得到的操作绝对合法，但受运行时间等限制，在搜索时优先考虑双方都落子在自己区域的情形，只有自己区域不能落子时再考虑对方领地全部可落子位置。

该 getLegalMoves 方法首先按模式进行分类处理：'position'模式下，根据随机序列得出己方合法下棋位置，如果己方无处落子，则考虑对方领地所有空位置，合法落子位置用元组表示；'direction'模式下，借用 Chessboard 类的 move 方法判断向各方向移动后棋盘是否发生变化，发生变化的即为合法移动方向，用整数表示。该方法最终返回一个由所有合法行动方式构成的列表。

2.2.3 决策树的构建：populateNodeChildren 方法

构建决策树是进行策略选择的根本步骤。我们通过递归调用 populateNodeChildren 方法，以当前局面作为根节点，生成整个决策树。单独 populateNodeChildren 方法的功能可以理解为生成给定节点的各级子节点，直至设定深度。

该方法的输入参数有当前回合数 currentRound，待生成子节点的根节点 node 和设定的搜索深度 depth。递归的基本结束条件为到达设定的搜索深度，或搜索到游戏结束（已完成 500 回合）。决策树的生成使用极小极大算法，即假设双方都足够理性，总采取最优策略，我方总选择得分最高的操作，而对方总选择使我方得分最低的操作；考虑双方交替进行的几个回合，进而从可能的策略中选出较好的策略。

在 output 中调用该方法时，先以当前局面作为根节点，调用 getLegalMoves 方法得到下一步所有可能的行动策略。然后进行深度优先搜索，依次生成各行动策略的各级子节点。最终到达设定搜索深度或预期游戏结束时，调用估值函数计算并返回叶子节点的得分。根据极小极大算法，对方操作节点即 MIN 节点的得分是其子节点得分的最小值，己方操作节点即 MAX 节点的得分是其子节点得分的最大值，递归出栈过程中按此规则依次向上更新节点得分。当最后一个节点更新完成时，决策树构建完毕，函数执行结束。

为了减少不必要的计算资源消耗，在限定时间内尽可能增加搜索深度，我们采取了 α-β 剪枝对决策树的生成过程进行优化，在搜索中同时记录每个节点得分的上下界，分别存储在 MoveNode 对象的 beta 和 alpha 属性中。以我方操作的 MAX 节点为例，该节点的得分由其子节点中的最高得分确定，而其父节点（MIN 节点）的得分是该 MAX 节点及其兄弟节点中的最小值。如果在深度优先搜索中，发现该 MAX 节点某个子节点的得分已经高于该 MAX 节点已经计算完成的兄弟节点中的最低得分，即说明该 MAX 节点在其兄弟节点中必然不是得分最小值，因而基于极小极大算法，对方在 MIN 节点操作时根本不会考虑当前正在计算的 MAX 节点，我们也就没有继续计算的必要，可以直接结束对该子节点的计算。此过程为 β 剪枝。对于对方操作的 MIN 节点，可以进行 α 剪枝，原理与之类似。这一 α-β 剪枝过程在递归调用 populateNodeChildren 方法时通过及时更新记录已搜索完成节点的分值而实现，有效减少了不必要的搜索次数。

在生成子节点的过程中，我们还调用了一些辅助方法，包括 getNewMode 方法，getNewRound 方法和 depth 方法。与命名相同，getNewMode 方法根据上一步的模式返回下一步行动模式，与当前玩家是否先手有关（如先手方'position'模式的下一步是后手方'position'模式）；getNewRound 返回下一步的回合数，后手方在'direction'模式下完成一次合并后，进入下一回合；depth 方法设定为分段函数，返回由剩余思考时间和当前游戏进行阶段决定的搜索深度，150 回合后且剩余思考时间大于 0.2s 时进行 10 层搜索，追求更好的表现，而其他情况下只进行 6 层搜索，防止超时判负。

2.2.4 估值函数：score 方法

决策树叶子节点的打分由估值函数进行。我们设计的估值函数需要传入一个 Chessboard 对象 board，即给定一种棋盘状态，对该局面进行打分；同时，当前回合数等因素也会纳入估值函数的考虑范围。随着整体算法的演进与完善，估值函数也经过了多次修改。几个主要版本的估值函数改动思路如下：

2.2.4.1 估值函数 V1.0

第一版估值函数以主程序 V1.0.0 为基础。对双方的场面按照单子战力、棋子地理位置（倍率，用字典储存）以及相邻两子（可合并或被吃）之间的额外加分来估值。

【基础估值方法】

单子战力：（500 回合，每回合新增棋子为双方各一个 2，棋子总数值最多 2000）

棋子	战力
2	1
4	5
8	25
16	125
32	625
64	5625
128	28125
256	140625
512	703125
1024	3515625

地理位置：

本版本的策略着重考虑中间界限的压制（可以压制对方合并，占据节奏上风），因此对于不同格子位置的棋子赋予不同的权重倍率：

×1.0	×1.0	×1.1	×1.2	×1.2	×1.1	×1.0	×1.0
×1.0	×1.0	×1.1	×1.2	×1.2	×1.1	×1.0	×1.0
×1.0	×1.0	×1.1	×1.2	×1.2	×1.1	×1.0	×1.0
×1.0	×1.0	×1.1	×1.2	×1.2	×1.1	×1.0	×1.0

棋子之间相互作用：

此项主要对未来棋子的战力进行估值，采用 Cobb-Douglas 函数形式来计算。

对于任意两个相邻棋子 A, B，记其数值分别为 a, b，对应战力为 m, n。则相互作用函数定义为：

$$f(a,b,m,n) = m^{\frac{b}{a+b}} \times n^{\frac{a}{a+b}}$$

例如，当 16 与 4 相邻时，$f = 35^{1/5} \times 3^{4/5} = 4.90355$；当 8 与 8 相邻时，$f = 10^{1/2} \times 10^{1/2} = 10$。计算时，按照每行遍历再每列遍历求和，遇到空格则继续搜索直到遇到对方棋子或边界。

【落子步骤估值方法】

每步落子有两种选择：①在己方规定位置落子；②在对方任意位置送子。对于两种选择，估值方法一致，比较差值 F 即可。

f = 己方单子战力之和（含地理位置权重）+ 己方棋子互作大小

g = 对方单子战力之和（含地理位置权重）+ 对方棋子互作大小

F = f - g

选择在对方送子当且仅当落在对方减小对方的互作量大于落在己方规定位置增加己方的互作量。

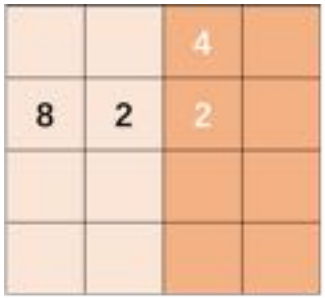
【合并步骤估值方法】

基于已有的场面，双方均可以上下左右四个方向移动。对于每个移动后分别估值，考虑如下一些因素：①是否发生合并导致战力显著增加；②地理位置改动产生的效应；③吃子导致一方战力大涨，另一方战力大跌；④下一次另一方移动是否有直接的被吃导致的战力大幅改变。

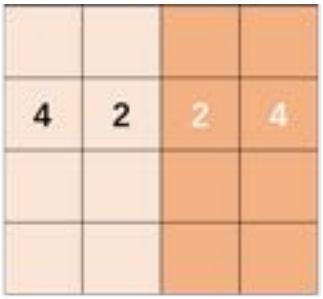
己方移动后：f = 己方单子战力之和（含地理位置权重）+ 己方棋子互作大小-（if exists）对方下一次合并吃掉己方棋子得到的棋子战力 ×3

*吃子的影响考虑在了战力变化中（占据对方格子有额外高权重 + 己方盘面棋子之和变大战力提高）

*最后一项的存在性：①无吃子可能则不存在；②吃子后的位置周围不存在与之相等的对方棋子不存在；③吃子后对方可以反吃，但反吃后己方拥有与反吃后相等棋子再反吃时不存在。如下图所示：



不存在最后一项



存在最后一项

g = 对方单子战力之和（含地理位置权重）+对方棋子互作大小

F = f - g

对方移动后：f = 己方单子战力之和（含地理位置权重）+ 己方棋子互作大小

g = 对方单子战力之和（含地理位置权重）+ 对方棋子互作大小-（if exists）己方下一次合并吃掉对方棋子得到的棋子战力 × 3

F = f - g

【缺陷】

此版本由于当时树的搜索深度不足，需要大量参数人工代替树的功能，不仅不够准确，参数数量也较多

2.2.4.2 估值函数 V2.0

在主程序更新，加入剪枝策略后，树的深度提升，这使得我们可以大幅简化估值函数。同时，注意到对于场面的判断完全依赖树，因此我们将棋子的战力差进一步放大。

单子战力：

棋子	战力
2	1
4	6
8	40
16	250
32	1500
64	10000
128	60000
256	400000
512	2500000
1024	20000000

在对更新棋子战力参数后的程序进行测试时，我们发现，过于简单的参数设计使得程序出现了两个问题：（1）吃子过于激进，容易被反吃（由于吃子可以大幅提升战力，但此时树的深度不足以考虑到再接下来被反吃，即需要 8 层左右）；（2）合成效率不高，棋子分散在棋盘的各个角落，大子小子的犬牙差互使得合并出现困难，经常只能合成 128 就不能移动。

基于这些情况，我们参考了 2048 单机版（https://github.com/ovolve/2048-AI）中 AI 的估值思路，将单调性、平滑性、空闲方块、最大子及棋子总战力均纳入考虑。以下是对这些指标的具体阐释：

- **初始 score1=100 分。**
- **单调性：**

判断当前最高数字所在区域（左上、左下、右上、右下），规定从这个区域往其它两个方向的方块应单调递减。之后遍历各个方向（每行每列），遇到不符合单调性的方块便减分，

如此得出单调性的分数。

单调性的减分公式为：score1 -= a * (两数差值 / 当前所有方块数字的平均数)

• 平滑性：

遍历各个相邻方块，数值相同便加分。

这个加分也并不固定，而是这两个相等的方块数值越大，加分越多。

平滑性的加分公式为：score1+=b*两个相同的数的数值

• 空闲方块：

空格数的加分公式为：score1+=c*log(己方拥有的空格数)

• 最大子：

添加最大的几个子的影响，考虑地理因素：

×2.0	×1.8	×1.6	×1.4	×0.8	×0.5	×0.0	×0.0
×1.8	×1.0	×1.0	×1.0	×0.8	×0.5	×0.0	×0.0
×1.8	×1.0	×1.0	×1.0	×0.8	×0.5	×0.0	×0.0
×2.0	×1.8	×1.6	×1.4	×0.8	×0.5	×0.0	×0.0

大子加分公式：score1 += d*(maxvalue*pos + secondmaxvalue*pos + thirdmaxvalue*pos)

• 棋子总战力加分公式：score1 += e * Σvalue

2.2.4.3 估值函数 V3.0

估值函数 V2.0 中得到的 score1（对手使用 score2）有 5 个变量，也即具有 4 个自由度。将其中一个变量设为 1，调节其他四个参量的值，得到的结果仍然不能稳定击败 botC。经过几十场对战的复盘，我们在主程序中加入防被吃模块（由 isEaten 判断），拒绝任何可能导致大于 4 的子被吃的策略，使得程序可以几乎完胜天梯 BotC。

但是，此时程序对 BotB 仍然难求一胜。在调参并不顺利的情况下，我们开始思考单机版的策略是否适合本次的对抗版。我们发现，追求平滑性、单调性使得很多吃子的良机以及占据格子的进攻性不复存在；同时，搜索单调性需要遍历双方棋盘，每步也要付出巨大的算力来估值，这是得不偿失的。因此，这个思考让我们再次选择简化估值函数，仅保留空闲方块、棋子总战力以及新加入的占据格子数。同时，这些参数均以分数形式、根据轮次增加而不断增加的形式表达：

空格数的加分公式为：score1+=a*round*己方拥有的空格数/32

占据格子数的加分公式为：score1+=b*round*己方具有所有权的格子/32

总战力的加分公式为：score1+=c*round*（我方棋子和）/（我方棋子和+对方棋子和）

之后，我们进一步优化，将 round 乘上不同的幂次，实现增长速度的区分。经历此次重大优化，我们的程序可以稳定搜索 9 层，与 botB 也可以实现五五开。

2.2.4.4 估值函数 V4.0

在 V3.0 的估值函数版本缩减了参数个数之后，调参变得相当简单，然而结果一直无法有进一步提升。这让我们意识到一些函数的设计还不合理。因此，我们又进行了如下改进：

首先，将空闲方块的参数剔除，原因是我们没有发现其数值的改变对结果有任何非偶然误差的影响（正是这个删除使得天梯赛面对采取给对手塞棋子的 Victor 队时被克制，出现了 Victor 在前八只能赢我们，而我们可以和 5~8 名的队伍不相上下，却让第九的 Victor 屡次大比分战胜我们的情况）；其次，将占据格子的函数不再使用线性函数，而是参考 S 型增长，在双方棋子对等时吃子的边际收益最大，而之后边际收益递减，即：

score1 += b * round / (1 + math.exp(1 - 2 * 占据格子数 / 32))

另外，对棋子的分组也增大区分度，采取指数形式。每个子根据 i=1~10 的等级加分：

score1 += 3.8 ** i

而最大子、次大子额外加分：

score1 += 3.5 ** maxValue

score1 += 3.8 ** secondMaxValue

此时的估值函数与主程序中 weWin 的判断和搜索深度 depth 的分段设计相结合，已经可以稳定小胜 botS。此估值函数一直应用到天梯赛结束和代码出线。

在考虑到被对手塞爆的影响后，在最终提交参加八进四比赛的代码中，我们又增加了己方不能移动的判断，对己方可能被塞爆的策略大幅降分，减少被塞爆的可能性。

2.2.5 特殊情况的额外判断：isEaten 和 weWin 方法

在代码中，我们对两种情况作出了额外判断：使用 isEaten 方法判断做出某决策后是否会导致己方棋子被吃，使用 weWin 方法判断此时场上棋子我方是否具有相对优势。

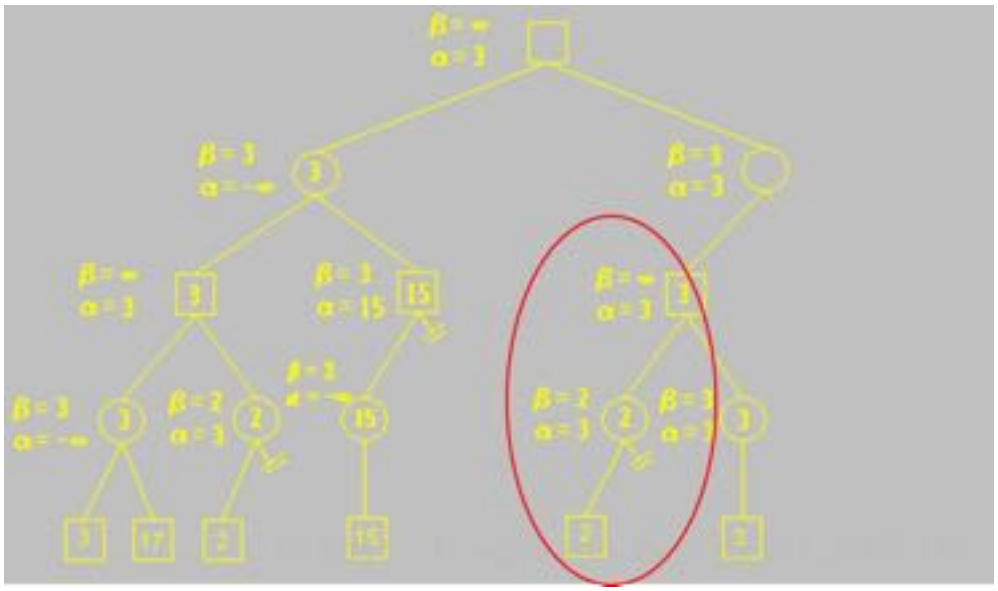
方法 isEaten 需要两个输入参数，分别是刚做完决策的一方 player 和当前棋盘状态 board；方法内的 minValue 表示需要防范被吃棋子的最小级别（minValue 取 2 时意味着防止大于 4 的棋子被吃）。与 position 方法相似，isEaten 也采取分别按行和列制作嵌套列表并遍历判断的方法。当大于最小级别的己方棋子与对方同级棋子相邻或中间为空时，isEaten 方法返回 True，其他情况返回 False。在 output 中，会按得分从高到低对拟选择的移动方式进行 isEaten 判断，如果判断可能被吃，就舍弃该策略而考虑得分次高的行动方式。

方法 weWin 按照胜负判定规则对当前棋盘 board 进行判断，按数值由高到低的顺序比较双方棋子个数。当己方棋子按此定义占据优势（即如果此时结束比赛，则己方获胜）时，weWin 返回 True，否则返回 False。在 output 中，会在选取决策前调用此方法：如果此时对方不能移动（被“塞爆”）且 weWin 返回 True，那么所有可能重新赋予对方活动空间的决策都会从待选策略中删除，因为在此情况下，只要保持对方一直不能移动，我方将可以自然获胜。

2.3 程序限制

经过多次修改完善，最终版本的程序相当稳定。由于 output 方法输出的操作仅在 getLegalMoves 给出的操作列表中选择，程序必然不会产生非法输出。在运行时间方面，我们通过不完全的 α-β 剪枝相当程度上提高了运算效率；在多次天梯测试后，我们寻求了搜索深度与运行时间的平衡，从而尽可能避免超时。

但是，从算法代码中也可以看出，α-β 剪枝的效率严重受到搜索次序的影响，而且本组代码的 α-β 剪枝并不完全，因此在罕见的极端情况下（如搜索次序恰好不能应用剪枝时）仍有超时的可能。现版本的剪枝局限于当前路径，剪枝时为逐层剪枝；而正版的 α-β 剪枝将 alpha 和 beta 作为全局变量，可以跨子树控制剪枝，相比于本版本的剪枝理论上可以减去更多搜索。因此，超时的可能性可以通过采用正版 α-β 剪枝算法进一步降低。



正版 α-β 剪枝算法如上图所示，alpha 和 beta 作为全局变量，在圆圈部分会对 alpha 和 beta 值进行传递。但在本代码采用的剪枝算法中，圆圈部分的 alpha 值依赖于叶节点的搜索（也就是 2），圆圈内的剪枝不会发生，因此剪枝效率有所降低。但是，使用正版的 α-β 剪枝后在接近的回合数下，采用正版剪枝的时间约为现版本剪枝的一半，但操作效果（胜率）远小于现版本。出现此现象的原因可能为估值函数与搜索不够契合，需要更大的搜索量支持决策。但受时间和精力限制，最终采取了不完全的 α-β 剪枝策略。

3 实验结果

3.1 测试数据

实验环境说明：

- 硬件配置：（CPU/内存）AMD Ryzen 7 4800H / 16 GB RAM
- 操作系统：（名称/版本）Windows 10 家庭中文版 1909
- Python 版本：（版本号）Python 3.7.7 (bundled with Thonny)

3.1.1 本地测试

本地测试的目的在于确定 AI 的内存占用没有超过规定的最大值。由于本地与服务器硬件设施、运算能力、算力分配策略等都不相同，本地运行的时间没有参考价值。

使用技术组提供的本地可视化调试工具 userinterface.py 进行测试，使用本组最终版本 AI 与默认随机 AI 进行对战，双方先后手各 10 轮，最高限制 500 回合，超时总时间设置为 20s（超时与否由天梯结果判断，本地相当于不做限制）。测试结果如下：

运行中 Python 程序的 CPU 占用率稳定在 9.0% – 9.7%之间，内存占用在 33 MB – 37 MB 之间，进行 20 轮比赛耗时约 206 s。对战随机 AI 的胜率为 90%（单次测试）。

3.1.2 天梯赛测试

由于天梯热身赛中代码不断更新，天梯累计的胜负数据说服力较弱。在本次报告撰写中，我们选取了几个主要版本与一个我们选定的“官方 Bot 组合”进行对战（12 场对战 BotD，12 场对战 BotC，12 场对战 BotB，6 场对战 BotA，6 场对战 BotS），统计结果如下：

己方版本	总场数	胜	负	平	胜率
V0.7.2	48	3	45	0	6.2%
V2.5.5	48	8	40	0	16.7%
V5.4.2	48	38	10	0	79.2%
V6.3.0	48	38	10	0	79.2%

版本 V0.7.2 到 V2.5.5 修改了搜索树和防被吃策略，针对 BotD 和 BotC 的胜率有所提升（如果只考虑对 BotD 和 BotC 的 24 场对战，两个版本胜率分别为 12.5%和 33.3%）；版本 V2.5.5 到 V5.4.2 的过程中经过了有效的剪枝优化，搜索深度增加近一倍，提升效果十分明显；V6.3.0 是提交的最终代码，是基于 V5.4.2 的小幅优化版本，可以看出二者水平相近，这也标志了我们在有限时间内对策略进行优化的极限。

上传到官方账号的最终代码在天梯赛 4050 战中获得了 2109 胜 1940 负 1 平的成绩，综合胜率 52.1%，最终等级得分 1388.12，以小组第七的名次出线。

3.2 结果分析

在天梯赛实战中，我们的算法运行稳定，几乎没有出现因为报错或超时而判负的情况。这得益于算法通过 getLegalMoves 方法严格给出合法输出，通过 depth 方法分段调节搜索深度，以及使用合理的剪枝和简化策略减少算力消耗。算法的主要时间开销在决策树的构建过程，我们的剪枝算法虽不完全，但也能够满足实战要求。在时间控制和算法稳定性方面，我们的代码达到了预期效果。

在比赛中，我们添加的一些额外判断也起到了显著的正面作用。在测试过程中，我们加入 isEaten 方法用于防备所有大于 4 的棋子被吃，这一看似极端保守的策略保证了我方大棋子的安全，也在测试中从不稳定击败 BotC 提升到几乎完胜 BotC。再如，我们设定 weWin 方法，判断我方具有相对优势且对方被困时，采用保持此局面的消耗策略，尽可能耗到双方都无法移动而游戏结束。该策略在实战中取得了突出效果，具体示例可见 3.3.1 经典战局 1 的分析。我们算法的总体胜率也证明了估值函数设计的合理性。

值得注意的是，在正式天梯赛中，我们发现了较多的相互克制现象。如我们的代码明显胜过胜过 Romeo 组和 Delta 组，而 Romeo 组和 Delta 组通常可以战胜 Victor 组，但 Victor 组又对我们的代码表现出了明显的优势。这使得天梯赛中我们的名次在 7 至 9 名之间上下振荡。我们推测，这一现象的发生与不同代码的策略风格（本质上是启发式规则）有关。

具体分析，部分代码的策略是优先在对方领地放置棋子，将其塞爆后开始不受干扰地进行自身合成，这相当于他们采用的启发式规则。但是，我们在构建决策树时优先考虑的是双方都在自己的领地放置棋子的情况，这使得对方的大部分行动都在我们的意料之外，我们的决策树不涵盖这些情况，自然也不能起到应有的效果。如果将全部策略都纳入考虑，算法的耗时又会大幅增加，搜索深度只能被迫减少。因此，我们的启发式规则在大部分情况下能起到良好效果，而对战另一些风格的代码时，启发式规则又成为了我们的天然劣势，

3.3 经典战局（可选）

3.3.1 经典战局 1：weWin 的实战应用

本对局来自 N19 八进四比赛中与 777 组的对战。对局先手方为对手 777 组，后手方为我方 8848 组。在算法总体思路和流程图中，我们提到，当对方不能移动且我方占据相对优势时（这一状态的判断由 weWin 方法实现），我们设定代码不会采取任何可能使对方“起死回生”的移动方式，从而保持我方的相对优势。本对局充分体现了这一策略在实战中的应用价值。（对局地址：http://gis4g.pku.edu.cn/ai_arena/results/4/N19%208%204/N-777%20vs%20N-8848/41469545638033864.txt/，完整对局也可在报告所附 GIF 图片中观看。）

如图，在第 438 回合，对手 777 组落子于(0, 0)处，此后 777 组没有合法移动方式，且此时我方占据相对优势（双方各 1 个 2048，我方有 1 个 1024，对方无 1024），只要保持当前局面稳定，我方必然能够获胜。因此，在后面的几个回合中，我方不进行任何可能给对方活动空间的操作，即在轮到对方合并操作前一步时保证不留给对方空位。最终，对方从第 438 回合起都没有获得合并机会，对局在第 463 回合因为双方都无法行动而结束。可以看到，对方从自己无法移动开始，其棋子排布未能产生任何变化。

3.3.2 经典战局 2：吞并对方大棋子

本对局来自天梯赛中与 Bravo 组的对战。对局先手方是对手 Bravo 组，后手方是我方 8848 组。本对局演示了在几乎被对方“塞爆”的情况下，利用对方失误吃掉对方棋子而占据绝对优势的过程。（对局地址：http://gis4g.pku.edu.cn/ai_arena/match/c6gpHSZt4R/1/，完整对局也可在报告所附 GIF 图片中观看。）

如图所示，第 380 回合对手向右合并，两个 256 棋子合成出 512，此时我方领地仅余 3 列，棋盘大部分区域都被对方占据。下一步我方向左移动，吞并对方 512 棋子并合成 1024。此时我方新合成的 1024 与对方 1024 相邻，且依靠旁边 2048 的存在，对方下一步不能向上移动吃掉我方的 1024，否则等于自投虎口。

此后的行动中，我方向下进一步吃掉对方的 1024；后续又找准机会，在第 385 回合将两个 2048 调至相邻并合成出 4096。至此，我方优势已经极为明显，对方回天乏力，最终对方最大棋子止步于 1024。

4 实习过程总结

4.1 分工与合作

小组主要通过微信群进行交流。开发过程中没有安排特定的组会时间，组员之间在微信群随时进行线上交流和代码共享，保障思路的连续性和合作的即时性。因为讨论过程在时间上连续，从 5 月 15 日至 6 月 2 日每一天都有线上讨论，微信聊天记录可以直接起到组会记录的作用。

在大作业完成过程中，组内各位成员灵活分工、默契协作，取得了良好的合作效果。各组员主要工作情况如下（按微信群成员名单排序）：

蔡奕腾：负责 MoveNode 的定义和初期代码编写调试，以及后续对接工作。

杨淳皓：负责所有估值函数的公式建立、对局思路设计、天梯调试代码并根据对局复盘优化策略；提供了实习报告中估值函数演进部分的材料。

齐浩宇：负责编写 isEaten 函数、weWin 函数、position 函数，研究制定策略，参与估值函数 score 的编写。

詹钊煌：负责初期代码 alpha-beta 剪枝部分、populateNodeChildren 函数辅助函数的编写及整合代码，后期负责决策树部分代码的改进；提供了实习报告中总体思路和数据结构说明的部分材料。

李哲：负责初期 getLegalMoves 和 populateNodeChildren 函数的简陋定义，负责实习报告的撰写。

14:42
聊天信息(5)
群聊名称
数算大作业8848钛金组
群二维码
群公告
备注
查找聊天记录
消息免打扰
置顶聊天
保存到通讯录

14:51
查找聊天记录
日 一 二 三 四 五 六
2020年5月
2020年6月
今天

4.2 经验与教训

启发式规则和剪枝算法的作用是本次实习项目最直观的经验。算法开发之初，我们受困于有限时间内难以达到令人满意的搜索深度，为此，我们采取了优先搜索双方都落子在自己区域情形的启发式规则（最初天梯上表现较好的代码大多落子在己方领地），极大地缩减了搜索规模，有效增加了搜索深度。启发式规则本质上起到了剪枝的作用，α-β 剪枝策略的应用也进一步节约了搜索时间。在这种棋类游戏中，寻求更好的剪枝策略以增加搜索深度和开发更有效的估值函数以进行更加准确的局面评估应该是算法优化的两个主要方向。

重视实战检验是我们本次实习项目总结出的另一条重要经验。许多理论上可行的操作不一定适用于实战，理论上有意义的改进必须达到提高实战能力的效果。例如，我们曾考虑过储存决策树第三层节点的方式，通过记忆化搜索节约时间；但在实践中我们发现时间反而延长，这可能是因为储存和比较局面的时间成本较高，且搜索中比较局限（双方都仅考虑落子在己方空间），故储存意义不大，因而最终没有采用这种改进方式。

另外，善于利用天梯对战平台，经常进行天梯对局的复盘和总结，对提高算法战力具有很大帮助。我们后期在实践中发挥了良好效果的 isEaten 和 weWin 两个额外判断模块，都是在总结我方天梯表现的基础上提出的。后期因为时间限制，没有开发更完善的防塞爆策略，否则对战能力应该有进一步提升。

代码的数据结构还可以进一步改进。我们代码的主要操作都直接写在 Player 类中，但许多方法实际上依赖于当前回合数、行动玩家和模式，导致这些方法之间需要频繁传递参数，对定义和调用方法都造成了一定不便。事实上，许多依赖于局面的方法可以写入 MoveNode 节点类，currentRound 等常用参数也可以作为 MoveNode 的属性进行记录。如果在代码写作前就对需要使用的函数和方法有全盘的考虑，通过数据结构的合理设置可以进一步简化代码，提高可读性和可维护性。

在代码规范性与版本管理方面，我们的工作也暴露了许多问题。首先，由于代码由全组成员共同协作完成，代码接口、注释风格等都不甚统一，改动代码时还需要注意适配问题，甚至可能需要重写别人的部分。此外，我们使用微信群传递代码，代码文件的命名虽然采用了版本号命名方式，但没有事先对更新的类型进行统一定义，对何时何种更新应该修改主版本号还是次版本号没有明确规范。协作过程中出现了一些版本混乱的问题，给代码整理和报告撰写带来了一定困难：如代码小幅修改后没有修改版本号，使得同一版本号对应不同代码文件；部分代码未按规范命名，使用“新”“最”等只有相对意义而缺乏时间特征的命名方式；有的代码基于旧版本修改，命名时导致了“旧版本”反而新于“新版本”的问题。

代码全部由一人完成是统一代码规范的最好方法，但是期末季时间紧张，代码只由一人写作并不现实。因此我们认为，在类似的工作中，算法实现还是应该多人协作完成，但可以由专人负责管理和维护代码，包括统一函数接口、优化代码结构等操作。事先熟悉 GitHub 等代码托管平台的使用也有助于提高团队协作效率。此外，在代码头部注释中撰写更新日志十分必要，这样可以迅速比较不同版本的差异，对比战力提升的关键因素，有助于进行算法的总结和改进。

4.3 建议与设想

大作业的总体安排比较合理，通过分 F19 和 N19 两组大致区分了基础不同的同学（虽然并不完全，或许还应该考虑学院分布）。为了让同学们对大作业有更充分的准备，我们建议在大作业开工之前向同学们简单介绍 GitHub、码云等代码托管平台的使用，这也是同学们将来可以用到的基本技巧。此外，我们希望更早发布大作业报告模板，报告撰写可以与算法开发同时进行，一方面便于记录开发过程和思路的转变，另一方面也有利于关键材料的及时存档。

在课程安排上，本学期的时间安排是周四更新慕课内容，周五上午有课时进行答疑和总结，周二下午讲解课程和作业，作业截止时间一般定为周三。由于周五的答疑经常涉及新放出的慕课内容，这样的时间安排对周四课较多的同学不太友好。而在作业提交截止前进行作业解析，也容易让人无所适从（已经写完的作业是否根据课堂内容修改？还没有写完作业是否先听解析？）。因此，我们建议周五或周六更新慕课和布置作业，给同学们一个周末的时间进行学习，并在周二课上讲解慕课（只讲解慕课内容和答疑），作业由同学独立思考完成，并定在周四截止提交。作业频率大约可以按照两周一次分布，利用周五上午的课讲解作业，此时作业已经提交，同时同学们对作业内容还有比较深刻的印象，有利于锻炼独立解决问题的能力。

关于我们实习作业的后续工作，我们设想可以改进算法以解决 3.2 中提到的“相互克制”现象。出现相互克制情况的本质是各方的代码思路和启发式规则互不相同，对某一类型的 AI 有优势的算法可能同时意味着对另一类型的劣势。据此，我们设想构建一个可以灵活切换多种风格的 AI，其采用的作战思路由对方在前几十回合表现出的风格倾向确定，即俗称的“看人下菜碟”。从理论上讲，只要存储几套采取不同启发式规则和作战思路的算法，以及不同算法风格之间的相互克制关系，就有希望解决 A > B > C > A 的闭环现象。但可以预见，如何准确判断对方的作战风格会成为算法设计中较大的困难，需要进一步详细的讨论。后续可以通过查阅资料寻找是否已经有人或团队实现了类似的思路。

“卷”是数算课程群里最常提及的关键词，也是本学期数算课的著名特点之一。在课程之初，确实很容易产生“他们怎么什么都会”“为什么我一天做完的作业他们只要一小时”的想法；甚至有些同学因此陷入自我怀疑，最终选择了中期退课。但是，我们想提醒后续选课的学弟学妹们，“卷”只是这门课客观存在的现象，远远不是课程的全部特点。必须承认，选修数算课的同学在基础上存在着很大差异，这些差异不是短时间可以弥补的；但是，数算的课程设置保证了每位认真学习的同学都能循序渐进地掌握相关知识，作业和测验的要求也尽可能降低了基础差异对分数评价的影响。换言之，只要在这门课程上付出合理的努力，总能获得对等的收获。要相信，我们对“大佬”的许多印象实际上受到了幸存者偏差的影响，课上更多的还是与我们相同起点的同学。在数算课上，最重要的是放平心态，在自己的能力范围内做好该做的事情。

5 致谢

感谢陈斌老师一学期的指导和帮助。陈老师风趣幽默的授课风格和丰富的课程资源让我们对数据结构与算法有了更加深入的理解，精心设计的作业和实习项目也有效提升了我们的实践应用能力，让我们受益匪浅。

感谢为大作业项目的顺利进行而默默付出的助教和技术组同学。他们搭建的对战平台和详细的说明文档为我们算法的改进提供了很大帮助，没有他们的幕后工作，我们的实习项目也不可能进行得如此顺利。

感谢在天梯平台上开源的小组，他们的思路为我们算法的改进提供了一定的启示和借鉴。也感谢在 GitHub 上开源的前辈们，他们的代码设计为我们的学习过程提供了有益的参考。

6 参考文献

[1] 单机版 2048 AI (用于参考估值函数的改进) - https://github.com/ovolve/2048-AI

[2] 命令行国际象棋 (用于参考决策树的建立) - https://github.com/marcusbuffett/command-line-chess

[3] alpha-beta 剪枝算法及实践 (用于参考 α-β 剪枝算法的实现) - https://blog.csdn.net/qq_31615919/article/details/79681063

数据结构与算法课程实习作业报告

李春江 林熠伟 欧阳筱羿 吴昊京 赵鑫淼*

摘要：本二〇四八 AI 算法主体上采用决策树，选定搜索深度，利用设计好的估值函数计算叶结点的评估值，在深度优先搜索的框架下通过 Alpha-Beta 剪枝加速 minimax 逐层向上的搜索过程，最后在根节点选择评估值最大的分支作为行动策略。我们在测试数据集和热身赛中均有较好的战绩，但在排名赛中出于未知原因时常超时而被淘汰。

关键字：决策树，minimax 算法，Alpha-Beta 剪枝，估值函数

1 算法思想.....1
1.1 总体思路.....1
1.2 算法流程图.....2
1.3 算法运行时间复杂度分析.....2
2 程序代码说明.....2
2.1 数据结构说明.....2
2.2 函数说明.....5
2.3 程序限制.....9
3 实验结果.....11
3.1 测试数据.....11
3.2 结果分析.....11
4 实习过程总结.....13
4.1 分工与合作.....13
4.2 经验与教训.....14
4.3 建议与设想.....14
5 致谢.....14
6 参考文献.....14

1 算法思想

1.1 总体思路

本二〇四八 AI 算法主体上采用决策树，选定搜索深度，利用设计好的估值函数计算叶结点的评估值，在深度优先搜索的框架下通过 Alpha-Beta 剪枝加速 minimax 逐层向上的搜索过程，最后在根节点选择评估值最大的分支作为行动策略。

估值函数对当前局面进行的评估主要分为 base 和 offval 两部分，其中 base 评估己方棋子的大小和位置，棋子级别越高、越远离两方交界处，base 的评估值越高，offval 评估己方棋子上下左右最邻近的棋子，如果邻近棋子也是己方棋子，则两者级别相差越多，offval 的

评估值越低，但如果邻近棋子是对方棋子且与己方棋子级别相同则会对 offval 产生一个依赖于级别高低的正贡献，最终由 base 和 offval 加权求和得到估值。

最后，为了提高决策的工作效率，并且实现基于当前局面有对策的攻防算法，我们采用在进入决策树进行决策之前首先会对当前局面进行分析，优先进行吞并和防守。

1.2 算法流程图

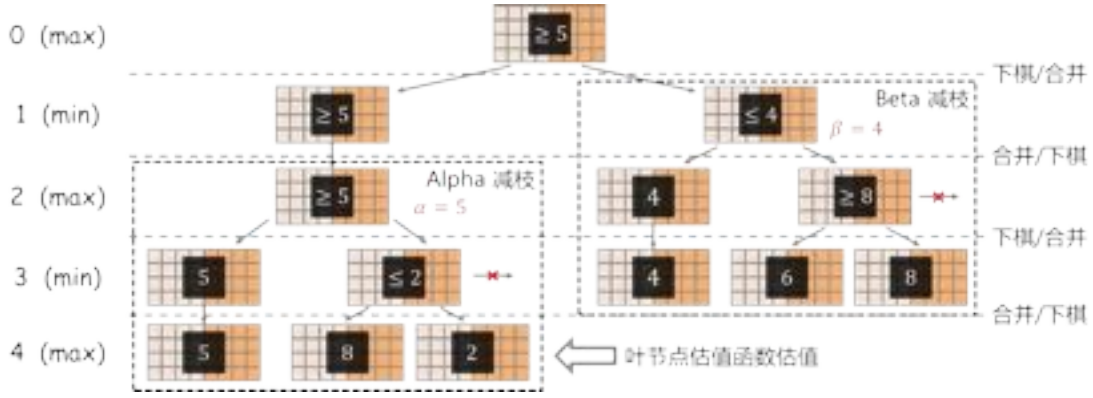


图 1 算法流程图

1.3 算法运行时间复杂度分析

由于棋盘大小固定，故对任一静态棋面用估值函数进行估值的时间复杂度都是$O(1)$的，由于决策树每一步往下都产生更多的分支，如果设搜索深度为N，那么总体的时间复杂度应该是$O(\chi^N)$的，设下棋时可能的位置数平均占棋盘位置总数的一半（即 16），而方向合并时可能的操作数一般为 4，故在不考虑 Alpha-Beta 剪枝时平均来说应当有$\chi \approx \sqrt{16 \times 4} = 8$，如果考虑到剪枝时间复杂度会有所改观，最优时可以达到$O(\chi^{N/2})$，最差时和没剪枝的时间复杂度相当，仍然为$O(\chi^N)$，平均而言时间复杂度为$O(\chi^{3N/4})$。

2 程序代码说明

2.1 数据结构说明

为了便于组内代码构建，我们首先重新定义了棋盘类 **GameBoard**，一些主要方法如下：

- __init__()初始化时首先分别导入棋盘和局数到 self.board 和 self.round，并把棋盘数据的原始格式 self.raw 转换为我们更适用的 self.matrix
- getBelong()返回棋盘上某位置是属于先手还是后手，属于先手返回 True，属于后手返回 False

```
def getBelong(self, position):
    return self.raw[position[0]][position[1]][1]
```

- getValue()返回棋盘上某位置的棋子级别（棋子的数值取以 2 为底的对数），如果为空位返回 0

```
def getValue(self, position):
    return self.raw[position[0]][position[1]][0]
```

- getMatrix()返回当前棋面，先手方为正，后手方为负，返回4×8的二维列表

```
def getMatrix(self):
    return [[self.getSignedValue((i, j)) for j in range(8)] for i in
range(4)]
```

- getEnemyAvailable()和 getSelfAvailable()分别返回对方和己方能下棋的位置
- getValuation()通过调用估值函数 calValue()返回对当前棋面的估值

```
def getValuation(self, player):
    return calValue(self.matrix, player)
```

- setChess()返回在某位置下棋后的新棋面

```
def setChess(self, player, position):
    boardNew = self.board.copy()
    boardNew.add(player, position)
    return GameBoard(boardNew, self.round + 1)
```

- moveChess()返回向某方向合并后的新棋面

```
def moveChess(self, player, direction):
    boardNew = self.board.copy()
    boardNew.move(player, direction)
    return GameBoard(boardNew, self.round + 1)
```

由于算法主体采用决策树，故主要的数据结构为树，我们自定义决策树类 **GameTree**，一些主要方法如下：

- __init__()初始化时定义棋盘 self.board、操作符 self.operand、深度 self.depth、子节点列表 self.child、父节点 self.parent、棋盘估值 self.value 和子节点估值列表 self.childValueList
- insertChild()以 self 为父节点插入子节点，同时子结点深度+1

```
def insertChild(self, board, operand):
    childNode = GameTree(board, operand, self.depth + 1, [], self)
    self.child.append(childNode)
    return childNode
```

- remove()把当前节点包括之后所有子节点都从决策树中去除，为后续剪枝做准备

```
def remove(self):
    self.child = []
    self.parent.child.pop()
    self.parent = None
```

- 计算估值时，由于对叶节点和其他节点计算估值的方法不同，故分别定义 calValueByFunc()和 calValueByChild()两个方法，其中 calValueByFunc()直接按照估值函数估值

```
def calValueByFunc(self, player):
    self.value = self.board.getValuation(player)
```

而 calValueByChild()则根据节点深度判断计算节点是 max 节点还是 min 节点，从而取子节点估值中的最大或最小值作为估值，此外这里面的 POS_INF 和 NEG_INF 设置为估值能取到的上限和下限

```
def calValueByChild(self):
    if len(self.child) != 0:
        for x in self.child:
            if x.value:
                self.childValueList.append(x.value)
        if len(self.childValueList) != 0:
            if self.depth % 2 == 0:
                self.value = max(self.childValueList)
            else:
                self.value = min(self.childValueList)
        else:
            if self.depth % 2 == 0:
                self.value = POS_INF
            else:
                self.value = NEG_INF
    else:
        if self.depth % 2 == 0:
            self.value = POS_INF
        else:
            self.value = NEG_INF
```

- getChoice()返回最后选择做出的操作 operand

```
def getChoice(self):
    if self.child:
        if self.child[-1].child:
            return self.child.pop().operand
        else:
            self.calValueByChild()
            return self.child[self.childValueList.\
                    index(self.value)].operand
    else:
        return None
```

此外，还有玩家类 **Player**，初始化时根据己方是先手还是后手生成己方 self.myID 和对方 self.opID，先手为 True，后手为 False，类似的，生成己方和对方对应的初始列范围 self.mycolumn 和 self.opcolumn。Player 内自置一个白痴 AI 方法 idiotai，当计算超时或报错时，此局由白痴 AI 代玩。Player 的主体内容写在 output 函数里，为防止超时，大体上为两种情况分别搜索 4 层和 2 层，例如当场面上可下位置超过 16、剩余计算时间不够多或者当

前局数较低时只搜 2 层，具体进行决策时会根据己方先手或后手、模式是下棋还是合并方向以及搜索深度来调用 treeFunc()函数，关于 treeFunc()具体内容可参见 2.2 节。

2.2 函数说明

这里主要说明三个函数，估值函数 **calValue()**和策略函数 **treeFunc()**以及攻防函数：

- 估值函数 calValue(matrix, player)传入参数为棋盘矩阵 matrix 和玩家 player，其中 matrix 是4×8的二维列表，player 为 True 时己方为先手，反之为后手。
  如前所述，calValue()分为 win，safe，link，edgesafe，clear，safe2 几个部分。为了区分先后手，在各个部分的命名前加上 off 或者 def。
  win：将双方的棋子从大到小排序，放入 winlist 列表，并一一比较大小，直到决出胜负。该部分仅从搜索至最大棋子等级-3 的棋子，一是为了减少时间，二是低等级棋子的作用在大部分时间可能比较有限。具体代码如下：

```
for i in range(min(len(offwinlist), len(defwinlist))):
    if offwinlist[i] > abs(defwinlist[i]):
        win = offwinlist[i] * offwinlist[i] * 30
        break
    if offwinlist[i] < abs(defwinlist[i]):
        win = -defwinlist[i] * defwinlist[i] * 30
        break
if win == 0:
    if len(offwinlist) > len(defwinlist):
        ow = offwinlist[len(defwinlist)]
        win = ow * ow * 30
    elif len(offwinlist) < len(defwinlist):
        dw = defwinlist[len(offwinlist)]
        win = -dw * dw * 30
```

  safe：评估最大棋子的安全性的，以己方棋盘的两角（例如先手方为[0][0]以及[3][0]）为最安全，其他位置的安全系数逐渐减少（按照己方两角-己方最靠里的边-其他位置递减）。具体代码如下（以先手方为例）：

```
for loc in offmaxloc:
    if (loc[0] == 1 or loc[0] == 2) and loc[1] == 0:
        offsafe += 2*offmax*offmax
    elif (loc[0] == 0 and loc[1] == 0) or (loc[0] == 3 and loc[1] == 0):
        offsafe += 10*offmax*offmax
    elif (loc[0] == 0 or loc[0] == 3) and loc[1] == 1:
        offsafe += 1*offmax*offmax
    elif (loc[0] == 1 or loc[0] == 2) and loc[1] == 1:
        offsafe += 0.5*offmax*offmax
    else:
        pass
```

但是后来 AI 改成进攻型之后取消了这个部分的估值。
link：找到己方最大棋子的位置（可以有多个），然后向四周递归搜索，寻找一个非严格递减的列表。link 具体数值的计算方法是，遍历数列每一项，如果与前一项相等，则乘上两倍当前项；比前一项少 1，则乘上当前项；否则，不乘当前项。如果最大棋子有多个，随后比较每个棋子的 link 数值大小，选出最大的。这是为了找到一个最好的合并成更大的棋子的路径。原本只搜索己方棋子，后来 AI 改成进攻性之后改为双方旗子都加入搜索。具体代码如下（以先手方为例）：

```
def recur(loc, round):
    if round == 4 or loc == []:
        return
    curmax = 0
    nextloc = []
    for i in range(4):
        if 0 <= (loc[0] + dx[i]) < 4 and 0 <= (loc[1] + dy[i]) < 8:
            cur = abs(matrix[loc[0] + dx[i]][loc[1] + dy[i]])
        else:
            continue
        if cur > curmax and cur <= matrix[loc[0]][loc[1]]:
            curmax = cur
            nextloc = [loc[0] + dx[i], loc[1] + dy[i]]
    if curmax != 0:
        lklist.append(curmax)
    recur(nextloc, round + 1)
    return
 for i in offmaxloc:
    lklist = []
    recur(i, 0)
    delta = 0
    partofflink = 1
    if lklist != []:
        partofflink *= lklist[0]
    for i in range(1, len(lklist)):
        if lklist[i] == lklist[i - 1]:
            partofflink *= lklist[i] * 2
        elif lklist[i] == lklist[i - 1] - 1:
            partofflink *= lklist[i]
    if partofflink > offlink:
        offlink = partofflink
```

edgesafe：边缘安全性。在棋盘中间两列寻找己方棋子，如果己方棋子和对方相邻棋子的绝对值相同，则可能被对方吃掉，认为安全系数减少。AI 改成进攻后原本计划取消，但是由于缺少调试，最终保留了下来。具体代码如下（以先手方为例）：

```
offedgesafe = 0
for i in range(4):
    for j in range(1, 3):
        for k in range(4):
            if matrix[i][j] <= 0:
                continue
            if 0 <= (i + dx[k]) < 4 and 1 <= (j + dy[k]) <= 3:
                cur = matrix[i + dx[k]][j + dy[k]]
            else:
                continue
            if cur < 0:
                if abs(cur) == matrix[i][j]:
                    offedgesafe += cur * cur * 10
```

clear：计算己方区域的空格数，空格越多 clear 值越高。该部分意在制造塞爆对方的局面，让对方无法动弹，同时避免被对方塞爆。具体代码如下（以先手方为例）：

```
offclear = 0
offclearcnt = 0
for i in range(4):
    for j in range(4):
        if matrix[i][j] == 0:
            offclearcnt += 1
offclear = offclearcnt * offmax * offmax
```

safe2：计算己方区域的对方棋子数，有对方棋子分数减小。该部分可以增强 AI 侵略性，使 AI 容易进入到对方区域，争取吃到对方棋子。具体代码如下（以先手方为例）：

```
offsafe2 = 0
    for i in range(4):
        for j in range(4):
            if matrix[i][j] < 0:
                offsafe2 += matrix[i][j] * matrix[i][j] * 10
```

假如己方第i个棋子位置可以由(r_i, c_i)表示，其级别为v_i，一共有N个己方棋子，则我们设计 base 具有形式

$$\text{base} = \sum_{i=1}^{N} v_i * 2^{|c_i-4|} * 2^{v_i}$$

可以看出棋子级别越大、位置离交界越远，base 越大。

而对于 offval 则总体分两种情况讨论，假如己方棋子位于己方棋盘上（即左侧），则在己方棋盘加上对方棋盘挨着己方棋盘的一列（即第五列）的范围内统计该位置上下左右四个方向最邻近的棋子，如果最邻近棋子也是己方棋盘，则两者级别差距越大，offval 减少得越多，如果两者级别相等，则对 offval 没有影响；但如果最邻近棋子是对方棋子且和己方棋盘级别相同，则此时可以通过合并方向吞并对方棋子，我们认为这种情况能

对估值产生一个较大增益。我们以左侧最邻近为例子给出 offval 的具体计算方法：

```
for i in range(column - 1, -1, -1):
    if offmatrix[row][i] > 0:
        offval -= offmatrix[row][column] * \
        abs(offmatrix[row][column] - offmatrix[row][i])
        break
    elif offmatrix[row][i] < 0 and \
        offmatrix[row][column] + offmatrix[row][i] == 0:
        offval += offmatrix[row][column] ** 2
        break
```

而如果己方棋子在对方棋盘上（即右侧），则只考虑上下左右紧挨着的棋子，如果紧挨着的棋子是对方棋子且和己方棋盘级别相同则类似地对 offval 产生一个正的增益，仍然以左侧紧挨着的棋子为例子可以给出 offval 的计算方法：

```
if column > 4 and offmatrix[row][column - 1] + \
    offmatrix[row][column] == 0:
    offval += offmatrix[row][column] ** 2
    break
```

最后将 base 和 offval 按照$10 * \text{base} + \text{offval}$加权求和得到估值。

- 决策函数 treeFunc(node, _alpha, _beta, mode, player, num)传入参数为树节点 node、剪枝参数_alpha、_beta、模式 mode、玩家 player 和搜索深度 num，其中 mode 为 1 时为下棋，mode 为 0 时为合并；player 为 1 时己方为先手，player 为 0 时己方为后手。在设置后基本传入参数后，首先构建下一层可行的操作集合

```
if mode:
    operList = copy.deepcopy(node.board.getEnemyAvailable(player))
    if len(node.board.getSelfAvailable(player)) != 0:
        operList.append(node.board.getSelfAvailable(player))
else:
    operList = []
    for i in range(4):
        if node.board.ifMove(player, i):
            operList.append(i)
operList.reverse()
```

接着按照深度优先搜索递归调用 treeFunc()，对叶节点调用 calValueByFunc()进行估值，再逐层向上调用 calValueByChild()进行估值并更新剪枝参数_alpha 和_beta，同时向上传递剪枝参数_alpha 和_beta 进行剪枝，最后用 getChoice()方法在根节点输出决策后进行的操作，具体代码如下：

```
if operList and len(operList) != 0:
    for operand in operList:
        new_board = node.board.operate(player, operand)
        new_node = node.insertChild(new_board, operand)
        if new_node.depth == num:
            new_node.calValueByFunc(new_player)
            if parent_alpha:
                if new_node.getValue() <= parent_alpha:
                    node.remove()
                    break
            if parent_beta:
                if new_node.getValue() >= parent_beta:
                    node.remove()
                    break
        else:
            alpha, beta = treeFunc(new_node, alpha, \
            beta, new_mode, new_player, num)
            if parent_alpha:
                if new_node.getValue() <= parent_alpha:
                    node.remove()
                    break
            if parent_beta:
                if new_node.getValue() >= parent_beta:
                    node.remove()
                    break

node.calValueByChild()
if parent_alpha:
    parent_alpha = max([node.getValue(), parent_alpha])
if parent_beta:
    parent_beta = min([node.getValue(), parent_beta])

if node.depth == 0:
    return node.getChoice()
else:
    return parent_alpha, parent_beta
```

- 攻防函数作为进入决策树之前的判断环节，并没有特别以方法的形式单独实现。攻防函数的核心思想就是，在当前局面有己方的棋子可以吞并对方的棋子实现侵略或者防守的时候，优先进行吞并，存在多种可能性的时候有限进行大数的合并；在当前局面有己方的棋子可以进行合并的时候，优先进行大数的合并；当我方棋子移动之后处在可以被对方棋子吞并的位置时，规避朝着这个方向的移动；当我方棋子进行吞并之后处在被对方棋子吞并的位置时，规避朝着这个方向的移动。
  在代码实现上，我们首先将矩阵和移动方向按照先后手进行了重新排列，统一以对手所在的方向为右边，在代码中实现为 offmatrix 和 UP, DOWN, LEFT, RIGHT 以简化书写，

对于左边的棋子，对于可以合并的己方或者对方棋子，加入词典 defdir(表示己方的重复元素)和 offdir(表示对方的重复元素)

```
    for i in range(column - 1, -1, -1):
        if offmatrix[row][i] == offmatrix[row][column]:
            defdir[offmatrix[row][column]] = LEFT
            break
        elif offmatrix[row][column] + offmatrix[row][i] == 0:
            offdir[offmatrix[row][column]] = LEFT
            break
```

对于右边位置的棋子，对于可以合并的己方或者对方棋子，加入词典 defdir 和 offdir，如果没有找到，就看对面的邻接位置上有没有可以合并的己方或者对方棋子，加入词典

```
    for i in range(column + 1, 4):
        if offmatrix[row][i] == offmatrix[row][column]:
            defdir[offmatrix[row][column]] = RIGHT
            break
        elif offmatrix[row][column] + offmatrix[row][i] == 0:
            offdir[offmatrix[row][column]] = RIGHT
            break
    else:
        if offmatrix[row][4] == offmatrix[row][column]:
            defdir[offmatrix[row][column]] = RIGHT
            break
        elif offmatrix[row][column] + offmatrix[row][4] == 0:
            offdir[offmatrix[row][column]] = RIGHT
            break
```

对于上下的操作与向左的操作类似

```
for j in range(row - 1, -1, -1):
    if offmatrix[j][column] == offmatrix[row][column]:
        defdir[offmatrix[row][column]] = UP
        break
    elif offmatrix[row][column] + offmatrix[j][column] == 0:
        offdir[offmatrix[row][column]] = UP
        break
for j in range(row + 1, 4):
    if offmatrix[j][column] == offmatrix[row][column]:
        defdir[offmatrix[row][column]] = DOWN
        break
    elif offmatrix[row][column] + offmatrix[j][column] == 0:
        offdir[offmatrix[row][column]] = DOWN
        break
```

此外，对于在对方区域内停留的棋子，也要尽可能的吞并对面的棋子

```
for column in range(4, 8):
    if offmatrix[row][column] > 0:
        if column < 7 and offmatrix[row][column + 1] + offmatrix[row][column] == 0:
```

```
            offdir[offmatrix[row][column]] = RIGHT
        elif column > 4 and offmatrix[row][column - 1] + offmatrix[row][column] == 0:
            offdir[offmatrix[row][column]] = LEFT
        elif row > 0 and offmatrix[row - 1][column] + offmatrix[row][column] == 0:
            offdir[offmatrix[row][column]] = UP
        elif row < 3 and offmatrix[row + 1][column] + offmatrix[row][column] == 0:
            offdir[offmatrix[row][column]] = DOWN
```

最后，将这些词典中的元素按照 offdir>defdir，同类数值越大越优先的顺序进行选择。当己方是先手方时，会用同样的方法考虑对方是否在己方行动之后有吞并的机会，如果有机会进行吞并，会进入决策树并且在决策树完成之后规避让对方有机可乘的行为。

2.3 程序限制

程序最大的限制在于计算超时，在每一次决策开始时，我们会先进行攻防算法的考核，如果不能有吞并的机会，就会进入四层搜索的决策树，当决策树可能存在问题或者决策树的结果由于会导致对面趁机偷家而被攻防算法排除的时候，我们会使用内置的白痴 AI 方法 idiotai 进行决策，虽然会大大影响整体 AI 的效能，但是不会报错。

3 实验结果

3.1 测试数据

实验环境说明：

- 硬件配置：2.2 GHz Intel Core i7 / 16 GB 2400 MHz DDR4
- 操作系统：macOS Mojave / 10.14.6
- Python 版本：3.7.1

测试方法采用与白痴 AI 对战，战绩良好，200 局的胜率如下表：

状态	平均用时	胜率	超时率	出错率
先手	3.912 s	79%	17%	0%
后手	3.347 s	84%	9%	0%

表格 1 与白痴 AI 对战战绩

热身赛战绩为：794 胜；1016 负；0 平。

3.2 结果分析

由于本算法用到的估值函数比较有侵略性，在能吞并对方棋子时对估值有较大增益，故往往能够通过吞并对方实现胜利，经典胜局如图，第 106 回合，我方为后手向左移动，吃掉对方 256 合成 512。

此后在 186 回合吃掉对方 512。在 249-250 回合连续吃掉对方 128 和 256。在 306 回合又吃掉对方 512。（下图为 186 回合与 306 回合，随后我方吃掉对方棋子）

最终我方得到了 2048，比对手最大值 64 的级别高出了 5。至于运行时间，在测试数据和热身赛中，本算法控制在限定时间内，但是在排名赛中时常出现超时。

图 2 经典胜局

4 实习过程总结

4.1 分工与合作

讨论：李春江 林熠伟 欧阳筱羿 吴昊京 赵鑫淼
算法设计和程序编写：李春江 林熠伟
报告撰写：欧阳筱羿

合作交流方式：腾讯会议+微信群

图 3 小组讨论 0523

图 4 小组讨论 0528

4.2 经验与教训

在实习过程中，整体思路比较清晰，但囿于时间限制没有进行更好的优化。显著问题是，算法编写时间比较长，并且整体架构和估值函数由不同同学编写，最终合体时间比较晚，没有时间在正式的天梯赛中进行调试，导致无法较好地解决在热身赛中没很少出现，但在正式赛事中却经常出现的超时情况，影响了比赛成绩。

另外尽管估值函数的设计考虑到了诸多因素，也进行了大量调试，最终作战效果仍然不算好。特别是与一些 AI 交战时，容易遇到“偷家”的情况：我方高等级棋子被对方棋子深入吃掉。我们对该情况做了特别优化，但仍然不能完全避免。

4.3 建议与设想

建议在以后的实习大作业过程中稍微延长准备时间和周期，避免与其他课程重要事宜的截止时间冲突。

5 致谢

感谢陈斌老师的课程指导和技术组对平台的搭建和维护。

6 参考文献

[1] Wikipedia contributors. (2020, June 6). Alpha–beta pruning. In *Wikipedia, The Free Encyclopedia*. https://en.wikipedia.org/w/index.php?title=Alpha%E2%80%93beta_pruning&oldid=96108

9584
[2] Github contributors.
https://github.com/lihongxun945/myblog/labels/%E4%BA%94%E5%AD%90%E6%A3%8BAI%E6%95%99%E7%A8%8B%E7%AC%AC%E4%BA%8C%E7%89%88

数据结构与算法课程实习作业报告

（陈思宇 姜鹏杨 沈超凡 万山铨 朱晗*）

摘要：本算法主体框架是蒙特卡洛决策树，即基于当下棋局的情况考虑可操作动作，从根节点开始生成多叉树并进行迭代，于叶结点处开始模拟，模拟的步数在事先确定，根据模拟结果更新决策树中的相应节点，在我们设定的最大迭代次数限制内重复上述过程。经过调试我们确定共迭代 20 次，每次模拟深度为 12，在 10 次的模拟对战中，算法在进行了平均 188 回合后便会调用应急策略，最终我们的代码锁定在了 950 余分。

关键字：蒙特卡洛决策树；喂子策略；攻击策略；剪枝策略；模拟

1 算法思想.....1
1.1 总体思路.....1
1.2 算法流程图.....2
1.3 算法运行时间复杂度分析.....3
2 程序代码说明.....3
2.1 数据结构说明.....3
2.2 函数说明.....3
2.3 程序限制.....5
3 实验结果.....5
3.1 测试数据.....5
3.2 结果分析.....7
3.3 经典战局（可选）.....7
4 实习过程总结.....7
4.1 分工与合作.....8
4.2 经验与教训.....10
4.3 建议与设想.....11
5 致谢.....11
6 参考文献.....12

1 算法思想

1.1 总体思路

本算法在经典的蒙特卡洛决策树算法上进行了相应改动。主体框架依然是蒙特卡洛决策树的框架，即基于当下棋局的情况考虑可操作动作，从根节点开始生成多叉树并进行迭代，于叶结点处开始模拟，模拟的步数在事先确定，根据模拟结果更新决策树中的相应节点，在我们设定的最大迭代次数限制内重复上述过程。

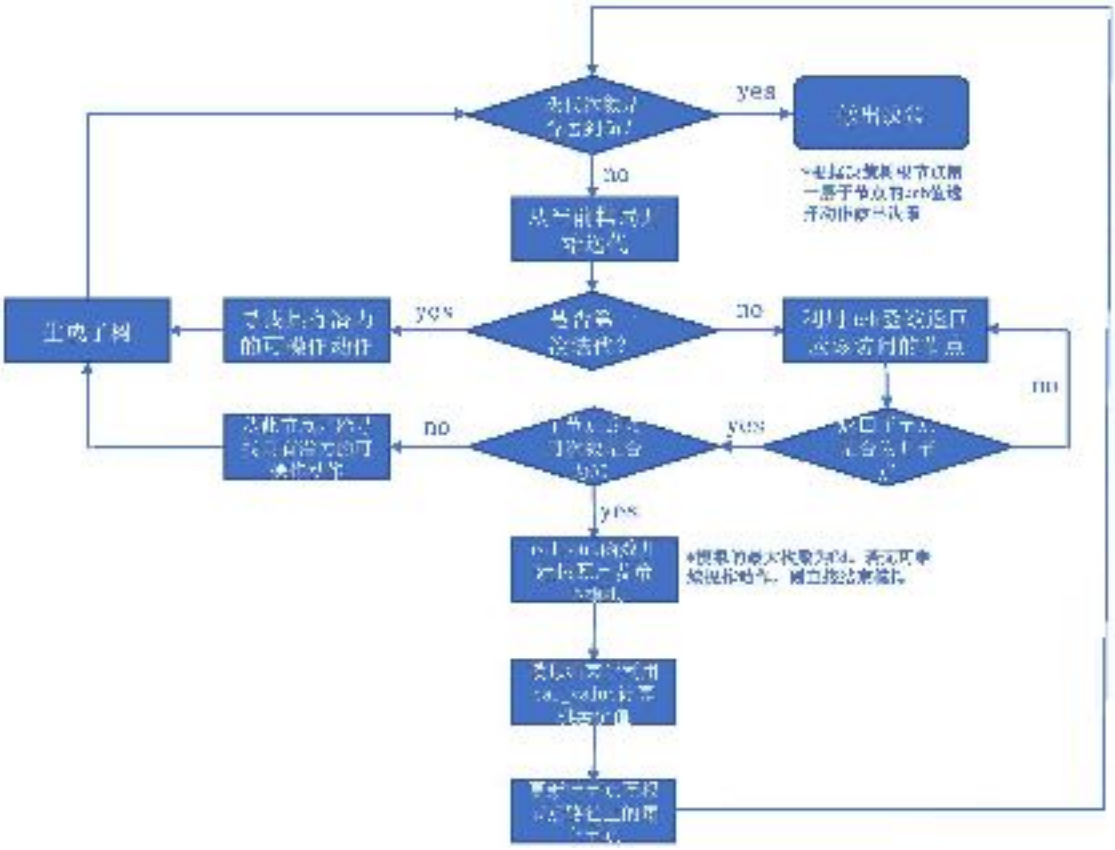
值得注意的是，由于本游戏存在两类动作，即落子和移动，同时需要考虑到先后手，因

此在每一次迭代中，一方面需要记录决策树的搜索深度以确定下一个可执行动作的发出者和类型；另一方面需要区别我方与对方在选择可执行动作时的差别，即我方始终选择令我们最有利的动作，而对方则始终选择令我们最不利的动作。此外，每一次的模拟过程中也同样需要记录模拟深度，理由类似。

在传统的决策树结构上，本组又做了以下创新：首先，从根节点生成子树的过程中并不考虑每一个可操作动作，而是仅挑选具有发展潜力的动作生成子节点。其中如何判断某一步动作是否具有发展潜力是一个难点；本组成员在多次讨论后认为能够干扰对方大数生成或给对方喂子以促成对对方的吞并即为具有潜力的可操作落子动作、能够吞并对方大数或向某一角落处堆叠即为具有潜力的可操作移动动作。其次，在对决策树每一次的迭代过程中，对于叶节点的模拟并不是采用随机动作，而是采用我们事先确定的启发性动作。其中包括启发性落子策略，按照干扰对方大数生成、落在我方棋局这一优先顺序设定动作被选择概率，再依据生成的概率随机选择具体动作；以及启发性移动策略，按照移动后得分的多少这一优先顺序设定动作被选择概率，同样依据生成的概率随机选择具体动作。最后，本组成员多次讨论关于模拟结果的判读，即棋局的估值问题。最终形成的意见是将求取我方棋局上所有数字的平方和作为棋局的价值函数。

此外，考虑到对局总时长的限制，本组特地设置了当剩余时长小于某一阈值时启用应急策略，即采用相对简单的落子或移动策略以避免超时。

1.2 算法流程图



1.3 算法运行时间复杂度分析

本组成员认为，此算法的主要时间开销用于每一次迭代中搜寻具有潜力的可操作动作以及每一次模拟中启发策略的调用过程。从算法整体来看，最大迭代次数为 Cn，最大模拟深度为 Cd，复杂度 O(Cn*Cd)。每一次的迭代过程中，或者生成某一已经访问过叶节点的子节点——此时需要调用潜力节点的搜索函数，其中落子情况下潜力节点的搜索函数基本上是对对方棋局上数字的遍历，而移动情况下潜力节点的搜索函数则是计算四个方向移动后棋盘分数的变化，涉及到对我方数字的遍历；或者从某一未经访问过的叶节点开始模拟——此时需要调用启发策略函数，启发策略函数与潜力节点搜索函数的时间复杂度相似。因此当考虑到每一次迭代过程中的细节时，时间复杂度则相对较高。

在实际的运行中，我们需要根据时间限制调整参数 Cn 与 Cd。在多次实验后，我们选择 Cd 为 12，Cn 为 20，即共迭代 20 次，每一次的模拟深度为 12。最终程序的运行结果显示，在 10 次的模拟对战中，算法在进行了平均 188 回合后便会调用应急策略。算法中应急策略设置为在剩余时间不足 0.2 秒时被唤起，这说明平均 188 回合算法的总消耗时间为 4.8s。算法的高时间复杂度的确是策略设计中的巨大缺陷，在后续改进中本组成员也未能取得实质性的突破，令人遗憾。

2 程序代码说明

2.1 数据结构说明

我们的代码主要运用了线性数据结构，以及自定义 Node 类。

线性数据结构——列表，依靠 python 内置的 list 类型实现。如：搜索树节点的子树列表、下一步可能操作的列表；嵌套列表包括储存某方棋子位置和对应的值。

自定义类——Node 类，节点类主要用于存储某个操作后棋盘的状态，包括当前局面回合数、我方或者对方的评分、该节点自身被访问的次数，可以引用父节点或者子节点，即上一步或下一步操作后的状态。每一个节点包含以下信息：

board	棋盘对象	n	该节点被访问的次数
child	进行迭代后的子树	value	该节点的 value 值
operation	operation 里的操作对应 child 里的子树	ucb	该节点的 ucb 值
father	父节点	belong	我方 or 对方

该类可以实现的功能在 2.2 中进行进一步阐述。

Player 类：没有新增属性，主要通过 stimulation 函数实现模拟，每次模拟会根据当前棋盘局面新生成一个 Node 对象。包含 isFirst 和 array 两个信息，前者用于判断是先手还是后手，后者用于读取该步骤对应的已知落子表。

蒙特卡洛决策树：以 Node 对象为节点，在课堂上提到的二叉树基础上建立多叉树。

2.2 函数说明

Node 类中定义的函数:

函数名称	功能及基本实现过程
calValue	计算模拟结果后的棋局价值：我方棋局所有数字的平方和
value_update	模拟结束后对叶节点至根节点路径上的值进行追溯和更新：每个节点的访问次数 n 加一；每个节点的 value 值进行加权平均更新，权重为该节点的访问次数 n
ucb_cal	计算每个子节点的 ucb 值以选择向下继续搜索的子节点。每个节点 ucb 值的计算公式如下（C 取 0.8）：$$\frac{v_i}{v_P}+C\times\sqrt{\frac{ln\,n_p}{n_i}}$$其中 vi 为当前节点的值，vp 为父节点的值，ni 为当前节点访问次数，np 为总访问次数。同时如果该节点还未经访问过，即 n=0，则默认其 ucb 值为 None
ucb_max	ucb 函数实现对子节点优先级别的排序：在每次迭代过程中己方回合返回 ucb 值最大的节点，对方回合则返回 ucb 值最小的节点。同时注意节点的 ucb 值可能为 None，None 的优先级最高
stimulate	迭代过程，即树的生成过程，如果遇到叶节点，则判断是否到访过，若没到访过且不是根节点则模拟，否则生成子树。注意本次比赛我方是先手还是后手，如果我方是先手，生成子树的顺序是我落、对落、我移、对移，否则顺序是我落、对移、我移、对落或我移、对落、我落、对移。
rollout	叶节点的模拟过程，用目前深度+模拟深度代表目前迭代深度并对该数除 4 取余。取余的 4 种结果分别对应四种操作：0 代表接下来选点；1 则需要进一步判断（有选点和选方向两种可能）；2 代表选择方向；3 与 1 相同。
sti_point	模拟过程中所进行一步的启发性落子，本步骤选取 pre_able 中的概率最大点做为模拟结果。
pre_able	输出当前全部的合法落子点，同时输出一个通过启发性策略赋值的相对应的概率列表
enemy_none_value_cal	与 oris_value_cal 对偶，根据启发性策略对不同点进行概率赋值
sti_ori	表示模拟过程中所进行一步的启发性移动，本步骤选取了 oriantation 中概率最大的点做为模拟结果
oriantation	输出当前全部的可操作方向，同时输出一个通过启发性策略赋值确定的相对应的概率列表
oris_value_cal	进行模拟之后的分数判断，在进行当前移动的模拟下计算我方在这一步移动之后的得分，在不进行当前移动的模拟下计算对方此时的得分状况，根据启发性策略对不同方向进行概率赋值
findneibourValue	对当前点周围的邻近点求 value 值，返回一个由这些值组成的列表
如果此时时间不够，则进行：	
getMargin	返回我方和对方的非零边界点
direction	把方向进行正序排序，排序的标准是按照该放下合并之后可以得到的分数

***Player 类中定义的函数*:**

函数名称	功能及基本实现过程
output	根据剩余时间是否大于 0.2，分别判断各种情况，输出最终所做的决策，给出己方下棋或合并的位置
stimulation	迭代过程，表示树的生成，并使得树的规模与节点可操作步骤的数量有关，如若该节点只有一个可操作步骤，则不必继续探索
attack	用于时间不够的情况下采取的吃对方子的策略，不进行模拟
getMargin	返回我方和对方的非零边界点

2.3 程序限制

本程序采用了蒙特卡洛算法，理论上是对所有的合法情况挂枝后进行模拟，且在编写程序的过程中，我方已经考虑了棋盘已满或者其他无合法输出的情况下的应对之策。而在实际过程中，由于时间限制，我方采取了手动剪枝策略，即选取部分的合法情况进行模拟；同时在时间即将耗尽的时候，采取更简单的方法选取落子位置或者移动方向，以此来杜绝因为报错使得比赛输掉的情况。

因此，从结果上来说，运行与模拟比赛时均未见相关报错问题与程序漏洞。

3 实验结果

3.1 测试数据

实验环境说明：

- 硬件配置：（CPU/内存）: Intel Core m5 CPU @1.2 GHz 双核; 8 GB 1867 MHz LPDDR3
- 操作系统：（名称/版本）macOS Catalina 10.15.3
- Python 版本：Python 3.7

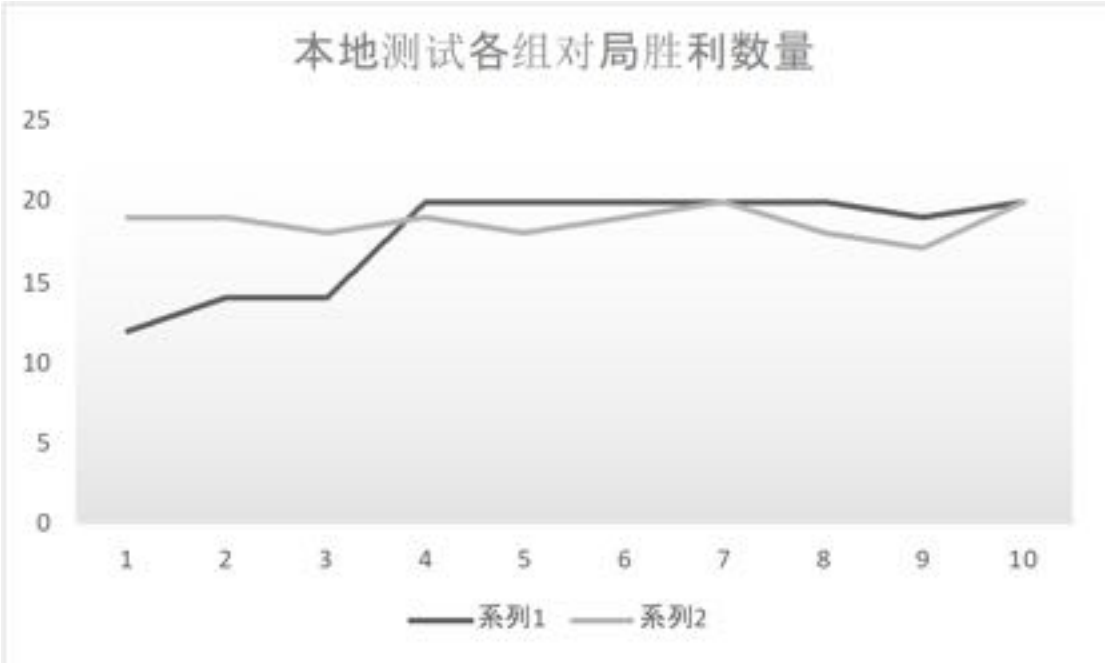
测试方法：

一、保证程序运行：

在本地的测试平台上与系统给定的默认代码以及带有建议逻辑判断的非随机决策代码进行对抗。通过不断地调试确保代码在任何情况下不会出错，以及能够保持一个较高的胜率。

在本地与默认随机决策代码共运行了 10 次 20 来回的对战，在两百场比赛中，随机运行代码获胜 21 次，其中 19 次是因为我方程序在某种特定情况下出现 bug，2 次是因为分数被击败。在修改代码之后的 140 局测试中，没有再出现 bug，而且仅出现了一次失败。

在与带有简单逻辑判断的非随机决策代码进行对抗时，我们同样也是测试了 10 组 20 来回的比赛。在 bug 已经全部排除之后，我们的结果是 187 胜 12 负。因此已经可以保证较高的胜率，下一步将是在平台上参与对抗。



<u>二、在平台上进行对抗并不断调试 value 函数：</u>

我们的代码在平台的上的经过不断调试。刚开始进行了 10 次匹配模式，比分从 1000 分掉落到 824 分。此时我们意识到代码的 value 计算有问题，并通过不断调整，在通过新旧代码的对抗来测试孰优孰劣。此后，我们也调整了每一次模拟的下一步方向的概率，并补充了不同的判断条件，调整了不同条件对于下一步模拟决策的概率函数。

最终我们在平台上的账号代码锁定在了 950 余分，在将近 600 次对抗中，有 250 余场的胜利。在与助教账户给出的不同层次代码的自由对抗中，胜率如下：

难度层级	胜率
S	0/30
A	1/30
B	5/50
C	32/60
D	45/60

3.2 结果分析

本算法最大的策略特点在于先手的喂子策略，在复盘一些棋局的过程中，我们发现了有较多的回合会出现喂子的决策安排，也即是先手时在对方的边界放子并立刻在下步骤中吃掉，不但没有损失，而且还可以压缩对方的空间。从效果上来看，该策略总体上效果可以，可以在一定程度上上保护自己的安全并压缩对方空间，给自己带来隐性的优势。但是又会出现一些局面导致自己突进的 4 分被吞噬，此时该策略可能得不偿失，但总体而言，我认为这种策略可以是一个新颖的，有突破点的策略想法。

我们通过不断的实验，确定了对于我们的策略而言较为安全的深度和次数，另外还设置了保险措施，保证自己在时间不够的情况下，可以按照不那么消耗时间的逻辑判断非随机策略进行决策，考虑到出现时间不够时一般步数已经超过 1200，局势发展应该已经有很大的区别度，孰胜孰劣已初现端倪，采取不那么优秀的算法不会改变棋局大局。因此这可以算是一种较为保险的措施。

但是我们的最后的深度和次数也仅仅是在 12 和 20。我们的时间均是消耗在了向下构建树的过程中。因为我们采用的是概率选择的生成树方法，剪枝不是很实用，我们的缩减代码主要是对建树过程进行简化等等。但从结果来看深度和次数都没有达到比较大的水平。

3.3 经典战局（可选）

我方判断，该改进版 2048 致胜的核心策略为：尽快生成大数，且不易被对手堵死。在这个核心策略框架下，为了尽快生成大数，获取优势并且能够同时保证该数安全：

一方面我们将吃子策略的优先级提前，

另一方面在棋局之初使用了喂子策略，使得“生成-移动-吃子”的步骤可以简化为“生成-吃子”，来尽可能地加速生成大数；

同时，我们还设置了攻击策略来围堵对方，使得“对方大数生成”的速度大大降低。

4 实习过程总结

4.1 分工与合作

在项目初期，小组成员使用腾讯会议进行了第一次线上交流，彼时大家对程序的实现仅仅是只言片语、纸上谈兵，并没有深入的思考和了解，于是本小组组内大致分为策略组和代码组，策略组负责提出完整的策略和后续改进，代码组负责提前熟悉平台接口、代码风格等。

随后，大家又进行了一次线上研讨交流，策略组的人简要介绍思路与框架，代码组的成员进行质询与改进，并定下代码初版的时间节点。

然而第一版的代码并不成熟，在运行时出现超时、无法落子等各种各样的问题，策略组和代码组之间的想法衔接并没有想象中的好，但 5 位成员并没有任何一个人提出任何抱怨，重整旗鼓，再一次共同商讨代码的整体思路有无改进。

与此同时，微信群内的消息总是不断。大家随时想到的新点子、新想法、新改进，或者注意到代码存在的 bug，随时在群内提出，总是能得到积极的回应。

本次商讨的结果是代码的各个函数、接口、功能、模块等进行了大换血，组内成员也不再按照策略组和代码组进行分工，而是针对代码撰写、实现、调试、质询等各个环节进行分工合作，各司其职。最终在大家的共同努力下，能在本地顺利运行的代码雏形终于诞生，但仍然存在放到代码竞技场中由于接口问题报错和胜率低、判断耗时长的种种问题，屡战屡败。

凌晨 12 点-3 点，不同的城市、不同的房间，亮起灯光，大家再次齐上阵，总结出了有待改进的三个问题并有针对性地分配了任务，一名组员尽全力解决一个问题，另外两名组员做各个函数功能的校验和细节上的修改。

修改版的代码终于在竞技平台上可以顺利运行并取得较为满意的胜率，大家各自又经过了一天的深思熟虑，思考代码有无继续改进的地方，从调试结果来看，在有限的蒙特卡洛树步骤里还是可以按照预想的方式走，但离理想状态只差临门一脚，5 月 29 日 22:54 分，终版代码上传到了竞技场中。

本小组的代码文件名从 player_初稿，到 player_2……到 player_10……最后到 player22_end，再到终于上传到代码竞技场上，有一种“打死也不改了”、“打不死再接着改”的感觉。虽然临近期末季时间紧张，总有时间安排不开的情况，但大家总能排除困难，没有条件也要创造条件，最终完成一次又一次的高效交流。

组内成员具体分工情况如下（按姓氏首字母排序）：

姓名	分工
陈思宇	初版代码部分函数功能的实现；对策略与代码进行质询，提出细节的改进意见，确保各函数接口及功能的对应，辅助撰写报告 2.1 环节
姜鹏杨	对于策略进行代码框架上的构建以及具体函数接口的编写，参与代码的调试优化，参与 value 函数的优化和模拟决策的优化，辅助撰写报告的 3 环节
沈超凡	策略研究与讨论，提出策略主要算法和框架（蒙特卡洛决策树），确定使用的数据结构和类；在初版代码框架下进行修改，调整部分函数，对超时的函数进行调整和 debug；辅助撰写报告的 1 环节
万山铨	策略研究与讨论、并编写文字版策略与代码组对接（对代码主体构造和主要功能函数进行规定，便于程序实现）；在初版代码框架下进行修改，调整节点模拟过程全部函数；为代码添加测试行，实现打印蒙特卡洛树结构与各主要功能时

	间消耗情况功能；找到增加时间复杂度（主要是因为形成了循环）的几个 bug，使得树的模拟深度和次数都有较大提升；辅助撰写实验报告的 2.3 和 3.3 环节
朱晗 (组长)	参与策略研究与讨论，对不完善的地方进行质疑和创新；对代码细节进行 debug、检查、校验，确保各函数接口及功能的对应；组织组员进行研讨交流，根据时间和能力协调组内分工，记录会议进程，调整项目进度；撰写整合报告

历次线上会议及微信群内交流图片如下：

（商讨策略，进行分工）

（历次会议共享屏幕，思路讲解）

（微信群内及时交流）

（代码终于完成，云合影留念）

4.2 经验与教训

得意之处：

我们认为，我组的各位同学之间的合作是十分友善的，高效和谐的协作交流给我们繁忙的期末季节省了很多时间。

固然我们不得不承认，由于各组员能力存在差异，在具体任务的分配上有多有少，在对整个项目的贡献上有大有效，但我们的分工是根据每个人的能力、精力、时间等实际情况决

定的，每个人都竭尽所能地贡献了自己最大的力量。

虽然最后没有出线，但 5 位组员都承认，这是本科期间最有默契、最有收获、最愉快的一次小组合作作业。

有待改进：

是什么蒙蔽了我们的双眼？是数理基础。这支由 4 位城环同学和 1 位经院同学组成的队伍虽然队名叫 Alpha，群名叫数算 Go，但并不像 AlphaGo 那样强大。

我们组内分为策略组和代码组，我们不得不承认的是，策略组在前期设计策略时，虽然提出了很好的设想，但没能很好地估算所需时间复杂度，导致后期实现时，不得已放弃了原有思路，临时更换，采取了保守算法，但付出的代价是胜率降低；我们还需要承认的是，代码组写出的代码几经修改仍存在冗余，并不如大佬们写得那样精致简洁、通俗易懂。

另外，在时间安排上也出现了小小的问题，比如大家还有其他的 ddl、低估了改代码的工作量、竞赛日期与考试时间冲撞等等，我们没有提前考虑到。

4.3 建议与设想

建议：

在组队方面：本课程的大多数同学先试图与熟悉的人抱团，实在缺人手的时候才会在课程群内寻人，这就导致了大佬聚集碾压、菜鸡抱团取暖，如果可以的话，老师和助教们可以先征集一定数量有意愿、有能力大佬们当各组组长，各位组长再征集组员、或组员主动报名。

在时间安排方面：大作业的提交时间安排在临近期末季，实习报告的提交时间更是安排在了期末考试时间上，对大多数普通同学而言，在 10 天左右的时间内完成策略的开发、代码的实现、参数的调试、报告的撰写是有很大难度的，尤其是在今年由于疫情原因有效交流的难度加大，所以希望老师可以体察民意，适当提前项目开始时间、并延长报告截止时间。

在竞赛的赛制、基础代码设施方面：感谢搭建竞赛平台和提出建议的各位助教和大佬们，本小组在使用过程中所有需求均得到满足，无改进意见。

寄语：

陈老师的数算课在北大本科所有课程中绝对是付出与收获成正比、绝对是物超所值的，全方位、多层次、有灵活性的课程设计让同学们能够做到自主学习、举一反三，希望陈斌老师的数算课程越来越丰富多彩，选修陈老师数算的学弟学妹们越来越有收获、越来越聚、越来越卷！

设想：

继续开放代码竞技场，建立长期的比赛机制，让有继续学习意愿的同学可以改进自己的代码，与优秀作业进行对战。

可以将前几届竞赛的内容进行整合并长期开放，让有意愿的同学随时随地都可以进行大作业的联系，增加趣味性和竞争力，扩展同学们应用所学知识的途径。

5 致谢

首先感谢陈斌老师一学期以来对数据结构与算法的倾心付出和对我们的倾囊相授，以及对各项活动的精心策划，让选修这门课的同学学有所获、学有所成、学有所想、学以致用。

感谢各位助教及时的答疑解惑和鼎力支持；

感谢技术组和可视化组的各位大佬们付出时间与精力；

感谢所有为贰零肆捌的设计与改进贡献意见的同学。

最后，我们小组要感谢所有组内成员的全力付出和共同努力，大家同心协作、勇于担当，使得整个项目在轻松愉快的氛围中高效完成，虽然成绩不尽如人意，但在这个过程中大家的取得的进步和收获的友谊使我们的付出并没有被辜负，革命友谊永存！

6 参考文献

[1]刘子正,卢超,张瑞友.基于蒙特卡洛模拟和 Z 检验的“2048”游戏并行优化算法[J].小型微型计算机系统,2016,37(03):562-566.

[2]https://www.jianshu.com/p/386868335da7?utm_campaign=maleskine&utm_content=note&utm_medium=seo_notes&utm_source=recommendation

[3] https://stackoverflow.com/questions/22342854/what-is-the-optimal-algorithm-for-the-game-2048

数据结构与算法课程实习作业报告

张云开*　程宇　于士翔　高铭　曹柱

摘要：本次大作业的内容是二零四八游戏的对战改编。我们最终所采取决策树的方法，通过 minimax 算法来判断下一步的最优解来进行游戏，同时我们还利用了剪枝的方法来优化 minimax 算法的时间。在天梯赛时我们的代码取得了第四的成绩，最终我们的队伍成绩为八强，惜（bei）败（bao）于（da）我们的天敌，但也算是取得了不错的成绩。

关键字：类 决策树 $\alpha-\beta$剪枝 minimax 算法

1 算法思想.....1
1.1 总体思路.....1
1.2 算法流程图.....2
1.3 算法运行时间复杂度分析.....2
2 程序代码说明.....2
2.1 数据结构说明.....2
2.2 函数说明.....3
2.3 程序限制.....5
3 实验结果.....6
3.1 测试数据.....6
3.2 结果分析.....6
4 实习过程总结.....6
4.1 分工与合作.....6
4.2 经验与教训.....9
4.3 建议与设想.....9
5 致谢.....9
6 参考文献.....9

1 算法思想

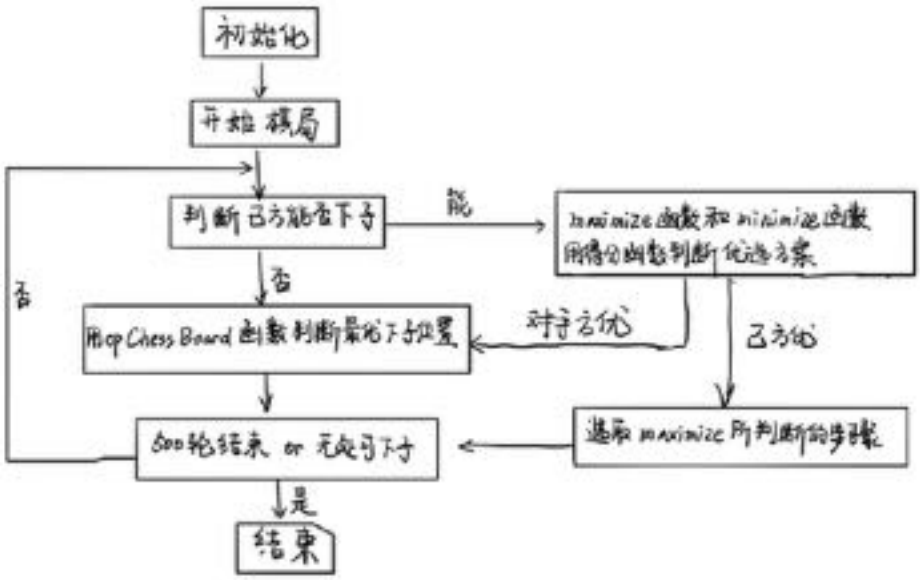
1.1 总体思路

最初，我们制定了“以己为主，对抗为辅”的对战策略，即将策略的主体部分放在了自身区域和下子合并，未针对与对手在边界上的对抗制定详细的策略，试图将课堂的 2048 游戏退化为传统 2048 游戏，该算法尽量将子集中在某一个角落，并且移动的方向优先选择偏向该角落的方向，为了达到该目的，设计了考虑棋盘棋子以及棋子间“距离”的得分函数，并通过优化该得分函数进行决策。然而，在进行一定的实践后，该策略“若对抗”的劣势暴露无遗，因此我们对该算法进行了改进。

改进后的算法采用 minmax 算法框架并采用决策树的剪枝对算法进行了一定的优化。其中的 max 代表的是在我方棋盘上的最优步骤，即使得得分函数最高的步骤，而 min 则代表在对方棋盘上的最优步骤。此外得分函数也在原来的基础上进行了较大的改进，除了考虑自身棋盘的合并，加入考虑了对方棋盘的棋子以及吃子的情况，更能适应该游戏“零和博弈”的特点。而定义的 opChessBoard 函数则用来判断在对方区域进行操作时的最优位置。最后返回出得分函数最高时的步骤选择。

1.2 算法流程图

流程图如下所示：



1.3 算法运行时间复杂度分析

我们的算法主要由六个函数构成，而根据代码可知，得分函数的时间复杂度最大，大 O 数量级为 $O(16^n)$，而通过剪枝后，其平均时间复杂度降为 $O(16^{n/2})$，其中 n 为决策树探索的层数。

2 程序代码说明

2.1 数据结构说明

主要采用的就是决策树的数据结构，然后通过递归得到的 minimax 算法，以及通过$\alpha-\beta$

剪枝来提高算法的效率，通过得分估值函数来进行判别。

2.2 函数说明

1. 初始化 player：

```
class Player:
    def __init__(self, isFirst, array):
        # 初始化
        self.isFirst = isFirst
        self.array = array
```

2. minimax 算法中的 maximize 函数：用来判断己方最优步骤。

```
def maximize(self,mode,currentRound,board,level,alpha,beta):
    '''
    max函数，即我方在某一步时我方的最佳选择，分为四种情况：先手、下子，先手、移动，等
    这四种情况下minmax操作的细节不同，因此进行较上分类。
    在下子的时候分为两种情况，在本方下以及在对方下，在本方下只有一种情况而在对方下可能有很多种情况，
    为了减少复杂度先直接将两种情况的值可能的position找出来然后进行下一步的探索。
    对于选取direction的算法，采用了基本的a-b剪枝。
    '''
    if level==0: #level = 0时，返回结果
        return None,self.boardScore(board)
    else:
        if mode=='direction': #direction模式，选择下面移动方向可能得到最大分数，或当时我的方向导分数
            bestMove=-1
            maxScore=-float('inf')
            if self.isFirst:
                nextMode='direction'
                nextRound=currentRound

            else:
                nextMode='position'
                nextRound=currentRound+1
            for move in [0,1,2,3]:
                clone=board.copy()
                avai=clone.move(self.isFirst, move)
                if avai:
                    nextAction,score=self.minimize(nextMode,nextRound,clone,level-1,alpha,beta)
                    if score > maxScore:
                        bestMove=move
                        maxScore=score
                    if maxScore >= beta:
                        break
                    alpha = max(maxScore,alpha)
            if bestMove==-1:
                bestMove=None
                nextAction,maxScore=self.minimize(nextMode,nextRound,board,level-1,alpha,beta)
            return bestMove,maxScore
        else: #position模式，选择对下子可在自己棋盘上下子选择分数最高的，若无法在己方下子，则只在对方下子
            if self.isFirst:
                nextMode='position'
                nextRound=currentRound
            else:
                nextMode='direction'
                nextRound=currentRound+1
            locPosition=board.getNext(self.isFirst, currentRound)
            if locPosition:
                clone=board.copy()
                clone.add(self.isFirst,locPosition)
```

```
                clone.add(self.isFirst,locPosition)
                bestPosition=locPosition
                nextAction,maxScore=self.minimize(nextMode,currentRound,clone,level-1,alpha,beta)
            else:
                bestPosition=None
                maxScore=-float('inf')
            opPosition=self.opChessBoard(not self.isFirst,board)
            if opPosition:
                clone=board.copy()
                clone.add(not self.isFirst,opPosition)
                newAction,score=self.minimize(nextMode,currentRound,clone,level-1,alpha,beta)
                if score > maxScore:
                    bestPosition=opPosition
                    maxScore=score
            if not bestPosition:
                maxScore=self.boardScore(board)
            return bestPosition,maxScore
```

3. minimax 中的 minimize 函数：用来判断敌方最优步骤。

```
def minimize(self,mode,currentRound,board,level,alpha,beta):
    '''
    minimize, 即对应我方操作的对方下一步最优选择, 同样对方的种情况: 先手, 下手; 先手, 移动; 等
    其实算法思想基本与maximize函数一样。
    对于选取direction的算法, 采用了基本的α-β剪枝。
    '''
    if level==0:
        return None,self.boardScore(board)
    else:
        if mode=='direction':
            bestMove=-1
            minScore=float('inf')
            if self.isFirst:
                nextMode='position'
                nextRound=currentRound+1
            else:
                nextMode='direction'
                nextRound=currentRound
            for move in [0,1,2,3]:
                clone=board.copy()
                availclone.move(not self.isFirst, move)
                if avai:
                    nextMove,score=self.maximize(nextMode,nextRound,clone,level-1,alpha,beta)
                    if score < minScore:
                        bestMove=move
                        minScore=score
                if minScore <= alpha:
                    break
                beta=min(minScore,beta)
            if bestMove==-1:
                bestMove=None
                nextAction,minScore=self.maximize(nextMode,nextRound,board,level-1,alpha,beta)
            return bestMove,minScore
        else:
            if self.isFirst:
                nextMode='direction'
                nextRound=currentRound
            else:
                nextMode='position'
                nextRound=currentRound+1
            opPosition=board.getNext(not self.isFirst, currentRound)
            if opPosition:
                clone=board.copy()
                clone.add(not self.isFirst,opPosition)
                bestPosition=opPosition
                nextAction,minScore=self.maximize(nextMode,nextRound,board,level-1,alpha,beta)
            else:
                bestPosition=None
                minScore=float('inf')
            locPosition=self.opChessBoard(self.isFirst,board)
            if locPosition:
                clone=board.copy()
                clone.add(self.isFirst,locPosition)
                newAction,score=self.maximize(nextMode,nextRound,board,level-1,alpha,beta)
                if score < minScore:
                    bestPosition=locPosition
                    minScore=score
            if not bestPosition:
                newAction,minScore=self.maximize(nextMode,nextRound,board,level-1,alpha,beta)
            return bestPosition,minScore
```

4. opChessBoard 函数：用来判断在对方的棋盘上怎样下子才是最优的位置。

```
def opChessBoard(self,first,board):
    empty_list=board.getNone(first)
    if not empty_list:
        return None
    else:
        if first:
            startCol=0
        else:
            startCol=4
        bestPoint=empty_list[0]
        maxGap=0
        for point in empty_list:
            if point[0]>0:
                lower=board.getValue((point[0]-1,point[1]))
            else:
                lower=12
            if point[0]<3:
                upper=board.getValue((point[0]+1,point[1]))
            else:
                upper=12
            if point[1]>startCol:
                left=board.getValue((point[0],point[1]-1))
            else:
                left=12
            if point[1]<startCol+3:
                right=board.getValue((point[0],point[1]+1))
            else:
                right=12
            gap_1=min(lower-1,upper-1)
            gap_2=min(left-1,right-1)
            if gap_1>maxGap or gap_2>maxGap:
                bestPoint=point
                maxGap=max(gap_1,gap_2)
        return bestPoint
```

5. boardScore 函数：是一个得分函数，用自己的一套方法来得到当时局面上的分数，从而判断最优方案。

```
def boardScore(self,board):
    '''
    评估函数主要从三个方面考虑棋盘情况：双方棋子大小，数量及位置，对方棋子大小，数量及位置；
    最为重要"棋子当前的"问题"之处。
    1、双方棋子：例如上存在更大、更多的双方棋子对我方更有利，如若双方有棋子插入对方阵地，则有位置加成。
    对于任意一个棋子，其基本得分为16**value（表现出对于大子的偏好，与点线最为的点乘比例因子），
    位置加成为v+16**max(5,value+2)，即很大对占领地长子至少都等等同比当前自己最大的5个子的价值
    2、对方棋子：计分规则与我方棋子相同，惩罚力度更强些
    3、我方棋子"问题"之处：考虑到双方棋子间实际会合并，因此两个大小相近的棋子尽量聚集在一处；
    设立"问题"为我方棋子与其周围棋子差距最小值，显然第一个最有可能合并的方向就计算等级。
    其一个棋子的"问题"是否为：gap=is"(gap)
    遍历与我方棋子的"问题"有关的所有可能情况，
    总得分=我方棋子得分-对方棋子得分-我方棋子"问题"惩罚
    '''
    if self.isFirst:
        startCol=0
    else:
        startCol=4
    score=0
    gapLoss=0
    boardRaw=board.getRaw()
    for row in range(4):
        for col in range(8):
            value=boardRaw[row][col][0]
            belong=boardRaw[row][col][1]
            if value>0:
                if belong==self.isFirst and col in range(startCol,startCol+4):
                    score = score + 16**value
                    surrList=[]
                    gapList=[]
                    if row>0 and boardRaw[row-1][col][0] !=0:
                        surrList.append(boardRaw[row-1][col])
                    if row<3 and boardRaw[row+1][col][0] !=0:
                        surrList.append(boardRaw[row+1][col])
                    if col>startCol and boardRaw[row][col-1] !=0:
                        surrList.append(boardRaw[row][col-1])
                    if col<startCol+3 and boardRaw[row][col+1] !=0:
                        surrList.append(boardRaw[row][col+1])
                    for surrPoint in surrList:
                        if surrPoint[1]==self.isFirst:
                            gapList.append(abs(value-surrPoint[0]))
                        else:
                            gapList.append(abs(value-10))
                    if gapList:
                        gap=min(gapList)
                    else:
                        gap=0
                    gapLoss += gap**(15-gap)
                elif belong==self.isFirst:
                    score += 4**(max(5,value+2))
                elif col in range(startCol,startCol+4):
                    score -= 4**(max(5,value+2))
                else:
                    score -= 16**value
    return score-gapLoss
```

6. 最后便是 output 输出函数，用来最后输出。

```
def output(self, currentRound, board, mode):
    if currentRound ==428 and (not self.isFirst) and mode != 'position': # 第428局后手方每步时long<1s
        theAction,score=self.maximize(mode,currentRound,board,4,alpha=-float('inf'),beta=float('inf'))
    elif currentRound == 499: # 第499局每步操作1s以内保底
        if mode == 'position':
            if self.isFirst:
                theAction,score=self.maximize(mode,currentRound,board,3,alpha=-float('inf'),beta=float("inf"))
            else:
                theAction,score=self.maximize(mode,currentRound,board,2,alpha=-float('inf'),beta=float("inf"))
        elif mode == 'direction':
            if self.isFirst:
                theAction,score=self.maximize(mode,currentRound,board,3,alpha=-float('inf'),beta=float("inf"))
            else: # 后手方最后一次移动，直接判断移动后的真实得分，而不是考虑敌方情况
                maxscore=-float('inf')
                theAction=None
                for move in [0,1,2,3]:
                    clone=board.copy()
                    away=clone.move(self.isFirst, move)
                    if away:
                        score=0
                        valueList=clone.getScore(self.isFirst)
                        for value in valueList:
                            score += 16**value
                        if score > maxscore:
                            theAction=move
                            maxscore=score
        else:
            return
    elif currentRound < 499: # 其余情况1s保底
        theAction,score=self.maximize(mode,currentRound,board,5,alpha=-float('inf'),beta=float('inf'))
    return theAction
```

2.3 程序限制

完善后的代码暂无出 bug 的情况出现。

3 实验结果

3.1 测试数据

实验环境说明：

- 硬件配置：（CPU/内存）：Intel(R)Core(TM)i7-7700HQ CPU @ 2.80GHz 2.80GHz / 7.88GB
- 操作系统：（名称/版本）win10 ,64 位操作系统
- Python 版本：（版本号）Python3.7.6

我们组利用的是 userinterface，在对代码修改后进行对战和分析记录。在热身赛中，对战结果方面：

AI	我们组的代码比分
2048Bot_D	10:0
2048Bot_C	10:0
2048Bot_B	7:3
2048Bot_A	8:2
s	4:2
ss	2:3/3:3/2:4
sss	0:6

热身赛结果统计表

3.2 结果分析

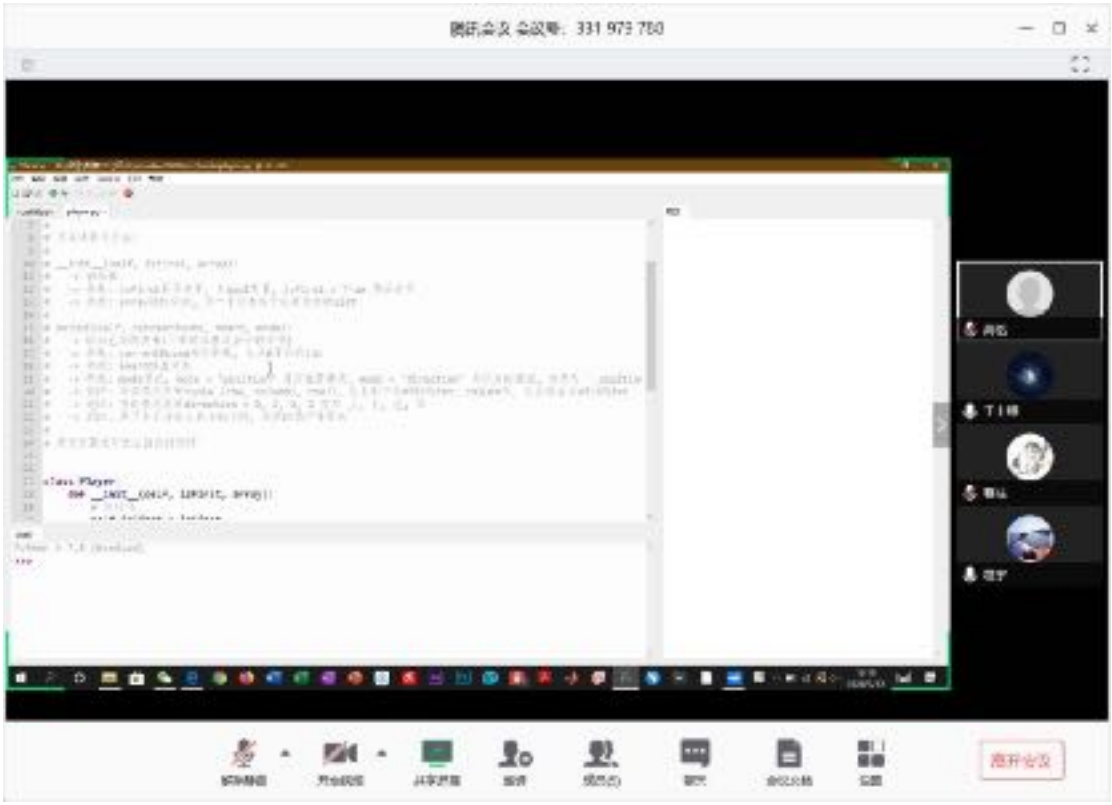
算法中的$\alpha-\beta$剪枝使得搜索树的节点数减少了，让决策树的层数可以提升到 6 层；而得分函数中的关于下在对方的子价值高 2 的决定，使得我方的子更容易在边界处合成大子侵占对方区域。

4 实习过程总结

4.1 分工与合作

我们小组的分工大致为：除了全组一起讨论分析思路以外，张云开、于士翔、程宇同学负责分别编写不同段落的代码以及代码的 debug；高铭同学负责记录和整理开会记录以及协助写报告；曹柱同学负责写报告。在大作业的编写过程中一共开了 4 次组会，由于疫情的影响组会采取的是腾讯会议在线讨论：

第一次组会（20200515）：主要任务是了解二零四八的玩法以及大作业的具体规则，同时对于算法进行初步的讨论。

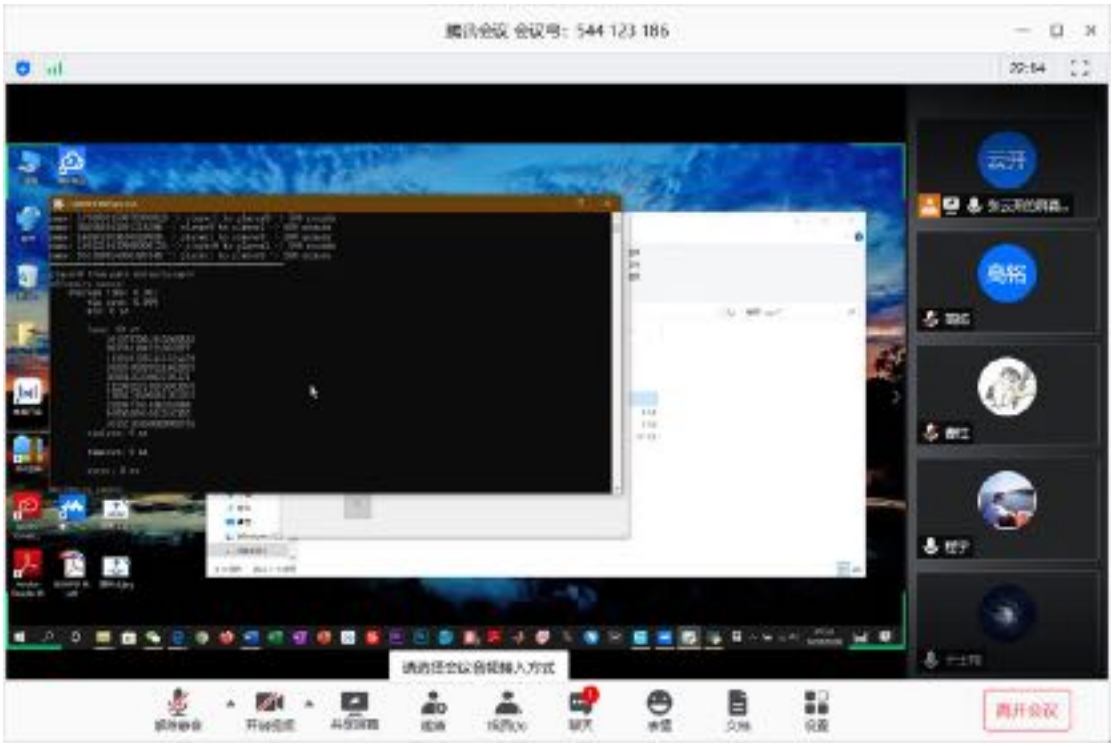


第一次组会

第二次组会（20200517）：该次讨论主要是关于合并的方向、优先级以及边界上的合并问题，和如何在劣势的情况下苟活下来，而后开始代码的编写。

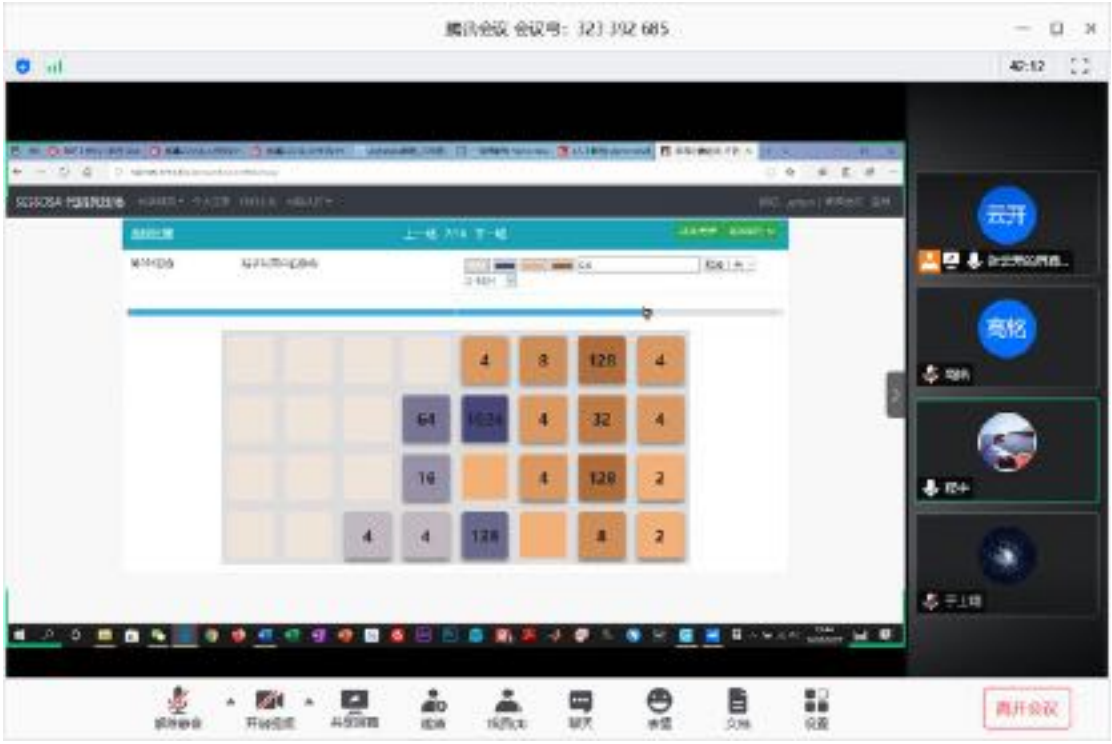
第二次组会

第三次组会（20200526）：首先进行 debug，将之前的代码合并并利用 AI 进行尝试，而后基于陈老师上课讲的 minimax 算法，对于我们的算法进行了改进，继续 debug。



第三次组会

第四次组会（20200527）：关于超时的问题进行讨论，分析老师所给的 AI，进一步优化我们的代码。



第四次组会

最后，在写代码的几位同学的熬夜肝和在群内讨论之下，最终的代码终于出炉，并且在天梯赛中取得了不错的成绩。

4.2 经验与教训

经验：在本次大作业的过程中，我们组分工明确，各司其职；在代码方面进行多次的模拟与改进，并且在和他人的对战中得到了许多启发，有利于代码的优化。

教训：1.分工明确后导致不同类型成员的交流比较少（~~例如我这写报告的对于具体的对战详情知之甚少，甚至连代码都自己看了好久~~），有点浪费时间，还是需要各成员之间多沟通交流。
2.在进行分工协作创建程序的过程中，需要更多的对自己的代码进行注释，将其主体思想以及一些细节体现出来，增加其可读性。

4.3 建议与设想

本次二零四八大作业，陈斌老师在组队和基础设施代码方面都非常好（应该），不过竞赛方面可以采取得分制晋级的方法，赢一个队伍得一分这种，要不然刚好碰到被克制的队伍就惨了（比如说我们）。

对明年选择数据结构与算法的学弟学妹们：数算这门课虽然要学的东西一大堆，而且可能有难度，但是其中有许多的乐趣的，而且学好了 python，以后的学习将受益无穷！

对后续工作的设想：可不可以模拟一个坦克大战一样的游戏，双方对战，还有 ai 坦克的这种。

5 致谢

1. 感谢陈斌老师一学期来的数据结构与算法课程的教导，以及对于算法思路的提点。
2. 感谢助教们以及技术组同学们对二零四八对战平台的贡献。
3. 感谢所有组员们对本次大作业的辛勤贡献，特别是写代码的三位同学的熬夜奋战，感谢大家的努力。

6 参考文献

（列出实习过程中用到的参考资料、网站链接等）

https://github.com/pkulab409/sessdsa.2048
http://www.icourse163.org/course/0809PKU037-1003471009

数据结构与算法课程实习作业报告

肖襄* 贺婉卓 刘懿阳 李泓德 赵霄美

摘要：

2048 本质上可以抽象成信息对称双人对弈模型，移动策略仅依赖对接下来格局的推理。使用核心算法为对弈模型中常用的带 Alpha-beta 剪枝的 Minimax。这个算法也常被用于如国际象棋等信息对称对弈 AI 中。

关键字：minimax 搜索树 Alpha-Beta 剪枝 DFS 深度优先搜索

1 算法思想 2

1.1 总体思路 2

1.2 算法流程图 4

1.3 算法运行时间复杂度分析 5

2 程序代码说明 2

2.1 数据结构说明 6

2.2 函数说明 7

2.3 程序限制 13

3 实验结果 13

3.1 测试数据 13

3.2 结果分析 14

4 实习过程总结 13

4.1 分工与合作 15

4.2 经验与教训 17

4.3 建议与设想 17

5 致谢 17

6 参考文献.....17

1 算法思想

1.1 总体思路

【总体思路】

2048 对战游戏可以抽象成一个信息对称的双人对弈模型，任一时刻对弈双方对格局所掌握的信息完全一致，移动策略仅依赖对接下来格局的评估及推理。将每一种格局组织成一棵树，树的每一个节点表示一种格局，而父子关系表示由父格局经过一步操作可以到达子格局，故可以通过以当前格局为根的格局树搜索来确定下一步的最优选择。实现的重点有三个，对局面形势的评估、向后搜索本身以及对超时、报错等异常情况的处理。评估操作需要尽可能准确反映局势优势，搜索操作需要在规定时间内尽可能加大搜索深度，这样得到的选择会更优。

另外，我们的对战中包含落子和移动两种情况的决策，但由于针对落子情况的搜索选择太多耗时较长，而针对移动情况的搜索只有四个方向且能得到比较明显的结果差异，故我们选择对落子“粗暴处理”，而把宝贵的时间用在性价比更高的加大移动方向选择的搜索深度上。

【主要数据结构和算法策略】

数据结构：

我们使用了一种相当于自动构建和析构的树状结构，但是我们没有必要存储每一个点的内容，所以我们没有特意地主动地去建树，但我们依然利用了这种思想

算法策略：

AI 采用的主要算法为带 Alpha-beta 剪枝的 Minimax，是一种信息对称 AI 对弈的经典格局树搜索算法：将每一种格局组织成一棵树，树的每一个节点表示一种格局，而父子关系表示由父格局经过一步操作可以到达子格局，故可以通过以当前格局为根的格局树搜索来确定下一步的最优选择。Minimax 假设对方拥有完美决策能力，每一次都能够将我方引入从当前看理论价值最小的格局方向，故我方的策略就是选择那些对方所能达到的让我方最差情况中最好的。故它是在搜索深度限制内寻找局部的、让己方损失最小的最优解。如总体思路中所述，我们在移动方向上采用此搜索算法，而对落子情况只采取“优先在己方落子，否则随机在对方落子”的策略。另外，还有控制最大搜索深度、准备 naiveAI 函数以应对报错等优化策略。

采用算法：

② Minimax 深度优先搜索算法：用 DFS 算法实现对上述格局树的搜索；

② Alpha-beta 剪枝算法：这是针对搜索过程的优化，目的是在发现不可能找到更优解的情况下及时剪枝，降低搜索耗时（具体见代码分析）；

③ 局面评价函数：依据单调性，平滑性、空格数、孤立空格数以及是否存在被吞并风险来计算本盘面的先手方优势度；

④ 针对超时、报错等的故障处理：

控制搜索深度：根据剩余时间确定最大搜索深度，并当己方时间比对方多时加大搜索深度，反之减少，目的是确保己方不超时的同时充分利用时间；

报错处理：若在评估、搜索或下棋输出时出现报错，则采用 naiveAI 系列算法，它不一定最优但绝对不会报错。

1.2 算法流程图

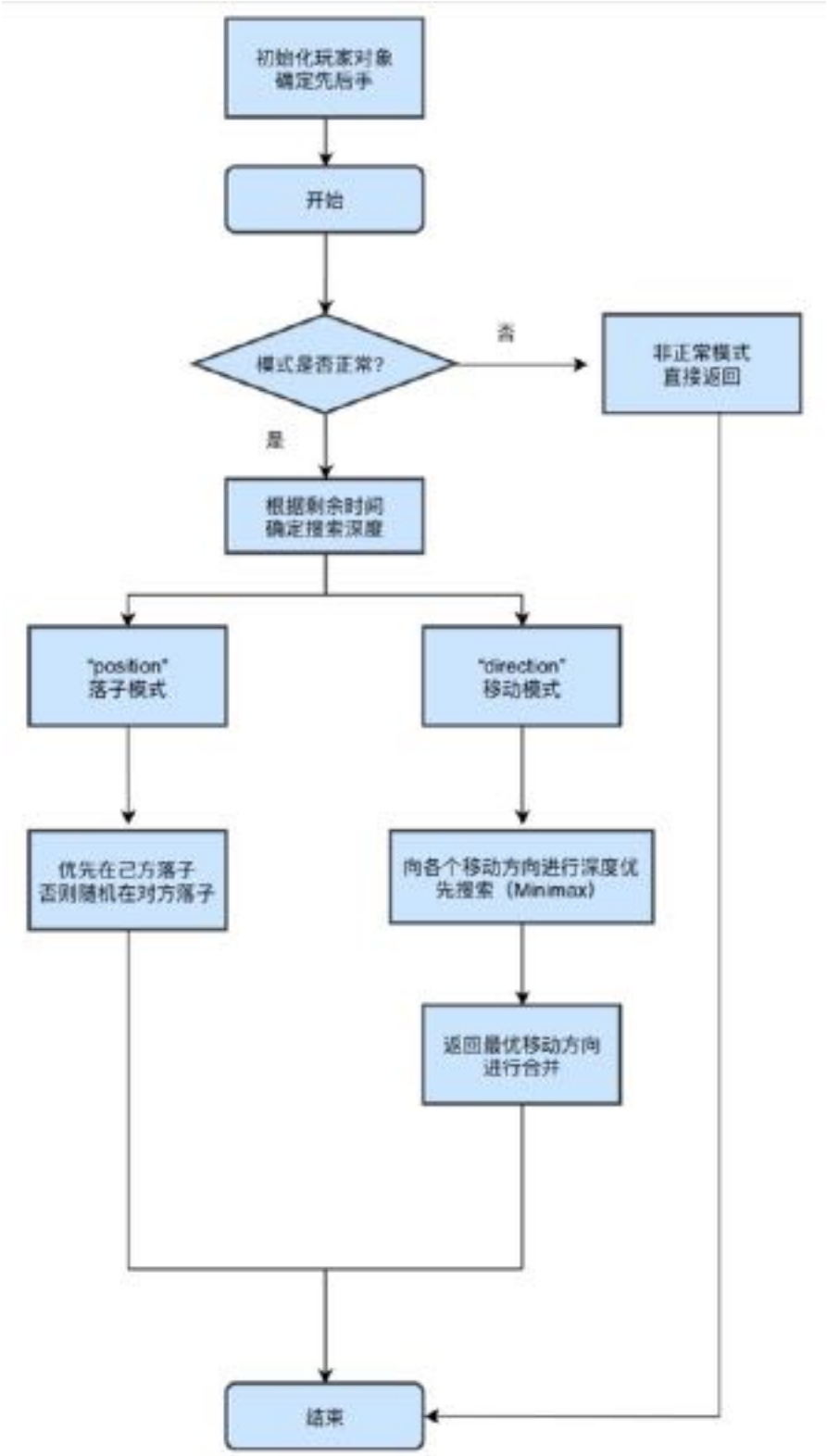


图 1 总算法流程图

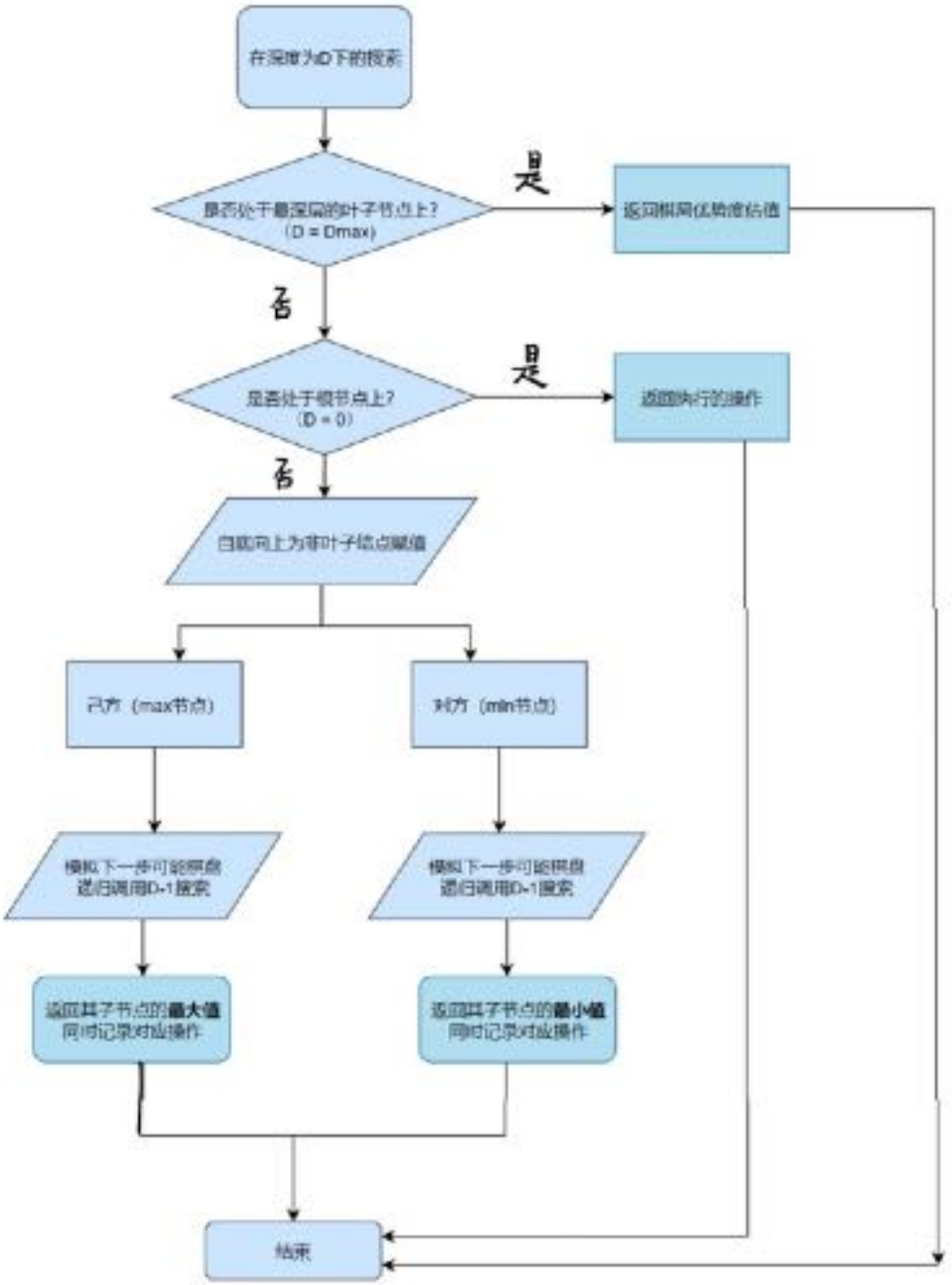


图 2 Minimax 搜索算法流程图

1.3 算法运行时间复杂度分析

从流程图可以看出，为了尽可能获得最大收益，即对局面更有效的预判，我们舍去了对落子位置的分析，因此算法的主要运行时间由合并移动时进行的搜索决定。其中，搜索层数

越深，耗时越长，时间将以 n 的指数级别增长。在搜索程序中，我们加入了 Alpha-beta 剪枝，以减少无效搜索，提高效率。另外，在实验中我们发现，运行时间会随着回合数变化而变化，故如果只设定单一的搜索深度，无法充分利用时间。改进后的算法根据剩余时间来确定最大搜索深度，动态地充分利用时间。并且为了保证不超时，当己方时间比对方多时可以加大搜索深度，若少时也相应降低搜索深度。

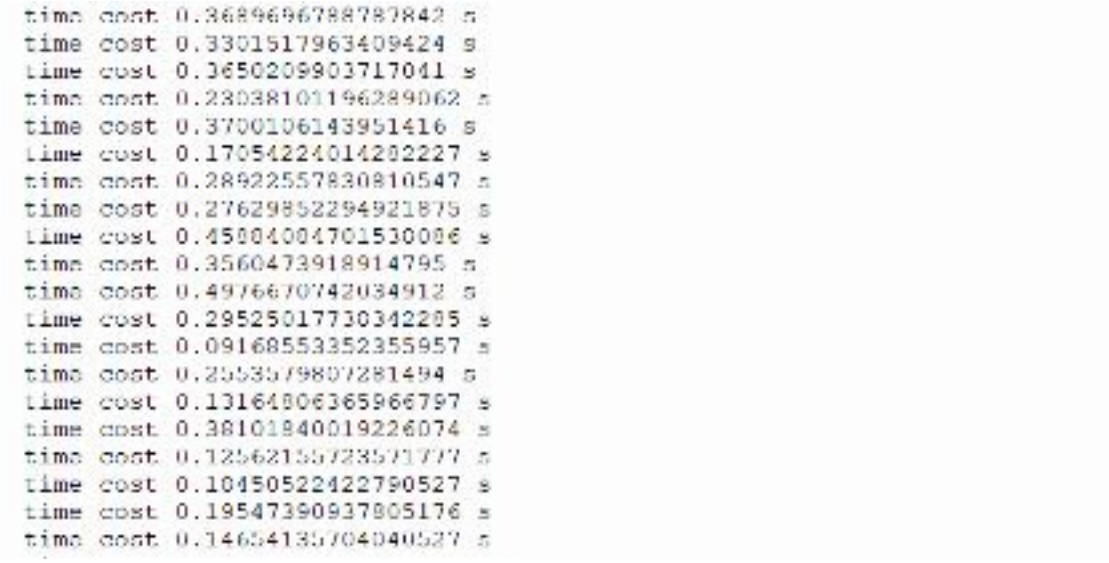


图 3 各回合运行时间差异较大

2 程序代码说明

2.1 数据结构说明

在我们的算法中，主要采用的是线性的数据结构，用列表构建棋盘，存储结果。搜索的时候用了树。没有定义其他类，只根据题目要求定义了 Player 类：

```
class Player:
    def __init__(self, isFirst, array): # 初始化玩家对象
        self.isFirst = isFirst
        self.arr = array
        self.maxRound = min(500, len(array))
```

里面包含了三个参数，是先手还是后手，随机序列，对战局数。

2.2 函数说明

（1）output()交互输出函数，接受输入'position'返回下棋位置，或者接受输入'direction'返回合并方向。首先确定搜索层数，根据时间动态变化搜索深度。

```
    def output(self, currentRound, board, mode):# self 是字符串形式
        # 非正常的 mode 直接返回
        if (mode != 'position') and (mode != 'direction'):
            return

        # 确定搜索的深度，动态的搜索深度有助于充分利用时间
        remainingTime = board.getTime(self.isFirst)
        if remainingTime<0.1:
            return self.naiveOutput(currentRound, board, mode)
        elif remainingTime<0.8:
            maxDepth = -4
        elif remainingTime<4.0 or currentRound < 40:
            maxDepth = -5
        elif remainingTime>board.getTime(not self.isFirst):
            maxDepth = -6
        else:
            maxDepth = -5
```

曾经想让机器根据程序所剩的时间自动调整搜索深度，但是查找了很多资料，深度优先搜索如何控制总搜索时长并不清楚，代码实现不出来，只能人工规定搜索层数。试了很多盘，在保证不超时的情况下让搜索层数最大，最终得到了这几个数据。接下来复制一个棋盘，考虑落子和合并。

```
        newBoard = board.copy()
        self.originalEval = self.evaluate(board)

        if mode == 'position': # 这里优先下在自己的位置
            another = board.getNext(self.isFirst, currentRound)  # 己方的允许落子点
            if another != (): return another

            available = board.getNone(not self.isFirst)  # 对方的允许落子点
            if not available:   # 整个棋盘已满
                return None
            else:
                return random.choice(available)
```

落子采取了简单粗暴的方法，调用给的随机函数，如果有返回值（己方能下子），则往随机函数给出地方下子，如果无返回值（己方已满）则随机往对方棋盘可行位置下子。在采用这种粗暴落子方法之前我们有尝试过全搜对方棋盘（显然是不合理的），还有启发式找一

些在对面可能落子的特定位置，比如对方出现同等级大子在同一行（列）面对面的情况，但是实现的效果都不好（应该是代码写的不太好）。反而是只在己方下子，留出时间增加搜索深度更好。

```
        if mode == 'direction':
            result = self.search(currentRound, newBoard, self.isFirst, mode, 0, sel
f.originalEval-19260817, self.originalEval+19260817, maxDepth)
            if result == 'flag': # 报错处理
                operateList = []
                scoreList = []
                for i in [0, 1, 2, 3]:
                    newBoard = board.copy()
                    if newBoard.move(self.isFirst, i):
                        operateList.append(i)
                        scoreList.append(self.evaluate(newBoard))
                if scoreList == []:
                    return None
                if self.isFirst:
                    return operateList[scoreList.index(max(scoreList))]
                else:
                    return operateList[scoreList.index(min(scoreList))]
            return result
```

合并采取了搜索算法。用 search()函数采用 minimax 算法，搜索，剪枝。

```
    # 搜索就是最朴素的 Minimax + AlphaBeta 剪枝，这里在'position'节点是
优先在己方下棋的，
    # 这样的好处是减少了'position'节点搜索的浪费的时间，让我们可以多搜几
层
    def search(self, currentRound, board, isFirst, mode, remainDepth
, alpha, beta, maxDepth):
        if remainDepth == maxDepth or currentRound == self.maxRound:
            return self.evaluate(board)
        else:
            if (isFirst == True) and (mode == 'position'): # max 节点
                nextEmpty = board.getNext(True, currentRound)
                if nextEmpty:
                    newBoard = board.copy()
                    newBoard.add(True, nextEmpty)
                    resultCache = self.search(currentRound, newBoard
, not isFirst, mode, remainDepth-1, alpha, beta, maxDepth)
                    if resultCache > alpha: # max 节点尽可能取大
                        alpha = resultCache
                        nextOperation = nextEmpty
```

```
                    if alpha > beta: # 剪枝
                        if remainDepth == 0:
                            if alpha == self.originalEval-19260817:
                                return 'flag'
                            return nextOperation
                        return alpha
                if remainDepth == 0:
                    if alpha == self.originalEval-19260817: # 这个是
没有备选的节点的情况，我们报错
                        return 'flag'
                    return nextOperation
                return alpha

            elif (isFirst == False) and (mode == 'position'): # min
节点
                nextEmpty = board.getNext(False, currentRound)
                if nextEmpty:
                    newBoard = board.copy()
                    newBoard.add(False, nextEmpty)
                    resultCache = self.search(currentRound, newBoard
, not isFirst, 'direction', remainDepth-1, alpha, beta, maxDepth)
                    if resultCache < beta: # min 节点尽可能取小
                        beta = resultCache
                        nextOperation = nextEmpty
                    if alpha > beta: # 剪枝
                        if remainDepth == 0:
                            if beta == self.originalEval+19260817:
                                return 'flag'
                            return nextOperation
                        return beta
                if remainDepth == 0:
                    if beta == self.originalEval+19260817:
                        return 'flag'
                    return nextOperation
                return beta

            elif (isFirst == True) and (mode == 'direction'): # max
节点
                for i in [0, 1, 2, 3]:
                    newBoard = board.copy()
                    if newBoard.move(True, i) == True:
                        resultCache = self.search(currentRound, newB
oard, not isFirst, mode, remainDepth-1, alpha, beta, maxDepth)
                        if resultCache > alpha:
```

```
                            alpha = resultCache
                            nextOperation = i
                    if alpha > beta: # 剪枝
                        break
                if remainDepth == 0:
                    if alpha == self.originalEval-19260817:
                        return 'flag'
                    return nextOperation
                return alpha

            elif (isFirst == False) and (mode == 'direction'): # min
节点
                for i in [0, 1, 2, 3]:
                    newBoard = board.copy()
                    if newBoard.move(False, i) == True:
                        resultCache = self.search(currentRound+1, ne
wBoard, not isFirst, 'position', remainDepth-
1, alpha, beta, maxDepth)
                        if resultCache < beta:
                            beta = resultCache
                            nextOperation = i
                    if alpha > beta: # 剪枝
                        break
                if remainDepth == 0:
                    if beta == self.originalEval+19260817:
                        return 'flag'
                    return nextOperation
                return beta
```

剪枝的方法是，已知在自己选择 A 策略后，对方再走一步，己方的最劣（最小值）为 K(A)。那么假如有 B 策略，选择它之后，对方再走一步己方的最劣（最小值）为 K(B)<K(A)，则直接剪去 B 及其下所有枝。这种剪枝方法建立在双方都足够聪明的基础上，因为站在对方角度，选择对自己最有利的选择必然是对己方最不利的（minimax）。那么相反的，对方选择 A 策略后，己方再走一步，最优（最大值）为 K(a)。如果有 B 策略，使对方选择后，己方再走一步，最优（最大值）为 K(b)>K(a)，那么就直接剪去 b 及其下所有枝。

（2）evaluate()估值函数，评估棋盘局面。总体思路是依据棋子数值大小、整个棋盘的单调

性、平滑性、空格数和孤立空格数，对当前局面打分。正值表示先手方优势，负值后手方优势，绝对值越大优势越大。

```
    def evaluate(self, board): # 计算本盘面的先手方优势度，依据单调
性，平滑性和空格数和孤立空格数
        score = 0

        for chess in board.getScore(True):
            score += 5 ** chess
        for chess in board.getScore(False):
            score -= 5 ** chess
```

首先对己方每个棋子计算 5 的相应次方加入分数中，目的在于鼓励多合成大子。

```
for r in range(4):
            for c in range(8):
                pos = (r,c)
                if self.isDangerous(True, board, pos):
                    score -= 6 ** board.getValue(pos)
                if self.isDangerous(False, board, pos):
                    score += 6 ** board.getValue(pos)
                if board.getValue(pos) >= 1:
                    if board.getBelong(pos) == True and c<4:
                        score -= 32 * self.grad(True, board, pos)
                    elif board.getBelong(pos) == True and c>3:
                        score += 32 * self.grad(True, board, pos)
        return score
```

接下来对棋盘中的每一个位置进行 dangerous 和 grad 罚分。

isDangerous()函数用以判断棋盘中所有棋子中是否有可以被对手的一次划动吞并的，如果有则进行罚分。函数中实际只考虑了自己棋子跟对方相邻并且数值相等的情况，但在同一行不相邻的两个等值棋子，如果中间没有其他棋子也可以被吃的。我们尝试加上对这种情况的罚分，但是代码写不好，参数调不出来，只能放弃。

```
    def isDangerous(self, isFirst, board, pos): # 返回其是否可以被
对手的一次划动吞并
        # 找出可行的相邻点
        directionList = [0, 1, 2, 3]
        for direction in directionList:
            newBoard = board.copy()
            if newBoard.move(not isFirst, direction) and newBoard
.getBelong(pos) != isFirst:
                return True
        return False
```

grad()是一个很简单的梯度罚分函数，用 around()函数返回每个位置周围的位置对应的数值，如果差距太大，说明平滑性不够，则进行罚分。

```
    def grad(self, isFirst, board, pos): # 返回一个脚写的点周围梯度罚分
        discrepancy = 0
        value_of_thisPosition = board.getValue(pos)
        # 找出可行的相邻点
        list_points = self.around(pos)
        # 对每个相邻点添加梯度值
        flag = 0
        for point in list_points:
            value_of_pointPosition = board.getValue(point)
            if (value_of_thisPosition-value_of_pointPosition)< -
3 and value_of_pointPosition != 0:
                flag += 1
        if flag >= 3:
            for point in list_points:
                value_of_pointPosition = board.getValue(point)
                if value_of_pointPosition != 0 and (value_of_thisPosition-
value_of_pointPosition) < -2:
                    discrepancy += 3 ** abs(value_of_thisPosition-
value_of_pointPosition)

        for point in list_points:
            value_of_pointPosition = board.getValue(point)

            if value_of_pointPosition == value_of_thisPosition+1:
                discrepancy -= 3 ** value_of_thisPosition
                break

            if value_of_pointPosition == value_of_thisPosition:
                discrepancy -= 6 ** value_of_thisPosition
                break

            else:
                discrepancy += 3 ** abs(value_of_thisPosition-
value_of_pointPosition)
        return discrepancy
```

代码中加分和罚分的数字依然都是人工~~炼丹~~调试的结果。写了一个 naiveAI，只搜索一层，调整判分参数，看哪个 AI 胜率最好。评估函数中还应该加入斜线上如果是相同的棋子就减分的操作。

2.3 程序限制

最终版 AI 因为采用的是比较保守的方法，所以目前为止在多次测试和实战对局中没有过出错或者超时的现象。

3 实验结果

3.1 测试数据

实验环境说明：

- 硬件配置：
  - CPU：Inter® Core(TM) i7-8550U CPU @ 1.80GHz 2.00GHz
  - 内存：8.00 GB DDR4 2400MHz
- 操作系统：win10 x64
- Python 版本：Python 3.8.3

在对代码进行测试和优化的过程中，主要利用平台让己方 AI 互博，按胜率进行择优，具体分为两步：

- 估值函数优化：先不考虑搜索深度，只进行一层搜索，对估值函数进行优化。计算本盘面的先手方优势度，依据单调性，平滑性和空格数和孤立空格数对局面进行估计，设计了三种方案，naive 1.1, naive 1.2, naive 1.3。其中 naive 1.2 着重从单调性上进行了改进，naive 1.3 着重考虑平滑性。安排三个代码两两进行对战，每次对战 100 局统计胜率，如图 3.1-1 所示，naive 1.1 的胜率明显高于其它两种 AI，性能更好。接下来在 naive 1.1 的基础上进行调节，采取同样的测试方法，结果如表 3.1-1 所示, 确定 naive 1.1.3 为最终估值方法

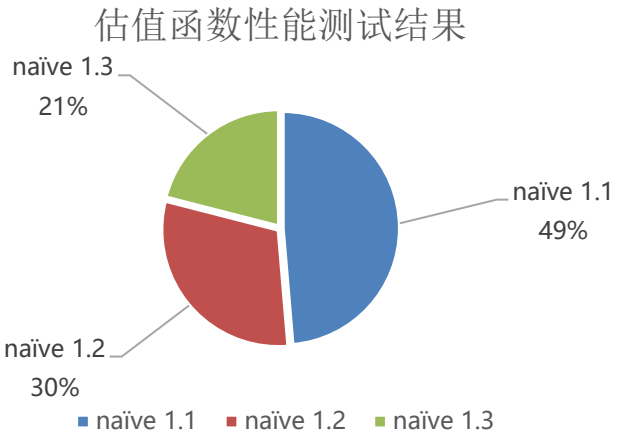


图 3.1-1 估值函数性能测试结果

表 3.1-1 估值函数优化测试结果

	对战结果（1.1.1） 胜出/总局数	对战结果（1.1.2） 胜出/总局数	对战结果（1.1.3） 胜出/总局数	总胜率
Naïve 1.1.1		45/100	39/100	42%
Naïve 1.1.2	55/100		42/100	48.5%
Naïve 1.1.3	61/100	58/100		59.5%

- 搜索深度优化：在估值函数确定的基础上，加入搜索层数，依然采用不同版本 AI 互博的方式进行优化.，最终以减少搜索对方棋盘落子为代价，增大搜索层数.

热身赛结果如下表所示：

表 3.1-2 热身赛结果

名称	对战数	胜出	胜率	等级分	排名	是否出线
currentAI	3490	1802	52%	1032.25	12	否

3.2 结果分析

在热身赛和实战对弈中,主要起作用的是搜索深度的增加：减少搜索对方棋盘落子，只进行必要的启发搜索，以此来降低搜索耗时．同时搜索也是算法的主要耗时步骤，为保证运行不超时，设置了对搜索用时的判断，根据时间来决定搜索深度，保障正常运行．最终将搜索层数增大到 6-7 层．这种策略使得算法在测试中取得极大进步，如图 3.2-1 所示，大幅提

高了胜率，并且出现压制性胜利. 但在最终对战中，出现对部分 AI 取得压倒性胜利，同时被部分 AI 压倒性战胜，说明我们的思路还不够周全，应该在估值函数处理上有进一步提升的空间.

图 3.2-1 算法改进成果

4 实习过程总结

4.1 分工与合作

肖襄

构建了代码的主要的框架

李泓德

优化完善了代码

贺婉卓

完成了实验报告的框架及大部分主要内容

刘懿阳

完成了实验报告的算法分析

赵霄美

完成了代码的性能测试和实验报告的性能分析的部分

5 月 19 日：初步定下实现方案

5 月 26 日：中期交流

6 月 8 日：实验报告的最终核定

4.2 经验与教训

在这次大作业的算法设计和代码实现中，首先我们意识到了一个问题：对每一个位置下棋后评估未必是最优的解决办法。考虑到大部分的时候，都是下在自己的这侧的棋盘是更优的选择，在决策和搜索的时候，我们完全可以默认在本方下棋，这虽然会降低在同样搜索深度下的搜索的结果的“优势度”，但是可以提高'position'节点至最多 17 倍的性能（实际操作的时候平均是 6~10 倍），在这种情况下，我们可以极大地增加搜索深度。

其次，我们认识到了启发式搜索的重要性。题目中设计在对方棋盘上指定位置下棋的功能，其实就是针对对方出现同等级大子在同行或列，马上就可以合并的情况。当出现这种情况时，在对方棋盘下棋阻挡对方合并就是很好的做法，但是应该如何实现？当棋子大到什么程度的时候再去阻挡？都是问题。我们有试过枚举每一行列，看是否有这种情况，调节参数实验，但是发现占的时间多效果也不好，或许也是用启发式搜索会更好一些？还有在写估值函数的时候，我们是人工调参数的，这样费的精力多效果也不是那么好，而且造成了多参数的耦合。如果我们想在估值函数中加入一个新的评价标准，不只要调节新加入的参数，有的时候还要调节旧的参数，才能使效果达到最好。如果用启发式方法可能会更好。

还有我们都认为优化估值函数提升估值效果，不如优化剪枝，增加搜索层数达到的效果好。这一点我们认为是可取的经验（也许取舍本身就是一门艺术吧），舍弃不那么重要的。

4.3 建议与设想

减少 pvp 占分的比重，相应地，可以提高 pve 占分的比重，别这么卷了

5 致谢

感谢 H 姓 19 级巨佬，在我们迷茫的时候给我们指点了方向

6 参考文献

[1] https://blog.codinglabs.org/articles/2048-ai-analysis.html

数据结构与算法课程实习作业报告

黄纵横* 陈一凡 毛思文 叶语锐 张辰玺

摘要：本次实习作业是“二〇四八”，相较传统的 2048 游戏增加了双人对战规则。我们小组整体围绕游戏规则与要求，以树的数据结构形式为整体框架，结合α-β剪枝算法、Minimax 算法和启发式算法进行优化处理，并通过多次实战确定参数设置以期在对战中获得更高的胜率。最终的实验结果可以看出，代码运行状况平稳良好，实验表现符合预期。最后在比赛中我们成绩是第 10 名。

关键字：二〇四八，博弈树，α-β剪枝算法，Minimax 算法

1 算法思想 2
1.1 总体思路 2
1.1.1 关于搜索的基本思路 2
1.1.2 博弈树与 Minimax 算法 2
1.1.3 **α-β**剪枝算法 5
1.1.4 启发式算法 7
1.2 算法流程图 9
1.3 算法运行时间复杂度分析 9
2 程序代码说明 10
2.1 数据结构说明 10
2.2 函数说明 10
2.3 程序限制 13
3 实验结果 13
3.1 测试数据 13
3.2 结果分析 14
3.3 经典战局 15
4 实习过程总结 16
4.1 分工与合作 16
4.2 经验与教训 17
4.3 建议与设想 17
5 致谢 18
6 参考文献 18

1 算法思想

1.1 总体思路

1.1.1 关于搜索的基本思路

棋类博弈问题本质上是搜索问题。搜索，简单来说就是博弈树按照目前情况推演出来的路线，选出一个最优推理路线的过程。搜索直接关系到智能系统的性能与效率，我们都渴望改进结构，使策略最优的同时尽量减少搜索时间。我们当然可以让计算机推演出所有对弈的可能性然后去选择最优走法，这很容易就能想到，但是这种搜索方法要耗费巨额的时间与空间，完全不符合我们实际应用的要求。因此我们一般采用的是另外两种搜索方法：盲目搜索和启发式搜索。我们一般会使用启发式搜索以减少时间复杂度。

启发式搜索一般会根据人类的总结的优秀经验给算法添加启发式信息。一旦我们的棋局符合触发启发式策略的要求，我们就会选择相应的决策方式。因此在应用启发式搜索的时候我们的决策是动态的，更加贴近人类的正常决策思考。

我们在搜索中又分为不同的搜索方式。一般可以大致分为两个基本类：广度优先搜索（BFS）与深度优先搜索（DFS）。BFS 盲目性较大，如果目标节点与初始节点较远 BFS 的搜索效率会很低，因为我们搜索了太多无用节点。DFS 在搜索效率上远远高于 BFS，但是因为搜索的不完备性，可能会导致我们无法获取问题的解，即使问题有解。虽然 DFS 的获取解的准确度没有 BFS 高，但是因为其性能优势，我们一般采用 DFS。

DFS 的缺点也不是不可以解决。为了解决 DFS 搜索不完备的问题，人们提出了有界深度优先搜索方法。这种方法会预先规定一个搜索深度界限作为限制值，当我们触及到这个阈值仍未找到答案时，我们就会去搜索他的兄弟节点。

1.1.2 博弈树与 Minimax 算法

我们在学习分治策略时了解到一个重要思想：解决复杂问题可以通过将问题拆解为多个简单的子问题，然后逐个击破。这些简单的子问题我们称他们为“与”节点。如果我们仍觉得他太难，就可以考虑等价变化，找到这个问题的同构，通过解决同构得到这个问题的解答。这些同构问题构成的节点，我们称他为“或”节点。将上述两种思路结合起来我们就得到了“与/或树”。一般与或树搜索过程如下：

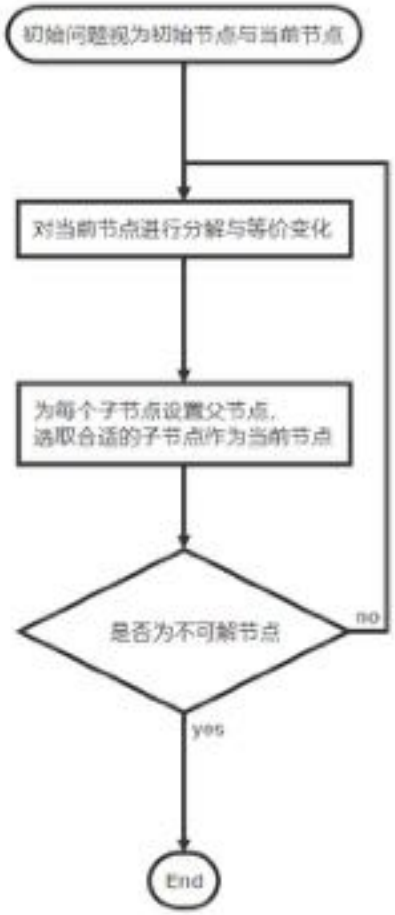


图 1 树的搜索过程

其中，可解节点有如下特征：（1）它是一个终止节点；（2）他是一个“或”节点，且子节点至少有一个为可解节点；（3）他是一个“与”节点，且子节点至少有一个为可解节点。不可解节点就是不满足上述三种条件的节点。

介绍完“与/或”树，我们就可以开始说明博弈树了。棋类博弈是一种简单博弈树，他满足完全信息、两人的零和博弈与非偶然博弈这三种特点。因此，人们在博弈时秉持着完全理性原则，都渴望自己收益最大化，对方收益最小化。我们站在 A 方的视角来看，每一个 A 方的可选策略都是“或”关系，而敌手方 B 的方案对于 A 来说都是“与”关系，因为 B 选择什么 A 并不清楚，他必须考虑所有可能性。我们根据这些信息，构建了一个“与/或”树，也就是博弈树。

博弈树有如下特点：（1）博弈的初始格局是初始节点；（2）博弈树中，“与”、“或”节点是逐层交替出现的。自己一方拓展的节点都是“或”关系，对方拓展的是“与”关系；（3）所有能使自己获胜的都是可解节点，使对方获胜的节点为不可解节点。

根据上述要求，我们自然地引入了一个全新的概念，那就是 Minimax 算法。这种算法的基本思想就是假设对方有完美决策能力，即在假设的权重比较下，对方的每一次决策都能达到对方最优、我方最劣；因此，在对方决策达到其自身收益最大化（我方收益最小化）时，我方在之前一回合的策略选择应使得我方被动收益最大化。我们把我方称为 max 节点，用方块表示；对方为 min 节点，用圆表示。我们给予如下的范例：

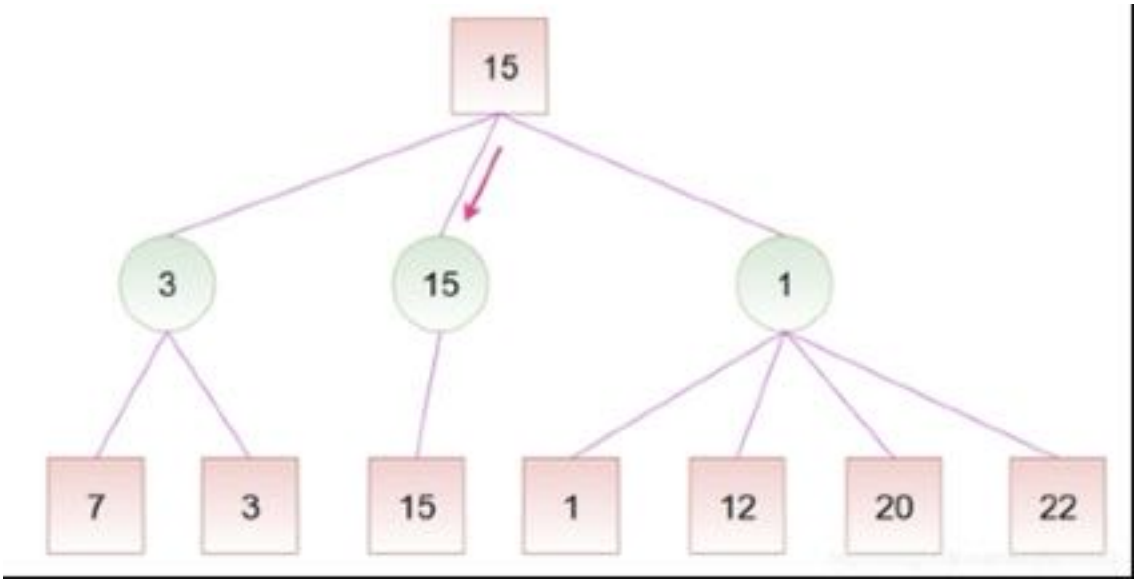


图 2 Minimax 算法范例

我们选择策略时必定选择图中箭头所示路径，因为对方为了使自己收益最大化必定选择居中的路径。

从上述分析可以看出，Minimax 算法不需要寻找理论最优解，因为理论最优解往往依赖于对手是否足够愚蠢，而在 Minimax 中我方完全主动，如果对方每一步决策都是完美的，则我方可以达到预计的最小损失格局，如果对方没有走出完美决策，则我方可能达到比预计的最悲观情况更好的结局。总之我方就是要在最坏情况中选择最好的，在对方的决策失误存在的情况下，我方应该能带到比 Minimax 理论结果更好的实际收益。

同样，在上述理论分析中存在两大问题，导致 Minimax 算法在“二〇四八”对战的实际应用的结果并不特别理想。

a. 过于复杂的完全搜索

在“二〇四八”对战中，一个单位回合包括了四期决策，其中“下棋”阶段（两期）产生 1~17 个分支，“合并”阶段（两期）产生 4 个分支，也即 2^6~2^9 种可能的布局。与常见的 Minimax 应用场景不同，这一数量级的博弈已经较为复杂，在实际实现过程中容易出现超时情况，使得搜索深度被约束在 1~2 轮。由于“二〇四八”的博弈可能是无限期的，即使在之后几回合中达到了胜利分数的最优，也难以顾及多期的影响，包括逐步实现最大棋子“升级”的关键回合，以及考虑棋子排列的工整与对方破坏工整的可能。因此，在后续的算法设计中，我们会将 Minimax 算法作为基本的思路，通过复杂度优化实现更多期的计算，并在局势评估中加入胜利分数以外的评价因子。

b. 衡量收益的困难

Minimax 算法立足于完全信息的理性博弈，但在实际对战过程中双方的理性可是不一致的，这就给每一期收益的衡量产生了较大的困难。我们注意到，除了决定胜负的分值大小以外，棋子的排列，边界领地的争夺，领地空位的剩余，都会对之后的回合产生相当的影响。例如，“下棋”阶段己方领地的“2”所产生的收益确实增加了己方的胜利分数，但在对方领地实行“阻止合并”的“2”事实上能够产生更大的未来收益。这是一期与两期所导致的收益不同。在回合无限而算力有限的情况下，我们不可能达到所有的深度，使得最后一期的胜利分数可知，从而无需考虑棋子的预期价值。因此，应该对期望出现的棋盘布局赋予更大的权重。但是，对战双方所给出的估值权重必然是不一致的，从而同一分支在双方理性决策下的收益也会有较大不同。也就是说，在我方的估值权重下，对方的“理性”是不确定的，对方 min 节点产生的实际选择可能与我方的估计并不相符，也就是一般意义上的“聪明反被聪明误”。在双方估值方式有别的情况下，游戏的回合实际上是不完全信息，Minimax 算法所适用的完全信息条件无法达到满足，进而有着算法失效风险。

尽管存在着上述问题，从本质上说，Minimax 的思路框架是可行的。算法复杂度问题的实质并

不在算法原理上，可以通过各类剪枝方式进一步优化；收益衡量的不确定性可能是必胜策略的来源，即，假设存在一个理想的估值权重，使得结算时的胜利分数正好落在局部最优的区间内——对方的随机行为不会使得终盘结果偏离预测分值太远。当然，这一理想权重目前仅仅是在理论上存在，实际的取值可能需要大量的本地训练。

1.1.3 α-β剪枝算法

Minimax 算法虽然好，但是复杂度简化上还远远不够。因为我们还是会扫描所有子节点才能做出选择。我们希望能够继续缩短时间，此时我们就要使用α-β剪枝算法。

关于剪枝算法，这个算法会有两个值：α和β。其中α代表的是 max 玩家（也就是己方的最佳选择）β 代表的是对方的选择，即 min 玩家。max 节点取最大倒推值，min 取最小倒推值。我们将初始节点设置为α=－∞和β=∞，即最坏的结果，然后在每个节点跟踪并更新α和β值。如果有那个点满足了α≥β，其余分支不用再搜索，进行剪枝。我们这里用一个例子来说明会更快。

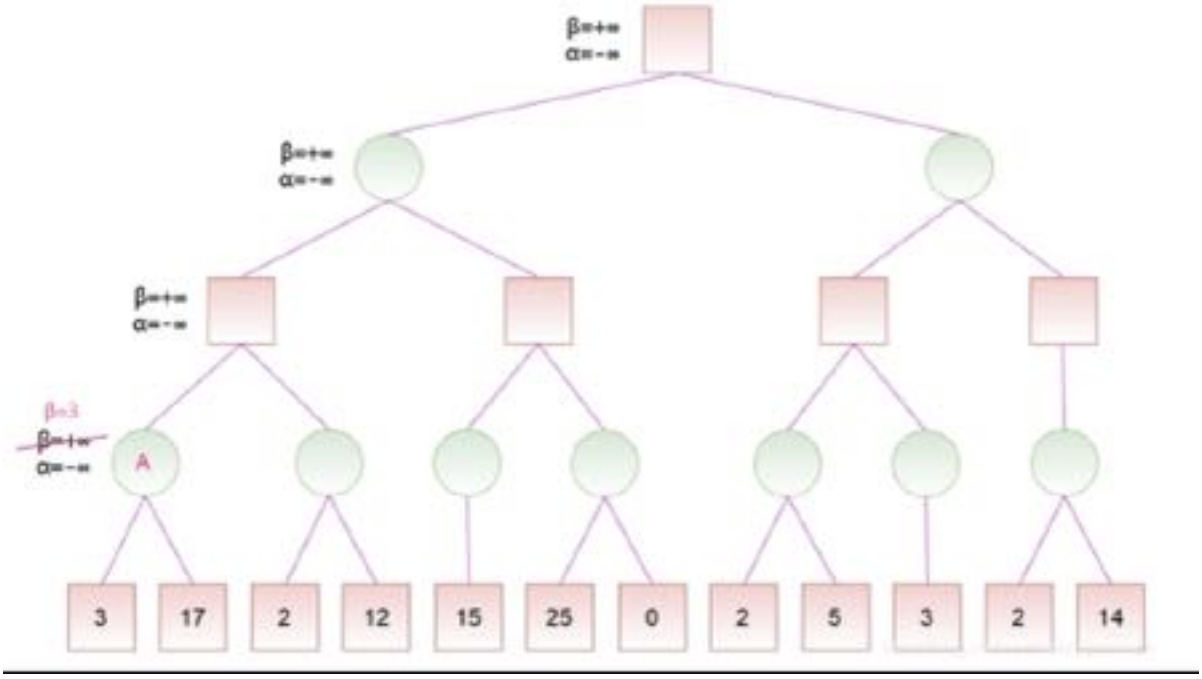


图 3 α-β剪枝算法范例（1）

这里我们仍使用方块表示 max 玩家，圆圈表示 min 玩家。初始化时，先令α=－∞，β=∞，节点记为 v，有－∞≤v≤∞。推到图中 A 节点时，我们进行比对，3 最小，A 改β为 3，A 的节点值赋 3。

A 的上一个节点因为要找寻最大值，3＞－∞，改α为 3。接下来研究 A 的兄弟节点 C：

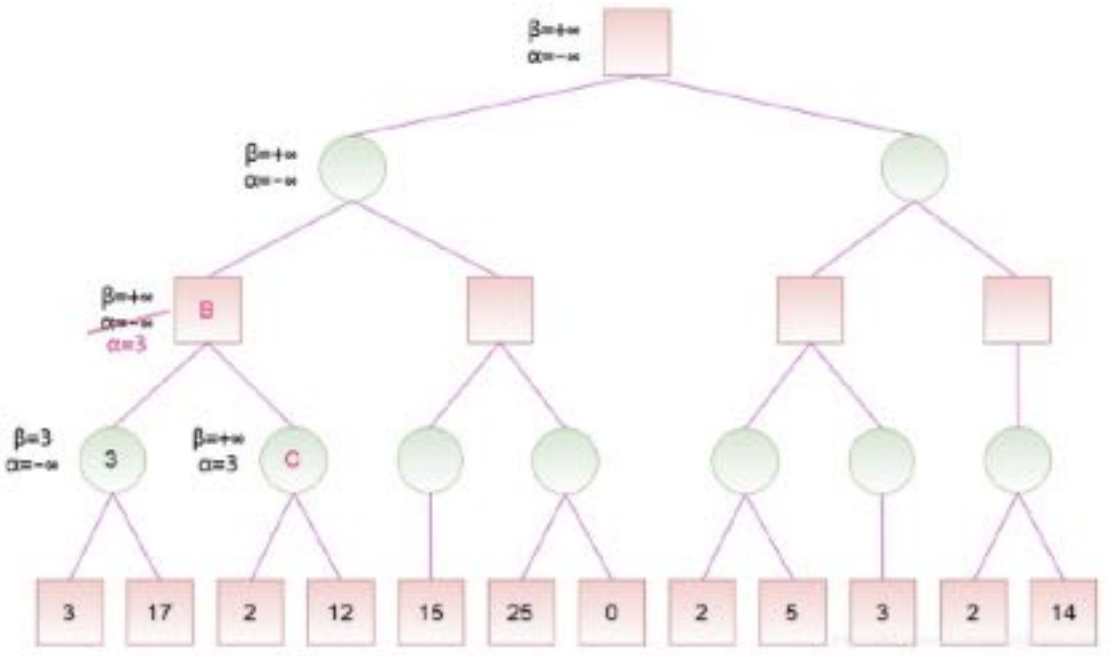


图 4 α-β剪枝算法范例（2）

B 将自己的α和β值传给 C，让 C 进行比较。先扫描到 2 所以 C 的β改为 2，α=3 大于β=2，满足α≥β，终止搜索，对 C 的 12 那个分支进行剪枝，C=2。因为 3>2，所以 B=3。

以此类推，最后得到结果如图所示，H=3：

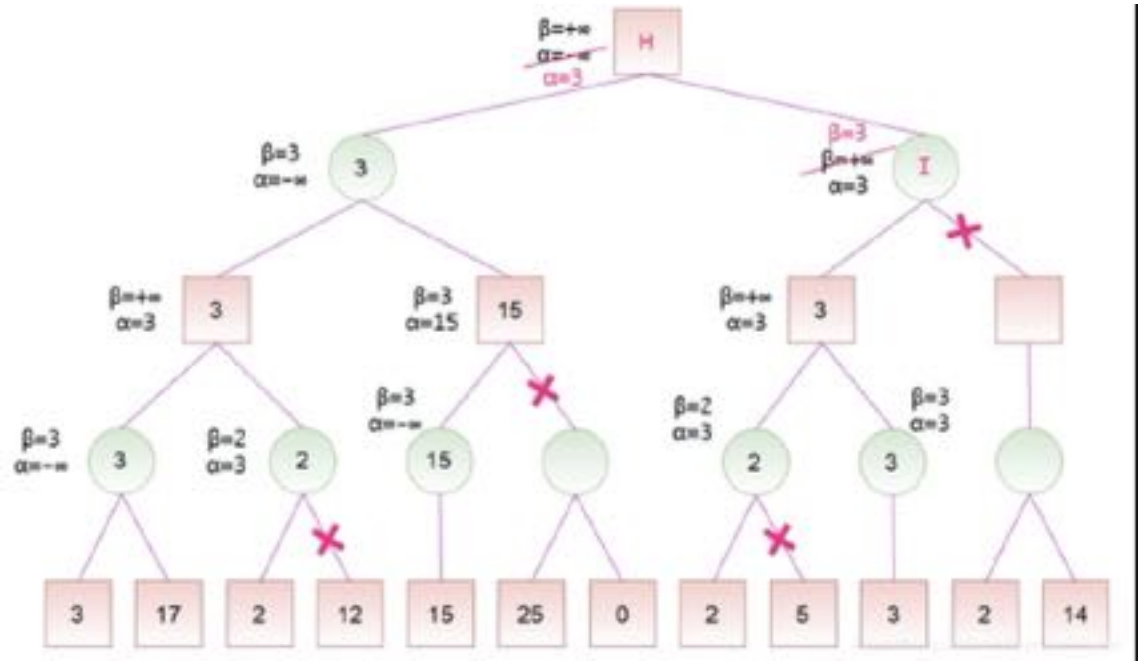


图 5 α-β剪枝算法范例（3）

我们根据这个例子可以看出来剪枝算法为我们消除了多个树的分支，极大简化了算法的流程。

与 Minimax 算法类似，剪枝算法收到估值权重的不确定性影响，应用的有效性有所限制。我们在使用剪枝后，将搜索深度提升到 6 层。

1.1.4 启发式算法

为了最终达到更高的胜率，本小组希望能够通过在剪枝算法基础上并结合一些启发式策略规则模拟人类决策，改进策略，提高胜率。

我们在查阅相关资料并多次比对实战后发现，虽然传统 2048 游戏没有双人对战模式，但仍有很多可以类比的地方，比如传统 2048 有每次滑动随机产生方块的特性，我们可以把它类比对局中对手给我们的棋盘下的棋子。因此这对我们的启发式策略有极大的借鉴意义。目前在 github 上最流行的 2048ai 为我们提供了很多优秀的启发式策略：

（1）单调启发式策略：确保图块的值都沿着左/右和上/下方向都增加或减少

图 6 单调启发式策略中双向完美单调网格

（2）空格启发策略：搜索所有空白格。尽可能使空白格多，这样我们的活动空间才比较大，有更大的空间调整策略

（3）平滑启发：尽可能使相邻的格子数字最为接近，以方便我们合并

图 7 不是很平滑的网格

除此以外，我们还希望改进α-β剪枝算法。根据以往的经验，良好的排序可以让我们剪掉尽可能多的分支，于是我们很感兴趣如何去找到这样的一个排序。历史启发就给予了我们这样的一个方向。以下为历史启发的基本思路：

首先指定两个规定：（1）如果一个节点能够引发剪枝，我们就认为他是一个优秀的节点；（2）如果一个节点没有引发剪枝。对于不同层次的优秀节点，我们要一次计算，得到一个适宜的值。

根据前人的经验，越靠近根节点或叶节点，策略对应的评价越高。靠近根节点的节点评价较高说明它是根据下面的很多子节点综合因素传递导致的，但是靠近叶节点评价较高只是因为情况较少，这样会导致预见性较差。因此，我们一般认为靠近根节点的才是优秀算法。根据 Schaeffer 的经验，我们依据深度给予分数可以帮助我们区分在不同层数下各种算法的优劣（比如有些算法在 n-1 层不是最优的，但是在 n 层是最优的），这个分数一般为 2n 。伪代码如下：

```
int AlphaBetaHH(int nPly, int alpha, int beta)
{
    if(Gamer over)
        return evaluation();     //游戏结束，返回估值

    if(depth == 0)               //叶子节点
        return evaluation();     //返回估值
    SortingMoves();              //根据走法好坏，调整节点顺序

    for(each possible move m)
    {
        MakeMove(m);
        value = -AlphaBetaHH(nply, -1, -beta, -alpha);
        UnMakeMove(m);
        if(value>alpha)          //如果是目前的最优解记录
            alpha = value;
        if(alpha >= beta)        //剪枝
            break;
    }
    SaveHistory(depth);          //按照层次记录走法的好坏
    return alpha;
}
```

图 8 历史启发伪代码

这个算法就是通过搜索找到好的策略，然后记录这个策略。当在其它层次发现相同策略时，就可以根据这样的记录来区别优秀和不优秀的节点。我们就可以将这些节点由优到劣依次排序，得到我们想要的最优算法。

1.2 算法流程图

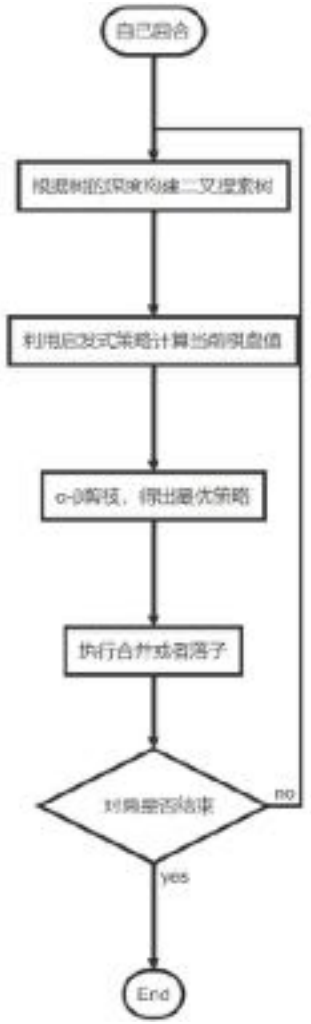


图 9 算法流程图

1.3 算法运行时间复杂度分析

我们使用了二叉树进行了搜索优化，而且α-β剪枝算法的构建树的过程是预设深度的 AVL 树，使搜索过程中变成 O(logn)。但是我们在求取棋盘各个节点的棋盘值的时候是迭代寻找，因此总体而言，算法复杂度为 O(n)。关于每一步的时间，我们能控制在比赛限定的范围时间内，即使在后期对弈，我们也能在 1s 以内的时间做出反应，并且在第四版修改过后速度更快。

2 程序代码说明

2.1 数据结构说明

我们主要使用了树这种数据结构，并且结合α-β算法对树进行改进。在不同的回合下，我们使用了不同深度的树：

```
global maxDepth
if board.getTime(self.isFirst) < 0.5: #如果对局剩余时间少于0.5，改为搜索深度2
    maxDepth = 2
elif board.getTime(self.isFirst) < 3 or currentRound < 40:  #对局时间小于3或者回合数小于40回合，搜索深度为4
    maxDepth = 4
else:   #其余情况，深度为6
    maxDepth = 6
```

图 10 代码数据结构

树的搜索深度越大，我们做的决策也是最有效的。由于开局前期没必要搜索的深度过大，我们设定只有在较后期才需要搜索 6 层。6 层也是我们所能做到非超时情况下最优深度。

2.2 函数说明

（1） nextBestMove

主要用来获取最佳走法，决定是下子还是合并。我们使用了递归的思想。

```
def nextBestMove(self, board, mode, depth, currentRound, alpha, beta, isFirst):
    '''
    获取最佳的走法
    :board:棋盘信息
    :mode:当前的模式（输入的是位置还是方向，如果是位置，进行下棋，否则合并）
    :depth:深度
    :currentRound:当前回合数
    :alpha:剪枝的alpha值
    :beta:剪枝的beta值
    :isFirst:是否先手，bool
    '''
```

图 11 nextBestMove 函数参数设置

获取走法策略使我们应用剪枝算法的地方。这里我们改进了传统 2048 的 AI 算法，因为传统 2048 中没有先后手的概念，但是我们这里有。因此先后手的差别，让我们在确定剪枝起点的时候也并不相同。图 10 中我们是先手，所以要从 min 点回推，最大化得分情况去剪枝；图 11 中我们是后手，因此要从 max 点去回推剪枝。但是无论如何，总的情况不变。

```
if not available: # 如果我们没有可行方案，结束回合
    alpha = max(alpha, self.nextBestMove(board, 'position', depth+1, currentRound, alpha, beta, not(isFirst)))
for position in available:
    tempBoard = board.copy()
    tempBoard.add(isFirst, position)    #存已方落子
    alpha = max(alpha, self.nextBestMove(tempBoard, 'position', depth+1, currentRound, alpha, beta, not(isFirst)))
    if beta <= alpha: # 剪枝
        break
```

图 12 nextBestMove 函数先手情况

图 13 nextBestMove 函数后手情况

（2） boardValue

这个函数主要用来推出棋盘的值，方便我们在 nextBestMove 中调用，去构建树。这里用到了四个启发式策略：分别是单调启发（图 13）、平滑度启发（图 13）、空格启发（图 14）和棋盘值启发（图 14）。需要注意的是，因为我们要考虑先后手关系，所以棋子的先后手的所属所改变的平滑度等各种指标是反向的，先手子可能考虑更多的空位和棋子的最大值，后手子则是一旦面对这些问题就是反向的变化，但是会更注重平滑度的成分。这是我们在不断地对对局侦查后得出的一个比较优良的结论。同时，我们结合了历史启发策略，及时记录了一些步数，加快了我们进行最优决策的时间。

当这个函数有 preCheckList 这个函数时，我们会在对方的棋盘去寻找最优空位，如果说我们找到了这个位置，我们会返回这个空位，并在 nextBestMove 中予以保存，方便我们去和自己棋盘的位置的操作进行比对，得出最优答案（图 15）

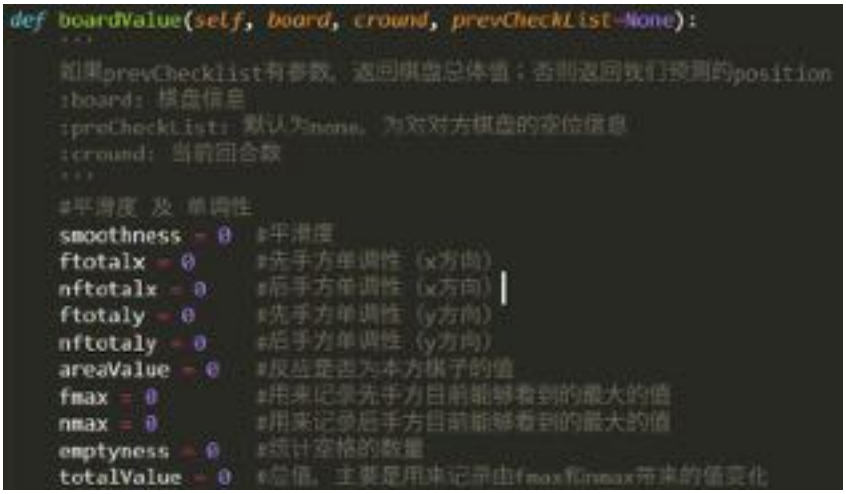


图 14 boardValue 函数参数设置

图 15 单调启发和平滑度

```
belong = myBoard[x][y][1]     #返回棋子归属
if belong:  #以下是对先手的棋盘的棋子情况统计
    areaValue += 1  #对区域值记1
    if myBoard[x][y][0] == 0:    #如果是空值
        emptyness += 1
    else:    #如果不是空值
        fmax = max(fmax, myBoard[x][y][0])
        totalValue += 2 ** myBoard[x][y][0]
else:    #如果不是先手方棋子，反向处理空值记和最大值
    areaValue -= 1
    if myBoard[x][y][0] == 0:
        emptyness -= 1
    else:
        nmax = max(nmax, myBoard[x][y][0])
        totalValue -= 2 ** myBoard[x][y][0]
```

图 16 空格启发和棋盘值启发

```
maxValue = fmax ** maxValueFixWeight - nmax ** maxValueFixWeigh
value = smoothness * smoothWeight\
        + monotonicity * monoWeight\
        + emptyness * emptyWeight * (cround ** 0.5) / 5\
        + maxValue * maxWeight\
        + totalValue * totalWeight\
        + areaValue * areaWeight
```

图 17 出现 preCheckList 函数时的处理

（3） directionCompare

主要核查移动方向的问题。我们在合并操作时会去比较各个方向的移动去选出一个最优移动，在平滑度和单调性比较完成之后，我们给出一套合并方向，提供给 nextBestMove 去进行合并的棋盘值计算（图 17）。

```
def directionCompare(self, isFirst, board):
    smoothx = 0 #x方向的平滑度
    smoothy = 0 #y方向的平滑度
    totalx = 0  #x方向的单调性
    totaly = 0  #y方向的单调性
```

图 18 directionCompare 函数参数设置

```
if smoothx >= smoothy:
    if totalx > 0:
        result.append(3)
        result.append(2)
    else:
        result.append(2)
        result.append(3)
    if totaly > 0:
        result.append(1)
        result.append(0)
    else:
        result.append(0)
        result.append(1)
else:
    if totaly > 0:
        result.append(1)
        result.append(0)
    else:
        result.append(0)
        result.append(1)
    if totalx > 0:
        result.append(3)
        result.append(2)
    else:
        result.append(2)
        result.append(3)
return result
```

图 19 合并方案

2.3 程序限制

搜索的层数还是偏少，虽然 6 层已经很多，但是很多大牛都完成了 9 层甚至是 10 层的搜索。

3 实验结果

3.1 测试数据

实验环境说明：

- 硬件配置：Intel® Core™ i5-8300H CPU @ 2.30GHz 2.30GHz 8.00GB
- 操作系统： Windows10 专业版
- Python 版本：Python 3.7.4

本组代码更迭版本如下，最终采用 V4 参加天梯赛。

版本	修改情况	修改结果
初稿	- 用了α-β剪枝和启发函数	- 实现初步打败白痴算法的效果（胜率九一开） - 启发函数较粗糙，参数单一，后续可以增加跟多维数 - 权重选择也参考了原版 2048 的调优，还没针对对战进一步优化

		- 只能向后推两期（半个回合），否则会超时报错，有待剪枝
V0	- 增加注释，改动一些细节 - 加了几个参数，增加对棋盘的预检测 - 提前剪掉一些分支，可以勉强达到 4 的深度，但不太稳定（300 回合左右超时）	- 参数的选择不合适，实际对战中，经常给对手送子，有待调整
V1	- 加入了窗口算法，以及一些细节优化	- 窗口算法并没有实际效用 - 实盘战斗中胜率下降（理论上胜率应该一致），有待查明原因
V2	- 参考了历史启发 - 参数略有调整 - 考虑非线性的估值 - 发现联网对战可能慢于本地	- 用时大概减少了 15% - 参数调整效果仍然不好
V3	- 估值函数里自己建立了棋盘，减少调用时间 - 修复了总是给对手下子的 bug - 增加了动态深度，现在最大深度为 6	- 修改效果明显 - 后续主要工作是改善估值函数
V4	- 再次调整了一些地方，特殊处理了关键棋子升级的 bug	- 人工测试不会再超时

3.2 结果分析

在代码 V1 版本中，我们加入了窗口算法，并进行了一定的细节优化，但在实战过程中发现用时并没有降低，胜率反而低了（理论上胜率应该是一致的），因而该方法并不适用于我们的代码，故而舍弃。

V2 版本参考了历史启发，用时减少了近 15%，效果显著，该算法思路已在 1.1.4 中详细介绍，它帮助我们的代码更具有智能性，模拟人类决策，找到更好的策略，因而十分符合该竞赛项目的目的和要求，也在实战中取得较好的效果。

此外，为了进一步提升算法效率，留更多的时间给搜索深度上，我们在估值函数中自己建立了棋盘，减少调用，成功将原先深度 4 且有可能超时的不利情况转化为最大深度为 6，并调整了一些细节处尽力避免偶发性的超时失误，实战结果很好地符合了预期。

从实战结果上可以看出，我们的算法风格平稳，前几版本由于搜索深度的缺陷在天梯上排名一般，但后续的几次改进都有显著的效果，排名上升迅速。但也正是由于算法不很激进，胜率一直徘徊在 40%-50%之间，很难取得更高的名次。

另一个后劲不足的原因可能在于参数调试的方面，由于缺乏一定的调参经验和理论的支撑，我们的改善参数的进展一直不太顺利，暴力调参也很难达到预期的效果，这也是在未来需要做更深入的了解和学习的地方。

3.3 经典战局

图 20-23 经典战局（1）

在此战局中，我方为后手方，交界处我方有两个相邻的 32，而对方有一个 64，这种情况是我方吞并对方 64 棋子的一个很好的机会，我们的算法也达到了这样智能的反应，选择向下移动合并 32 棋子，给对手方留出一个空地便于我方下一步的进攻，对手方也如我们所预期的进入这个空格，我方成功合并吃掉对方棋子生成 128。

图 24 经典战局（2）

将对手棋盘彻底堵死也是一种战略选择。例如此战局中，我方为先手，我方选择进攻性更为强的策略，挤占对方棋盘，大量吞噬对方棋子，并在较早的时刻就将对手彻底堵死（俗称“塞爆”），随后的战况由进度条中的趋势可以看出，我方利用该优势迅速合成大量大数棋子，而对方战局也因此一蹶不振。

4 实习过程总结

4.1 分工与合作

小组分工情况如下：
黄纵横：组织会议，分析对战，报告撰写
陈一凡：编写代码
张辰玺：编写代码
毛思文：调整参数，报告撰写
叶语锐：报告撰写，报告整合

受疫情影响，本次实习过程中的讨论会议均在线上完成，以微信群和微信语音通话为平台进行较为高效的沟通与交流。

以下为每次会议的具体内容和工作安排：

第一次会议	5.16	完成初稿，敲定代码大致架构
第二次会议	5.17	确认小组分工
第三次会议	5.24	沟通近期各小组成员任务完成情况； 确认报告撰写分工；

部分微信群讨论截图如下：

图 25-28 微信群讨论截图

4.2 经验与教训

本次大作业过程中我们小组分工明确，各成员按照自己擅长的领域积极完成自己份内的任务，并乐于分享资源，共同解决难题，努力朝着有利于小组任务完成的目标贡献自己的力量。即使在疫情期间，大家无法当面沟通交流想法，仍然能够通过微信平台表达自己的观点和意见，无特殊情况不缺席任何一次会议，严谨真诚地对待每一个组内 ddl，因而作为一个小组大作业，能够起到合作共赢的效果对大家就已经是一次宝贵且有意义的经历。

当然在前期，我们小组也遇到一些障碍，比如沟通上不及时导致任务安排的不够高效，初稿代码胜率不高等。但我们小组都及时发现了这些问题，并且积极地去解决他们，不断完善代码，在修改代码后反遭胜率降低的情形下依然不急躁不放弃，寻找问题的根源。

本次实习作业需要提升的地方有：（1）工作效率和沟通效率可以进一步提高（2）可以安排更多的人手在调试更优的参数上。

4.3 建议与设想

此次大作业的设计很有创意性，基于传统的 2048 游戏规则并将其改为双人对战，占领对方格子等设计也很细致有趣。整个大作业进程和竞技场等系统总体比较完善，竞技场的设置也便于大家了解自己代码的状况，有利于进一步修改完善。

今年特殊情况大家被迫线上合作大作业其实也是一大挑战，如何做好沟通也成为此次大作业一大难点。因而感觉今年大作业准备时间略显仓促，规则的最终敲定也花费了不少的时间，期末季大家时间大多比较紧张，所以建议老师能较早地公布大作业规则，明晰大作业中需要完成的各项任务，便于大家提前进行准备和思考，也利于大家呈现更高质量的代码。

寄语：数据结构与算法是一门不可多得的好课，对于编程小白来说也是一门硬核课。可能学习

过程中有时会觉得困难重重，但涅槃之后一定会对数据结构知识体系拥有一个基本而牢固的架构，这会对未来各领域的学习提供良好的基石。

5 致谢

感谢陈斌老师一学期的辛苦的直播教学，您的充满干货、幽默风趣、精彩纷呈的课堂以及本人可爱和善、耐心细致的性格让能够我们享受于这门课程。

感谢袁泽、陈天翔、冀锐、张赖和、杨佑铭、魏江南助教对这门课的付出，耐心解答同学们的疑问。同时感谢助教和技术组的同学为我们搭建了大作业平台，让大家顺利完成各项任务。

感谢所有组员积极配合，勤勤恳恳做好了每一项工作。

6 参考文献

[1] https://blog.csdn.net/qq_31615919/article/details/79681063
[2] https://blog.csdn.net/qq_32767041/article/details/80634244
[3] https://github.com/ovolve/2048-AI
[4] http://blog.codinglabs.org/articles/2048-ai-analysis.html
[5] 王骐. 博弈树搜索算法的研究及改进[D]. 浙江大学.

数据结构与算法课程实习作业报告

刘子琦*　李凌霄　　丁若愚　　荀鼎　　王锦

摘要：基于传统 2048 单机游戏修改而来，在通过 python 语言构造的平台上的对战版本 ai 算法，设计的主要目标为在五百回合的对战中合成较大的数字战胜对手。基于此，我组主要编写了合并时的搜索树，其中用到 minmax 搜索树及 Alpha-Beta 剪枝，并引进对局面的评估函数。最终，我组同学分工合作，完成了 ai 的设计，并参与天梯对战，取得不错的成绩。

关键字：搜索树　深度搜索　Alpha-Beta 剪枝　估值函数

1　算法思想.....1
1.1　总体思路.....1
1.2　算法流程图.....2
1.3　算法运行时间复杂度分析.....4
2　程序代码说明.....4
2.1　数据结构说明.....4
2.2　函数说明.....4
2.3　程序限制.....12
3　实验结果.....13
3.1　测试数据.....13
3.2　结果分析.....14
3.3　经典战局（可选）.....14
4　实习过程总结.....15
4.1　分工与合作.....15
4.2　经验与教训.....17
4.3　建议与设想.....17
5　致谢.....17
6　参考文献.....17

1 算法思想

1.1 总体思路

1.1.1 综述

算法的整体思想是基于一个给局面的评分函数和一个搜索树，通过搜索树完成搜索后利用评分函数选择对自己最有利的局面，从而实现对局面的控制。

1.1.2 关于搜索树及 Alpha-Beta 剪枝

Minimax 算法常用于棋类等由两方较量交替落子的程序。算法使用 DFS 来遍历当前局势以后所有可能的结果，通过最大化自己和最小化对手的方法获取下一步的动作，实际上就是树形结构的递归算法。MAX 层代表己方，其节点值为子节点的最大值。MIN 层代表敌方，其节点值取决于子节点的最小值。在实现上，首先要确定最大搜索深度 depth，在深度为 detpth 的叶子节点上用 evaluate 函数对其估值，然后自下而上将每个节点赋值，在根节点获得取得 MAX/MIN 值的路径即位我们所做出的决策。由于 Minimax 算法随着深度增加计算耗时呈现指数级膨胀，因此因为 Alpha—Beta 剪枝算法。

Alpha-Beta 剪枝算法用于减少 Minimax 算法在搜索树中评估的节点数。该算法维护两个值，即 alpha 和 beta，分别表示 Max 层的下界和 Min 层的上界。基本原则为：

- 当一个 Min 节点的 β 值≤任何一个父节点的 α 值时 ，剪掉该节点的所有子节点
- 当一个 Max 节点的 α 值≥任何一个父节点的 β 值时 ，剪掉该节点的所有子节点

1.1.3 关于评估函数

为实现搜索到最优解，我们选择引入评估函数对于局面加以判断。评估函数的中心思想即为选择合适的描述局面的指标，并对这些指标赋予不同的权重。

我们选择了平滑性、单调性、空位数、最大值和总分值五个指标进行描述。平滑性是指每个方块与其直接相邻方块数值的差，其中差越小越平滑；单调性指方块从左到右、从上到下均遵从递增或递减；空位数顾名思义，即为局面中某方的空位，显然，空格越少的一方在对局过程中更容易“无路可走”；最大值判据与本对战基础胜利条件一致；总分值是我们对于局面的一个综合考量，兼有鼓励合成的作用。其中，最大值指标最为重要，其他指标的权重则在不断的调试中逐渐确定。最终得到一个较为优越的评估函数。

1.2 算法流程图

1.2.1 对战中每一回合对战流程图，见图 1.2.1

图 1.2.1：每一回合对战流程图

1.2.2 搜索树深度搜索及调用评估函数流程简图如下图 1.2.2

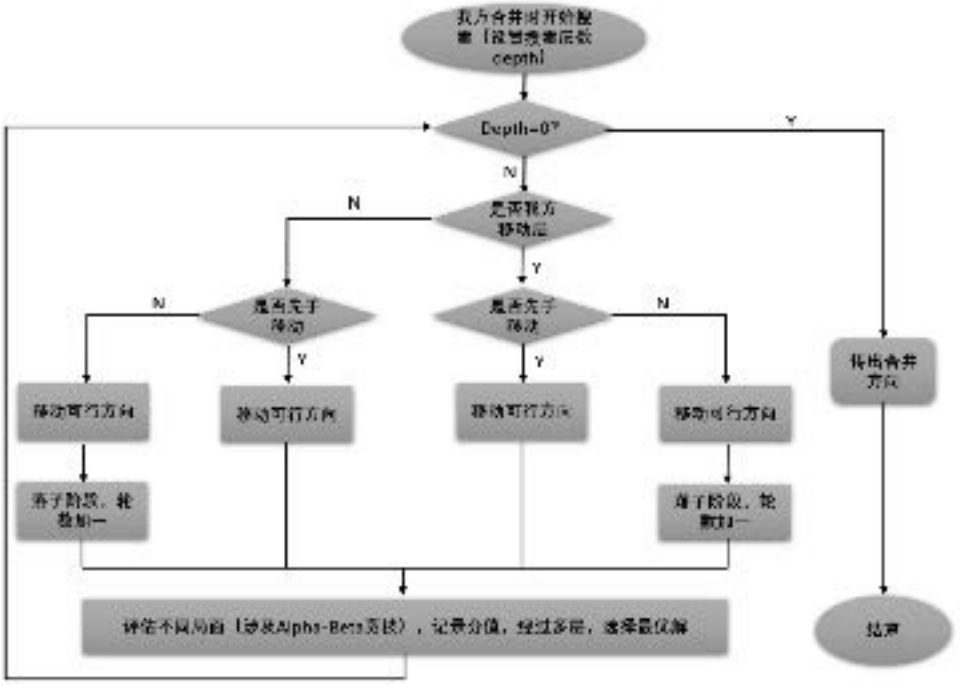


图 1.2.1：搜索树深度搜索及调用评估函数流程简图

1.3 算法运行时间复杂度分析

本算法的运行时间复杂度可以拆分为几部分来分析，首先是对搜索树的分析，搜索的每一层所需要继续向下搜索的最大可能性是四种，考虑到无法移动的情形，实际可能性小于四种，但是总体而言仍显著是大于 1 的。因此，不考虑 alpha-beta 剪枝的对搜索层数的复杂度平均而言是$O(a^n)$ (1<a<4)。而 alpha-beta 剪枝从数学期望上而言大致可以做到对其中一定比例的搜索点进行剪枝，但是仍然改变不了整体指数级别的复杂度。所以整体对于搜索层数的复杂度仍然保持$O(a^n)$ (1<a<4)的情况。

接下来是对评估函数的复杂度分析，这需要对其中各个函数进行评估。

copyboardlist 函数，统计时对每个棋子进行遍历，所以对棋子数 n 的复杂度为$O(n)$。

getsmooth,getmono,getvaluescore 函数都是对每个棋子进行遍历，但是后两者都是遍历一次，所以也是$O(n)$级，而 getsmooth 由于每两个相邻的棋子都要进行一次计算，而实际上每个棋子最多与四个棋子相邻，并且考虑到剔除重复以及 2048 本身游戏特点会使棋子集中在一起，所以也是$O(n)$级。

而 fevaluate 中对棋子进行统计的部分是$O(n)$级，调用上述三个函数也是$O(n)$级，所以总体而言还是$O(n)$级。

对于总评估函数 evaluate，一共调用两次 copyboardlist 和 fevaluate，所以考虑上述结论，对棋子数仍然是$O(n)$级。

由上面的分析，两部分算法各自的复杂度分别是搜索对层数的$O(a^n)$级和评估函数对棋子的$O(n)$级，所以对于一个搜索层数为 h 棋子数为 n 的搜索局面，总复杂度为$O(n \cdot a^h)$级

2 程序代码说明

2.1 数据结构说明

本算法主要采用字典、列表、集合及元组等各种数据结构。字典主要用于对盘面信息按照（位置：等级）记录和改变；列表与集合分别用作等级与位置信息的收纳，在评估时更为方便；元组则是作为 minmaxsearch 函数的返回值出现。以上数据结构信息的具体用法在 2.2 函数说明部分有更多的介绍。

2.2 函数说明

2.1.1AI 接口函数

2.1.1.1 __init__函数

此函数为 Player 类的初始化函数。

参数:isFirst 表明是否先手,为 bool 变量；参数 array 随机序列,为 tuple 变量。

代码如下：

```
1.  def __init__(self, isFirst, array):
```

```
        # 初始化
        self.isFirst = isFirst
        self.array = array
```

2.1.1.2 output 函数

此函数为回合中己方的执行决策函数。

参数 currentRound 当前轮数,与对战平台类变量 self. currentRound 遵循相同约定；参数 board 为棋盘对象的拷贝,Chessboard 类；参数 mode 为模式:mode = ‘position’对应位置模式,mode = ‘direction’对应方向模式。

返回值:位置模式下返回下棋的坐标,方向模式下返回合并的方向。

我组将此函数依据 mode 的不同分为两类操作：位置模式下优先落子于己方领地，若己方领地无空位则随机落于对方领地；方向模式下利用 minmax 搜索寻找局部最优解，此时会调用搜索树 minmaxsearch 函数。

代码如下：

```
    def output(self, currentRound, board, mode):
        # 输出己方决策的函数

        # 在落子模式下优先对自己方进行落子，然后如果满了就去对方落子
        if mode == 'position':  # 给出己方下棋的位置
            another = board.getNext(self.isFirst, currentRound)  # 己方的允许落子点
            if another != ():  # 如果己方允许下棋优先在己方下
                return another

            available = board.getNone(not self.isFirst)  # 对方的允许落子点
            if not available:   # 整个棋盘已满
                return None
            else:
                shuffle(available)
                return available[0]

        # 在合并模式下利用 minmax 搜索得到最优合并方向
        else:
            # 控制搜索层数
            searchdepth = min(5, ((500 - currentRound) // 2) * 2 + 1)  # 搜索深度预设为 5，同时防止临近末尾 round 超过 500
            restime = board.getTime(self.isFirst)
            if (restime - 0.1) / 5 < (500 - currentRound) / 500:  # 如果时间使用过多则降低搜索层数
                searchdepth = min(3, ((500 - currentRound) // 2) * 2 + 1)
            if currentRound >= 499:  # 对临近末尾进一步降低层数
                searchdepth = 1

            # 调用函数进行搜索
```

```
        searchResult = minmaxsearch(
            board, self.isFirst, self.isFirst, currentRound, searchdepth)  # 进行 minmax 搜索
        return searchResult[1]
```

2.2.2 搜索树 minmaxsearch 函数

搜索树采用 Minmax 算法，依次搜索我方与敌方可能策略，给出最优解，具体的，搜索树的构建参考前文 1.1

参数 board 为棋盘对象的拷贝，Chessboard 类；参数 myfirst 表示是否轮到自己所走层，为 bool 类；参数 first 表示是否轮到先手方合并，为 bool 类；参数 currentRound 当前轮数,与对战平台类变量 self. currentRound 遵循相同约定；参数 depth 为当前剩余搜索层数，为 int 类；参数 alpha 与 beta 用于 Alpha-Beta 剪枝，为 float 类。

返回值：此函数返回一元组，元组首项为搜索后结果估值，数据类型为 float；次项为搜索得到的合并方向，数据类型为 int。

由于每一回合由落子和移动两个阶段组成，先后手时对搜索树的处理略有不同——搜索后手时需评估移动及下一回合落子之后的局面，参数 first 特为此而设置；而参数 myfirst 则是为区别本方与对方以适用 minmax 算法而设。

此函数调用了 evaluate 函数用以评估及 addchess 函数用以实现先后手操作差异。

代码如下：

```
def minmaxsearch(board, myfirst, first, currentRound, depth, alpha=- float('Inf'), beta=float('Inf')):
    # 目前仅仅引入了对移动的搜索树，由于对落子的搜索过于繁琐而且收益不高，所以暂时没有考虑

    if depth == 0:  # 搜索到指定深度后返回评估
        # 为了退到最后一层的时候得到方向，所以要返回方向，这里是为了跟前面形式一致也返回一个值
        return (evaluate(board, myfirst), None)

    # 在深度没有到达指定层数时继续进行搜索
    else:
        # 为了避免 alphabeta 剪枝过程受到极端局面影响而效果不好，对搜索顺序做随机处理
        directionlist = [0, 1, 2, 3]
        shuffle(directionlist)

        # 进行下一层搜索
        if first == myfirst:  # 如果是搜到轮自己走的层，对应 max 节点，取搜索 max 值
            action = None  # 对默认行动方向的初始化
            nowvalue = - float('Inf')  # 对默认分数的初始化
            if first:  # 如果是搜到先手方移动，接下来进入后手方移动环节，不用落子直接搜索
                for direction in directionlist:  # 遍历四个方向
                    newboard = board.copy()  # 对棋盘做 copy
                    if newboard.move(first, direction):  # 如果新的棋盘可以做合并
                        searchResult = minmaxsearch(
```

```
                            newboard, myfirst, not first, currentRound, depth - 1, alpha, beta)  # 递归调用搜下一层，此时轮数不变，深度减一
                        value = searchResult[0]
                        if value >= nowvalue:  # 如果值大于当前值，更新结果
                            nowvalue = value
                            action = direction
                        alpha = max(alpha, value)  # 计算 alpha 值
                        if alpha >= beta:  # 引入 alpha-beta 剪枝
                            break
            else:  # 如果搜到后手方移动，在移动完成后还要加上落子部分，然后才能再搜下一轮的先手方移动
                for direction in directionlist:
                    newboard = board.copy()
                    if newboard.move(first, direction):
                        # 在移动后对先后手方分别落子，然后再进行下一层的搜索
                        if currentRound < 499:
                            addchess(newboard, True, currentRound + 1)
                            addchess(newboard, False, currentRound + 1)
                        searchResult = minmaxsearch(
                            newboard, myfirst, not first, currentRound + 1, depth - 1, alpha, beta)  # 此时由于已经完成一轮，目前轮数要加一
                        value = searchResult[0]
                        if value >= nowvalue:
                            nowvalue = value
                            action = direction
                        alpha = max(alpha, value)
                        if alpha >= beta:
                            break
            # 如果四个方向都没有移动，就没有值覆盖原有的 action，返回方向为 None，符合要求
            return (nowvalue, action)
        else:  # 搜对手一层同理，对应 min 节点，取搜索 min 值，上下两段唯一区别就是换了换下面的地方的大于号
            action = None
            nowvalue = float('Inf')
            if first:
                for direction in directionlist:
                    newboard = board.copy()
                    if newboard.move(first, direction):
                        searchResult = minmaxsearch(
                            newboard, myfirst, not first, currentRound, depth - 1, alpha, beta)
                        value = searchResult[0]
                        if value <= nowvalue:
                            nowvalue = value
```

```
                            action = direction
                        beta = min(beta, value)
                        if alpha >= beta:
                            break
            else:
                for direction in directionlist:
                    newboard = board.copy()
                    if newboard.move(first, direction):
                        if currentRound < 499:
                            addchess(newboard, True, currentRound + 1)
                            addchess(newboard, False, currentRound + 1)
                        searchResult = minmaxsearch(
                            newboard, myfirst, not first, currentRound + 1, depth - 1,
    alpha, beta)
                        value = searchResult[0]
                        if value <= nowvalue:
                            nowvalue = value
                            action = direction
                        beta = min(beta, value)
                        if alpha >= beta:
                            break
            return (nowvalue, action)
```

2.2.3 局面评估相关函数

2.2.3.1 evaluate 函数

此评估函数用于评估局面分数，帮助搜索树寻找最优解。
参数 board 为棋盘对象的拷贝，Chessboard 类；参数 first 表示是否为己方，为 bool 类。
返回值：两者分数之差（局面分数），为 float 类。
此函数调用了 copyBoardlist 函数分割棋盘并各自调用 fevaluate 函数评估分数。
代码如下：

```
def evaluate(board, first):
    # 评估函数

    # 对棋盘进行 copy
    myboardlist = copyBoardlist(board, first)
    opboardlist = copyBoardlist(board, not first)

    # 对自己和对方各自进行评分
    myScore = fevaluate(myboardlist)
    opScore = fevaluate(opboardlist)

    # 返回两者分数之差
```

```
13.     return myScore - opScore
```

2.2.3.2 fevaluate 函数

此函数用以评估 evaluate 函数中分割的单方局面分数，对其中不同维度的设置思路及遍历算法前文 1.1 中已有具体说明。

参数 boardlist 为分割而成的单方局面，为 dict 类。

返回值：单方局面分数，为 float 类。

此函数调用了 getvaluescore 函数、 getsmooth 函数、getmono 函数以计算局面分数。

代码如下：

```
1.  def fevaluate(boardlist):
2.      # 针对具体某个玩家的评分
3.
4.      # 不同评分维度的评分权重
5.      sommthWeight = 1.0  # 平滑性权重
6.      monoWeight = 1.0  # 单调性权重
7.      emptyWeight = 1.0  # 空位数权重
8.      maxWeight = 20  # 最大值权重
9.      scoreWeight = 1.0  # 总分数权重
10.
11.     # 初始化
12.     emptynum = 0
13.     maxlevel = 0
14.     chesslist = set()
15.     valuelist = []
16.
17.     # 遍历棋盘进行统计
18.     for pos in boardlist:
19.         if boardlist[pos] == 0:
20.             emptynum += 1
21.         elif boardlist[pos]:
22.             chesslist.add(pos)
23.             valuelist.append(boardlist[pos])
24.             if boardlist[pos] > maxlevel:
25.                 maxlevel = boardlist[pos]
26.
27.     # 计算评分
28.     return getsmooth(boardlist, chesslist) * sommthWeight + \
29.            getmono(boardlist, chesslist) * monoWeight + \
30.            emptynum * emptyWeight + maxlevel * maxWeight\
31.            + getvaluescore(valuelist) * scoreWeight
```

2.2.3.3 getvaluescore 函数

此函数用以得到一方计算当前局面总分值，以代入 fevaluate 函数中参与计算。
参数 valuelist 为一方盘面棋子级别表，为 list 类。
返回值：单方总分值，为 int 类。
代码如下：

```
def getvaluescore(valuelist):
    # 对分值进行评分
    sum = 0
    for val in valuelist:
        sum += val ** 2
    return sum
```

2.2.3.4 getsmooth 函数

此函数用以进行盘面平滑性的评分，以代入 fevaluate 函数中参与计算。
参数 boardlist 为分割而成的单方局面，为 dict 类；参数 chesslist 为棋子坐标集，为 set 类。
返回值：单方平滑性值，为 int 类。
代码如下：

```
def getsmooth(boardlist, chesslist):
    # 对平滑性进行评分

    # 初始化
    smoothness = 0
    checkedlist = set()
    nearlist = [(-1, 0), (1, 0), (0, -1), (0, 1)]

    # 遍历棋子，计算求和
    for pos in chesslist:
        checkedlist.add(pos)  # 对已经计算过的不重复计算
        for move in nearlist:
            newpos = addmove(pos, move)  # 得到临近位置的坐标
            if isInBoard(newpos) and boardlist[newpos] != 0:
                if not newpos in checkedlist:
                    if not boardlist[newpos]:
                        smoothness -= abs(boardlist[pos])  # 与对方格子相邻
                    else:
                        # 与己方棋子（非空格）相邻
                        smoothness -= abs(boardlist[pos] - boardlist[newpos])
    return smoothness
```

2.2.3.5 getmono 函数

此函数用以进行盘面单调性的评分，以代入 fevaluate 函数中参与计算。

参数 boardlist 为分割而成的单方局面，为 dict 类；参数 chesslist 为棋子坐标集，为 set 类。

返回值：单方单调性值，为 int 类。

代码如下：

```
def getmono(boardlist, chesslist):
    # 对单调性进行评分

    # 初始化
    totalmonolist = [0, 0, 0, 0]
    directionlist = [(-1, 0), (1, 0), (0, -1), (0, 1)]

    # 遍历棋子，进行计算
    for pos in chesslist:
        for i in range(4):
            move = directionlist[i]
            newpos = addmove(pos, move)
            if newpos in chesslist:
                if boardlist[pos] > boardlist[newpos]:
                    totalmonolist[i] += (boardlist[pos] - boardlist[newpos])
    return max([totalmonolist[0], totalmonolist[1]]) +\
           max([totalmonolist[2], totalmonolist[3]])
```

2.2.4 操作辅助函数

2.2.4.1 addchess 函数

此函数用以搜索树 minmaxsearch 中落子阶段的实现，其逻辑等同 output 中落子。

参数 board 为棋盘对象的拷贝，Chessboard 类；参数 first 表示是否为己方落子阶段，bool 类；参数 currentRound 当前轮数,与对战平台类变量 self. currentRound 遵循相同约定。

此函数调用 board.add 实现落子。

代码如下：

```
def addchess(board, first, currentRound):
    # 逻辑同默认 ai，优先在自己方下，己方下满下对方，全下满的话就不动棋盘
    another = board.getNext(first, currentRound)  # 己方的允许落子点
    if another != ():
        board.add(first, another)  # 如果可以在自己方落子优先在自己落子
    else:
        available = board.getNone(not first)  # 对方的允许落子点
        if available:   # 若整个棋盘还没满
            board.add(first, available[0])  # 在对方挑一个点落子
```

2.2.4.2 isInBoard 函数

此函数用以判断某个位置是否在棋盘内。
参数 pos 表示坐标，为元组类型；返回值为 bool 类型。
代码如下：

```
def isInBoard(pos):
    # 判断某个位置是否在棋盘内
    if pos[0] < 0 or pos[0] > 3:
        return False
    if pos[1] < 0 or pos[1] > 7:
        return False
    return True
```

2.2.4.3 addmove 函数

此函数用以移动操作，将某个移动加到某个位置得到新位置
参数 pos、move 均为元组类型；返回值同样。
代码如下：

```
def addmove(pos, move):
    # 将某个移动加到某个位置上得到新位置
    return (pos[0] + move[0], pos[1] + move[1])
```

2.2.4.3 copyBoardlist 函数

此函数用以复制单方棋盘，以进行后续的估值操作。
参数 board 为棋盘对象的拷贝，Chessboard 类；参数 first 表示此方，bool 类。
返回值：boarddict 代表单方棋盘，为 dict 类。
代码如下：

```
def copyBoardlist(board, first):
    # 将某一方的棋盘 copy 下来
    boarddict = {}
    for i in range(4):
        for j in range(8):
            if board.getBelong((i, j)) != first:
                boarddict[(i, j)] = False
            else:
                boarddict[(i, j)] = board.getValue((i, j))
    return boarddict
```

2.3 程序限制

本算法虽然实现了方向模式下的智能化，但对于落子模式仍然采取了较为简单的决策。

实际上，算法设计过程中，我组成员认为对落子的优化成本收益比例过大，没有在代码中实现此处的更进一步。当然，结合资料与对局过程，我们也提出了一些优化方案，在 4.3 建设与设想中有更进一步的说明。

测试过程中主要修复 bug：搜索树未按照预期调用评估函数，搜索树在临近 500 轮时搜索轮数超过 500 轮的报错，在搜索层数降为 1 时返回值的报错。

在极端条件下可能的问题：本 ai 采用的搜索层数控制最低搜索层数为 3，因为以目前主流 CPU 在三层下时间都最够充裕，但是如果运行环境实在太差还是有可能出现总思考时间大于 5s 的问题的。

3 实验结果

3.1 测试数据

实验环境说明：

- 硬件配置： Intel Core i7-7500U/ 8.00GB RAM
- 操作系统： Windows 10
- Python 版本：3.8.1

测试时主要依托本地调试环境，利用不同 ai 的对战，对参数和算法进行优化，具体以表格在下面列出：

ai 版本	与上一版本变化	目的	局数	对上一版本胜率
1.0	引入评估函数	测试评估函数	400	约 75%
1.1	调节参数	初步调试参数	约 2000	约 90%
1.2	添加新评估算法	优化评估函数	约 2000	约 70%
2.0	引入搜索树	测试搜索树	500	约 90%
2.1	引入剪枝	加深搜索层数	500	约 75%
3.0	调节参数	进一步调节参数	约 5000	约 70%
3.1	搜索层数控制	尽可能利用时间	500	约 60%
3.2	评估函数算法	降低评估时间	500	约 50%
3.3	修复 bug	解决特殊情况报错	500	约 50%

表 3.1.1：测试 ai 版本流程（1.0 版本上一版本为随机 ai）

在调节参数的过程中，最初 1.0 版本使用的是文献中的数值，然后 1.1 版本针对文献数值做了调整改动，主要调节了平滑性和而 3.0 版本在 2.1 版本上进行了更加精细的参数调节，主要方法是利用改动后的 ai 与改动前进行对战，最终每个参数在正负 20%的调节范围内没有出现在 400 局比赛下超过 45-55%的胜率波动。

天梯赛前，主要利用个人账号与官方 ai 进行对战调试，所以热身赛没有留下天梯战绩。

3.2 结果分析

在测试部分，在对战中可以发现，在评估函数的作用下主要完成了对己方局面的把控，增强了己方在己方棋盘内将棋子进行调控安排以便合并的能力。在引入搜索树后主要增强了己方防御对方吃子以及进攻吃掉对方大子的能力。总体而言可以达到在几百局对局数量上完胜随机 ai 的效果，也体现出了这个算法的稳定性。

算法运行时间在预期之中，通过控制时间和搜索层数的配合可以很好完成充分利用时间而不超时的目标。主要时间开销在于计算不同局面的评估函数上。

3.3 经典战局（可选）

采用搜索树之后，在对战过程中我们发现，经常会有所谓“钓鱼操作”，即迫使对方走至我们需要的位置或是以一低级别棋子引诱对方，使对方掉入陷阱。如下图 3.3.1 与图 3.3.2 是两个例子。

图 3.3.1：经典案例一，此刻对方将向下吃掉我方 1024

图 3.3.2：经典案例二，此刻对方仅能向下合并，我方级吃掉其 2048

4 实习过程总结

4.1 分工与合作

4.1.1 小组分工

本着自愿、自主、发挥个人长处的原则，除小组共同参与讨论提出优化外，小组分工如下：

刘子琦：函数主体（包括 Alpha-Beta 剪枝、估值函数）的初步代码实现，参与代码完善和，进行本地和调试，参与报告完善，上传代码；

李凌霄：参与搜索树伪代码的代码化，以个人账号进行代码上传观察运行情况并提出修改意见，寻找代码对落子策略的优化；

丁若愚：提出决策树初期构想，对决策树可能性进行分支，完成决策树伪代码框架；

荀　鼎：参考 minmax 算法进一步优化决策树伪代码，编写 Alpha-Beta 剪枝预处理；

王　锦：依据代码、结果进行报告撰写，整理参考文献，记录小组时间轴、分工。

4.1.2 合作与交流

由于疫情原因，本次大作业组内成员均为线上交流，主要召开了两次研讨会：

5 月 12 日 20:00，腾讯会议。主要讨论内容为规则的熟悉，大体策略的提出以及成员分工，会议记录如下图 4.1.1。

图 4.1.1：第一次组内研讨会

5 月 29 日 08:30，腾讯会议。主要讨论内容为进度的汇报，天梯战局的了解，代码的改善及具体分工核实，会议记录如下图 4.1.2。

图 4.1.2：第二次组内研讨会

此外，组内微信群就代码、策略也有不少实时讨论，各组员之间基于分工不同信息共享也有交流。

4.2 经验与教训

经验：实验过程中考虑过加入启发式搜索：AlphaBeta 剪枝算法严重依赖于每一层节点扫描的顺序，前面节点生成的 alpha 决定了后面节点剪枝的多少。加入针对每一层生成的节点事先进行粗略的排序，（细致的排序需要在进行一次 alpha-beta 剪枝，开销过大无意义。）但未实现，选取了随机避免一些极端情况。

教训：最好还是写一个自动调试程序，开始错误估计了调参数的时间，后面发现实际上人工调试的时间跟直接写一个调参数的程序比起来可能还要长。

4.3 建议与设想

关于算法优化：前文 2.3 中曾提到，此算法对于落子并未做更多考虑。实际上在后期我们也提出了一些优化方案，接近写出一个完整的落子方案。我们以为，落子阶段需要考虑的是落子于己方和对方之间的优劣，以及在对方何处落子。为此，我们准备引入连通块，配合之前存在的平滑性，落子时尽可能使对手的连通块增多平滑性降低，已达到落子时的局部最优解。此部分的初步代码附于报告后的附录一中。

关于大作业与课程：在考虑如何完成大作业的同时，也应该考量我们课堂中学到的“数据结构与算法”该如何与作业有机的结合起来。数算是一种工具，更是一种思维。希望学弟学们也能在数据结构与算法课堂上获益匪浅。

5 致谢

小组实习是数据结构与算法课程的一部分，首先真诚感谢陈斌老师的细致指导，感谢各位助教对课程维护和答疑讲解方面做的贡献。就本次小组实习而言，感谢陈老师、助教、技术组和内测组的同学为平台的开发维护调试所做的努力；感谢参与对战的其他小组，使得我们能够不断地进行算法优化；感谢组长和负责代码完成和测试的同学们勇于攻坚，不懈努力。

6 参考文献

[1]2048AI 设计与实现（CSDN）:https://blog.csdn.net/qq_32767041/article/details/80634244#222048%E6%99%BA%E8%83%BD%E6%8F%90%E7%A4%BA%E5%AE%9E%E7%8E%B0
[2]
[3]陈斌老师数算课 PPT：http://gis4g.pku.edu.cn/course/pythonds/
[4]《数据结构与算法》课本：http://gis4g.pku.edu.cn/course/pythonds/
[5]Minimax:https://en.wikipedia.org/wiki/Minimax#Minimax_algorithm_with_alternate_moves ，Alpha-beta pruning: https://en.wikipedia.org/wiki/Alpha%E2%80%93beta_pruning
[6]一个五子棋 AI 思路:https://kimlongli.github.io/2016/12/14/%E5%A6%82%E4%BD%95%E8%AE%BE%E8%AE%A1%E4%B8%80%E4%B8%AA%E8%BF%98%E5%8F%AF%E4%BB%A5%E7%9A%84%E4%BA%94%E5%AD%90%E6%A3%8BAI/

附录

附录一：落子时的优化代码：

```
def output(self, currentRound, board, mode):
    if mode == 'position':  # 给出己方下棋的位置
        another = board.getNext(self.isFirst, currentRound)  # 己方的允许落子点
        available = board.getNone(not self.isFirst)  # 对方的允许落子点
        if another == () and not available: # 整个棋盘已满
            return None
        else:
            score = []
            for pos in available:
                newboard = board.copy()
                newboardlist1 = copyBoardlist(newboard, not self.isFirst)
                newboard.add(not self.isFirst, pos)
                newboardlist2 = copyBoardlist(newboard, not self.isFirst)
                score.append((pos, tevaluate(newboardlist2)-
tevaluate(newboardlist1)))
            if another != ():
                newboard2 = board.copy()
                newboard2list1 = copyBoardlist(newboard2, self.isFirst)
                newboard2.add(self.isFirst, another)
                newboard2list2 = copyBoardlist(newboard2, self.isFirst)
                score.append((another, tevaluate(newboard2list1) - tevaluate(newboard2
list2)+4.9)) #关于落子的评估函数，我现在用的是考虑 连通块 和 平滑性的办法
            score.sort(key = lambda x:x[1], reverse = True)
            #返回最佳情况
            return score[0][0]

def copyBoardlist(board, first):
    # 将某一方的棋盘 copy 下来
    boarddict = {}
    for i in range(4):
        for j in range(8):
            if board.getBelong((i, j)) != first:
                boarddict[(i, j)] = False
            else:
                boarddict[(i, j)] = board.getValue((i, j))
    return boarddict

```

```
def tevaluate(boardlist):
    chesslist = set()

    for pos in boardlist:
        if boardlist[pos]:
            chesslist.add(pos)

    return (-1)*getsmooth(boardlist, chesslist) + countIslands(boardlist)


def countIslands(opboardlist): #找连通块的函数，用来站在对方的角度评估我方格局，投子的目的是要使别人陷入不利格局，连通块的个数要变多，平滑性要降低
    islands = 0

    marked = {(i, j):True for i in range(4) for j in range(8)}
    for x in range(4):
        for y in range(8):
            if opboardlist[(x, y)] and opboardlist[(x, y)] != 0:
                marked[(x, y)] = False

    for x in range(4):
        for y in range(8):
            if not marked[(x, y)]:
                islands += 1
                mark(opboardlist, (x, y), opboardlist[(x, y)], marked)

    return islands

def mark(boardlist, pos, value, marked):
    if isInBoard(pos) and boardlist[pos] == value and not marked[pos]:
        marked[pos] = True

        directionlist = [(-1, 0), (1, 0), (0, -1), (0, 1)]
        for i in range(4):
            mark(boardlist, addmove(pos, directionlist[i]), value, marked)

def getsmooth(boardlist, chesslist):
    # 对平滑性进行评分

    # 初始化
    smoothness = 0
    checkedlist = set()
    nearlist = [(-1, 0), (1, 0), (0, -1), (0, 1)]

```

```
    # 遍历棋子，计算求和
    for pos in chesslist:
        checkedlist.add(pos)  # 对已经计算过的不重复计算
        for move in nearlist:
            newpos = addmove(pos, move)  # 得到临近位置的坐标
            if isInBoard(newpos) and boardlist[newpos] != 0:
                if not newpos in checkedlist:
                    if not boardlist[newpos]:
                        smoothness -= abs(boardlist[pos])  # 与对方格子相邻
                    else:
                        # 与己方棋子（非空格）相邻
                        smoothness -= abs(boardlist[pos] - boardlist[newpos])
    return smoothness

def isInBoard(pos):
    # 判断某个位置是否在棋盘内
    if pos[0] < 0 or pos[0] > 3:
        return False
    if pos[1] < 0 or pos[1] > 7:
        return False
    return True


def addmove(pos, move):
    # 将某个移动加到某个位置上得到新位置
    return (pos[0] + move[0], pos[1] + move[1])

print(tevaluate({(0,0):0, (0,1):2, (0, 2):0, (1,1):1, (1,2):2,(0,3):0,(0,4):False,(0,5):False,(0,6):False,(0,7):False,\
                (1,3):0,(1,4):False,(1,5):False,(1,6):False,(1,7):False,(2,0):0,(2,1):2,(2,2):0,(2,3):0,(2,4):False,(2,5):False,\
                (2,6):False,(2,7):False,(3,0):0,(3,1):0,(3,2):0,(3,3):0,(3,4):False,(3,5):False,(3,6):False,(3,7):False,(1,0):2}))
```

数据结构与算法课程实习作业报告

胡靖飞* 罗诗蕾 曹彧 霍怡然

摘要：我们小组的算法思想是在进行落子和合并前，先进行计算分析最优结果，再进行决策。整体算法策略是使用三级策略树与剪枝，从叶节点开始对每个节点进行得分计算，从而挑选策略、进行剪枝，输出最优的落子与合并策略。经过多次修改后，在最终天梯过程中，虽没有出现算法出错和超时的情况，但仍需进一步改进。

关键字：决策树　α－β剪枝

1　算法思想.....1
1.1　总体思路.....1
1.2　算法流程图.....2
1.3　算法运行时间复杂度分析.....3
2　程序代码说明.....4
2.1　数据结构说明.....4
2.2　函数说明.....4
2.3　程序限制.....7
3　实验结果.....7
3.1　测试数据.....7
3.2　结果分析.....8
4　实习过程总结.....9
4.1　分工与合作.....9
4.2　经验与教训.....11
4.3　建议与设想.....12
5　致谢.....12
6　参考文献.....12

1 算法思想

1.1 总体思路

二〇四八游戏原型就是我们众所周知的 2048，通过移动随机出现的 2，来使相等的数进行融合，目标是在格子没被占满的情况下，尽可能得到最大的数，即 2048。在本次游戏中，将 2048 改成了回合制 AI 对抗游戏，两队轮流在两个相邻的 4*4 棋盘上进行下棋、合并的操作，并且在边界异色可以同数吞并，最终以最高数的一方获胜。

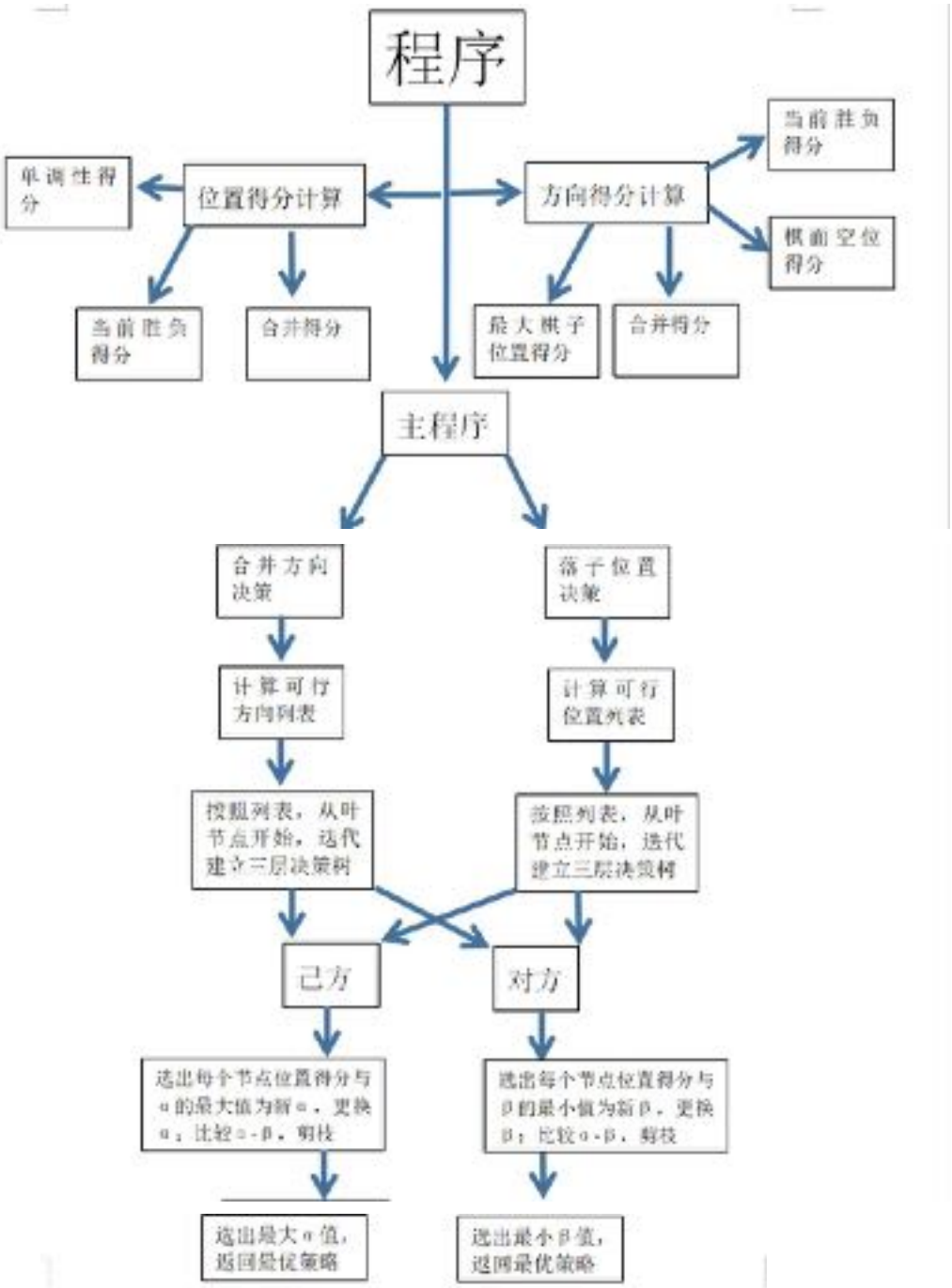
在最开始的讨论中，每位成员结合自己对游戏的理解，都对该游戏的算法策略提供了自己的看法，在第一次讨论后，达成了初步统一，决定采用决策树和贪心策略作为本次游戏的核心算法策略。贪心算法是指所求问题的整体最优解可以通过一系列局部最优的选择，即贪心选择来达到，在本游戏中，利用贪心算法可以尽可能使在己方棋盘中得到较大的数，从而获得胜利；决策树是在已知各种情况发生概率的基础上，通过构成决策树来取净现值的期望值大于等于零的概率，评估项目风险，判断其可行性的决策分析方法，在本游戏中可以利用决策树来分析在选择落棋子时，在己方和对方区域进行落子以及合并时采用方向的最大化收益，从而获得胜利。

在第二次讨论后，在此基础上进行了一些改进，利用决策树为主体策略，增加最基础的 AlphaBeta 剪枝算法和分数算法，这也是我们最终所采用的游戏算法策略。AlphaBeta 剪枝算法是对 Minimax 方法的优化，假设双方都按照自己认为的最佳着法行棋；对给定的盘面用一个分值来评估，这个评估值永远是从一方（搜索程序）来评价的，己方有利时给一个正数，对方有利时给一个负数。从我们的搜索程序（通常把它称为 Max）看来，分值大的数表示对己方有利，而对于对方 Min 来说，它会选择分值小的着法。

我们算法的核心思想还是在落棋子和选择方向合并棋子之前判断各种情况，找到符合己方利益最大的着法和合并方法。

1.2 算法流程图

总算法——分为两部分，落子前分析在己方和在对方落子哪种条件下优势更大；合并前分析朝四个方向哪个方向优势更大。



1.3 算法运行时间复杂度分析

算法在实际运行中的时间开销主要花在计算落子位置和合并方向上。对时间复杂度影响较大的是落子决策函数 finddirection/findposition 中的三层递归，以及估值函数的双层嵌套循环。估值函数的双层嵌套循环为 0（n2），此处的 n 为给定的棋盘长度，三层递归的复杂度为 0（n2），此处的 n 为可行落子列表的最大可能长度。其他程序最大的时间复杂度为 0（n）。因此总的时间复杂度为 0（a2b2），a 为给定的棋盘长度，b 为可行落子列表的最大可能长度。

2 程序代码说明

2.1 数据结构说明

主要用列表、元组等储存信息，使用树结构进行策略判断。特殊之处在于使用了三级策略树的方法，这是对于树的延伸，对于每一个节点有打分，得到节点的α-β值，根据两个值的相对大小进行剪枝简化运算。Player()类中有对于每次下棋的位置与方向选择和策略树的剪枝与赋值，在 Player()类外定义了每个节点的取值计算方式。主要使用的是线性数据结构，使用条件判断、循环结构与函数的嵌套调用实现程序。

2.2 函数说明

	名称	输入	输出	功能	算法策略
Player 类	__init__	是否为先手方 isFirst（bool 变量），随机序列 array		初始化棋盘对象	
	output	当前轮数 currentround,棋盘对象 board，模式 mode（位置/方向）	1）位置模式时，返回 tuple(a,b)代表选择的落子位置 2）方向模式，返回 0-3 的一个数值，分别代表 4 个合并方向	给出己方下棋与合并的决策	位置模式下，如果己方空位置≥14，则选择己方位置落子；如果己方空位＜14，再使用 findPosition 寻找最优落子位置，如果返回结果为空，选择己方落子，如果己方已满，选择对方的第一个空位落子。 方向模式下，如果是下棋的第一步，则寻找所有可能的方向，使用此列表的第一个方向；不是下棋的第一步时，使用 findDirection 寻找最优方向,如果寻找不出来，使用所有可能方向的第一个方向。
	findPosition	当前轮数 currentround,棋盘对象 board，深度 depth，α&β值，是否为己方做决策	在 max 节点输出α值与最优落子位置（元组） 在 min 节点输出β值与对方的最优落子位置	寻找最优落子位置	这里对策略树进行了得分赋值与剪枝，根据得分判断最优落子位置。 这里使用了递归循环的方法。节点为三步预测策略树的最后一步是终止条件，返回此时棋盘的位置得分。如果当前不是三步策略树的最后一步，则使用

		(policymaker：bool)	（元组） 当搜索深度为0（叶节点）时，返回当前局面的估值分数		candidatesPosition 函数得到当前情况所有可行落子点，对于所有可能情况复制当前棋盘对象并落子计算得分，计算得分的方法为：在己方做决策的节点（max）调用自身，得到此节点的子节点的分值 newscore,如果 newscore>α，就替换α，并将其做为当前得分；此外，如果此节点的β≤替换后α，不再计算之后的子节点，直接剪枝，返回当前的分值与位置。 对于对方做决策的节点（min），得到子节点的得分后，将β替换为β与 newscore 中更小的值，得分为替换后β；且剪枝条件变为β≥α。
	FindDirection	当前轮数 currentRound,棋盘对象 board，深度 board,α-β值，是否为己方做决策 (policymaker：bool)	在 max 节点，输出节点的α值与选定最优方向（0-3 的数值） 在 min 节点，输出节点的β值与对方最优方向（0-3 数值） 当搜索深度为0（叶节点）时，返回当前局面的估值分数	寻找最优合并方向	这里对策略树进行了得分赋值与剪枝，根据得分判断最优合并方向。 这里使用了递归循环的方法。节点为三步预测策略树的最后一步是终止条件，返回此时棋盘的方向得分。如果当前不是三步策略树的最后一步，则使用 candidatesDirection 函数得到当前情况所有可行合并方向，对于所有可能方向复制当前棋盘对象并计算方向得分，计算得分的方法为：调用自身，得到此节点的子节点的分值 newscore，如果 newscore>α，就替换α，并将其做为当前得分；如果此节点的β≤替换后α，不再计算之后的子节点，直接剪枝，返回当前的分值与位置。（max 节点） 而对于 min 节点，终止条件与自身调用不变，得到子节点的得分后，替换β为β与 newscore 中更小的值，得分为替换后β；且剪枝条件变为β≥α。
	candidatesPosit	当前轮数 currentRound，棋盘对	列表 candidates	获取此轮可落子的全	使用提供的 getNone 方法得到对方棋盘所有可落子位置，使用提供的 getNext 方法得到己方指定

	ion	象 board,是否为先手方		部位置	的落子位置并合并到 getNone 返回的列表最后一位，列表即为此轮所有可能位置。
	candidatesDirection	当前轮数 currentRound，棋盘对象 board,是否为先手方	列表 candidates	获取此轮可以合并的全部方向列表	可合并方向共有四个，当合并后棋盘发生变化时，就将此方向加入列表，四个方向全部试过后，输出列表
赋值函数	monotonicity	是否为先手方，棋盘对象 board，嵌套列表 boardlist	先手方时，输出数值 diff，后手方时，输出数值-diff	获取当前棋盘对象最大值的单调性情况的得分	首先使用 for 循环找出最大棋子位置，从上下左右四个方面分析，如果棋子未在棋盘边缘并且从最大棋子到棋盘边缘的棋子没有按照从大到小的单调排列时，根据这一列棋子大小改变 score 值，分别计算先手方与后手方的 score，返回值 diff=先手方 score-后手方 score
	maxlocation	是否为先手方，棋盘对象 board，嵌套列表 boardlist	先手方时，输出数值 diff，后手方时，输出数值-diff	获取当前棋盘对象最大值位置情况的得分	用 for 循环判断最大棋子位置，如果在角落得分为 1,在边缘得分为 0.5，都不在得分为 0，用先手方得分减去后手方得分，返回得到的值。
	smoothness	是否为先手方，棋盘对象 board，嵌套列表 boardlist	先手方时，输出数值 diff，后手方时，输出数值-diff	获取当前棋盘相同值相邻情况的得分	使用双层 while 循环，如果是先手方，从左上角开始向右/下搜索邻格是否有相同数值棋子，如果有，对 score 加分，如果是后手方，从右上角向左下搜索，如果有相同数值的相邻棋子就加分，最后 diff= 先手方 score-后手方 score，先手方的返回值为 diff，后手方的返回值为-diff
	emptyspace	是否为先手方，棋盘对象 board	先手方时，输出数值 diff，后手方时，输出数值-diff	当前棋盘对象空格情况的得分	取 score 为一方空间的空格数+1 的对数，输出数值 diff=先手方 score-后手方 score
	chessvalue	是否为先手方，棋盘对象 board	数值 1 或 0	按照当前棋盘对象判断的胜负，获胜为 1，失败为 0	先获取先手方与后手方全部棋子的升序排列列表，逐一 pop 出当前列表最大值，如果某一方 pop 出来的值更大，则此方为胜利；如果直到一方全部 pop 出之前两个列表完全相同，则列表更长的一方获胜。

	position_evaluation	是否为先手方，棋盘对象 board，嵌套列表 boardlist，当前轮数 currentRound	总得分数值 totalscore	棋盘对象位置得分	分别给单调性得分、合并得分、胜负得分各一个权重值，用其权重值乘具体数值，加总为最终得分，（前 400 轮和 400 轮之后的权重值不同）
	direction_evaluation	是否为先手方，棋盘对象 board，嵌套列表 boardlist，当前轮数 currentRound	总得分数值 totalscore	棋盘对象方向得分	分别给最大棋子位置、合并得分、空棋得分、胜负得分各一个权重值，用其权重值乘具体数值，加总为最终得分，（前 400 轮和 400 轮之后的权重值不同）

2.3 程序限制

在我们的测试中，曾经出现过超时的情况，这是由于 α－β 剪枝混淆造成的程序 bug，后期在我们的主程序员修改下无此种问题，在后续的测试与对战中，程序无错误与超时。

3 实验结果

3.1 测试数据

实验环境说明：

- 硬件配置：（CPU/内存）：2.3 GHz Intel Core i5/8GB 2133 MHz LPDDR3
- 操作系统：（名称/版本）：macOS High Sierra/10.13.6
- Python 版本：（版本号）：3.8.0

一开始的程序因为参数设置错误，导致经常在将棋子下到对面的棋盘，而且总会超时，发现原因是因为剪枝的规则有一半写反了，进行了修正之后可以进行一局正常游戏并且不会超时，也不会出现诡异的下棋路线。在一次次测试中最终使得程序能够正常运行。基本上没有参加热身赛，测试算法的方法就是和 AI 比或者 AI 互相比。

3.2 结果分析

在小组分工后，因为主要负责编程主体的同学退课了，而且没有即时通知大家，导致时间有些被延误，同时程序留下了巨大的 bug，如剪枝的初始值设置反了，总是会出现 violate 与超时的问题。所以我们组花了大量的时间在修复 bug 上面，为了使得能够正常的进行一局游戏，而没有足够的时间对于算法的进一步修订，导致参与天梯赛的结果可想而知。下图是我们天梯赛的结果。排在倒数第二，倒数第一是因为程序问题总是报错，导致碰到他是总能 10:0 取胜，而在对于大多数队伍面前，都是只能获得一两局胜利，对于少数队伍能够保证打平或者是微弱劣势输掉比赛。

算法本身在对决中起到所预想的作用，并且在一些劣势局面下可以反败为胜，但是没有对算法进行进一步优化与修订，导致算法所起到的策略停留在最基本的状态，所以在比赛中难以取得胜利。

因为获胜对于我们来说异常艰难，所以称不上是经典战局，在寻找一些比赛后也发现了一些本队表现的亮点。比如下面这场比赛在一直劣势的情况下，吞并对方 1024 实现了翻盘，这是吞并前后的图片对比。

4 实习过程总结

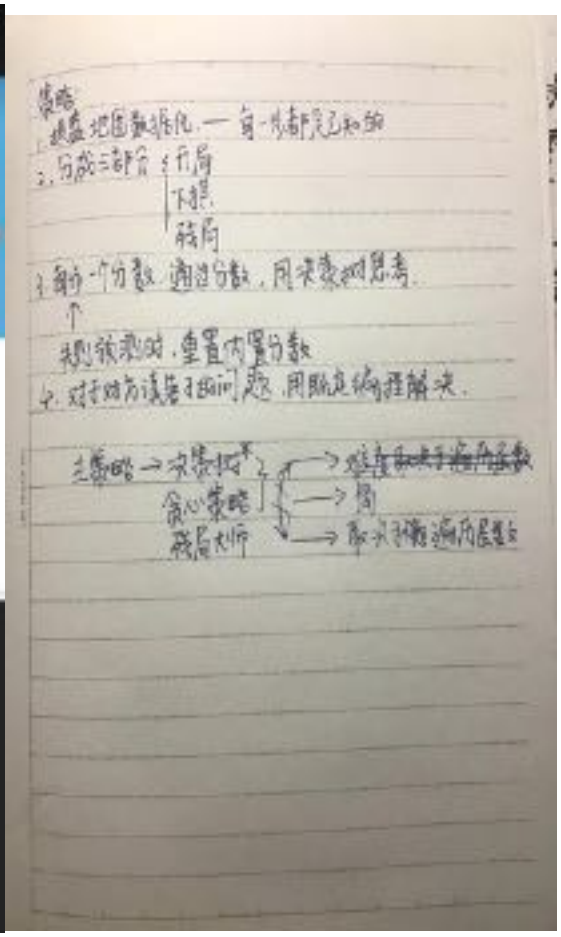
4.1 分工与合作

（1）小组分工

胡靖飞*	1.负责所有除开直接编写代码和书写报告部分的所有工作 2. 进行信息整合以及后期参与 debug 工作。
罗诗蕾	1.代码主体部分的构建，主要负责决策树运行的分数算法以及参数的调试部分。 2. 天梯实验的主要检验者，后期优化的主要人员 3. Debug 主要人员
曹彧	1、实验报告部分编纂工作 2、参与前期讨论算法策略
霍怡然	1、实验报告部分编纂工作 2、参与前期讨论算法策略

（2）合作交流方式包括线上讨论、语音会议与各策划组的交流
（3）组会记录

会议照片



会议笔记

主程序员 debug

4.2 经验与教训

教训：

（1）由于最后的时间相对较紧，在得分的权重方面调整不足，影响了算法的最优判断，也影响了得分。这反映了组内存在时间不足、沟通较少的情况，这件事情也给了我们很大教训，反映了在小组作业中，及时与沟通的重要性。

（2）在算法开发过程中，因为小组内李江一同学中期退课并没有及时和大家沟通，导致相关的部分转交较为匆忙，花费大量时间去更正程序，使得没有时间来更新算法策略。

（3）小组前期分工存在一定问题，所有同学应当适量参与编程或策略制定工作，这样才能更加清楚小组策略和具体代码实现并给予修改意见，否则在编写实验报告上会花费大量时间，而且如果全员参与程序编写，也可能不会出现一人退课导致整体影响过大的局面发生。

经验：

（1）算法中，最优位置与最优方向的判断是根据一个总得分来确定的，这个总得分（方向得分与位置得分），是由诸多小得分乘以权重后加总得到的，最开始的权重是根据几轮的对战经验得到的，后来我们发现，当轮数超过一定值时，当前棋面有多少空位等得分重要性下降，而最大棋子位置得分的重要性上升，因此，我们在轮数大于 400 后更改了权重得分，情况大幅度改善。这说明了多次对战的重要性，因为时间不够，对于权重情况我们没有进行进一步更改，理想情况下，

我们希望能够进行更细致的权重划分，每一段轮数都有一组权重，并且能达到足够精准。
（2）在算法开发过程中，我们进行了多轮更改，第一版没有出现明显 bug，但是运行时间过长，后来成员进一步检查发现是剪枝的赋值混淆导致的问题，第二版的时间明显缩短，但是得分很低，这是由于组内人员对于 zip（）函数的错误使用，导致棋面有一半位置没有遍历到，更改这个问题之后最终得到了第三版。这反映了我们成员出现了一些基础知识上的错误，正是这两个基础问题带来了致命的错误，也反映了基础的重要性。此外，说明我们小组成员的检查细致到位，取长补短、互相学习，也正是小组作业的意义。

4.3 建议与设想

（1）本代码还可以进一步改善：
A）在决策树计算得分时，若对方没有按自己预料的方式落子，则重置内置分数，重新计算
B）通过更多次对战和对游戏的深入分析，引入更多启发式规则，缩短 AI 的思考时间并加大搜索深度
C）minimax 搜索树+alpha beta 剪枝的搜索深度较小，可以尝试蒙特卡洛树等算法
（2）希望可以将大作业的提交与竞赛时间提前，与期末季时间稍微错开。
（3）希望之后能在退课后开始大作业的组队，防止人员变动带来的影响。
（4）希望开放往年优秀组的代码与竞赛过程，这样可以学习多种代码，增加丰富性。

5 致谢

感谢陈斌老师这一学期的教导和各位助教的帮助。
最后感谢我们组的每一位成员。
感谢 github 上的指导和大佬的教导。

6 参考文献

https://blog.csdn.net/danengbinggan33/article/details/72378684
https://github.com/ovolve/2048-AI
https://blog.csdn.net/qq_32767041/article/details/80634244

数据结构与算法课程实习作业报告

邱康怡* 罗文林 王秩铃 宁依华 邱中怡

摘要：本次 2048 游戏的算法设计我们小组运用的主要算法是 minmax 决策树配合 α - β 剪枝，通过选择合适的落子位置和合并方向使得我方估值函数尽可能大，同时增加了一定的启发式算法，以缩短运行时间；估值函数 value 设置有五个影响因素，分别是局面当前总分、我方分数分布的单调性、平滑性、我方分数最大值、和当前棋盘空格数；为了使运行时间和估值精度相平衡，搜索深度设定为 4 层。最后获得的代码能正常运行，不存在 bug，且智能程度与官方 bot_B、bot_C 等级相近。

关键字：minmax 决策树， α - β 剪枝，估值函数，启发式算法

1 算法思想.....2
1.1 总体思路.....2
1.2 算法流程图.....3
1.3 算法运行时间复杂度分析.....6
2 程序代码说明.....6
2.1 数据结构说明.....6
2.2 函数说明.....7
2.3 程序限制.....11
3 实验结果.....11
3.1 测试数据.....12
3.2 结果分析.....14
3.3 经典战局（可选）.....15
4 实习过程总结.....16
4.1 分工与合作.....16
4.2 经验与教训.....17
4.3 建议与设想.....18
5 致谢.....18
6 参考文献.....18

1 算法思想

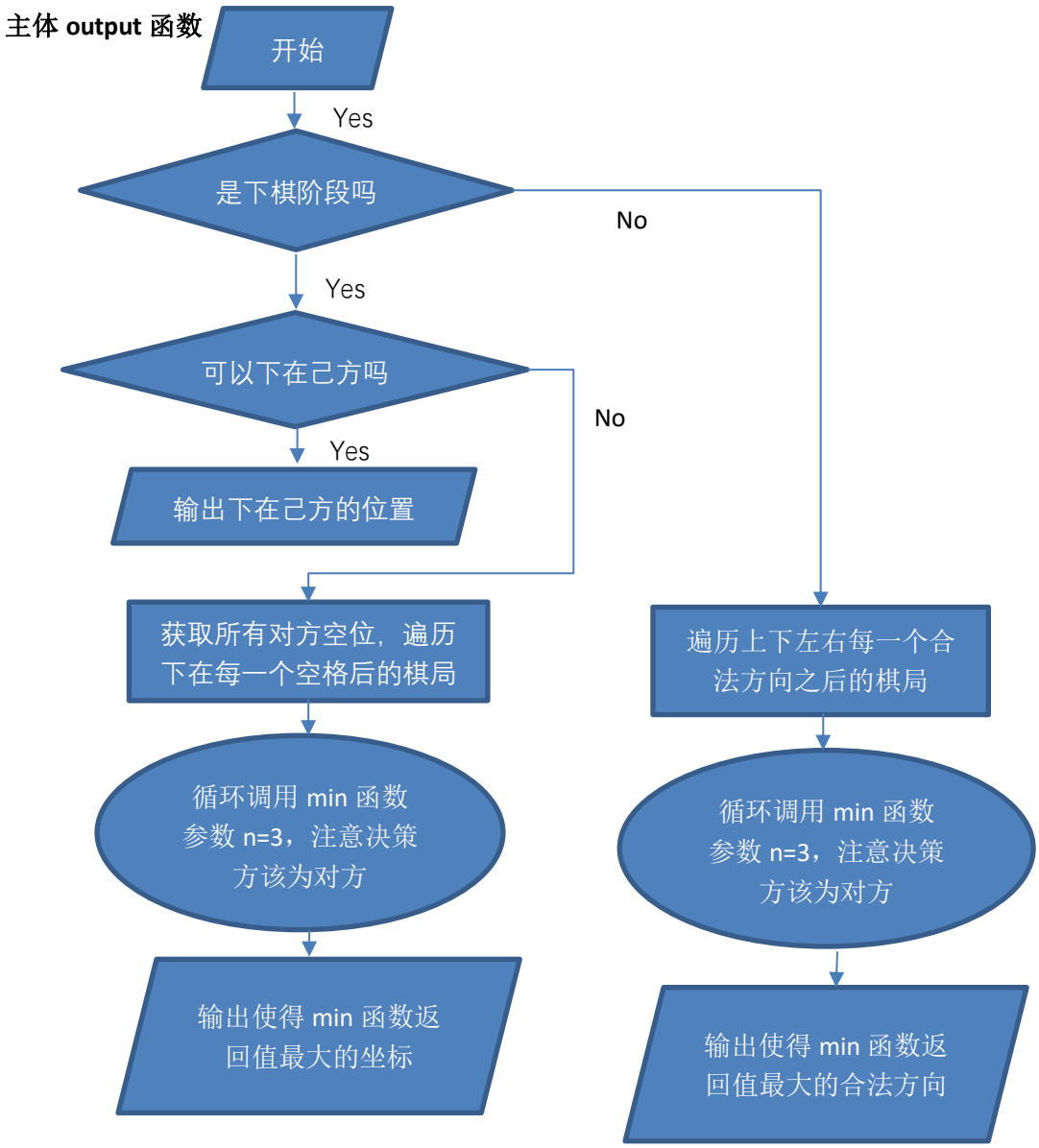
1.1 总体思路

算法的总体思路是下棋阶段能在己方下棋就己方，而只能下在对方时与合并阶段选择都比较多，此时假定己方和对方会做出对自己最有利的选择，对未来的棋局搜索四层找到使己方得分最高的下法。

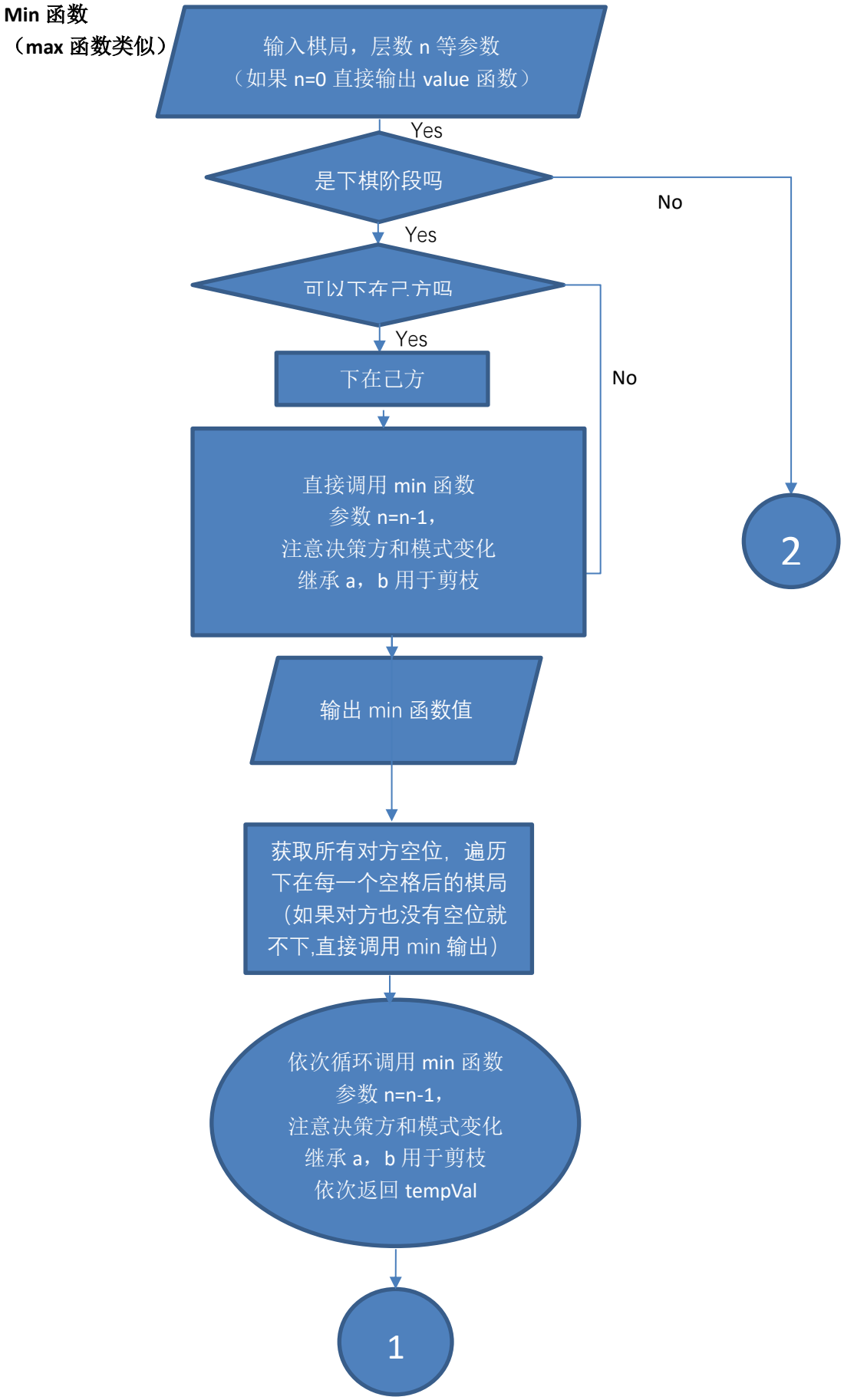
运用了 min-max 算法、递归思想、alpha-beta 剪枝方法、启发式规则等算法。

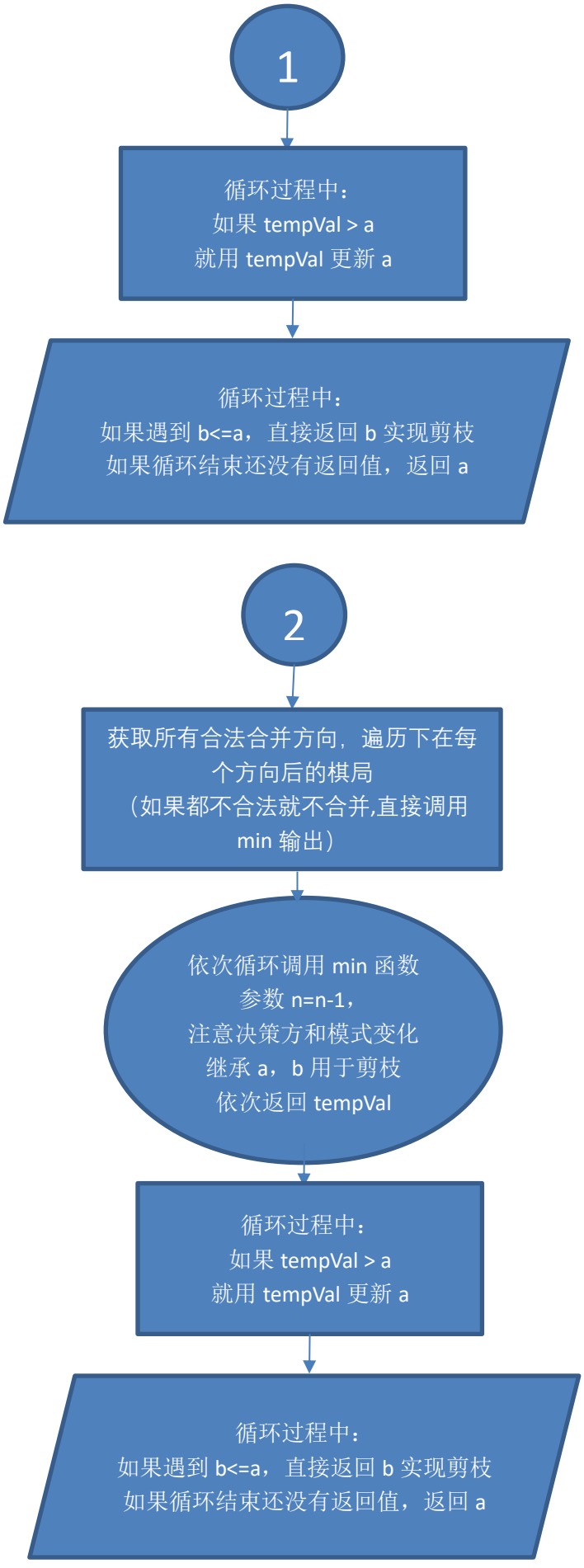
具体来说：由于通过观察大部分棋局，发现绝大部分情况下在己方下棋收益更大，故下棋阶段己方有位置则直接在己方下；不能在己方下棋时和合并阶段则运用极小化极大的算法思路，根据自己定义的 value 函数，计算棋盘对于己方的价值，使用最大最小化函数的相互调用进行多层递归去搜索经过一个回合的四次决策后价值最大的决策，并运用 alpha-beta 剪枝缩短所用时间。

1.2 算法流程图

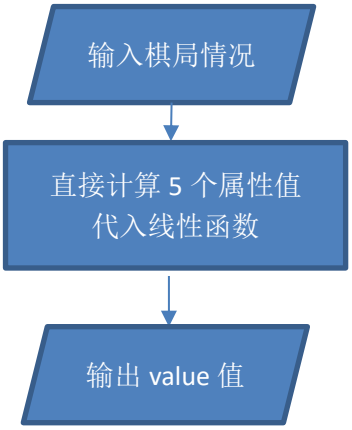

主体 output 函数
开始
Yes
是下棋阶段吗
No
Yes
可以下在己方吗
No
Yes
输出下在己方的位置
获取所有对方空位，遍历下在每一个空格后的棋局
循环调用 min 函数参数 n=3，注意决策方该为对方
输出使得 min 函数返回值最大的坐标
遍历上下左右每一个合法方向之后的棋局
循环调用 min 函数参数 n=3，注意决策方该为对方
输出使得 min 函数返回值最大的合法方向

Min 函数
（max 函数类似）




1
循环过程中：
如果 tempVal > a
就用 tempVal 更新 a
循环过程中：
如果遇到 b<=a，直接返回 b 实现剪枝
如果循环结束还没有返回值，返回 a
2
获取所有合法合并方向，遍历下在每个方向后的棋局
（如果都不合法就不合并,直接调用 min 输出）
依次循环调用 min 函数
参数 n=n-1，
注意决策方和模式变化
继承 a，b 用于剪枝
依次返回 tempVal
循环过程中：
如果 tempVal > a
就用 tempVal 更新 a
循环过程中：
如果遇到 b<=a，直接返回 b 实现剪枝
如果循环结束还没有返回值，返回 a

Value 函数



1.3 算法运行时间复杂度分析

通过代码结构容易看出，我们的 output 函数是 O(N)的时间复杂度，min-max 函数也是 O(N)时间复杂度，value 函数中的单调性、平滑性、局面分数也是 O(N)的。由于四层嵌套的特性，函数需要调用一次 output、3 次 min-max 函数、一次 value 函数，所以代码总体时间复杂度应该是 $O(N)^5$ 的。这里的 N 是回合数。

实验如下：

	output 函数	min-max	value	整个代码
标准回合平均时间（s）	0.00012	0.10742	0.61512	1.031
减少一半回合	0.00003	0.04237	0.2521	0.9231

经过多次实验并与标准幂函数比对，基本可以判断整个代码和其他单一函数的所耗时间的关系是低于 6 次幂高于 4 次幂的。和 $O(N)^5$ 的判断一致

2 程序代码说明

2.1 数据结构说明

数据结构的使用较为简单，主要是函数的互相调用。我们在 value 函数的定义中加入了 5 个属性以量化判断局势的优势情况。在每个下棋与合并的环节中，我们从逻辑上用到了树的深度优先搜索的思想。

2.2 函数说明

2.2.1 Output 函数:

作为主体函数，调用了 min-max 函数和 value 函数。算法的逻辑是下棋阶段能在己方下棋就己方，如果只能下在对方时或者在合并阶段，因为选择都比较多，此时遍历每一种己方的合法选择，对未来的棋局搜索四层，调用 min 函数，max 函数，并运用 alpha-beta 剪枝。最终调用 value 函数，找到使己方得分最高的下法。

2.2.2 min-max 函数：

2.2.2.1 简介

对于 min-max 函数，我们从逻辑上用到了树的深度优先搜索的思想，min 函数和 max 函数的相互调用隐含了递归的思想。附带有 α-β 剪枝，用以避免多余的无意义的算力消耗。

2.2.2.2 结构

（1）由于该决策需要众多参数，考虑诸多情况，为避免函数体过长导致调试困难和容错率低容易出 bug 的问题，在实现层面对该函数进行拆分，分为比较对手决策，取最小值的_min 函数和实现自己决策，取最大值的_max 函数。两个函数的深层搜索都是通过迭代，引用对方实现的。
（2）_min 和_max 函数的搜索深度为 4，覆盖一个回合的四次决策，即每次决策都要考察双方的一次落子和一次合并。并用整数 n 来记录，初始值为 3，每进行一层搜索就减少 1，其值为 0 时停止，返回对应局势的价值函数。

2.2.2.3 实现

两个函数的基本功能类似，只在具体的选取最大值和最小值上有所区别。

（1）首先关于参数，函数需要的参数有：
- Board：棋盘类型，当前棋盘
- player 布尔值，代表利用该函数决策的玩家，_max 函数 player 是我们自己，_min 函数的 player 是对手
- mode：字符串，方向和位置，要求作出的行动，是合并还是落子
- n：记录搜索深度，到达要求的深度就停止
- currentRound：代表当前回合数
- a，b：整数，记录棋面的价值，是 α-β 剪枝中的两个重要参数，初始值为-99999 和 99999

Output()函数中对_min(),_max()函数的引用如下，可清楚展现 7 个参变量，及其对应作用：

```
if self.isFirst == True:  # 先手情况下
    tempVal = self._min(nextBoard, not self.isFirst, mode, 4, currentRound,  99999, 99999)
else:  # 后手情况下
    tempVal = self._min(nextBoard, not self.isFirst, 'direction', 4, currentRound, -99999, 99999)
```

```
if self.isFirst == True:  # 先手情况
    tempVal = self._min(nextBoard, not self.isFirst, mode, 3, currentRound, -99999, 99999)
else:  # 后手情况
    tempVal = self._min(nextBoard, not self.isFirst, 'position', 3, currentRound - 1, -99999, 99999)
```

（2） 关于函数的本体：

- 终止条件：函数本身是一个相互引用的迭代函数，其终止条件是记录搜索深度的 n 变为 0，如果为 0，直接利用价值函数返回此时局面的价值。
- 规模的缩小：每一次决策都会将记录深度的 n 减 1，并重新调用。
- 形势的判断：首先判断要求给出合并还是落子：

对于落子，分为三种情况：

如果己方区域有位置可以下，那就直接下在己方，然后进入下一层，在_min 中引用_max 作为下一层，_max 中引用_min 作为下一层。

如果己方无位置可下，则获取对方区域的空位，对每个空位都进行一次判断，_max 选择价值最大的落子，_min 则选择价值最小的落子。

如果没有地方可以落子，就直接到下一层。

需要注意的是，先手和后手有明显的区别，涉及到下一步是合并还是落子，因 此在进入下一层是需要单独判断。

对于合并，分为两种情况：

如果可以合并，就依次访问四个合并方向，判断哪个好，然后根据这一步做出决策的玩家选择最大值或者最小值，并进入下一层。

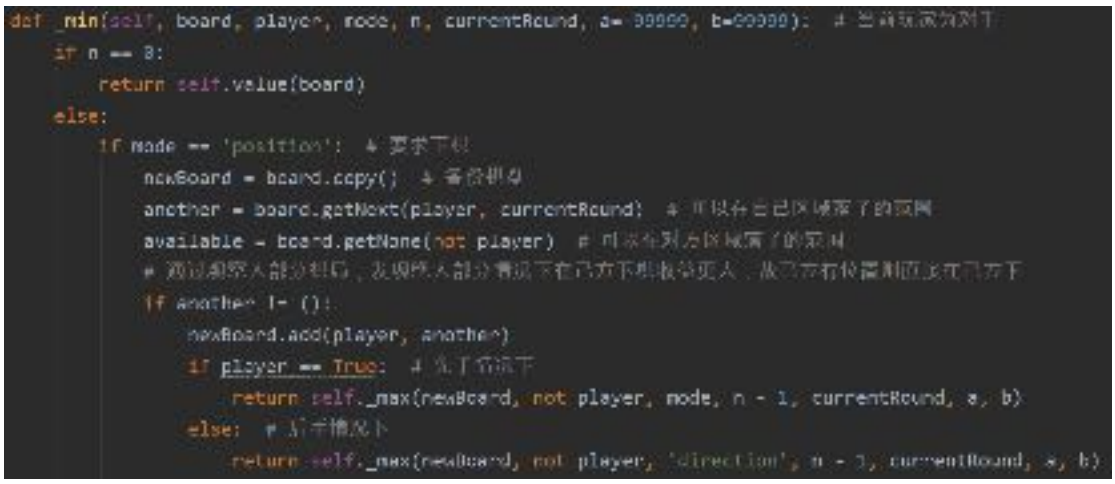
如果四个方向都不可以合并，那就直接进入下一层。

需要注意的是，先手和后手有明显的区别，涉及到下一步是合并还是落子，因此在进入下一层是需要单独判断。

2.2.2.4 α - β 剪枝

引入 a 作为_max 记录决策的最大值，b 作为_min 记录决策的最小值，下一层会继承上一层的 a，b 值，对于_min, 如果出现下一层_max 返回的值大于上一层的已有的最小值就直接枝剪，不进行进一步计算；对于_max，如果出现下一层_min 返回的值小于已有的最大值，也直接剪枝，不再进行搜索。通过这样完成枝剪。

具体的代码展示如下，只显示_min()函数，对_max()同理：



```
def _min(self, board, player, mode, n, currentRound, a=-99999, b=99999):  # 当前玩家为对手
    if n == 0:
        return self.value(board)
    else:
        if mode == 'position':  # 要求下棋
            newBoard = board.copy()  # 备份棋盘
            another = board.getNext(player, currentRound)  # 可以在己方区域落子的位置
            available = board.getNone(not player)  # 可以在对方区域落子的位置
            # 通过观察人部分棋局，发现绝大部分情况下在己方下棋都会更大，故己方有优先判断权
            if another != []:
                newBoard.add(player, another)
                if player == True:  # 先手情况下
                    return self._max(newBoard, not player, mode, n - 1, currentRound, a, b)
                else:  # 后手情况下
                    return self._max(newBoard, not player, 'direction', n - 1, currentRound, a, b)
```

```
            elif available:  # 在对方区域，即将完全进行对决，寻找决策
                for choice in available:  # 依次考察在对方每个可落子点下的收益
                    newBoard = board.copy()  # 备份棋盘
                    newBoard.add(player, choice)  # 下在该点
                    if player == True:  # 先手情况下
                        tempVal = self._max(newBoard, not player, mode, n - 1, currentRound, a, b)
                    else:  # 后手情况下
                        tempVal = self._max(newBoard, not player, 'direction', n - 1, currentRound, a, b)
                    if tempVal < b:  # 取最小值的情况
                        b = tempVal
                    if b <= a:  # 如果b<=a,产生剪枝，不需要
                        return a
                return b
            else:  # 不能在己方区域落子之处
                if player == True:  # 先手情况下
                    return self._max(board, not player, mode, n - 1, currentRound, a, b)
                else:  # 后手情况下
                    return self._max(board, not player, 'direction', n - 1, currentRound, a, b)
```

```
        else:  # 对方选择合并
            directionList = [2, 3, 0, 1]
            # 关于这个合并方向，我这样写的：选择当前棋盘有优先级的，最后一个方向为当前的输入值，可以更直观和整齐一个减小得出
            nextDir = None
            bestVal = -value  # 相关棋局这个方向每个块可以合并，不论个方向哪个可以，为-value
            for direction in directionList:  # 对每一个方向判断，以
                newBoard = board.copy()  # 备份棋盘
                if newBoard.move(player, direction) == True:  # 对这个方向已经具有判断移动并导致合并发生
                    hasMove = True
                    if player == True:  # 先手情况下
                        tempVal = self._max(newBoard, not player, mode, n - 1, currentRound, a, b)
                    else:  # 后手情况下
                        tempVal = self._max(newBoard, not player, 'position', n - 1, currentRound + 1, a, b)
                    if tempVal < b:  # 取出最值的那个情况
                        b = tempVal
                    if b <= a:  # 如果b<= a,即达成剪枝，不必考虑
                        return a
            if hasMove:  # 如果这个方向可以合并
                return b  # 最后返回b，即最小情
            else:  # 如果这个方向都不可以合并
                if player == True:  # 先手输入1
                    return self._max(board, not player, mode, n - 1, currentRound, a, b)
                else:  # 后手情况下
                    return self._max(board, not player, 'position', n - 1, currentRound, a, b)
```

2.2.3 Value 函数

Value 函数是用 5 个参数构造的线性函数，5 个参数分别为：局面分数、单调性、平滑性、空格数、最大值。

五个参数可根据实际情况设置对应权重，经过多次调试，得到较为准确的权重占比，下面是 value()函数代码展示：

```
# 影响value函数的有5个因素，局面分数、单调性、平滑性、空格数、最大值
def value(self, board):
    totalValweight = 1.0  # 权重 可更改
    monoweight = 0.2  # 单调性1 权重
    mono2weight = 0.4  # 单调性2 权重
    smoothweight = 0.2  # 平滑性 权重
    emptyweight = 0.8  # 空格数 权重
    maxweight = 1.0  # 最大值 权重

    res = monoweight * self.monotonicity(board) + mono2weight * self.monotonicity2(board) \
          + smoothweight * self.smoothness(board) + emptyweight * self.emptyCell(board) + \
          maxweight * self.maxValue(board) + totalValweight * self.totalValue(board)
    return res  # 计算并返回结果
```

2.2.3.1 局面分数 totalValue

（1）简介：

局面分数的值越大对自己越有利，是自己分数减去对方分数的值。

（2）实现：

通过遍历场上每一格，统计总分数来实现。

2.2.3.2 单调性

2.2.3.2.1Monotonicity

（1） 简介：

以己方最大值为辐射中心，辐射长度为 4 的十字形范围内，从最大值点朝每个方向出发的单调（递减）性。

（2） 详述：

最理想的情况是返回值为 0，出现上升情况则要累加上升值。

这个参数本应是越小越好，辐射中心出发一路向下才最理想，就像从山顶下坡，不希望下坡过程中有向上的阻碍。但是为了方便构造 value 函数变成越大越好，所以我们返回其的相反数。

（3） 实现：

使用 tx，ty=x，y，通过不断更新 tx，ty 实之成为下一次循环己方被比较的点。

2.2.3.2.2 Monotonicity2

（1） 简介：

整个棋盘己方棋子的横向、纵向单调性，越大越好。

（2） 实现：

创建 totals 列表 = [0, 0, 0, 0]，四个维度，分别表示为横向递减，横向递增，纵向递减，纵向递增。

先按行遍历求每一行己方相邻棋子的数值单调性，后一个比前一个小即横向递减，则 totals[0]+=横向递减的值，横向递增则 totals[1]+=横向递增的值。再同样地按列遍历求每一列己方相邻棋子的数值单调性，记录 totals[2] ，totals[3]。

最后返回 max(totals[0], totals[1]) + max(totals[2], totals[3])

2.2.3.3 平滑性 smoothness

（1）简介：

光滑性，即整个棋盘己方棋子按照价值排列是否连续，值越大越好。

（2）实现：

先按行遍历每一行己方相邻棋子的价值差绝对值之和，再遍历每一列相邻棋子，最后加总所有价值差的绝对值之和，取相反数。

2.2.3.4 空格数 emptyCell

（1）简介：

此处的空格数为己方区域的空格数，空格越多选择越多，所以越大越好。

（2）实现：

用 getNone 可以很容易地实现。

2.2.3.5 最大值 maxValue

（1）简介：当前棋盘己方最大的棋子的价值，越大则己方越有优势。

（2）实现：用 getScore 可以很容易地实现。

2.3 程序限制

我们的代码当然不是完美的，限制之一是我们运用了启发式规则：由于我们观察到大部分棋局，发现绝大部分情况下在己方下棋收益更大，故己方有位置则直接在己方下。这个观察结果是有一定限制条件的，在己方已经非常拥挤的情况下，如果仍然下在己方，会大大提高卡死自己的概率；而在对方空格不多的情况下，也是下载对方处可以加速对方被卡死的进程。但这个规则在较早的轮次还是相当有效的，由于下棋轮数不多，该启发式规则有一定的存在价值。

3 实验结果

3.1 测试数据

实验环境说明：

- 硬件配置：内存：8GB
  处理器：Intel(R) Core(TM) i5-7200U CPU
- 操作系统：Microsoft Windows 10
- Python 版本：Python 3.7

我们采用了两种办法对算法进行测试，首先，在完成代码进行本地调试之后，我们将算法与给出的简单 AI 算法进行对战，初步观察算法在先后手的胜率、对局的回合数以及运行的时间。在确定算法具有一定的智能后，我们将算法上传到天梯上与官方 bot 以及其他同学的算法进行对战，进一步判断算法的优势和劣势，进而调整后续的优化方向。

（1）与简单 AI 算法的对局结果

测试程序：

```
if __name__ == '__main__':
    main(['D:\PycharmProjects\sessdsa_2048_master\N19player\Player1.py', \
          'D:\PycharmProjects\sessdsa_2048_master\src\player.py'],\
         debug=True)
```

比赛结果：

```
defensive cases:
    average time: 1.031
        win rate: 100.00%
        win: 10 at
            1222299247899466755
            1052042025875253255
            1031643616278544395
            485042839191535631
            380492688973570067
            282659459441451030
            151658522896197224 9
            206414485083090973
            556927264419479584
            1053678021490294819
        lose: 0 at

        violate: 0 at

        timeout: 0 at

        error: 0 at
```

```
offensive cases:
    average time: 0.961
        win rate: 90.00%
        win: 9 at
            2110526807702548993
            297671419768635397
            1049984060989530121
            1664893131381485580
            1612503225290645521
            149131780126703637
            1860458363937628183
            409686506144628763
            997292977299012094
        lose: 1 at
            2030850938946486305
        violate: 0 at

        timeout: 0 at
```

可以看到在与简单 AI 算法的对局中，我们的代码在先手的情况下胜率为 90%，在后手的情况下胜率可以达到 100%，比赛平均运行时间为 1 秒左右，属于正常范围。

（2）天梯平台上与官方 bot 的对局结果

算法在与白痴算法的对局中表现不错，我们将其上传到天梯上与官方 bot 进行比赛，结果如下所示：

由于我们小组在编写代码的进度较慢，热身赛参与度不够，我们简短统计统计了最终代码在天梯上参与比赛的胜率、回合数等信息，结果汇总为如下表格：

对战代码名称	对战结果(我方为前一比分）
530_slim	2:4
这次不是 Sophie 酱	1:5
2048Bot_D	5:1
2048Bot_C	6:0
胜率	58.33%

3.2 结果分析

总的来看，比赛的结果是符合预期的，在天梯赛上的对局中，我们的算法平均可以保持50%的胜率。在测试过程中，我们认为最终版本相比于初期的版本，有以下三点优化起到了表位明显的效果。

第一是多方面考虑影响 value 函数的因素。由于时间原因，我们最初的算法中 value 函数设置的较为简单。然而比赛最终的目标是得到最大的数字，经过讨论我们认为比赛的核心策略是得到一个最为准确的 value 函数，这决定了棋盘可以向正确的方向移动以得到更大的数字。因此，在查阅资料的基础上，我们考虑到了影响 value 的五个因素：局面分数、单调性、平滑性、空格数、最大值，并且都在比赛的算法中有所体现。

第二是对影响因素的权重进行探索。在确定影响因素后，我们对于因素的权重进行了探索，采用不同权重的算法进行天梯对战、自身对战等方法，通过胜率得到最优的权重配比。最终的算法效果良好，可以在比赛中达到 2048 甚至 4096 的成绩。

最后是对于算法运行时间的优化。在测试的过程中，我们的算法有过超时的问题，主要

的时间开销是花费在 value 函数上。针对这一问题，我们在 minmax 函数部分采用 alpha 剪枝的方法，有效缩短了代码的运行时间。

3.3 经典战局（可选）

在 6 月 7 日与 530_slim 对战的一次比赛中，本组的 4096 数字是在对方的棋盘上出现的，具体过程见下图：

可以发现在我方下达向右移动的指令之后，我方的 2048 吃掉了对方的 2048，事实上，我们在 value 部分的代码中除了在局面分数中考虑到使得我方分数最小，对方分数最大；在其他的四个因素中都是以我方分数最大为依据。这个比赛现象可能大部分是出于一种巧合，但也提醒我们在判断价值是可以尽可能多的降低对方的价值。

4 实习过程总结

4.1 分工与合作

本小组的交流合作主要采取微信和腾讯会议线上交流的方式。通过微信来进行任务的分配与跟进，通过腾讯会议来进行讨论对于算法的搭建与优化的想法，同时交流写代码过程中的疑惑和经验。由于小组成员来自不同的院系，彼此的时间安排差异较大，故比较遗憾的是在实习作业过程中仅举行过两次线上会议。

第一次腾讯会议与 5 月 25 日举行，主要讨论了对于算法的大致想法，初步将任务分为基础代码、调试与优化以及报告三个部分，并分配了任务。

第二次腾旭会议于 5 月 30 日举行，此时基础算法已经完成，主要的任务是优化算法。我们将重点放在 value 函数的构造，调参以及 alpha 剪枝的建立上。但代码在优化的过程中出现报错的情况，对代码进行本地调试花费了我们较多的时间。

最后报告的分工情况如下：

基础算法实现：罗文林（包括 minmax 函数，output 函数，简单 value 函数）

算法优化：邱中怡、邱康怡（value 函数优化），罗文林（加入剪枝），王秩铃（修改剪枝和 minmax 函数）

调试与天梯运行：邱康怡、王秩铃

实习报告：宁依华（12 部分）、邱中怡（34 部分）、其他人完成了报告的函数说明和修改。

为了认真完成本次大作业，我们小组五个人多次召开腾讯会议讨论，下面有截图展示：

4.2 经验与教训

（1）经验

整个实习作业任务量其实比较大，需要全组的同学相互配合，发挥自己的优势。在这一点上，我们组做的比较好，从最开始便有分工合作的意识，将代码和报告的任务相对独立开来，可以保证写代码和写报告同学思维的连贯性。另外，每个人的任务分配较为平均，可以使得大家有足够的参与度，从这次的实习作业中有所收获。

（2）教训

此次实习作业我们组最大的一个问题是对时间的安排不够合理，同时对于关键时间点不够重视。本组成员均是跨院系选课，在期末季可能大家忙于自己其他课的任务，忽视了有关实习作业的比赛流程与提交时间的通知。由于一直对比赛流程理解错误，我们组过早的提交了有错误的代码，使得在天梯赛开始之前比赛积分就所剩无几，再加之记错了最后提交代码的时间，没能来得及在天梯上提交修正之后的代码。事实上，最后完成的代码效果并不差，

如果可以记住提交的时间，我们还是有信心可以取得不错的成绩的。

其次，在完成实习作业的过程中，我们小组的沟通频率较低。须知一个好的想法往往是在讨论中得来的。另外，由于我们在完成作业的过程中讨论不够，大家对于算法细节问题想法不一致，这一定程度上增加了合作写代码和报告的难度。

最后是代码阅读的问题。在小组合作的过程中，整个算法需要不同的成员来完成。由于每个人写代码的思维不同，如果算法格外复杂，则会导致代码的相互交流和修改非常困难，往往需要花费很大的时间与精力来看懂其他人的代码。因此在合作撰写代码时要注意加上详细注释，有机会要进行充分的沟通交流。

4.3 建议与设想

本次实习作业 2048 是一个我们比较熟悉并且非常有趣的游戏。规则简单但是背后的思想算法具有无限拓展的可能。非常感谢老师、助教和技术组的同学为这次作业花费了巨大心血，为我们创造了一个宝贵的运用数据结构与算法知识的机会。

对本次实习作业的建议主要有下面两点。其一是赛制问题，由于是否出现取决于天梯排名，而较早提交代码的小组排名由于比赛轮数多，即使最后的代码胜率很高，可能也难以上升太多名次。也许可以考虑在天梯赛时积分初始化，重新进行排名。另一个建议是，大作业的难度实际上还是比较大的，在一个组 4 到 5 人的情况下依旧很容易出现搭便车的行为，可以尝试稍微降低大作业的难度，将小组的人数控制在 2-3 人，这样的可以使每个人都参与进写代码的过程中。

最后，对于 2048 游戏，我们只是从一个角度来实现算法，最优的策略依旧没有定论。在后续的实习作业中，我们可以从对个角度进行思考，也可以在我们的算法中添加更多的情况，从而使算法的胜率更高。

5 致谢

首先，感谢小组每一位成员。感谢大家积极参与到了实习作业中，认真完成自己的任务，为算法的实现付出的努力；感谢大家在有同学的算法出现问题时积极帮对方思考解决的办法；感谢大家在我们组没有及时交上代码后，依旧尽心尽力的完成了一个优秀的算法；也感谢大家的相互鼓励，使得这次的实习作业虽有波折，但依旧有一个圆满的结束。

另外，感谢老师提供的实践的机会，感谢助教们在微信群中的耐心答疑，感谢技术组的同学为我们构建的对战平台，最后，也要感谢天梯平台的 2048Bot_ABCDSSSSSS，通过实战给了我们很多优化的思路。

6 参考文献

[1]什么是游戏 2048 的最佳算法？https://blog.csdn.net/kimylrong/article/details/21967869

数据结构与算法课程实习作业报告

卢诗钰* 季祥君 林柏仲 邢泽宇 胡宇瑄

摘要：构造决策树的同时进行 alpha-beta 剪枝来减小树的规模，并对未来棋盘进行估值，采取相应的决策。AI 利用最大数、空位数、平滑性三种指标，增添权重后进行求和返回估值大小。

关键字：决策树、alpha-beta 剪枝

1 算法思想.....2
1.1 总体思路.....2
1.2 算法流程图.....3
1.3 算法运行时间复杂度分析.....3
2 程序代码说明.....4
2.1 数据结构说明.....4
2.2 函数说明.....4
2.2.1 Alpha-Beta 剪枝函数.....4
2.2.2 估值函数.....5
2.3 程序限制.....5
3 实验结果.....6
3.1 测试数据.....6
3.2 结果分析.....7
4 实习过程总结.....7
4.1 分工与合作.....7
4.2 经验与教训.....8
4.3 建议与设想.....9
5 致谢.....9
6 参考文献.....9

1 算法思想

1.1 总体思路

在实现 AI 的过程中，我们一开始就通过讨论确定了主要采用 Alpha-Beta 剪枝作为决策树的基础，构建一系列指标并加权得到估值函数，把两者结合在一起作为总体思路。

关于决策树部分，第一次讨论时大家厘清了在模拟各步骤的所有可能性后，直接进行估值并选择估值最大的动作，并忽略其余节点。决策树使用二叉树数据结构，基本结束条件为对叶节点调用估值函数，递归深度 depth 由零开始，每次递归调用深度加一，直到深度等于剪枝深度 PRUNEDEPTH。有组员提出了这种基于“贪心”想法的算法可能不能得到长期下的最优解，但在后来实操过程中发现时间限制无法预留额外未来 N 步的深度，因此最后仅限于本回合的估值大小来进行判断。决策步骤首先区分我方为先手和后手两种情况，此后每轮的四个步骤各对应一个 phase（0、1、2、3），调用“position”时从头开始计算，调用“direction”时进入“我方选择合并方向”对应的层，然后更新深度参数并迭代，直到达到深度上限返回估值。成员们设想时希望对我方下棋的所有可能、对方下棋的所有可能、我方所有合并方向和对方所有合并方向都进行搜索，经调试发现会超时报错。在尽可能的多搜索和超时之间，我们小组最后在“对方下棋的所有可能”这一步（即我方为先手方情况的第 2 步）中进行删减，假定对方优先按随机序列下棋，没有空位时在我方通过 random.choice 选择一个位置。对于己方下棋来说，我方优先考虑于本地棋盘下棋，后考虑于对方棋盘下棋。

关于估值函数，我们查阅了一些原版 2048 的相关资料，了解到如最大数、平滑性、单调性等等可用于估值的方法，结合了对战版本的计分方式，构造了最大数、空位数和平滑性三个指标。单调性一开始也是指标之一，但在决策树优化之后仍然常出现超时问题后，组员发现计算单调性的时间复杂度较高，参数调小也会使决策丧失一些合并机会，而且去掉之后棋盘表现也相当单调，因此决定舍弃。最后的估值函数采用了三个指标线性相加的方式。

1.2 算法流程图

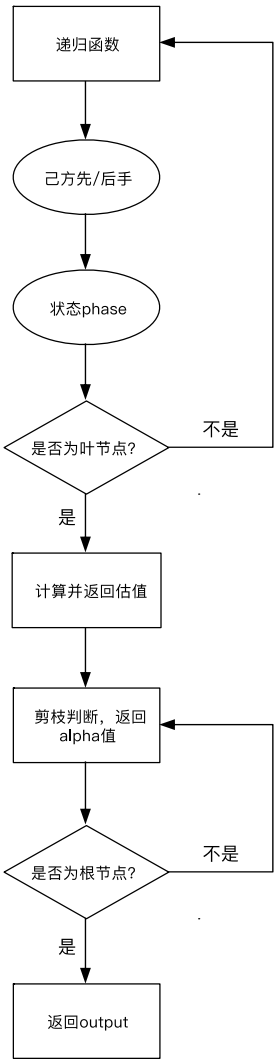


图 1：决策树流程图

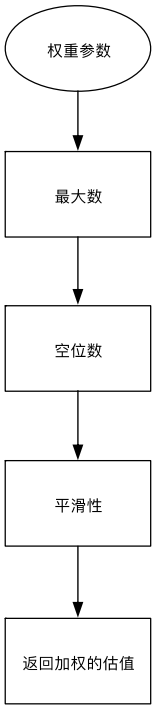


图 2：估值函数流程图

本部分流程图只给出大致的算法思路，和相对重要的逻辑判断部分，尤其在决策树部分的迭代涉及到当前状态的判断和更新，具体参看代码说明。

1.3 算法运行时间复杂度分析

算法运行时间主要受棋盘中棋子数量影响，可以分为 _alphabeta 和 _evaluate 两个函数构成。假设己方棋子数量为 p，对方棋子数量为 q，对方棋盘平均空位数为 n，剪枝深度为 PRUNEDEPTH，令 $k = \text{PRUNEDEPTH}/4$。每一次进入 phase 0 时，时间复杂度可能为 $O(1)$ 或 $O(n)$，最好情况时我方都在己方棋盘下棋（位置已定），最坏情况时我方遍历对方棋盘空位。其他三个 phase 的时间复杂度为常数。估值函数的时间复杂度主要来源于平滑性，为 $O(p+q)$。我们的决策树类似于“间歇性”的四叉树，“四叉”的原因在于合并时需要分别递归四种不同方向。综合以上，算法的时间复杂度最好为 $O(4^{2k})$，最差为 $O(n^k \cdot 4^{2k})$。

2 程序代码说明

2.1 数据结构说明

本组的 AI 程序主要采用线性列表和树作为基本数据结构类型。我们将当前棋局以顺序列表的形式存放，并作为博弈树的根节点。每个子节点进行一步下棋 / 合并操作，到该节点形成新的棋局。我们通过估值函数对每个节点的棋局进行估值，并通过博弈树的结构，对最优下棋 / 合并位置进行搜索。

2.2 函数说明

程序定义的函数主要由两个部分组成：Alpha-Beta 剪枝函数和估值函数。

2.2.1 Alpha-Beta 剪枝函数

从根节点开始采用深度优先的方式构造格局树，在构造每个节点时，都会读取此节点的 alpha 和 beta 两个值，其中 alpha 表示搜索到当前节点时已知的最好选择的下界，而 beta 表示从这个节点往下搜索最坏结局的上界。由于我们假设对手会将局势引入最坏结局之一，因此当 beta 小于 alpha 时，表示从此处开始不论最终结局是哪一个，其上限价值也要低于已知的最优解，也就是说已经不可能此处向下找到更好的解，所以就会剪枝，提高运算速度。

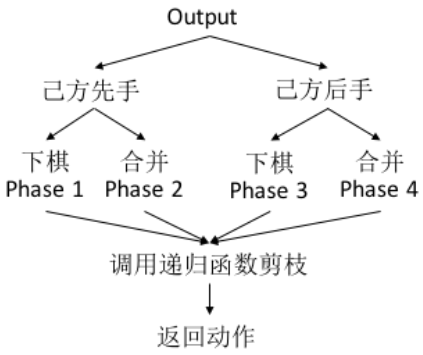


图 1：Alpha-Beta 剪枝函数逻辑

剪枝函数中分两种主要情况：己方为先手或后手，并进行大致相同的操作。为了分辨四种情况（先 / 后手下棋、先 / 后手合并），我们引入 phase 变量（0、1、2、3 分别对应各种情况）。

对于己方下棋来说，我方优先考虑于本地棋盘下棋，后考虑于对方棋盘下棋。由于随机序列已定下在己方棋盘下棋的位置，因此不用 for 循环，直接进行递归调用；反之，由于己方可选择在对方任意空位下棋，因此需要使用 for 循环后进行递归调用。若己方、对方的棋盘都满，则跳过下棋回合。

另外，对于己方合并来说，我方对上、下、左、右四个方向进行合并测试。若合并后棋盘有变化，说明该方向为合法的合并方向，并往下进行递归调用。

无论是下棋或合并阶段，对于己方来说，只要递归深度为零（根节点），则剪枝函数返回相应的动作。程序的基本结束条件即对叶节点调用估值函数后，返回浮点数。

board	棋盘：Chessboard 类
phase	当前游戏状态：0 代表先手选择下棋位置，1 代表后手选择下棋位置，2 代表先手选择合并方向，3 代表后手选择合并方向
alpha	Alpha 值：初始值用 9223372036854775807 模拟 $+\infty$
beta	Beta 值：初始值用$-$9223372036854775807模拟 $-\infty$
currentRound	当前游戏轮次
depth	递归深度：初始值为 0，每次递归调用深度加一，直到深度等于剪枝深度 PRUNEDEPTH，即递归至叶节点时进行上述操作

表 1：Alpha-Beta 剪枝函数之参数

2.2.2 估值函数

在该函数中，我们主要采用平滑性、空位数和棋子最大值这三个启发式指标。

board	棋盘：Chessboard 类

表 2：估值函数之参数

平滑性指遍历所有棋子，对于每个棋子，如果它存在相邻棋子，那么计算它与相邻棋子的对数差（取绝对值），然后加总，总和越小越平滑。

空位数用本方空位数与对方空位数之差进行估值，我们认为空位数越少对玩家越不利。

棋子最大值用本方最大级别与对方最大级别之差进行估值，我们认为棋子最大值越大对玩家越有利。

最终，对三个启发式指标分别设置权重，输出最后的评分。

2.3 程序限制

当剪枝深度过大时（例如 depth 大于 4 时），由于需要搜索的节点数呈指数级增长，运算时间会很长，导致累计运行时间超出最大时间限制以至于回合意外结束，直接判负。

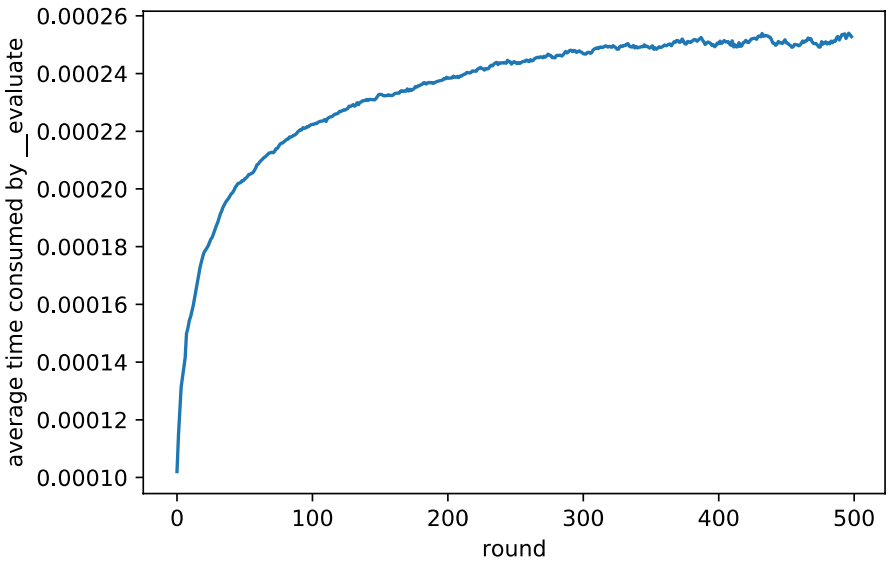
3 实验结果

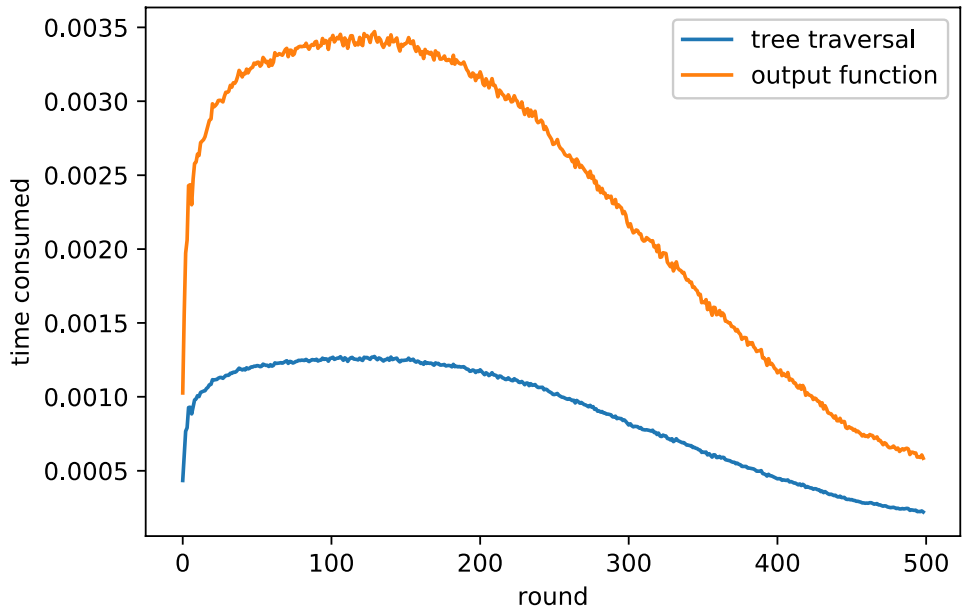
3.1 测试数据

实验环境说明：

- 硬件配置：i9-9980HK@2.3GHz (Turbo Boost disabled)/32GB RAM
- 操作系统：macOS Catalina/10.15.5
- Python 版本：3.8.3

在本地与自身对弈 1000 回合，记录了 output 函数运行消耗的时间以及单次 __evaluate 函数消耗的时间，并根据这些数据计算出了在 output 函数在决策树中搜索所花的时间。





同时在网站上进行了在线热身赛，并根据对战结果调整权重和__evaluate 函数。最终的版本与网站上的 2048Bot_C 和 2048Bot_B 比赛结果如下表。

2048Bot_C	8/2	7/3	7/3	5/5	5/5	5/5	9/1	8/2
2048Bot_B	0/10	0/10	0/10					

3.2 结果分析

算法中采取的 alpha-beta 剪枝一定程度减少了需要进行的搜索次数，但在实际中依然还是只能达到搜索深度为 4，否则可能出现较多的超时。算法的运行时间比设计时的预期略大。

根据测试结果可以知道，主要的运行时间开销发生在 __evaluate 函数对当前局面进行估值的环节，占总时间的 70% - 80%，而在决策树中搜索所花的时间大约在 20% - 30%。如果能够改善估值函数的计算速度，或者对其进行一些简化，或许可以提高搜索深度。

4 实习过程总结

4.1 分工与合作

我们小组的分工大致为：林柏仲和邢泽宇同学负责代码实现，季祥君、卢诗钰和胡宇瑄同学负责大报告的撰写。此外，所有同学都在组会讨论和日常交流上积极参与，出谋划策。

我们合作和交流的方式为组会和微信群，历次组会记录如下：

微信群日常交流记录如下：

4.2 经验与教训

首先，本组在写估值函数时先采用了平滑性、空格数和最大数三个启发式指标，后来又加入了单调性和孤立空格数，发现 AI 性能没有变好且超时现象更为严重，于是我们选择去掉后面加进的两个启发式指标。在各启发式指标的权重设置上我们存在一定的改进空间。

其次，我们在代码现版本的基础上会遇到两个合并问题。第一，我们观察 AI 的操作情况，发现有时候上下合并更为值得时 AI 却持续地进行左右合并而损失机会；第二，当己方可以合并对方的棋子时，往往合并后就被相邻的棋子吃掉（附近有合并后相同大小的棋子）。在以上发现下，我们尝试采用先验策略，即当对方最大的棋子在我方棋盘上，且我方可对其进行合并操作时，就直接阻断递归，马上合并。虽然能为 AI 增加此种操作，但实践时会常常遇到超时问题，因此选择较为较保守的版本。

最后，本次合作过程中并未善用云端版本控制的功能，导致代码处理有些混乱。除了队友无法及时查看进度以外，也容易造成多余的 bug。主要原因在于对版本控制的操作不太熟悉，个别组员以往并无合作写代码的经验，希望通过这次合作能有所改进。版本控制的重要性在于能够查看代码旧版本，而本地编辑进行储存后，除非自行建立新文档，否则下次操作就无法浏览旧版本了。

4.3 建议与设想

首先建议学期开始时设定好学期计划，期中季前后布置期末大作业。这样不仅能督促学生提早组队，也给提供能力不足的学生更长时间调试代码。

最终竞赛的直播时间比较仓促，基本只能看到比分和复盘的一些关键节点，希望以后可以在周五的课比完，周二的大课上具体复盘并请一些优秀小组分享一些算法思路，因为提交的实习报告是延后公开的，实效上差一些。

祝愿明年选修数据结构与算法课程的学弟学妹能在实习作业与竞赛中有更好的表现，在课程中有更多的收获。

5 致谢

首先感谢陈斌老师一学期以来的辛苦教导，线上授课依旧不减课程的质量；感谢袁泽、陈天翔、张赖和、冀锐、魏江南和杨佑铭六位助教，为课程不断付出，耐心解答我们的问题；感谢技术组的各位同学，与老师助教一起搭建平台，保证了本次实习作业的顺利开展；感谢一同选课的同学们，虽然没能实际见到，但大家在线上的交流仍然十分活跃，课程氛围很好。最后，十分感谢小组每一位成员的付出，大家来自不同年级的院系，但是都能够积极参与、辛苦付出，最终取得了成果，谢谢你们。

6 参考文献

https://blog.csdn.net/marble_xu/article/details/90647361
https://blog.csdn.net/shi_weihappy/article/details/23345159
http://blog.codinglabs.org/articles/2048-ai-analysis.html
https://blog.csdn.net/qq_32767041/article/details/80634244

数据结构与算法课程实习作业报告

姜治仲* 张皓然 冯子琦 李宇轩

摘要：在此“二〇四八”游戏 AI 算法程序中，我们设计了一个针对棋盘局面的估值函数，进行了框架改进和参数调整。在此基础上，我们使用 minimax 值法构建了深度为三层的决策树，同时对估值函数进行拓展以部分实现深度为四至五层的决策，并使用 Alpha-Beta 剪枝方法来降低运算的时间复杂度。经过分析和测试，此算法不同阶段的大 O 数量级有三种情况分别为$O(n^3), O(n^2), O(n)$（n 为棋盘面积），主要时间开销分布于计算估值函数。在正式天梯赛中胜率在 47-48%左右，排名第八出线，较好的实现了“二〇四八”游戏的决策。

关键字：决策树 minimax 值 Alpha-Beta 剪枝

1 算法思想.....2
1.1 总体思路.....2
1.2 算法流程图.....5
1.3 算法运行时间复杂度分析.....8
2 程序代码说明.....9
2.1 数据结构说明.....9
2.2 函数说明.....9
2.3 程序限制.....11
3 实验结果.....12
3.1 测试数据.....12
3.2 结果分析.....15
3.3 经典战局.....16
4 实习过程总结.....12
4.1 分工与合作.....18
4.2 经验与教训.....21
4.3 建议与设想.....22
5 致谢.....22
6 参考文献.....22

1 算法思想

1.1 总体思路

1.1.1 估值函数

在“二〇四八”游戏中，衡量某一棋盘局面的估值涉及三个方面，当前每个棋子的级别、其对于棋盘局面是否有利于后续合并的评价，以及双方对抗过程中发生的吞并。后两方面的引入使得在决策树深度为三的前提下，模拟深度更高的决策（约为深度四至五层），得到更好的结果。

鉴于“二〇四八”游戏规则的特殊性，棋盘上每个棋子的价值并不能直接用它们的级别表示。我们采取的是以x为底数，以棋子的级别为指数的值，再在前面加上系数n，通过调整x和n的大小实现调参。（后文所涉及到的x和n均与此处含义相同）。对于我方和对方的棋子分别在前面加上相反的正负号，进行相加，得到最终的估值。

1.1.1.1 棋子的级别

为体现两个级别相同的棋子合并后价值大于合并前，我们使$x = 8$，后续观察对局情况发现，吞并对方棋子获得更大“领地”的一方占有优势，所以对于占领着对方领地的棋子进行额外赋值。

1.1.1.2 棋子的梯度

显然当棋面上有较多级别相同的棋子相邻时，通过一次合并可以获得很大的收益，当相邻棋子级别差距较大时不利于后续合并。所以我们用梯度表示相邻棋子间级别差的情况，己方梯度高意味着棋面较差。

1.1.1.3 吞并对方棋子

这是本算法中最复杂的一部分，我方吞并对方棋子、对方吞并我方棋子带来的影响是非常大的，级别较大棋子的吞并甚至可以一步改变比赛的结局。在这部分中，我们将 x 调整为 256 来体现其重要性。

而吞并过程本身是复杂的，下面举两个简单的例子。

图 1 吞并过程示例一（我方先手）

首先是最简单的吞并情况，如图 1，我方为先手，若对方向左合并，我方（2,4）的棋子可能被吞并。

图 2 吞并过程示例二（我方先手）

更复杂一层的情况，如图 2，我方为先手，我方向下合并可以吞并对方（2,4）的棋子，但下一步若对方向左合并，可以吞并我方此棋子，我方的合并操作是不利的。

合并操作之间还穿插着落子操作，使得问题更加复杂。我们针对“我方先手落子阶段”“我方先手合并阶段”“我方后手落子阶段”“我方后手合并阶段”，配合不同的落子操作，分别设计了不同估值规则，并考虑多层。当我方有棋子面临不利时，在估值中减去一定的值。

1.1.2 决策树

1.1.2.1 决策树的构建

决策树分为三层，针对不同阶段构建不同的决策树，每一层的含义如下表 1 所示。

表 1 决策树结构

对应阶段	第一层	第二层	第三层
我方先手，落子阶段	当前棋面	我方落子	对方落子
我方后手，落子阶段		我方落子	对方合并
我方先手，合并阶段		我方合并	对方合并
我方后手，合并阶段		我方合并	对方落子

1.1.2.2 minimax 值法和剪枝

按照 minimax 值法构建决策树，在第三层，站在对方的角度使用我们的算法，找出使自己估值最大时的情况（其余剪枝去除），在这些最大值中选择最小值，回溯第二层对应我方操作即为输出结果。

1.1.3 补充的算法策略

此部分内容用文字描述，并通过在决策树中增加细节来实现（具体见流程图），其中有若干个特殊名词将在流程图中出现，其含义与此处解释相一致。

1.1.3.1 “防守 2”

当我方棋子(x_1, y_1)可能被对方棋子(x_2, y_2)吞并时，有两种方式可以避免，第一种方法

是向某个方向合并，将此棋子从危险的位置挪开，第二种方法是在(x_1, y_1)和(x_2, y_2)间的对方领地落子“2”。这两种方法使用情况均有限制，第一种方法遇到合并后无变化（即不能合并）的情况不能使用，第二种方法遇到(x_1, y_1)和(x_2, y_2)相邻时不能使用。在我们的算法中这两种方法互相补充，并且为了鼓励占领对方领地，优先使用第二种方法。

1.1.3.2“进攻 2”

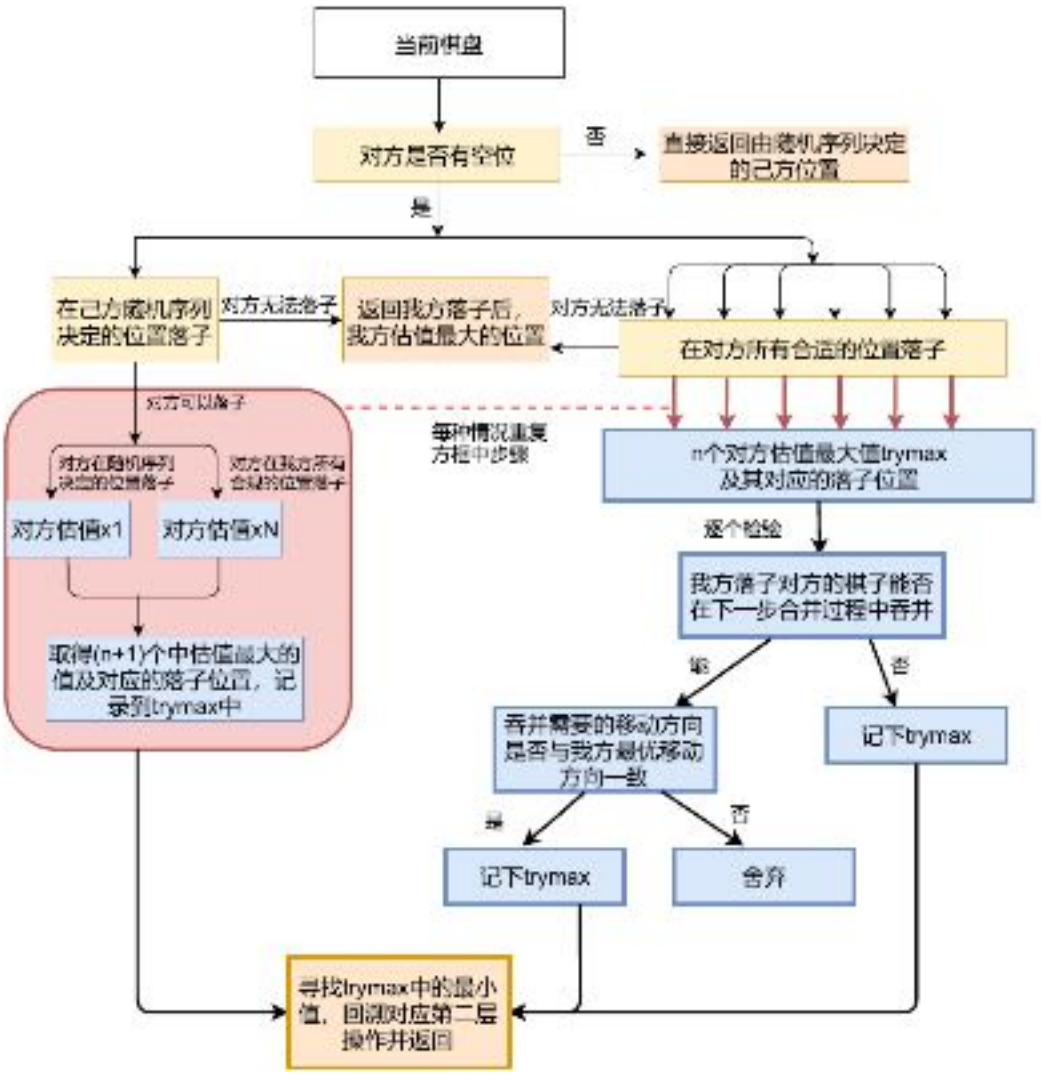
如果我方在对方领地落子，并可以在下一次合并阶段吞并这个棋子，相比直接在己方落子，可以起到多占对方领地的效果。

1.2 算法流程图

图例：

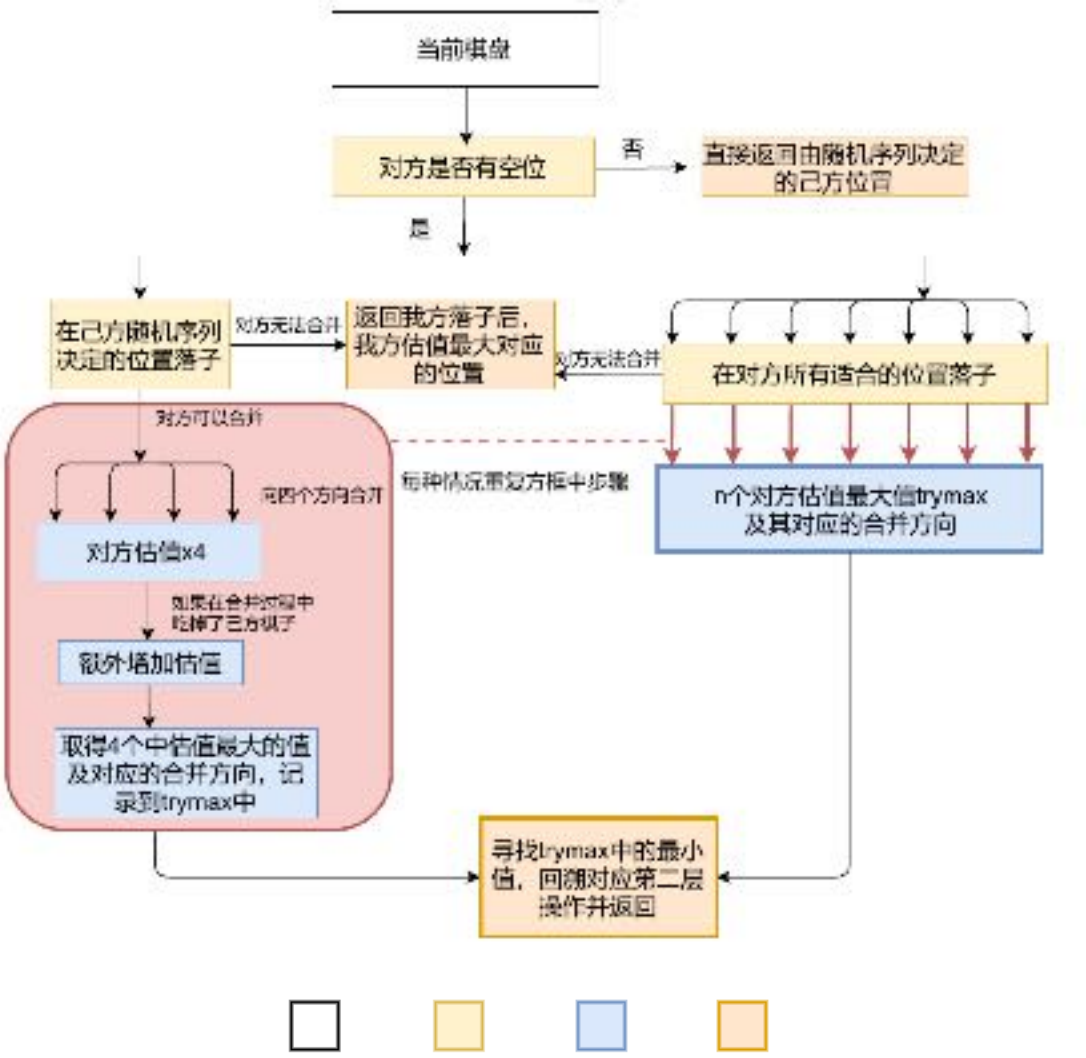
决策树第一层 决策树第二层 决策树第三层 搜索和返回操作

1.2.1 我方先手，落子阶段

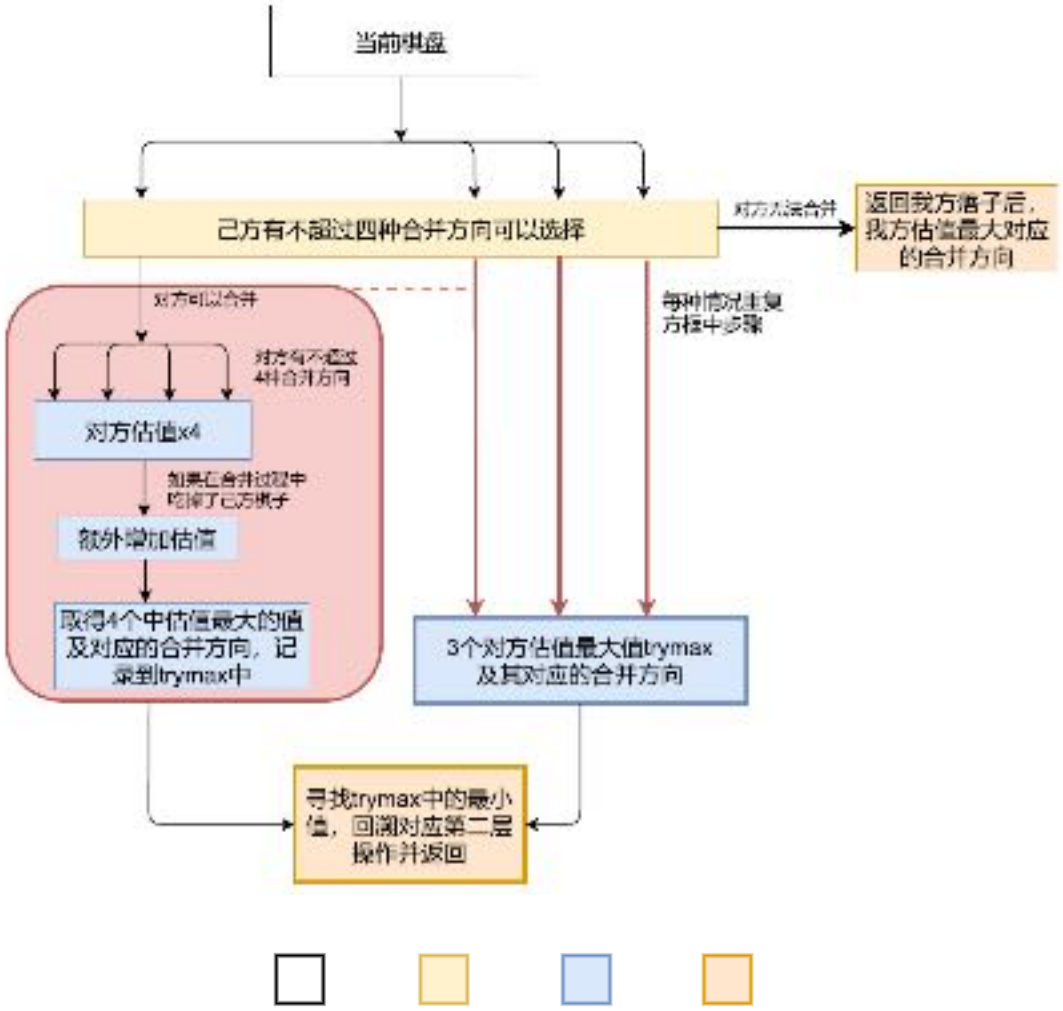


当前棋面 先手落子 后手落子 搜索和返回操作

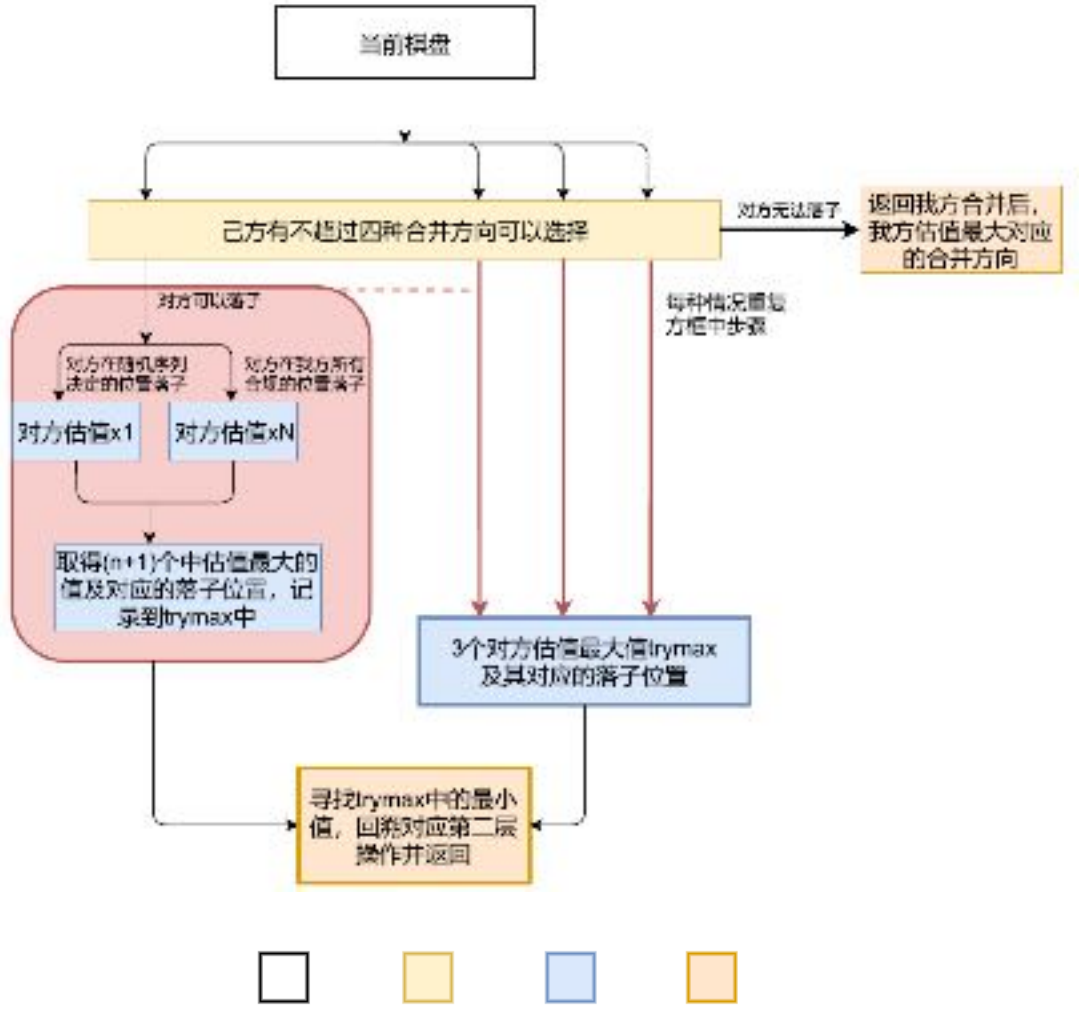
1.2.2 我方后手，落子阶段



1.2.3 我方先手，合并阶段



1.2.4 我方后手，合并阶段



1.3 算法运行时间复杂度分析

估值函数涉及到多次对棋面整体的扫描，并且其中的计算都是幂的计算，我们认为算法的运行时间将主要分布在估值函数上。对于决策树本身，因为深度为三层且剪枝及时，实际时间开销不会过多。本地测试的结果也证实了这一点，估值函数的运行时间占总时间的 90%左右，在估值函数中吞并函数又占了 60%左右。在天梯对战平台上，最终版本没有出现过超时的情况。

通过对代码的分析，假设在棋盘有一半位置被占据时，先手落子阶段复杂度的大 O 数量级为$O(n^3)$后手落子和后手合并的复杂度为$O(n^2)$，先手合并的复杂度为$O(n)$（n 为棋盘面积的一半）。其中估值函数的复杂度为$O(n)$，其余复杂度由决策树提供。

2 程序代码说明

2.1 数据结构说明

算法采用的主要数据结构为多叉树，不同于课堂中的链表，为了节约运算时间我们用字典进行的储存。用一个字典储存由一个点发出的所有子节点，key 值对应操作，value 值对应操作后的估值，直接通过在字典里搜索最大或最小值即可完成剪枝，操作性强且时间复杂度低。

2.2 函数说明

2.2.1 核心函数

1.`__init__(self, isFirst:bool, array:list)` 初始化函数

参数：`isFirst` 是否先手，为 `bool` 变量，`isFirst = True` 表示先手
参数：`array` 随机序列，为一个长度等于总回合数的 `list`

2.`state(self:Player, isFirst:bool, board:Chessboard, next_direction: bool, temp:int)->int` 估值函数

参数：`isFirst` 某方，`board` 当前棋盘，`next_direction` 下次执行合并操作的一方，`temp` 当前阶段（先手落子：1，后手落子：2，先手合并：3，后手合并：4，后同）
返回：`isFirst` 方的棋盘估值

与“总体思路”部分叙述一致，估值包括三部分，是整个棋盘对于 `isFirst` 方的估值。其中调用了 `grad` 和 `merge` 函数。对于 `isFirst` 方己方棋子，在棋子级别的基础上减去梯度，加上吞并收益值，对方棋子则在每一部分的值前面加上负号，再加入到估值当中。

3.`grad(self:Player, isFirst:bool, board:Chessboard, pos: Tuple[int, int])->int` 梯度函数

参数：`isFirst` 某方，`board` 当前棋盘，`pos` 位置坐标
返回：棋盘上位置坐标为 `pos` 的棋子对于 `isFirst` 方的估值
具体为：
（1）指定棋子周围与其级别差小于 3 的棋子数小于或等于 1 时，梯度增加。
（2）指定棋子周围有级别相同的棋子，梯度降低。
（3）在第二种情况基础上周围还有级别大 1 的棋子，梯度进一步降低。

4.`merge(self:Player, isFirst:bool, board:Chessboard, pos: Tuple[int, int], next_direction:bool, temp:int)->int` 吞并收益函数

参数：`isFirst` 某方，`board` 当前棋盘，`pos` 位置坐标，`next_direction` 下次执行合并操作的一方，`temp` 当前阶段
返回：一个衡量位置坐标为 `pos` 棋子被对方吞并可能性及吞并后损失程度的 `int` 类型值

相比于后文的 eat 函数，此函数考虑到了多轮合并操作的最终结果。如果我方可以吞并对方的某个棋子，并且在三回合内可见的不会被对方吞并回来，则返回一个比较大的值。同样，如果对方可以吞并我方棋子，并且在三回合内可见的不会被我方吞并回来，则返回一个非常小的负值。对于吞并后又被反吞并的情况，都返回 0。

5.output(self:Player, currentRound:int, board:Chessboard, mode:str)->Tuple[int, int] or int 决策树第二层搜索函数，给出己方的决策(下棋的位置或合并的方向)

参数：currentRound 当前轮数，为从 0 开始的 int
参数：board 棋盘对象
参数：mode 模式，mode = 'position' 对应位置模式，mode = 'direction' 对应方向模式，如果为 '_position' 和 '_direction' 表示在对应模式下己方无法给出合法输出
返回：位置模式返回 tuple (row, column)，row 行，从上到下为 0 到 3 的 int；column 列，从左到右为 0 到 7 的 int
返回：方向模式返回 direction = 0, 1, 2, 3 对应 上，下，左，右

6.trymax(self:Player, currentRound:int, board:Chessboard, isFirst:bool, mode:str)->int 决策树第三层搜索函数

参数：currentRound 当前轮数，board 当前棋盘，isFirst 某方，mode 模式
返回：第二层一个节点的所有子节点剪枝后，剩下的一个节点对应的估值

这个函数从决策树第二层的一个节点出发，搜索该节点的所有子节点中估值最大的节点并记录，将其他分支减去，传递到 output 函数进行汇总和第二层搜索。

2.2.2 辅助函数

1.eat(self:Player, isFirst:bool, raw_before: List[List[Tuple[int, bool]]], raw_after: List[List[Tuple[int, bool]]])->int 实际吞并函数，相比于 merge 函数，此函数考虑一个确定的合并过程

参数：isFirst 某方，raw_before 合并前表示棋盘状态的嵌套列表，raw_after 合并后表示棋盘状态的嵌套列表
返回：一个衡量 isFirst 方在一次合并操作后吞并对方棋子情况的 int 值

2._inside(self:Player, isFirst:bool, pos:Tuple[int, int]) ->bool 判断某个棋子是否在初始对方领地

参数：isFirst 某方，pos 位置坐标
返回：对于 isFirst 方，如果位置坐标为 pos 的棋子位于最开始 not isFirst 方的领地（且在边界上），则返回 True，否则返回 False

3.isBoundary(self:Player, isFirst:bool, board:Chessboard, pos: Tuple[int, int]) ->bool 判断某个棋子是否在边界上

参数：isFirst 某方，board 当前棋盘，pos 位置坐标
返回：如果位置坐标为 pos 的棋子属于 isFirst 方并位于边界上，则返回 True，否则返

回 False

4.around(self:Player, pos:Tuple[int, int])->List[Tuple[int, int]]

参数：pos 位置坐标
返回：pos 上下左右方的位置坐标组成的列表（不包括超出棋盘范围的）

5.isDangerous(self:Player, isFirst:bool, board:Chesboard, pos:Tuple[int, int])->bool 判断位置坐标为 pos 的棋子是否处于危险状态（可以被对方的一次合并操作所吞并）

参数：isFirst 某方，board 当前棋盘，pos 位置坐标
返回：如果处于危险状态则返回 True，否则返回 False

6.isDangerousStrict(self:Player, isFirst:bool, board:Chessboard, pos:Tuple[int, int])->bool 判断位置坐标为 pos 的棋子是否处于严格危险状态（与对方某个数值相同的棋子处于边界相邻状态，互相有吞并对方的可能性）

参数：isFirst 某方，board 当前棋盘，pos 位置坐标
返回：如果处于严格危险状态则返回 True，否则返回 False

7.clean(self, available: List[Tuple[int, int]], board:Chessboard, isFirst:bool)-> List[Tuple[int, int]] 对不合适坐标位置的剔除函数，考虑落子在对方内部时，仅考虑可以使对方梯度增加或在对方边界上的位置

参数：isFirst 某方，board 当前棋盘，available 是 not isFirst 方空位坐标组成的列表
返回：剔除不符合规则的位置坐标后的列表

2.3 程序限制

最终版本的代码在天梯对战平台上没有出现过报错，但出现过未按照我们的想法落子或合并的情况，经分析原因如下：

1.由于使用了 minmax 方法构建决策树，对手算法与我们算法决策上的一致性在一定程度上影响我们算法的有效性。当对手不按照我们的决策落子或合并时，会出现问题。比如下图中的情况，我方为先手，在对方的（3,4）位置落子显然不是很合理。

图 3 不合理情况示例（我方先手）左：我方落子前 右：我方落子后

2.在我方先手落子阶段时，为降低时间复杂度“我方最优移动方向”（见流程图）未考虑对方的落子，在大多数情况下不会出现问题，但偶尔会出现不按照我们思路合并的情况。

3 实验结果

3.1 测试数据

实验环境说明：

- 硬件配置：Intel i7-7500U CPU @2.70GHz 2.90GHz ， RAM:8.00GB
- 操作系统：Windows 10 家庭中文版
- Python 版本：Python 3.8.1

3.1.1 胜率

我们在编写最初几个版本的算法时，首先是通过和默认 AI，也就是随机 AI 进行对战来验证程序有效性，以初步确认决策与算法是优于随机的。

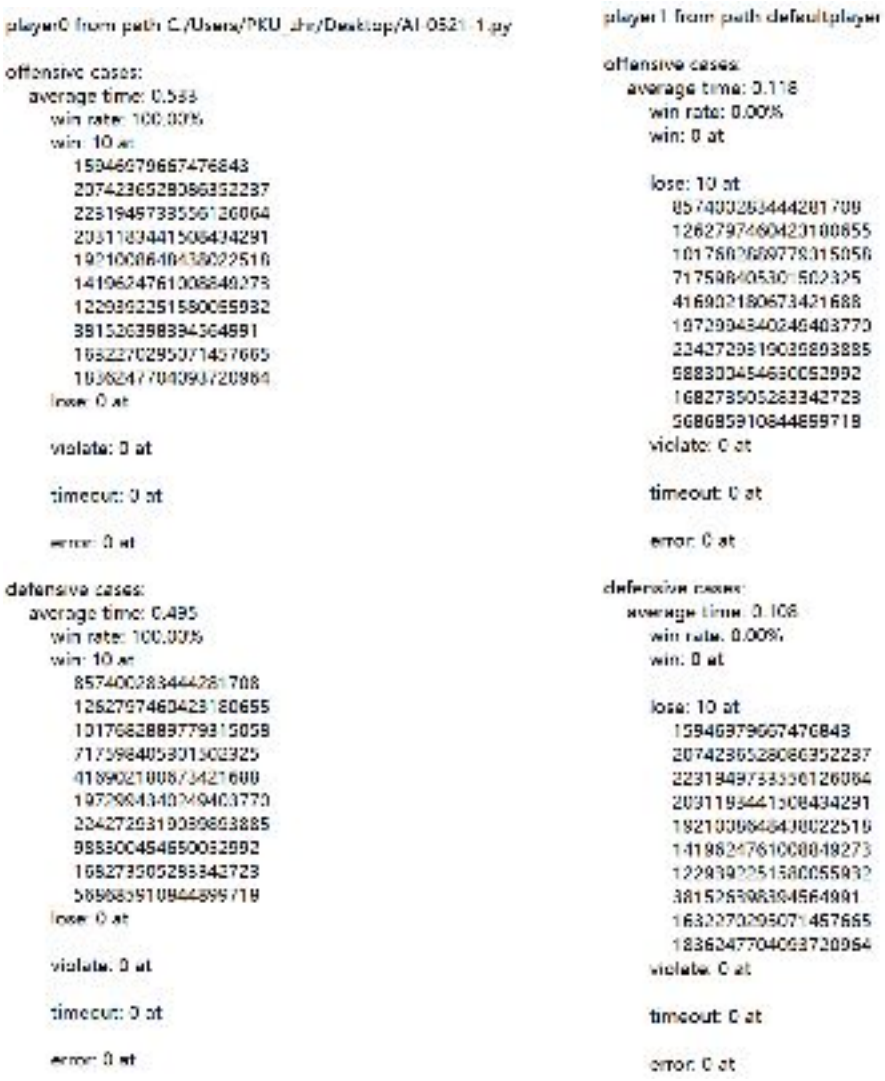


图 4 历史代码与默认 AI 的历史对战结果

从上图也看出来，初始代码的复杂度是比较低的，在本地执行一局平均仅在 0.5 秒上下，

而这样复杂度的算法已经能较为稳定地战胜随机算法了。所以之后进行的改进，使用与随机AI对战的方法也就无法满足需求了。于是我们在本地通过各版本之间进行对战的测试方法，来进行进一步的优化筛选。

图 5　0521-3 版本与改进版本之间的本地对战结果

如上图所示，左边为改进版本，从胜率来说是明显优于原始版本的，说明这一次的算法优化与改进是有效并且可以采用的．在热身赛开始之前，我们主要采用这种方法来进行测试．当热身赛开始之后，我们几位组员会把在本地无法快速测试出的代码上传，根据热身赛结果评判算法优劣．我们最终几个版本算法的对战结果如下所示。

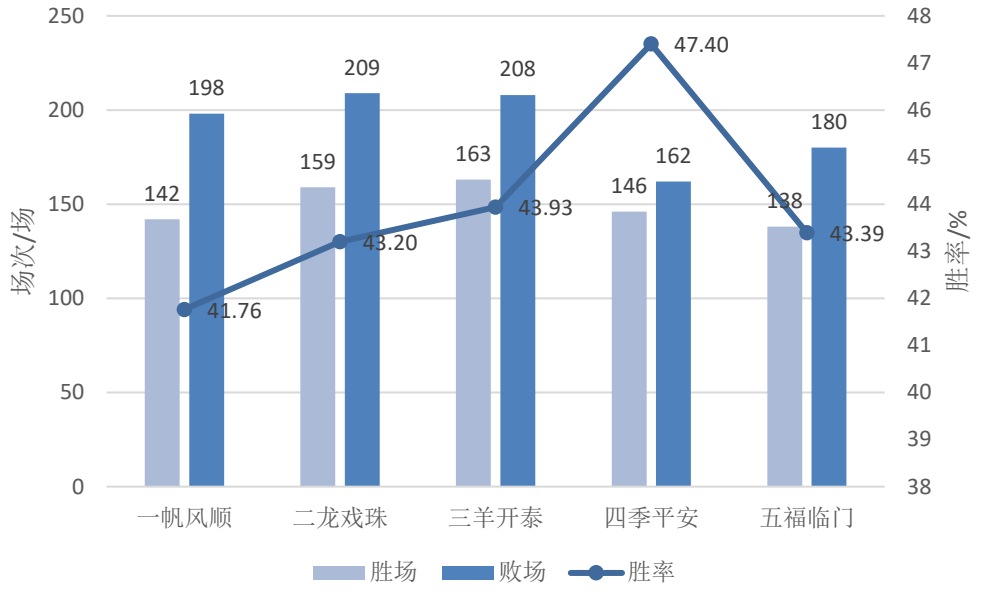


图 6　最终五个版本代码热身赛对战结果

其中，每个版本的代码相互之间进行对战，并与 Bot A，S，SS 等官方 AI 进行对战，综合以上结果，选出了”四季平安”作为最后的提交代码，在天梯平台上所用总时间大概在 4 s 左右，基本达到对战强度要求。

直观起见，我们列举代码与 Official Bot 的对战结果若干，可以看到各代码的战力比较优秀，应付 Bot A 与 S 效果较好，甚至可以与 Bot SS 势均力敌。

图 7　最终五个版本代码与 Official Bot 对战结果

3.1.2 时间开销

在我们的代码当中，主要运算包括决策树和估值函数两部分，估值函数中又包含梯度函数和吞并函数，在这里通过屏蔽这几个重点函数，在本地测试与随机 AI 对局时计算时间，大致判断各步骤的时间开销。每组数据均是对局 500 回合所花费的总时间。

如果把完整版程序和屏蔽某个函数后的运行时间的差作为此函数的时间开销，会发现梯度函数和吞并函数时间的总和大于估值函数，这从直观上是不合理的。出现此种现象的原因主要在于函数的屏蔽直接改变了棋盘状况。因此这部分实验结果只能作为一个粗略的参考。

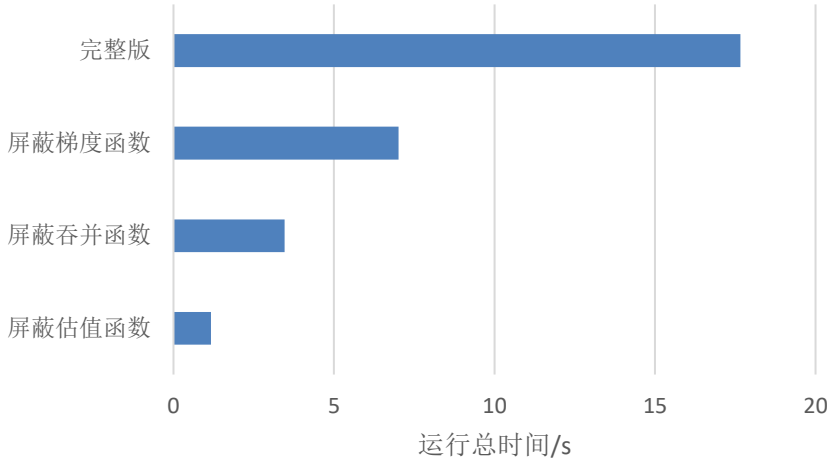


图 8 屏蔽重点函数后 500 回合运行总时间

由于天梯对战平台有加速系统，算法 500 回合的用时大约在 4.2s，没有超时风险。

3.2 结果分析

总的来说，本算法在实战中的表现是较为不错的，在 N19 小组中取得了第八名的好成绩，在此过程中，有一些算法是起着关键作用的.首先，我们的”先手进攻 2”和”防守 2”策略是取得良好效果的，即作为先手，在不被吃掉的前提下，我们可以在对方边界下”2”，然后凭借先手优势在下一次移动时右移吃掉”2”，这样我们在对方边界处有一个”4”，这可以阻碍对手的合并移动，造成干扰效果;而”防守 2”，即当对方是下一次移动方，且我方有较大价值的棋子可能被对方下一次吞并时，如果二棋子中间有空位，那么在对方空位处下”2”即可防止价值的损失，这在实战当中是屡试不爽的。

此外通过 eat 函数，鼓励自己合并和吞并对面，通过 grad 函数使得我方相同棋子尽量靠近从而利于合并等等，众多的估值函数相互作用，产生了奇妙的反应。

至于运行时间，在打满 500 回合 2000 次操作的情况下，在天梯对战平台上用时大约在 4.2s，这是在我们预期之中的，既没有超时的风险，又尽可能最大化代码的时间复杂度，我认为是相当好的结果.本地测试结果表明,主要的时间开销花在对于局面的估值函数 state 上，因为里面涉及到了多次的全局搜索，包括对于每个棋子的周围梯度，每个棋子是否受吞并威胁等，实际上这也是我们的主要策略所在，即可以适当减小决策树的高度，而尽量对局面的估值做到透彻以及有远见。

3.3 经典战局

在正式的分组对战中，我们与 N19 战区的小组第一进行了多次比拼，虽然大多落败，但是有某些战局我们的代码有着优秀的表现，与榜首打成平手。

我们选取此次对局中的第 2 场进行观赏，我们为先手方。

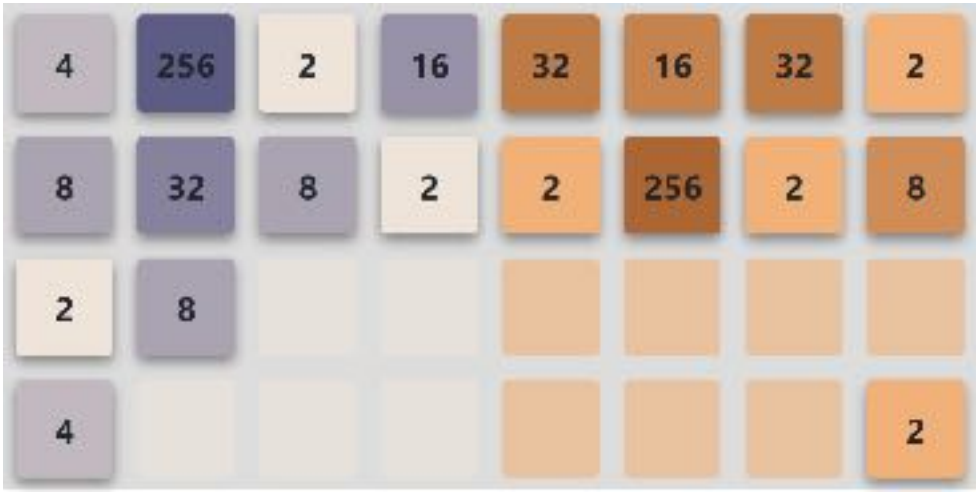
在这一步，我们要进行移动方向的选择，如果仅仅是按照棋子价值评估的方法，我们可能会选择向左移动，这样两个 8 合成 16，且没有被后手吞并的风险。

但是我们看到，我们的 AI 选择向上移动，此时我们发现，对方(2，4)的 8 正在我们的威胁之下，对方若不选择向上逃跑，那么我们将吞并掉这个 8.之所以我们会做出向上移动的决策，是因为 merge 函数里，将”对对方产生吞并威胁”的赋值较大，并且在决策树中，对”吞并对方棋子”的赋值也较大，这使得我们作为先手，富有攻击性。

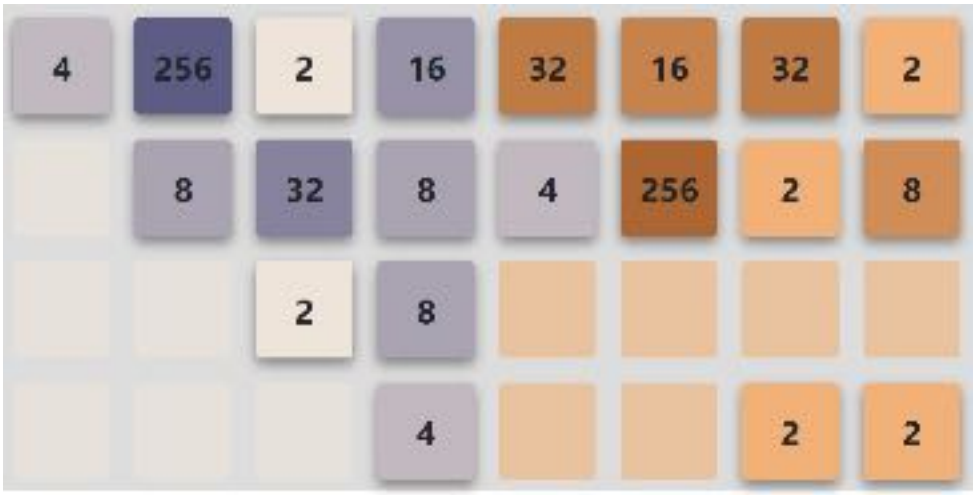
随后，后手未选择逃跑，于是我们将对方的棋子 8 收入囊中，且我方没有棋子受到威胁。

让我们再看下一个局面:

此时轮到我们先手落子，看一看我们代码的选择:



我们的算法选择将棋子 2 下在对方的边界(1，4)处，这是因为作为下一次移动方，这个 2 不会被对方移动，且我们能够在下一步吞并这个 2 并且不受威胁，又可以占领对方一块领地。



我们看到，在这样的吞并后，形成了良好的效果，对方(0，4)处的 32 被困在角落，这就是我们尽力吞并对方领地的意图所在，干扰对方的移动会使得我们受益，有时会产生意想不到的效果。

4 实习过程总结

4.1 分工与合作

4.1.1 小组分工

组长姜治仲同学主要负责代码方面。在前期负责代码的整体框架搭建，中后期负责代码中很多功能的实现和优化，同时也在天梯上测试代码并复盘纠错。

冯子琦同学在前期负责研究这个 2048 对战游戏胜利的套路并总结出来，在前期的代码

中提供了一些启发式思想，中后期负责在天梯上大量对战，测试代码更换不同参数的结果，并复盘纠错，本地人机对战测试本小组的代码哪些功能还没有实现，并参与了一部分报告的编写。

张皓然同学主要负责一些功能的实现和优化，同时也在天梯上测试代码并复盘纠错，并参与了一部分报告的编写。

李宇轩同学在前期与冯子琦同学双人对战来研究这个 2048 对战游戏胜利的套路，后期主要负责报告的统筹规划及编写，并且将代码的思路整理了出来。

小组合作与交流的方式有两个：微信与腾讯会议，微信主要是平时分享对战结果、代码纠错以及大量的讨论。腾讯会议或者微信电话是每一次组会，四人在线讨论，解决了很多问题。

4.1.2 历次组会记录

第一次组会在 2020 年 5 月 14 日，以微信电话的方式进行，冯子琦同学首先给大家展示了单机 2048 怎样得高分。我们小组首先阅读并讨论了游戏规则，观看了天梯上面高手的对战，就这个游戏获胜的策略进行了讨论，并用共享文档的方式写出了前期的思路，确定了代码所用的算法与大体框架。

第二次组会在 2020 年 5 月 24 日，以腾讯会议的方式进行，这 10 天中，在小组同学共同努力下，代码的框架已经形成，部分功能也已经实现。并且以 100%的胜率战胜了随机移动的白痴 AI，但是在天梯热身赛中出现了一些 bug：无缘无故给对面落子；在先手下进攻 2 时没有考虑合并后被后手反吃的风险；在轮数多了之后有超时的情况。第二次会议主要是为了解决这个问题，在三个小时的讨论和修改中，我们解决了部分问题，排除了一些错误的答案和思路，最主要的成果是确定了估值函数 state：鼓励合成大数；警惕边界被吞并；考虑我方内部的平滑性。

第三次组会在 2020 年 5 月 25 日，以腾讯会议的方式进行，第三次组会建立在第二次组会的基础上，增加了合并的 merge 函数，解决了无缘无故给对面下棋的 bug，让代码的战斗力更进一步。“他山之石，可以攻玉”，我们在观看了当时天梯第一名 RBQ-4 的比赛后，启发了我们，我们之后主要采用了进攻为主的思想，在对方内部的赋值增加，没有威胁的话是占住对面领地不动的，这样压缩了对面的空间，对面更容易憋死。

第四次组会在 2020 年 5 月 26 日，以腾讯会议的方式进行，第四次组会主要是针对我们代码在天梯热身赛上的表现提出了一些优化，是投放到正式天梯前的一次大讨论和大改：正式确定下来落子阶段“防守 2”和“进攻 2”的逻辑和次序；重写了 merge 函数，在时间复杂度的允许上增加了 merge 的搜索深度；增加了 eat 函数，鼓励自己合并和吞并对面；并且针对 grad 函数的赋值进行了讨论和实验。随后与官方 BOT 进行对战，找我们存在的问题。

第五次组会在 2020 年 5 月 26 日，以腾讯会议的方式进行，第五次组会发现了代码中一个正负号的小 bug，并且发现了内部混乱的问题；我们针对这个问题对 grad 函数进行了修改，加入了更多的估值来确保内部的平滑性。

第六次组会在 2020 年 5 月 31 日，以微信电话的方式进行，第六次组会是在天梯封闭

前开的，主要讨论的内容就是在热身赛中测试不同参数的代码，选出表现最好最稳定的参与天梯排位赛。大家在这个会议中都积极地在热身赛中测试不同参数地代码，最后选除了最合适的代码参加 24 小时的天梯排位赛。

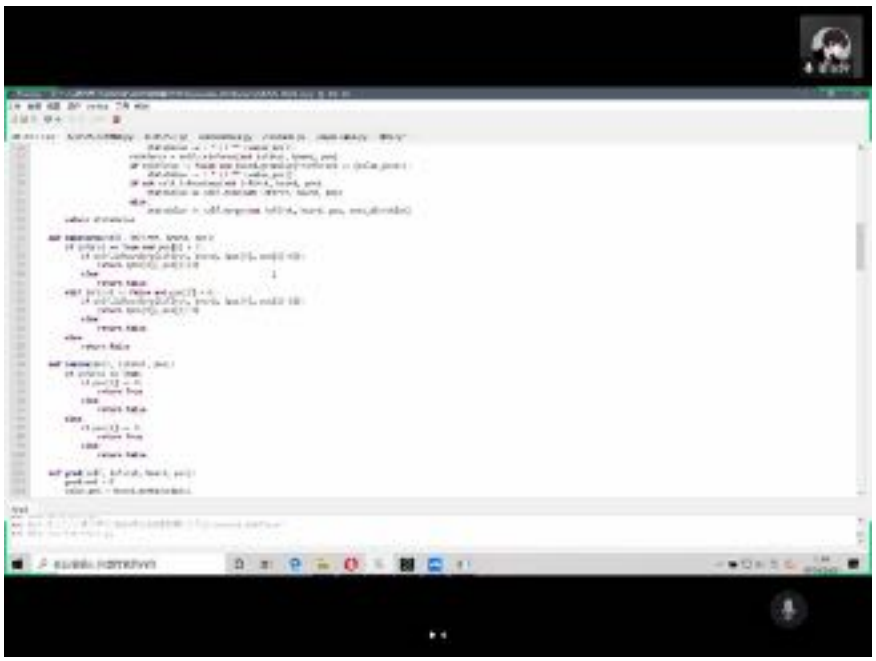


图 9(1) 使用腾讯会议讨论代码

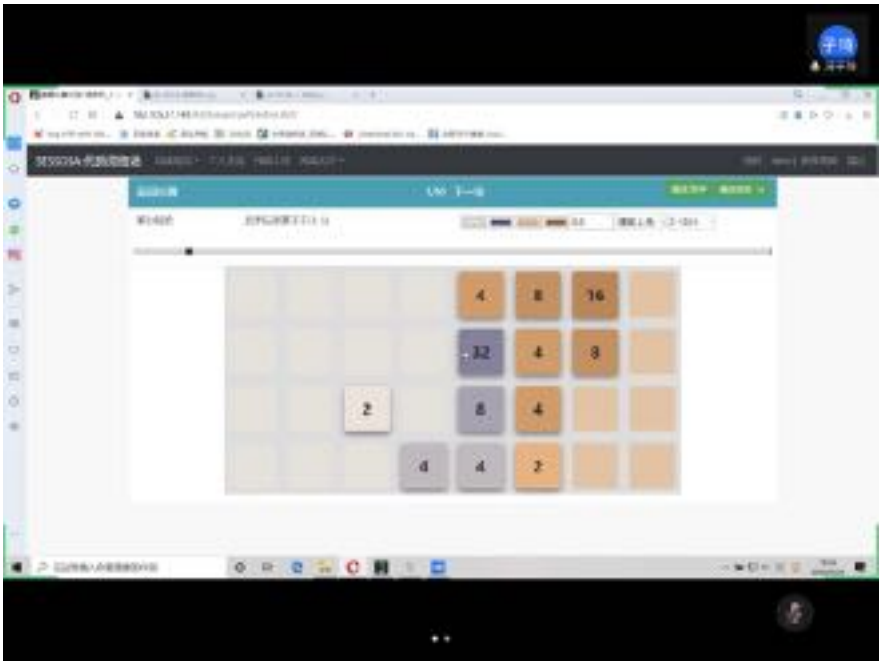


图 9(2) 使用腾讯会议分析战局

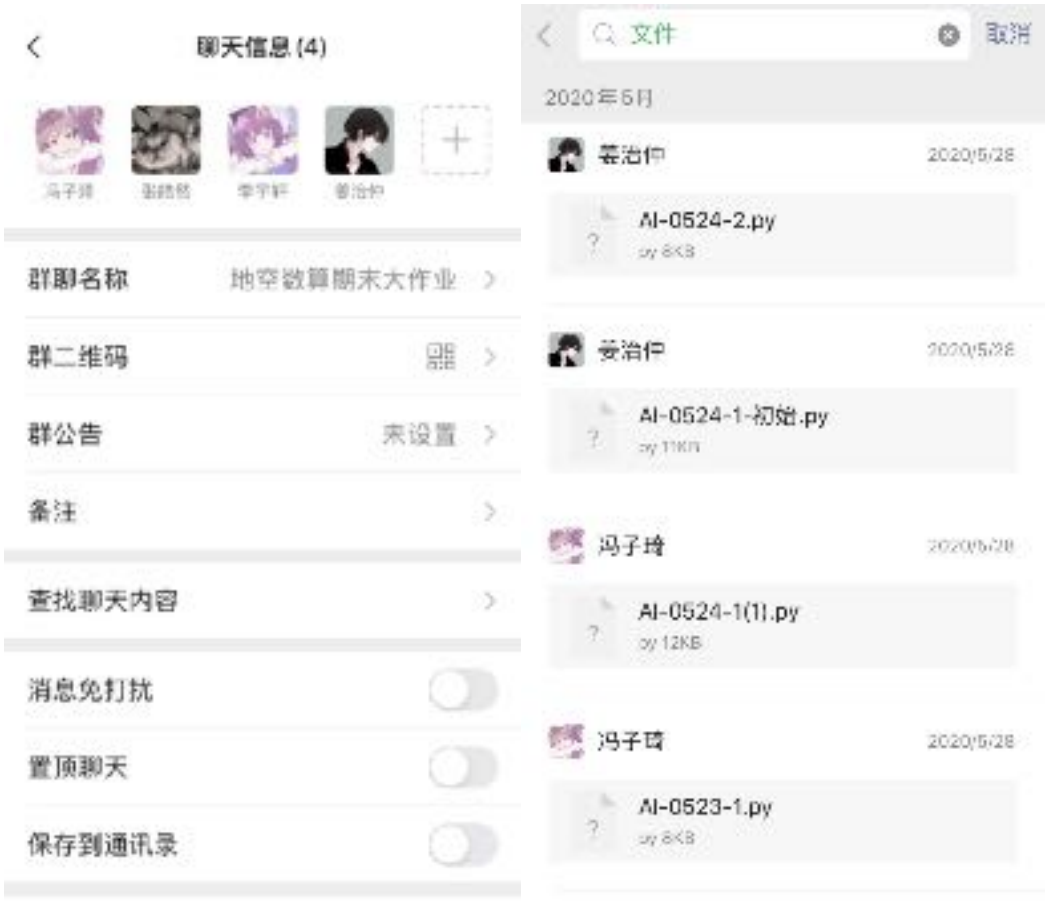


图 9(3) 使用微信群讨论

4.2 经验与教训

我们小组的同学彼此都很熟悉，所以在讨论中的氛围很好。我们组在进行核心部分的工作期间讨论频率很高，大家都积极参与，提出自己宝贵的意见，我们的分工是非常合理的，每个人都最大化了自己的价值，把每个人的长处都结合在了一起。每一次讨论虽然用时很长，但是大家都用心投入到工作中去，解决了许多关键的问题。

总结的工作经验：

1.一个小组要明确分工，找到每个人适合的工作；同时也要重视小组合作和讨论，“众人拾柴火焰高”，擅长写代码的同学去写代码，擅长思考和策略性游戏的同学去出谋划策。

2.编写代码的同学要注意代码的规范性和适当加以注释让代码通俗易懂，便于找到问题所在，我们组在这方面还是很好的。

3.“他山之石，可以攻玉”，要多向高手学习，研究他们胜利的过程和因果，自己才会变得更强。

4.一开始的思路和框架要搭的尽可能大，方便后面的填充、优化。

5.因为这个游戏的特殊性，大家的代码可以大致分成防守型和进攻型，因此在天梯赛和淘汰赛选用什么样的策略也非常重要。

可以改进的地方也有不少：例如在代码思路上，我们做的还是不够完善，还是有些实用

的功能没有实现，一些发现的问题也没有很好的解决。因为时间有限，我们采取的算法可能也不是最好的。代码的框架有时候也是一个制约，我们每一次版本更迭时改进的部分可能不是那么明显，还是要围绕代码的中心思路去调整，有时候显地无能为力。

4.3 建议与设想

本次实习作业的建议：我们认为这个对抗 2048 的游戏非常有意思，比之前的游戏更加注重逻辑，随机序列和时间限制的加入也让比赛变得充满戏剧色彩。本次实习作业在组队、基础设施代码、竞赛等方面我认为都做的很好，就是这个游戏本身是棋类游戏，大家不同的策略可能是相互克制的，采用淘汰赛制希望可以设立胜者组、败者组采用双败淘汰制，比赛会更加精彩。

寄语：希望之后选这门课的学弟学妹要认真对待大作业，不仅是一次很好的团队合作，展示自我的机会，还能学到课本上学不到的很多东西。面对新事物不要恐惧，要学会与团队讨论，分享自己的看法，吸纳他人的优点。不仅有大作业，DIY 分形树和 microbit 创意作品制作都是发挥你们无限创意的空间。这是为数不多逃离死记硬背的标准化考试还能大有收获的机会，要好好享受。陈斌老师上课的氛围严肃活泼，助教还有群里大佬随时都能解答你的疑惑，数算的收获是任何一门死记硬背的课都无法比拟的。

5 致谢

感谢陈斌老师提供这次实习大作业的机会，并为我们讲解了很多关于对战策略和深度学习的知识。

感谢技术组的助教和同学们为我们搭建的对战平台以及附带的功能丰富的 api。

当然最应该感谢的还是我们实习小组的每一名组员，在代码构建以及后续调试过程中付出了很多时间和精力。

6 参考文献

http://jluyeyu.leanote.com/post/极大极小搜索算法

数据结构与算法课程实习作业报告

韩泽尧* 李煦东 杨佳宇 刘金禹 王之昱

摘要：主要思想是为局面设置一个估值函数，使用 minmax 算法搜索最优策略，使用 αβ 剪枝增加搜索层数。此外，使用在对方空格较多的时候只在己方落子的启发式规则；用当前剩余时间和回合数决定搜索层数，在不超时的前提下充分利用思考时间；对落子和移动操作使用不同的估值函数等。

关键字：minmax 算法、αβ 剪枝、决策树

1 算法思想 1
1.1 总体思路 1
1.2 算法流程图 2
1.3 算法运行时间复杂度分析 2
2 程序代码说明 3
2.1 数据结构说明 3
2.2 函数说明 3
2.3 程序限制 7
3 实验结果 7
3.1 测试数据 7
3.2 结果分析 8
4 实习过程总结 8
4.1 分工与合作 8
4.2 经验与教训 10
4.3 建议与设想 11
5 致谢 11
6 参考文献 12

1 算法思想

1.1 总体思路

首先为棋局设计一个估值函数，用来评价现在的我方局面。估值函数值越大，说明我方的局面优势越大。我方进行落子、移动的决策时，总是倾向于最大化估值函数；但是在一步操作后保证了估值函数最大化，不一定能够保证数步后估值函数依然是最大的，于是在决策时需要多考虑几步。

每次我方的决策都尽可能的最大化估值函数，并假定每次对手的决策都尽可能的最小化估值函数，依此在每次决策的时候都考虑若干步以后的局面，再进行决策，以保证若干步之后的估值函数是最大的。

此外，在搜索过程中，使用 αβ 剪枝来尽可能减少不必要的搜索过程，再结合一些启发

式规则，尽可能节约时间，来保证足够深的搜索深度。同时，再针对剩余回合数和时间来灵活设置搜索深度来避免超时或是时间的不充分利用。

1.2 算法流程图

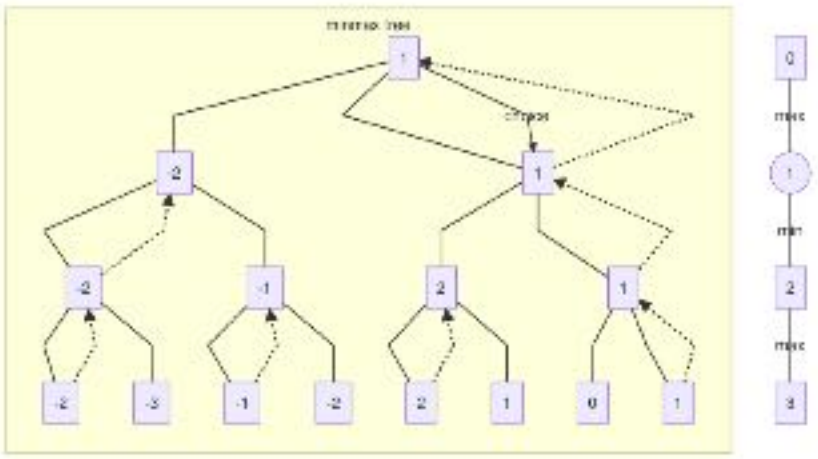


图 1：minmax 树示意图

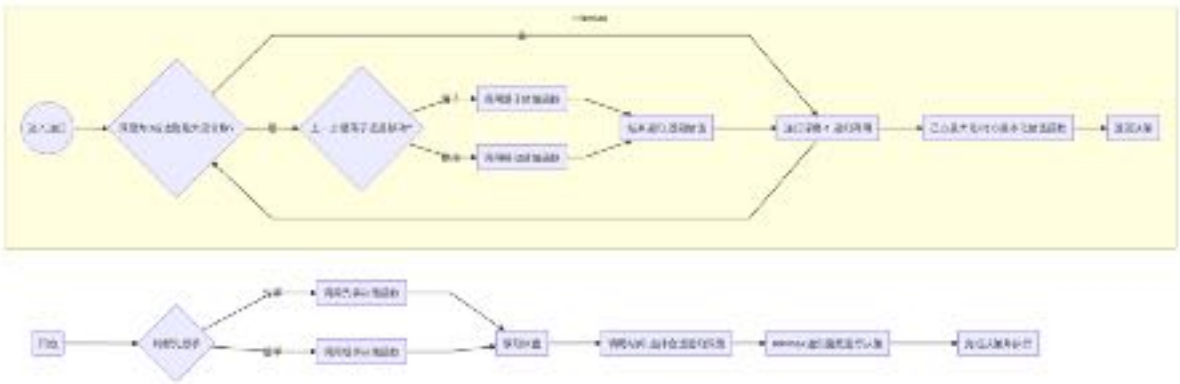


图 2:算法流程图

1.3 算法运行时间复杂度分析

主要复杂度在于 minmax 决策中的递归搜索。在棋盘上有 n 个可落子位置时，进行搜索搜索要考虑 4 种可能性；在移动的决策中一般会有 4 种可能的情况。如果搜索 h 层，大致会有 h/2 次是对落子操作的搜索，有 h/2 次是对移动操作的搜索，并假定棋盘上可落子数目无显著改变，时间复杂度大约为$O[(4n)^{h/2}]$.

我们虽然采取了 αβ 剪枝，减少每层不必要的搜索，使时间复杂度大幅减少，但是对于搜索深度 h 而言，仍然是指数级别增长。

在对局中，如果对方空格多于 4 格，我们就不考虑在对方棋盘上落子，这启发式规则可以极大的限制 n 的大小，复杂度大致减少到$O(4^h)$.

可见在该算法中搜索深度 h 会显著的影响决策用时。虽然搜索深度越深，决策的效果越好，但是，实际战斗中思考时间有限，要尽量充分利用时间，但同时也不能超时。所以我们采取了对搜索层数进行了实时的调节，根据剩余每回合的平均时间来调整每次决策的搜索深度。在这样的时间管理策略下，整个对局既不会轻易超时，也能充分利用 5s 的决策时间。

2 程序代码说明

2.1 数据结构说明

程序并未直接使用各种数据结构，但在代码中运用了一些数据结构的思想。在 minmax 决策中使用了递归函数调用，实际上是对线性结构栈的应用。在搜索中实际上应用了树结构的思想，不同的决策作为树的不同子节点，最后再从叶节点逐层决策返回到根节点。

2.2 函数说明

初始化：传入先后手，棋子权重，最大局数，最长思考时间，开启时间管理的回合数阈值。
getActions：传入棋盘、当前回合、当前玩家，返回该玩家该回合所有可能的落子位置，其中在对方落子的所有位置返回为一个列表。
getDepth：传入当前回合和剩余时间，返回决策的递归深度，用于时间管理。
weightSum：传入棋子分数列表，返回他们的权重之和，用于局面估值。
scoreMove：传入棋盘，返回移动操作后的局面估值。
score：传入棋盘和当前回合数，返回落子操作后的局面估值，用于决策。
_minMaxRecur：递归调用，返回若干次递归后 minmax 决策下的局面估值，用于决策。
minmaxDecisionF：先手决策函数，传入棋盘、当前回合和模式，返回最终决策。
minmaxDecisionNF：后手决策函数，传入棋盘、当前回合和模式，返回最终决策。
output：根据我方先后手分别调用各自决策函数，返回最终决策。

其中核心递归函数代码如下：

```
def _minMaxRecur(self, board, depth, phase, currentRound, alpha=-(2E9), beta=+(2E9)):# 返回值为局面估值
# phase 0 1 2 3，分别对应每一回合的先手落子、后手落子、先手移动、后手移动四次操作
    if depth <= 0 or currentRound >= self.maxRounds:
        if phase == 1 or phase == 2: # 是落子操作的下一步
            return self.score(board, currentRound)
        else:   # 是 move 之后的下一步
```

```
                return self.scoreMove(board)
        peer = not bool(phase % 2)  # peer 表示当前操作的一方
        flag = bool(peer == self.isFirst)  # flag = True 时为自己回合，False 时为先手方回合
        if phase == 2 or phase == 3:
            if phase == 3:
                currentRound += 1
            for d in [0, 1, 2, 3]:  # 可以改顺序 方便剪枝
                newBoard = board.copy()
                if newBoard.move(peer, d):
                    curScore = self._minMaxRecur(newBoard, depth - 1, (phase + 1) % 4, currentRound,
alpha, beta)
                    if not flag and (curScore < beta):
                        beta = curScore
                        if alpha >= beta:
                            return alpha
                    elif flag and (curScore > alpha):
                        alpha = curScore
                        if alpha >= beta:
                            return beta
        else:
            nextMove, posLst = Player.getActions(currentRound, board, peer)
            if nextMove is not None:
```

```
                posLst.insert(0, nextMove)  # 插在最前面，方便 alpha-beta 剪枝
            if len(posLst) < 5:  # 对方棋盘上可以下的空比较少时才考虑向对方棋盘上落子
                for pos in posLst:
                    newBoard = board.copy()
                    newBoard.add(peer, pos)
                    curScore = self._minMaxRecur(newBoard, depth-1, phase + 1, currentRound, alpha,
beta)
                    if not flag and (curScore < beta):  # alpha-beta 剪枝
                        beta = curScore
                        if alpha >= beta:
                            return alpha
                    elif flag and (curScore > alpha):
                        alpha = curScore
                        if alpha >= beta:
                            return beta
            else:
                newBoard = board.copy()
                newBoard.add(peer, posLst[0])  # 空格比较多时，能下自己这儿下自己这儿，下不了自己
这儿下第一个对手空格处
                # 这个设计很容易导致自己被卡死（即容易被对手一连塞爆几十回合无法脱身），但经过对
比，多搜几层带来的好处更多
                curScore = self._minMaxRecur(newBoard, depth-1, phase + 1, currentRound, alpha, beta)
                return curScore
```

```
        return alpha if flag else beta
```

另外两个估值函数的函数代码如下：

```
def score(self, board, currentRound):   # 用于落子的估值函数
        myScoreLst = board.getScore(self.isFirst)
        if board.getNext(self.isFirst, currentRound) is None:
            total = -self.weight[myScoreLst[-1]] /3.0   # 无空可走的情况酌情扣分
        else:
            total = 0
        RivalScoreLst = board.getScore(not self.isFirst)
        total += self.weightSum(myScoreLst) - self.weightSum(RivalScoreLst)
        return total   # 估值函数考虑了己方棋子分数、对方棋子分数和己方是否被卡死



def scoreMove(self, board):   # 用于 move 操作的估值
        myScoreLst = board.getScore(self.isFirst)
        RivalScoreLst = board.getScore(not self.isFirst)
        total = self.weightSum(myScoreLst) - self.weightSum(RivalScoreLst)
        # move 操作的估值函数中所有空格有额外两分
        total += (len(board.getNone(self.isFirst)) - len(board.getNone(not self.isFirst))) *2.0   # t
        return total
```

为避免占用篇幅过长，只给出比较核心的代码。更多代码详见另附的程序文件，或者通过文章后面的 GitHub 链接访问。

2.3 程序限制

观察对局，发现三个明显缺陷：
1. 如果已经通过吃子进入对方棋盘，而后一步为了使分数最大又再一次深入对方阵地，就会导致这个棋子看起来属于自己，实际上无法回来，在数个回合之后一定会被对方吞并，实际上造成分数损失（这个缺陷应该各种算法都有可能存在，除非进行过针对性的优化，而且某些情况下可以达到干扰对方的作用，但是整体上弊大于利）。
2. 决策时在对方空格较多的时候直接选择在己方棋盘上落子，有时候会很容易被对方卡死，从而无法取胜，甚至在卡死之后被对方胁迫，不得不给对方吃子。但这样的策略也有好处，因为更多的时候在对方空位较多时往往都会下在己方，因此可以减少很多没有必要的搜索，大大地节约时间，保证其它操作有更深的搜索深度。
3. 没有考虑盘面平滑性，有些时候在随机序列并不理想时，会出现几个比较大的子相互隔开无法进一步合并的情形。但棋子权重的设置略微弥补了这一点，不过总体上还是会出现这样的无法合并的情形。

3 实验结果

3.1 测试数据

前期代码的编写和参数的调整由组员各自在自己电脑上进行，在主要代码成形之后，每个人在各自调整参数后都在本地和之前的代码进行内战，选取出最优的参数。

为了贴合实战，中后期所有的测试全部在天梯平台上用个人账号进行，各自调整代码或者参数后在天梯上和旧版本、其他同学的代码进行实战测验，根据实战效果来决定是否对代码进行这样的调整。

因为各个组员各自都参与了修改代码、测试自己的想法的过程，故在小组天梯赛正式开始前进行了大量的尝试，测试后效果良好的代码改动会上传到 GitHub 上，因此可以从 GitHub 的 commit 历史中看到一步步测试完善的过程。在小组天梯赛开始后代码只有细微的改动，通过每天的系统自动发起的比赛结果来对改动作出评判调整。

下面截取本组在三个时间点的对战情况：

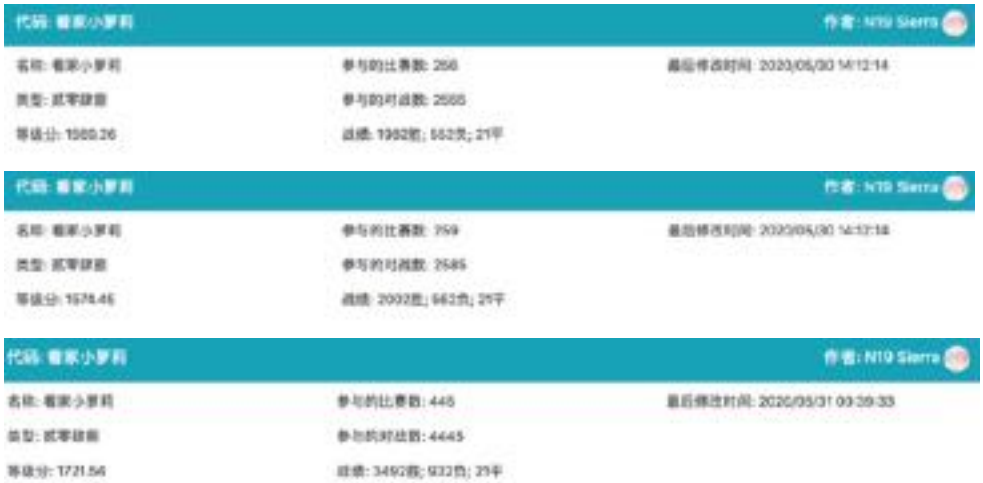


图 3:小组赛战绩

在之后的天梯赛和淘汰赛中，本组代码均表现良好，最终获得 N19 赛区冠军，在友谊赛中输给了 F19 第一名。

3.2 结果分析

在我们的算法使用了如下策略：

策略	效果
相同棋子合成高一级棋子后分数变高（1+1>2）	鼓励合并棋子，不容易卡死，容易获得高级棋子
对落子时对是否卡死额外扣分，在移动时对空格数目额外计算分数	防止卡死，容易发现卡死对方的操作
根据剩余时间控制递归深度	有效利用时间，不至于浪费时间或者超时
递归中进行 αβ 剪枝	排除明显劣势策略，减少运算时间，加大搜索深度，得到更优策略
落子时如果对方空格较多且能下在己方则不用搜索，直接下在己方位置	避免不必要的搜索，节省时间，从而加大其它操作决策的搜索深度

主要时间开销在于 minmax 递归，具有指数复杂度。通过根据剩余时间控制迭代深度进行时间管理，我们基本上不会超时而且一局总使用时间基本在 4s 以上，最大化利用已有时间，在不超时的前提下尽可能利用 5 秒的时间，来搜索更多的层数，提高胜率。

可以看出，我们的代码在自己分组的竞技场中表现良好，偶尔会出现与对方僵持不下的状况，推测是因为随机序列导致。总体上超时和报错较少，且发挥稳定，。

4 实习过程总结

4.1 分工与合作

韩泽尧：编写 minmax 算法主体部分

李煦东：添加 αβ 剪枝

杨佳宇：调节时间管理参数，协助撰写报告

刘金禹：调节空格分数列表，撰写报告

王之昱：调节棋子分数，协助撰写报告

我们组很早就开了第一次会，进行分工，编程能力较强的同学主要负责为程序进行结构上的改进（如撰写主题，增加剪枝、时间管理函数等），其余的同学主要负责调整参数，观察对局来提出建议。这里的分工并不是严谨的，大家都对本组代码的最终形成做出了自己的贡献，并且都参与了调参的工作，上面只是列出了较为重要的贡献。

我们组的交流方式为集中的线上小组会议与平时的微信群交流相结合。在几个重要时间点利

用腾讯会议来总结当前成果，交流并制定目标，统筹分配下一步工作。在平时则在微信群里实时交流每天的进展，并在 GitHub 上同步更新。

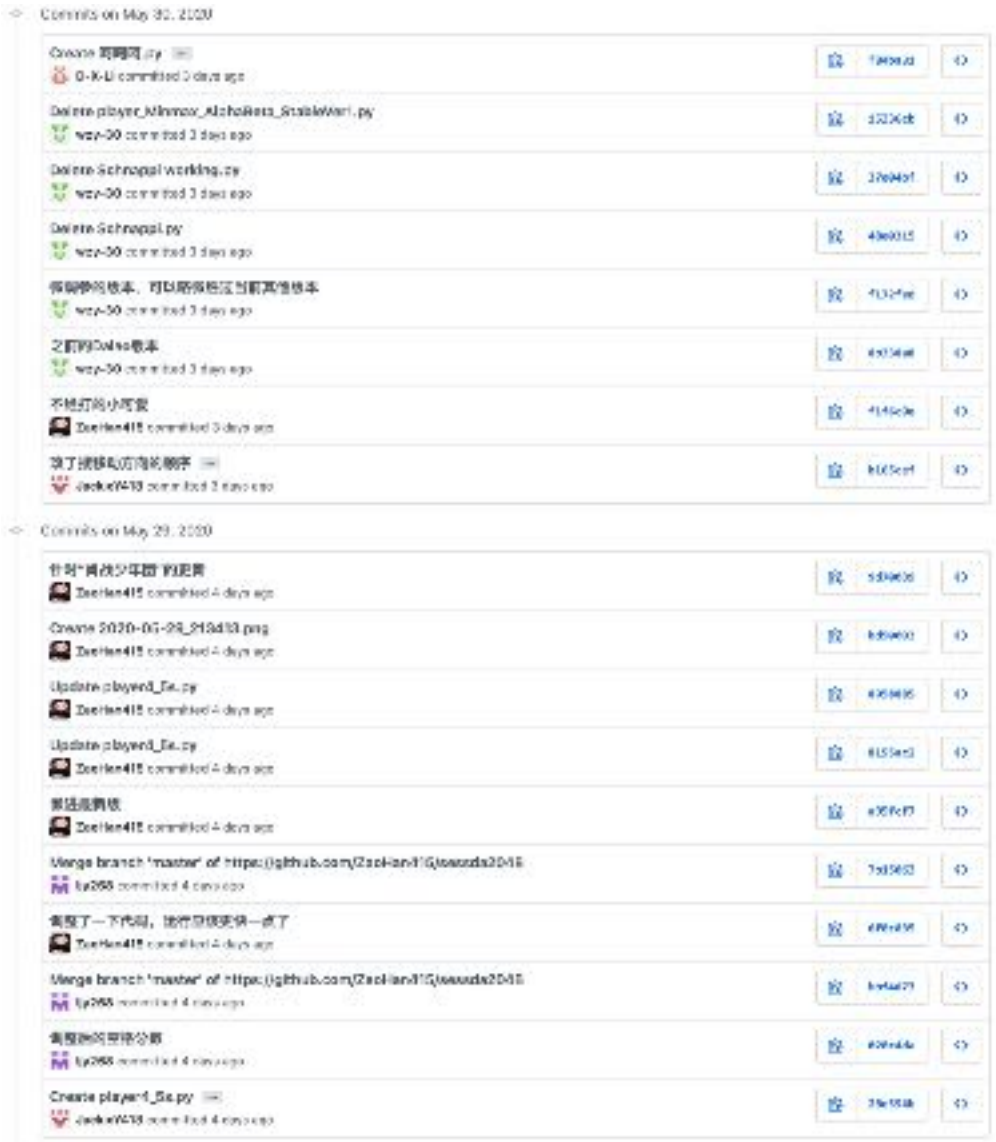


图 4：Github 上 commits history 部分截图

附本组 GitHub 项目网址：https://github.com/ZaoHan415/sessda2048

另附本组第一次和最后一次组会的合影：

图 5:小组会议合影

4.2 经验与教训

本组在这次实习作业中的得意之处在于开工较早，早在五一假期 minmax 算法和 αβ 剪枝就已经基本成型，已经可以完成正常的比赛。这使得我们的时间更为充裕，有更多的时间去测试并优化我们的代码。

另外我们组的整体思路是让每个人都可以尝试自己的思路和想法，没有规定只有某一两个人编写代码，这就使得在一些参数的调整上每个人都参与进来，相比一两个人参与算力提高了不少，在中后期我们在代码结构、启发式规则、估值函数的选取等方面思路也因此非常开阔，对各种情况的利弊也都能够有一个大致的了解。

此外，我们所有人都在密切关注天梯动态。出现哪个组上传了特别强的代码就立马去和对方对战，观察棋局，寻找问题，这使得我们的代码能够一直跟着天梯版本走，在 5 月中旬曾一度进入天体前三，仅次于 RBQ 系列代码和古明地恋系列代码。

本组成立前期由于个人能力不同、分工不够明确、某些前期方案（如蒙特卡洛算法，存储搜索结果等）被否定，导致了在开发出初代代码后有一段效率比较低迷的时期。所幸我们

开工较早，后面通过交流，以及各位组员的包容理解，大家都在组里更加明确自己该干什么，找到了属于自己的工作和位置。或许，如果从一开始就交流充分的话，我们的效率和分工可以更加合理，也能让最后的结果更加令人满意。

使用一般的 minmax 算法时，关键在于调整估值函数，因此有大量的参数需要调整。由于不太会在程序中和网页交互，我们并没有选择使用程序进行代码测试，搜索最优参数的过程一直是人工完成的，导致非常耗费时间精力。由于本地的运行环境与网络上有较大差异，导致必要的调整得出的结果只有上传到网络平台才能得知其最终性能，这导致了大量的时间成本的浪费（等待匹配的过程中）。也许使用程序测试代码可以大幅提高中后期优化代码的效率，这一点是值得我们改进的。

代码方面，在与 F19 第一名的对战中，我们的代码以较大的劣势输掉。通过技术组同学在课堂上的透露，得知对方的估值函数十分复杂。事实上在估值函数的选取上我们虽然有各种不同的尝试，但是依然没有去尝试如平滑性等因素，可能是这样的原因让我们最终不敌对方（展示了 F19 同学的真正实力）。如果我们能够花更多的时间钻研估值函数和调整参数的话，或许可以取得更好的效果。

4.3 建议与设想

这次实习作业在竞技场基础设施、竞赛等方面技术组的同学、老师和助教们都已经做得很好了。

对于竞赛赛制我们有一点小建议，就是希望在比赛时加快进度，同时加入复活赛或者改为双败淘汰赛，使得比赛中的一些随机性因素的影响更小。这个随机性不光指比赛的随机序列，也指在淘汰赛中匹配的随机性。当然这只是一个美好的期待，毕竟一节课的时间很有限，也可以理解在现实条件下的赛制已经很合理了。

另外，此次代码天梯上在后期出现了这样一种现象：上传代码，和别人的代码测试对战，打完就删掉原来的代码的情况。这样固然可以防止自己的代码被对手针对，进行专门优化击败自己的代码，但是也影响了比赛的环境和良好的竞争氛围，使得在后期天梯上很难找到太多其他组的最新版小组代码来进行切磋测试。当然这也是小组的一种策略，但还是感觉这给后期代码的优化产生了一定的阻碍。希望明年老师能提出并改良一下这个问题，让大家都可以公平快乐的享受到这次比赛！

在这里也希望明年的学弟学妹可以享受这次大作业。不必将这看作一种任务或者是负担，几个人一起，想办法把一个游戏玩好，这难道不也是一件很嗨皮的事情吗？虽然在写代码、debug 的过程中会感到有一些枯燥，但是在你拥有了属于自己的 AI 之后，会觉得一切努力都是值得的。

5 致谢

感谢技术组的张宇昊等同学以及各位助教，耐心地解答我们关于规则的疑问，不断地完善竞技场代码，为我们带来了更好的对局体验。

感谢各位排行榜上的大佬们，你们的存在是我们进步的方向和动力。

感谢陈斌老师，您在本学期的课程的教授以及在直播课上对 minmax 算法的讲解，都对本组作品的产生和完善产生了十分重要的促进作用。

感谢这次大作业，让我们五个人有机会成为组员，共同完成这一有趣、艰巨，同时富有

意义的任务。

6 参考文献

minmax 算法：
http://web.cs.ucla.edu/~rosen/161/notes/minimax.html

αβ 剪枝：

https://www.hackerearth.com/blog/developers/minimax-algorithm-alpha-beta-pruning/
http://web.cs.ucla.edu/~rosen/161/notes/alphabeta.html/

数据结构与算法课程实习作业报告

（唐云潇*，陈杜钰，陈尧，张雅清，曾逸杰）

摘要：（如何评价局面，构建尽量深的搜索是我们组算法的两大核心。我们通过多个评价标准使得本方具有单调性的 2048 优势局面，同时鼓励棋子向对方领域移动。在此评估函数下构建了搜索树来预测多轮对弈后的盘面）

关键字：（决策树、minimax 搜索）

1 算法思想.....1
1.1 总体思路.....1
1.2 算法流程图.....1
1.3 算法运行时间复杂度分析.....2
2 程序代码说明.....3
2.1 数据结构说明.....3
2.2 函数说明.....3
2.3 程序限制.....3
3 实验结果.....6
3.1 测试数据.....7
3.2 结果分析.....7
4 实习过程总结.....8
4.1 分工与合作.....8
4.2 经验与教训.....8
4.3 建议与设想.....8
5 致谢.....9
6 参考文献.....9

1 算法思想

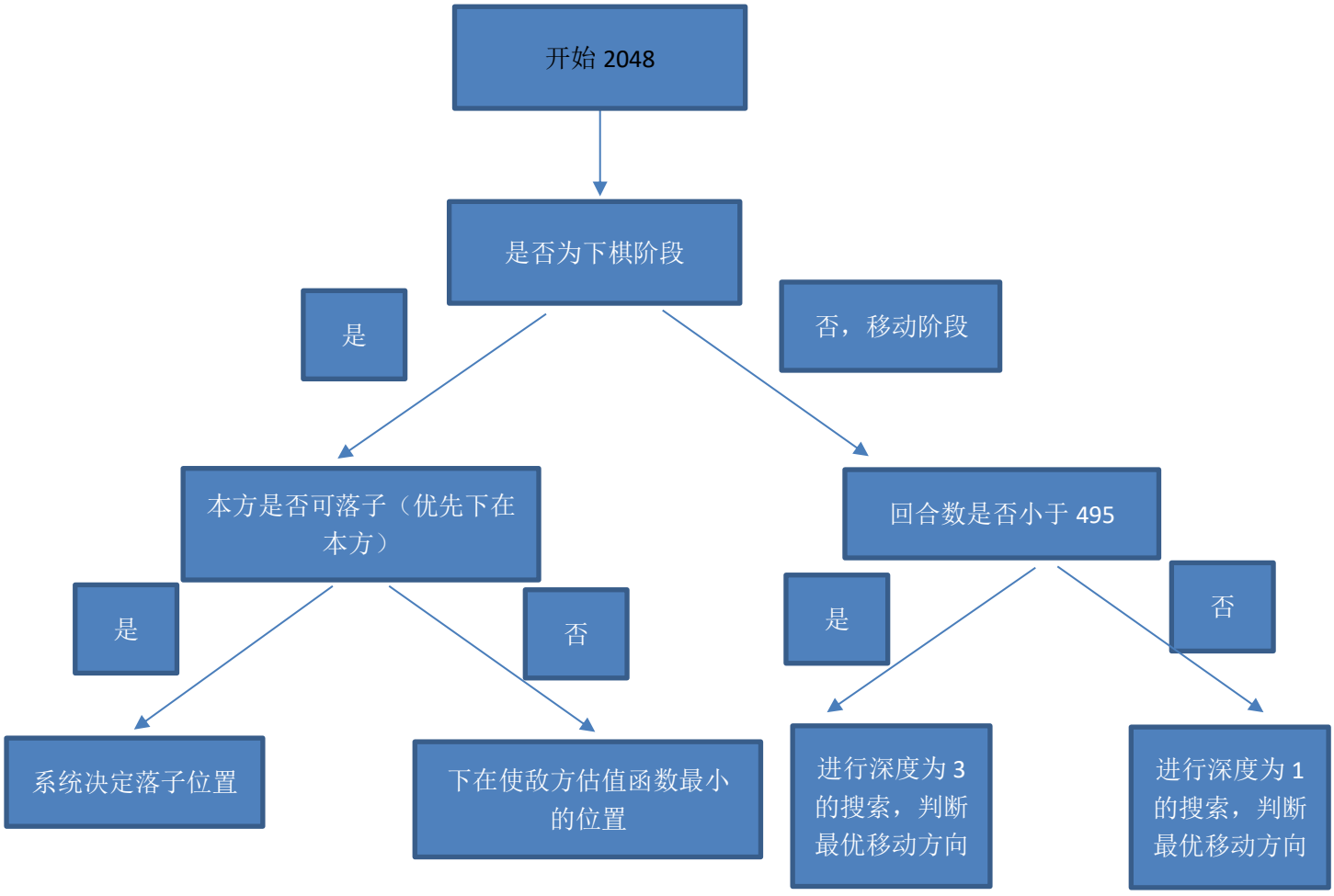
1.1 总体思路

总体思路：在下棋阶段，优先落子本方，落子位置由系统给出，如若本方无法落子，下在使敌方估值函数最低的位置；在移动阶段，在 495 回合内，则构建搜索深度为 3 的搜索树，寻找使三回合后使己方估值函数最大的初始移动方向，若大于 495 回合，则只需进行深度为 1 的搜索。

采用的算法策略：估值函数倾向于将己方最大值棋子放于与对方交界的角落位置、如若可以将尽量吞并边界棋子并避免己方边界棋子被吞并、让整个棋盘的平滑度（定义为整个棋盘棋子值的大小均匀程度，棋子差值越大，平滑度越低）尽可能低、己方棋盘的空地尽可能多，通过不断天梯赛总结出各项指标在估值函数中的权重生成最有利于比赛的估值函数。

主要数据结构与算法：树、列表
算法从最初设想到实现确定的过程：组员在平时玩 2048 游戏时总结出一条经验：将己方最大值棋子放于与对方交界的角落位置，并在天梯赛中发现大部分队伍采取此项作法；另外，由于本次 2048 比赛允许吃子吞并、落子敌方，故将边角棋子尽量躲避被吞并、尽量吞并对方棋子考虑在内；最后，由于比赛是根据己方最大子数值大小（如若打平，再考虑第二大子）作为胜负依据，故希望己方棋子值尽量集中到一起，所以希望己方棋盘的平滑度尽可能低。

1.2 算法流程图



1.3 算法运行时间复杂度分析

我们的算法主要时间花费在如何下棋的函数 output 与如何移动的函数 search 上。而这两部分的函数时间复杂度都主要取决于判断所获分数值的函数 judge。Judge 函数包含许多支函数，其中支函数最大的时间复杂度是 O(n^2)，因此 judge 函数的时间复杂度也是 O(n^2)，我们的算法时间复杂度也大约为 O(n^2)，这也符合我们实际运行的结果。

2 程序代码说明

2.1 数据结构说明

本程序中采取的数据结构主要是简单的线性数据结构—列表和树形数据结构—搜索树。

2.2 函数说明

1.output: 输出己方决策，决定本回合在哪里下棋或往哪个方向移动。若为下棋阶段，则优先考虑在己方下棋，若不能，则在对方使得 judge 函数值最小的位置下棋。

```
    def output(self, currentRound, board, mode):
        #输出决策
        if mode == 'position':#下棋阶段
            myPlace = board.getNext(self.isFirst, currentRound)
            if myPlace != (): return myPlace
            #优先下棋在本方
            opposite = board.getNone(not self.isFirst)
            if opposite:
                #如果需要下棋于对方，寻找估值函数给出最低值的位置
                current = 1000
                for place in opposite:
                    fakeboard = board.copy()
                    fakeboard.add(not self.isFirst, place)
                    score = self.judge(fakeboard, not self.isFirst)
                    if score < current:
                        current = score
                        posi = place
                return posi

        elif mode == 'direction':#移动阶段
            return self.searchDepth(board, currentRound)
```

2.searchDepth: 深度优先搜索，当前回合数小于 495 时搜索深度为 3，当前回合数大于 495 时搜索深度为 1,。其搜索方式与 search 函数是相同的，但调用结束后将返回移动方向决策，并取 alpha 值最高者为最终移动方向。

```
    def searchDepth(self, board, currentRound):
        if currentRound<495:
            depth = 3
        else: depth = 1
        alpha = -10000
        for i in range(4):
            fakeboard = board.copy()
            if fakeboard.move(self.isFirst, i):
                score = self.search(fakeboard, depth -1, 'position', currentRound)
                if score >= alpha:
                    alpha = score
                    result = i
        return result
```

3.search: 构建四叉搜索树，并从底层开始逐层将四个子节点的最大值加到父节点上。为了更准确地模拟，在构建搜索树时我们增添了下棋环节，简单起见，我们考虑在本方下棋。

```
    def search(self, board, depth, mode, currentRound):
        if depth == 0:
            score = self.judge(board, self.isFirst)
            return score

        if mode == 'direction':
            alpha = -10000
            curScore = self.judge(board, self.isFirst)
            for i in [0,1,2,3]:
                fakeboard = board.copy()
                if fakeboard.move(self.isFirst, i):
                    score = curScore + self.search(fakeboard, depth-1, 'position', currentRound)
                    if alpha < score:
                        alpha = score
            return alpha

        if mode == 'position':
            simuround = currentRound + 1 - depth
            myPlace = board.getNext(self.isFirst, simuround)
            if myPlace != ():
                board.add(self.isFirst, myPlace)
            return self.search(board, depth, 'direction', currentRound)
```

4.boardBoundary：返回本轮的棋盘 y 方向边界。

```
    def boardBoundary(self, isFirst):#返回本轮的棋盘y方向边界
        if isFirst:
            return 0, 3
        else:
            return 4, 7
```

5.neighbor：返回某位置给定方向的相邻位置的坐标。

```
    def neighbor(self, direction, position):#返回某位置给定方向的相邻位置坐标
        if direction == 0:
            return position[0]-1, position[1]
        elif direction == 1:
            return position[0]+1, position[1]
        elif direction == 2:
            return position[0], position[1]-1
        elif direction == 3:
            return position[0], position[1]+1
```

6.smoothness：光滑性是指每个方块与其直接相邻方块数值的差，其中差越小越光滑。例如 2 旁边是 4 就比 2 旁边是 128 光滑。一般认为越光滑的格局越好。

```
    def smoothness(self, board, place):#光滑性，整个二维列表的数值平方越大，值越小
        smoothness = 0
        Ystart, Yend = self.boardBoundary(place)
        for y in range(Ystart, Yend+1):
            for x in range(4):
                currentValue = board.getValue((x, y))
                if currentValue != 0:
                    for direction in [1,2]:
                        target = self.neighbor(direction, (x, y))
                        if target[0]<=3 and target[1]>=Ystart:
                            value = board.getValue(target)
                            if value != 0:
                                smoothness -= abs(value - currentValue)
        return smoothness
```

7. monotonicity：单调性指方块从左到右、从上到下均遵从递增或递减。一般来说，越单调的格局越好。本函数中我们就认为棋子的最大值越靠近与对方交界的角落位置越好。

```
    def monotonicity(self, board, place):#单调性，使棋子的最大值靠近与对方交界的角落位置
        score = [0, 0, 0, 0]
        start, end = self.boardBoundary(place)

        for y in range(start, end+1):
            x = 0
            while x < 3:
                currentValue = board.getValue((x, y))
                target = x+1
                targetValue = board.getValue((target, y))
                score[0] -= targetValue - currentValue
                x = target



        for x in range(4):
            y = start
            while y < end:
                currentValue = board.getValue((x, y))
                target = y + 1
                targetValue = board.getValue((x, target))
                if not self.isFirst:
                    score[1] -= currentValue - targetValue
                else:
                    score[1] += targetValue - currentValue
                y = target

        return (score[0],score[1])
```

8. Empty：返回本方棋盘空格个数。一般来说，空格子越少对玩家越不利，因为越容易出现无法移动的情况。所以我们认为空格越多的格局越好。

```
    def Empty(self, board, place):#返回本方棋盘空格个数
        start, end = self.boardBoundary(place)
        count = 0
        for y in range(start, end+1):
            for x in range(4):
                if board.getValue((x, y)) == 0:
                    count += 1
        return count
```

9. attack：返回在对方棋盘上的本方棋子的数值之和（数值为以 2 为底棋子大小的对数）数值越大对己方越有利。

```
    def attack(self, board, place):
        attack = 0
        start, end = self.boardBoundary(not place)
        for y in range(start, end + 1):
            for x in range(4):
                if board.getBelong((x, y)) == place:
                    attack += board.getValue((x,y))
        return attack
```

10.protect：返回在己方棋盘上的对方棋子的数值之和（数值为以 2 为底棋子大小的对数）数值越小对己方越有利。

```
    def protect(self, board, place):
        count = 0
        start, end = self.boardBoundary(place)
        for y in range(start, end + 1):
            for x in range(4):
                if board.getBelong((x, y)) != place:
                    count += board.getValue((x,y))
        return count
```

11.judge：这是己方输出决策中最重要的部分，如何评价当前格局的价值是重中之重。在游戏中除了终局外，中间的格局并无非常明显的价值评价指标，因此需要用一些启发式的指标来评价格局。那些分数高的“好”格局是容易引向胜利的格局，而分低的“坏”格局是容易引向失败的格局。这些指标就是上面定义的函数 6,7,8,9,10 以及当前棋局上最大的棋子。给各个启发式指标的值赋予权重（即这些参数）相加，得到估值。

```
    def judge(self, board, place):#给各函数赋权重相加，得到估值
        smoothWeight = 0.5
        monoWeight = 1.2
        emptyWeight = 3
        maxWeight = 1
        attackWeight = 1
        protectWeight = 2

        empty = self.Empty(board, place)

        attackPoint = self.attack(board, place)*attackWeight \
                      + self.protect(board, place)*protectWeight

        score = self.monotonicity(board, place)*monoWeight \
                + self.smoothness(board, place)*smoothWeight \
                + math.log(empty if empty != 0 else 0.5)*emptyWeight \
                + board.getScore(place)[0]*maxWeight + attackPoint
        return score
```

2.3 程序限制

目前为止的测试和天梯上的实战对局中，我们的程序暂未出错。

3 实验结果

3.1 测试数据

实验环境说明：

- 硬件配置：（CPU/内存）i5-8300H
- 操作系统：（名称/版本）Windows10
- Python 版本：（版本号） python3.8

我们主要通过官方给出的本地对战平台进行算法的测试，保证 AI 代码在一般情况下不出现超时、错误等问题并对于随机的算法有高胜率。随后通过热身赛进行线上测试，初期根据代码表现调整大结构，如评价函数的选取，搜索树的选择（蒙特卡洛树或 minimax 搜索树）；后期则基于丰富的实战经验进行调参。热身赛的结果如下图所示：

名称	类型	战绩
naive	贰零肆捌	122赛1213战- 579胜602负32平
naive2	贰零肆捌	40赛392战- 104胜228负60平
naive3	贰零肆捌	5赛51战- 14胜37负0平

其中 naive 为本次竞赛我们采用的代码，naive3 是我们加强搜索的探索算法，然而没能在正式比赛前完成故放弃。

3.2 结果分析

本算法主体分为搜索和评估两部分，其中重点的是评估部分，我们在函数中对棋子向对方领地移动的情况加以鼓励，从而取得了很好的效果，在测试中大大提高了对于 default 算法的胜率，且在热身赛中时我们的算法天梯排名提升。

但复杂的评估函数消耗了过多的计算资源，使得同样重要的搜索部分深度很大程度上收到限制。虽然最终通过降低搜索深度将算法运行时间拉回了规定值以下，但很大程度上影响了我们的算法的表现，与其他组的算法产生了差距。

4 实习过程总结

4.1 分工与合作

本次大作业由五个来自不同学院的同学组成，于是我们也按照各人的专长分工。物理学院的唐云潇同学和数学学院的陈尧同学负责算法和编程部分与报告撰写，工学院的曾逸杰同学负责天梯的代码试运行与报告撰写，历史学院的陈杜钰同学和光华学院交换生张雅清同学则主要负责报告的撰写。

我们的主要合作交流方式是通过腾讯会议，以及微信上的聊天交流。组会总共进行过两次，分别在 5 月 23 日和 5 月 31 日。下图展示了 5 月 31 日的会议截图。

图 1 会议记录截屏

4.2 经验与教训

本次实习作业中，我们体会到边学习边摸索的乐趣，总体而言，我们的整体算法思路和结构是比较清晰的，但过程中对于评价函数的设计与优化与决策树的实现都十分富有挑战性，完成这项作业之后，同学们体会到了令人难忘的成就感。

我们一些比较好的经验包括：（1）同学们的代码注释写的十分清楚，这十分便于组员之间的交流与理解，提高了我们的沟通效率；（2）整体算法的思路和结构比较清晰；（3）组里科院的同学主动性很强，对团队起到的帮助很大。

我们希望能得到更多提升的部分是：（1）基础设施代码部分，我们没有做 αβ 减值，搜索深度不强，如果深度更大，做出的决策会更好，我们前期在评价函数上耗费的精力比较多，没有意识到搜索深度的重要性，这一点很大程度上影响了我们的发挥；（2）如果有机会，希望可以和别的小组互相交流和切磋一下代码，这样也许可以看出更多的问题和进步空间；（3）虽然由于疫情影响，本学期实习作业只能线上进行交流，但若有机会，线下交流给各组员带来的参与感和积极性会更好；（4）我们小组在代码编写的阶段没能让全员参与，没能让全员深入体验到游戏的乐趣这一点比较遗憾。

4.3 建议与设想

我们认为本次实习作业总体上体验很好，兼具趣味与挑战性。对于组队，如果老师允许随机组队就更好了，这样可以一定程度上避免“大佬抱团”的现象。另外，如果还有时间，也许我们自己也会再减一下评价函数，换成蒙特卡洛树搜索试试看，以优化最后的结果。

最后，我们想对明年选修此课程的学弟学妹们说，此门课程十分有趣，你的收获将会和所花的时间成正比，所以请多多投入。以后这门课程将变成和高等数学一样的基础课了，十分重要。但建议同学们不要在学分比较多的学期选此门课程，因为地空数算客观上比较花时间。

祝学弟学妹们考试顺利，OJ 愉快，数理基础节节高！

5 致谢

感谢陈斌老师提供如此精彩的课程与课业期间的支持，我们全组同学收益良多。
感谢各位助教，及时解答我们的问题并参与有价值的讨论。
感谢组长唐云潇，对本次作业的贡献最大，是算法与代码部分的主力。
感谢陈尧同学在算法和代码部分的贡献。
感谢曾逸杰同学对测试和报告的贡献。
感谢陈杜钰同学和张雅清同学对报告撰写的贡献。

6 参考文献

[1] 2048-AI 程序算法分析, http://blog.codinglabs.org/articles/2048-ai-analysis.html
[2] https://gitee.com/chbpku/sessdsa.2048

N-17 Victor 小组实习作业报告

朴灿国、雷哲轩、姚志睿*、武蒙、董文涛

摘要：对抗性 2048 游戏是一种完全信息的零和博弈，在本文中，我们使用经典的含有 alpha-beta 剪枝的博弈树搜索算法，配合四种启发式评估函数，从棋盘为节点，以操作为边，构建博弈树，并进行深度优先搜索，达到搜索深度后截止搜索，得到叶节点的评估值，并在向根节点使用 minimax 算法回传结果的过程中，进行 alpha-beta 剪枝，以较高的效率得到最优策略。经过估值函数的调整，我们得到了一类进攻性的 AI，根据与 2048Bot_S 的对战结果，我们的 AI 的棋力与 2048Bot_S 相当，并在最终的天梯比赛中获得第九名的成绩。

关键字：博弈树，minimax 法，alpha-beta 剪枝，深度优先搜索

1 算法思想.....1
1.1 总体思路.....1
1.2 算法流程图.....2
1.3 算法运行时间复杂度分析.....3
2 程序代码说明.....3
2.1 数据结构说明.....3
2.2 函数说明.....4
2.3 程序限制.....13
3 实验结果.....13
3.1 测试数据.....13
3.2 结果分析.....14
3.3 经典战局.....17
4 实习过程总结.....18
4.1 分工与合作.....18
4.2 经验与教训.....19
4.3 建议与设想.....20
5 致谢.....20
6 参考文献.....20

1 算法思想

1.1 总体思路

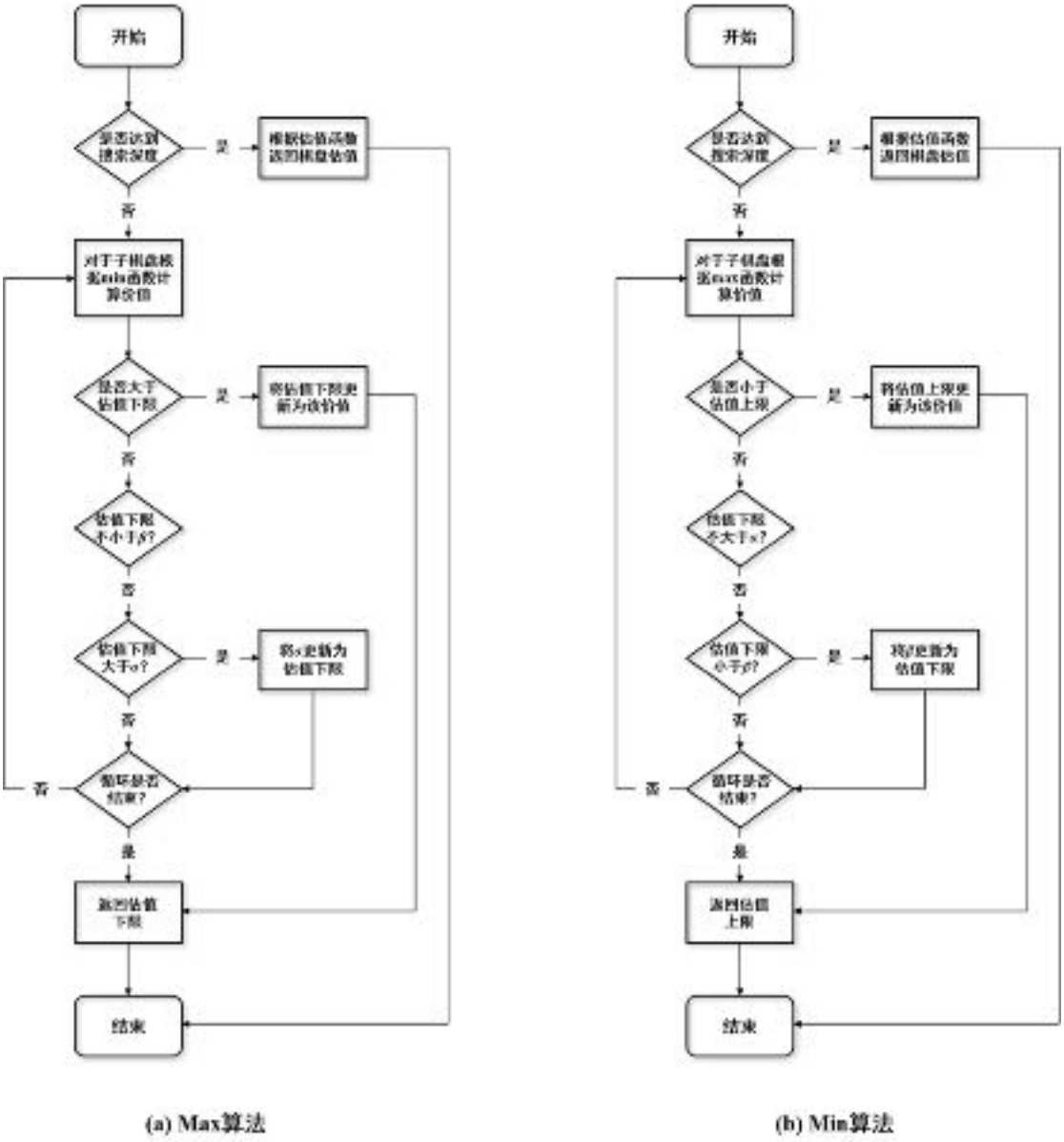
本小组采用的是完全信息零和博弈对战的经典算法，即以棋盘为节点，以操作为边，构建博弈树，并对其进行深度优先搜索，在特定搜索深度上设定截止条件，通过四种估值函数计算其价值，使用 minimax 算法回传评估值，并在 alpha-beta 剪枝的帮助下迅速得出当前局面的最佳操作。在接下来的篇幅中，我们将逐个介绍 minimax 算法，alpha-beta 剪枝以及四

种估值函数。

在实习过程中，我们算法的实现分为了两个阶段，其中数据结构发生了比较大的变化。起初我们在没有查阅足够参考资料的情况下，从对于博弈树与 minimax 算法的理解出发，初步开始了代码的编写。具体地，结合课程内容中树的构建的知识，构造了“博弈树”数据对象与“节点”数据对象，并通过博弈树搜索的算法找出当前棋盘的最佳操作。后来通过查阅一些网络资料[1]与参考书[2]，放弃了之前的方案，采用已经成熟的代码框架，直接在 Player 类中构建 alpha_beta_cutoff_search 方法来实现博弈树的搜索。

1.2 算法流程图

本小组算法的两个最大最小值函数的流程图如图 1 所示。



(a) Max算法　　(b) Min算法

图 1 算法流程图

在程序中输入当前回合数，模式与当前棋盘后，程序会计算当前棋盘可以进行的操作，

将这些操作作用于当前棋盘，以生成子棋盘，并计算子棋盘的 min value，之后会发生 min value 函数与 max value 函数的互相递归调用。当达到预定的搜索深度时，触发基本结束条件，计算评估值，在回传 minimax 值的过程中，进行 alpha-beta 剪枝，最终返回当前的最佳策略。

1.3 算法运行时间复杂度分析

我们的算法运行时间构成可以分为两个主要部分来讨论：博弈树的构建中叶节点价值的计算（估值函数的计算）以及估值的回传与剪枝。

在算法设计早期我们就发现，在生成子节点的时候，如果将所有可能的操作带来的棋盘都加入博弈树中的话，还未达到理想的搜索深度就会超时。通过分析，我们认为在博弈树中，生成了许多在对方棋盘下棋的分支，因此使博弈树的分支因子急剧增大。经过讨论，我们认为在对方棋盘下棋是游戏中后期才需要采取的限制对方大子合并的操作。游戏开局的首要目标是在自己的棋盘下棋来增大棋子的级别。在游戏早期便考虑在对方棋盘下棋不仅费时，而且对于对战不利。因此，我们在用于返回可行操作的函数中添加了限制：即只有当对方棋盘空位少于一定数值（最终设定为 6）时才会返回其可行操作，否则只返回己方操作。这对于节省 AI 思考时间与前期棋子的积累有很大作用，将每一层的节点数限制在最大值 7 以下。

其次，博弈树的搜索深度也明显对于复杂度有很大的影响。我们对于搜索深度设计了一个考虑当前操作（下棋或移动）与可能操作数的函数，用于判断搜索深度。总体思路是下一步可能的操作数越多，搜索层数越低。最终最多的搜索层数定为 4。因此，我们的算法要考虑的棋盘节点是较少的，最多的情况为$7 \times 4 \times 7 \times 4 = 784$个。

另一方面，我们的估值函数包含三个部分，分别是总体棋子数值、平滑性以及棋子分布特征。总体棋子数值的计算与棋子类型（多少种不同数值的棋子）的数目有关，因为要使用循环累加他们的和，但是通常不会太大，计算量是较小的。而平滑性与棋子分布特征的计算量占主要部分，虽然是固定的，但是用到了嵌套循环对棋盘各点进行遍历。我们通过设计 minimax 算法的结束条件，为估值函数的计算预留了 0.5 秒的时间，所以通常情况下不会导致超时。具体关于估值函数的分析会留在代码说明部分进行。

通过以上分析，如果认为最大搜索深度是 d，平均分支因子为 a，那么如果不考虑 alpha-beta 剪枝时，时间复杂度为$O(a^d)$；如果考虑 alpha-beta 剪枝时，由于本程序对于后继状态采用的是随机顺序，所以大概是$O(a^{3d/4})$。

2 程序代码说明

2.1 数据结构说明

在本小组在初期采用的算法中，自定义了 BoardNode 类作为构建博弈树的节点，通过嵌套的循环构建博弈树各层节点之间的关联，然后通过 minimax 函数（递归函数）计算当前棋盘对应的最大估值的子棋盘，从而给出最佳操作。但经过讨论，这种通过嵌套循环的博弈树构建方法灵活度比较低，对应的搜索深度不容易调整，我们希望将搜索深度作为一个参数传入，这样能根据对战情况灵活调整。于是，我们在查阅资料后，将博弈树的构建与 minimax 函数融合在同一个函数中，不再采用自定义的 BoardNode 类进行树的构建。

在最终版本中，我们采用的数据结构为博弈树，是课中所学的基本的树结构的拓展，将棋盘设为节点，可行的操作设为边。其中，我方操作后生成的节点是 min 节点，对方操作后生成的节点是 max 节点。

2.2 函数说明

output 函数如图示 1 所示。其中简单的调用了 alpha_beta_cutoff_search 函数（α-β 剪枝搜索函数），输入参数为当前棋盘(board)，当前回合数(currentRound)，先后手(self.isFirst)，最大回合数(len(self.array))，同时在输入模式(mode)为 position 与 direction（分别表示下棋与移动）时，分别输出一个元组或一个整数，表示下棋位置与移动方向。

```
#输出方法，输入 currentRound（当前局数），board（棋盘），mode（模式），输出下棋的位置或移动的方向
def output(self,currentRound:int,board, mode:str)-> tuple or int:
    if mode == 'position':  # 给出己方下棋的位置
        return self.alpha_beta_cutoff_search(board,currentRound,'position',self.isFirst,len(self.array))
    elif mode == 'direction':  # 给出己方合并的方向
        return self.alpha_beta_cutoff_search(board,currentRound,'direction',self.isFirst,len(self.array))
    else:
        return
```

图示 1　output 函数

α-β 剪枝搜索函数的内容如图示 2。其中包含了四个子函数：

1. cal_next_round()，接受参数为当前回合数(currentRound)，操作模式(mode)，操作者(布尔值 player, 我方操作时为 True)，先后手(isFirst)，返回操作后的回合数（一个整数）。

2 & 3. max_value()与 min_value()（分别对应 max 节点与 min 节点），接受参数为当前棋盘(cuurentBoard)，当前回合数(currentRound)，操作模式(mode)，先后手(isFirst)，父节点估值下限(alpha)，父节点估值上限(beta)，搜索深度(depth)，最大回合数(max_round)，返回当前节点的估值（一个浮点数）。

4. cutoff_test()，接受参数当前棋盘(cuurentBoard)，当前回合数(currentRound)，模式(mode)，操作者(布尔值 player, 我方操作时为 True)，先后手(isFirst)，搜索深度(depth)，返回是否达到搜索深度（一个布尔值）。

在上述函数的基础上，剪枝搜索函数中首先初始化最佳估值为负无穷，父节点估值上限为正无穷（此处当前棋盘的父节点其实不存在，因此初始化为正无穷），最佳操作为 None。然后调用 getActions 函数获取下一步可能的棋盘列表进行迭代，将其作为 min 节点计算价值（因为这些节点对应的下一步操作应由敌方进行），并对最大估值与最佳操作进行更新，最后返回相应的最佳操作。

```
def alpha_beta_cutoff_search(self, currentBoard, currentRound: int, mode: str, isFirst: bool,
                             max_round) -> tuple or int:
    # 当检索深度增加时，currentRound 会增加，本函数用来计算经操作后 currentRound 是否增加
    def cal_next_round(currentRound, mode, player, isFirst):...
    # 计算 currentBoard 的 maxvalue，并更新 alpha，beta，
    def max_value(currentBoard, currentRound, mode, isFirst, alpha, beta, depth, max_round):...
    # 计算 currentBoard 的 minvalue，并更新 alpha，beta，
    def min_value(currentBoard, currentRound, mode, isFirst, alpha, beta, depth, max_round):...
    # 判断是否到达搜索深度
    def cutoff_test(currentBoard, currentRound, mode, player, isFirst, depth):...

    best_score = -np.inf  # import numpy as np
    beta = np.inf
    best_action = None
    for a in Player.getActions(currentBoard, currentRound, mode, True, isFirst):
        v = min_value(Player.result(currentBoard, a, True, isFirst),
                      cal_next_round(currentRound, mode, False, isFirst),
                      Player.next_mode(False, isFirst, mode), isFirst, best_score, beta, 1, max_round)
        if v > best_score:
            best_score = v
            best_action = a
    return best_action
```

图示 2　α-β 剪枝搜索函数

```
def cal_next_round(currentRound,mode,player,isFirst):
    if not isFirst == True: #如果我方是后手，只当我方移动后 currentRound += 1
        if mode == 'direction' and not player:
            new_currentRound = currentRound + 1
            return new_currentRound
        else:
            return currentRound
    if isFirst == True:
        if mode == 'direction' and player:
            new_currentRound = currentRound + 1
            return new_currentRound
        else:
            return currentRound
```

图示 3　cal_next_round()函数

cal_next_round()函数的内容如图示 3。只有两种特定的情况会导致下一步操作后当前回合数增加，即我方作为后手进行移动，以及我方为先手并且对方进行移动。

max_value()函数列于图示 4。前面提到，max_value()函数与 min_value()函数相互调用，构成递归。二者接受的参数当前棋盘、当前回合数、操作模式、先后手、α、β、搜索深度以

及最大回合数，返回相应节点的估值。

```
def max_value(currentBoard, currentRound, mode, isFirst, alpha, beta, depth, max_round):
    # 基本结束条件：超过回合数上限或者达到搜索深度
    if currentRound >= max_round or cutoff_test(currentBoard, currentRound, mode, True, isFirst, depth):
        return Player.value(isFirst, currentBoard)
    value = -np.inf # 初始化价值为负无穷
    for operations in Player.getActions(currentBoard, currentRound, mode, True, isFirst):
        # 通过调用 min_value()函数计算子节点（均为 min 节点）的价值，对价值变量 value 进行更新
        value = max(value, min_value(Player.result(currentBoard, operations, True, isFirst),
                                     cal_next_round(currentRound, mode, False, isFirst),
                                     Player.next_mode(False, isFirst, mode), isFirst, alpha, beta, depth + 1,
                                     max_round)) # 其中调用 min_value()函数时深度增加
        # beta 是父节点的估值上限，在调用此 max_value()函数前调用的 min_value()函数中进行了更新
        if value >= beta:  # beta 剪枝
            return value
        # 对节点的估值下限进行更新，在循环中下一次调用 min_value()时作为父节点的估值下限传入
        alpha = max(alpha, value)
    return value
```

图示 4　max_value()函数

相应的，min_value()函数列于图示 5 中，其结构与 max_value()函数完全一致，只是由于 min 节点的性质，一些判断语句与赋值操作与 max_value()函数中是相反的。我们小组的算法的核心部分即在于由 max_value()与 min_value()这两个函数实现的 minimax 算法，以及其中涉及到的 α-β 剪枝。

```
def min_value(currentBoard,currentRound,mode ,isFirst,alpha, beta, depth,max_round):
    if currentRound >= max_round or cutoff_test(currentBoard,currentRound,mode,False,isFirst,depth) :
        return Player.value(isFirst,currentBoard,currentRound)
    v = np.inf
    for a in Player.getActions(currentBoard,currentRound,mode,False,isFirst):
        v = min(v, max_value(Player.result(currentBoard,a,False,isFirst),
                cal_next_round(currentRound,mode,True,isFirst),
                Player.next_mode(True,isFirst,mode),isFirst,alpha, beta, depth+1,max_round))
        if v <= alpha:
            return v
        beta = min(beta, v)
    return v
```

图示 5　min_value()函数

此外，用于判断是否达成基本结束条件的函数 cutoff_test()列于图示 6。可以看出，我们对于搜索深度的划分考虑到了可能的操作数、操作模式、剩余思考时间这些因素。最后我们对剩余思考时间也进行了判断

此外，在上述函数中还调用了三个函数，分别是 getActions()，result()与 next_mode()，

这三个函数作为 player 类中的静态方法出现。getActions()函数的内容见图示 7。本函数接受的参数为当前棋盘、回合数、操作者、操作模式，返回下一步可能的棋盘组成的列表。其中，按照一回合内的四个阶段进行对应的讨论。 在 1.3 节提到我们对于要考虑的节点数有所限制，这在此 getActions 函数中体现出来，图示 6 中第 9 行与第 27 行的条件分支判断即规定了在对方棋盘下棋的可能位置（即对方棋盘空位）超过 6 的话，则不考虑在可能操作之内。

```
1.  def cutoff_test(currentBoard, currentRound, mode, player, isFirst, depth):
2.      cutoff_depth = 0
3.      _position = len(Player.getActions(currentBoard, currentRound, mode, player, isFirst))
4.      if mode == 'position':
5.          if _position > 8:
6.              cutoff_depth = 2
7.          elif 5 < _position <= 8:
8.              cutoff_depth = 3
9.          elif 3 < _position <= 5:
10.             cutoff_depth = 3
11.         elif 1 < _position <= 3:
12.             cutoff_depth = 4
13.         else:
14.             cutoff_depth = 4
15.     if mode == 'direction':
16.         if _position > 8:
17.             cutoff_depth = 2
18.         elif 5 < _position <= 8:
19.             cutoff_depth = 3
20.         elif 3 < _position <= 5:
21.             cutoff_depth = 3
22.         elif 1 < _position <= 3:
23.             cutoff_depth = 4
24.         else:
25.             cutoff_depth = 4
26.     if currentBoard.getTime(isFirst) > 0.5:
27.         if depth > cutoff_depth:
28.             return True
29.         else:
30.             return False
31.     else:
32.         return True
```

图示 6　cutoff_test()函数

```python
@staticmethod  # 输入当前棋盘，局数，模式，玩家，返回可以进行的操作
def getActions(chessboard, currentRound: int, mode: str, operator: bool, isFirst: bool) -> list:
    action_1, action_2, actions = [], [], []
    if mode == 'position' and operator == True:  # 我方下棋可以进行的操作
        if len(chessboard.getNone(not isFirst)) != 0:
            action_1 = chessboard.getNone(not isFirst)
        if len(chessboard.getNext(isFirst, currentRound)) != 0:
            action_2 = [chessboard.getNext(isFirst, currentRound)]
        if len(action_2) != 0 and len(action_1) > 6:
            return action_2
        elif len(action_2) != 0 and len(action_1) <= 6:
            return action_1 + action_2  # List[Tuple[int,int]]
        elif len(action_2) == 0 and len(action_1) != 0:
            return action_1
        else:
            return [None]
    if mode == 'direction' and operator == True:  # 我方移动可以进行的操作
        for i in range(4):
            if chessboard.copy().move(isFirst, i):
                actions.append(i)
        return actions if len(actions) != 0 else [None]
    if mode == 'position' and operator == False:  # 对方下棋可以进行的操作
        if len(chessboard.getNone(isFirst)) != 0:
            action_1 = chessboard.getNone(isFirst)
        if len(chessboard.getNext(not isFirst, currentRound)) != 0:
            action_2 = [chessboard.getNext(not isFirst, currentRound)]
        if len(action_2) != 0 and len(action_1) > 6:
            return action_2
        elif len(action_2) != 0 and len(action_1) <= 6:
            return action_1 + action_2  # List[Tuple[int,int]]
        elif len(action_2) == 0 and len(action_1) != 0:
            return action_1
        else:
            return [None]
    if mode == 'direction' and operator == False:  # 对方移动可以进行的操作
        actions = []
        for i in range(4):
            if chessboard.copy().move(not isFirst, i):
                actions.append(i)
        return actions if len(actions) != 0 else [None]
```

图示 7　getActions()函数

result()函数列于图示 8 中。result()函数接受的参数为当前棋盘，当前操作，操作者以及先后手，返回该操作得到的新棋盘。其中调用了棋盘 Board 类的函数.add()。

```python
@staticmethod
def result(currentBoard, action, player, isFirst):
    _currentBoard = currentBoard.copy()
    if type(action) == tuple:
        if player == True:  # 如果我方下子
            _currentBoard.add(isFirst, action)
            return _currentBoard
        else:
            _currentBoard.add(not isFirst, action)
            return _currentBoard
    if type(action) == int:
        if player == True:  # 如果我方移动
            _currentBoard.move(isFirst, action)
            return _currentBoard
        else:
            _currentBoard.move(not isFirst, action)
            return _currentBoard
    if action is None:
        return _currentBoard
```

图示 8　result()函数

此外，next_mode()函数列于图示 9，其中中包含一些条件分支，分情况讨论了在特定先后手与当前操作模式的情况下下一步的操作模式，接受参数为操作者、先后手与当前操作模式，返回下一步的操作模式(字符串，position 或者 direction)。

此外，另一个重要的部分是棋盘的估值函数，其内容如图示 10。

其中子函数有四个，分别为 total_score()、smooth_score()、average_score()、chessboard_score()以及 empty_score()，他们列于图示 11 中，其作用在代码的注释中有说明，其设计思路会在第三部分结果分析中进行阐述。另外还有一个函数 matrix()，如图示 12。此函数用于生成一个由字典表示的棋盘，key 为表示位置的元组，value 为棋子数值。

```
@staticmethod
def next_mode(player, isFirst, mode):
    # 计算下一个 mode
    # 输入当前 mode，当前操作者是先后手，返回下一个 mode
    if player == True:  # 当前操作者是我方
        if isFirst and mode == 'position':
            return 'direction'
        if isFirst and mode == 'direction':
            return 'position'
        if not isFirst and mode == 'position':
            return 'position'
        if not isFirst and mode == 'direction':
            return 'direction'
    if player != True:  # 当前操作者是对方
        if not isFirst and mode == 'position':  # 当前操作者是先手，上一个 mode 是'position'
            return 'direction'
        if not isFirst and mode == 'direction':
            return 'position'
        if isFirst and mode == 'position':
            return 'position'
        if isFirst and mode == 'direction':
            return 'direction'
```

图示 9　next_mode()函数

```
# 价值函数，返回棋盘的价值
@staticmethod
def value(isFirst, chessboard):
    matrix = Player.matrix(isFirst, chessboard)
    smooth_weight = 0.05
    average_weight = 4
    chessboard_weight = 2 ** 10
    empty_weight = 2 ** 6
    return Player.total_score(isFirst, chessboard) - smooth_weight * Player.smooth_score(matrix) + \
           average_weight * Player.average_score(isFirst, chessboard) + \
           chessboard_weight * Player.chessboard_score(isFirst, chessboard) + \
           empty_weight * Player.empty(isFirst, chessboard)
```

图示 10　value()函数

```
1.  # 计算棋盘上棋子的数值之和，返回值为我方棋子数值之和与敌方棋子数值之和的差
2.  @staticmethod
3.  def total_score(isFirst, chessboard):
4.      self_score = 0
5.      selfList = chessboard.getScore(isFirst)
6.      for chess in selfList:
7.          self_score += 2 ** chess
8.      oppo_score = 0
9.      oppoList = chessboard.getScore(not isFirst)
10.     for chess in oppoList:
11.         oppo_score += 2 ** chess
12.     return self_score - oppo_score
13.
14. # 衡量棋子分布平滑性的函数，原则是相邻棋子的数值相差太大对于合并没有好处
15. @staticmethod
16. def smooth_score(matrix):
17.     score = 0
18.     oppo_score = 0
19.     for x in range(4):
20.         for y in range(8):
21.             if matrix[(x, y)] > 0:
22.                 if 0 <= x + 1 <= 3:
23.                     score += abs(abs(matrix[(x, y)]) - abs(matrix[(x + 1, y)]))
24.                 if 0 <= x - 1 <= 3:
25.                     score += abs(abs(matrix[(x, y)]) - abs(matrix[(x - 1, y)]))
26.                 if 0 <= y + 1 <= 7:
27.                     score += abs(abs(matrix[(x, y)]) - abs(matrix[(x, y + 1)]))
28.                 if 0 <= y - 1 <= 7:
29.                     score += abs(abs(matrix[(x, y)]) - abs(matrix[(x, y - 1)]))
30.             elif matrix[(x, y)] < 0:
31.                 if 0 <= x + 1 <= 3:
32.                     oppo_score += abs(abs(matrix[(x, y)]) - abs(matrix[(x + 1, y)]))
33.                 if 0 <= x - 1 <= 3:
34.                     oppo_score += abs(abs(matrix[(x, y)]) - abs(matrix[(x - 1, y)]))
35.                 if 0 <= y + 1 <= 7:
36.                     oppo_score += abs(abs(matrix[(x, y)]) - abs(matrix[(x, y + 1)]))
37.                 if 0 <= y - 1 <= 7:
38.                     oppo_score += abs(abs(matrix[(x, y)]) - abs(matrix[(x, y - 1)]))
39.     return ((score - oppo_score) / 4)
```

图示 11-1　估值函数的子函数

```
# 衡量棋子归属的函数，属于我方棋子越多，分数越高，而属于对方的棋子会使分数降低
@staticmethod
def chessboard_score(isFirst, chessboard):
    self_score = 0
    for x in range(0, 4):  # row
        for y in range(0, 8):  # column
            belong = chessboard.getBelong((x, y))  # Tuple[row,column]
            if belong == isFirst:  # 棋子属于我方则加分，否则降分
                self_score += 1
            else:
                self_score -= 1
    return self_score

# 与棋子总数值函数类似，只是将分数变为棋子数值的平方和开根号的差，将估值函数多样化
@staticmethod
def average_score(isFirst, chessboard):
    self_score = 0
    selfList = chessboard.getScore(isFirst)
    for chess in selfList:
        self_score += (2 ** chess) ** 2
    oppo_score = 0
    oppoList = chessboard.getScore(not isFirst)
    for chess in oppoList:
        oppo_score += (2 ** chess) ** 2
    return np.sqrt(self_score) - np.sqrt(oppo_score)

# 计算我方棋盘空位数目的函数，原则是我方空位多说明棋子的数值较大，并且表明未被锁死（即己方棋盘无空位，只能在对方棋盘下棋，损失大）
@staticmethod
def empty(isFirst, chessboard):
    return len(chessboard.getNone(isFirst))
```

图示 11-2　估值函数的子函数

```
# 建立一个表示棋盘棋子分布的字典，key 为表示位置的元组，value 为表示棋子数值的整数
# 传入的 chessboard 参数为棋盘类
@staticmethod
def matrix(isFirst, chessboard):
    matrix = {}
    for x in range(0, 4):  # row
        for y in range(0, 8):  # column
            belong = chessboard.getBelong((x, y))  # Tuple[row,column]
            value = chessboard.getValue((x, y))
            if belong == isFirst:
                matrix[(x, y)] = 2 ** value
            else:
                matrix[(x, y)] = -(2 ** value)
    return matrix
```

图示 12　matrix()函数

2.3 程序限制

本程序在设计中考虑了超时的问题，所以在cutoff_test()函数中设立了剩余时间的阈值，当小于 0.5 s 时，会停止搜索，直接返回估值。同时本程序通过多次调试，已经解决了几乎所有 bug，故本程序的鲁棒性较好，暂未发现问题。

3 实验结果

3.1 测试数据

实验环境说明：

- 硬件配置：　Intel Core i7-6700HQ/16GB 2133MHz LPDDR3
- 操作系统：　mac OS Catalina 10.15.3
- Python 版本：　3.8

在完成初版 AI 时使用的是随机数评估函数，对棋盘随机赋值，来测试 alpha-beta 剪枝搜索的鲁棒性。经过多次的调试，排除了大部分 bug 之后，再将随机数评估函数替换为正式评估函数 value，调试各个评估函数的权重。

$$value = a * total\ score + b * average\ score + c * smooth\ score + d * empty\ score$$

根据初步试验结果，当 a = 1，b = 0，c = 0，d = 0 时，AI 就可以以 100%的胜率战胜本地调试工具中的随机数 AI，说明 total score 在这四个评估函数中是最重要的。

接下来以 a = 1，b = 0，c = 0，d = 0 版本的 AI 为对手，在本地调试工具中逐步改变 b，c，d 的数值，改变后便与原 AI 对战 100 局计算胜率，以确保权重的改变会带来胜率的提高。理论上，要求得最佳参数，需要使用遗传算法，在胜率相关的一个三元函数中找到全局最大值点的，但是由于时间与水平有限，只能通过手动调参进行测试，参数的变化与 AI 的策略

之间的关系会在下一部分进行分析。

根据接近接近 100 次的调整参数，最终得到了天梯版本的估值函数权重。热身赛中，最好成绩在第七名，排名时常在 7～9 之间浮动（图 2）。与 2048Bot_S 对战的胜率为 50%（图 3），但是对于 SS 和 SSS AI，胜率很低，如果以此为参考标准，可以认为是 S 级 AI。

进入	#7	N19 Victor组	41.3% in 2920	1394.2
进入	#8	N19 8848组	52.5% in 3160	1381.61
进入	#9	N19 Romeo组	48.5% in 4330	1357.14
进入	#7	N19 Romeo组	48.3% in 4190	1339.96
进入	#8	N19 8848组	52.2% in 3070	1337.03
进入	#9	N19 Victor组	41.1% in 2840	1333.96

图 2　N19 Victor 组 AI 排名

图 3　N19 Victor 组 AI 与 2048Bot_S 对战结果

3.2 结果分析

在测试中，我们查阅了一些成熟的 2048AI[1]，初步筛选出了几种启发式评估函数。

首先考虑的就是棋子分布启发式评估函数。根据传统 2048 游戏策略，级别高的棋子落在角落，而且剩余棋子最好成单调性排列在两条边上，如下图（图 4）所示：

图 4　有利的棋子分布

所以我们计划使用一个矩阵（图 5）作用于我方的棋盘来使这种分布拥有更高的分数：

2^6	2^5	2^4	2^3
2^5	2^4	2^3	2^2
2^4	2^3	2^2	2^1
2^3	2^2	2^1	2^0

图 5　评估矩阵（一种情况）

$$\text{pattern score} = \sum_{(x,y)\in self} \text{pattern}(x,y) \times \text{value}(x,y)$$

代码列于图示 12。但是测试结果证明，这种评估函数在对战型 2048 游戏中过于死板，根据我们的观察，在对战型 2048 中，并不存在级别大的子落于角落更有利这个启发式规则。所以我们在后来抛弃了这个启发式规则，从中提炼出了一个更普适的启发式规则：平滑性启发式规则。

平滑性启发式规则考虑了每个我方棋子的数值与其周围棋子的数值之差的绝对值：

$$smooth\ score = \frac{1}{4} \sum_{(x,y)\in self} (|value(x,y) - value(x-1,y)| + |value(x,y) - value(x,y-1)| + |value(x,y) - value(x+1,y)| + |value(x,y) - value(x,y+1)|)$$

当评估函数中只考虑我方棋子总价值与平滑性时，就可以达到较高的胜率：

$$value(chessboard) = total\ score + 0.05\ smooth\ score$$

对战中我方的棋子分布如下（图 6）：

图 6　只应用总价值与平滑性后的棋子分布（我方为后手）

```
@staticmethod
def pattern_score(isFirst,matrix):
    pattern_left_1 = {(0,0):2**8,(0,1):2**7,(0,2):2**6,(0,3):2**5,(1,0):2**7,(1,1):2**6,(1,2):2**5,(1,3):2**4,(2,0):2**6,(2,1):2**5,(2,2):2**4,(2,3):2**3,(3,0):2**5,(3,1):2**4,(3,2):2**3,(3,3):2**2}
    pattern_left_2 = {(0,0):2**5,(0,1):2**4,(0,2):2**3,(0,3):2**2,(1,0):2**6,(1,1):2**5,(1,2):2**4,(1,3):2**3,(2,0):2**7,(2,1):2**6,(2,2):2**5,(2,3):2**4,(3,0):2**8,(3,1):2**7,(3,2):2**6,(3,3):2**5}
    pattern_right_1 ={(0,4):2**2,(0,5):2**3,(0,6):2**4,(0,7):2**5,(1,4):2**3,(1,5):2**4,(1,6):2**5,(1,7):2**6,(2,4):2**4,(2,5):2**5,(2,6):2**6,(2,7):2**7,(3,4):2**5,(3,5):2**6,(3,6):2**7,(3,7):2**8}
    pattern_right_2 ={(0,4):2**5,(0,5):2**6,(0,6):2**7,(0,7):2**8,(1,4):2**4,(1,5):2**5,(1,6):2**6,(1,7):2**7,(2,4):2**3,(2,5):2**4,(2,6):2**5,(2,7):2**6,(3,4):2**2,(3,5):2**3,(3,6):2**4,(3,7):2**5}
    if isFirst:
        self_value_1 = 0
        self_value_2 = 0
        for x in range(0,4):
            for y in range(0,4):
                self_value_1 += matrix[(x,y)]*pattern_left_1[(x,y)]
                self_value_2 += matrix[(x,y)]*pattern_left_2[(x,y)]
        self_value = max(self_value_1,self_value_2)
        return self_value
    else:
        self_value_1 = 0
        self_value_2 = 0
        for x in range(0,4):
            for y in range(4,8):
                self_value_1 += matrix[(x,y)]*pattern_right_1[(x,y)]
                self_value_2 += matrix[(x,y)]*pattern_right_2[(x,y)]
        self_value = max(self_value_1,self_value_2)
        return (self_value/4)
```

图示 13　pattern_score()函数

可以发现，这种组合会使棋子的分布更加平滑，从而使合并变得更加容易。同时我们还发现这种组合会产生一种在大级别的棋子周围的单调性分布，与 pattern 的作用类似，但是更加灵活。

我们又考虑了平均值启发式评估函数，当棋子发生合并时，其总价值未发生变化，但是平均值增加，所以如果考虑我方棋子的平均值，可以来提高合并效率，测试证明加入平均值评估函数可以提升胜率。

我们还考虑了棋盘格子启发式评估函数，因为根据对战的观察，当我方棋子占据对方棋盘时，大部分情况都是有利的，所以考虑这个评估函数，可以增加我方 AI 的进攻性。对战实验证明，加入启发式评估函数可以提高胜率。

最后，使我方 AI 进攻性特别强的评估函数，是棋盘空格启发式评估函数。根据游戏经

验，我方的空格数越少，则越容易被卡住，即塞爆，无法合并也无法在我方地盘下子，处于很大的劣势。考虑了空格评估函数后，我方 AI 会在对战中期，一直在对方棋盘下棋，同时也在占据对方棋盘格子，从而使对方棋盘可下的格子越来越少，最终被塞爆。使用了这种战略后，一个最有代表性的标志就是对战的得分曲线，可以发现我方的分数存在先高后低的规律（图 7）。

图 7　对战得分的分差曲线（我方为红色）

根据我们的分析，棋盘空格启发式评估函数使我们的 AI 在对战某些 AI 时，胜率提高很多。其中一个重要因素是，对方的 AI 未考虑棋盘空格数。此时，我方的优势只有在棋局中后段才开始体现出来，而对方的检索深度又无法达到如此的深度，所以会短视地选择在自己的棋盘下棋，从而被塞爆。

选择了这些评估函数后，我们的搜索深度可以达到 3～4 层，运行时间在预期内，平均运行时间约为 4.5 s。算法设计中，alpha-beta 剪枝还有提升的空间，所以其实还可以进一步优化。

3.3 经典战局

在天梯赛中，一个比较明显的特点就是，不同组的 AI 会互相压制。此处仅举例与 8848 组的对局，来体现我们组 AI 的进攻性。

与 8848 组的经典对局：我方为后手

在第 67 回合，8848 的棋盘已经布满了棋子，而我方只有 5 个棋子，同时还有两个在对方的棋盘上。此时观察估计得分曲线可以发现，8848 的得分会在一段时间内持续升高。

在第 183 回合，此时 8848 的棋盘已经被塞爆，无法进行合并和下子，即被塞爆。同时观察得分曲线可以发现，我方得分从此时开始迅速上升，而对方得分一直保持不变。

在第 393 回合，我方已经合成了 2048，而对方依然保持 183 回合的状态，未发生改变。之后我方的会继续合成 4086，直到获得胜利。

这个对局分析表明，我方 AI 的进攻性非常强，对于无法处理己方被塞爆的情况的 AI，我方 AI 的胜率非常高。

4 实习过程总结

4.1 分工与合作

游戏策略、参考书与资料查阅由武蒙、董文涛完成，代码部分中博弈树构建与搜索由朴灿国与雷哲轩完成，估值函数部分由朴灿国与姚志睿完成；实验报告由朴灿国与雷哲轩完成。本小组的讨论主要通过微信群与腾讯会议进行，代码的合作通过 github 仓库实现

(https://github.com/Yao-and-Lei/teamwork2048).微信群中部分讨论的截图如图 8.

图 8 部分微信讨论截图

Github 仓库部分提交记录如图 9。

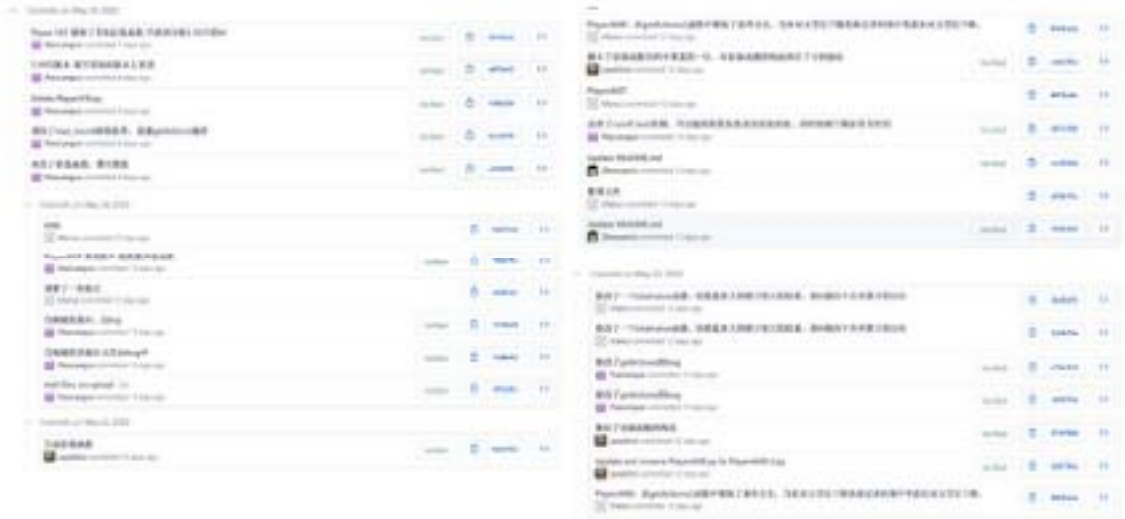


图 9 仓库提交部分记录截图

4.2 经验与教训

在编写程序的过程中，遇到了很多困难。首先就是寻找博弈树的参考资料，组内实现的第一个版本其实是按照课内关于二叉树的实现进行的，后来改为参考《人工智能：一种现代的方法》[2]内关于对抗搜索的内容，根据书中提供的伪代码，实现了博弈树构建的大体框架。我认为，在进行较为大型程序设计的时候，最忌讳的是从 0 开始，使用有效的参考资料，依照已经成型的代码框架，去进行填充和修改，是非常重要的一种技能。

在博弈树的构建中，我们考虑了搜索深度与局面复杂度之间的关系。当可行操作的数量越少时，搜索深度越大，这在一定程度上，增加了我们的 AI 对于残局的处理能力。但是我

们的剪枝过程依然可以通过启发式的一些规则，对搜索顺序进行排序，从而优化剪枝效率。同时，我们也可以用字典储存下已经探索过的局面，从而避免遇到通过不同顺序的操作得到同一种局面而带来的重复计算。

在 debug 阶段，编写者的耐心得到了极大的考验，甚至不得已使用了小黄鸭 debug 法，一行一行代码对着空气讲，才找到许多细节上的 bug。这里也要感谢技术组，开发了本地调试工具，为编程人员验证代码鲁棒性提供了效率非常高的工具。

最后的困难在于进行估值函数的参数调整。最可靠的方式是使用遗传算法或者其他机器学习等方法自动进行参数的优化，但是由于能力与时间有限，只能手动调参。对于结果的可靠性，很大程度上依赖感觉和比较基本的假设。所以在这方面，还有待提高。

4.3 建议与设想

在竞赛得分方面，建议按照天梯名次进行梯度给分，将出线与未出线设为一个门槛，有点太突兀了。

本次的实习作业还有很大提升的空间。接下来可以通过使用遗传算法，对评估函数的权重进行进一步优化。还可以进一步改进博弈树的构建效率，甚至可以考虑学习使用蒙特卡洛法来提升搜索深度。

5 致谢

感谢技术组同学与助教为大作业费心开发对战游戏；感谢陈斌老师在大作业过程中的指导与帮助；感谢微信群内同学们的讨论给我们带来的思路上的启发；感谢小组内每一位同学的努力与付出。

6 参考文献

[1]: https://github.com/comp3211finalprojectgroup/204 8_ai
[2]: Stuart J. Russell, Peter Norvig: 《人工智能：一种现代的方法》，清华大学出版社，第 137 页

数据结构与算法课程实习作业报告

叶琳*，王霆浩，王璠璘，田佳，田缘

摘要：本算法分为合并和下棋两个部分。合并部分采用搜索树和蒙特卡洛算法结合的方法，下棋部分采用完全的启发式算法。调整关键参数和尝试多种搜索、模拟的方案，通过测试运行，最终选出了合适的参数和模拟方案。本算法平均运行时间不超过四秒，在天梯小组赛中获得 N19 组第二名。

关键字：蒙特卡洛算法 四叉搜索树 启发式算法

1 算法思想.....2
1.1 总体思路.....2
1.2 算法流程图.....3
1.3 算法运行时间复杂度分析.....4
2 程序代码说明.....5
2.1 数据结构说明.....5
2.2 函数说明.....7
2.2.1 通用函数接口.....7
2.2.2 合并函数接口.....10
2.2.3 下棋函数接口.....13
2.3 程序限制.....19
3 实验结果.....19
3.1 测试数据.....19
3.1.1 策略测试.....20
3.1.2 参数调整.....22
3.2 结果分析.....24
3.2.1 策略的优劣比较.....24
3.2.2 对部分参数所参与的策略的分析.....25
3.3 经典战局.....26
4 实习过程总结.....26
4.1 分工与合作.....26
4.2 经验与教训.....29
4.2.1 代码版本不统一.....29
4.2.2 编程工作于调参工作协调不够.....29
4.2.3 程序内部运行过程把控不足.....30
4.3 建议与设想.....30
4.3.1 对实习作业的建议.....30
4.3.2 实习作业的后续工作.....30
5 致谢.....37
6 参考文献.....37

1 算法思想

1.1 总体思路

本算法可粗略分为合并和下棋两个部分。

合并部分：以吃棋得分优先，在能吃棋的情况下构建吃棋搜索树（后简称“吃棋树”），返回一个吃棋有利的方向；在不能吃棋的情况下采用蒙特卡洛模拟搜索算法，返回一个合并有利的方向。

下棋部分：完全的启发式规则，根据经验来决定下棋的位置。

算法最初是受到单人 2048 游戏的启发，考虑到构造完全搜索树对时间的限制以及剪枝的复杂性，最后决定采用时间复杂度较小蒙特卡洛算法。轮到己方移动时，对于己方合并的所有可能方向进行“模拟”：假设在己方向某方向移动后，游戏继续进行，通过随机的方式模拟之后双方的下棋和合并。在一定回合数之后给棋盘“打分”，即判断棋盘上己方的有利程度。蒙特卡洛算法精华的“随机平均”思想在于这样的“模拟”要多次进行，可以在很大的程度上减少单次的模拟中的随机成分。通过多次的模拟，得到的平均分作为该方向的分数。实际移动时便从可行方向中选取分数最大的方向。

只有蒙特卡洛模拟的算法在最初的测试中遇到了问题：双人 2048 游戏是不仅仅是对合并有所要求，双方间的交战也是一个重点。在测试中总能发现对方会吃掉我方的棋子，这样的情况一旦多次发生，我方棋子的总分就会变低，局势会向对我方不利的方向发展。从这个角度看，“吃与被吃”应当是优先考虑的东西。所以在合并的时候优先考虑双方“互吃的情形”。于是 “吃棋树”应运而生——只关注双方吃与被吃的树结构。这个结构只关注着互相在合并模式中是否会有互相吃掉对方的棋的现象发生，一旦某一回合双方都没有互相吃棋就会停止延伸，再对每个节点进行评分，根节点的分数反映了能否在吃棋中获利。由于关注点很集中，这棵树往往比较小。实际操作时就将“吃棋树”和“模拟”结合。如果判断能在“吃与被吃”中得到更好的结果，那么优先“吃与被吃”；如果双方在此回合内不会发生“吃与被吃”，那么就采用模拟的方法找出对我方的大数合并最佳的方向。

考虑到如果合并和下棋同时使用搜索算法，可能无法做好时间的分配。而在实际的对战中，下棋这个步骤其实在多数情况下并不需要多步模拟，可以直接根据当前的盘面做出合适甚至最佳的判断。因此我们在下棋回合的算法是完全的启发式规则，无搜索算法。

总的来说下棋有两种选择，一种是给己方增加子力，另一种是在对方棋盘下棋来干扰对方，根据不同的局势来选择不同的方法。下棋时还有一些细节情况。例如，有些数在双方的交界，面临被对方的另一个同级的棋子吃的危险时，我方在下棋时有机会将棋子下在二者中间，这种保护策略避免了更大的损失；如果己方的棋子下在己方的领地中之后马上就会被对方吃掉，那么不如主动将棋子填在对方的领地内，等等。同样都是对方会增加子力，但是主动将棋子下在对方可以防止对方侵占我方的领地。其余的大部分情况，我方都会将棋子下在我方的领地中，用较高效的合并策略较快合成大子。

以上提到的吃棋、合并以及下棋部分的算法，将会在函数部分具体阐述实现过程。

1.2 算法流程图

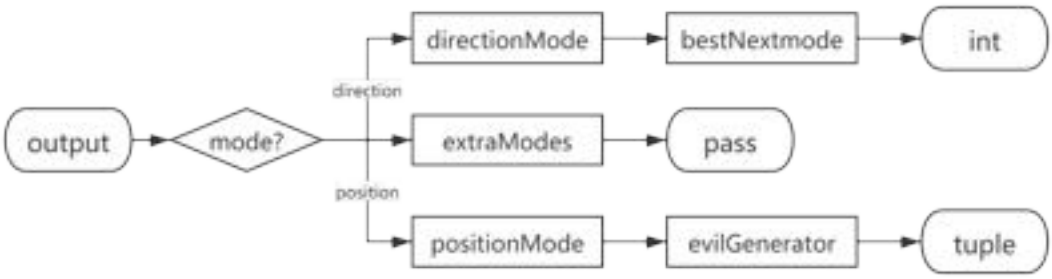


图 1 算法总流程

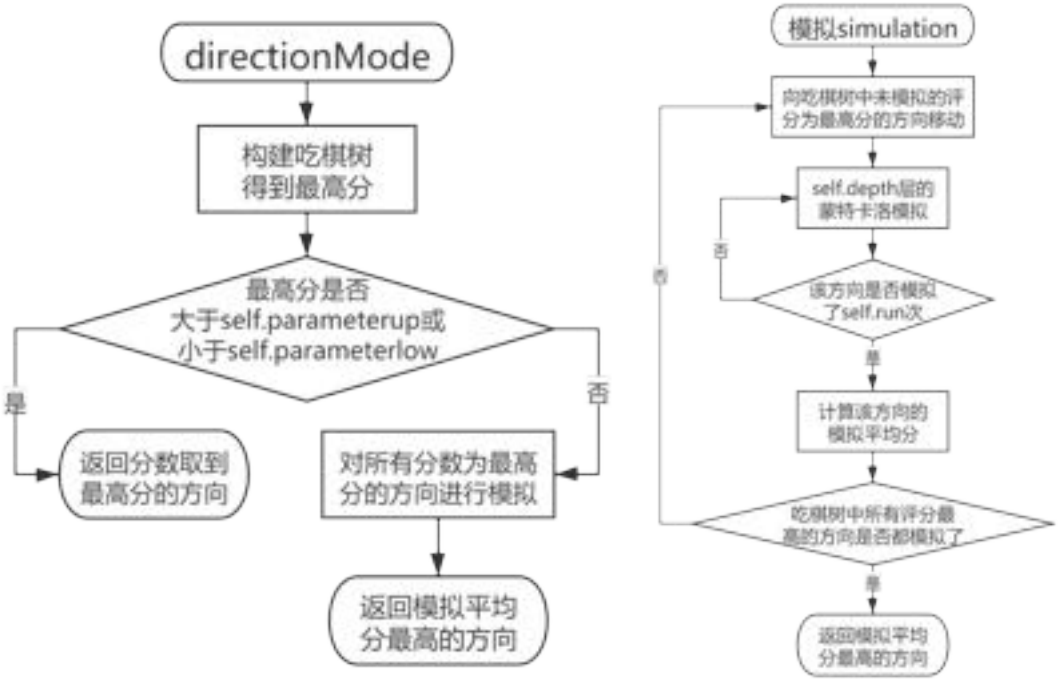


图 2 directionMode 和 simulation 流程图

图 2 左图是合并模式的算法流程图。首先构造一棵“吃棋树”（在第二部分的数据结构详细说明构造流程），并得到各方向移动的最高评分。如果最高分大于 self.parameterup 或小于 self.parameterlow（两个待调整的参数），则返回取到最高分的方向；否则对所有分数取到最高分的方向进行蒙特卡洛模拟，并返回模拟中平均分最高的方向。

图 2 右图是蒙特卡洛模拟的流程图。对于进入蒙特卡洛模拟的某个方向，棋盘先向该方向移动，然后随机模拟游戏继续进行 self.depth 回合，整个模拟重复 self.run 次。模拟全部完成后，返回模拟中平均分最高的方向。

由于下棋部分是启发式规则，执行流程就是判断当前局势是否满足特定策略。通过判断来返回特定位置。由于流程较为简单，因此不再作图阐述。

1.3 算法运行时间复杂度分析

下棋算法复杂度估计：下棋中仅有一次 copy 操作，没有任何的递归和循环，因此整体是线性结构，时间复杂度为 O(1)，并且占用时间很少。

吃棋树算法复杂度估计：在大部分情况，吃棋树都不会超过 5 层，在增加了剪枝策略之后，会减少很多无用的方向。真正的吃棋树相比完全四叉树会稀疏很多，可以估计吃棋树的最大量级是 O(4^depth)。在实际运行的时候，由于深度较浅，一些方向不能移动，节点的个数经常在 20 个以内。（见 2.1 中的例子，吃棋树经常会比较稀疏）

蒙特卡洛算法复杂度估计：采用 5*3 的蒙特卡洛模拟时，主要时间花在 copy 和 move 操作上。蒙特卡洛的时间一般会比较多。在棋盘开阔的时候，可移动的方向较多，更多的方向会进行模拟。棋盘不方便移动的时候，需要多尝试移动几次才能得到合理的模拟方向，这时也会额外增加 copy 和 move 的次数。如果按照模拟 3 个方向估计，copy 和 move 要执行的总次数会在 50 以上。总体来讲，模拟会比吃棋树更占时间，量级是 O(depth*run)，其中 depth 是模拟深度，run 是模拟次数。

图 3 是下棋和合并的用时对比，可以看出下棋的时间几乎可以忽略不计。图 4 是吃棋树和蒙特卡洛用时对比。吃棋树的时间普遍要少于模拟的时间。

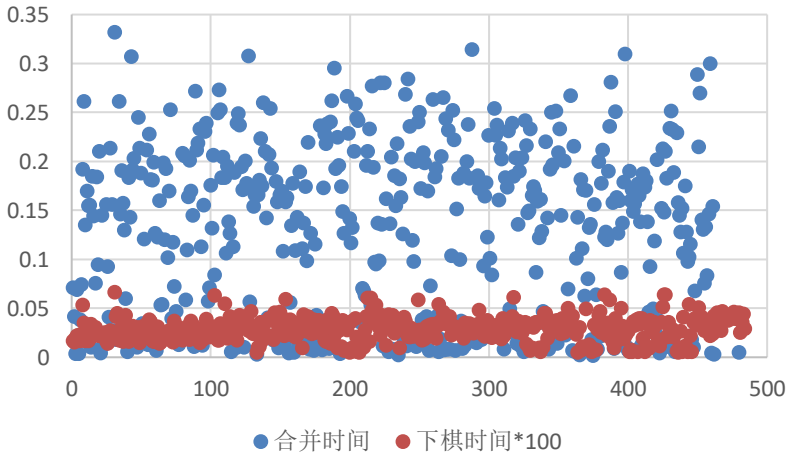


图 3　合并和下棋用时对比

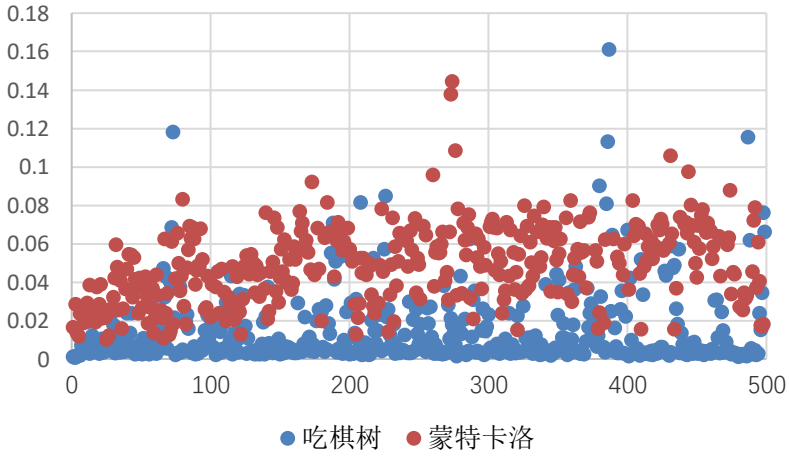


图 4　吃棋树和蒙特卡洛用时对比

2 程序代码说明

2.1 数据结构说明

算法中使用的数据结构是“吃棋树”，是一种比较精简的树结构。该“吃棋树”与课程中提到的数据结构“树”的不同有三点：

第一，该结构是四叉树，故在实现时与二叉树不同，采用一个字典来存储子节点。

第二，该结构还具有给吃棋评分的功能，所以节点中也要添加相关的内容。

第三，该结构代表着双方的轮流合并，在节点中需要存储相关信息。

一个节点代表的是游戏进行的一个状态。下面是树的节点 Node 的类定义代码：

```
1.  class Node:
2.      def __init__(self,direction=None,score=0,belong=None):
3.          self.direction = direction
4.          self.score = score
5.          self.parent = None
6.          self.childlist = {}
7.          self.belong = belong
8.          self.result = None
9.          self.board = None
```

下面逐一说明__init__方法中各个属性的含义：

self.belong：区分双方属性。一个节点和它的子节点、父节点（如果有的话）的 belong 属性应该都是相反的。这里规定 belong 是在这个节点处接下来要进行移动操作的一方。那么 not belong 便代表着从上一个节点到这个节点的过程中进行了移动操作的一方。

self.direction：记录移动方向。即（not belong 一方）从上一个节点向 direction 方向移动以后产生了此节点。根节点该属性值为 None。

self.score：记录移动中吃棋的分数。即从上一个节点向 direction 方向移动的后，移动的一方（not belong 一方）吃掉另一方的棋子的总分数，该分数为非负数。

self.parent：该节点的父节点。

self.childlist：该节点的所有子节点。采用的字典的结构，字典的 value 值即是一个子节点，其 key 值取为从当前节点变成其子节点时移动的方向（即这个子节点的 direction 属性），这样便于按照方向检索节点。

self.board：该节点处的盘面。

self.result：最终打分，其含义是：从当前节点开始计算，让盘面继续发展直到吃棋树结束时双方的子力盈亏（注意这里并没有包括上一步双方是否发生了吃棋）。在实际运行中，result 为正分，代表从这个节点开始之后的发展中我方会获利；反之，result 为负分代表从这个节点开始之后的发展中我方会亏损。局部的盈亏并不能代表总体，最关键应当是整棵树的根节点的 result。

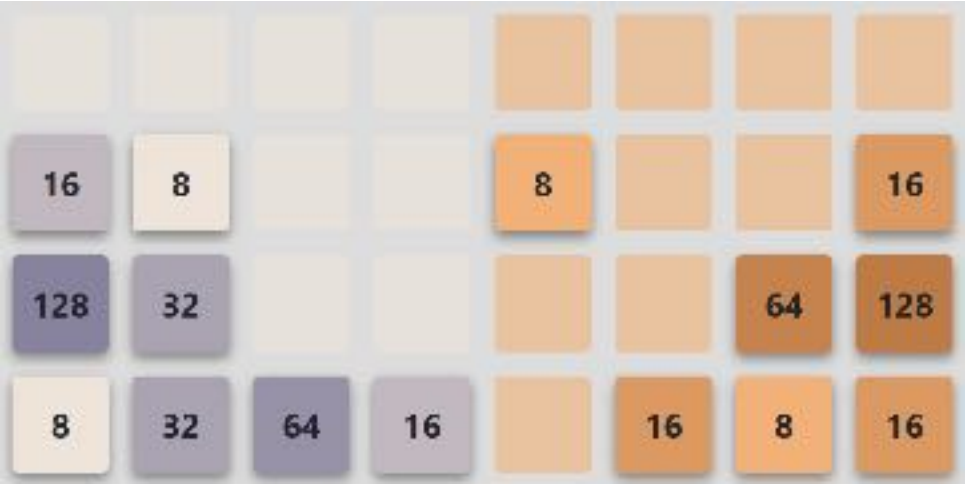


图 5 吃棋树示例局面 1，34 回合，先手

http://gis4g.pku.edu.cn/ai_arena/match/gdKlbRYyl0/2/

下面以上图局面为例说明吃棋树的生成和评分原理。

如图，现在轮到先手方合并。创建一棵吃棋树，其根节点（0 号节点）存储了现在的盘面信息。此时先手方有两种选择：方向 0 和方向 3，分别对应下一层的 1 号和 2 号节点。其中先手方选择方向 0 时不会吃掉对方棋子，所以 1 号节点中记下 0 分（score）；先手选择方向 3 时会吃掉价值为 2 的棋子，所以 2 号节点中记下 2 分。按照这个规则继续添加节点。

树的结束条件有两个，一个是深度达到阈值（self.depth，最终选取为 8），另一个是连续两步双方都没有吃棋，比如从节点 1 向后发展，对方无论选择什么方向都不会发生吃棋。连续两步没有吃棋时，吃棋树停止延伸。

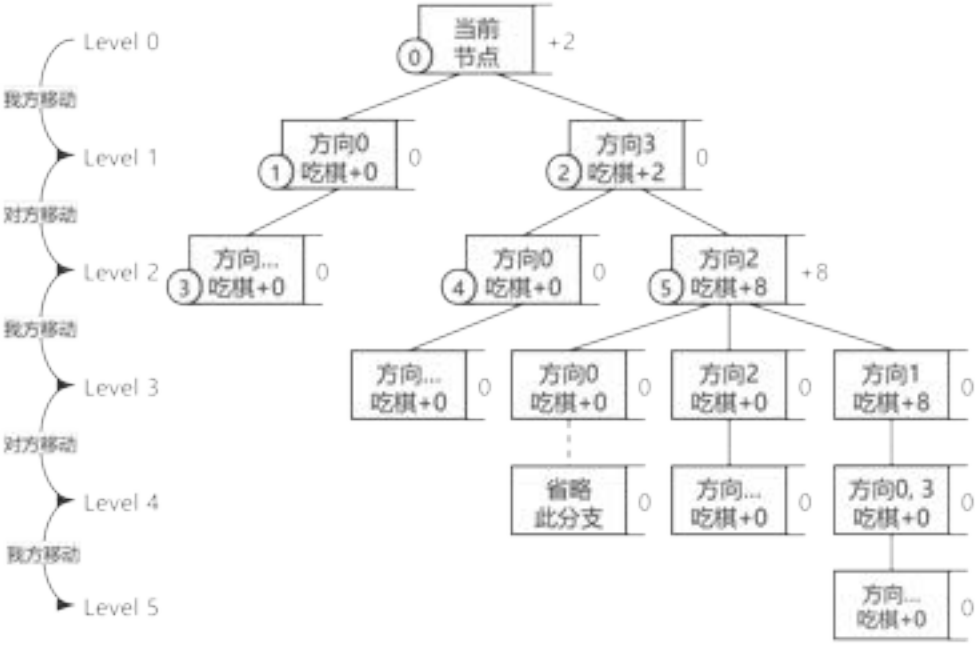


图 6 吃棋树示意图

按照规则生成整棵树后，从叶节点开始最终评分（result）。叶节点评分应当为 0。最终评分从下方逐级向上评分。经过逐级评价，3, 4, 5 号节点的最终评分为 0, 0, +8。由于从 1 号节点到 3 号节点也没有吃棋，1 号节点评分应当是 0。下面求 2 号节点的最终评分。在 2 号节点之后，如果对方选择节点 4，这一步不会吃棋，并且 4 号节点最终评分是 0，得到的评分应当还是 0；如果对方选择节点 5，这一步对方吃掉了我方价值为 8 的棋子，而 5 号节点的总分是+8，说明在这之后我方可以赚得价值为 8 的棋子，二者相互抵消，得到的评分还是 0。由此得到 2 号节点的评分是 0。1 号和 2 号的最终评分都是 0，从根节点到 2 号节点我方吃掉了对方价值为 2 的棋子。所以根节点的评分是 2，最佳的移动方向应当是节点 2 的方向，即方向 3。通过吃棋树，我方得到了正确的方向，在此回合中获利。

图 7　吃棋树示例局面 2，35 回合，先手

http://gis4g.pku.edu.cn/ai_arena/match/gdKlbRYyl0/2/

2.2 函数说明

2.2.1 通用函数接口

为使得后续合并、吃棋、下棋部分函数更为简洁可读，本部分主要是一些通用函数接口，包括不同的评分策略、获取当前对局信息等。

2.2.1.1 评分策略

由于不同局势采用的策略不同，对应的评分策略也不同，现将其一一列举，后续使用时不再详细说明评分方法。

scoreBoard(self, board, belong) -> float：给定一个棋盘 board，给定一方 belong，进行评分。此评分为经典的 2048 游戏中的评分方法。在经典的 2048 游戏中是按照相同棋子合并时方棋子的大小进行加分。如产生一个 2 级棋子（即 4）需要一次 2 与 2 合并成 4，分数就是 4。产生一个 3 级棋子（即 8）需要有两次 2 与 2 合并成 4，以及一次 4 与 4 合并成 8，所以分数是 4+4+8=16。对于一般的级数 x 做同样的分析可以得到其对应的总分为 2**x*(x-1)。scoreBoard 是将所有棋子按这种方式求得的总分求和。这个评分的特点是：合并更快的棋盘将得到更高的分数，更大的数发生合并将得到更高的分数。

```
def scoreBoard(self, board, belong) -> float:
    scoreList = list(map(lambda x: 2**x*(x-1), board.getScore(belong)))
    score = sum(scoreList)
    return score
```

special_scoreBoard(self, board, belong) -> float：给定一个棋盘 board，给定一方 belong，进行评分。scoreBoard 中 1 级棋子的评分为 0。special_scoreBoard 在前者基础上进行了轻微的调整，将棋子打分规则变成了 2**x*x，使得 1 级方块的得分并不是 0。同时保留了 scoreBoard 的特点：合并更快的棋盘将得到更高的分数，更大的数发生合并将得到更高的分数。

```
def special_scoreBoard(self, board, belong) -> float:
    scoreList = list(map(lambda x: 2**x*x, board.getScore(belong)))
    score = sum(scoreList)
    return score
```

sumBoard(self, board, belong) -> float：给定一个棋盘 board，给定一方 belong，进行评分。按照每个棋子的大小 2**x（x 为其级数）直接计算 belong 方棋盘中棋子的总分。在没有下棋的时候，belong 一方的 sumBoard 评分发生变化说明在合并中吃掉了对方的棋子、或者有棋子被对方的棋子吃掉。这个方法主要运用在“吃棋树”里。

```
def sumBoard(self, board, belong) -> float:
    scoreList = list(map(lambda x: 2**x, board.getScore(belong)))
    score = sum(scoreList)
    return score
```

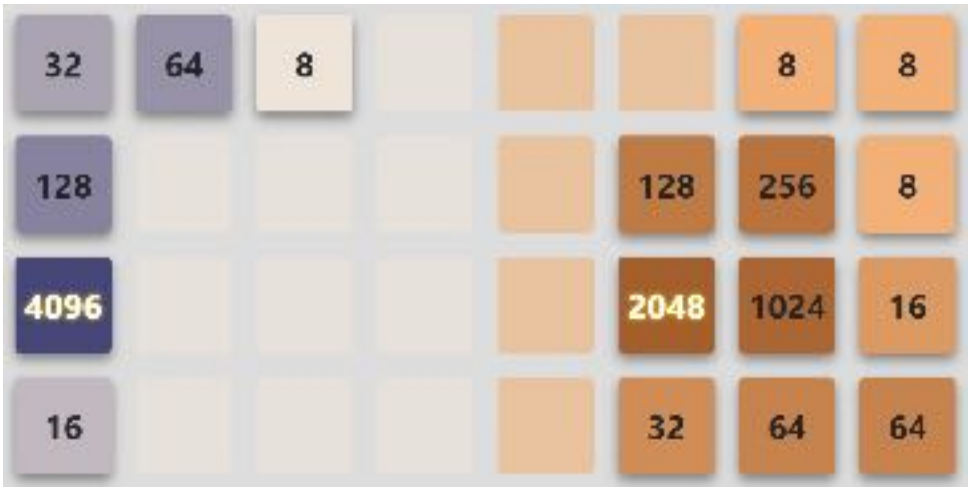


图 8　同一棋局下各种不同的评分函数值

scoreBoard：先手：10338；后手：7366

special_scoreBoard：先手：9284；后手：6452

sumBoard：先手：1054；后手：914

2.2.1.2 单调性与平滑性

为了将大数聚集到一块易于合并，同时使得盘面更简洁有规律，该方法对局面单调性和平滑性进行评估。

单调性：试图确保棋子的级数沿着同一个方向增加或减少，这样的评价函数也会使得较大级数的棋子聚集在角落，使得盘面井井有条。

平滑性：趋向于减小相邻棋子的级数差，利于相同级数的棋子合并。

具体评分思路：通过最大棋子的位置或所用策略选择一个或两个维度的单调方向，将该方向上的所有我方非空格的棋子依次排列，前一个棋子的级数减去后一个棋子的级数得到级数差。若级数相同，则总分加 1；若为顺梯度，则总分加上该级数差的倒数；若为逆梯度，则总分减掉该级数差。最后使用 combineScore(self, board, belong) -> float 将 mono 值与棋盘评分结合，给出一个合并的评分。

此评分规则在原评分规则的基础上，同时兼顾了单调性与平滑性，使用该评分函数之后的局面也更为有条理。但由于某些原因，这种评分在实际的对战中表现不佳，在最后的代码中未能使用，会在第四部分详细说明。

getMono1(self, board, maxCell, belong) -> float：最大级数棋子有多个时，只考察左右维度的单调性，其中 maxCell 为最大级数棋子列表；返回一个单调性函数值

getMono2(self, board, maxCell, belong) -> float：最大级数棋子只有一个时，考察两个维度的单调性，其中 maxCell 为最大级数棋子的位置；返回一个单调性函数值。横向维度与纵向维度的比例为 0.8：0.2。

getMono3(self, board, belong) -> float：对于吃棋主导的盘面时，尽量让大数靠近分界线，规定单调性从中间向两边递减，返回一个单调性函数值。

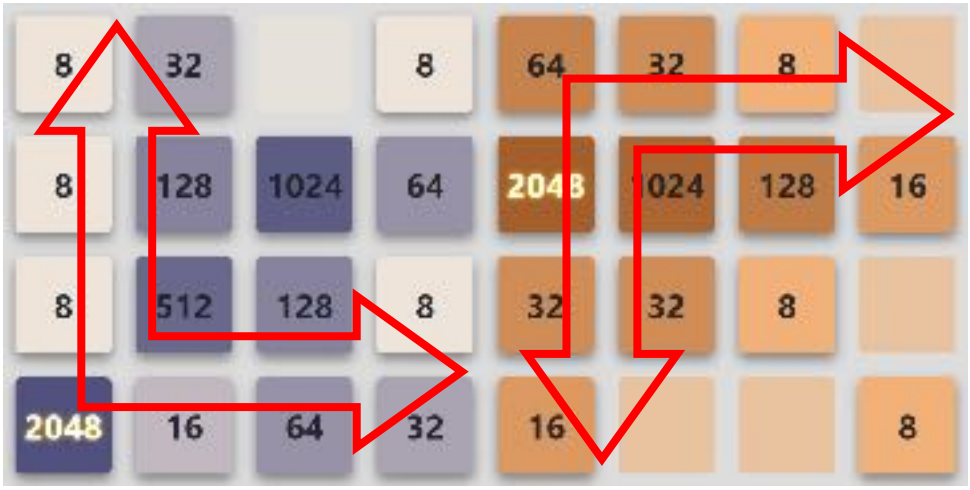


图 9　getMono 单调性方向与评分，显然后手的单调性与平滑性更好

getMono1（一个维度）：先手：-16.36；后手：5.67

getMono2（两个维度）：先手：-13.69；后手：2.26

getMono3（中间最大）：先手：-18.25；后手：5.67

2.2.1.3 获取对局信息函数

getStatus(self, board, position:tuple)：返回目标位置 position 的归属与值(belong, value)

getWinner(self, myChess, eneChess)：返回当前棋盘的胜者，其中 myChess 和 eneChess 分别是自己和对方棋盘的所有棋子级数列表。

isOverStep(self, cell)：查看当前位置是否越界，越界返回 1，不越界返回 0。

findMyMaxCell(self, board, myMax)：找到最大值位置，返回一个 Pos 列表。

hasEaten(self, board, belong)：判断某方有没有吃棋，吃棋返回 1，没吃棋返回 0。

2.2.2 合并函数接口

2.2.2.1 合并通用函数接口

testEat(self,board,direction,belong)：模拟 belong 方将 board 向 direction 方向移动，并返回 belong 方棋子总和的增加量。如果不能移动，则返回 None。这个方法用来计算在合并中吃棋的总大小。

testCombine(self,board,direction,belong)：模拟 belong 方将 board 向 direction 方向移动，并返回 belong 方棋子 scoreBoard 属性的增加量。如果不能移动，则返回 None。这个方法用来计算在移动中发生合并的棋子的总大小。

合并方案有很多，在第 1 节的思路、和流程图中展示是最终确定的版本。下面的介绍中详细介绍最终使用的方案，并简述了一些未使用函数和方案的思路。

先介绍两个被调用的方法：

self.priorityTree(board, self.isFirst)：self.priorityTree 是一个有多种方案的树结构，最终采用的是“吃棋树”，即 makeEatTree 方法。除此以外还有 two_level_eat 和 two_level_combine 两种，在 2.2.4 中会更详细地介绍。这些树返回一个记录了各个方向评分的字典。

multi_Simulation(self, currentRound, testBoard, direction, simulation)：这个方法是蒙特卡洛模拟的通用入口。给定当前的回合数和棋盘状态，在给定模拟的起始方向 direction，先向 direction 方向合并后再通过蒙特卡洛模拟，游戏进行 self.depth 回合后进行打分，在 self.run 次模拟后取平均分。其中模拟时有多种方案，对应函数中的 simulation 属性。在 2.2.5 中会更详细地介绍。

```
1.  def multi_Simulation(self,currentRound,testBoard,direction,simulation)
2.          actual_count = 0
3.          total_score = 0
4.          board = testBoard.copy()
5.          if not board.move(self.isFirst,direction):
6.              return -10000
7.          for i in range(self.run):
8.              endBoard = self.simulationMode(currentRound, board)
9.              score = self.totalScore(endBoard,self.isFirst)
10.             total_score += score
11.         average = total_score / self.run
12.         return average
```

合并部分通过在 directionMode 函数中调用 self.bestNextMode 函数来进行（见 1.2）。

self.bestNextMode 代表着寻找最佳合并方向的方案。方案共有两种，分别是 bestNext 和 bestNext2。其中 bestNext 只有蒙特卡洛模拟，bestNext2 是吃棋树和蒙特卡洛模拟的结合。最终采用的是 bestNext2 方案，与 1.2 节的流程图一致。代码如下，其中第 2-4 行得到吃棋树的最高分，5-6 行判断最高分是否满足阈值条件，7-12 行为蒙特卡洛模拟。

```
1.  def bestNext2(self, board):
2.          res = {}
3.          preference = self.priorityTree(board, self.isFirst)
4.          best = max(preference, key = preference.get)
5.          if preference[best] > self.parameterup or preference[best] < self.pa
    rameterlow:
6.              return best
7.          for aDir in list(preference.keys()):
8.              if preference[aDir] < self.parameterlow:
9.                  continue
10.             res[aDir] = self.multi_Simulation(self.currentRound, board, aDir
    ,self.simulationMode)
11.         best = max(res, key = res.get)
12.         return best
```

2.2.2.2 合并树相关函数

合并策略中的树有三种。

two_level_eat(self, testBoard, belong)：该树使用了嵌套列表。双方各进行一步合并，其中后合并的一方将选择对自己最有利的方向，最后返回一个字典，记录着 belong 一方向各个方向移动的评分。评分的方法是 sumBoard。

two_level_combine(self, testBoard, belong)：将 two_level_eat 的评分标准换成 scoreBoard。

两棵树的区别在于评分的规则不同。其缺陷是共同的——树的搜索深度有限，在复杂局面下容易得到对己方不利的结果。将其层数扩展到更多，便得到“吃棋树”。

makeEatTree(self, testBoard, belong)：吃棋树中包括了制造吃棋树、逐级评分，返回列表三个过程，函数分为三段。

part1: def makeTree(testBoard, belong, currentNode, status=0, level=0)：采用递归的形式。这一部分是制作吃棋树的过程，从 currentNode 开始向各个可能的方向发展这棵树。Status 属性用来记录有几步没有发生吃棋。level 用来记录吃棋树的层数，吃棋树层数的限制是 self.depth，最终定为 8。

part2: def getTreeResult(node)：打分方式采用“min,max”思路，即认为我方每次都会挑尽量有利与自己的，对方也会给出最强的应手。该函数也是递归的，关键在于分数的传递。代码如下。其中第 7,11 行是“min, max”思路下的评分流程。

```
if not node.childlist:
    node.result=0
else:
    if node.belong==self.isFirst:
        for cnode in node.childlist.values():
            getTreeResult(cnode)
        node.result=max(cnode.result+cnode.score for cnode in node.childlist
.values())
    else:
        for cnode in node.childlist.values():
            getTreeResult(cnode)
        node.result=min(cnode.result-
cnode.score for cnode in node.childlist.values())
```

part3：后面的部分是调用前两个函数，并输出要求的字典。

2.2.2.3 蒙特卡洛模拟

蒙特卡洛对游戏进行的模拟用下棋和合并模拟的接口实现。（最终合并采用的方式是 rand_myMove 和 rand_eneMove，即经典的蒙特卡洛随机模拟）。

insertTitle(self,board,currentRound)：模拟双方轮流下棋，优先在己方领地下棋。

testMove(self,testBoard,direction,belong)：检查 testBoard 是否可以往 direction 移动。

rand_myMove(self,currentRound,board)：己方向一个随机的可移动方向进行移动。

rand_eneMove(self, currentRound,board)：对方向一个随机的可移动方向进行移动

wise_myMove(self, currentRound,board)：更为“聪明”的己方移动方式。每一步都先利用 priorityTree 得到“最佳决策”。

wise_eneMove(self, currentRound,board)：更为“聪明”的对方移动方式。每一步都先利用 priorityTree 得到“最佳决策”。

蒙特卡洛模拟有多种方案。其中最终采用的是 wise_Simulation。方法介绍如下。

depthRestricted_Simulation(self,currentRound,testBoard)：限制深度为 self.depth 的模拟。当棋盘中有一方无法移动或无法下棋时就结束模拟，返回棋盘。当己方棋子的最大值变大时也会退出模拟。

Simulation(self,currentRound,testBoard)：不限制深度的模拟，结束条件与上一种模拟方法一致。

wise_Simulation(self,currentRound,testBoard)：限制深度为 self.depth 的模拟。当一整个回合双方都没有任何下棋或移动时结束模拟。这种的结束条件更为合理。下面代码中只显示了己方为先手时的模拟。代码中的 move_block 和 insert_block 属性以及 breakSimulation 函数用来了解这一回合双方是否有下棋。先手的模拟顺序为对方合并，双方下棋，我方合并。后手仅需改变模拟顺序即可。

```
board = testBoard.copy()
insert_block = 0
move_block = 0
oriChess = testBoard.getScore(self.isFirst)
if not oriChess: return board
def breakSimulation():
    return True if insert_block == 2 and move_block == 2 else False
if self.isFirst:
    for i in range(self.depth):
        if currentRound>=499:
            break
        move_block += 1 if not self.eneMove(currentRound,board) else 0
        currentRound += 1
        insert_block += 2 if not self.insertTile(board,currentRound) else 0
        move_block += 1 if not self.myMove(currentRound,board) else 0
        if breakSimulation(): break
        currentChess = board.getScore(self.isFirst)
        if not currentChess:
            break
        if currentChess[-1] > oriChess[-1]: break
else:  return board
```

2.2.3 下棋函数接口

下棋部分都是启发式算法，主要遵循以下原则：

1. 一般情况下，尽量下在己方，减少给对方送分。

2. 特殊情况包括任意一方将要被卡死、我方大子将要被对方吃掉等等，根据具体盘面的不同做出不同的下棋决策。

3. 由于代码是顺序执行的，因此代码顺序影响到决策的优先级：为了防止非法输出，先排除某方盘面已满的极端情况；然后是一系列的特殊情况处理，这些特殊情况不考虑双方得分优势，直接把给对方捣乱或保护我方最大利益作为最优解；如果没有出现上述情况，则根据下在己方是否有利来做决策。

在特殊情况时具体使用到的主要策略的函数如下（按优先级排列）：

2.2.3.1 保护策略 Protect

保护我方大子不被对方吃掉，在我方无法后退时堵住对方进攻道路。

图 10　保护 Protect 策略：484 回合，先手

http://gis4g.pku.edu.cn/ai_arena/match/hP33amczzt/0/

```
1.  danger = self.CellinDanger(board)
2.  if danger:
3.      protectCell = self.getProtect(enePos, danger[0], danger[1])
4.      if protectCell: return protectCell
```

CellinDanger(self, board)：检查是否有危险棋子，其中 board 为当前棋盘；返回危险方向和危险棋子的坐标。

首先将第三列到第六列所有我方级数大于阈值（采用阈值为 self.protectmin = 3）的棋子列入保护列表，然后按级数大到小排列，依次检查是否有危险的棋子。如果有，则返回危险棋子的位置和危险方向；否则返回空值。

由于盘面规则，该检查方法很多时候都会被中断，因此最后所用的时间也很少，并且有效的检查出了危险的棋子和危险方向，优先选择大的进行保护。

原先仅对进攻、上和下三个方向进行危险棋子的搜寻，后来发现当我方棋子深入对方棋盘内部时，对方也有可能从第四个方向退回吃棋，因此最后修改为四个方向都检查。至于保护范围没有继续扩大的原因，一是为了节约时间，二是考虑到深入对方内部三排甚至四排的棋子很难再回到本方领地，保护价值不是很大，因此没有将保护范围扩大至整个棋盘。

图 11　只有三个方向的 Protect 保护策略存在的漏洞：453 回合，先手

http://gis4g.pku.edu.cn/ai_arena/match/xPqfzF8MaL/0/

getProtect(self, enePos, direction, cell)：获得保护大棋的位置，其中 enePos 为对方空位，direction 和 cell 分别是上一步得到方向和位置；返回下棋的坐标。

得到危险棋子位置及危险方向后，在该棋子朝那个方向的位置移动一格，检查是否可以在该位置下棋。如果可以，则返回下棋位置；否则返回空值。

2.2.3.2 堵截策略 Block

对方空格数较少时在对方棋盘下棋，以便卡死对方。

图 12　Block 堵截策略：419 回合，后手

http://gis4g.pku.edu.cn/ai_arena/match/zAXOgZ1MT3/3/

```
if len(enePos) < self.blockmin:
    posList = []
    for blockCell in enePos:
        if not self.findMergeCell(board, blockCell, enePos):
            posList.append(blockCell)
    if posList:
        evilCell = (self.findEvilCell(board, posList))
        if not self.isFirst and evilCell:
            if board.getValue((evilCell[0], 4)) != 1: return evilCell
        if self.isFirst and evilCell: return evilCell
```

在对方空格数较少的时候（采用阈值为 self.blockmin = 4），在对方的空格选择一个干扰对方合并的最佳位置下棋，以便卡死对方。选择标准：该位置周围没有空格和级数为 1 的棋子；该空格四周的棋子级数和最大。

2.2.3.3 包围策略 Surround

让对方大棋周围充满小棋，阻碍对方的大棋合并。

图 13　Surround 包围策略：321 回合，后手

http://gis4g.pku.edu.cn/ai_arena/match/6mOEozwCao/5/

```
if eneMax > self.surmin:  # 对于 256 以上的数有效
    surCell = self.Surround(board, maxList, enePos) # 返回 None 或一个 Pos
    if surCell and not self.findMergeCell(board, surCell, enePos):
        return surCell
```

Surround(self, board, maxlist, enePos)：查找对方大于阈值的棋子周围是否有空位置，其中 maxlist 是对方大棋位置列表，enePos 是对方空位；返回可行位置中的最优位置。

逐个检查对方的大棋（采用阈值为 self.surmin = 8）周围是否有空位。如果有，则逐个检查该空位是否符合要求，最后选出最优位置。选择标准：该位置周围没有空格和级数为 1 的棋子；该空格四周的棋子级数和最大。

2.2.3.4 疏通策略 Unblock

对方有空位，但我方将要被卡死，即空格数较少（采用阈值为 self.myblankcell = 4），且下棋位置周围没有空格和级数为 1 的棋子时，下在对方。

当我方后手时，如果我方靠近分界线处有级数为 1 的棋子，而此时决定下在对方的位置恰好是同一行，可能会给对方带来进攻的机会，使得我方盘面更卡。因此为了避免这种情况的发生，需要多加一些判断。

图 14　Unblock 疏通策略：478 回合，后手

http://gis4g.pku.edu.cn/ai_arena/match/URZdFRBaBJ/1/

```
myBlankPos = board.getNone(self.isFirst)
if len(myBlankPos) < self.myblankcell and not self.findMergeCell(board, myPos, myBlankPos):
    surCell = self.Surround(board, maxList, enePos) # 返回 None 或一个 Pos
    if surCell and not self.findMergeCell(board, surCell, enePos):
        return surCell
    else:
        evilCell = (self.findEvilCell(board, enePos))
        if not self.isFirst and evilCell:
            if board.getValue((evilCell[0], 4)) != 1: return evilCell
        if self.isFirst and evilCell: return evilCell
```

2.2.3.5 自私策略 Selfness

开局一段时间内，最大级数较小时（采用阈值为 self.placeinmy = 7），为使我方能够快速合并出大棋，尽量下在我方：

```
if myMax < self.placeinmy:
    if myPos and self.my_Safe(myPos,board): return myPos
```

普通情况下，如果下在我方安全，则下在我方：

```
if myPos and self.my_Safe(myPos,board):
    return myPos
return (self.findEvilCell(board, enePos))
```

2.2.3.6 下棋通用函数接口

my_Safe(self, pos, board)：检查在我方下棋的这个位置是否安全，其中 pos 为我方将要在我方棋盘下棋的位置；返回该位置是否安全。

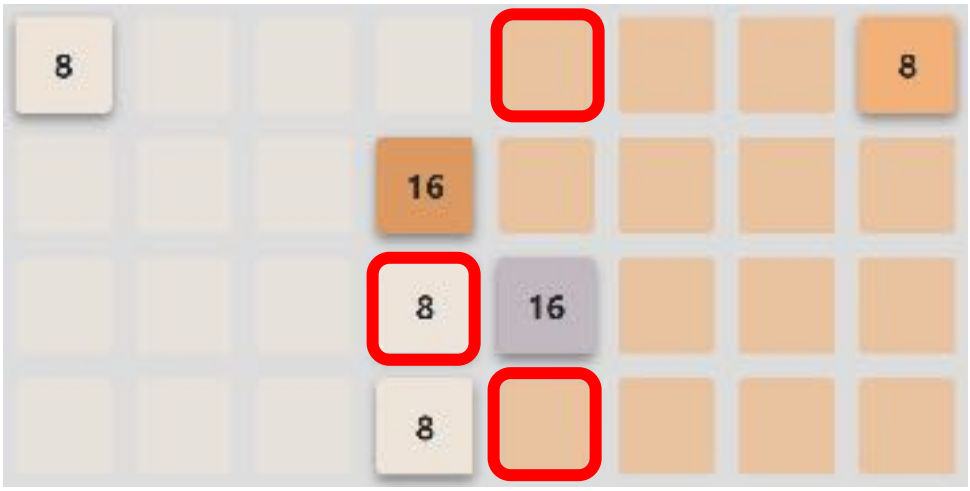


图 15　无保护的 my_Safe：3 回合，后手

http://gis4g.pku.edu.cn/ai_arena/match/Pr1gKtjsIs/5/

我方为后手时，在己方下棋时假如要下在边界位置，恰好对方同行有级数为 1 的棋子，并且我方这个位置附近没有级数为 2 的棋子对它进行保护，这时候对方可能会对我方发起进攻，占领我方领地。为了防止在己方下棋时不要喂到人嘴里，返回这个位置是否安全。

图 16　有保护的 my_Safe：196 回合，后手

http://gis4g.pku.edu.cn/ai_arena/match/Pr1gKtjsIs/5/

由于 my_Safe 仅仅通过盘面，而不是通过更深的搜索判断“安全”，因此这个安全可能是“伪安全”，仍有可能级数为 2 的棋子为了保护自己不被吃，而不能保护该级数为 1 的棋子。但由于时间的限制，并且这种情况出现概率小，以及 1 级 2 级的棋子对盘面总分影响不大等原因，最后并未加上搜索。

evilScore(self, board, cell)：对当前棋子对对方的干扰程度进行评分，其中 cell 为对方的某个空位；返回一个分数值。评分标准就是将该位置周围棋子的级数相加，如果周围有空格，则朝该方向再进一格，取新格进行计算。

findEvilCell(self, board, enePos)：返回在对方棋盘上下棋的最佳位置，其中 enePos 可能是对方棋盘的部分空格；返回 enePos 中 evilScore 最高的位置。

findMaxCell(self, board, eneMax)：找到对方最大级数棋子的位置，其中 eneMax 是对方棋子的最大级数；返回一个 Pos 列表。

findMergeCell(self, board, cell, posList)：查找给下棋的位置周围有无空位或者有无级数为 1 的棋子可以合并，其中 cell 是下棋位置，posList 是我方或者对方的空位；如果有，返回 1，没有返回 0。

2.3 程序限制

在测试过程中，程序在 499 回合有很大的概率报错。经过检查，由于模拟中包含了下棋的过程，在游戏的最后会出现 currentRound 超过 499 的情况，此时合并方法不受影响，但下棋方法的 getNext 会出现错误。最终在模拟时增加一个结束条件，在回合数超过最大回合数时停止模拟。

调整一些关键参数（self.treedepth, self.depth, self.run）后可能会出现超时的现象，测试确定 self.depth 与 self.run 的乘积不能超过 20，否则会有严重的超时现象。第 3 章有更详细的测试结果。

3 实验结果

3.1 测试数据

实验环境说明：

- 硬件配置：Intel(R) Core(TM) i7-8550U CPU @ 1.80GHz 2.00GHz / 16.0GB
- 操作系统：Windows 10
- Python 版本：Python 3.8 64-bit

我们主要通过在本地进行对战来测试不同版本代码的优劣，以此决定使用何种策略和参数。涉及超时的测试则在代码竞技场上进行。

测试内容包括策略测试和参数测试。为方便调试和修改，我们将待测试的接口名称和参数汇总起来，写在了 Player 类的__init__方法中（具体代码见附件，其中注释有“to be checked”字样的为需要测试的部分）。

3.1.1 策略测试

AI 的每个部分可以采用不同的策略，每个部分的可选策略如下：

表 1 **构建蒙特卡洛树的条件：bestNextmode**，具体代码见附件

名称	功能
bestNext	从当前局面直接构建蒙特卡洛树，依据各方向评分给出合并决策
bestNext2	1. 在构建蒙特卡洛树之前，首先确定的决策树排除掉部分不好的合并方向 2. 对于剩下的方向，再构建蒙特卡洛树 达到一定阈值时才进入的蒙特卡洛方法：采用吃棋优先策略，利用 priorityTree（见下文）的返回值，设置阈值进入模拟，即当返回值大于 parameterup 或小于 parameterlow 时进入模拟。

表 2 **决策树的构建与评分方法：priorityTree**，具体代码见附件

名称	功能
two_level_eat	层数为 2；使用 sumBoard 评分方法（表 3），只有吃掉对方的棋子才能提高分数
two_level_combine	层数为 2，使用 scoreBoard 评分方法（表 3），合并己方的棋子与吃掉对方的棋子均可以提高分数
MakeEatTree	结束条件包括：1. 到达层数限制；2. 双方均不能吃掉对方棋子；使用 totalScore 进行评分（表 3）

表 3 **决策树的评分策略：totalScore**，具体代码见附件

名称	功能
scoreBoard	给出当前棋盘的得分，得分函数为$score = \sum 2^x \cdot (x-1)$，其中$x$为己方每个格子上棋子的级数。
special_scoreBoard	给出当前棋盘的得分，得分函数为 $score = \sum 2^x \cdot x$。
sumBoard	给出当前棋盘所有棋子分数总和，即$\sum 2^x$。
getMono	给出当前单调性分数。单调性是指，棋盘中己方一行或一列上的棋子分数大小尽可能单调排列，这样有利于后续的合并。
combineScore	scoreBoard 与 mono 的结合版。

表 4　**蒙特卡洛树的结束条件：simulationMode**，具体代码见附件

名称	功能
wise_Simulation	结束条件为：双方均不能进行决策
Simulation	结束条件为：一方不能进行决策
depthRestricted_Simulation	结束条件为：1. 到达深度限制；2. 一方不能进行决策

注：三者均为蒙特卡洛树模拟方法，区别在于结束条件不同

表 5　**蒙特卡洛树中的合并方法：eneMove（敌方），myMove（我方）**，具体代码见附件

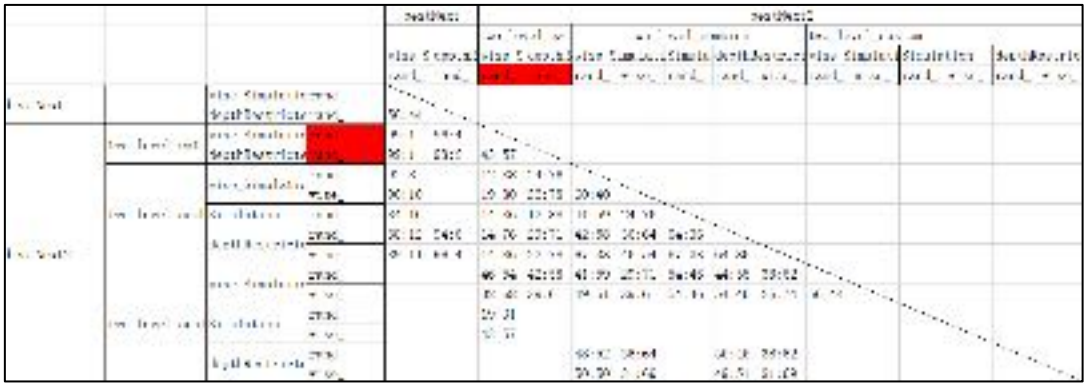
名称	功能
rand	在可能的决策中随机选择
wise	构建 2 层的决策树，选择评分最高的合并方向

注：为了简便，敌我双方的合并方法采用了相同的策略。

以上这些策略可以有多种组合。通过将不同的组合两两对战，可以判断各个组合的强度。由于 makeEatTree 是开发中期加入的方法，一开始的组合中不包括它。由于一些组合会导致超时，表 6 仅记录了未超时组合的对战结果。

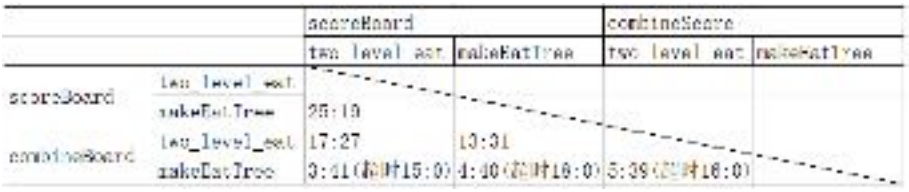
结果显示，bestNext2 + two_level_eat + wise_Simulation + rand 和 bestNext2 + two_level_eat + depthRestricted_Simulation + rand 这两个组合（即图中标红的两个组合）与其他组合对战时优势明显，其中前者又略胜于后者。这为之后的优化提供了基础和方向。

表 6　策略组合对战结果



加入 makeEatTree 方法后，进一步测试了 makeEatTree 与 two_level_eat 对 AI 决策能力的影响（表 7），发现 makeEatTree 强于之前的 two_level_eat。至此我们确定了最终采用的策略组合（表 8）。

表 7　makeEatTree 与 two_level_eat 对 AI 决策能力的影响



注：totalScore 选择了 scoreBoard 方法，makeEatTree 的层数限制为 8

表 8　最终策略

构建蒙特卡洛树的条件：bestNextmode	bestNext2
决策树的构建与评分方法：priorityTree	makeEatTree
蒙特卡洛树的结束条件：simulationMode	wise_Simulation
蒙特卡洛树中的合并方法： **eneMove（敌方），myMove（我方）**	rand_eneMove rand_myMove
决策树的评分策略：totalScore	scoreBoard

3.1.2 参数调整

确定策略后进行调参工作。需要调整的参数和对战结果如下：

1. parameterup, parameterlow：进入蒙特卡洛搜索的阈值，当 priorityTree 的返回值大于 parameterup 或小于 parameterlow 时进入模拟。“0, -5”表示 paremeterup = 0, parameterlow = -5，以此类推。

表 9　parameterup, parameterlow 的测试结果

	0, 0	0, -5	5, -5	10, -5	15, -5	20, -5
0, 0						
0, -5						
5, -5	20:80					
10, -5	24:76	7:13				
15, -5	29:71	15:35		62:38		
20, -5		12:38	40:51	55:45	47:52	

2. blockmin：对对方大于该值的棋子实行 block 策略，破坏对方的单调性和平滑度（注：该方法与最终代码使用的 block 策略有所不同：最终代码的 block 策略是指当对方空格较少时在对方下棋将其卡死）；surmin：对对方大于该值的棋子实行 surround 策略，使对方的大数周围充满小数，阻碍对方合并出更大的数。“4,4”表示 blockmin = 4, surmin = 4，以此类推。

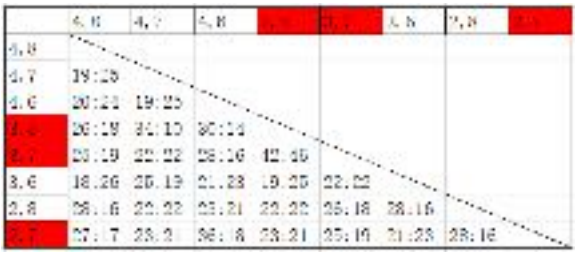
为了简洁，此处只放了 blockmin = surmin 时的测试结果。其它数值组合下的测试结果与之类似，有胜率差距较小、循环克制的特点。

表 10　blockmin, surmin 的测试结果

	4, 4	5, 5	6, 6	7, 7	8, 8	9, 9
4, 4						
5, 5	52:48					
6, 6	56:44	47:53				
7, 7	54:46	46:54	53:46			
8, 8	47:53	48:52	48:52	43:57		
9, 9	53:47	53:47	47:53	50:50	43:57	

3. myblankcell：我方空格数小于该值时，下在对方；placeinmy：我方最大级数小于该值时，下在我方。“4,8”表示 myblankcell = 4, placeinmy = 8，以此类推。

表 11　myblankcell, placeinmy 的测试结果



4. leveldiffer：判断优先的最大级数差值，即当我方棋子最大级数比对方最大级数高几级以上时视为我方占优势。我方占优势时将棋子下在对方棋盘，以求堵死对方（注：最终代码停用了该方法）：

表 12　leveldiffer 的测试结果

	1	2	3	4
1				
2	27:23			
3	23:27	26:24		
4	29:21	24:26	29:21	

5. protectmin：我方大于该值的棋子进入保护列表，搜索其是否有被吃风险，如有，则在该棋子与对方可能进攻的棋子之间下棋，阻挡对方的进攻道路：

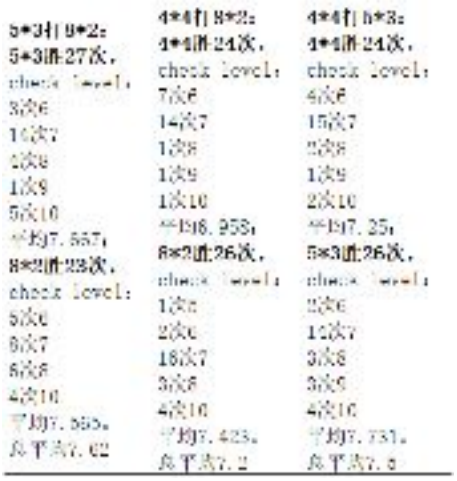
表 13　protectmin 的测试结果

	1	2	3	4	5
1					
2	24;26				
3	23:27	25:25			
4	27:23	24:25	23:28		
5	23:27	24:26	26:24	25:25	

综合上述数据，我们发现，(parameterup, paremeterlow)为(0, 0)时有明显优势；(myblankcell, placeinmy)为(3, 8)(3, 7)(2, 7)时有微弱优势；其他参数没有明显区别。

最后对 run 和 depth 进行调参，发现在对战平台压力小的时候，run*depth 大于 32 时开始出现超时；对战平台运行程序多、压力大时，乘积大于 20 时开始出现超时。考虑到正式天梯赛时服务器运行负荷较大，我们主要测试了乘积较小时（4*4, 5*3, 8*2）的对战情况，既看胜率，也考虑 check level，结果如下：

表 14　run 和 depth 的测试结果



可以看到三者胜率相差不大，4*4 获胜时 check level 明显较低，5*3 获胜时 check level 略高。最终选择了 run = 5, depth = 3。

3.2 结果分析

3.2.1 策略的优劣比较

在 3.1.1 中，我们通过在本地运行决出了胜率较高的策略组合，并在天梯上测试保证这些策略不会严重超时。下面对这些策略的强弱原因做定性分析。

1. 对于构建蒙特卡洛树的条件（bestNextmode），从当前局面直接开始构建蒙特卡洛树的 bestNext 与在一定条件下才进行模拟的 bestNext2 的区别是显然的。后者由于采用了吃棋优先的算法，故更具有进攻性，更易将对方棋子占为己有，胜率更高。

2. 在决策树的构建与评分方法（priorityTree）中，专注于吃棋的 two_level_eat 和同时看重吃棋与合并的 two_level_combine 算法均只进行了浅层判断，而吃棋树 makeEatTree 用树结构对吃棋做了更深的模拟，可以模拟双方连续进行互吃等情况，能够考虑到未来更多的棋局可能。

3. 蒙特卡洛树的模拟方法（simulationMode）中最强的方法是 wise_simulation，即双方均不能进行决策时才结束模拟。相比只要有一方无法进行决策就结束模拟的方法，该方法会在模拟中花费更多的时间，故在 wise_simulation 中加入了剪枝策略。从测试及比赛结果来看，其对运行时间的控制是成功的。

4. 对于蒙特卡洛树中的合并方法，需要指出的是，为了简便，以上测试中，我方合并（myMove）和敌方合并（eneMove）所使用的方法是相同的。在测试中发现，二者均采用构建 2 层决策树的 wise 方法时，超时非常严重（无论在 wise 中判断最好和最坏情况所用的参数 searchlimitup 和 searchlimitlow 如何取值），但不能排除 wise 在时间允许的情况下强于随机方法 rand 的可能，故又对 wise_myMove 与 rand_eneMove 组合（记为组 1）、rand_myMove 与 wise_eneMove 组合（记为组 2）以及 rand_myMove 与 rand_eneMove 组合（记为组 3）做了测试，测试结果见表 15：

表 15　rand 和 wise 组合的测试结果

run=5, depth=3	6:4(组1超时4次，组2超时6次)	0:10(组1超时10次)	1:9(组2超时9次)
run=4, depth=3	20:18(组1超时7次，组2超时11次)	18:20(组1超时20次)	15:20(组2超时18次)

可见，即便敌我合并方法中只有一个使用 wise 方法，超时仍然严重。减小模拟的次数 run 和深度 depth 可减少超时，但情况并未发生根本变化，且考虑到 run 和 depth 过小会影响模拟效果，故决定还是两者都用 rand 方法。

其实从对战数据可以看出，若除去所有超时的情况，那么 wise 方法略强于 rand 方法。这是因为前者在每次移动前都用 2 层搜索树，更加智能，而后者的移动是完全随机的。当然，也正是由于这个原因，wise 方法所需的时间开销远大于 rand 方法，在此处使用搜索树所带来的优势并不能消除超时带来的劣势，故应当将时间分配给程序的其它环节。

在最终选取的策略组合中，主要的运行时间开销发生在蒙特卡洛树模拟阶段（wise_simulation）。

3.2.2 对部分参数所参与的策略的分析

1. 进入搜索的阈值（parameterup 和 parameterlow）均为 0 时最强，其实让我们感到有些惊讶，因为将它们设为 0 就意味着即便只是能吃掉对方分数为 2 的棋子，我方也会放弃合并自己分数较大的棋子，优先去进攻对方。这个结果说明，进攻型 AI 的确在赛场上有很大的优势。究其原因，可大致分为两点：进攻能够在增加己方棋盘得分的同时减小对方的棋盘得分，且进入对方领地能够减少对方的移动空间。这也说明 bestNext2 这种吃棋优先策略对于攻击性的提高乃至胜率的提高是非常有效的。

2. blockmin 和 surmin 所参与的 block 策略和 surround 策略有利于将对方卡死、阻止对方的大数合并。

3. myblankcell 和 placeinmy 参数用于快速判断向哪一方下棋。在我方空格少于一定值时向对方下棋，有利于防止我方丧失移动空间；在我方棋子最大级数小于一定值时向我方下棋，有利于在前期尽快获取资源、增加级数。

4. protectmin 参数参与保护我方分数较大的棋子的策略，若大数有被吃风险，则在该棋子与对方可能进攻的棋子之间下棋，阻挡对方的进攻道路。在开发这个保护策略之前，我方处在双方棋盘交界处的大数很容易被对方吃掉，而在有了保护策略之后，这种情况明显减少乃至消失。毫不夸张地说，吃大数是可以扭转战局的，故保护大数的策略在对战中发挥了很大的作用。

3.3 经典战局

本算法虽然为保守型算法，但是在吃棋博弈方面有较深的搜索层数（8 层 4 回合），加上关键局面下棋的配合，因此在双方互吃交战的过程中表现出一定的优势。本次展示的经典战局就是双方互吃的交战局面。

图 17　经典战局估值得分变化，我方先手

从估值得分变化可以看到在比赛约五分之三的阶段，双方开始交替吃棋，最终我方获得了吃棋的大优势，首先合成了 2048，并将我方被堵死的局面转化成了对方被堵死的局面，扭转了局势。

图 18　经典战局：吃棋树层层博弈，271~283 回合，先手

http://gis4g.pku.edu.cn/ai_arena/match/_S_CUktva7/3/

4 实习过程总结

4.1 分工与合作

本部分用三份表格分别呈现了分工（表 16）与协作（表 17）的方式，以及小组共同协作的历程（表 18）。

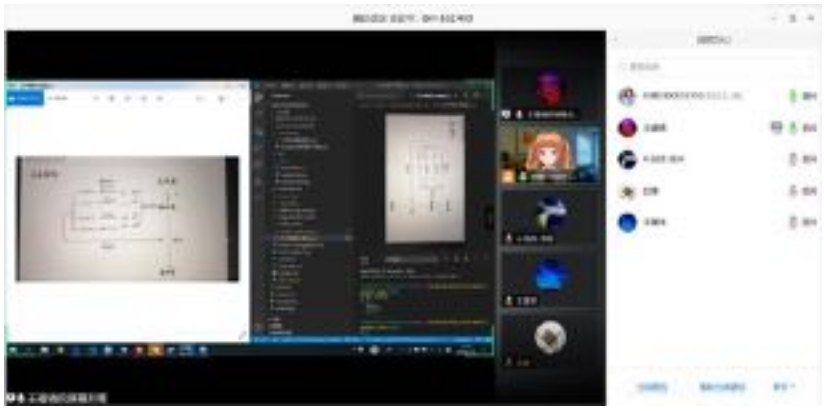
表 16：成员分工

成员	工作
叶琳（组长）	下棋算法的构思及代码编写；算法测试与参数调整
王霆浩	合并算法的优化及代码编写；算法测试与参数调整
王璠璘	合并算法的构思及代码编写
田佳	算法测试与参数调整
田缘	

表 17：合作与交流方式

平台	功能
微信群	微信群是本小组主要的交流工具，用于成员间及时地进行沟通。 通过微信群： （1） 算法构思者与代码编写者可以进行互相沟通 （2） 代码编写者与算法构思者向程序测试者发布任务，测试者则给予反馈；通过反馈，前者能更好认识当前算法及代码的合理性及改进方向 （3） 每位成员都可以在群里放出 bug/AI 不合理行为的截图，与负责该部分代码的同学进行沟通并尝试解决 （4） 小组成员间还可以日常的交流与鼓励
微信私聊	为了不影响其他成员，两个成员之间更密切的协作还可以通过私聊完成
腾讯会议	用于进行更高效的语音与视频组会
Gitee	代码的共享；代码进化日志的保存

表 18：协作剪影

大事记	截图为证
Masters 的 第一次圆桌会议	
第一次任务发布	

Git 学习

头脑风暴

日常交流

比赛当日	

（PS：组长毒奶体质，一看比赛就必输，四强赛最后一刻实在忍不住看了比分，最后 24:26 惜败 N19-666 组，未能进入 N19 四强）

4.2 经验与教训

4.2.1 代码版本不统一

代码版本不统一是本次实习工作的最大问题。由于组员们对 git 并不十分熟悉，难以使用 git 进行高效合作；在工作后期，已经编写出了众多不同版本的程序。它们相互之间区别不大，但确实存在区别。

由于程序代码较为复杂，加之编写程序的人员较多，程序员互相之间往往使用版本不同的代码进行工作。更为严重的是，每人会使用不同的习惯进行命名，相关的修改说明也没有在代码中清楚地解释，导致了一系列 bug 修复与代码合并方面的问题。

可能的改进方案包括：

1. 要求程序员在向他人提交自己的工作时，必须在程序的开头写上本版本的修改之处；如果使用 git 协作，则还需要在提交日志中写明最为重要的更改。
2. 虽然参与比赛得代码必须整合到一个 py 文件之中，但在平时得工作中也可以使用 module 的方式进行编写；这样，每人都能够更好地维护与扩展自己的部分，而不用担心妨碍了他人的工作。

4.2.2 编程工作于调参工作协调不够

参数的设置有时对 ai 的强度具有重要的作用，有时也能为算法的改进提供启示。然而，参数的设置需要通过不少的比赛进行试验，程序员无力承担相关的工作，因此交由负责调试的同学进行把控。

交接的过程中，程序员没能良好地表达对于参数设置范围及参数效果的需求，负责调试的同学起初也没能清晰地向程序员呈现自身的工作，一定程度上阻碍了程序的继续进步。

4.2.3 程序内部运行过程把控不足

在工作中后期，由于程序的复杂性，许多 bug 并不能由单步调试发现；同时，AI 在不少情况下会表现出反常行为。程序员主要通过 print 检查程序内部的运行过程，但这会遗落众多的中间过程。

可能的改进方案包括：将程序的内部变量输出到外部的文本文件中，从而可以更清晰地理解程序出现的问题。

4.3 建议与设想

4.3.1 对实习作业的建议

单淘汰赛这一赛制适合于 8 名选手实力相近，且不会互相克制的情况。但对于同学们编写的 AI，这两种假设往往不成立。希望在以后的竞赛中设置败者组及复活赛机制，以便更好地反应各个 AI 的强度。

4.3.2 实习作业的后续工作

实习的最后，本组也考虑了能够进一步降低复杂度的新算法；但由于时间限制，最终没能完成，现简要陈述如下。

4.3.2.1 总体思路

在之前的算法中，存在着一些不良特性，包括：

1. 重复计算不利于复杂度的降低。AI 每次都会从当前棋盘创建节点并生成博弈树。然而，新生成的棋盘中，有不少都与上一次决策中所生成的棋盘极为相似（因为博弈树的构建过程中，没有对方 AI 的下棋过程，所以并不与新生成的棋盘完全相同），这表明这一算法存在着重复计算的问题。

2. 下棋策略并未采用博弈树，智能受限。实战中出现了以下局面（图 19）；由于下棋策略完全基于当前盘面考虑下棋位置，并未构建博弈树，因此并未给出合适的下棋位置。虽然可以通过参数调整解决 AI 对这一盘面的判断问题，但这也说明不采用博弈树的下棋策略存在着本质上的缺陷，而这一缺陷阻碍了下棋策略的进一步提升。

图 19　未加搜索，自寻死路，下棋后只能向右移动，送掉了 2048 398 回合，先手，http://gis4g.pku.edu.cn/ai_arena/match/DAAtY_uYq_/0/

图 20　未加搜索，“多余”的 Protect 保护策略，338 回合，先手 http://gis4g.pku.edu.cn/ai_arena/match/nbO9nrKrf1/0/

3. 合并策略构建的博弈树的不包括下棋过程，可能存在缺陷。合并策略所构建的博弈树考虑了多次交换棋子的状况，比如如下对局过程：

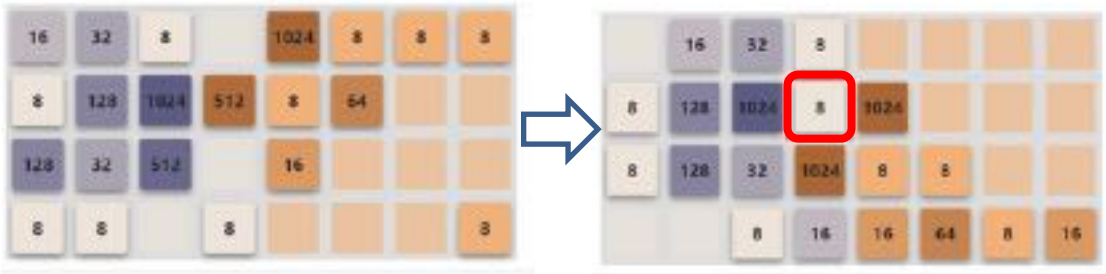


图 21 博弈树博包括下棋过程的缺陷，先手（因对方 ai 代码删除，无链接）

对于这一对局过程，AI 会判断我方优势，从而进行第一步向右合并，“引诱”对方向下合并吃掉我方 512，以便我方吃掉对方的 1024。然而，由于对方可能会通过下棋阻碍我方的 1024 吃掉对方的 1024，因此以上的第三步可能不能进行。这表明，由于未引入双方的下棋

过程，AI 所构建的博弈树也可能存在一定缺陷。

4. 对于盘面的掌控力度不够，单调性和平滑性不佳。在测试过程中，发现两种评分方法 scoreBoard（只考虑棋子级数）和 combineScore（同时考虑棋子级数和单调性、平滑性）在实际应用中，前者的表现更好，为此我们对最后的代码进行了测试：对两种不同评分函数进行了平滑性和单调性（统称为 mono 值）的测定。

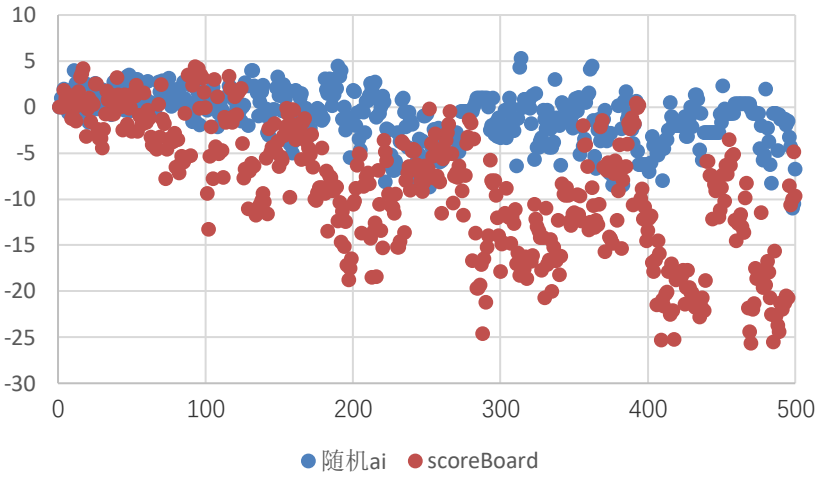


图 22　scoreBoard 为评分策略的 mono 值变化

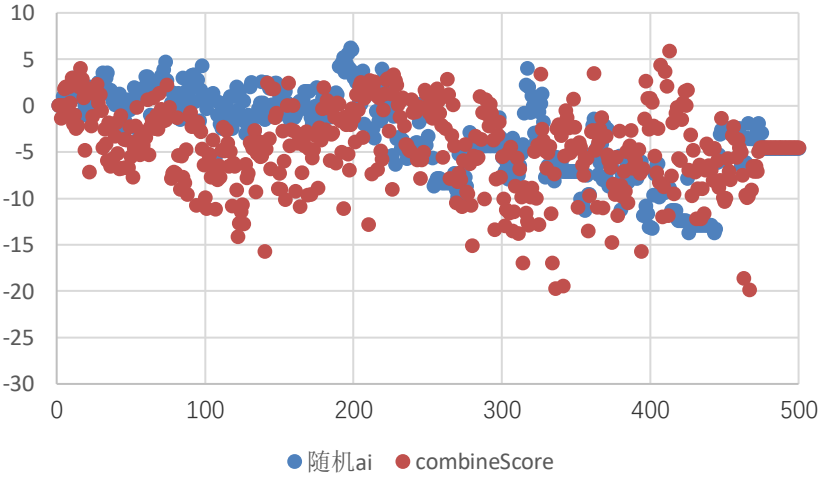


图 23　combineScore 为评分策略的 mono 值变化

由图 22 和图 23 发现，评分策略变为 combineScore 后，盘面的 mono 值得到了一定的改善，其最小值从-25 减小到了-20；多次测试后发现该现象是一致的。但是在天梯上与其他算法相互博弈时，其表现不佳，分析原因如下：

（1） 对于大数和小数没有区别，两个 1024 相邻和两个 8 相邻计算的分数是一样的。（因此上图测试时，随机 ai 的 mono 值一直高于我方 ai）

（2） mono 和 scoreBoard 的占比不当，有时会发生为了维持高 mono 值而不合并大数的情况，导致盘面较满。

（3） mono 只考虑本方棋子，对于对方占领我方领地的棋子忽略不计，不能真实地反映盘面实际的平滑性与单调性。

基于以上分析，新 AI 希望实现以下功能：

（1） 将下棋过程引入博弈树，完善博弈树

（2） 将博弈树同时引入下棋策略，进一步提高 AI 智能

（3） 重复使用已经考虑到的棋盘，降低算法复杂度，并利用节省的时间进一步提高搜索深度，或增加蒙特卡洛树的采样次数。

（4） 优化单调性和平滑性评分函数。

4.3.2.2 算法流程

新 AI 的流程如图 24，陈述如下：

（1） 将平台传入的当前局势转化成节点。

（2） 在原有博弈树中寻找新节点。

（3） 构建/扩充博弈树。

（4） 对所有可能的选择进行评分。

（5） 依据评分最高的选择做出决策。

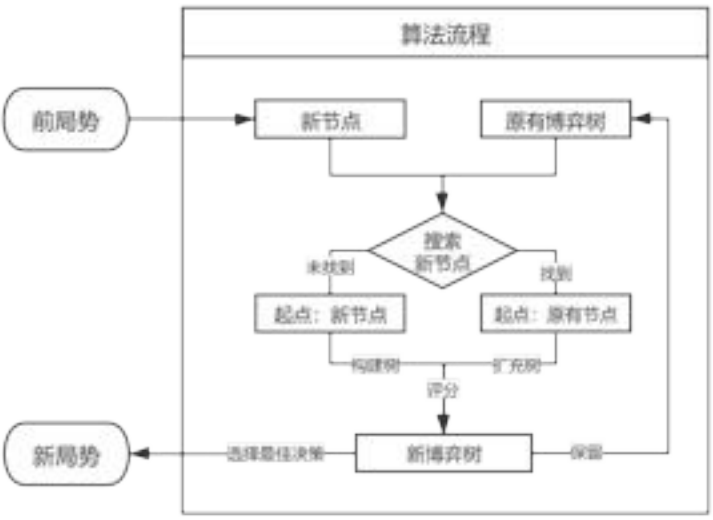


图 24　新 AI 总体流程

找到当前棋盘的流程图 25，只需依次搜索原有博弈树中对应层的节点即可。

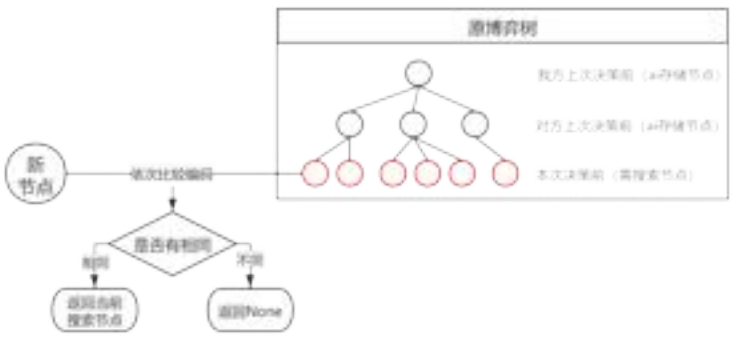


图 25　找到新节点的流程

构建/扩充博弈树并评分的流程采用了递归的方法（图 26）。

递归的结束条件包括：

（1）到达层数限制；

（2）本次合并与上次合并均没有吃掉对方的棋子。

递归结束时，直接通过棋盘估值给出本节点的评分。不满足结束条件时，会进行如下操作：

（1） 选择一个可能的决策

（2） 确保该决策位于子节点中

（3） 从该子节点开始，构建/扩充博弈树并评分

（4） 尝试所有决策并评分后，对该节点评分。

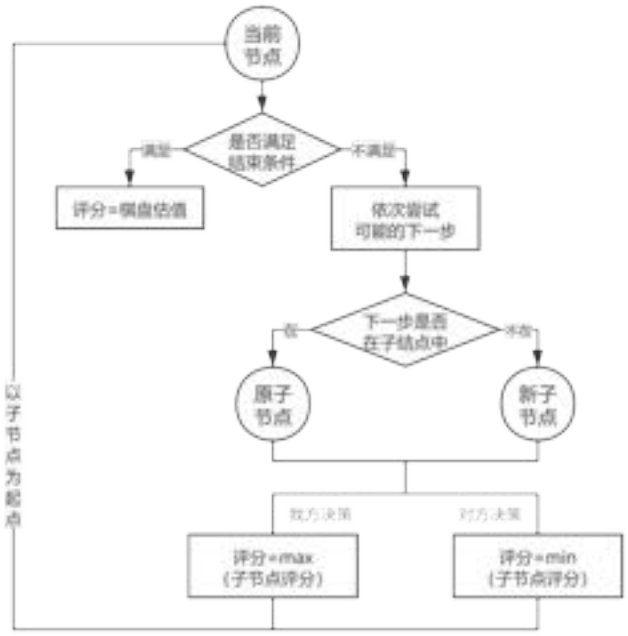


图 26　构建/扩充博弈树并评分的方法

4.3.2.3 数据结构说明

虽然采用了树的数据结构，但仅定义了节点类，而没有定义整棵树。

表 19：节点类的属性，具体代码见附件

属性	作用
board	供操作的棋盘类对象
board_id	棋盘的编码
currentRound	当前轮次
Player	Player 的标签，用于调用 Player 的方法
next_step	下一步的行动者（先手/后手）与决策类型（下棋/合并）
possible_steps	下一步的可能决策（坐标/方向组成的列表）
winning_mark	我方与对方的得分之差（估值维度 1）
eating_mark	我方棋子之和与对方棋子之和的差，判断一方是否吃棋
mono	盘面的单调性与平滑性得分（估值维度 3）
counter	我方占有的格子数目（估值维度 2）
markseries	用于评分的各个维度构成的字典，key 为权重，value 为相应分数
mark	节点最终评分

4.3.2.4 函数说明

表 20：节点类方法，具体代码见附件

名称	功能
get_winning_mark	给出我方与对方的得分之差（估值维度 1） 得分定义为：Sum（2^X*(x-1),x 为棋子级数）
get_eating_mark	给出我方棋子之和与对方棋子之和的差，用于判断一方是否吃棋 棋子之和定义为：Sum（2^X,x 为棋子级数）
gettiles	给出我方拥有的格子数（估值维度 2）
findnode	寻找新节点
updatetree	构建/扩充博弈树并评分
_mark	在 update 中调用，封装好的评分方法 选用了三个估值维度，加权求和即为节点评分 三个估值维度分别为：1：我方与对方的得分之差（权重 10000），2：我方拥有的格子数（权重 8000000），3：单调性与平滑性（权重 100）

表 21：Player 类与原 AI 不同的方法，具体代码见附件

名称	功能
output	接受来自平台的信息并给出决策
enemy_action	上一步决策的类型，用于生成新节点
insert	根据当前盘面给出可供选择的下一步下棋位置（列表）
find_vulnerable2	先手时新增的下棋策略：在对方的边界上插入一个能立刻被自己吃掉的 2，在 insert 中调用

4.3.2.5 结果分析

实验环境说明

- 硬件配置： CPU：Intel® Core™ i7-7500U CPU @ 2.70GHz 2.90GHz
  RAM：8.00GB（7.85GB 可用）
- 操作系统： Windows 10 家庭中文版
- Python 版本：3.7.0, 64-bit

表 22：新 AI 的决策能力与复杂度对比（20 次比赛，对局原 AI，本地运行）

	先手	后手
新 AI	胜率 70%，平均用时 26.883s	胜率 60%，平均用时 40.697s
原 AI	胜率 40%，平均用时 8.027s	胜率 30%，平均用时 8.326s
参数设置：新 AI		
决策树	maxtree（深度限制）	8（2 轮）
下棋策略	blockmin（block 策略阈值）	4
	surmin（surround 策略阈值）	8
	protectmin（protect 策略阈值）	3
参数设置：原 AI		
蒙特卡洛树	run（采样次数）	8
	Depth（深度限制）	2（2 轮）
决策树	treedepth（深度限制）	8（8 轮）
下棋策略	blockmin（block 策略阈值）	4
	surmin（surround 策略阈值）	8
	protectmin（potect 策略阈值）	3
	myblankcell（在对方下棋的阈值）	3
	placeinmy（在对方下棋的阈值）	7
	leveldiffer（最大级数差值）	2

（1）决策能力。当搜索深度设置为 8（即 2 轮决策）时，新 AI 的决策能力显著强于原 AI，但仍然不能取得压倒性胜利（表 22）。

（2）复杂度分析。虽然新 AI 的决策能力有所提高，但消耗的时间也出现了显著提高。其原因是节点数的大幅提高（表 22，图 27）。

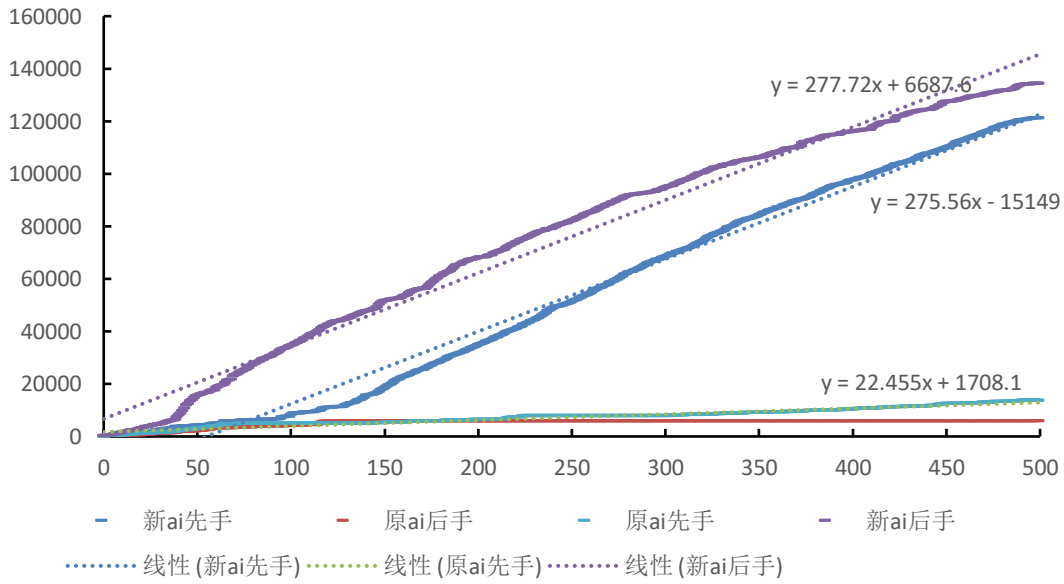


图 27　新 AI 与原 AI 的节点数比较

可以发现，新 ai 的节点数比原 ai 提高了十余倍；并且，后手时产生的节点相对较多。

这解释了新 AI 耗时显著提升，且后手耗时更多的原因。值得注意的是，虽然节点数提高了十余倍，但消耗时间仅提高了 2~4 倍，这表明程序在其他方面的开销降低了；可能原因包括：原 AI 中，蒙特卡洛树占用了不少的时间资源。

（3）目标达成情况。新 AI 中，在下棋策略中引入了博弈树，并在先手时添加了进攻性的下棋手段，对 AI 决策能力有一定的提升作用；同时，博弈树中引入了下棋过程，相较于原来深度更大（8 轮）的博弈树，当前 2 轮的博弈树也能达到不错的效果。

然而，减少重复计算的目标在当前的算法中并不能实现。完全的 2 轮博弈树包含 885 个节点（假设下棋只有 1 种选择，而合并有 4 个方向），可以复用的节点有 37 个，复用率不足 1/16。

4.3.2.6 进一步的展望：更高效的剪枝策略

完全的 2 轮博弈树包含 885 个节点（假设下棋只有 1 种选择，而合并有 4 个方向），可以复用的节点有 37 个。而依靠现有的剪枝策略（我方与对方均不能吃掉对方的棋子），每轮增长的节点数平均为 275 个，已经将每轮增长的节点数限制到了原来的 1/3 不到，但这远远不够。

可以考虑通过使用 alpha-beta 剪枝进一步减少节点数。由于在当前的评分策略下，必须要找到叶节点才可以对父节点进行评分，因此 alpha-beta 剪枝策略可以用于叶节点的剪枝，但这仍然有望进一步降低算法的复杂度。

5 致谢

感谢陈斌老师一学期的辛勤耕耘，虽然短短的一学期不能将所有的算法教给我们，但是为我们以后的深入学习指明了道路。

感谢助教与技术组同学的辛苦付出，在我们编写算法的过程中解决了很多问题。

感谢各位组员的努力，祝大家肝好肾好身体好成绩好！

6 参考文献

单人 2048 算法集合（单调性、平滑性启发）：
https://stackoverflow.com/questions/22342854/what-is-the-optimal-algorithm-for-the-game-2048
蒙特卡洛算法：
https://github.com/ronzil/2048-ai-cpp
下棋算法：
https://aj-r.github.io/Evil-2048/
http://aj-r.github.io/2048-AI/
本组 gitee 库：
https://gitee.com/master_2048/master_2048

数据结构与算法课程实习作业报告

王奕力*　吴伟霖 徐潇彤 张俊龙 赵月悦

摘要：二零四八属于完全信息的双人同步零和博弈问题。因此，我们组的算法的基本框架是构建决策树，并进行 AlphaBeta 剪枝，采用估值函数和 minmax 算法做出决策，同时聚焦于对决策树和估值函数的优化。该算法在搜索深度和估值函数之间形成了较好的平衡，取得了较好的成果，进入了 F19 组的八强，但组外有组，仍然有优化空间。

关键字：决策树 AlphaBeta 剪枝 估值函数 minmax 算法 启发式算法

1 算法思想.....1
1.1 总体思路.....1
1.2 算法流程图.....2
1.3 算法运行时间复杂度分析.....7
2 程序代码说明.....7
2.1 数据结构说明.....7
2.2 函数说明.....7
2.3 程序限制.....12
3 实验结果.....12
3.1 测试数据.....12
3.2 结果分析.....13
3.3 经典战局（可选）.....14
4 实习过程总结.....16
4.1 分工与合作.....16
4.2 经验与教训.....18
4.3 建议与设想.....18
5 致谢.....18
6 参考文献.....19

1 算法思想

1.1 总体思路

我们组算法的总体思路是从每人一份的算法草案中筛选、提炼出来的：每一次调用都会以当前局面为根节点，并对可能操作进行遍历，通过深度优先递归搜索构建一棵决策树，同时进行 AlphaBeta 剪枝。每一个节点都通过调用估值函数得到一个局面分数，以叶节点分数为依据，通过 minmax 算法做出决策。

在基本的“构建决策树-进行 AlphaBeta 剪枝-采用估值函数和 minmax 算法做出决策”

的框架下，我们主要聚焦于决策树的优化加深和估值函数的改进。

在时间允许的范围内，搜索层数越深，ai 也就越“聪明”，但同时耗费的时间也是指数上涨的，所以如何在尽可能少的时间搜索更深的层数就成了优化的关键问题之一，进一步思考，如何“把时间用在刀刃上”，在关键处扩大搜索范围、加深搜索层次，在平凡处减少时间耗费，也成为重要的思考方向。

而估值函数，则是一个局面好坏的评判标准。由于估值函数被调用的次数随着层数增加呈指数增长，所以估值函数的“性价比”，或者说，“耗时-性能比”，就至关重要：性价比低的估值函数，耗时长，对局面判断的帮助小，如果去掉并用省下的时间相应加深搜索层数，实力反而提升。

在以上认识的基础上，我们的算法经历了大概以下几个大版本的变化：

1、初代的版本里，囊括了决策树、剪枝、估值函数等必要组成部分，估值函数包括绝对分数、相对分数（对方-己方）、平滑度、单调性、空格数、按位置加权分数、相对位置函数、死局判断。（但经过测试发现，除了绝对分数和相对分数，这些初代函数大多数性价比较低，后来都进行了删改和优化。）

2、发现经常下在对面白给的问题，同时，搜索在对面下子的可能性过多，耗费大量时间，因此在对面空格数较多（>3）且己方有空可下时，不搜索在对面落子，大幅度节约了时间成本。

3、针对超时问题设置了剩余时间判断，当剩余时间较小时减少搜索层数，基本解决了超时问题。

4、为了保证在不超时的情况下充分利用时间和减少时间浪费，搜索的层数由当前回合数、剩余时间、先后手以及当前模式决定。（如图 1.1.1）

5、为了解决其他估值函数与棋子级别得分的数量级随着回合数增加差距拉大，逐渐无效的情况，其他各项估值函数分数的权重中包含绝对分数，使数量级始终相近。

6、通过启发式算法，优化各个估值函数，大规模优化修改后，最终证明性价比较高并得到采用的估值函数有：绝对分数、相对分数、空格数、缩角、大数保护、单调性、平滑性、边界防守、死局……并且通过观察对局在在开局、中盘和收官时的不同特征，根据重要性在不同阶段分别选用不同的估值函数和权重组合。

7、发现大数被吃问题，为了保护大数，在父节点和子节点之间出现估值函数骤降时（表示己方大数被吃，并且不能通过移动或在己方下子避免），扩大搜索范围，尝试在对面落子堵住对方的路，以保护己方大数。

当然，大版本的飞跃是靠中间无数个小版本积累起来的，在此就不一一赘述了。

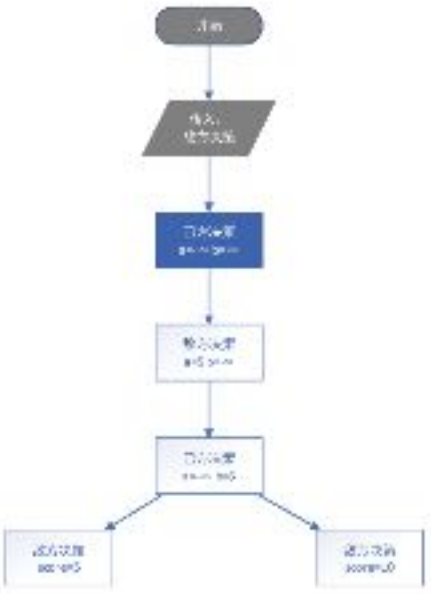
1.2 算法流程图

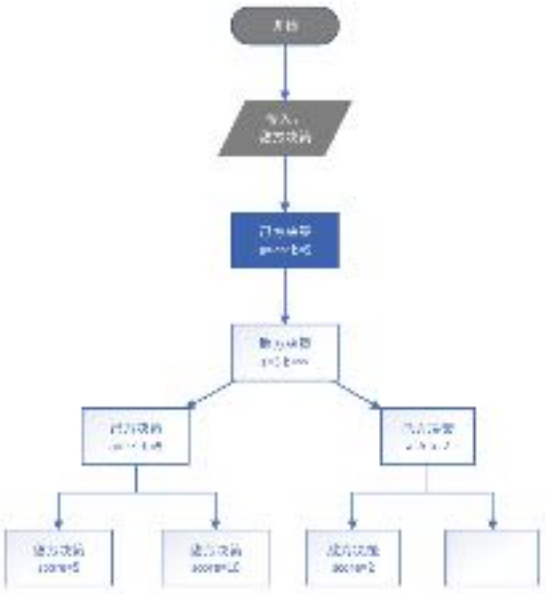
我们的算法主要有树和估值函数两部分，其中估值函数部分我们将在函数说明中详细解释，在此主要介绍决策树部分的流程。

现在以搜索四层为例进行说明：

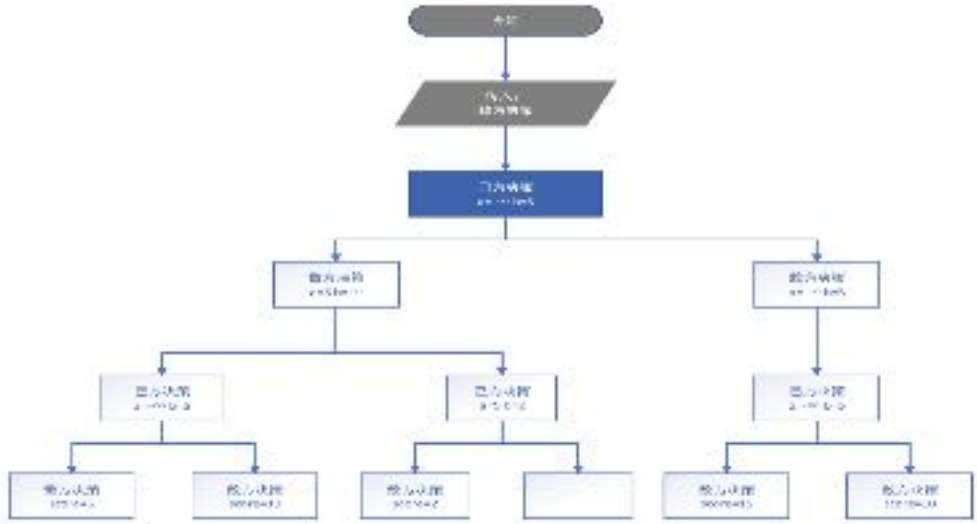
如图：

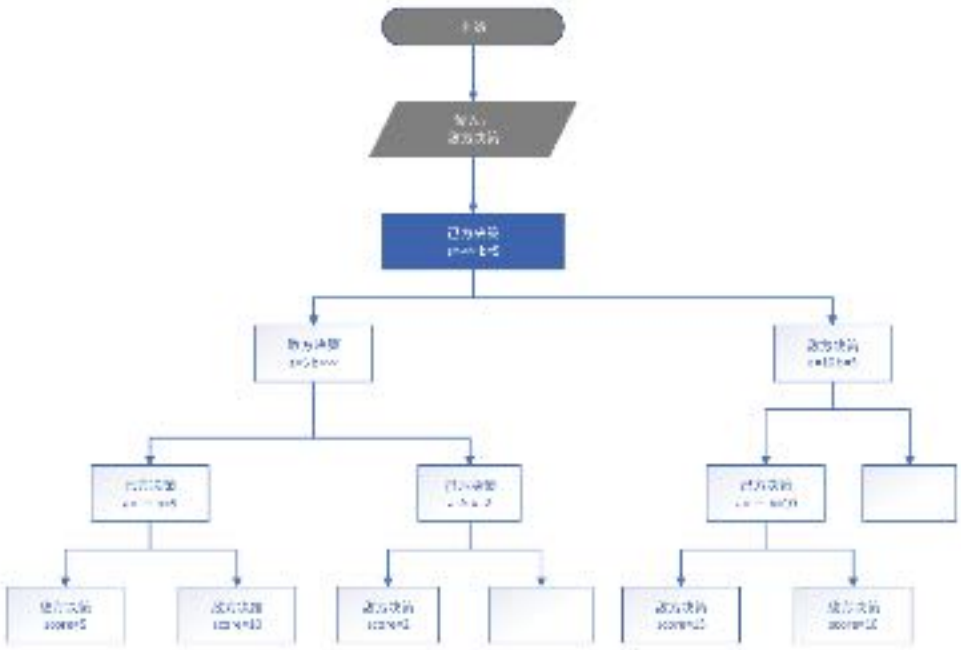
敌方（第四层）尽量使得己方获利最小，因此当遇到使得己方获利更小的情况，则需要修改β。这里 5 小于正无穷大，所以己方（第三层）β修改为 5；10 大于 5，不用修改β。





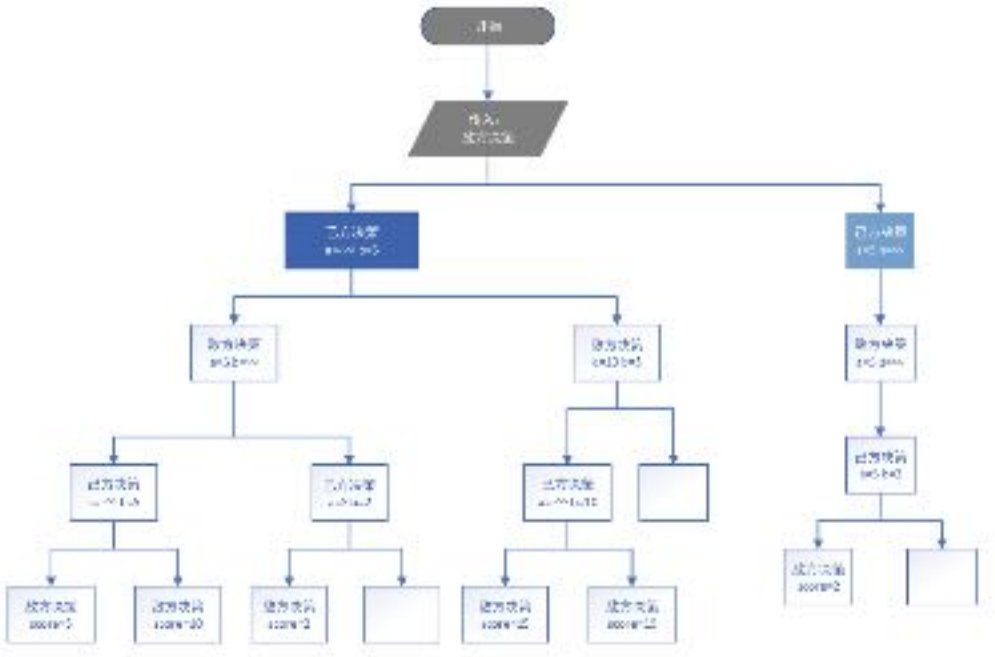
己方（第三层）尽量使得己方获利最大，因此当遇到使得己方获利更大的情况，则需要修改 α 。这里 5 大于负无穷大，所以敌方（第二层） α 修改为 5；敌方（第四层）拥有一个方案使得己方获利只有 2，α =5，β =2， α > β，说明己方（第三层）只要选择第二个方案，则敌方必然可以使得己方的获利少于己方（第三层）的第一个方案,这样就不再需要考虑敌方(第四层)的其他候选方案了,因为己方（第三层）根本不会选取第二个方案。



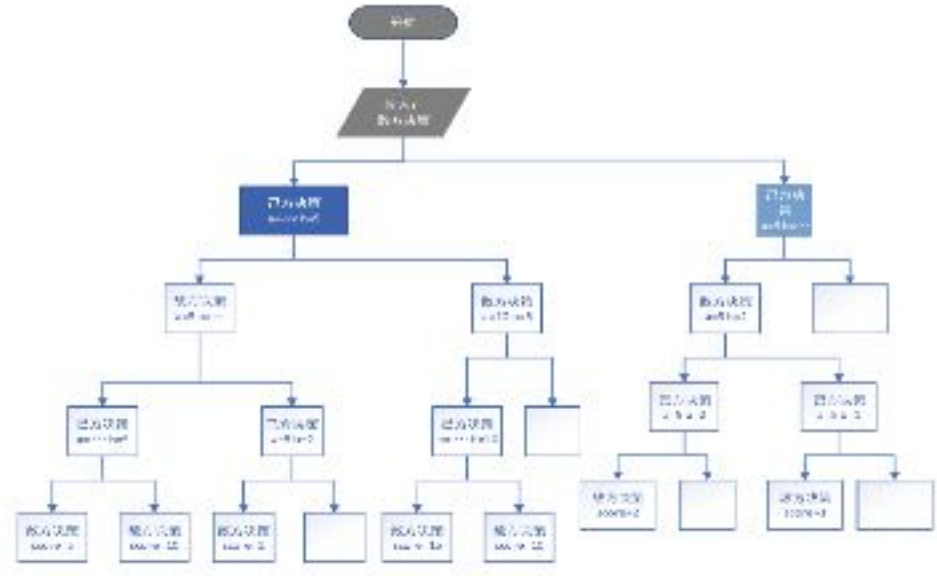


敌方（第二层）要使得己方利益最小,则敌方（第二层）的第二个方案不能使得己方的获利大于 β ，也就是 5。但是若敌方(第二层)选择第二个方案，己方（第三层）可以选择第一

个方案使得己方获利为 10, α =15, β =5, α > β , 故不需要再考虑己方(第三层)的第二个方案, 因为 B(第二层)不会选择第二个方案。

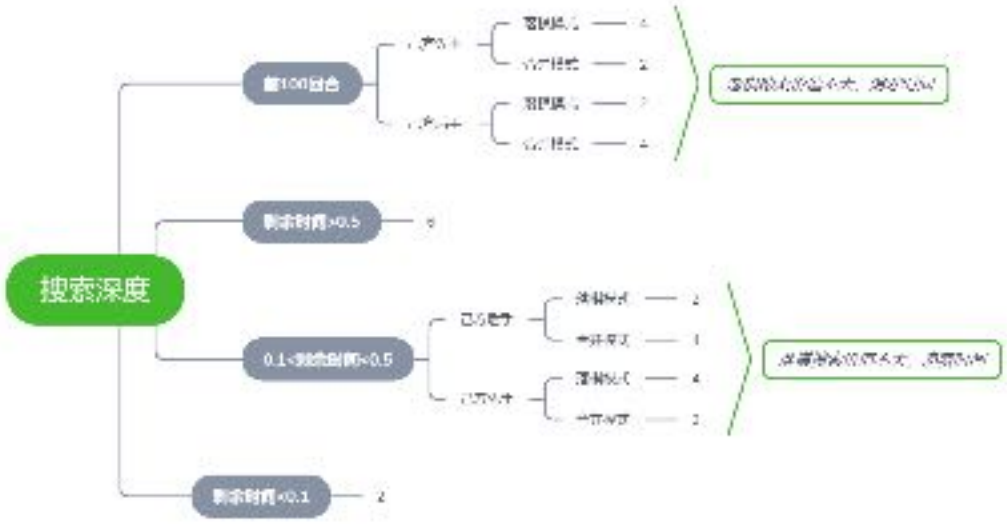


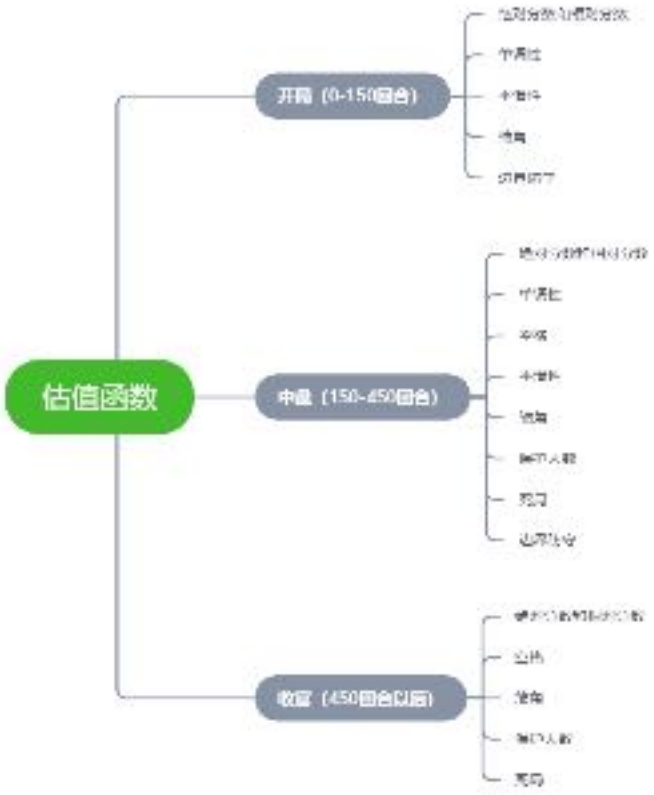
己方（第一层）使自己利益最大,也就是己方（第一层）的第二个方案不能差于第一个方案, 但是己方（第三层）的一个方案会导致利益最多为 2, 小于 5, 所以己方（第三层）不会选择第一个方案, 因此敌方（第二层）也不再搜索第二个方案。



敌方（第二层）要使得己方利益最小,则敌方（第二层）的第一个方案使得己方的获利最多为 1。所以己方（第一层）不会选择第二个方案, 因此敌方（第二层）也不再搜索第二个方案。

另外附上搜索深度判断和估值函数选用的参考图：





1.3 算法运行时间复杂度分析

我们算法的主要时间开销在于决策树节点对于估值函数的调用，如果以搜索深度作为数据规模，不考虑剪枝的情况下，由于每增加加深一层节点数翻倍，可以理解为对层数最差是O（2^n）。但剪枝大大降低节约了时间，实际时间视情况而定。

在估值函数中，以棋盘大小为数据规模，有些时间复杂度为O（n）,有些为O（1），总体复杂度在O（n）左右。

2 程序代码说明

2.1 数据结构说明

主要数据结构：

树，非线性结构，线性结构。

自定义类：

节点类（Node），搜索树类（SearchTree），玩家AI类（Player），估值函数类（Score）

扩展和改进：

保留了树的基本结构：父节点、子节点、节点、根、叶等特性。但由二叉树扩展为多叉，同时配合了剪枝。

2.2 函数说明

首先，将程序用到的所有函数、类型、接口、功能等，归纳成表如下：

其中，我们选取较为重要的函数进行进一步展示分析：

```
def allscore(self):
    '''
    最终分数
    在不同阶段采取不同计算方式
    分数包含：1.己方棋子级别得分，2.双方分差，3.各项对局评估的函数
    各项函数的权重中包含己方得分myscore，使函数得分的数量级与myscore相近
    '''
    leftround = 500 - self.node.current_round
    belong = self.node.belong
    mode = self.node.mode
    board = self.node.board
    finalscore = 0
    space = {self.isFirst: board.getNone(self.isFirst),\
             not self.isFirst: board.getNone(not self.isFirst)}  # 双方空位
    vallist = {self.isFirst: board.getScore(self.isFirst),\
               not self.isFirst: board.getScore(not self.isFirst)}  # 双方棋子
    chess = self.getAll(board, self.isFirst)  # 己方所有棋子位置
    myscore = self.value_absoluteScore(vallist[self.isFirst])  # 己方绝对分数
    opscore = self.value_absoluteScore(vallist[not self.isFirst])  # 对方绝对分数
    relativescore = myscore - opscore  # 相对分数
    q = opscore if relativescore < 0 else myscore  # 分数权重，加强相对分数变动的影响
    finalscore += myscore  # 绝对分数
    finalscore += relativescore * q  # 相对分数

    if leftround < 50:  # 450到合法后
        finalscore += self.value_blank(board, space[self.isFirst]) * 0.3 * myscore  # 空格
        finalscore += self.value_corner(board, vallist[self.isFirst]) * myscore * 0.3  # 角数
        finalscore += self.value_protect(board, vallist[self.isFirst], chess, self.isFirst) * myscore
        if not space[self.isFirst]:  # 己方无空位，可能出现死局情况
            finalscore -= self.value_deadgame(board, self.isFirst) * 1 * myscore
        if not space[not self.isFirst]:  # 对方无空位，可能出现死局情况
            finalscore += self.value_deadgame(board, not self.isFirst) * 1 * myscore

    elif leftround < 350:  # 150-450阶段
        if vallist[self.isFirst] and max(vallist[self.isFirst]) <= 6:
            finalscore += self.value_mono(board, space[self.isFirst], chess,
                                          space[self.isFirst]) * myscore * 0.5  # 单调性
        finalscore += self.value_blank(board, space[self.isFirst]) * 0.3 * myscore  # 空格
        finalscore += self.value_smooth(board, space[self.isFirst], chess) * 0.3 * myscore  # 平滑性
        finalscore += self.value_corner(board, vallist[self.isFirst]) * myscore * 0.3  # 角数
        finalscore += self.value_protect(board, vallist[self.isFirst], chess, self.isFirst) * myscore
        if not space[self.isFirst]:  # 己方无空位，可能出现死局情况
            finalscore -= self.value_deadgame(board, self.isFirst) * 1 * myscore
        if not space[not self.isFirst]:  # 对方无空位，可能出现死局情况
            finalscore += self.value_deadgame(board, not self.isFirst) * 1 * myscore
        finalscore -= self.value_border(board, self.isFirst) * myscore * 0.1  # 边界防守
        finalscore += self.value_border(board, not self.isFirst) * myscore * 0.1

    else:  # 0-150阶段
        if vallist[self.isFirst] and max(vallist[self.isFirst]) <= 6:
            finalscore += self.value_mono(board, space[self.isFirst], chess,
                                          space[self.isFirst]) * myscore * 0.5  # 单调性
        finalscore += self.value_smooth(board, space[self.isFirst], chess) * 0.3 * myscore  # 平滑性
        finalscore += self.value_corner(board, vallist[self.isFirst]) * myscore * 0.3  # 角数
        finalscore -= self.value_border(board, self.isFirst) * myscore * 0.1  # 边界防守
        finalscore += self.value_border(board, not self.isFirst) * myscore * 0.1
```

Allscore 函数主要用于加权得出最终分数。由于在早些版本的测试分析中，发现一些前期被侵蚀、后期易塞爆的问题。

我们受到棋类游戏中开局、中盘和收官概念的启发，结合二零四八的特点，根据我们的经验、计算和测试，将整局分以 150 和 450 为界分为三个大阶段。

前期不考虑死局、塞爆和空格数，减少计算量，并且注重边界防守，避免落入劣势。中期估值函数最为复杂，考虑因素最多。后期主要注重于死局、塞爆和空格，同时利用缩角进行大数保护。

```
504     def value_absoluteScore(self, vallist):
585         '''
586         绝对分数
587         '''
588         newlst = [2 ** (i * 1.2) for i in vallist]
589         score = sum(newlst)
590         return score / 10
591
```

我们实验了很多版本的绝对分数函数，有分数直接相加，有 10 ** (i - 1)再对数压缩，有 2 的幂次等等，在实验范围内，这个函数能较好地平衡大数和总分数。

```
714     def value_corner(self, board, vallist):
715         '''
716         大数获得：当最大数大于32时，基本还需双方边角的角上的部分（加权的系数分）；
717         在没两双方边角的加上部分（上对应幂次的开方分）
718         '''
719         maxL = vallist[-1] if len(set(vallist)) > 5 else 0
720         if maxL != 0:
721             LOR = 0 if self.isFirst else 7
722             if board.getValue((0, LOR)) == maxL or board.getValue((3, LOR)) == maxL:
723                 return maxL
724             elif board.getValue((1, LOR)) == maxL or board.getValue((2, LOR)) == maxL:
725                 return maxL ** 0.5
726         return 0
```

```
728     def value_protect(self, board, vallist, chess, belong):
729         '''
730         保护大数
731         大数在边界上时，必须为它预留可移动的空间，以为被对方堵死时，上前移动
732         '''
733         if not vallist or max(vallist) < 6:
734             return 0
735         score = 0
736         poslist = self.getPosition(board, max(vallist), belong)
737         if belong:
738             for pos in poslist:
739                 if pos[1] < 3 or (pos[0], pos[1] + 1) in chess:  # 不在边界上;或在边界上，后方拥有己方棋子作
740                     break
741                 else:  # 在边界上，且无棋子保护
742                     rowlist, collist = [], []
743                     for i in range(3):
744                         c = board.getValue((pos[0], i))
745                         if c == 0:  # 此行有空位，可移动
746                             break
747                         else:
748                             rowlist.append(c)
749                             if len(rowlist) > 1:
750                                 if c == rowlist[i - 1]:  # 有可合并对象,可移动
751                                     break
752 
753                     else:  # 该行堵满，无法移动
754                         for i in range(4):
755                             c = board.getValue((i, pos[1]))
756                             if c == 0:  # 此行有空位，可移动
757                                 break
758                             else:
759                                 collist.append(c)
760                                 if len(collist) > 1:
761                                     if c == collist[i - 1]:  # 有可合并对象,可移动
762                                         break
763                         else:  # 行列均堵
764                             score -= board.getValue(pos)
765         else:
766             for pos in poslist:
767                 if pos[1] > 4 or (pos[0], pos[1] - 1) in chess:  # 不在边界上;或在边界上，后方拥有己方棋子作
768                     break
769                 else:  # 在边界上，且无棋子保护
770                     rowlist, collist = [], []
```

测试中发现大数位于边界线上容易在后期被吃导致暴毙的情况，因此增加了 value_corner 函数和 value_protect 函数，当最大数靠角落的时候给予额外加分进行鼓励，且在边界上尽量给大数预留逃跑空间。

```
795     def value_border(self, board, belong):
796         '''
797         边界保卫，防止边界位置被对方蚕食
798         '''
799         b = 3 if belong else 4
800         count = 0
801         for i in range(4):
802             if board.getBelong((i, b)) != belong:
803                 count += 1
804         return count
```

针对边界容易被对方蚕食的问题，开发 value_border 函数，保卫边界。

除此之外，还有不少函数，在此不再一一赘述。

2.3 程序限制

该程序在极少数情况下（某场对局后期过于复杂时）可能出现略微超时，绝大多数情况下不会出错。

该算法在后手时效果不如先手，还有待优化。

3 实验结果

3.1 测试数据

实验环境说明：

- 硬件配置：（CPU/内存）:i5
- 操作系统：（名称/版本）:windows10
- Python 版本：（版本号）:python3.8

一、测试方法不完全列举：

1、时间检测

估值函数里 allscore 每个函数运行一万次，打印时间；在本地对局中，给 player 增加属性 i，用于每次调用估值函数时进行记数，统计一局对局中估值函数的调用次数。

2、估值函数的参数调整与问题发现：

通过在本地复盘当时局面，打印出决策过程中搜索到的盘面以及该盘面对应的各项分数，发现运行结果与预期不同/权重参数不合理的地方，并针对该实例进行调整（通过实例具体分析，而不是盲目地东调调西调调然后观察微弱的胜率差）；

3、评估搜索层数和估值函数的耗时与收益关系

Ps:测试代码举例：（大部分测试函数已经遗失）

```
def buildboard(self,f):
    boardlist = [None for i in range(4)]
    with open(f,'r') as t:
        for i in range(4):
            boardlist[i] = str(t.readline()).split(' ')
    for i in range(4):
        for j in range(8):
            if boardlist[i][j][0] == '+':
                if int(boardlist[i][j][1:]) != 0:
                    self.belongs['True'].append((i,j))
                    self.board[(i,j)] = Chesswar(True, (i,j), int(boardlist[i][j][1:]))
            else:
                if int(boardlist[i][j][1:]) != 0:
                    self.belongs['False'].append((i,j))
                    self.board[(i,j)] = Chesswar(False, (i,j), int(boardlist[i][j][1:]))
```

二、测试结果

部分测试数据及结果如下表所示：

T1	P2	100	175	57.00%
T1	2048Bot_B	100	46	46.00%
T2	2048Bot_B	200	141	70.50%
T1'	T1	100	81	81.00%
T0	N3	90	28	33.33%
傘兵一号	T0	100	99	99.00%
T2	2048Bot_B	100	96	96.00%
N3	T0	200	101	50.50%
D0	N3	100	84	84.00%
傘兵一号	2048Bot_A	100	83	83.00%
Player3	defend	50	31	62.00%
new2	new1	40	27	67.50%
P2	P1	80	34	42.50%
P3	P1	200	157	78.50%
P3	W3	200	77	38.50%
T1	T0	200	153	76.50%
T0	Tj	20	20	100.00%
T0	T2	100	85	85.00%
傘兵二号	傘兵四号	50	36	72.00%
D1	D0	50	29	58.00%
D1	2048Bot_S3	50	21	42.00%
defend	D1	55	32	58.18%
D3	2049Bot_S1	100	40	40.00%
defend	2050Bot_S1	200	106	53.00%

败场分析1	场数	比例
大数被吃	48	48.00%
微弱劣势	27	27.00%
塞爆>100回合	4	4.00%
未塞爆惨败	21	21.00%
总场数	100	100.00%

败场分析2	场数	比例
大数被吃	37	37.00%
微弱劣势	32	32.00%
塞爆>100回合	7	7.00%
未塞爆惨败	24	24.00%
总场数	100	100.00%

败场分析3	场数	比例
大数被吃	21	21.00%
微弱劣势	43	43.00%
塞爆>100回合	9	9.00%
未塞爆惨败	27	27.00%
总场数	100	100.00%

败场分析4	场数	比例
大数被吃	9	9.00%
微弱劣势	78	78.00%
塞爆>100回合	4	4.00%
未塞爆惨败	9	9.00%
总场数	100	100.00%

3.2 结果分析

由于加入了大数靠角 value_corner、大数保护 value_protect、边界守卫 value_border 函数，我们组的 AI 在对局中能够较好地保护自身，很少出现大数被吃暴毙的现象。

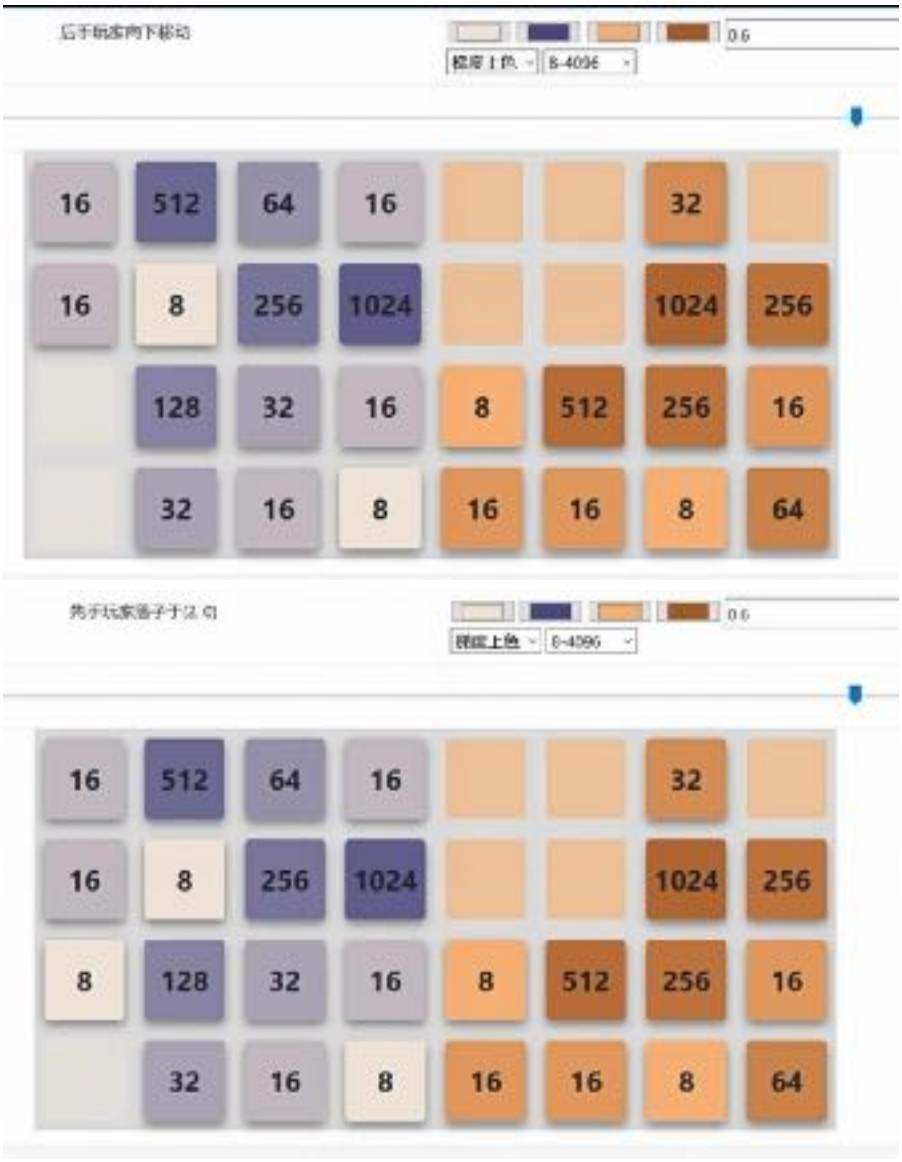
同时，由于我们加入了根据剩余时间决定搜索层数的策略，我们的 AI 时间把控往往很

好，极少出现超时，且很好地利用了给定的时间进行较深的搜索。同时，搜索层数根据条件判断节约了大笔时间用于关键的地方。

前中后期估值函数的不同让 AI 在不同时期展现出不同的战略：前期减少计算量，并且注重边界防守，避免落入劣势。后期注重于死局、塞爆和空格，同时利用缩角进行大数保护。

同时，我们的 AI 也有不足之处，例如，缺少进攻性的估值函数导致进攻性不足，常常只在边界线试探。再者，合并情况还不够理想，有时出现几个较大数相隔较远很长时间难以合并的现象。

3.3 经典战局（可选）



16 512 64 16 32
16 8 256 1024 1024 256
8 128 32 16 8 512 256 16
8 32 16 8 16 16 8 64

32 512 64 16 32
16 8 256 1024 1024 256
128 32 16 8 512 256 16
32 16 8 16 16 8 64

32 512 64 16 32
16 8 256 2048 256
128 32 16 8 512 256 16
32 16 8 32 8 64

4 实习过程总结

4.1 分工与合作

分工：

1、在确定算法方向前，每个人都各自搜集资料，拟写一份算法草案。在 5 月 18 号晚上召开第一次小组会议，针对五份算法草案进行讨论，最终筛选提炼出算法思路。

2、代码编写和优化阶段分为两组。其中吴伟霖、张俊龙、王奕力参加估值函数组，负责编写和优化估值函数，徐潇彤、赵月悦参与决策树组，负责决策树编写和优化。

3、实习报告撰写和代码各版本之间的测试由王奕力负责。

合作：

主要通过微信群和腾讯会议进行交流。

完整组会共有四次：

1、初步分工和时间规划；

2、草案讨论和提炼；

3、完整代码修 bug 和优化方向确定；

4、测试和 ddl 前讨论会。

除此之外，微信群里常常在进行讨论。

4.2 经验与教训

大家都积极参与讨论，及时完成任务，有疑问能很快得到解答，有问题也能很快解决，小组气氛融洽，每一个人都能自由发表自己的见解。由于组员积极和分工明确，每一个问题都能很快落到实处，没有出现互相推卸。美中不足的是，我们的进程开始较晚，前期进度较慢，导致后期较肝，时间安排还可以更加合理。

从算法的层面上说，我们一开始的估值函数走了不少弯路。第一次写的估值函数几乎都是性价比较低，占用时间资源高于自身的效益，导致第一轮估值函数几乎作废。这一点是我们在规划的时候没有想到的，导致了时间的浪费。好在大家没有气馁，都积极地去解决问题，写出了新的优质的估值函数。

4.3 建议与设想

这次的大作业做下来的确收获不少，也很有趣味性，但也有美中不足之处：

对抗性略微欠缺。对于大部分组来说，二零四八的对抗面较窄，仅在边界线处时不时发生接触战。且在对面下子这一设定没有发挥充分的作用，一是会大大增加搜索量，二是大部分情况都是白给。

算法多样性略微不足。除了炼丹，大部分组都是决策树+估值函数的算法，区别更多在于搜索深度和估值函数的优化，多样性空间较小，新奇感略有不足。

明年的学弟学妹们，大作业真的很有趣也有意义，小组讨论到凌晨的记忆，一起肝 DDL 的回忆，都极其珍贵。

5 致谢

感谢老师和助教的陪伴和耐心解答，也感谢他们为我们准备了这样一个有趣的舞台！

感谢技术组的辛苦开发，让我们能够享受到大作业的魅力！
感谢组员们的积极参与和互相帮助！

6 参考文献

各种算法相关资料
https://blog.csdn.net/zkybeck_ck/article/details/45645031
https://www.zhihu.com/question/23029850
https://www.zhihu.com/question/23006903/answer/23367814
http://ovolve.github.io/2048-AI/
https://www.bilibili.com/video/BV1gi4y187EL?from=search&seid=10294946636915829809
https://blog.csdn.net/illikang/article/details/82019945

数据结构与算法课程实习作业报告

曹珂* 陈鹏 杜泽宇 何承芝 张子羲

摘要：本算法通过决策树和估值函数从可能的落子方案中选取最佳的一个，并利用启发式规则提高在决策树上搜索的效率。最终受限于算法效率，算法短时间内搜索的决策树深度不够，棋力属于同中等水平。

关键字：决策树 剪枝 树

1 算法思想.....2
1.1 总体思路.....2
1.2 算法流程图.....3
1.2.1 output 正式决策.....3
1.2.2 position 模式决策.....4
1.2.3 direction 模式决策.....5
1.2.4 决策树流程.....6
1.3 算法运行时间复杂度分析.....6
2 程序代码说明.....7
2.1 数据结构说明.....7
2.2 函数说明.....7
2.3 程序限制.....12
3 实验结果.....13
3.1 测试数据.....13
3.2 结果分析.....14
3.2.1 天梯赛结果：.....14
3.2.2 热身赛战绩.....14
4 实习过程总结.....15
4.1 分工与合作.....15
4.1.1 小组分工：.....15
4.1.2 小组交流.....15
4.1.3 时间节点.....15
4.2 经验与教训.....17
4.3 建议与设想.....18
5 致谢.....18
6 参考文献.....18

1 算法思想

1.1 总体思路

算法的总体思路是通过递归函数建立博弈树，估值函数的估值让大子子值占主导部分，对方权重高于己方；利用先手侧重进攻后手侧重防御的启发性规则结合 alpha-beta 剪枝增加博弈树深度，使之考虑得更加长远；必要时考虑“手动”决策节约时间资源。

算法的确定过程：

估值函数：

【大子优先】一开始的估值函数考虑了棋子的分布，导致代码复杂且参数繁多，还占用大量时间资源；另外，ai 有些情况下竟然忽视对大子的合并。受博弈树终局判断胜负方式的启发，我们采用幂函数将每个子的子值换算为实际分数，当幂函数的阶数 N 趋向于无穷的时候，估值函数趋向于从大到小比较每一个子值（val）的离散函数，这就实现了对终局判胜负方式的模拟。实验表明，这方法确实使 ai 非常重视合并，步步逼近胜局。后来试着换用幂指函数和阶乘函数，原理没有改变，只是重视程度有所变化。

【积极攘外】实战中，我们发现 ai 不能很好的处理边界问题，经常放弃可行的进攻，很保守，容易被对方攻入。为了提高消灭对方与避免对方吞噬的积极性，我们试着加重对方总分数的比重（原先是我方换算子值求和减去对方的，权重都为一）。仅将对方总分权重上调为 2 时，边界问题就已基本上得到妥善处理。

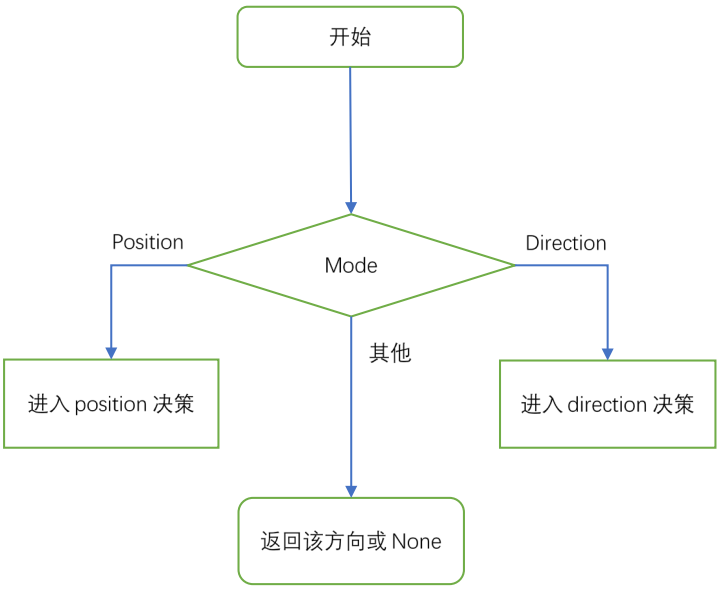
“手动”落子：

【先手辅助进攻，后手避开进攻】由于剪枝效率迟迟没有提高，我们对落子这一主要分支处下手，直接手动判断，先手若可辅助进攻则落子对方对应处，后手在不受对方进攻威胁时选择落子己方。实践表明，这样的“手动”落子方式基本上是“明智”的，是可以基本代替 ai 决策的，而且使博弈树加深了一倍。

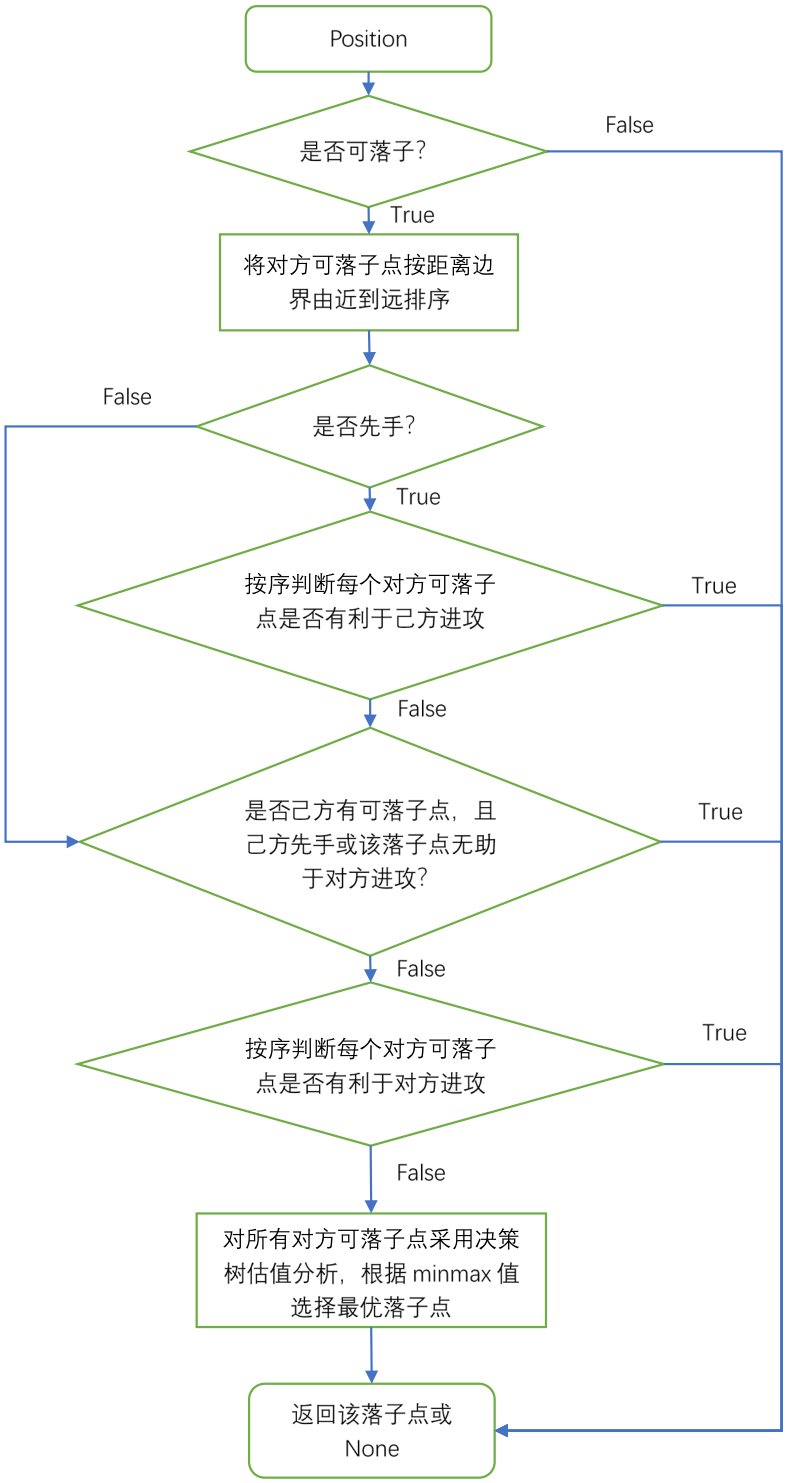
最后，方向由 ai 搜索决定。因为我们并没有想出比较高效可行的启发式规则，尝试过模拟大佬常用的塞爆，但没成功。

1.2 算法流程图

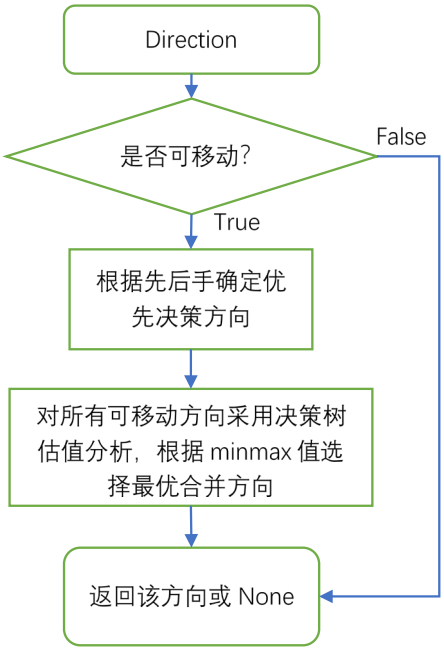
1.2.1 output 正式决策



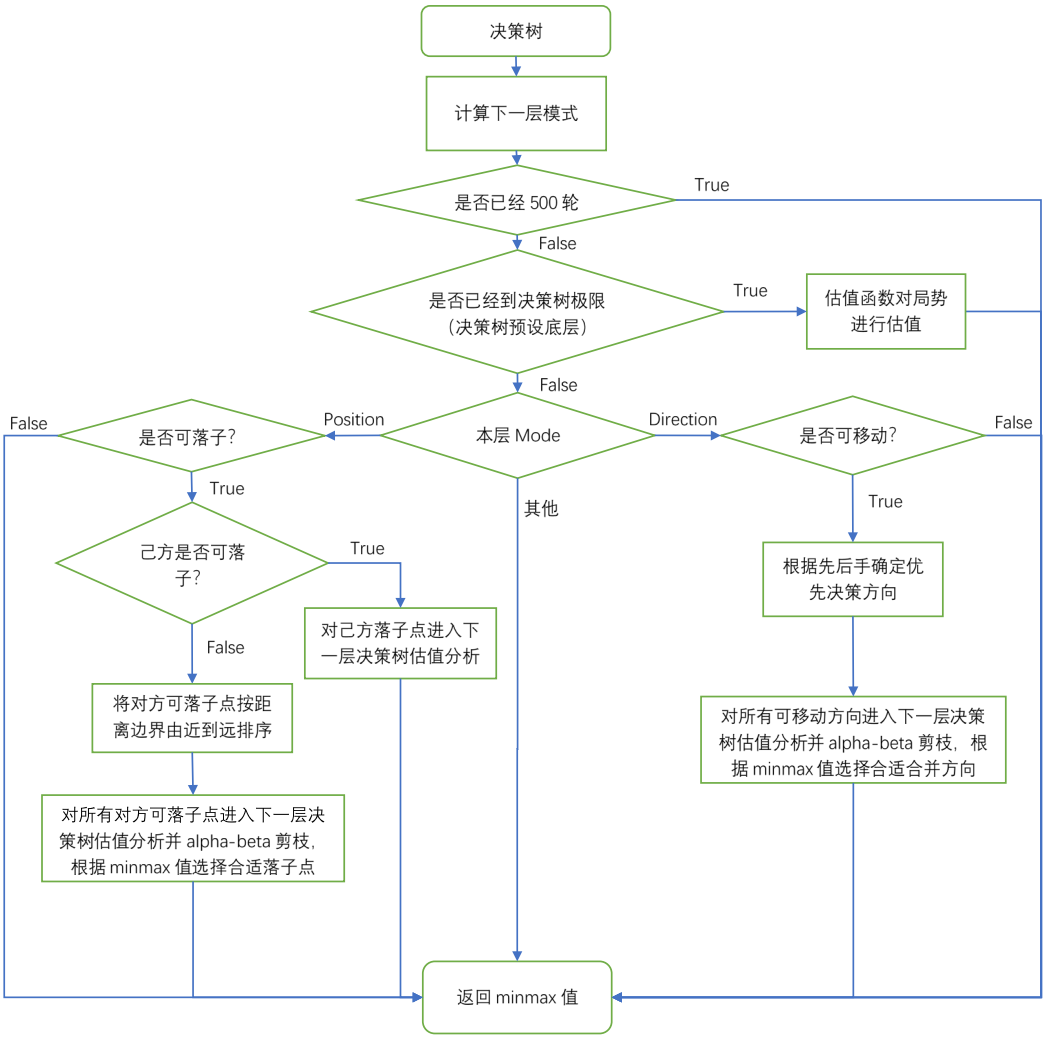
1.2.2 position 模式决策



1.2.3 direction 模式决策



1.2.4 决策树流程



1.3 算法运行时间复杂度分析

设算法中决策树的深度为 h，当前最大棋子分数为 s，当前对方空位数为 n。

算法对应问题的数据规模主要是 h。

我们的算法在整个决策过程中，除了搜索到比赛结束（第 500 轮时）之外，叶节点全部为估值函数，而估值函数为当前所有棋子分数的阶乘之和，其大 O 数量级为 O(s)，由于 s<14，故可忽略。

该算法时间复杂度主要由决策树决定。

在决策树中，平均 direction 模式与 position 模式的层数各占一半。

其中每个 direction 模式的节点，无论是根节点还是非根节点，平均而言 4 个方向均会产生分支。

position 模式的非根节点，平均情况下己方一般是有空位可下的，而且一般情况分数比

下在对方空位高很多，由于特殊的剪枝算法，此时所有对方空位几乎全部被剪枝，因此等效为只产生一个分支。

而在较坏的情况下，己方已无空位可下，在较多轮中对方平均空位个数为 8，此时会产生 8 个分支。

对于以 position 模式作为根节点，平均情况下将直接判断可下位置，不进入决策树。在判断的过程中，排序过程的时间复杂度为 O(nlogn)，can_attack、can_help_attack 相关的计算时间复杂度均为 O(n)，因此该步骤时间复杂度的大 O 数量级为 O($nlogn$)。由于 n<16,属于常数量级，和博弈树一个节点的运算时间相差无几。可以认为是构建了了 h = 1 的博弈树。

较坏的情况下，无法直接做出判断时，和非根节点一样平均将产生 8 个分支。

由上面的分析可知，在平均情况下，以 direction 模式作为根节点时的算法时间复杂度为$4^{h/2}$，即大 O 数量级为 O(2^h)；以 position 模式作为根节点时的算法时间复杂度为1，即大 O 数量级为 O(1)。

在较坏情况下，无论是以 direction 模式还是 position 模式作为根节点时，算法时间复杂度都为$8^{h/2}4^{h/2}$，即大 O 数量级为 O($(4\sqrt{2})^h$)。

2 程序代码说明

2.1 数据结构说明

算法中主要采用了线性表、用于决策的树这两种数据结构。

线性表主要用于储存数据以及对数据进行排序，使数据便于运算。

树主要用于通过递归构建决策树，叶节点用于估值函数，其他非根节点计算并向上传递 minmax 值，根节点选择 minmax 值最大的分支作为最终决策。其中对节点进行了 alpha-beta 剪枝以及根据其他启发式规则（对己方落子点和对方落子点的直接判断）的剪枝改进，大大减少了总叶节点的数量，提高了决策树的效率。

我们的算法中未自定义类。

2.2 函数说明

```
1. ab_eval(isFirst: bool, h: int, board: Chessboard, player: bool,
   currentRound, mode: str, a: int, b: int) -> int
```

参数：isFirst 己方， h 当前决策树高度（以叶节点为 1 层），player 当前决策方，a、b 为 minmax 辅助计算值

返回：该节点的 minmax 值

功能及算法策略：算法的核心部分，构建包含 alpha-beta 剪枝的决策树节点，包括 position 模式和 direction 模式，分别对所有的可行方案进行搜索。其中，position 模式额外含有落子己方优先的剪枝策略。h=1 时为叶子节点，将调用估值函数。

内包函数 winner(board)：当搜索到游戏结束时，调用 winner 函数判断胜负直接判值，以作为叶节点 minmax 值。返回胜方。

```
def ab_eval(isFirst,h,board,player,currentRound,mode,a,b):
    def winner(board):
        # 获取所有棋子并计数
        results = {True: board.getScore(True), False: board.getScore(False)}

        scores = {True: {}, False: {}}
        for level in range(1, 14):#14是设定最大的价值
            scores[True][level] = results[True].count(level)
            scores[False][level] = results[False].count(level)

        # 比较比分
        for _ in reversed(range(1, 14)):
            if scores[True][_] != scores[False][_]:
                winner = scores[True][_] > scores[False][_]
                break
        else:#特殊用法，和break的for配对
            winner = None #平手

        return winner
    #找到下一步玩家、模式
    do = [(True,'position'),(False,'position'),(True,'direction'),(False,'direction')]
    do_dict = {(True,'position'):0,(False,'position'):1,(True,'direction'):2,(False,'direction'):3}
    nextorder = ( do_dict[(player,mode)] + 1 )%4
    nextRound = currentRound
    next_h = h - 1
    if nextorder == 0:#进入下一轮
        nextRound = currentRound + 1
    nextplayer , nextmode = do[nextorder]

    #最小结束条件
    if currentRound == 500:#500时已结束，不能下
        '''''好像是500,不清楚'''
        if winner(board) == isFirst:#我方判胜
            return 10**105              #目前maxsize定为10**100
        elif winner(board) != isFirst:#对方胜
            return -10**105
        else:#平局
            return 0
            '''''感觉0还不是很科学'''

    if h == 1:#没结束但到了搜索底层
        return easy_eval(isFirst,board)
```

```
        '''''此处考量两种估值方式'''
    #可模拟下一步
    if mode == 'position':
        #所有可下的位置:己方随机位置及对方空位
        avils = board.getNone(not player)
        avils.sort(key = lambda x:x[1] , reverse = False if player else True ) #按列 col 排，如果我方先手则小 col 靠内，正序；反之逆序
        #启发式规则：棋局比较空的时候下自己
        my = board.getNext(player, currentRound)
        if my:
            avils.insert(0,my)
        '''''此处涉及启发性思维，先考虑自己的子，还是对方的子'''
        for avil in avils:#尝试每一步
            nextboard = board.copy()
            nextboard.add(player, avil)#更新棋盘
            evl = ab_eval(isFirst,next_h,nextboard,nextplayer,nextRound,nextmode,a,b)#继续估值
            if player == isFirst:  #当前玩家是本方，Max 节点
                if evl > a:
                    a = evl
                if a > b:
                    return b #返回最大值，剪枝
            else:#当前玩家是对方
                if evl < b:
                    b = evl
                if b <= a:
                    return a #返回最小值，剪枝
            if avil==my:
                break
    if mode == 'direction':
        dl = [2,0,1,3] if not player else [3,0,1,2]
        for direction in dl:
            '''''此处涉及启发性思维，究竟哪个方向优先?'''
            '''''先手靠左'''
            nextboard = board.copy()
            if nextboard.move(player, direction):#可能的移动方向并已发生移动
                evl = ab_eval(isFirst,next_h,nextboard,nextplayer,nextRound,nextmode,a,b)#继续估值
                if player == isFirst:  #当前玩家是本方，Max 节点
                    if evl > a:
                        a = evl
                    if a > b:
                        return b #返回最大值，剪枝
                else:#当前玩家是对方
```

```
                    if evl < b:
                        b = evl
                    if b <= a:
                        return a #返回最小值，剪枝
    if player == isFirst:
        retevl = a    #给上一个 min 节点返回最小值
    else:
        retevl = b    #给上一个 max 节点返回最大值
    return retevl
```

2. easy_eval(isFirst: bool, board: Chessboard) -> int
参数：isFirst 己方
返回：叶节点估值
功能及算法策略：叶节点估值函数。采用对棋子价值阶乘之和作为估值，并增大对方比重以积极攘外。
内包函数 trans(val)：返回棋子价值的阶乘

```
def easy_eval(isFirst,board):
    def trans(val):
        '''''目前对棋子本身价值权重采取阶乘函数，基本没有忽略合并大子'''
        f = 1
        for i in range(1,val+1):
            f = f * i
        return f

    myscores = board.getScore(isFirst)
    otherscores = board.getScore(not isFirst)
    mysum = sum(list(map(trans,myscores)))
    othersum = sum(list(map(trans,otherscores)))
    sumScore =  mysum - othersum*1000
    '''''加重对方比重，积极攘外'''
    return sumScore
```

3. isMine(board: Chessboard, position: Tuple[int, int]) -> bool
参数：position 位置坐标
功能：判断某位置是否在本方领域（调用了函数 ab_eval 的参数 isFirst）

```
def isMine(board,position):   # 判断是否在本方领域中
    return board.getBelong(position) == isFirst
```

4. theNext(position: Tuple[int, int], direction: int) -> bool
参数：position 位置坐标，direction 方向
返回：移动后的位置
功能：获取指定方向的相邻位置

```
def theNext(position,direction):  #direction,上、下、左、右
    delta = [(-1,0), (1,0), (0,-1), (0,1)][direction]
    return (position[0] + delta[0], position[1] + delta[1])
```

5. can_help_attack(board:Chessboard, position:Tuple[int, int], posplayer: bool) -> bool
参数：position 位置坐标，posplayer 探究方
功能：判断该落子点是否可辅助探究方进攻

```
def can_help_attack(board,position,posplayer):#辅助进攻,落子于 player
    alls = [(row,col) for row in range(4) for col in range(8)]
    chesses = [(row,col) for row in range(0,4) for col in range(0,8) if (row,col) not in (board.getNone(True) + board.getNone(False))]
    direct = 3 if posplayer else 2 #落子先手则向右探寻后手是否可进攻
    #nextpos 为模拟 position 向一个方向逐步移动后的位置：
    nextpos = theNext(position,direct)
    for s in range(8):        #s 为 nextpos 与 pos 之间空的距离
        if nextpos in alls:#棋盘内
            if board.getBelong(position) == posplayer:#和落子方同方，则无法攻击
                break
            else:#异方
                if nextpos not in chesses:#对方空格
                    nextpos = theNext(nextpos,direction)#继续前行
                else:#对方子
                    if board.getValue(nextpos) == 1:#2 可吃
                        return True
    return False
```

6. can_attack(board: Chessboard, position: Tuple[int, int], posplayer: bool) -> bool
参数：position 位置坐标，posplayer 攻击方
功能：判断该落子点是否会受攻击方进攻

```
def can_attack(board,position,posplayer):
    alls = [(row,col) for row in range(4) for col in range(8)]
    chesses = [(row,col) for row in range(0,4) for col in range(0,8) if (row,col) not in (board.getNone(True) + board.getNone(False))]
    #nextpos 为模拟 position 向一个方向逐步移动后的位置：
    for direction in [0,1,2,3]:
        nextpos = theNext(position,direction)
        for s in range(8):        #s 为 nextpos 与 pos 之间空的距离
            if nextpos in alls:#棋盘内
```

```
                    if board.getBelong(position) == posplayer:#和攻击方同方
                        nextpos = theNext(nextpos,direction)#继续前行
                    else:#异方
                        if nextpos not in chesses:#对方空格,无法进攻
                            break
                        else:#对方子
                            if board.getValue(nextpos) == 1:#2 可吃
                                return True
        return False
```

2.3 程序限制

在代码竞技场中，由于代码竞技场的加速作用，程序在当前所使用的决策树及估值函数、剪枝方法，以及决策 10 层的层数条件下，程序未出现超时或者出错的现象，平均计算时间一般在 1.5 到 3 秒左右。

如果尝试将层数提高至约 11-13 层，尤其是处于先手时，超时的概率会随之增加，具体取决于对手的程序的情况。

10 层时若剪枝基本效率为 0，落子选择数接近最大情况，那么很有可能博弈树的规模达到 11 层以上量级，从而导致超时出错。

比如以下这种极端情况博弈树规模极大：

对方				我方			
							2
128				2			
				avil			
128							

对方过强，将我方子全吃（剩个 2）而且自己即将全合并。

此时 direction 模式时，边界优先搜索而往边界移动很可能被吃，是最坏情况。10 层有两次落子选择，显然落子己方最差，中间 13 个分数一样，落在 128 之间最好。若第二次落子决策情况基本一样，则博弈树规模约为 4*15*4*15*4 ≈ 4**7。而就算是 11 层，一般情况落子 4 个选择（或正常剪枝的等效结果），规模为 4**6，11 层已经很有可能超时，那么

10 层时这种极端情况就很可能会导致超时。

3 实验结果

3.1 测试数据

实验环境说明

- 硬件配置：i5-9300H 16G
- 操作系统：Windows10
- Python 版本：3.8.2

测试方法：

1、利用本地 tools 文件中 round match 程序对不同参数的 ai 进行比较。

2、在代码竞技场与 official bots 进行对抗，测试代码竞赛能力。

3、在代码竞技场新旧版本互相挑战，更新代码。

测试内容：

1、估值函数的幂次 n trans(x)：return x^n

从 1 开始，不断增大估值函数的幂次，在 1~10 范围内，随着 n 增大，代码能力增强，较大 n 可以战胜较小 n，当 n 较大后（10~10000），不同参数版本之间差别不大，可认为是平手，最后选择 n=20。

后期又测试了其他函数形式，最终选择了效果更好的阶乘的形式。

2、决策树高度 h

h	Average time(offensive) /s	Average time (defensive) /s
8	2.653	1.303
9	2.281	1.143
10	4.988	4.805
11 (超时)	5.042	4.834

不超时的前提下，h 越大，ai 越强

在早期版本落子决策较复杂的情况下，h 最大达到 5，h>5 后出现超时，为增大决策树高度，简化了落子决策，优先下在己方，h 增大到 10。

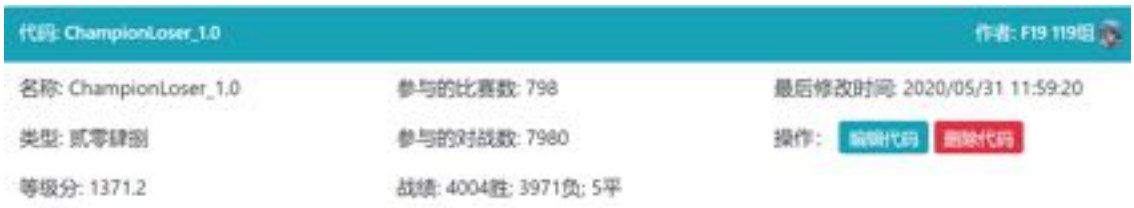
3、估值函数中 rw

总分数=我方分数—rw 倍对方分数 sum Score = my sum – other sum * rw

最初 rw=1，己方和对方比重相同，增大 rw，增加进攻性，与原版本相比能力提升，不断增大 rw，发现到达一定大小后 rw 不再产生影响。

3.2 结果分析

3.2.1 天梯赛结果：



优先下在己方，算法比较保守，实现了较大的决策树高度，初期战绩较好，但后期出现了边界失守的问题，通过增大 rw 的值，增加对方分数比重，提高算法进攻的积极性。

通过大幅度简化估值函数节省时间资源，增大决策树深度的策略意义较大，将胜率提高了很多。

算法在运行时间上控制较好，在预期内，没有出现超时情况。

3.2.2 热身赛战绩

对手（关闭后）	胜	负	胜率
Uniform	1	9	10.00%
Sierra	1	9	10.00%
X-Ray	1	9	10.00%
3721	0	10	0.00%
Bravo	83	147	36.09%
007	142	168	45.81%
666	42	158	21.00%
Romeo	85	155	35.42%
Kilo	136	173	44.01%
2333	353	177	66.60%
Tango	128	261	32.90%
Alpha	247	273	47.50%
Delta	219	229	48.88%
520	288	122	70.24%
India	57	43	57.00%
Papa	136	54	71.58%
Hotel	164	36	82.00%
Victor	73	27	73.00%
404	18	2	90.00%
777	24	6	80.00%

对战性能较好的代码时表现并不是很好

4 实习过程总结

4.1 分工与合作

4.1.1 小组分工：

算法框架： 小组讨论完成
算法实现与完善：陈鹏 张子羲
算法调试：杜泽宇
撰写算法说明：陈鹏
撰写实验报告：杜泽宇
撰写实习报告：曹珂

4.1.2 小组交流

组内交流主要通过视频会议、微信群聊、码云来实现。在算法确定之后未进行长时间的组会，改由通过微信群、发 issue 等方式完成交换思想和提出建议。相比于视频会议，组员可以有更长的时间考虑某个问题，有新的建议或想法也可以快速表达。

4.1.3 时间节点

2020-5-16 第一次组会，初步分配任务，每一位组员完成自己对于项目实现的基本思路
2020-5-19 第二次组会，确定了以决策树+剪枝+启发式规则为核心的算法为实现方案
2020-5-19 码云仓库建成，分享代码和收集建议的平台
2020-5-26 估值函数完成
2020-5-27 第一个可以无错运行的 AI 完成
2020-5-29 算法突破，搜索效率大大提高，大幅提高了 AI 的实力
2020-5-31 天梯排名首次大于等于 12

第一次组会合影

第二次组会留念

疯狂改进代码

经典白给局

第一次击败 bot_S

4.2 经验与教训

整个开发后期小组成员积极参与到代码的改进和调试工作上，开发工作高效进行。码云仓库的使用使得小组内代码共享、想法交换变得十分快捷方便，特别是调试阶段大量的代码修改、调参，在码云上可以看的很清晰。但是工作前期缺乏 ddl 的督促，小组缺乏动力，工作进展十分缓慢，导致最终没有充分的时间将组员的想法付诸实践检验。历史告诉我们 ddl 是第一生产力，开发时应该对整个工作进行仔细的规划，设置合理的 ddl，流程安排尽量使得组员之间依赖性不强。

coder 最好是一人负主责。我们队一开始没有确定主要的 coder，队员就有点“消极怠工”，险些成为 ddl 战士（虽然也差不多）。coder 应及时向队友清晰讲解新增属性的思想，便于更多人参与，集思广益。然后参数调试的任务可以合理地及时地分配给辅助的同学，配合起来效率更高。

调试参数一定要有清晰的理论图景指导，不然很抓瞎。一开始考虑增大对方子值的影响的“攘外”思想是正面教材，有清晰的思路，检验也很顺利。而像后来的幂指函数调试失败就是典型反例。另一方面，也不必完全囿于现有理论，有时多尝试反而有意想不到的结果，

比如队友张子羲同学大胆尝试极端剪枝方法，ai 性能确实大幅上升。

分工上可以更合理。一个 coder，一个参数调试，一个数据收集者（紧密观察 ai 在天梯的表现，整合战报，组内进一步分析）。数据收集真的很重要，更关键的是信息的整合，看到本质问题。这最好由辅助 coder 担任，因为他更了解 ai 的原理。

自己的想法要尽量凝练而清晰，注重表达，这样便于与同学交流合作。感觉前部分时间和队友交流花了不少时间解释一下想法。

整理、记录工作很重要，不然常常有重复工作，浪费时间。

4.3 建议与设想

可视化可以做得更精致。本次活动后期战况可视化做得比较好，前期观看比赛总归是有点无聊且费劲

陈斌老师的讲解很细致，风格很活泼，很亲民的感觉；大作业安排很有新意，作业安排也会根据同学们的压力来修改。希望学弟学妹能有所收获。

天梯可以增加人机对战，且个人可以参与排名；加入陈老师大战各路 AI 参与天梯排名顺带祝学弟学妹在大作业玩得开心。

后续或许可以尝试炼丹？或者将这一算法思想的框架应用在其他棋类游戏上，又或者尝试更多的人工智能算法，进一步提高 AI 的实力。

5 致谢

感谢陈斌老师的决策树讲解和一学期的教授
感谢助教的组织、批阅
感谢技术组的工具、平台支持
感谢所有组员投入的时间和心血
感谢 OfficialBot 的陪练

6 参考文献

http://www.icourse163.org/course/0809PKU037-1003471009
https://github.com/pkulab409/sessdsa.2048

数据结构与算法课程实习作业报告

张芳瑞* 杜广昊 张续航 张哲玮 李骁

摘要：在本次二〇四八小组实习作业中，本组 AI 的主要算法原理为通过搭建针对棋盘 board 类的决策树，用 min-max 方法和估值函数判断最优决策，涉及到自定义类 TreeNode 的构建及 Player 类的完善。在天梯对战中，本组 AI 取得不俗的成绩，但最终以几分之差遗憾倒在小组赛上、未能出线。

关键字：min-max 方法的决策树、启发式规则、二〇四八

1 算法思想 2
1.1 总体思路 2
1.2 算法流程图 3
1.3 算法运行时间复杂度分析 4
2 程序代码说明 4
2.1 数据结构说明 4
2.2 函数说明 4
2.3 程序限制 5
3 实验结果 5
3.1 测试数据 5
3.2 结果分析 8
3.3 经典战局 9
4 实习过程总结 9
4.1 分工与合作 9
4.2 经验与教训 10
4.3 建议与设想 11
5 致谢 11
6 参考文献 12

1 算法思想

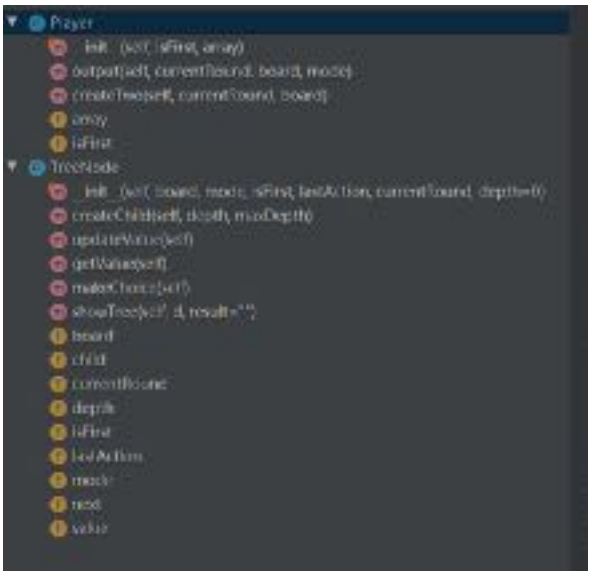
1.1 总体思路

在编写 AI 的过程中，本组大致思路为由易到难逐层递进，即先实现底层功能，再依据策略逐步提升 AI 性能。版本更迭流程如下表:

AI 版本	更新内容
1.0	只考虑合并决策优先级，基本完成可对战 AI 的实现
2.0	可构建一层决策树用来搜索合并决策的 AI，模仿算分搭建估值函数
3.0	在前一代 AI 的基础上增加决策树深度，采取 min-max 决策树，完善针对超时的代码、调整估值函数（最终采用版本）
4.1	在第三代 AI 的基础上优化估值函数，并尝试加入启发式规则
4.2	在第三代 AI 的基础上采取 Alpha-Beta 剪枝
4.3	在第三代 AI 的基础上增添进攻性策略，关注强调地盘入侵
4.4	第三代 AI 的基础上关照对可能实现大规模消去的合并策略的考量,避免过早塞爆

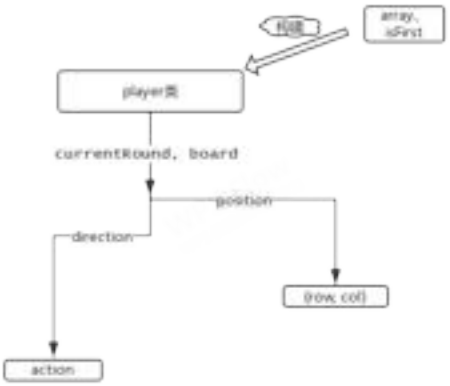
```
def getValue(self):  # 估值函数，返回估值
    vList = self.board.getScore(self.isFirst)
    v = 0
    for val in vList:
        v += 17 ** val
    return v
```

最终稿采用的估值函数代码

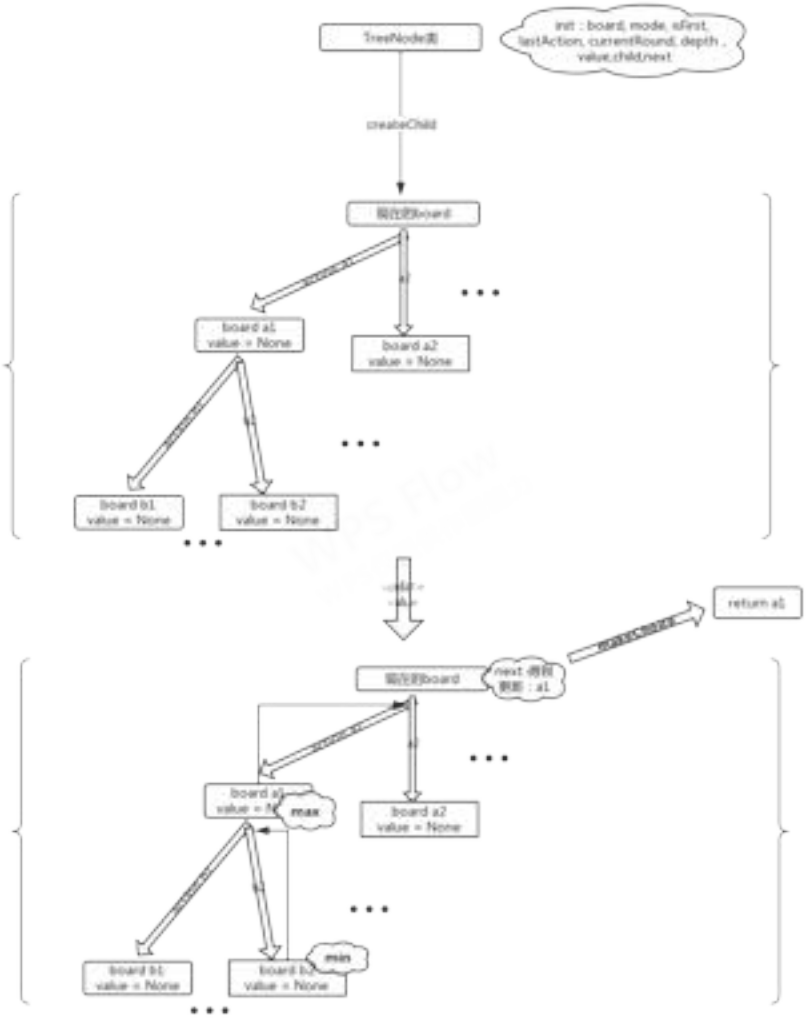


算法内部的类和函数

1.2 算法流程图



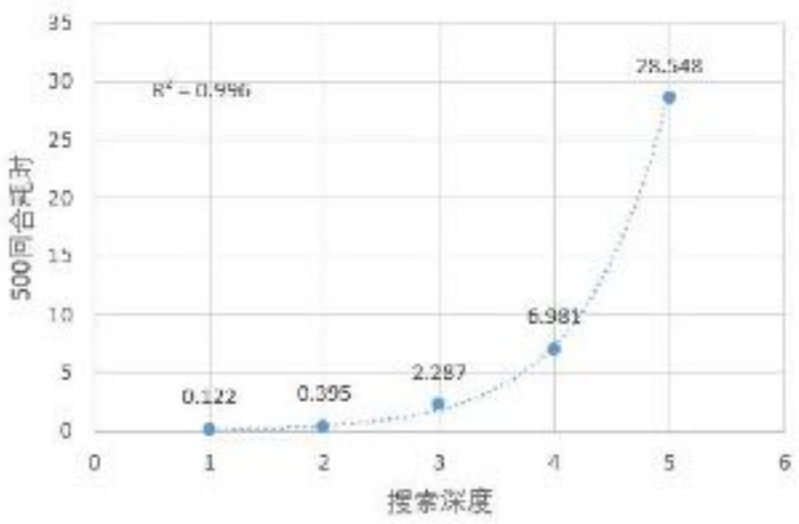
代码工作整体流程



算法具体流程

1.3 算法运行时间复杂度分析

考虑到算法采用了树的结构，其时间复杂度应该是指数级别。为了验证该结论，我们对算法的时间复杂度进行了测试，所需要消耗的时间如下图所示：



时间复杂度测试散点图

可见算法的时间复杂度至少是指数级别。

在搜索深度为 3 的情况下，构造树结构、对各结点赋值用时共 2.287 秒，如果只是进行赋值语句但不调用估值函数，则构建树结构和赋值共用时 2.080 秒。若只构建树结构，则用时 2.050 秒。该实验结果非常有力地说明，算法主要时间消耗是在构建树结构上。

2 程序代码说明

2.1 数据结构说明

主要数据结构：

min--max 决策树

自定义类：

Player 类：主要实现 AI 的底层功能，如接收棋盘信息、输出下棋方式等；

TreeNode 类：内部实现 min--max 决策树构建，负责估值的维护更新、路径的保存等，并运用了课程所学的广度优先搜索、启发式规则等。

2.2 函数说明

Player 类：

createTwo:

主要功能：根据 currentRound 和当前棋盘 board，返回落子方式

具体实现：

A. 判断是否可以下子，若可以则下在己方。

B. 否则，针对 getNone 返回的每个对象，综合其 val 和所在行己方边界是否有子判断是否下在这里

output:

mode 为”position”时，调用 createTwo 方法进行落子。

mode 为”direction”时，通过 TreeNode 类建立决策树，具体如下：

1.可用时间决定搜索深度，并获取当前局势；

2.通过调用 TreeNode 中的 createChild 建立决策树，建立完成后调用 TreeNode 的 update 方法更新树的估值，并在估值的过程中实现 min--max 决策；

3.调用树的 makeChoice，通过估值返回合并决策。

TreeNode 类

各节点对应棋盘及其相关局势、路径等参数。

createChild：

主要功能：输入最大深度，生成相应子树。

具体实现：落子不向下建立节点只做更新（落子决策唯一），针对每种合并决策建立子节点，递归调用子节点的 createChild 以实现决策树的构建。

getValue：

估值函数，针对节点的当前棋盘 board 返回估值。

updateValue：

主要功能：更新决策树各节点估值。

具体实现：通过 min--max 算法，向上更新节点估值，保存路径。

makeChoice:

通过路径返回决策内容。

2.3 程序限制

在多次调试下，代码无明显错误，在运行过程中几乎不会超时，偶尔出现被塞爆的情况，也已通过内部调参、估值函数调整将代码性能调至最优。

3 实验结果

3.1 测试数据

实验环境说明：

- 硬件配置：（CPU/内存） i7-9750H
- 操作系统：（名称/版本） Windows 10
- Python 版本：（版本号） 3.7.4

本组代码前期主要依靠技术组提供的 userinterface 进行测试，利用这个控件完成了前两代代码的编写。但是，在逐渐加深搜索深度时，组内成员发现代码的本地运行时间与在天梯网站上相比更长。我们发现这是由于技术组在天梯网站上使用了 cpp 加速库，所以从第三代代码编写开始，我们的绝大部分测试都是在天梯完成。

在天梯赛以及热身赛当中，我们的代码胜率如下表所示：

	第一代	第二代	第三代	第四代		
				估值函数	大量剪枝	进攻性
先手胜率	预计为 0	15.4%	71.6%	57.6%	33.2%	不到 10%
后手胜率	预计为 0	13.2%	67.9%	45.3%	31.7%	81.5%
是否进入热身赛	否	是	是	是	是	是
是否进入天梯赛	否	否	是	否	否	是
是否最终采用	否	否	是	否	否	否

值得一提的是，组里曾经开发过几种不同类型的第四代代码。主要包括三种：

第一种，调整估值函数，除累加各个棋子的值外，遍历每行每列，每个棋子入栈，比较是否有相同且相邻的棋子来调整估值。由于使用栈导致时间复杂度急剧增加，只好进一步缩减层数，得到的效果不如第三代。

```
def getValue(self):#估值函数，返回估值
    vList = self.board.getScore(self.isFirst)
    v = 0
    for val in vList:
        v += 17**val
    if self.isFirst:      #如果是先手：
        waiting=self.board.getRaw()
        end=[0,0,0,0]
        for i in range(0,3):
            test=Stack()  #建立栈
            for j in range(0,3):
                if waiting[i][j][0] !=test.peek():
                    test.push(waiting[i][j][0])  #若无相同元素则入栈
                else:
                    end[i]+=waiting[i][j][0]
                    test.pop()      #有相同元素则记录，并弹出栈
        endend=max(end)    #求出四行中能合成的最大值
    else:   #如果是后手：
        waiting=self.board.getRaw()
        end=[0,0,0,0]
        for i in range(0,3):
            test=Stack()
            for j in range(4,7):
                if waiting[i][j][0] !=test.peek():
                    test.push(waiting[i][j][0])  #若无相同元素则入栈
                else:
                    end[i]+=waiting[i][j][0]
                    test.pop()      #有相同元素则记录，并弹出栈
        endend=max(end)     #求出四行中能合成的最大值
    alpha=0.2
    return (1-alpha)*v+alpha*endend
```

4.1 版本的估值函数

第二种，采用更大量的剪枝，力争加深搜索深度。但是，在开发剪枝算法时，并没有使算法的时间复杂度下降，反而变得极不稳定，故最终未采用。

```
def updateValueCut(self):
    #叶节点先取到值
    if self.isLeaf():
        self.value = self.getValue()
    #接下来向上传递

    if self.isMax():#自己是max节点，则父节点是min
        #首先处理父节点
        if self.parent == None:
            return
        if self.parent.value == None:
            #如果父节点还未被赋值过，就直接赋值
            self.parent.value = self.value

            #父节点处理完了，处理爷节点
            if self.parent.parent != None:
                if self.parent.parent.value == None:
                #同样是没被赋值过，直接赋值
                    self.parent.parent.value = self.value
                elif self.parent.parent.value >= self.parent.value:
                #如果爷节点的值已经大于当前节点，那么父节点就可以停止探索
                    self.parent.value = self.value
                    self.parent.confirm = True

        elif self.parent.value > self.value:
            #如果父节点的值大于self的值，就赋值，因为父节点是min要找最小值
            self.parent.value = self.value

            #父节点处理完了，处理爷节点
            if self.parent.parent != None:
                if self.parent.parent.value == None:
                #同样是没被赋值过，直接赋值
                    self.parent.parent.value = self.value
                elif self.parent.parent.value >= self.parent.value:
                #如果爷节点的值已经大于当前节点，那么父节点就可以停止探索
                    self.parent.value = self.value
                    self.parent.confirm = True
```

剪枝部分代码

第三种，增加算法的攻击性。其攻击性体现在估值函数上，因此我们以如下两个参数作为调整变量：

1) 基于对各组代码对战的观察，我们发现许多小组在一开始都不会选择进攻。因此我们设置起始回合数 round_begin，超出该回合数后再开始调用进攻性的算法。

2) 赋予特殊棋子更高的权重 a，从而使决策树偏向于进攻。这样的特殊棋子包括在对方阵营的棋子和在己方阵营边界上的棋子。

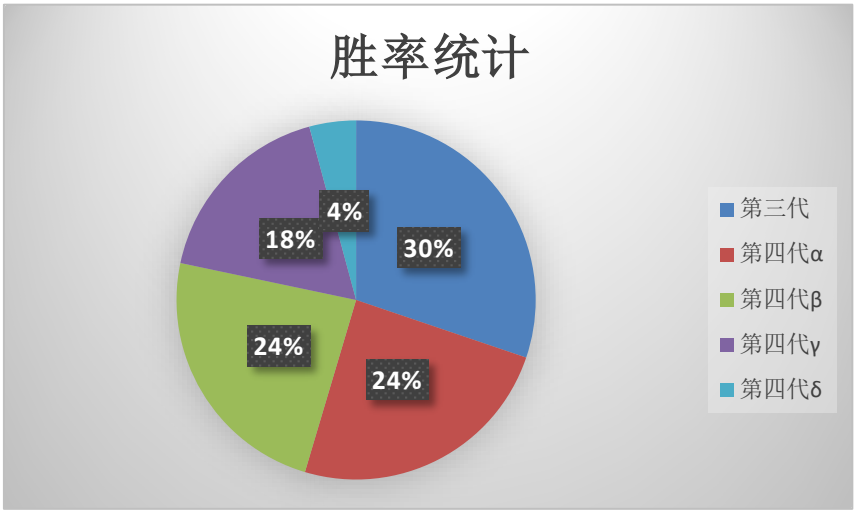
```
def getValue(self):#估值函数，返回估值
    vList = self.board.getScore(self.isFirst)
    v = 0
    tList = self.board.getScore(not self.isFirst)
    for val in vList:
        v += 17 ** val
    for val in tList:
        v -= 17 ** val

    n = len(self.board.getNone(not self.isFirst))
    a = 0
    round_begin=100
```

```
    n = len(self.board.getNone(not self.isFirst))
    a = 0
    round_begin=100

    if self.isFirst and n <= 12 and self.currentRound > round_begin:

            n = len(self.board.getNone(not self.isFirst))
            a = 0
            for row in range(4):
                if self.board.getBelong((row,5)) == self.isFirst:
                    a += 0.03 * self.board.getValue((row,5))
                if self.board.getBelong((row,4)) == self.isFirst:
                    a += 0.085 * self.board.getValue((row,4))
                if self.board.getBelong((row,3)) == self.isFirst:
                    v += 0.11 * self.board.getValue((row,3))
                else:
                    v -= 0.1 * self.board.getValue((row,3))
            for row in range(4):
                for col in range(4):
                    if self.board.getBelong((row,col)) != self.isFirst:
                        a -= 0.1 * self.board.getValue((row,col))
```

```
elif (not self.isFirst) and n <= 12 and self.currentRound > 100:

    for row in range(4):
        if self.board.getBelong((row,2)) == self.isFirst:
            a += 0.03 * self.board.getValue((row,2))
        if self.board.getBelong((row,3)) == self.isFirst:
            v += 0.085 * self.board.getValue((row,3))
        if self.board.getBelong((row,4)) == self.isFirst:
            v += 0.11 * self.board.getValue((row,4))
        else:
            v -= 0.1 * self.board.getValue((row,4))
    for row in range(4):
        for col in range(4,8):
            if self.board.getBelong((row,col)) != self.isFirst:
                v -= 0.1 * self.board.getValue((row,col))
if v < 0:
    v = v * (2 - a)
else:
    v = v * (1 + a)

return v
```

进攻性更强的估值函数

在实际测试过程当中，我们不断微调 round_begin 和 a 两个参数，得到了四种不同版本的算法。我们将其与第三代算法一起共五个代码放在本地轮战，综合胜率如下所示：



三代与四代代码胜率统计图

由于综合来看，各种关于第四代的测试在稳定性上都不如第三代，因此最终保留了第三代的代码。

3.2 结果分析

在测试、热身和实战对弈之中，本算法的决策树和启发式规则起到了主要作用。在热身赛和天梯赛的初期，本算法达到了非常理想的效果，后期排名则趋于浮动。

算法初期的运行时间比预想中的要长，但因为天梯网站加入 cpp 加速包之后运行效率明显提高，我们此后将搜索深度逐渐加深，通过多次实验探究出不会超时的最大搜索深度。

由 1.3 小节理论和实验给出结果可知，本组算法主要运行时间开销在于构建树结构环节。另外，本组曾经尝试过换用更加复杂的估值函数，时间复杂度随之明显增加。换用估值函数后，在平均搜索深度（3 层）调用时用时 2.42 秒。可见，本小组算法时间分配是优化树结构和采用较简单的估值函数的综合结果。

3.3 经典战局

由于我们的算法采用了比较保守的运行方式，所以在实际运行中达到了意料之外的效果。以下是我们在天梯赛初期一次对局的照片：

早期比赛结果照片

通过这场比赛可以看到，本组代码多次以使对方超时的方式获得了胜利。考虑到对方的代码在其他组对战时几乎没有超时的情况，可见我方算法稳定性极好，从而让对方计算量过大而导致超时。

4 实习过程总结

4.1 分工与合作

小组分工：

杜广昊：报告修改、后期工作。

李骁：报告初稿、思路贡献。

张哲玮：代码编写、算法开发、思路贡献。

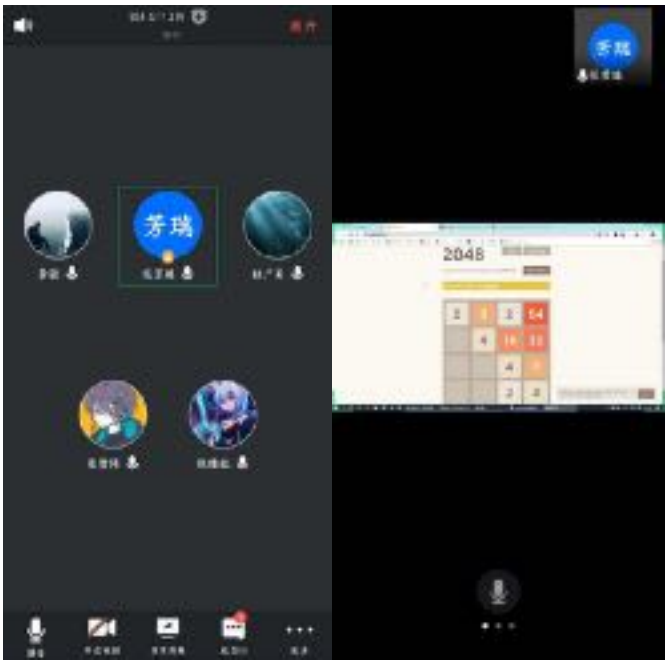
张续航：思路贡献、算法开发。

张芳瑞：报告初稿、组织调度。

合作交流方式：

小组内的主要交流方式是微信群聊，遇到关键性事项时，组内也会通过线上会议的方式开展组会。负责算法开发的同学会将最新版本代码发到微信群内，供大家讨论商议。各组员

角色并非一成不变，遇到问题时都会积极参与讨论，为 AI 编写或算法方向建言献策。



组会现场，组长组织商讨分工、并讨论出了小组第一稿 AI 的设想

小组微信群，组员反复调整代码、相互交流

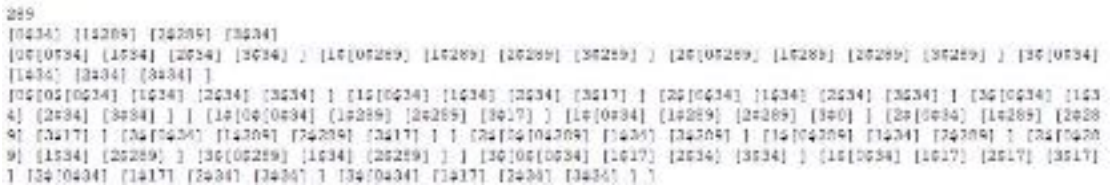
4.2 经验与教训

我们的得意之处：

1.虽然时间匆忙、但是在组员的共同努力下完成了一个基于基础设施平台的可对战、高胜率的智能 AI。

2.在算法开发过程中，我们还通过实现决策树的可视化输出，来实时监测代码运行情况，

极大地提高了算法开发效率



决策树可视化输出效果

3.组员间和谐工作、为相关代码的完成提供了好的氛围和环境，各位成员积极配合工作

获得的经验教训：

1.在思考自身 AI 不足的同时缺少对对手、对天梯的观察，致使小组赛前不久发现胜率与积分不匹配的情况

2.在对付实力和排名低于自己的 AI 中却不能取胜，可能是存在算法能力不稳定、存在缺陷的原因

4.3 建议与设想

实习作业建议：

1.优化基础设施代码的可视化部分，提供对设施内部实现的简单介绍；

2.组队环节可以较为集中，最好在课程开始时划分明确的时间段，避免拖延时间过长或引发“组队焦虑”。

3.竞赛环节可添加“复活赛”和“表演赛”，使竞赛增加更多悬念和趣味性。

致学弟学妹们的寄语：

首先要恭喜你们，选到了这门“宝藏课程”！但同时也要提醒你们，这门课的“趣味性”与“挑战性”并存。虽说课程任务量大，但只要肯投入时间和精力，相信你们一定会收获颇丰。更重要的一点，时刻保持你的信心！因为这门课大佬、“卷王”无数，小心打击过大，一蹶不振，最终退课跑路（笑）。

实习作业后续设想：

可以在该网站或另开设网站公开天梯赛对战平台，对北京大学内部或其他机构平日开放，供大家学习参考。对于参与竞赛的学生，可以提升自己的代码水平并提高自己的理解；对于陈斌老师和数算课的各位助教来说，可以积累运营对战平台的经验，并根据反馈，使平台的视觉效果和体验效果更好。

另外的设想是，将这次大作业的内容做成棋类游戏，供大家游玩。

5 致谢。

小组成员一致向以下人员表示感谢：

感谢陈斌老师的悉心教学和指导，并给我们这次参与大作业实习的机会。

感谢技术组的各位同学们，为本次二〇四八对战的顺利进行保驾护航。

感谢地空数算的各位助教，为我们解答问题和提供建议。

感谢比赛过程中的各位对手，是你们间接完善了我们的代码，让它更加成熟。

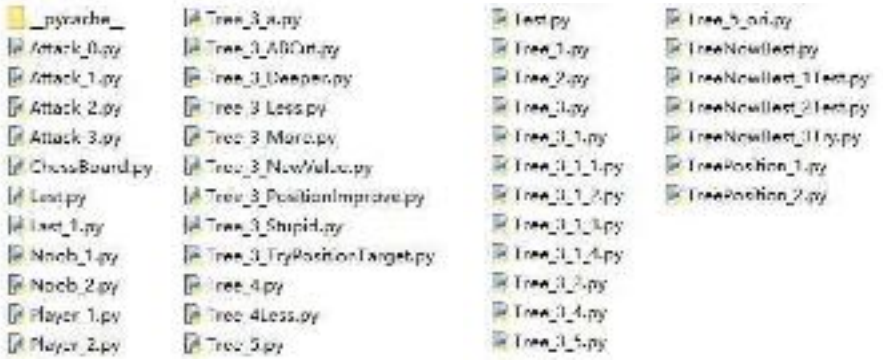
组长在此感谢以下组员的贡献：

感谢李骁同学撰写了实验报告相关部分的初稿，并在集体商讨中给出了自己的想法；

感谢杜广昊同学为实验报告做了整体修改、美化，并始终支持组内工作；

感谢张续航同学提供的算法思路，并时时传递天梯赛赛况；

感谢张哲玮同学负责了绝大部分的算法开发、时间检验以及代码编写和测试工作。在天梯赛结束前夜一直监测代码运行情况到很晚，并且始终高度关注天梯赛赛况。



历代代码开发

6 参考文献

[1]《决策树与决策森林》，https://www.cnblogs.com/fionacai/p/5894142.html

[2] 张效见，《五子棋计算机博弈系统的研究与设计》[D]安徽大学.2017.05.01

[3]《决策树算法的容错性研究及其应用》 [J]中国新通讯 2019.02.20

数据结构与算法课程实习作业报告

李志昊* 吴芊染 何沛一 赵乐石 胡踵书

日期：2020 年 6 月 9 日

摘　　要

在本次的2048 对抗赛中，本组采用了 Minimax 搜索树算法，结合 $\alpha-\beta$ 剪枝与持久化搜索树提高搜索效率，并运用启发式规则和简洁的估值函数完成决策，在天梯小组赛中成功出线. 虽然不幸在提交代码最后时刻没有进行全面的测试，导致在直播赛中败北，但在复盘中为观众提供了欢乐，不失为一种成功.

关键词： Minimax,$\alpha-\beta$ 剪枝，持久化搜索树，聚合估值，启发式规则

目录

1 算法思想 **3**

1.1 总体思路 . . . 3

1.2 算法流程图 . . . 4

1.3 算法运行时间复杂度分析 . . . 5

2 程序代码说明 **5**

2.1 数据结构说明 . . . 5

2.2 函数说明 . . . 5

2.2.1 class TreeNode . . . 5

2.2.2 class Tree . . . 8

2.2.3 class Player . . . 10

2.3 程序限制 . . . 11

3 实验结果 **11**

3.1 测试数据 . . . 11

3.2 结果分析 . . . 11

3.3 经典战局 . . . 12

4 实习过程总结 **14**

4.1 分工与合作 . . . 14

4.2 经验与教训 . . . 14

4.2.1 聚合估值的探索 . . . 14

4.2.2 持久化的失误 . . . 15

4.3 建议与设想 . . . 16

5 致谢 **16**

参考文献 **17**

1 算法思想

1.1 总体思路

AI 的内置功能大致为: 参考Wikipedia, 构建 Minimax Tree , 每个节点储存该步步骤及棋局状态; 根据下一步棋局可能成为的状态构建子节点; 使用估值函数对每个节点进行估值; 最终 $\alpha-\beta$ 剪枝得到可能得到的最有利情况.

首先, 我们的估值函数由三个部分组成: 棋子价值、位置价值和空格价值.

- 由于一个合并后的棋子价值大于两个合并前的棋子, 故将棋子价值函数设为以 2.5 为底的指数函数.
- 刚开始, 我们采取的是较为保守的战略. 由于在对方一侧的棋子难以移动, 同时容易被吞噬使得对方总体价值提高, 我们选择在对方的棋子价值降低. 但由于实验发现这会导致己方棋盘位置逐渐被对方占据, 最终难以执行有效步骤, 我们改变了战略. 改为将对方距离我方最近的一侧的列价值升高, 随着与我方距离增加, 价值依次降低. 这是由于我们认为这可以促进我方棋子吞并对方棋子, 同时距离我方阵营较近, 便于撤退且难以被吞食. 但在实测过程中, 我们又发现这会使得 AI 引导对方的棋子吞并己方棋子以使得该棋子深入, 因为这在我们的价值判断体系中认为价值是在降低的. 但事实上这是不利于我方的. 最终我们得到结论: 在对方棋盘位置的棋子可以占领对方的位置, 而深入敌方虽有危险, 但却可以在很大程度上起到阻碍合并的作用. 故我们最终认为对方棋盘位置的价值应适当整体大于己方, 且各个位置价值相同. 故将自己一侧棋盘位置价值均设为 1, 对方位置均设为 6.
- 由于在空位较多的情况下若将己方一个空位价值设的过高, 则会导致 AI 偏向于向对方位置下棋, 相当于给对方 “送子” ; 而当己方空位价值过少时, 若被堵死, 则之后无路可走, 棋局成为对方的单人秀. 因此, 空格的价值函数应当满足: 在空格个数较多的时候, 单个空格价值相对较低, 低于一个落子的价值; 而空格较少时, 一个空格价值极高. 因此, 选择了

$$\text{空位价值}=\frac{256}{\text{该方空位数}}.$$

最终每一方的价值为该侧每个棋子自身价值与位置价值相乘的求和加上空位价值构成. 棋盘的总价值由先手方减去后手方价值构成.

落子时, 建立一个队列, 用于储存下子位置. 判断若对方空格位置少于三, 则将对方所有空位入列; 若己方棋盘, 若空格位置大于二, 则将己方将要下棋位置加入队列. 这是因为可以避免己方堵死, 尽力堵死对方. 若此时队列仍为空, 则按次序依次加入: 若己方可下棋, 加入己方位置; 若对方可下棋, 随即加入对方一位置. 若仍无法下棋, 则直接构建无步骤对应子节点.

合并时, 将四个位置入列, 若无法合并, 则直接构建无步骤对应子节点.

为了使得 AI 获得目光长远、矢志不渝的优良品质, 应当使 AI 具有执行长远策略的能力. 因此, 搜索树的层数应当尽量提高, 以尽量选择得到全局的最优解而非局部. 为了实现这个目的的同时能够节省时间开销, 我们选择了固定化搜索树, 这样可以避免重复构建相同的子节点, 从而节省时间. 同时, 每次遍历完一组子节点后, 将子节点按照价值从小到大排序, 每次遍历节点时, 先手方从后向前遍历, 后手方从前向后遍历, 使得剪枝能够在最大程度上起作用.

最终执行时, 为了尽量使得搜索层数更加深, 我们采取了动态规划时间的措施, 使得主要时间内可以搜 9 层至 8 层, 而若时间已将尽, 则将层数下调至 4 层. 这样使得对于大部分局不受影响,

只对少数局势有较小影响.

1.2 算法流程图

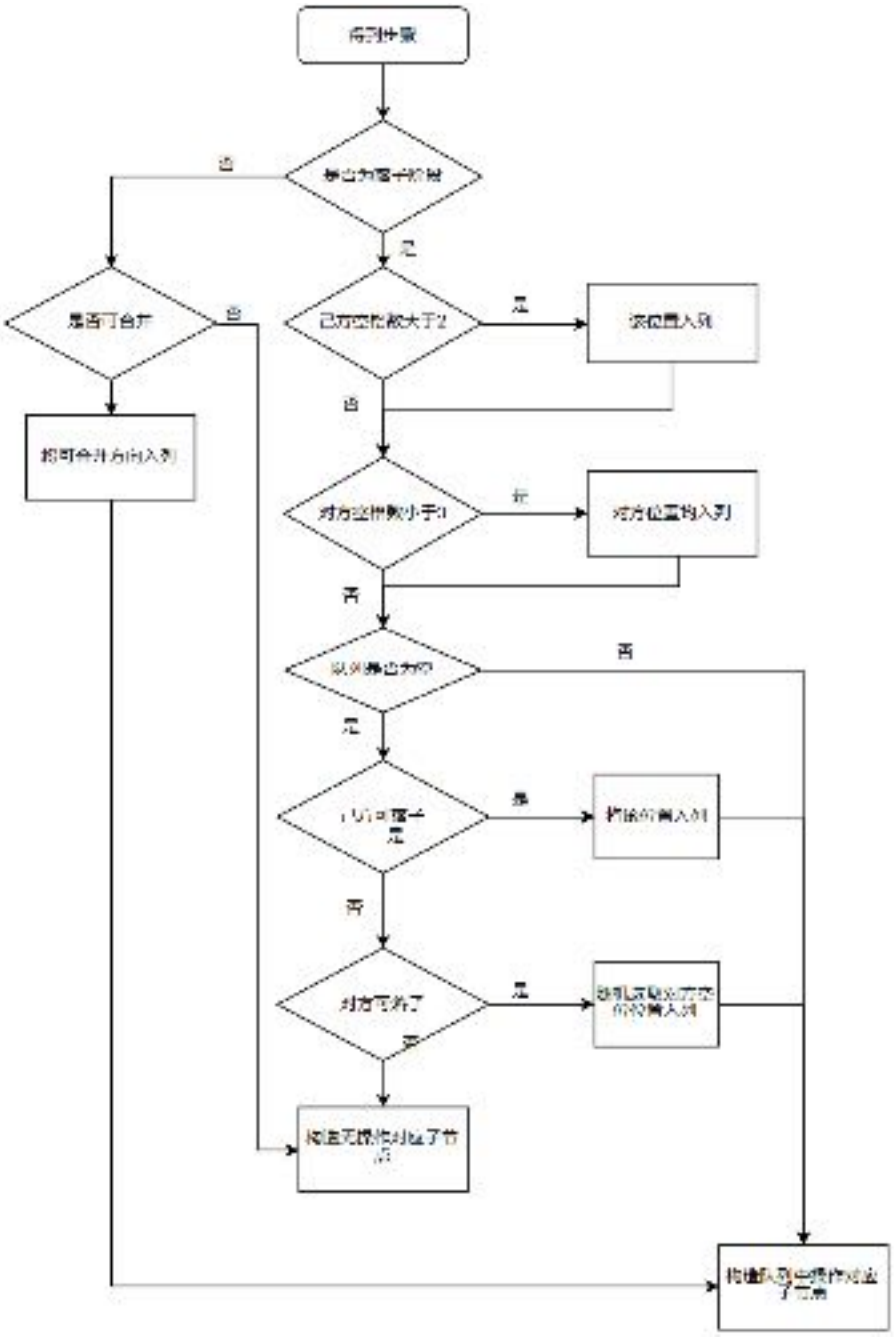


图 1: 算法流程图

1.3 算法运行时间复杂度分析

代码中的运算主要在于对每个节点估值、构造子节点以及构造 MiniMax 搜索树三个部分, 其中估值函数每次均为遍历一遍棋盘, 因此所需时间为 $O(1)$, 而构造子节点时每次最多构造 4 个, 时间复杂度也为 $O(1)$. 对于 MiniMax 搜索树, 由于节点个数与搜索层数成指数相关, 因此时间复杂度为 $O(2^N)$. 每次将子节点按照价值排序后, 能够大幅提高剪枝效率, 从而可以降低底数. 因此在加上持久化树过程后, 最好情况为每次对方所下位置均被我方预测到, 此时 MiniMax 搜索树不需重建, 只需向下拓展一层即可, 此时对于已经构建好的树只需要进行对比操作, 可以降低时间复杂度的常数部分.

2 程序代码说明

2.1 数据结构说明

算法中主要采用了树的数据结构. 算法 AI 中用到了`TreeNode`、`Tree`、`Player` 3 个自定义类. 其中`TreeNode`类是搜索树结点,`Tree`类是 MiniMax 搜索树,`Player`类是游戏调用的玩家类. 算法总体上采用 MiniMax 搜索树与 $\alpha-\beta$ 剪枝技术, 其中 $\alpha-\beta$ 剪枝可以减少树结点的分支数量, 使搜索的结点数减少, 进而可以在相同的时间内搜索更多的层数.

2.2 函数说明

2.2.1 class TreeNode

```
class TreeNode:
    # Minimax搜索树节点
    def __init__(self, board, currentRound, mode, isFirst, parent=None):
        # 初始化搜索树节点
        self.board = board
        self.currentRound = currentRound
        self.mode = mode
        self.isFirst = isFirst  # 是否先手,决定min-max序列
        self.parent = parent    # 父节点
        # 子节点以元组存储在列表中
        # 元组第一项是产生该子节点的操作,第二项是子节点本身
        self.children = []
        self.value = 0  # 局面价值
        self.alpha = -float('inf')  # alpha 边界值
        self.beta = float('inf')    # beta 边界值
```

`__init__`: 初始化搜索树结点, 结点属性包含当前的局面、回合数、下棋模式、先后手、父结点、子节点、当前局面可行操作列表、局面价值及 α、β 值.

```
    POSITION_VALUE_LIST = [1, 1, 1, 1, 6, 6, 6, 6]
    POSITION_VALUE_LIST3 = [6, 6, 6, 6, 1, 1, 1, 1]
    def GetValue(self):
```

```
        # 盘面价值函数
        value = 0
        temp_board = self.board.getRaw()  #获取盘面信息
        empty_value = 256 / (len(self.board.getNone(True)) +
                             1) - 256 / (len(self.board.getNone(False)) + 1)
                                #空位价值
        for x in range(4):
            for y in range(8):
                rank, belong = temp_board[x][y]
                if rank != 0:  #非空位
                    if belong:   #正手方,用POSITION_VALUE_LIST
                        value += (2.5**rank) * POSITION_VALUE_LIST[y]
                    else:        #后手方,用POSITION_VALUE_LIST3
                        value -= (2.5**rank) * POSITION_VALUE_LIST3[y]
        self.value = value - empty_value    #合计价值
        return self.value
```

GetValue: 当前局面价值判断. 采用先手方总价值减去后手方总价值作为当前棋盘价值, 其中一方的价值包含棋子价值和空位价值.

$$\text{空位价值} = \frac{256}{\text{该方空位数}}$$

$$\text{棋子价值} = \sum 2.5^{\text{棋子级别}} * \text{位置权重}$$

其中位置权重由POSITION_VALUE_LIST和POSITION_VALUE_LIST3给出, 分别对应先手方和后手方的位置权重. 权重作为启发式规则的一部分可以让棋子更偏向于向对面移动, 可以加速塞满对方, 使己方更容易获胜.

```
    def BuildChild(self):
        # 构建子节点
        # 结构:(action,node)
        if self.mode == 'position':
            # 下子阶段
            position_list = []
            my_list = self.board.getNone(self.isFirst)   #我方空位
            my_position = self.board.getNext(
                self.isFirst,
                self.currentRound) if self.currentRound < 500 else ()
                #我方在己方下子处位置
            others_list = self.board.getNone(not self.isFirst)       #对方空位
            if len(others_list) < 3:
                #如果对面空位少于3,则将空位位置加入到待选位置列表中
                position_list = others_list
            if len(my_list) > 2 and my_position:
                #如果己方空位大于2且可在己方下子
                #则将该下子位置加入到待选位置列表中
                position_list += [my_position]
            if not position_list:  #如果待选位置列表为空
```

```
                if my_position:
                    position_list = [my_position]
                if others_list:
                    #从对方空位中随机选取一个加入待选位置
                    position_list += [random.choice(others_list)]
                elif not position_list:
                    #既无己方空位也无对方空位可下
                    #则以无操作建子节点
                    child = (None,
                             TreeNode(
                                 self.board, self.currentRound,
                                 'position' if self.isFirst else 'direction',
                                 not self.isFirst, self))
                    self.children.append(child)
                    return

            for pos in position_list:
            #将待选位置列表中的位置一个个取出建立对应子节点
                _board = self.board.copy()
                _board.add(self.isFirst, pos)
                child = (pos,
                         TreeNode(_board, self.currentRound,
                                  'position' if self.isFirst else 'direction',
                                  not self.isFirst, self))
                self.children.append(child)

        elif self.mode == 'direction':
            # 移动阶段
            _board_moved = False
            for direction in [2, 3, 0, 1]:
                #对于每个可移动方向建立对应子节点
                _board = self.board.copy()
                board_moved = _board.move(self.isFirst, direction)
                if board_moved:
                    _board_moved = True
                    child = (direction,
                             TreeNode(
                                 _board, self.currentRound if self.isFirst else
                                 (self.currentRound + 1),
                                 'direction' if self.isFirst else 'position',
                                 not self.isFirst, self))
                    self.children.append(child)
            if not _board_moved:
                # 无处移动
                child = (None,
                         TreeNode(
                             self.board, self.currentRound if self.isFirst else
```

```
                                (self.currentRound + 1),
                                'direction' if self.isFirst else 'position',
                                not self.isFirst, self))
                self.children.append(child)
```

BuildChild: 为当前结点生成可行操作和子结点.

在下子阶段时, 若对面的空位数小于 3, 则将对面空位加入可行操作. 若己方有空位且空位数大于 2 则将己方下子位置也加入可行操作. 当两个条件都不满足时则将可行的所有操作均加入可行操作列表. 最后对每个可行操作复制棋盘进行下子并生成子节点. 其中用到的启发式规则是空位较多时优先己方下子, 而当对面空位较少时考虑下在对面加速对面塞满.

在移动阶段时, 按[2,3,0,1]的顺序考虑所有方向, 若为可行方向则将其加入子节点. 采用[2,3,0,1]的搜索顺序会优先搜素 2 3 方向, 有助于提高剪枝效率, 快速找到最优解.

若无任何可行动作时, 生成一个空动作的子节点.

2.2.2 class Tree

```
class Tree:
    # Minimax搜索树
    def __init__(self, currentRound, board, mode, isFirst, maxLength):
        # 构造Minimax搜索树
        self.root = TreeNode(board, currentRound, mode, isFirst)  #根节点
        self.maxDepth = maxLength - currentRound                  #最大深度
```

__init__: 初始化 Minimax 搜索树, 树属性包含根节点与最大深度, 根据最大长度和当前轮数确定, 防止搜索时超过回合.

```
    def sink(self, last_action, board, currentRound, mode, isFirst):
        # 下沉到子节点
        self.root.alpha = -float('inf')
        self.root.beta = float('inf')
        for childNode in self.root.children:
            if last_action == childNode[0]:
            #如果找到上次操作对应的子节点,则继承该子节点下的树
                self.maxDepth += self.root.currentRound - childNode[
                    1].currentRound
                self.root = childNode[1]
                return
        #否则重建树
        self.root = TreeNode(board, currentRound, mode, isFirst)
        return
```

sink: 此函数为树的持久化服务, 为了尽量减少每回合搜索与以前搜索的重复, 我们记录该回合之前的操作, 并调用上回合的树找到对应的子节点来继承, 如此一来对树的更新相比次次重建效率大幅提高. 如果对手的决策不在原来的子节点考虑中或者被剪去, 则原树只能抛弃重建.

```
    def Search(self, currentNode, depth):
```

```
        depth = min(depth, self.maxDepth)
        if depth > 0:
            # 要求搜索深度不为0,继续递归搜索
            if not currentNode.children:
                currentNode.BuildChild()
            # minimax 搜索 + alpha-beta 剪枝
            # 继承父节点的alpha-bet值
            if currentNode.parent:
                currentNode.alpha = currentNode.parent.alpha
                currentNode.beta = currentNode.parent.beta
            # 最大化本方
            if currentNode.isFirst:
                currentNode.value = -float('inf')
                for childNode in currentNode.children[::-1]:
                    self.Search(childNode[1], depth - 1)
                    currentNode.value = max(currentNode.value,
                                            childNode[1].value)
                    currentNode.alpha = max(currentNode.alpha,
                                            currentNode.value)
                    if currentNode.beta <= currentNode.alpha:
                        # beta 剪枝
                        break
            # 最小化对手
            else:
                currentNode.value = float('inf')
                for childNode in currentNode.children:
                    self.Search(childNode[1], depth - 1)
                    currentNode.value = min(currentNode.value,
                                            childNode[1].value)
                    currentNode.beta = min(currentNode.beta, currentNode.value)
                    if currentNode.beta <= currentNode.alpha:
                        # alpha 剪枝
                        break
            currentNode.children.sort(key=lambda x: x[1].value)
        if depth == 0:
            currentNode.GetValue()
```

search: 这是搜索树进行搜索的核心部分, 在传入节点与深度后, 进行递归调用, 最底层是直接根据盘面调用估值函数进行价值的计算. 往上层走, 正手方选取子节点的最大价值作为自己的价值, 在不断比较中更新自己的下限, 如果自己的价值下限高于父节点的价值上限, 则无需进行后续搜索, 可直接剪枝; 后手方选取子节点的最小价值作为自己的价值, 在不断比较中更新自己的上限, 如果自己的价值上限小于父节点的价值下限, 则也无需进行后续搜索, 可直接剪枝.

```
    def output(self, time_left):
        #根据剩余时间确定搜索深度
        if time_left > 1.5:
            depth = 9
```

```
        elif time_left > 0.3:
            depth = 8
        else:
            depth = 4
        self.Search(self.root, depth)
        if self.root.isFirst == True:
            #先手取盘面最大价值
            action = self.root.children[-1][0]
            self.root = self.root.children[-1][1]
            return action  #返回操作
        elif self.root.isFirst == False:
            #后手取盘面最小价值
            action = self.root.children[0][0]
            self.root = self.root.children[0][1]
            return action  #返回操作
```

output: 接收一个剩余可用时间, 进行动态时间管理, 根据时间不同来确定搜索深度, 以达到最优化, 函数中调用search函数完成搜索, 根据正反方的不同选取策略, 并将树下沉到对应子节点.

2.2.3 class Player

```
class Player:
    # 棋盘基本输入输出类
    def __init__(self, isFirst, array):
        # Player 对象的初始化过程
        self.isFirst = isFirst
        self.array = array
        self.Tree = None
        self.maxLength = len(array)
```

__init__: 初始化Player, 玩家属性包含正后手、确定己方下子的随机数列、搜索树、最大长度.

```
    def output(self, currentRound, board, mode):
        # 决策时调用函数
        time_available = board.getTime(self.isFirst) #获取剩余时间
        last_action = board.getDecision(not self.isFirst)
        last_action = None if not last_action else (
            last_action[0] if len(last_action) == 1 else last_action)
        if currentRound == 0:
            self.Tree = Tree(currentRound, board, mode, self.isFirst,
                             self.maxLength)
        if mode[0] == '_':
            self.Tree.sink(last_action, board, currentRound, mode[1:],
                           self.isFirst)
            self.Tree.sink(None, board, currentRound, mode[1:], self.isFirst)
            return
        else:
```

```
            self.Tree.sink(last_action, board, currentRound, mode,
                           self.isFirst)
            return self.Tree.output(time_available)
```

output此函数在玩家决策时调用, 最开始回合初始化树, 以后回合获取对手上一步的操作, 由此确定下沉的方向, 调用搜索树中的sink, 实现持久化, 最后返回对应树的输出, 即己方策略.

2.3 程序限制

在持久化方面, 如果对手的策略不在我们原树的子节点中, 如对手采用某种启发式规则、对方没有采用我方构想的启发式规则、对方的策略在剪枝过程中减去等情况, 即" 不按我方常理出牌", 则搜索树只能重建, 如果这种情况经常发生, 则很可能导致超时报错.

在前一代直播赛的代码中,getNext里传入的回合数没有加上限制, 导致在高回合数时会产生超最大回合数的情况导致报错, 现已修复.

3 实验结果

3.1 测试数据

实验环境说明:

- 硬件配置: Intel(R)Core(TM)i7-9750H CPU @ 2.60GHz 2.59GHz
- 内存: 8GB
- 操作系统: Windows10 x64
- Python 版本: Python 3.8.1 (tags/v3.8.1:1b293b6, Dec 18 2019, 23:11:46)[MSC v.1916 64 bit (AMD64)] on win32
- 平台基础: 数据结构与算法大作业代码对战平台

代码的主要测试方式是在大作业对战平台上和其他代码进行对战, 因此主要性能依靠对战平台的配置.

表 1是小组天梯赛中的部分对局和战绩

3.2 结果分析

考虑实战效果, 在实战对弈过程中.

- 本算法中的估值函数发挥了重要作用. 这其中让棋子深入敌方占据对方位置的策略有效地阻碍了对方棋子地合并, 可以降低对方棋子的总价值. 对于空格价值的利用策略即当空格的数量多时价值较低, 当空格数量少时价值较高, 则大幅度防止过度给对方下棋, 同时还避免了己方被堵死. 在 MiniMax 搜索树方面,$\alpha-\beta$ 剪枝技术起到了减少树结点的分支从而避免部分的无用计算, 节省计算力, 可以搜索更多的层数; 树的持久化技术则减少了每个回合的搜索量和以前的重复搜索, 一般情况下在对树进行更新的时候就不必每次都要重建搜索树, 大幅度提升了效率.

对战代码	比赛结果	积分变动
F19 007	8:2 胜出	1542.91+19.26
F19 2333	7:3 胜出	1566.55+6.15
F19 Romeo	6:4 胜出	1552.21-4.32
F19 Sierra	4:6 败北	1586.49-0.45
F19 119	10:0 大胜	1547.89+25.67
F19 666	9:1 大胜	1559.33+19.16
F19 Golf	5:5 平手	1558.78+10.10
F19 X-ray	4:6 惜败	1571.11-5.02
F19 Echo	3:7 败北	1589.17+0.03
F19 Uniform	4:6 败北	1592.96+3.62

表 1: 小组赛天梯对战结果表

- 上述策略和算法都起到了良好的效果. 提升己方价值, 降低对方价值, 尽量避免己方棋子被对方棋子吞并, 尽量避免己方的移动方向被堵死. 在算法层面, 剪枝技术避免了不必要的计算, 节约了时间; 搜索树持久化技术则有效避免了重建搜索树的时间损耗. 使得 AI 的战斗力大大提升.
- 算法的运行时间在预期之中. 由于搜索层数的多少是依据运行时间来动态分配, 因此当剩余时间比较多的时候, 可以搜索更多的层数, 当剩余时间比较少的时候, 可以适当降低搜索层数; 同时搜索树持久化技术也节省了重建的时间, 当搜索树未重建时, 可以适当增大搜索层数, 当搜索树发生重建时就适当减少搜索层数. 如此动态调整使得时间的利用更加安全高效, 几乎不会发生超时的现象.
- 算法主要的运行时间开销发生在搜索树的构建与搜索阶段. 当搜索层数增大一层时, 运行时间会大幅度增加.

3.3 经典战局

在我组代码与 F19 Romeo 组的代码对战的过程中, 出现了十分经典有趣的局面. 我方为后手方, 棋盘右侧领地是我方属地. 原本两组代码势均力敌, 局面一直相持不下, 这个时候我方代码使出了十分狡猾的策略. 注意到此时两组代码的最高级别棋子都是 1024, 而且两方的 1024 都位于边界位置. 同时两方代码都各自有一个棋子 16 位于边界位置. 如**图 2**所示.

我方 AI 在之前的操作步骤中并没有对己方 16 这一棋子进行保护, 主要在于判断到对方的最大棋子 1024 已经位于边界位置, 因此己方棋子 16 的价值便不是很高, 对方的棋子 1024 的价值十分的高, 因此我方 AI 等待对方 AI 向右合并来吞并我方的棋子 16. 如**图 3**.

此时对方的棋子 1024 已经位于我方棋子 1024 的正下方, 并且已经轮到我方操作. 在我方向下合并之后, 直接吞并了对方的 1024, 此时我方的最高级别棋子变为 2048, 而对方的最高级别棋子被替换为 256, 双方原本焦灼的局面瞬间被打破, 我方代码瞬间取得了对敌方 AI 的压倒性优势.

图 2: 经典对局 1

图 3: 经典对局 2

并最终以三级压制轻松拿下比赛. 如**图 4**.

图 4: 经典对局 3

4 实习过程总结

4.1 分工与合作

本小组合作分工如**表 2**

组员	分工
李志昊	重编了基本 $\alpha-\beta$ 剪枝代码, 编写了持久化搜索树 v2, 编写部分估值函数 (已弃用)
吴芊染	完全编写了估值函数、下子策略 (最终采用)
何沛一	本地测试, 编写部分启发规则, 编写部分估值函数 (已弃用)
赵乐石	本地测试, 编写部分估值函数 (已弃用)
胡踵书	编写了持久化搜索树 v1, 编写部分启发规则, 编写部分估值函数 (已弃用)

表 2: 组员分工情况表

4.2 经验与教训

4.2.1 聚合估值的探索

本组在初期的代码建设中曾经使用过博弈树的聚合估值技术, 聚合估值的出现源于极小极大过程 (Minimax 搜索) 存在的问题.

由于在极小极大过程中, 非叶结点的倒推估值是其子结点估值的极值, 这样就不可避免地丢失某些已经获得的信息, 甚至做出不甚合理的判断. 如**图 5**所示, 极小极大过程选择的最佳着法是

B. 分析博弈树叶结点的估值可知, 选择 A 和选择 B 差别不大. 但 A 结点可生成许多价值极高的结点, 而 B 不能. 尽管假设 MIN 节点每步都做极理智的选择是合理的, 但在这种情况下, 由于 A 的极大的潜在价值, 因而选择 A 是合理的. 无论是业余的还是职业的人类棋手都会选择 A.(廖士中 (1992))

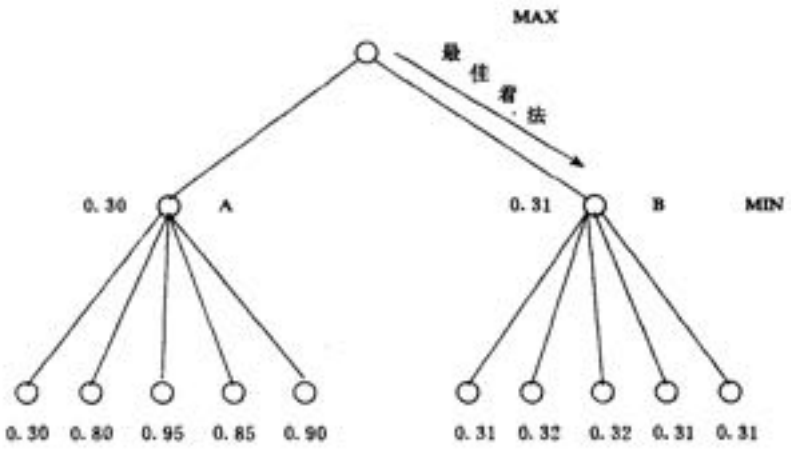


图 5: 聚合估值出可能应用情况举例

本组根据文中的聚合估值方法进行了编程与实战, 发现实战效果并不如意. 如**图 6**.

图 6: 聚合估值与上一代 AI 的战斗

在与其他小组的对战中, 聚合估值也并未有令人惊异的表现, 反而是因为聚合估值需要针对一个决策节点的所有子决策节点进行加权操作. 繁重的计算导致了搜索层数的下降, 反而使得 AI 的综合性能下降了.

4.2.2 持久化的失误

为了追求性能的最大化, 提升搜索层数, 我们采用了持久化搜索树. 但是在编写过程中发生了一个逻辑问题, 组员均未发现.

问题在sink函数的第4行附近. 错误地将修改根节点 $\alpha-\beta$ 值的过程放在了修改树根节点为子节点之后. 虽然咋听上去很有道理, 但是参考Search函数的第8行, 子节点的搜索过程中会继承父节点的 $\alpha-\beta$ 值, 导致“下沉”操作的修改被覆盖.

这个隐形问题一直没有被发现, 组员一度认为持久化的繁琐反而导致了性能的下降. 在出线小组提交代码的最后关头, 问题被发现, 经过修改和经过简单的测试 (**图 7**), 发现可以大幅提升搜索效率和性能. 但是仓皇修改导致报错 (未检查 500 局上限限制), 错失进步的机会.

图 7: 新持久树与错误持久树的对局

4.3 建议与设想

- 对战方面: 希望对战局数多一点,10 盘对于结果的判断还是很模糊, 希望老师能申请到强劲的服务器.
- 建议增设一个出线小组问题检查机制, 在结束提交代码之后依次简单测试一下代码有没有问题, 有的话通知该组组长可以做一下修改. 避免“组长临时添 bug, 组员两行泪”的结局, 对组员也较为公平的.
- 组队方面: 可以用平时的 H 系列作业等来作为划分组队依据, 做一做“大佬均衡”. 避免多位大佬挤在一个组里, 每个组也至少得有一个大佬帮忙带着.
- 对学弟学妹: 恭喜你们踏上了内卷的满满征途, 祝你们卷卷更健康, 越卷越快乐!

5 致谢

感谢陈斌老师在这一个学期数据结构与算法课程中的悉心教导, 丰富的课程内容与生动的讲解让我们收获知识, 并运用于大作业之中.

感谢助教老师们的倾情付出, 他们认真负责批改我们的作业, 解答我们的问题, 组织了大作业平台的搭建, 天梯的管理, 比赛的安排, 课程的成功离不开他们的付出.

感谢技术组的同学们对平台的建设与维护, 感谢他们在天梯中提供的水平测试 AI, 特别是戴天翔同学的大力支持.

感谢在天梯中与我们组对战的同学们, 代码的改进离不开与他们的切磋.

最后感谢小组内每一位成员对本次大作业对战与报告书写做出的贡献.

参考文献

廖士中, 1992. 博弈树的聚合估值方法及其剪枝技术[J]. 辽宁师范大学学报 (自然科学版)(2):20-28.

WIKIPEDIA. Minimax[EB/OL]. https://en.wikipedia.org/wiki/Minimax.

数据结构与算法课程实习作业报告

陈康垚，胡小龙，李永康，王梓钰*，张明杰

摘要：根据局面构建博弈树，采取启发式规则对树进行简化，对叶节点建立评估函数进行评分并通过 MinMax 方法和 alpha-beta 剪枝向上传递,最终获得最优下一步。最终版算法在大量对战未出现明显 bug，可以看出一定的前瞻性。但由于进攻性较弱，在天梯赛中 4278 胜 3592 负，胜率 54.4%，排名并不理想。

关键字：博弈树搜索，剪枝，启发式规则

1 算法思想.....1
1.1 总体思路.....1
1.2 算法流程图.....2
1.3 算法运行时间复杂度分析.....2
2 程序代码说明.....4
2.1 数据结构说明.....4
2.2 函数说明.....5
2.3 程序限制.....10
3 实验结果.....10
3.1 测试数据.....10
3.2 结果分析.....11
3.3 经典战局（可选）.....12
4 实习过程总结.....113
4.1 分工与合作.....113
4.2 经验与教训.....14
4.3 建议与设想.....14
5 致谢.....14
6 参考文献.....15

1 算法思想

1.1 总体思路

对抗版 2048 游戏是双方的完全信息博弈，故采取了通用的博弈树搜索算法。程序先根据游戏提供的信息以当前局面为根节点，以所有可能的下一步的局面为子节点，层层深入，构建了一个树结构；再对设定深度的叶节点通过一个函数（value）进行静态局面估计，并通过 MinMax 方法向上传递评估得分；最后取根节点的子节点中得分最高的作为下一步走法返回。这是在最开始确定的基本思路。

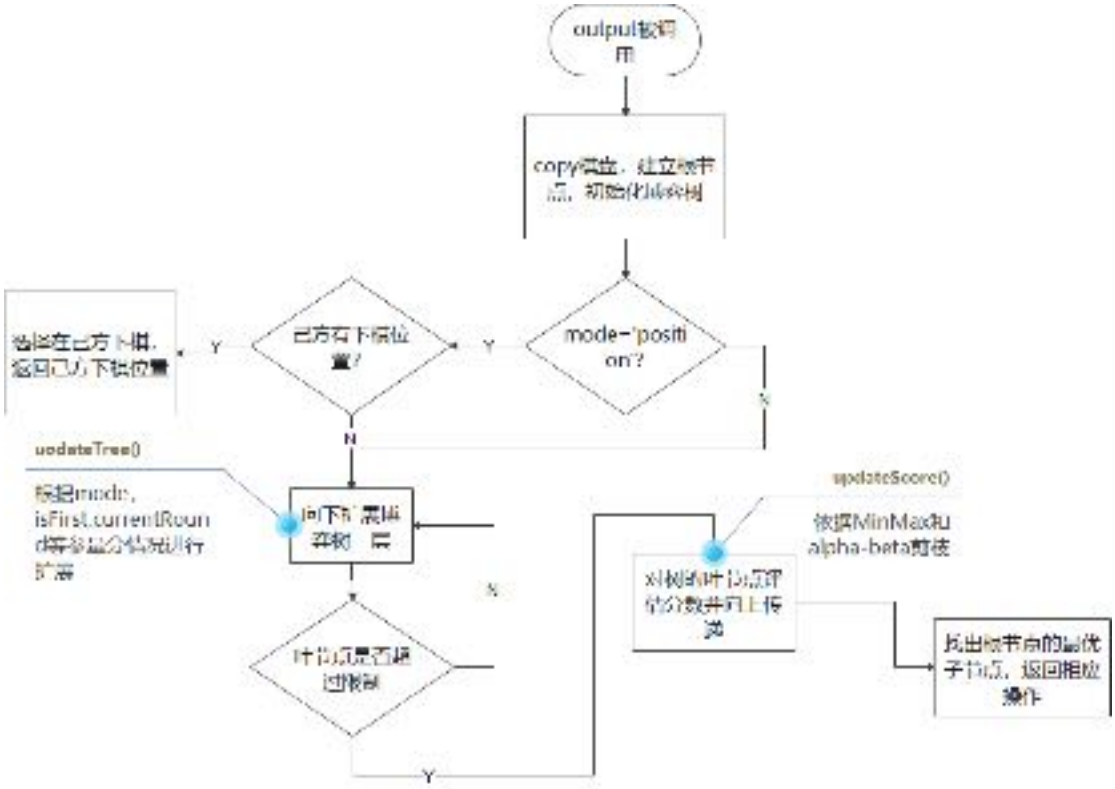
在算法具体实现和测试过程中，又做了一些尝试和策略的调整：由于时间资源有限，采

用了 alpha-beta 剪枝，并加入了一些预剪枝规则，且调整了搜索次序，达到了更好的剪枝效果；经过对对局的分析发现在自己方下子在多数时候都是较优选择，于是加入启发式规则，若己方有空位则直接选择在己方下子，很大程度上简化了树的构建，降为最多四叉树，从而加深了探索层数；对 value 函数也参考网上常规 2048 算法进行了各种尝试，以期达到更好的前瞻性，不过效果并不很理想。由于运算时间限制，最终选择了较为简单的 value 函数，搭配较深的搜索层数的算法。

在实际研究算法的过程中我们发现搜索层数是决定战斗力的首要因素，因此后期修改致力于通过简化树结构，加强剪枝效果等来节约时间从而使搜索更深。其中对 value 进行改变尝试发现复杂的 value 导致了不小的时间资源消耗，不如多搜索几层。故而最终放弃了复杂的 value 函数。在层数的控制上我们利用树的叶节点总数来调节，一层一层扩建树直到叶节点总数超过限定数量。将 0 到 499 回合分为四个阶段，每个阶段赋予不同的最大叶节点总数，具体值通过综合考虑时间限制和重要性程度来确定。最终层数大概在 6 到 9 层不等。

1.2 算法流程图

主要流程（一些细节省略）：



1.3 算法运行时间复杂度分析

算法运行时间主要由构建决策树，计算局面估值和剪枝三部分组成。

1. 构建决策树。

每一个回合会构建四层树，分别对应己方下棋，对方下棋，己方合并，对方合并，而由于己方下棋时，一般会直接选择下在本方，合并方向通常有 4 个选择，只有对方下棋时要构

造的局面较多，大约在 10 个左右，至多为 16 个（即本方的空位数），所以一个回合总共可以构建的局面至多有 4*4*16=256，若以回合数作为问题规模，则复杂度的大 O 数量级为 O(256**n)，若以树的层数为问题规模，则复杂度的大 O 数量级为 O(4**n)，二者均为指数复杂度。

2. 局面估值。

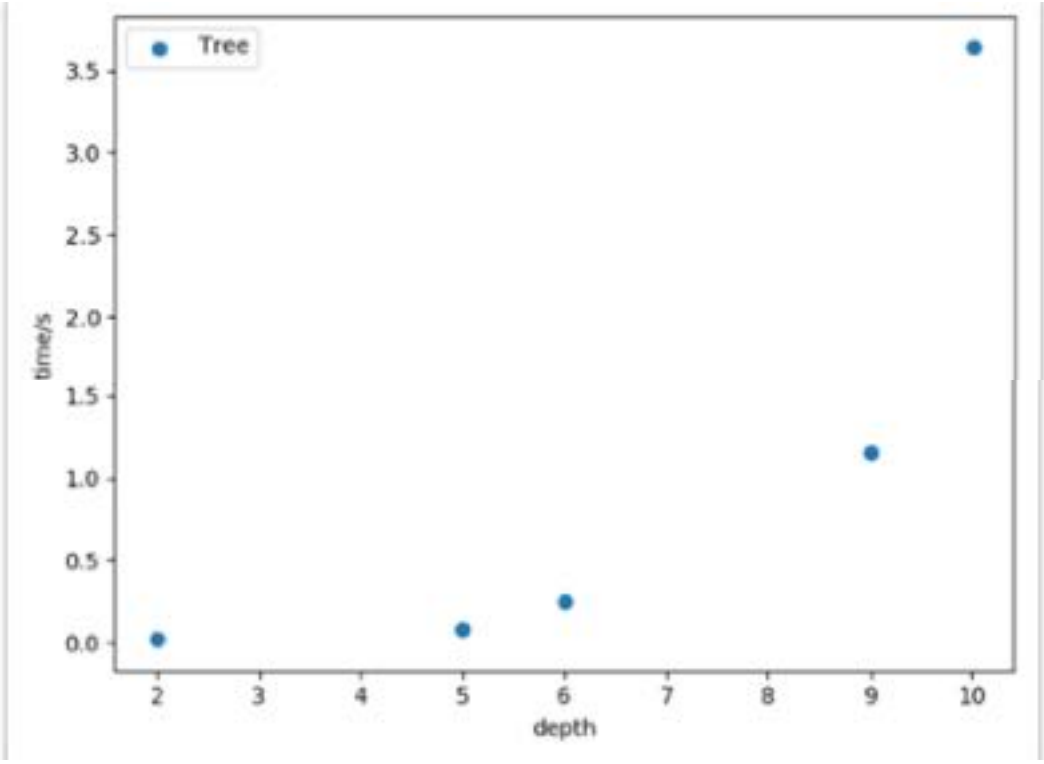
局面估值对当前搜索深度下，所有的最终的局面（叶节点）进行估值，叶节点的个数为 4**n，其中 n 为搜索深度，即决策树的层数。而每一次的估值，利用了己方与对方棋子的大小，耗时与盘面中的棋子个数有关，若以盘面中棋子个数为问题规模 n，那么每一次的估值复杂度为 O(n)，考虑到棋盘中至多有 32 个棋子，单次的估值对于一次完整的决策来说可视为常数复杂度，而一次决策的总的估值的复杂度则为 O(4**n),其中 n 为搜索深度，亦为指数复杂度。

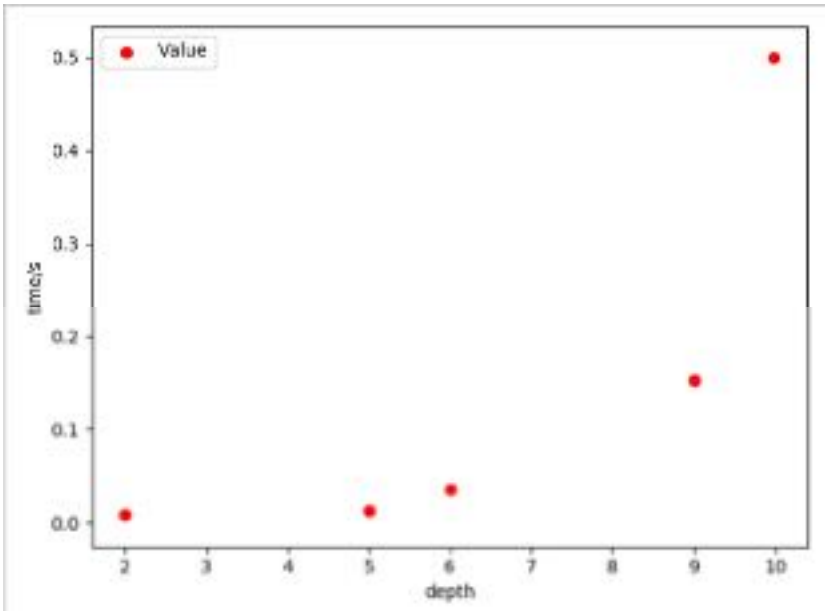
3. 剪枝。

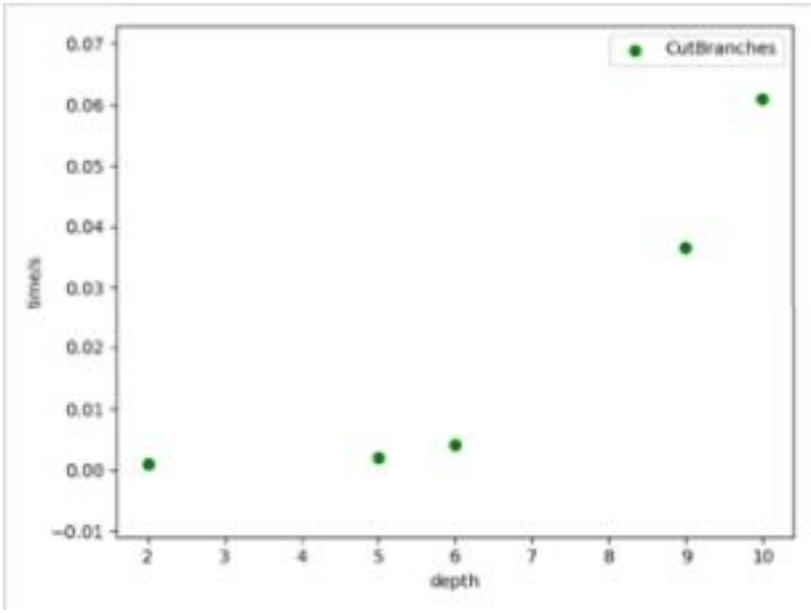
剪枝对整个树进行 DFS，尽管在不同的情形下，剪枝效果不同，搜索的节点数目也不同，但就数量级来看，需要遍历进行判断的节点数目仍大致与整个树的节点个数相同，若树的层数为 n，则复杂度约为 O(4**n)，指数复杂度。

综上，算法的整个复杂度为 O(4**n),其中 n 为搜索深度，为指数复杂度。

4. 实际运行开销。





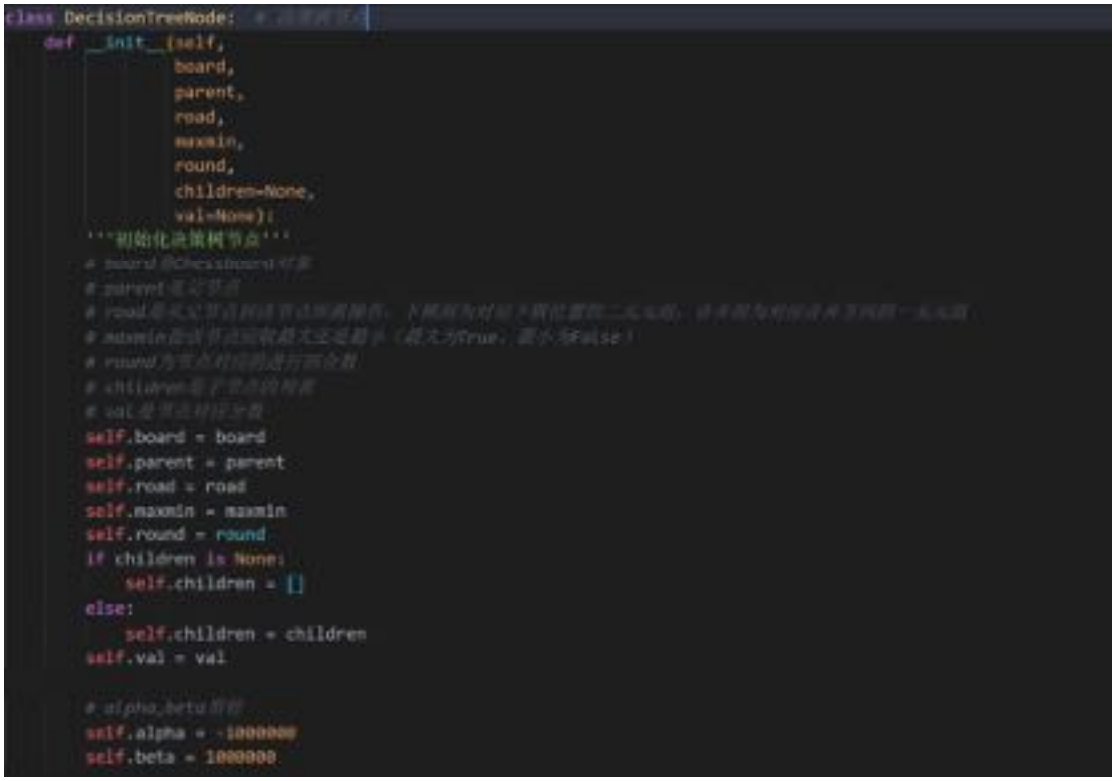


在回合数为 20 的情况下，对三部分算法的运行时间进行测试，Tree 指建树时间，value 和 cutbranches 分别指估值和剪枝时间，纵轴显示的是时间，横轴显示的是搜索深度，由于指数增长的原因，当深度到达 10 层时，时间已经过长无法再进行测试，但已初步显示出指数增长的趋势，同时，可以看出，主要的时间开销集中在建树上。

2 程序代码说明

2.1 数据结构说明

主要采用了树的数据结构，辅以线性数据结构（python 列表实现）。
自定义了博弈树节点类和博弈树类：

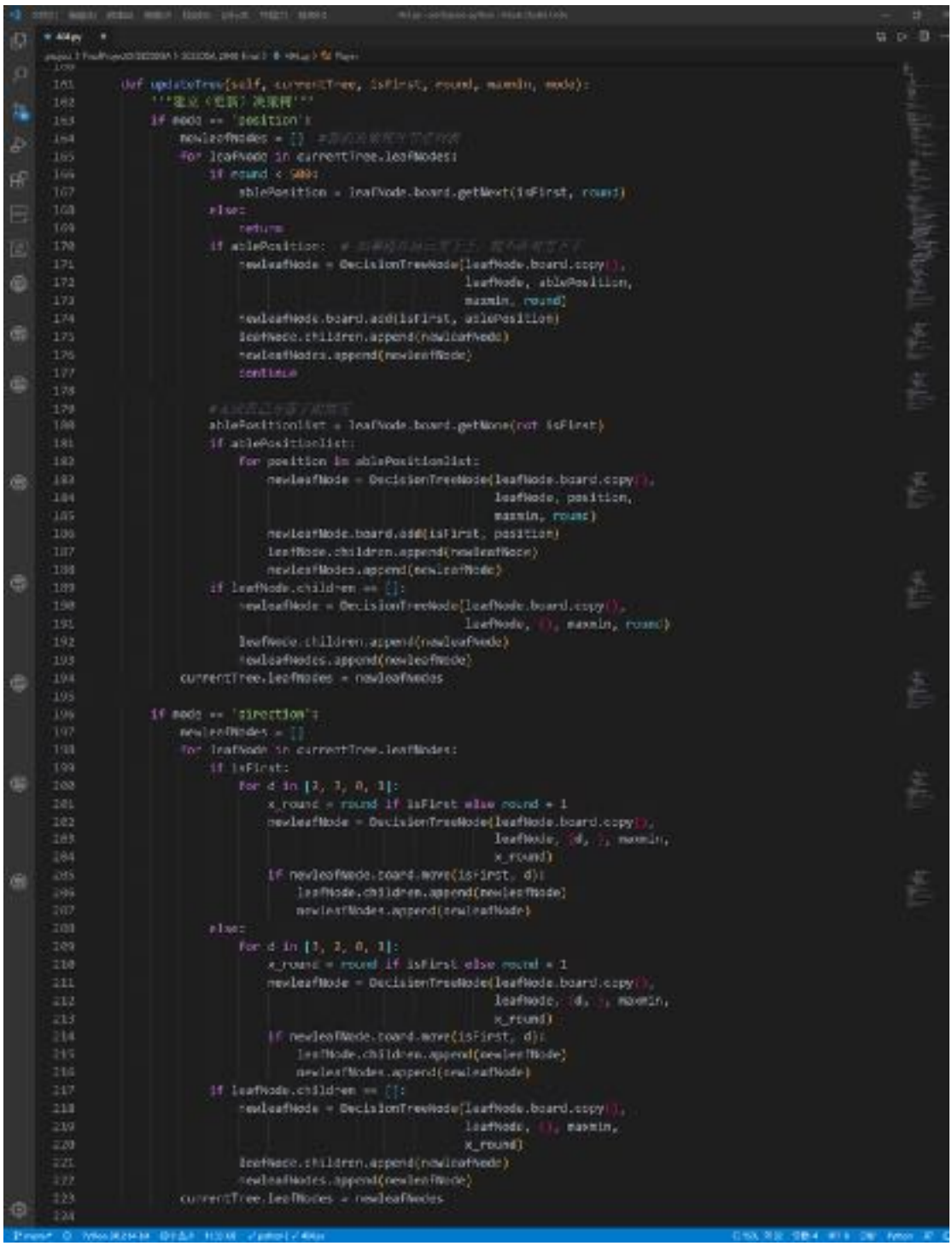


博弈树节点类设置了若干属性：
board:最主要属性，存储节点对应的棋盘对象（Chessboard）。
parent:父节点，即上一步对应的节点。
road:借用了图里边的概念，这里表示从父节点到现节点所用操作。
maxmin:下一步操作方。
round:回合数。
children:子节点列表，即经过一步操作可以到达的局面对应的节点的列表。
val:节点评分。

博弈树类除了根节点属性外，还设置了一个叶节点的列表。由与每次操作实际上相当于从博弈树叶节点向下探索一层，故在构建树的过程中维护这个叶节点的列表从而方便进行树的扩建。

2.2 函数说明

updateTree():



该函数功能是在传递进来树的现有基础上再扩建一层。采取的策略是在己方，对方尚有空位的情况下下棋环节只构建在各自方下棋的子节点（从对局中发现的普遍策略，可以理解成一次启发式剪枝,大大简化了博弈树）；另外在合并环节构建子节点的次序是优先左右方向，其次上下方向，（如图：2,3,0,1）左右方向进可攻，退可守，往往是产生较优局面的方向，如此建树可以优化 alpha-beta 剪枝的效果。

updateScore():

```
def updateScore(self, root):
    '''递归更新节点分数'''
    if root.children == []:
        #对于叶节点，直接调用价值函数计算分数
        root.val = self.value(root.board, root.round)
    else:
        if root.maxmin:
            #这一层为max层，更新alpha的值
            for child in root.children:
                self.updateScore(child)
                if child.val > root.alpha:
                    root.alpha = child.val
                if root.parent:
                    if root.alpha > root.parent.beta:
                        root.alpha = 1000000
                        break
            root.val = root.alpha
        else:
            #这一层为min层，更新beta的值
            for child in root.children:
                self.updateScore(child)
                if child.val < root.beta:
                    root.beta = child.val
                if root.parent:
                    if root.beta < root.parent.alpha:
                        root.beta = -1000000
                        break
            root.val = root.beta
```

该函数功能为对构建好的树的叶节点评估分数后向上传递。采用的是递归方法，DFS,过程中进行 alpha-beta 剪枝，对最后选择策略结果无影响的分支被砍去。

value():

```
def value(self, board, round):
    '''估值函数'''
    #上帝视角的是双方的棋子和棋盘及己方的不能下数
    myScoreList = board.getScore(self.isFirst)
    yourScoreList = board.getScore(not self.isFirst)
    l1 = len(myScoreList)
    l2 = len(yourScoreList)
    if l1 > l2:
        s = l1
        yourScoreList = [0 for i in range(l1 - l2)] + yourScoreList
    elif l1 < l2:
        s = l2
        myScoreList = [0 for i in range(l2 - l1)] + myScoreList
    else:
        s = l1
    val = 0
    for i in range(1, s + 1):
        val += (myScoreList[-i] - yourScoreList[-i]) / (i * i)
    c = 0.1
    # 空位得分
    blankPositions = board.getNone(self.isFirst)
    val += c * len(blankPositions)
    return val
```

该函数功能为根据回合数及棋盘状况对叶节点进行评分。为节约时间，最终版本的 value 采

用了较为简单的构造，主要考虑了两个因素。首先是盘面上双方子大小的比较，将双方对应位次的子大小作差并乘上相应权重，权重随排名靠后而减小（突出大子的重要性）。其次是对空位的赋分，这样可以增强合并性，改善拥堵情况（拥堵在这个游戏中是致命的）。空位的赋分大小通过实战进行了调参。

上述是最终的 value,最开始的 value 较为复杂，考虑的情况比较多，但是效果并不理想：

```
def value(self, board, currentRound=0):
    '''估值函数，综合考虑平滑性、单调性、空位数、最大数、孤岛，组成线性函数'''
    smooth_weight = 0.1  #平滑性权重
    mono_weight = 1.3  #单调性权重
    empty_weight = 2.7  #空格权重
    max_weight = 1.0  #最大值权重
    islands_weight = 2  #孤岛权重
    result = 0
    result += self.smooth(board, currentRound) * smooth_weight
    result += self.empty(board, currentRound) * empty_weight
    result += self.Max(board, currentRound) * max_weight
    result += self.mono(board, currentRound) * mono_weight
    result += self.islands(board, currentRound) * islands_weight
    return result
```

里面的五个函数：

```
def mono(self, board, currentRound=0):
    '''单调性的估值函数'''
    totals = [0, 0, 0, 0]

    #左右方向
    for i in range(4):
        current = 0
        nex = current + 1
        while nex < 4:
            while nex < 4 and board.getValue((i, nex)) == 0:
                nex += 1
            if nex >= 4:
                nex -= 1
            current_value = board.getValue((i, current))
            next_value = board.getValue((i, nex))
            if current_value > next_value:
                totals[0] += next_value - current_value
            elif next_value > current_value:
                totals[1]  = current_value - next_value
            current = nex
            nex += 1

    #上下方向
    for j in range(4):...

    return max(totals[0], totals[1]) + max(totals[2], totals[3])
```

```
def smooth(self, board, currentRound=0):
    '''局面平滑性的估值函数'''
    result = 0
    for i in range(4):
        for j in range(8):
            value = board.getValue((i, j))
            if value != 0:
                if j < 7 and board.getBelong((i, j + 1)) == self.isFirst:
                    right_value = board.getValue((i, j + 1))
                    result -= math.sqrt(abs(value - right_value))
                if i < 4 and board.getBelong((i + 1, j)) == self.isFirst:
                    down_value = board.getValue((i + 1, j))
                    result -= math.sqrt(abs(value - down_value))
    return result
```

```
def empty(self, board, currentRound=0):
    '''空位的估值函数'''
    #返回己方的空格数
    count = 0
    for i in range(4):
        for j in range(8):
            if board.getValue((i, j)) == 0 and board.getBelong(
                (i, j)) == self.isFirst:
                count += 1
    return count

def Max(self, board, currentRound=0):
    '''最大值的估值函数'''
    #返回己方的最大值
    mine = board.getScore(self.isFirst)
    if mine != []:
        return mine[-1]
    else:
        return 0
```

```
def islands(self, board, currentRound=0):
    '''判断孤岛并返回估值'''
    #返回连通个数
    count = 0
    direction = [(-1, 0), (1, 0), (0, 1), (0, -1)]
    for i in range(1, 3):
        for j in range(8):
            if board.getBelong((i, j)) == self.isFirst:
                for k in range(4):
                    if board.getBelong(
                        (i + direction[k][0],
                         j + direction[k][1])) == self.isFirst:
                        break
                else:
                    count += 1
    for j in range(1, 7):
        if board.getBelong((i, j)) == self.isFirst:
            if board.getBelong(
                (i, j - 1)) != self.isFirst and board.getBelong(
                    (i, j + 1)) != self.isFirst:
                count += 1
    return count
```

2.3 程序限制

在封闭修改前发现在天梯上突然出现报错，经查为搜索突破 500 回合限制，及时进行了修改。除此外，在天梯大量对局中并未发现错误，对于超时也有相应的避免机制。

3 实验结果

3.1 测试数据

实验环境说明：

- 硬件配置：
  - CPU：Intel（R）Core（TM）i5-8265U CPU @1.60GHz 1.80GHz
  - 内存：8.00GB（7.85GB 可用）
- 操作系统：Windows 10 版本 1909
- Python 版本：python 3.8.2

由于我们组的决策树和估值函数是分开来写的，因此测试分为两部分，一部分是对决策树的测试（改进前搜索 3~4 层，改进后搜索 7~8 层），另一部分是对估值函数的测试；两部分是同时平行进行的。

第一部分：对于决策树的测试。

由于在天梯上运行的速度更快，因此这一部分测试在天梯上进行。测试结果如下：

AI 名称	等级分	胜率
test03（改进前）	848.11	20.75%
chlwz03（改进后）	1062.38	50.89%

两者采用的估值函数是一样的，由此可以看出，搜索层数的加深可以大大提高胜率。

第二部分：对于估值函数的测试。

由于一开始的估值函数采用线性结构，即对棋盘平滑性、单调性、最大数、空位数、孤岛数的估值采取线性相加的形式，因此我们的测试仅限于更改前四种因素的权重。测试结果如下：

平滑性	胜率	单调性	胜率	最大值	胜率	空格数	胜率
0.1	32.63%	0.1	23.33%	0.1	31.11%	0.1	34.21%
0.2	30.00%	0.2	20.00%	0.2	41.11%	0.2	35.79%
0.3	36.32%	0.3	22.22%	0.3	33.33%	0.3	33.33%
0.4	35.79%	0.4	24.44%	0.4	43.33%	0.4	37.89%
0.5	34.21%	0.5	26.67%	0.5	45.56%	0.5	40.53%
0.6	35.70%	0.6	20.00%	0.6	44.44%	0.6	41.11%
0.7	37.89%	0.7	33.33%	0.7	44.44%	0.7	43.58%
0.8	40.53%	0.8	37.78%	0.8	38.89%	0.8	45.67%
0.9	37.89%	0.9	35.56%	0.9	38.89%	0.9	40.32%
1	37.37%	1	34.44%	1	34.44%	1	36.32%

使用标红的权重组成最终的估值函数，并与其他的测试 AI（多个）对战（两者的差别仅在于五个数的权重），前者的胜率总是保持在 40%以上，后者的胜率只有 20%左右。

但是当决策树改进完成之后，由于估值函数过于复杂，导致超时，因此我们减小了搜索层数（减至 3~4 层），并且和未减小层数的 AI（采用一种较为简单的估值函数）同时放到天梯上，结果如下：

AI 名称	等级分	胜率
test05（搜索层数少）	972.4	44.00%
chlwz03（搜索层数多）	1062.38	50.89%

由于时间限制，我们不得不舍弃了较为复杂的估值函数（test05），在对 chlwz03 的估值函数进行调整之后，在天梯上进行了测试，结果如下：

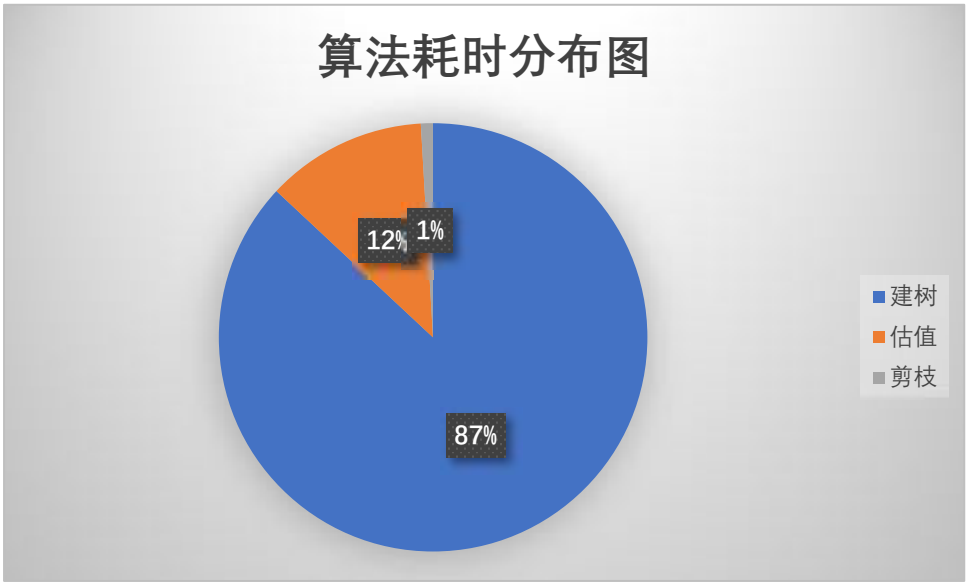
AI 名称	等级分	胜率
chlwz03	1062.38	50.89%
chlwz05	1045.36	59.16%

最终决定采用 chlwz03

第三部分：对于运行时间的测试。
在 userinterface 上进行测试，由于与天梯上的环境有所差别，代码在大约 200 回合时超时，最终各部分耗时结果如下：

搜索深度：6-9 层之间变动（根据当前回合数调整）。

用时：time1=4.01260s(建树耗时)， time2=0.55891s(估值耗时)， time3=0.04254s(剪枝耗时)



3.2 结果分析

可以看出，决策树的深度是影响胜率和等级分的主要因素。在建立 minmax 决策树时，我们采用了如下启发式规则：1、己方可以落子时，优先考虑己方落子。2、移动顺序设为【左、右、上、下】。再配合 α - β 剪枝，可以将搜索层数扩大至 8 层而不引起超时。

时间分析：主要开销集中在建树上，估值用时其次，剪枝效果较好，占用时间很少。
实际比赛中，代码基本可以坚持到 500 回合结束，与预期相符，不过若能进一步优化树结构，也许可以节约更多的时间以加大搜索的深度，同时腾出更多时间改进估值函数。

3.3 经典战局（可选）

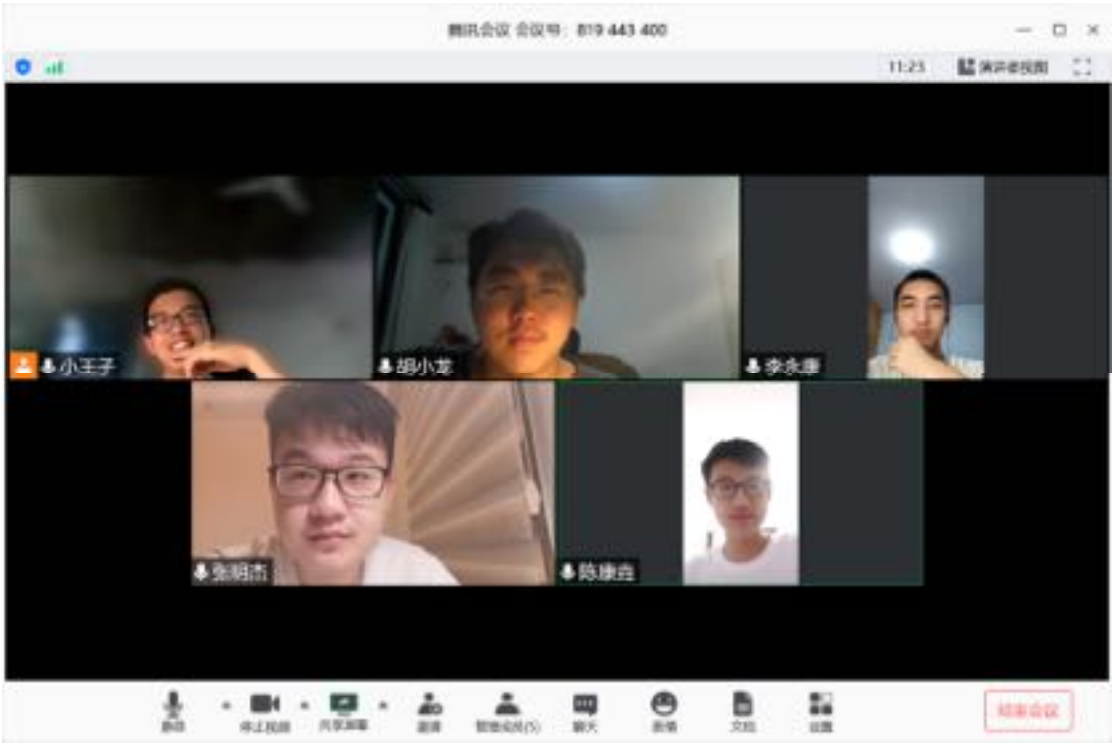
在我方（后手）对战其他小组代码时，出现了上图的局面：在第一张图中，对手只能向下或右合并。倘若向下，则吃掉我方的 512 之后又会被我方的 1024 吃掉；对手选择向右移动，

结果是 512 被我方吃掉。在估值函数中，我们加大了最大值的比重，以及“吃掉”对方棋子的得分比重。

4 实习过程总结

4.1 分工与合作

线上会议照片：



最后一次会议的合照↑

由于众所周知的原因，本学期的数算课程只能在网上进行，我们小组的讨论也全部在网上进行。相较于线下交流，线上会议给我们的讨论带来了更多的困难。

我们组的代码由陈康垚和李永康负责，王梓钰、张明杰、胡小龙参与对程序的调试以及构想启发式规则。我们先对 2048 游戏本身进行了更加深入的了解，并且在前 3 次的会议上确定了以决策树、剪枝加上参数（value 函数）和启发式规则的基本框架。在开始的时候，我们组对于 2048 对战游戏的理解有一些小失误，在随后的组会中我们及时发现并调整了算法，并将 2048 游戏本身的启发式规则加入到了算法之中。在代码构建完成之后，我们开始针对对抗性调整参数并且在热身天梯上对战观察效果。此外，我们在网上搜寻了很多资料，学习并将适合本竞赛的启示加入到了代码之中。

最后，我们组因代码实力不足，遗憾未能出线。我们简要讨论了我们的代码，并且交代好实习报告的工作之后便结束了本次竞赛中的合作。

4.2 经验与教训

本次大作业中，我们采用了课程所讲的决策树来编写代码。在这个过程中，我们加深了对于树这个结构的理解，可以更加熟练地用 python 来编写代码以实现算法。在 debug 工作完成后，我们的代码初现雏形。之后，我们及时发现了热身天梯上的离谱对局，并且针对这些对局中反映的 bug 优化了 value 函数。与上学期的《计算概论》课程的大作业相比较，这次的大作业需要更多的算法思想，并且需要更好地去优化算法。在这次的实习过程中，我们感受到了算法设计和修改过程中的艰辛，给我们留下了宝贵的经验。

毕竟我们止步 34 强，我们的算法不能谓之完全成功。在后来观战天梯的过程中，我们发现我们的代码相较分数更高的代码有着明显的劣势。最为神奇的是，在 5 月 30 日还与我们组不分伯仲的 007 组，竟然在最后杀入了 12 强（我也不知道他们做了什么样的优化）。我们组开始讨论大作业的时间较晚，实际讨论算法并编写出具有战斗力的代码的时间非常紧迫，这直接导致了我们组在最后痛苦的调参过程中无法做到最好。如果我们组能够再有一周的时间，我相信我们可以做得更好，只可惜时间不等人。我们的代码一直碍于时间不足，最后只能平衡决策树层数和参数，最终选择了增加决策树层数，但依然没有解决算法对抗性疲软的问题。

本次大作业，我们克服了线上交流不畅的困难，也很好地实现了我们的算法。但是遗憾的是，我们过短的准备时间和对于对抗性的理解不充分，让我们没能更好地优化剪枝并且提高代码的实力。

4.3 建议与设想

本次大作业直接暴露了线上讨论效果的参差不齐。我建议如果在之后的课程中无法进行线下讨论，可以适当地提早开放大作业，以给各个小组更加充分的准备时间。同时，我建议在推出大作业的同时制定一个进度参考表，以激励各个小组更加充分地利用这些时间已给出更好的算法。

由于在组队上不可避免地会出现实力差距，我希望以后的竞赛可以设计更加具有启发意义的游戏，让计算机基础不那么好的同学也能够用自己的想法，通过讨论和查阅资料来完成很好的算法。

我强烈建议优化天梯的匹配和积分机制：1、减少赢了之后却掉分的情况；2、提高实力相近的组之间匹配到的机率。

最后，欢迎明年的学弟学妹选修本课程，让我们一起体验用 python 实现算法与结构的乐趣。

5 致谢

感谢陈斌老师启发我们使用最大最小树算法和 Alpha Beta 剪枝实现 2048 的 AI！感谢助教团队开发出对用户非常友好的对战平台和对战程序！感谢所有组员的辛勤付出！

6 参考文献

https://towardsdatascience.com/2048-solving-2048-with-monte-carlo-tree-search-ai-2dbe76894bab 2048 蒙特卡洛算法实现 2048AI

https://stackoverflow.com/questions/22342854/what-is-the-optimal-algorithm-for-the-game-2048 What is the optimal algorithm for the game 2048?

https://blog.csdn.net/qq_32767041/article/details/80634244 2048AI 设计与实现（Minimax Search 和 Alpha Beta Pruning 实现 2048AI）

http://www.icourse163.org/learn/PKU-1003471009?tid=1450209444#/learn/announce 陈斌老师《人工智能与信息社会》，基于决策树的智能搜索部分

数据结构与算法课程实习作业报告

陈仲韬* 付皓竹 林维贤 邹景成 黄轩拓

摘要：建立决策树和估值函数是我们算法的核心。每次以当前的局面为根节点，以己方（或对方）上下左右移动四种情况为四个子节点，考虑五层，建立一个高度为六的决策树，然后通过估值函数和 minimax 算法选出综合起来对我方最有利的下法。代码在竞技场中表现良好，在天梯上最好成绩达到了第 13 名。

关键字：决策树　估值函数　minimax 算法

1 算法思想.....2
1.1 总体思路.....2
1.2 算法流程图.....4
1.3 算法运行时间复杂度分析.....5
2 程序代码说明.....5
2.1 数据结构说明.....5
2.2 函数说明.....6
2.3 程序限制.....11
3 实验结果.....12
3.1 测试数据.....12
3.2 结果分析.....12
3.3 经典战局（可选）.....13
4 实习过程总结.....15
4.1 分工与合作.....15
4.2 经验与教训.....17
4.3 建议与设想.....18
5 致谢.....18
6 参考文献.....18

1 算法思想

1.1 总体思路

我们算法的思路是建立关于 2048 的决策树，再通过估值函数和 minimax 算法选出综合起来对我方最有利的下法。

估值函数分别为 updatevalue 和 updatevalue2 两种。其中 updatevalue 更为全面，兼顾了对于盘面棋子值的总和的估算以及对于棋子进攻性与防守性的考虑。而 updatevalue2 则缺少了对于进攻性与防守性的考虑，仅对盘面棋子进行估值。关于两个估值函数中共同的盘面棋子估值部分，我们采取了如下思路进行实现：通过用户文档所提供的的 getScore（）函数来获取双方的棋子值的级数列表，并按照预先设计的对应计数字典进行赋值，将我方棋子按正数，同时对方棋子按照负数计入 total（updatevalue）或 score（updatevalue2）。关于计数字典的设计，考虑到比赛规则中最终比拼的并不是单纯的数值总和大小，而是优先最高级数大小，再考虑个数差异，所以为保证这条规则在 ai 的竞技中能够被考虑到，我们按照将每个级数所代表的数值设置为了 16 次方的相应指数。选择 16 次方也是代表着，一整个己方棋盘内的下级级数棋子都不如一个更大级数的棋子在计分中的作用更大。而关于具体进攻性和防守性的实现，主要是将己方被侵占或者己方侵占对方的棋子赋予了一个 16 倍的权重，将其重要性等同于一个更高级数的棋子以保证 ai 对于进攻和防守的重视。同时进攻性和防守性选用的 16 的权重和赋值字典的 16 次方，也都是在本地进行左右互搏之后验证，选择的已知尝试范围内的最优解。另外需要说明的是，因为 updatevalue 采用了用户文档中的 getRaw（）函数，导致其性能相较于 updatevalue2 比较低，所以我们在经过本地的循环测试以及对于对决的总结后，选择在前 400 局采用相对简单，耗时较少的 updatevalue2 函数而后 100 局则采用更全面也更耗时的 updatevalue 函数。同时在考虑向对方棋盘落子的情况中，也即应用 choose 函数的情况下，则全局应用 updatevalue 函数。

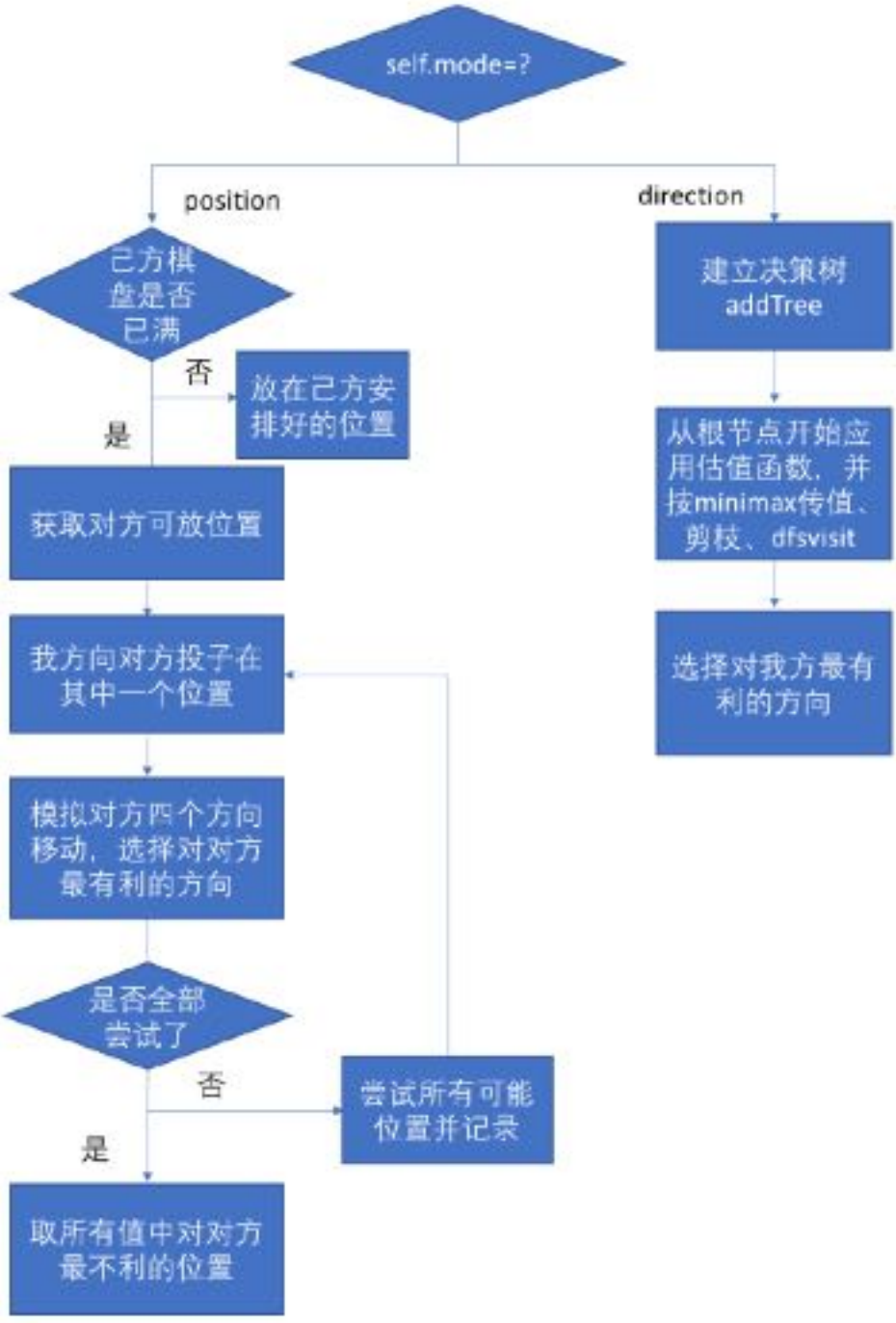
我们的决策树则采用四叉树结构讨论合并方向的最优解，并递归调用自身以达到加深搜索层数的目的。在建立四叉树的过程中除根节点外的子节点，都是经过最开始的合并加上顺序随先后手变化的我方落子，敌方落子和敌方合并之后的棋盘局面。对于树节点中的敌方落子与我方落子，我们都默认先在己方落子再在对方区域落子，并且树内节点的敌我双方互相在对面落子时为节省运算力，我们选择直接进行随机。而敌方合并的环节，我们选择进行一层四个方向的搜索选择有效且使得我方估值结果最小的方向（也即对于敌方最优的方向解）。

最后进行方向选择时，我们选择将每个有效方向的子节点和它本身的 value 值汇总到一起，并且统计它所代表的的子树的大小（size 值），其实就是节点数，并且相除得到平均，选择最大的。这样代表我们选择了最有可能获得更高棋盘分数的方向。对于主函数中的位置放置问题，依然是优先落子己方，但是在己方无处可放要向对面方向放置的时候，我们应用一个设计好的 choose 函数，对可落子位置进行遍历，检查落子后的有效合并方向，并选择合并后最小的值作为该位置的值，最后选取所有位置中值最大的，也就是在敌方最优走法情况下我方优势最大的落子位置，进行落子。

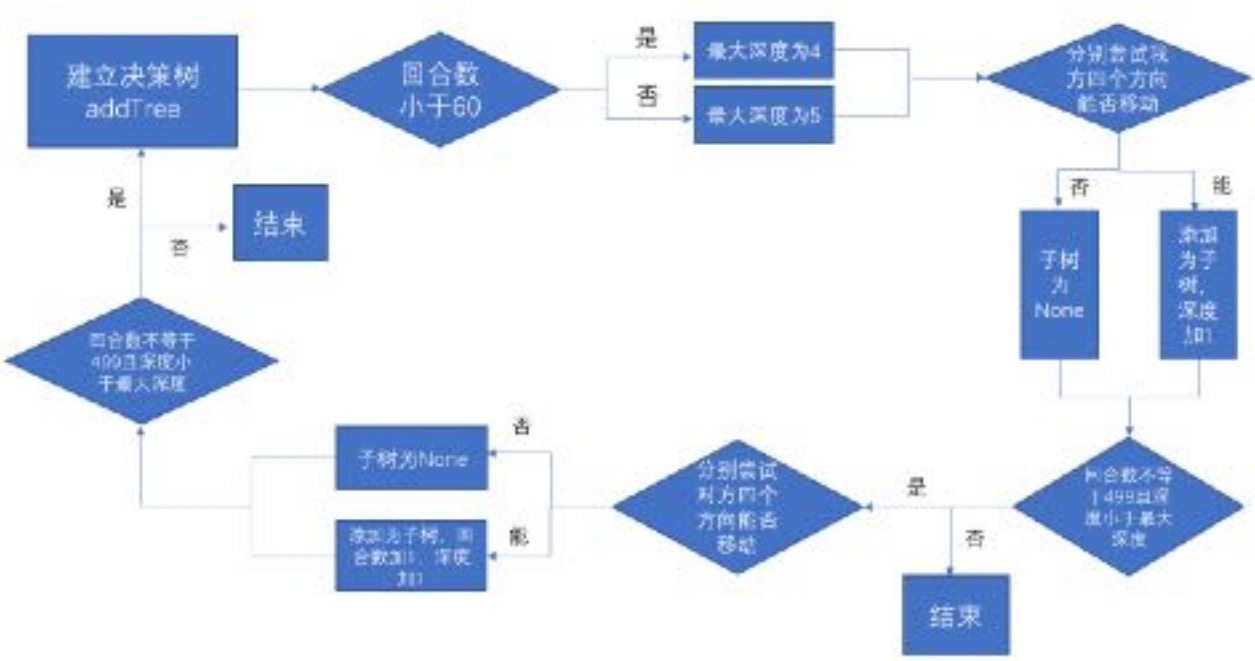
需要说明的是，在构建决策树的过程中，我们最先设定的是递归六层，但是在实战中我们发现很容易超时，改为递归四层所给时间又没有得到充分利用，在这种情况下我们最初是想通过优化我们的数据结构来解决这一问题，因此，我们设计了字典版本和列表版本的决策树，采用字典存储叶节点的方式，避免了深度优先访问。字典的 key 储存估值，value 储存历史移动方向记录，但是耗时并没有太大的改善，经过分析后，我们猜测原因是字典输出 dict.keys()同样需要遍历，列表取 max 又要 O(n)的时间复杂度，结果改善就不大。为了解决这一问题，在向其他组大佬请教后，我们决定将递归层数设置为五层，这与我们以前的认识是相悖的，因为一回合要建两层树，我们以为必须整回合地建树，即递归层数应该是偶数层，但从其他组哪里我们意识到递归五层其实是可以的。于是改变递归层数后，我们的算法立即得到了明显的提高，在天梯上提高了四百分，因此形成了我们的最终版本代码。

1.2 算法流程图

总图：


self.mode=?
position
direction
己方棋盘是否已满
否
放在己方安排好的位置
是
获取对方可放位置
我方向对方投子在其中一个位置
模拟对方四个方向移动，选择对对方最有利的方向
是否全部尝试了
否
尝试所有可能位置并记录
是
取所有值中对对方最不利的位置
建立决策树addTree
从根节点开始应用估值函数，并按minimax传值、剪枝、dfsvisit
选择对我方最有利的方向

决策树（以我方先手为例）



（注：流程图中省略了投放棋子的部分，在代码中写了）

1.3 算法运行时间复杂度分析

我们算法的时间开销主要花费在建立决策树和对决策树进行深度优先搜索上，由于建立决策树和进行深度优先搜索时每次都涉及到上下左右四个方向，因此估计我们算法的时间复杂度是 O（4n）。经过测试我们得到的数据是：搜索 4 层时，500 回合用时 1s 左右，5 层时用时为 4s 左右，6 层时就完全超时，预测用时 16s。

2 程序代码说明

2.1 数据结构说明

在最终版代码中，我们主要用到了树这一数据结构，由于对战中涉及到上下左右四种情况，我们在课堂上讲的二叉树的基础上进行了扩展，建立了四叉树。

2.2 函数说明

1. moveaction:根据 mode 返回决策

```
def moveaction(self):
    if self.mode == "position":
        myPosition = self.board.getNext(self.isFirst, self.currentRound)  # 我方被允许的位置
        Empty = self.board.getNone(not self.isFirst)  # 对方可下空位
        if myPosition != ():
            return myPosition
        elif Empty != []:
            position = self.choose(Empty, self.board)  # 从对方可下空位中选取落子位置
            return position
        else:
            return
    if self.mode == "direction":
        action = self.buildTree()  # 建树并返回决策方向
```

2. choose: 选择对方棋盘内的落子位置

```
def choose(self, position_list, board):  # 选择对方棋盘内的落子位置
    position_dict = {}  # 位置字典
    for i in position_list:
        valuedict = {}  # 价值字典
        for j in range(4):
            copy = board.copy()
            # 选择可移动方向并在价值字典中记录其方向和移动后估值
            if copy.move(not self.isFirst, j): valuedict[j] = updatevalue(copy, self.isFirst)
        if valuedict != {}:
            position_dict[i] = min(valuedict.values())  # 选取最小估值（敌方最优）
    if position_dict != {}:
        position = []
        for _ in position_dict.keys():
            # 选取最小中的最大（敌方最优情况下对我方最有利的选择）
            if position_dict[_] == max(position_dict.values()):
                position.append(_)
        return random.choice(position)
    else:
        return random.choice(position_list)
```

3. buildTree:建立树并返回决策方向

```
def buildTree(self): # 建立树，并返回决策方向
    root = TreeNode(True, None, self.board, self.currentRound, self.depth)
    decision = DecisionTree(root, self.isFirst)
    decision.addTree(decision.root) # 建立整个决策树
    self.dfsvisit(decision.root) # 从根节点开始，深度优先访问，并传递 value
    scoredict = {} # 用于记录方向对应的 value
    up, down, left, right = decision.root.upTree, decision.root.downTree, \
                            decision.root.leftTree, decision.root.rightTree
    if up != None:
        scoredict[up.value] = 0
    if down != None:
        scoredict[down.value] = 1
    if left != None:
        scoredict[left.value] = 2
    if right != None:
        scoredict[right.value ] = 3
    if scoredict == {}:
        return
    else:
        maxv = max(scoredict.keys())
        return scoredict[maxv]
```

4. addTree:按照给定最大层数建立子树

```
def addTree(self, currentNode):
    if currentNode.currentRound < 60:
        maxdepth = 4
    else:
        maxdepth = 5
    for i in range(4):
        copy_board = currentNode.board.copy()  # 深度拷贝棋盘
        if copy_board.move(self.isFirst, i):  # 尝试己方移动棋盘,如棋盘变化，合并有效
            childtree = TreeNode(False, currentNode, copy_board, \
                                 currentNode.currentRound, currentNode.depth + 1)

            if i == 0:
                currentNode.upTree = childtree
            elif i == 1:
                currentNode.downTree = childtree
            elif i == 2:
```

```
                currentNode.leftTree = childtree
            else:
                currentNode.rightTree = childtree


            if currentNode.currentRound != 499 and \
               childtree.depth < maxdepth:
                for j in range(4):
                    copy_board = childtree.board.copy()  # 深度拷贝棋盘
                    if self.isFirst:
                        if copy_board.move(not self.isFirst, j):  # 尝试对方移动棋盘,如棋盘变化，合并有效
                            copy_board = self.myset(childtree, copy_board)
                            copy_board = self.hisset(childtree, copy_board)
                            childchildtree = TreeNode(True, childtree, copy_board, \
                                                      childtree.currentRound+1, childtree.depth + 1)
                        else:
                            childchildtree = None
                    else:
                        copy_board = self.hisset(childtree, copy_board)
                        copy_board = self.myset(childtree, copy_board)
                        if copy_board.move(not self.isFirst, j):  # 尝试对方移动棋盘,如棋盘变化，合并有效
                            childchildtree = TreeNode(True, childtree, copy_board, \
                                                      childtree.currentRound+1, childtree.depth + 1)
                        else:
                            childchildtree = None
                    if childchildtree != None:
                        if j == 0:
                            childtree.upTree = childchildtree
                        elif j == 1:
                            childtree.downTree = childchildtree
                        elif j == 2:
                            childtree.leftTree = childchildtree
                        else:
                            childtree.rightTree = childchildtree

                        if currentNode.depth + 2 >= maxdepth or \
                           childchildtree.currentRound >= 499:
                            pass
```

```
                        else:
                            self.addTree(childchildtree)
```

5. dfsvisit：用于深度优先访问、传递 value、alpha beta 剪枝

```
def dfsvisit(self,treeNode):
    '''''新增函数，用于深度优先访问、传递 value、alpha beta 剪枝'''
    treeNode.color = 'grey'
    for childNode in treeNode.getChildren():
        if childNode.color == 'white': # 结束条件
            self.dfsvisit(childNode) # 递归调用
    if treeNode.isLeaf():
        # 是叶节点，则应用估值函数
        if self.currentRound < 400:
            treeNode.value = updatevalue2(treeNode.board,self.isFirst)
        else:
            treeNode.value = updatevalue(treeNode.board,self.isFirst)
    else:
        # 不是，则传递 value
        childTreeValue = []
        for childNode in treeNode.getChildren():
            childTreeValue += [childNode.value]
        if treeNode.type: # 如果是 max 节点
            treeNode.value = max(childTreeValue)
        else:             # 如果是 min 节点
            treeNode.value = min(childTreeValue)

    if treeNode.parent:  # 如果有父节点，上传 alpha、beta 值

        if treeNode.type: # 如果是 max 节点，父节点为 min 节点，上传 beta
            treeNode.parent.beta = treeNode.value

            if treeNode.parent.parent: # 如果有父父节点
                if treeNode.parent.parent.alpha: # 如果父父节点有 alpha 值
                    if treeNode.parent.parent.alpha >= treeNode.value:
                        # 与该子节点有共同父节点的节点无需探索。
                        for childNode in treeNode.parent.getChildren():
                            childNode.color = 'grey'

        else:             # 如果是 min 节点，父节点为 max 节点，上传 alpha
            treeNode.parent.alpha = treeNode.value

            if treeNode.parent.parent: # 如果有父父节点
                if treeNode.parent.parent.beta: # 如果父父节点有 beta 值
```

```
                    if treeNode.parent.parent.beta <= treeNode.value:
                        # 与该子节点有共同父节点的节点无需探索。
                        for childNode in treeNode.parent.getChildren():
                            childNode.color = 'grey'
```

6. getChildren: 返回子树构成的列表

```
def getChildren(self):
    '''''新增函数，返回子树构成的列表'''
    childrenList = []
    if self.upTree:
        childrenList += [self.upTree]
    if self.downTree:
        childrenList += [self.downTree]
    if self.leftTree:
        childrenList += [self.leftTree]
    if self.rightTree:
        childrenList += [self.rightTree]
    return childrenList
```

7. updatevalue: 估值函数

```
def updatevalue(board, belong):  # 后期使用，进攻型估值
    chessdict = {0 : 0, 1: 16, 2: 256, 3: 4096, 4: 65536, 5: 1048576, 6: 1677721
6,
                 7: 268435456, 8: 4294967296, 9: 68719476736, 10: 1099511627776}

    total = 0
    list_board = board.getRaw()
    if belong:
        for i in range(4):
            for j in range(4):
                if list_board[i][j][1] is not belong:
                    total -= 16 * chessdict[list_board[i][j][0]]
                else:
                    total += chessdict[list_board[i][j][0]]
                if list_board[i][4 + j][1] is belong:
                    total += 16 * chessdict[list_board[i][4 + j][0]]
                else:
                    total -= chessdict[list_board[i][4 + j][0]]
    else:
        for i in range(4):
            for j in range(4):
                if list_board[i][j][1] is belong:
```

```
                    total += 16 * chessdict[list_board[i][j][0]]
                else:
                    total -= chessdict[list_board[i][j][0]]
                if list_board[i][4 + j][1] is not belong:
                    total -= 16 * chessdict[list_board[i][4 + j][0]]
                else:
                    total += chessdict[list_board[i][4 + j][0]]
    return total

def updatevalue2(board, belong):  # 前期使用，保守型估值
    chessdict = {1: 16, 2: 16 ** 2, 3: 16 ** 3, 4: 16 ** 4, 5: 16 ** 5, 6: 16 **
6,
                 7: 16 ** 7, 8: 16 ** 8, 9: 16 ** 9, 10: 16 ** 10, 11: 16 ** 11}

    myscore = board.getScore(belong)
    hisscore = board.getScore(not belong)
    score = 0
    for i in myscore:
        score += chessdict[i]
    for j in hisscore:
        score -= chessdict[j]
    return score
```

2.3 程序限制

在最开始建立决策树的时候因为各种大小 bug，导致我们的决策树一直并不算完全成功，直到将各种 bug 都修复之后，我们才尴尬的发现了问题。由于没有任何优化措施，我们的程序在很低的层数下就会超时，当然也有一些原因是早期的估值函数选用了 numpy 库进行了过于全面的估值以致于超时。在后期逐步的开发中，我们通过调整决策树建立数据基础，增加剪枝以及更替估值函数来尽力避免超时的问题，并且确保不会出现超出对局数而报错等问题的 bug。只是由于我们能力的限制，即使到了比赛末期，也依然未能为我们的 ai 添加什么有效的先验知识来提升其性能，所以尽管我们的程序在时间方面有着很好的表现，实际的性能依然较天梯榜前列的小组有着很大的差距。

3 实验结果

3.1 测试数据

实验环境说明：

- 硬件配置：i7-8765 CPU @ 1.80GHz 1.99GHz 内存： 8G
- 操作系统：windows 10
- Python 版本：Python 3.7

在写出决策树和新旧估值函数后，我们注意到算法的运算时间会大幅度增加，为了减少用时，我们写了三个版本的决策树，分别是 minimax 版本、字典版本和列表版本，分别与新旧估值函数进行组合，以实验哪一个版本在兼顾到时间限制的情况下能够得到较好的结果。

3.2 结果分析

代码版本	实验过程	实验结果
Minimax 版本	前 60 回合预演 4 层，之后预演 5 层，采用双估值函数。	稳赢 botA，最好情况可到 9:1 与 botS 经常平局，最好 7:3
字典版本	预演 4 层、考虑对方移动时采用贪心策略、采用更全面的估值函数	打 robot-B 比分（我方）4:6，超时三次 打 robot-A 比分（我方）3:7，超时两次。
列表版本	预演 4 层、考虑对方移动时采用贪心策略、采用更全面的估值函数	打 robot-B 比分（我方）4:6，超时四次 打 robot-A 比分（我方）0:10，超时两次。

本次实验让我们意识到改变我们算法的结构，意图以此节省深度优先访问的时间是不可行的，正确设计递归层数才是王道，可以说是为我们指明了方向。算法的主要运行时间开销在于建立决策树和进行深度优先搜索的过程。

3.3 经典战局

在开发小组程序过程中，我们发现，在最初不考虑进攻与防守（即为越界吞噬进行特殊赋值）的情况下，我们的程序往往会被不断地偷子，进而在初期就逐渐被拉开差距，导致后期无力回天。在加入对进攻和防守的考虑之后，我们的程序成功的在对局中吞噬对方的 2048 进而取得胜利，但是同时，在面对更加强大的 ai 时，往往会因为进攻性，而被诱敌深入，可能是最终导致我方大数被卡在边界最终无奈被吞噬，又或者是我方较大数在吃点对方的诱饵之后被对方的更大数吞噬（大鱼吃小鱼）。我们小组的经典战局主要源于大数互吃，并未能够出现决赛直播中全局压胜最终报错自杀的局面（真是可惜呢~摊手）。总结：由于程序的局限性，往往不能够设计出精妙的陷阱进行吞噬，许多压胜局面还是有一点莽的，但是也是有 ai 提前计算的因素，只是这方面也是我们和顶尖 ai 的差距所在，有很大的提升空间。

A. 经典胜利 http://gis4g.pku.edu.cn/ai_arena/match/nA_ZUGK2Ik/6

我方对战 F19 victory，6 ： 4。本局中我方为先手。

在第 257 回合，我方 AI 本可以向下移动合并自己的 1024，却选择优先吃对方的 1024，这显示了我们的估值函数的合理与可靠：这样即增加了我方的分数，又减小了对方的分数。

在这场比赛的最后，也是重新吞并了之前对方侵占到我们这来的棋子，还占领了对方的大块棋盘，这是因为在大于 400 回合时我们对估值函数进行了微调，使代码的防御性和攻击性加强了。

这是刚进入 400 回合时的情况，被占领很多。

然后很快反击，最后占领对方的棋盘。

B. 经典失败 http://gis4g.pku.edu.cn/ai_arena/match/xC8lqJsPxa/7

我方对战 F19 119，0 ： 10。本局中我方为后手。

为何这是一场经典失败呢？是因为我们最开始的小数不停的被偷，导致和数速度跟不

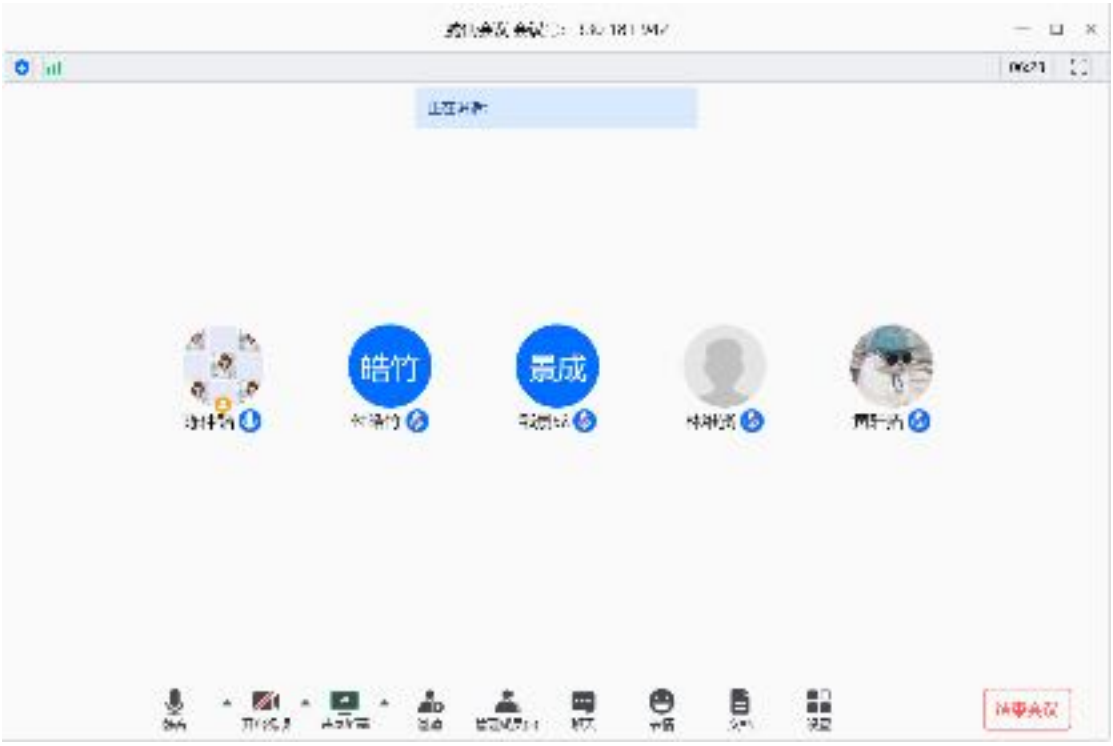
上，最后也是因为对方把较小的数挤到了我们的棋盘里面，和数效率大大降低。这是因为我们的估值函数采用指数式，对小数的重视程度不足，导致前期不断被偷。后期即使加强了防御性和攻击性，但是这部分附加的分还是完全不能与大数合并相比，在忙着合并的时候被小数乘虚而入。可见，面对进攻性较强的代码我们组会输的很惨，这一场更是被剃了光头。

4 实习过程总结

4.1 分工与合作

我们小组的分工是陈仲韬、付皓竹负责决策树算法的构想与实现，邹景成、林维贤负责估值函数的构想与实现，黄轩拓负责会议记录和大作业报告的书写。

我们小组的合作与交流方式是每周三、周日通过腾讯会议进行线上的相互交流、讨论，分享自己的算法与遇到的问题。除此之外，我们每天都将自己大作业的成果上传到腾讯文档中，通过腾讯文档的共享功能让每个人都能够及时了解小组取得的成果，打破了因疫情导致的不能面对面交流的难题。同时，我们小组的专属微信群也是我们交流的重要方式，自己的想法、与大作业有关的通知我们都会发布在微信群里进行交流。



腾讯会议 会议号：696 767 388
05:05
陈仲韬
村皓竹
林继贺
黄轩拓
结束会议

数算期末大作业小组(5)
我的一层是含有四个操作的
就是从我的合并开始
直到下一次自己的合并
那样可以 minimax 将
对啊
再写一点剪枝
为啥我还是没特别清楚，这个 minimax
1. 先遍历一遍四种方向
2. 在每一种方向里面
遍历敌人的四种方向
我写了 minimax
超时超得太厉害
可能是剪枝失败了

数算期末大作业小组(5)
大家对决策树有什么建议多聊一聊啊还有 minimax 怎么实现，有想法的讨论讨论
修复了一个大 bug 反而更容易超时这件事确实心痛
唯一的意义好像就是打白痴打的更开心了
我们还要继续优化
我准备从先验知识入手，看看吧，顺便试一试动态改变建树情况
更新了文档：列表版本的决策树。三层仍然会超时，我自闭了
继续加油
我觉得树的结构已经优化到极限了
怀疑是建树的时候或者其他地方耗时了。。。

在最开始，我们的代码效果并不是很好，由于将递归层数设置为六层，频频出现超时的情况，在多次协商并请教了其它小组后，我们决定把递归层数设置为五层，超时的问题得以解决。但在观察之后的对局中，我们发现我们的算法在前期往往具有较大的优势，但在后期由于缺乏进攻性而经常出现被反超的情况，为此我们决定增强我方算法在后期的进攻性。我们在经过本地的循环测试以及对于对决的总结后，选择在前 400 局采用相对简单，耗时较少的 updatevalue2 函数而后 100 局则采用更全面也更耗时的 updatevalue 函数。同时在考虑向对方棋盘落子的情况中，则全局应用 updatevalue 函数。通过两种估值函数来增强我方在后期进攻性的策略一度取得了比较好的效果，在一晚上我们的算法在天梯上多得了四百多分，在排行榜上最高达到第十三名，但是最后还是惜败。

4.2 经验与教训

经验：

这次数算大作业虽然我们一直坚持到了最后也没能出线，但是，在我看来这一整个过程是十分有收获的。从最开始的分工开会，到之后的熬夜编写第一次树的结构，从被白痴 ai 打败到开始完虐白痴乃至最后甚至可以暴打官方 botA。可能对于很多非常巨的小组来说这都不算是什么成就，但是对于我这个菜鸡来说，每一次看着我们的 ai 有所成长都非常开心。尽管我们在 ai 的研发过程中有许多不足的地方，比如：估值函数的很多数据是凭感觉来的，又或者是通过大量的比赛然后人肉调参这种在比赛末期才醒悟的低效率方法，但是我依然认为我们在这之中是有所收获的。从单纯的节点链接法到用列表和字典去实现决策树，从始终无法使用的 minimax 到最终为我们带来了最后一搏的剪枝，估值函数也在不断调整修改以适应新的决策树，以满足运算的时间标准，在这之中我们都有着自己的成长。希望如果再有类似的任务，我可以更有行动力一点，也可以多找大佬们学习交流，长长见识，不要低效工作；也希望自己可以有更好的查文献的方法，能够帮助自己通过学习更快地提升自己；最重要的，是希望自己有一个好的数理基础，可以“遇山平山，遇海填海”。

教训：

1. 决策树增加层数才是硬道理，事实表明递归四层和递归五层的胜率是真的不一样，我们在为了减少时间开销上采用了三种版本进行比较还是浪费了不少时间。
2. 进攻有利于获胜，既可以让己方的子增长，又可以占据敌方一个格子，起到打乱对方布

局的作用，我们的算法在进攻性上感觉还是不够强，这是值得我们思考和改进的地方。

4.3 建议与设想

1. 2048 作为大作业还是十分有趣的，也确实对我们的编程能力有很大的提升，但感觉天梯出现队伍还是少了一点，建议由三分之一出线改为二分之一出线。
2. 在决赛的过程中，我们只能看到每队总的比分，希望以后的决赛能够可视化，让比赛的过程呈现在我们眼前。
3. 对学弟学妹的寄语：在陈老师的数算课上你真的能够学到很多东西，对自己的能力也有很大的提升，下一届的学弟学妹们如果是巨佬的话很推荐你们选这门课，但是如果相对基础薄弱的话，选这门课就要深思熟虑了，希望要有一个挑战自己的勇气，只要能够坚持下来，一定能够看见一个全新的自己！

5 致谢

1. 感谢陈斌老师本学期对我们的指导，感谢助教对我们疑问的及时解答，感谢技术组的同学维大作业顺利进行所做出的贡献。
2. 感谢 echo 小组为我们提供的宝贵经验
3. CSDN（面向 CSDN 编程石锤（虽然其实最后也没能用上查到的双链表结构 允悲））
4. 也感谢我们小组中每一位成员为大作业付出的努力。

6 参考文献

[1] 2048 项目文档：https://gitee.com/chbpku/sessdsa.2048/
[2] 代码竞技场：http://gis4g.pku.edu.cn/ai_arena/home/
[3] 顾小豆. 数据结构之通用树结构的实现：
https://blog.csdn.net/u014754841/article/details/79330826
[4]_ocean.2048 游戏的最佳算法：
https://blog.csdn.net/ocean0815/article/details/25748037?utm_source=blogxgwz6
[5] weixin_30627341.2048-AI 程序算法分析：
https://blog.csdn.net/weixin_30627341/article/details/96913829

数据结构与算法课程实习作业报告

郑若楷 郑爽 雷天琦 王晨旭 *赵时

摘要：在移动决策中，我们采用 minimax 算法，在执行决策的每一步都对后五步的局势进行模拟，从而找到最利于我方的决策，构成决策树，并进行$\alpha\beta$剪枝以减小算法时间复杂度。

关键字：minimax 算法、决策树、剪枝算法、深度优先搜索

1 算法思想.....1
1.1 总体思路.....1
1.2 算法流程图.....2
1.3 算法运行时间复杂度分析.....3
2 程序代码说明.....3
2.1 数据结构说明.....3
2.2 函数说明.....4
2.3 程序限制.....12
3 实验结果.....12
3.1 测试数据.....12
3.2 结果分析.....14
3.3 经典战局.....15
4 实习过程总结.....15
4.1 分工与合作.....15
4.2 经验与教训.....17
4.3 建议与设想.....19
5 致谢.....19
6 参考文献.....19

1 算法思想

1.1 总体思路

本算法把落子和移动分开。由于落子的选择较多，不便于向后搜索，因此对落子的选择依据检测我们深思熟虑概括出的七种典型情形，并对可落子位赋予相应的权重进行平衡选择。

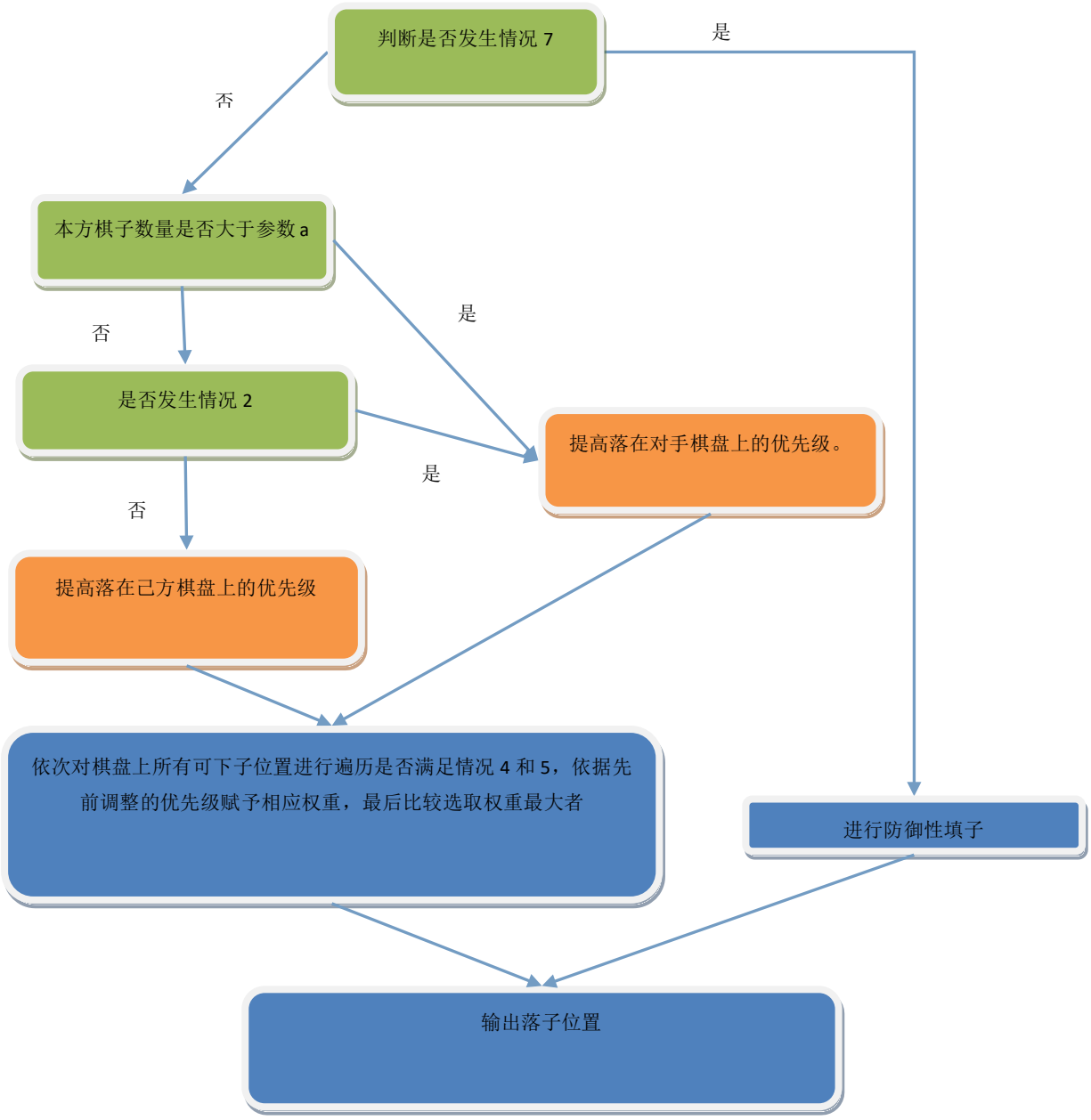
在移动函数中，我们采用 minimax 算法，让算法模拟从此步移动开始后五步敌我落子和移动的相关情况，得到一棵高度为 5，有 1024 个对应后五步可能情况的叶的决策树。对于每一种情况，ai 结合此时场上的敌我最大值、在一系列合并中我方和敌方互相吞并的值、五次移动种我方和敌方各自合并的值、在仅仅移动一次后敌我最大值等局势信息，对每一种情况用估值函数赋以权重。回溯时，轮到我方决策即选择使此权重最大，每次敌方决策使此权重最小，直至返回第一步，得到五步预测下我方应该执行的、可能对我方最有利的决策。对于往

后走可能出现的敌我四个方向都不能移动的情况，也均进行分类讨论，保证权重对各种情况均一致、有效。在无明显优势方向的情况下，我们采用“莽夫策略”，即以往敌方棋盘移动优先，以期尽可能吞并对方棋子，占据场地优势。

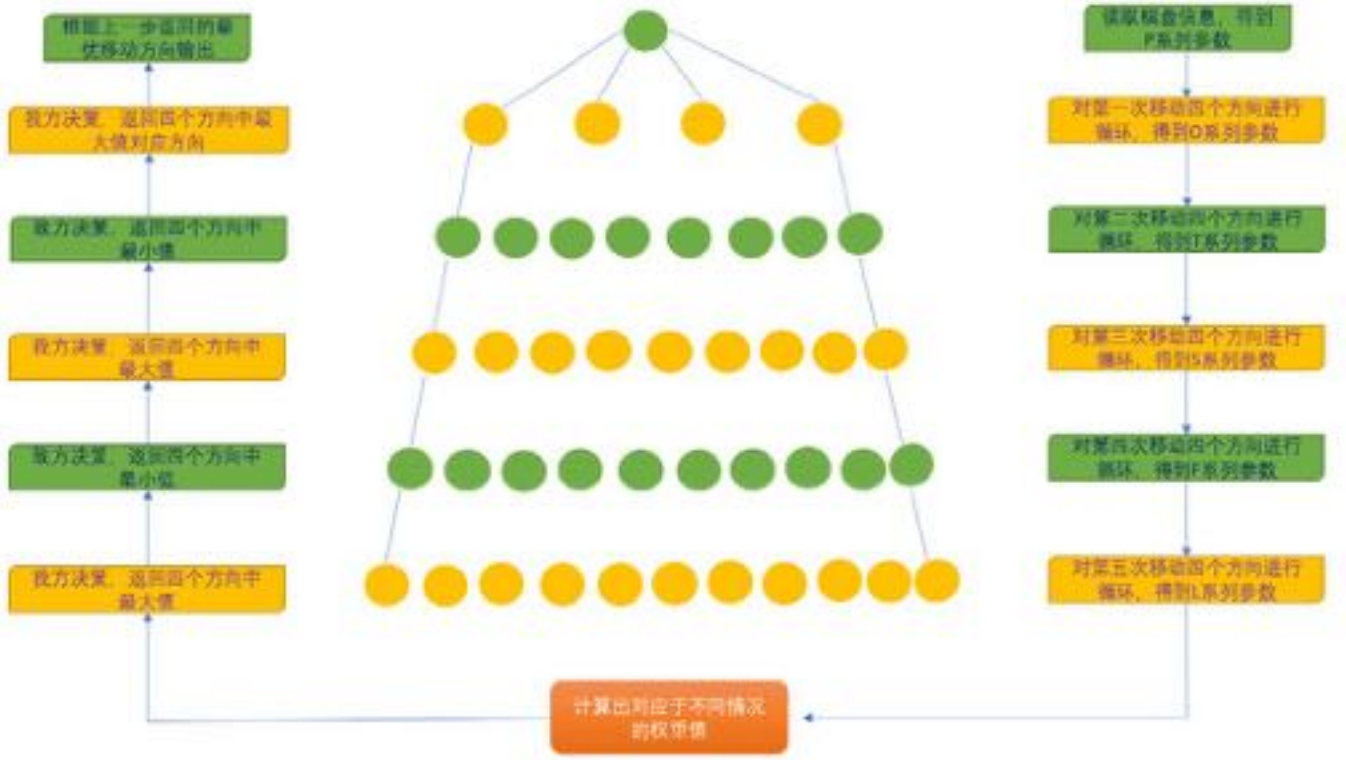
我们小组认为，**此算法的优劣主要依据的是往后迭代的深度**，因此我们采用$\alpha\beta$剪枝减去不必要的枝叶，将算法的深度从两层一步步提高到了五层，并尽可能在各种情况减少时间。最终的代码兼顾了算法的竞争力和运行时间，大部分情况下均比较稳定，只在少数情况出现超时现象。

1.2 算法流程图

落子函数：



移动函数：



1.3 算法运行时间复杂度分析

设棋盘上的总棋子数为 n；在每次移动中，均调用 getScore 函数返回排序的敌我棋子，平均时间复杂度是 O(logn)，而进行移动的 move 函数大致是 O(n)的数量级，每次移动有四个方向，一般来说进行了 4 的五次方即 1024 次尝试，因此移动函数的每次调用时间复杂度大致为 O(1024n)。考虑到剪枝算法，估计真实时间大约在 O(700n)。
实际运行中，算法在本机跑的大约时间为 7-8s。在网站上由于进行加速，大约时间在 4.5s 左右，在局面极其复杂的特定情形下会出现超时现象。

2 程序代码说明

2.1 数据结构说明

本组代码主要运用列表、元组、字典等基本结构，同时隐式的采用递归的方式运用了树的概念。运用列表对树的叶的值进行存储和排序，通过元组存储可能的移动方向和权重值。我们采用 minimax 算法和剪枝算法，用递归函数进行深层的探索；并在找到最优策略后沿树结构回溯，最终输出策略。

2.2 函数说明

本代码由两部分组成，一部分是处理落子情况的落子代码，一部分是执行移动的 minimax 算法，同时在 minimax 算法中也对落子函数进行了调用。

1.落子函数主要由三部分函数构成：

（1）两个用于辅助判断各种情况的函数

①findnearst 函数：通过传入的棋盘，位置和方向，返回此位置沿此方向距离最近的棋子信息（归属 belong，价值 value，位置 position），如果没有返回 None。一开始，我们采用的是直接判定某位置上下左右相邻位置的棋子，但这样显然不严谨，因为合并是一条直线进行的，因此定义 findneast 函数用于寻找最近的棋子。

```
def findnearest(self, position, direction, board):
    x = position[0]
    y = position[1]  # x, y为position坐标

    if direction == 0:  # 向上找直到遇到第一个getValu
        if x == 0:
            return None
        else:
            find = False
            i = 1
            while i <= x and not find:
                if board.getValue((x - i, y)) != 0:
                    find = True
                    pos = (x - i, y)
                    val = board.getValue(pos)
                    bel = board.getBelong(pos)
                i += 1
            if find:
                return [bel, val, pos]
            else:
                return None
```

```
elif direction == 1:  # 向下找直到遇到第一个getVa
    if x == 3:
        return None
    else:
        find = False
        i = 1
        while i <= (3 - x) and not find:
            if board.getValue((x + i, y)) != 0:
                find = True
                pos = (x + i, y)
                val = board.getValue(pos)
                bel = board.getBelong(pos)
            i += 1
        if find:
            return [bel, val, pos]
        else:
            return None
```

```
elif direction == 2:  # 向左找直到遇到第一个getV
    if y == 0:
        return None
    else:
        find = False
        i = 1
        while i <= y and not find:
            if board.getValue((x, y - i)) != 0:
                find = True
                pos = (x, y - i)
                val = board.getValue(pos)
                bel = board.getBelong(pos)
            i += 1
        if find:
            return [bel, val, pos]
        else:
            return None
```

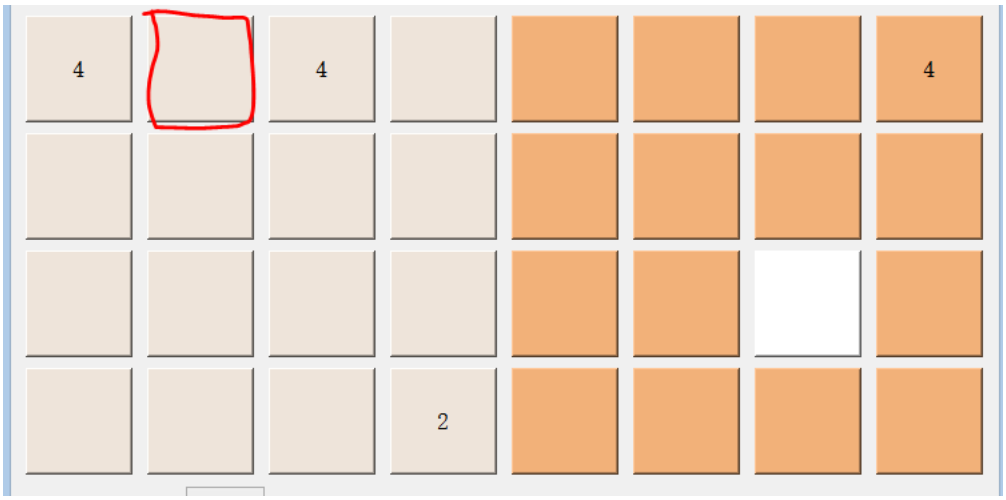
```
else:  # 向右找直到遇到第一个getValue不为0，否则
    if y == 7:
        return None
    else:
        find = False
        i = 1
        while i <= (7 - y) and not find:
            if board.getValue((x, y + i)) != 0:
                find = True
                pos = (x, y + i)
                val = board.getValue(pos)
                bel = board.getBelong(pos)
            i += 1
        if find:
            return [bel, val, pos]
        else:
            return None
```

②isborder 函数：通过传入的棋盘给出己方最深入对方的棋子集合 dangerList，因为这些棋子最容易被吃掉，用于下面形式判断函数中。

```
def isborder(self, board):  # 给出一方处于边界地区的棋子，以列表形式返回棋子的位置，若没有棋子属于边界
    border = []
    wes = [0, 1, 2, 3]  # 因为最多只有4个棋子会处于边界，因此用列表储存尚未找到的位置以减少不必要的搜索
    # 先后手搜索方向不同，这样也有利于降低时间复杂度
    if self.isFirst:
        for i in wes:
            if board.getBelong((i, 7)) == self.isFirst and board.getBelong((i, 6)) == self.isFirst:
                border.append((i, 7))
                wes.pop(wes.index(i))
        for j in range(1, 7):
            if wes != []:
                for i in wes:
                    if board.getBelong((i, 7 - j)) == self.isFirst and board.getBelong(
                            (i, 6 - j)) == self.isFirst and board.getBelong((i, 8 - j)) != self.isFirst:
                        border.append((i, 7 - j))
                        wes.pop(wes.index(i))
        if wes != []:
            for i in wes:
                if board.getBelong((i, 0)) == self.isFirst and board.getBelong((i, 1)) != self.isFirst:
                    border.append((i, 0))
                    wes.pop(wes.index(i))
    else:
        for i in wes:
            if board.getBelong((i, 0)) == self.isFirst and board.getBelong((i, 1)) == self.isFirst:
                border.append((i, 0))
                wes.pop(wes.index(i))
        for j in range(1, 7):
            if wes != []:
                for i in wes:
                    if board.getBelong((i, j)) == self.isFirst and board.getBelong(
                            (i, j + 1)) == self.isFirst and board.getBelong((i, j - 1)) != self.isFirst:
                        border.append((i, j))
                        wes.pop(wes.index(i))
        if wes != []:
            for i in wes:
                if board.getBelong((i, 7)) == self.isFirst and board.getBelong((i, 6)) != self.isFirst:
                    border.append((i, 7))
                    wes.pop(wes.index(i))
    return border
```

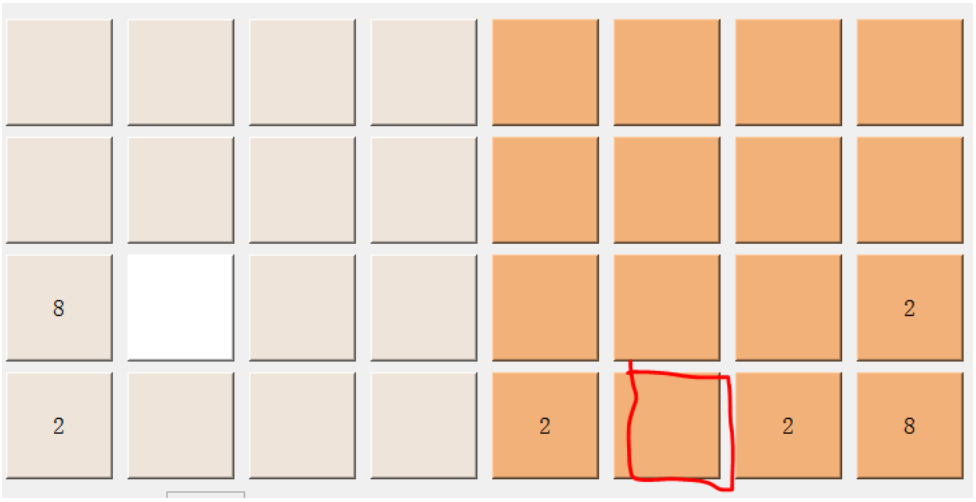
（2）4 个情景判断函数，用于判断如果落子在某个位置上是否会发生某种情况，会用到前两个辅助函数

① scene2 函数：用于判断如图情况是否会发生的函数。落子是否阻挡本方合并，如果阻挡，返回阻挡的方向便于下一步分析，没有阻挡返回 False。其中有一个参数用于调整判断阻挡合并的最低值，比如阻挡 4 和 4 合并并没有什么大问题，阻挡 8 以上的合并就会让人非常头疼。



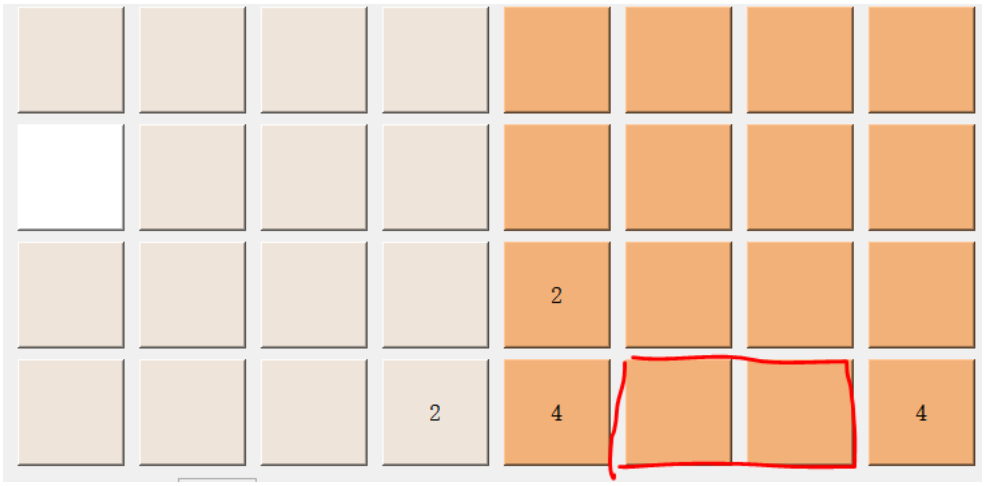
```
def scene2(self, belong, position, board):
    upChess = self.findnearest(position, 0, board)
    downChess = self.findnearest(position, 1, board)
    leftChess = self.findnearest(position, 2, board)
    rightChess = self.findnearest(position, 3, board)

    if upChess != None and downChess != None:
        if upChess[0] == belong and downChess[0] == belong:
            if upChess[1] == downChess[1] and upChess[1] > 2:  # 这里的2是可调参数
                return 0
    if leftChess != None and rightChess != None:
        if leftChess[0] == belong and rightChess[0] == belong:
            if leftChess[1] == rightChess[1] and leftChess[1] > 2:
                return 3 if self.isFirst == True else 2
    return False
```

② scene4 函数：用于判断如图情况是否会发生的函数。在对手落子的位置四周是否有 2，如果有 2 的情况下在此下子相当于送子，对敌有利，对己不利。如果有，返回 True，否则返回 False。



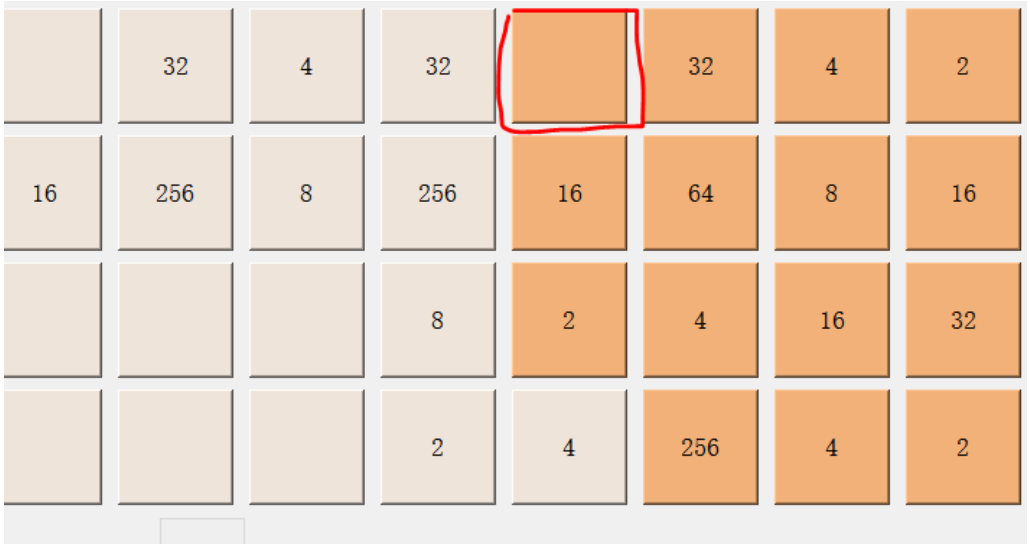

```
def scene4(self, belong, position, board):  # 这里一开始选取上下左右相邻格的棋子，后期调整为选取
    upChess = self.findnearest(position, 0, board)
    downChess = self.findnearest(position, 1, board)
    leftChess = self.findnearest(position, 2, board)
    rightChess = self.findnearest(position, 3, board)

    if upChess != None:
        if upChess[0] == belong and upChess[1] == 1:
            return True
    if downChess != None:
        if downChess[0] == belong and downChess[1] == 1:
            return True
    if leftChess != None:
        if leftChess[0] == belong and leftChess[1] == 1:
            return True
    if rightChess != None:
        if rightChess[0] == belong and rightChess[1] == 1:
            return True
    return False
```

③ scene5 函数：用于判断如图情况是否会发生的函数。在对手方面落子是否会干扰对方合并，如果干扰返回干扰的位置，否则返回 False。其中有一个参数用于调整判断阻挡合并的最低值，比如阻挡对手 4 和 4 合并是没有意义的，因为 2 很容易合成 4 而继续合并。




```
def scene5(self, belong, position, board):  # 情况5
    upChess = self.findnearest(position, 0, board)
    downChess = self.findnearest(position, 1, board)
    leftChess = self.findnearest(position, 2, board)
    rightChess = self.findnearest(position, 3, board)

    if upChess != None and downChess != None:
        if upChess[0] == belong and downChess[0] == belong:
            if upChess[1] == downChess[1] and upChess[1] > 2:  # 这里的2为可调参数，用于调整判断
                return position
    if leftChess != None and rightChess != None:
        if leftChess[0] == belong and rightChess[0] == belong:
            if leftChess[1] == rightChess[1] and leftChess[1] > 2:
                return position
    return None
```

④ scene7 函数：用于判断如图情况是否会发生的函数。自己边界的大子是否可能会在下一步移动中被吃掉，如果是，返回防守位置，否则，返回 None。



```
def scene7(self, board):  # 情况7，防御性填子
    dangerList = self.isborder(board)
    if self.isFirst:
        for i in dangerList:
            if board.getValue(i) > 1:  # 这里最好分开讨论以防止将位置索引到棋盘外引发bug
                rightEnemyChess = self.findnearest(i, 3, board)  # 右侧的敌人棋子
                if rightEnemyChess:
                    if rightEnemyChess[1] == board.getValue(i) and (rightEnemyChess[2][1] - i[1]
                        return (rightEnemyChess[2][0], i[1] + 1)
        return None

    else:
        for i in dangerList:
            if board.getValue(i) > 1:
                leftEnemyChess = self.findnearest(i, 2, board)  # 左侧的敌人棋子
                if leftEnemyChess:
                    if leftEnemyChess[1] == board.getValue(i) and (i[1] - leftEnemyChess[2][1])
                        return (i[0], i[1] - 1)
        return None
```

（3）主函数 luozi 函数：分为先手后手各自判断，代码形式基本一样在判断方向和权重上理由改变。根据传入的回合数和棋盘给出代码认为当前回合最合适的位置。

```
def luozi(self, currentRound, board):


    if self.isFirst:  # 先手
        another = board.getNext(self.isFirst, currentRound)  # 己方的允许落子点
        available = board.getNone(not self.isFirst)  # 对方的允许落子点
        lista = board.getScore(self.isFirst)  # 己方的棋子价值表
        listb = board.getScore(not self.isFirst)  # 对方的棋子价值表
        a = len(lista)  # 己方的棋子数量
        b = len(listb)  # 对方的棋子数量

        if self.scene7(board):  # 优先保护本方大子
            return self.scene7(board)
        else:
            if a < 15:  # 此处15为可调参数，用于调整判断我方阻塞是否发生的最低值
                choice = []  # 以二维列表形式储存各个位置及其落子权重，最后排序取最大值，下面的1000，
                if another != ():  # 保守型，优先判断下在本方是否会阻碍本方落子
                    if self.scene2(self.isFirst, another, board):
                        self.nextchoice = self.scene2(self.isFirst, another, board)
                        choice.append([another, -1000])
                    else:
                        choice.append([another, 100])
                if available:
                    for i in available:
```

因为在比赛中一旦大子被吃掉基本就会无力回天，所以本代码将保护本方边界上的棋子的安全设为优先级最高的部分，先进行 scene7 判断，一旦判断表明需要保护立刻落子保护。

```
                for i in available:
                    if self.scene5(not self.isFirst, i, board) and not self.scene4(not self.
                        choice.append([i, 500])
                    else:
                        choice.append([i, 0])
            if choice == []:
                return None
            else:
                choice.sort(key=lambda x: x[1], reverse=True)
                return choice[0][0]

        else:  # 场上本方棋子数以超过14，调整权重，使得优先下在对方
            choice = []
            if another != ():
                choice.append([another, -1000])
            if available:
                for i in available:
                    if self.scene5(not self.isFirst, i, board) and not self.scene4(not self.
                        choice.append([i, 500])
                    else:
                        choice.append([i, 0])
            if choice == []:
                return None
            else:
```

在比赛中一旦本方被塞爆会对本方很不利，因此，给出一个参数用于调整落子策略，经过几百场比赛的调试，最终调整为本方棋子数少于 14 个时优先在本方下棋，先在本方合并，如果阻碍本方大子合并就去对方进攻。如果本方棋子数大于 14 个就优先进攻阻塞对方的合并从而塞爆对方取得优势。

```
                return None
            else:
                choice.sort(key=lambda x: x[1], reverse=True)
                return choice[0][0]


    else:  # 后手，在参数上与先手略有不同
        another = board.getNext(self.isFirst, currentRound)  # 己方的允许落子点
        available = board.getNone(not self.isFirst)  # 对方的允许落子点
        lista = board.getScore(self.isFirst)  # 己方的棋子价值表
        listb = board.getScore(not self.isFirst)  # 对方的棋子价值表
        a = len(lista)
        b = len(listb)

        if self.scene7(board):
            return self.scene7(board)
        else:
            if a < 15:  # 场上本方棋子数以超过14，调整权重，使得优先下在对方
                choice = []
                if another != ():
                    if self.scene2(self.isFirst, another, board):
                        self.nextchoice = self.scene2(self.isFirst, another, board)
                        choice.append([another, -1000])
                    else:
```

后手方代码形式基本与先手相同，但后手在参数选择上更加偏向于保守型，而先手方更偏向于攻击性，在代码中体现为各种场景的权重发生变化。

```
                else:
                    choice.append([another, 100])
            if available:
                for i in available:
                    if self.scene5(not self.isFirst, i, board) and not self.scene4(not self.
                        choice.append([i, 300])
                    else:
                        choice.append([i, 0])
            if choice == []:
                return None
            else:
                choice.sort(key=lambda x: x[1], reverse=True)
                return choice[0][0]

        else:
            choice = []
            if another != ():
                choice.append([another, -1000])
            if available:
                for i in available:
                    if self.scene5(not self.isFirst, i, board) and not self.scene4(not self.
                        choice.append([i, 300])
                    else:
                        choice.append([i, 0])
```

```
        if choice == []:
            return None
        else:
            choice.sort(key=lambda x: x[1], reverse=True)
            return choice[0][0]
```

2.移动函数主要由三部分函数组成：

①judgeout 函数：通过传入的具体局势信息，返回此情况的权重，单独列出这个函数可以方便对参数进行调节。传入的参数有：两步后敌我最大值，五步后敌我最大值，我移动前后的所有棋子之和，五步中对方和我方合并的值，最终返回一个具体权重。

```
def judgeout(self,mymax1,hismax1,mymax,hismax,mysum,mypresum,
             Ocombinevalue=0,Scombinevalue=0,Tcombinevalue=0,Fcombinevalue=0,Lcombinevalue=0):
    #此函数是用来计算估值函数，单独列出来方便我们调参
    #combine系列参数均为在我们预设为5的情况下的合并值
    #mysum，mymax等参数是关于节点上的棋盘局势
    judge = (mymax - hismax) * 1000 + (mymax1-hismax1)*200-3 * (mysum - mypresum) \
            + Ocombinevalue + Scombinevalue - Tcombinevalue - Fcombinevalue + Lcombinevalue
    return judge
```

②eataftermove 函数：判断某次移动后，self.ifFirst 一方的合并情况，返回的是合并值的和除以合并的个数（以此鼓励更大的合并）。

```
def eataftermove(self, new, prevalue, myospre, hispospre, prevothervalue):
    # 这个函数用来判断某个方向移动了之后，我们会怎么样更新
    valuelist = new.getScore(self.isFirst)
    othervaluelist = new.getScore(not self.isFirst)
    mypos = len(valuelist)
    hispos = len(othervaluelist)
    if (myospre - mypos)!= 0:
        combinevalue = (sum(prevalue) - sum(valuelist) + (myospre - mypos) + (hispospre - hispos)) \
        / (myospre - mypos) # 返回的是一次合并的总和，由于它只是给value加，实际是要加上负数的
    else:
        combinevalue = 0
    return combinevalue # 最终返回的是双方合并的值的和（都为负）
```

③主函数 minimax：此函数区分先后手，首先整理当前棋盘局势信息并写入存档，之后模拟我方下一次移动的四个方向，并对每个可移动方向调用递归函数。递归函数会返回对应此方向在五次移动后对我方最有利的估值函数，之后经过列表排序，确定最优方向进行移动。

```
    def minimax(self, currentRound, board):  # 移动的核心函数
        if self.isFirst == True:  # 区分先后手
            alpha = -1000000
            belta = 1000000
            directionList = [3, 0, 1, 2]
            copyboard = board.copy()

            sumvalue_me = copyboard.getScore(self.isFirst)  # 在无任何移动的时候的场上局
            sumvalue_other = copyboard.getScore(not self.isFirst)
            sumpos_me = len(sumvalue_me)
            sumpos_other = len(sumvalue_other)
            mypresum = sum([2 ** (i) for i in sumvalue_me])
            self.cache[0] = (sumvalue_me, sumpos_me, sumpos_other, sumvalue_other)  # 将
            self.cache['presum'] = mypresum  # 将未移动前我方最大值存档

            movelist = []
            for i in directionList:  # 这里先模拟第一次移动，之后进入迭代函数
                copy = board.copy()
                if copy.move(self.isFirst, i):
                    0sumvalue_me = copy.getScore(self.isFirst)  # 我第一次移动后的总值

        else:  # 区分先后手
            alpha = -1000000
            belta = 1000000
            directionList = [2, 1, 0, 3]
            copyboard = board.copy()

            sumvalue_me = copyboard.getScore(self.isFirst)  # 在无任何移动的时候的场上
            sumvalue_other = copyboard.getScore(not self.isFirst)
            sumpos_me = len(sumvalue_me)
            sumpos_other = len(sumvalue_other)
            mypresum = sum([2 ** (i) for i in sumvalue_me])
            self.cache[0] = (sumvalue_me, sumpos_me, sumpos_other, sumvalue_other)  #
            self.cache['presum'] = mypresum  # 将未移动前我方最大值存档

            movelist = []
            for i in directionList:  # 这里先模拟第一次移动，之后进入迭代函数
                copy = board.copy()
                if copy.move(self.isFirst, i):
```

④ 先手后手的递归函数 progress 和 progress2：对接下来的四步移动进行递归操作，对于每个特定深度除了进行移动并在存档（self.cache）中记录棋盘相关参数以外，也执行该深度移动的特定操作，如在该落子的位置落子和在一回合结束后记录敌我最大值代入估值函数进行计算。递归函数的结束调节是深度为 0，此时计算估值函数并返回，之后依次向上进行计算每一层的最大和最小，最终递归函数返回一个后四步移动的最优函数值。

```
def progress(self,currentRound,board,depth,directionList,alpha,belta):
    #此函数是先手的递归函数，由于我们只递归了五层，在特定层还有落子、取max值等特定操作，因此分开来写
    if depth==0:  #此为递归的结束条件，如果深度为0，就要对估值函数进行计算
        mysumvalue = board.getScore(self.isFirst)
        mysum = sum([2 ** i for i in mysumvalue])  #得到我现在的值之和
        mypresum = self.cache['presum']  #读取存档，得到五次移动之前的我方之和
        mymax = max((board.getScore(self.isFirst) + [0]))  # 己方最大值
        hismax = max((board.getScore(not self.isFirst) + [0])) #敌方最大值
        0combinevalue= self.cache[1][-1]   #对五次移动敌我合并的存档进行调用
        Tcombinevalue = self.cache[2][-1]
        Scombinevalue = self.cache[3][-1]
        Fcombinevalue = self.cache[4][-1]
        Lcombinevalue = self.cache[5][-1]
        judge=self.judgeout(self.cache['max1'][0],self.cache['max1'][1],mymax,hismax,mysum,mypresum,
                            0combinevalue,Scombinevalue,Tcombinevalue,Fcombinevalue,Lcombinevalue)
        return judge  #返回估值，结束此次递归
    if depth==4:
        minjudge=1000000    #给出一个名义上的最大值，用于选取这一步的极小值
        for i in directionList[::-1]: #对方移动，优先移动方向反转
            copy=board.copy()
```

Progress2 函数除了落子位置不同，其他与 pregress 完全相同。

```
def progress2(self, currentRound, board, depth, directionList, alpha, belta):
    # 此函数是后手的递归函数，由于我们只递归了五层，在特定层还有落子、取max值等特定操作，因此分开来写
    if depth == 0:  # 此为递归的结束条件，如果深度为0，就要对估值函数进行计算
        mysumvalue = board.getScore(self.isFirst)
        mysum = sum([2 ** i for i in mysumvalue])  # 得到我现在的值之和
        mypresum = self.cache['presum']  # 读取存档，得到五次移动之前的我方之和
        mymax = max((board.getScore(self.isFirst) + [0]))  # 己方最大值
        hismax = max((board.getScore(not self.isFirst) + [0]))  # 敌方最大值
        Ocombinevalue = self.cache[1][-1]  # 对五次移动敌我合并的存档进行调用
        Tcombinevalue = self.cache[2][-1]
        Scombinevalue = self.cache[3][-1]
        Fcombinevalue = self.cache[4][-1]
        Lcombinevalue = self.cache[5][-1]
        judge = self.judgeout(self.cache['max1'][0], self.cache['max1'][1], mymax, hismax, mysum, mypresum,
                              Ocombinevalue, Scombinevalue, Tcombinevalue, Fcombinevalue, Lcombinevalue)
        return judge  # 返回估值，结束此次递归
    if depth == 4:
        minjudge = 1000000  # 给出一个名义上的最大值，用于选取这一步的极小值
        for i in directionList[::-1]:  # 对方移动，优先移动方向反转
            copy = board.copy()
```

2.3 程序限制

由于在移动函数中后手设置了两次落子，导致时间略有超标。在棋盘极度复杂的情况下，后手可能出现超时现象，但是比例较低。程序设计整体合理，除了超时以外，不会出现其他任何错误情况。

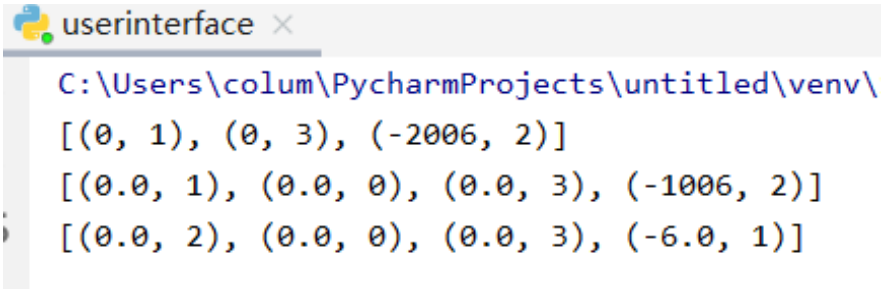
3 实验结果

3.1 测试数据

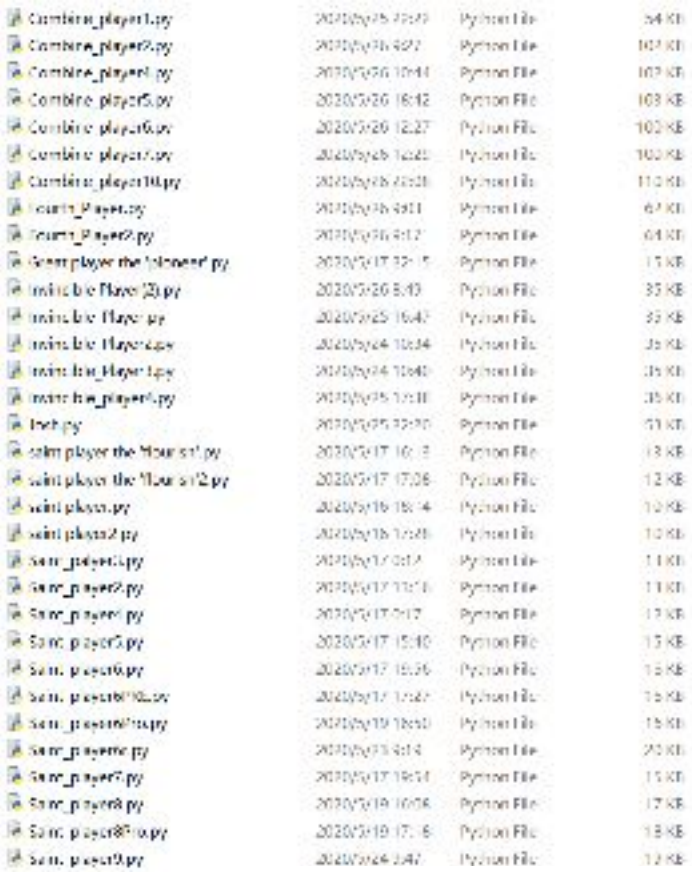
实验环境说明：

- 硬件配置：（CPU/内存）i5-9300H 处理器，8GB 内存
- 操作系统：（名称/版本）Window 10 操作系统
- Python 版本：（版本号）Python 3.7

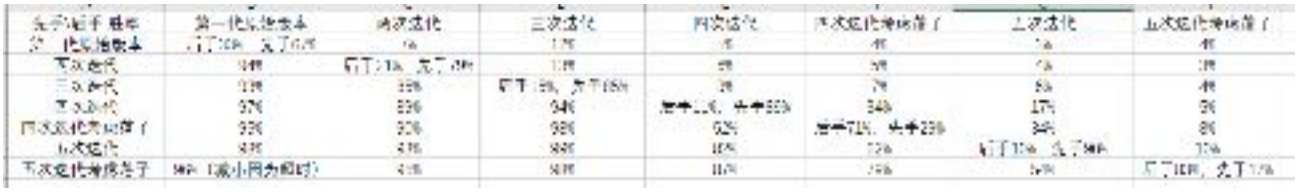
我们在本地利用技术组编写的 Userinterface 进行可视化的程序纠错，通过报错功能检查是否有代码错误；通过 print 方法显示相关参数，来检查设置的函数和权重是不是正确计算。并人为的让棋盘呈现特殊局面，检查在一些极端情况下程序的表现。



我们的算法经历了多代更新，我们通过修改 constant 中的参数，在 roundmatch 中让不同代代码进行多达百场的较量，来选定更好更合适的代码。



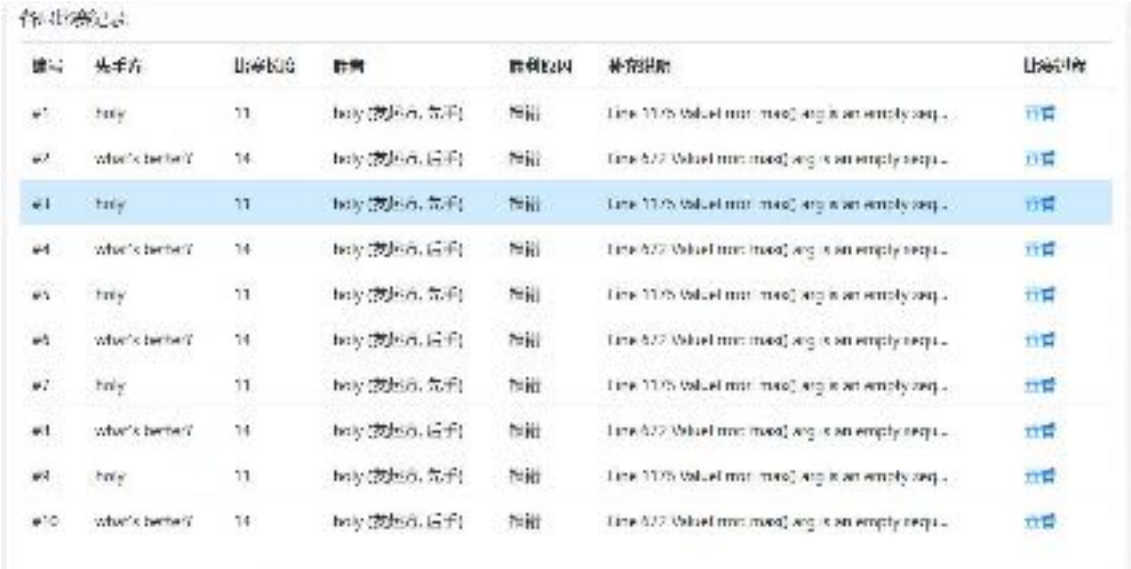
我们的代码在不断的进化！



算法更新换代内战胜率图

我们还在天梯上用不同代代码进行测试和较量，并参加匹配。最高天梯积分能达到 1400+，并曾跻身过天梯热身赛赛前十；最终，我们的代码在天梯赛的激烈竞争中成功突围出线！

天梯表现



天梯上进行修改与调参

天梯赛成功出线！

3.2 结果分析

通过代码内战和天梯赛的实验过程，我们小组发现在移动函数中，往后迭代更深必然会带来 AI 整体能力的大幅提高，每往后多看一层，算法就完全升级了一个档次。因此，思路的转变促成了方法的深刻变革，使得算法的竞争力由“中世纪”的愚昧走向“现代”的开明，从落后的单步策略走向先瞻性极强的 minimax 方法；我们小组才有剪枝等一系列方法，尽可能压缩时间，终于使算法能够往后迭代到第五层。

此外，在往后迭代的过程中，在适当的位置插入敌我双方落子的操作，也对算法性能的提升至关重要。我们惊讶的看到，在插入落子之前，相同的 AI 内战，一般都是先手占优势。但插入智能落子之后，由于存在了落子堵截、阻碍移动等后手优势策略，反而是后手占尽了优势。但总的来说，插入落子比仅仅进行移动的迭代，能力上有了一个较大提升。

3.3 经典战局

经典战局 1：这个**哦哈哈哈 AI** 一直是我们组的梦魇，前面几代代码不管在天梯打得如何，在他面前往往都是不堪一击，基本都是大比分惨败。我们升级到五次迭代代码后，终于能稳定击败他。这次胜利也给了我们组极大的鼓舞。

经典战局②：我们的第五代代码终于打败了官方代码的最高水平，并且对 2048Bot 系列全部代码都能保持稳胜了。

4 实习过程总结

4.1 分工与合作

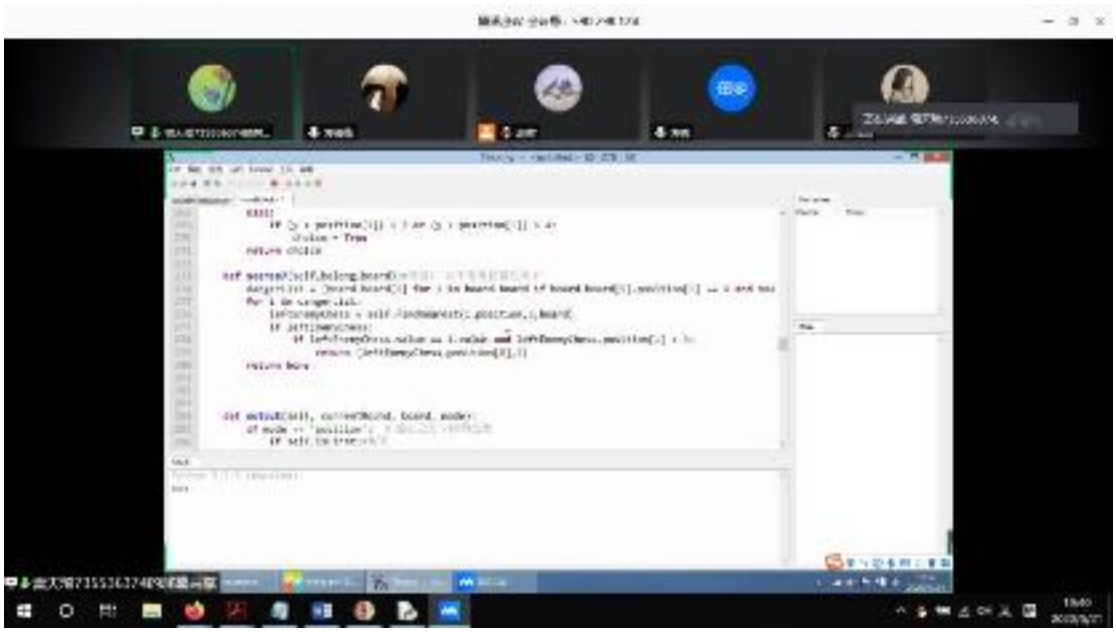
郑爽、雷天琦、王晨旭负责落子部分的程序设计
郑若楷、赵时负责移动部分的程序设计，组合程序并进行一系列调参

本小组主要通过微信群进行交流和协作，并定期召开组会交流汇报成果。
5 月 14 日建立好小组，进行组内分工。
5 月 14 日-5 月 17 日对初始给出的随机算法进行尝试性编程，产生“懦夫策略”与“莽夫策略”
5 月 17 日第一次组会：大家交流对程序的初步想法，对可能实现的 AI 提出建议并讨论可行性，实现具体分工，开始编程。

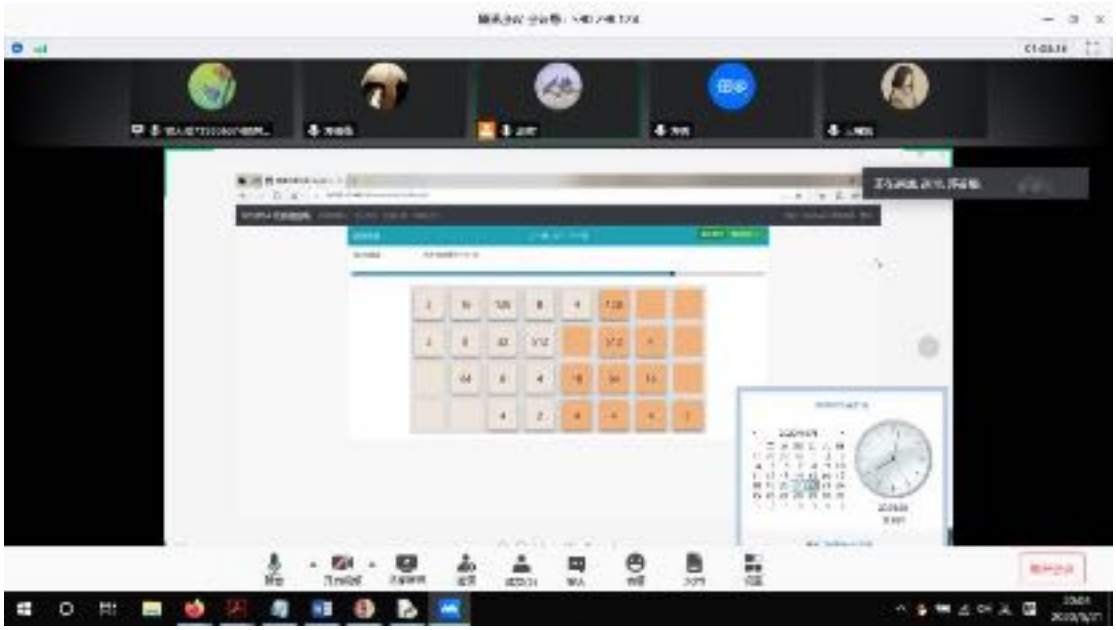
5 月 17 日-5 月 21 日分组各自编写部分，移动算法从人为设定规则移动进化为前瞻式的移动最优判定

5 月 21 日第二次组会：将已完成的两部分代码分别进行讲解，不同编程小组相互之间提出建议，对下一步算法的发展提出展望和构思。

5 月 22 日–参赛落子与移动合并，且产生了落子一部分并入 minimax 函数的尝试，一系列调参后敲定最终参赛版本



组会交流下一步计划



典型案例分析和时间安排

小组合影

之后我们在微信群中进行交流讨论：

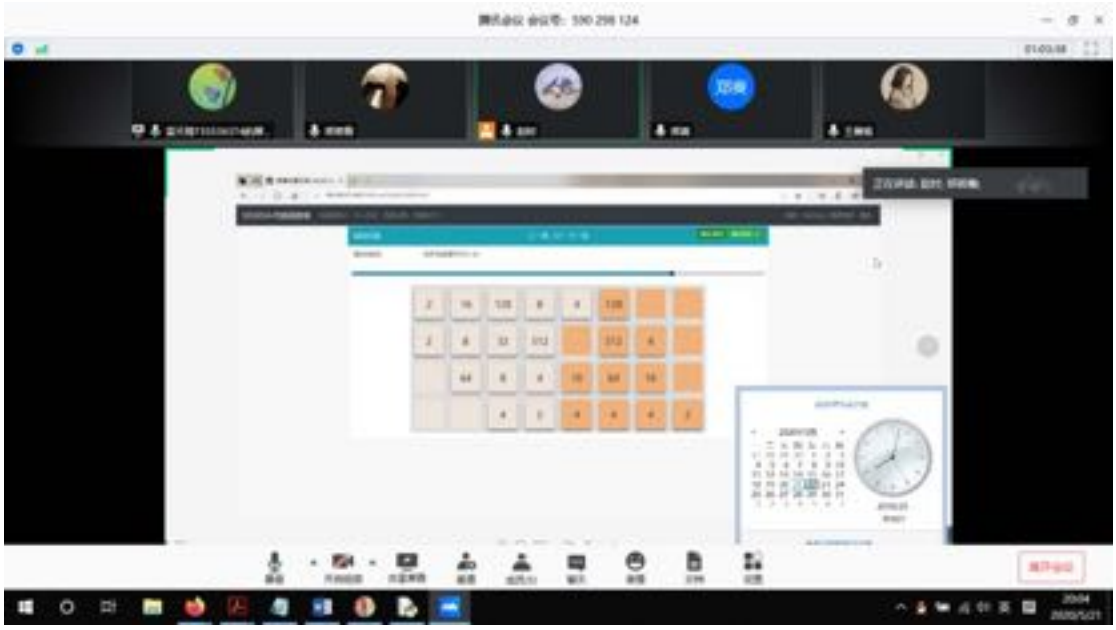
4.2 经验与教训

①编写算法不能莽撞着急，而应广泛地借鉴在相关领域已经成型的算法的思路，善于站在巨

人的肩膀上，先学习、再具体实践。我们小组原先在移动函数中编写了涉及许多判定条件的复杂移动函数，对于合并、边界、危机等一系列情况进行了巨额的讨论。然而某一次偶然的搜索，我们发现了在博弈性 AI 中常用的 minimax 算法的介绍视频，经过分析发现此算法几乎涵盖了包括我们讨论在内的所有情况，使得算法较之前大有改进。

②编写算法之前应将需要解决的问题模型化。我们在编写算法落子函数时，对 2048 游戏中出现的种种情况进行了广泛而深刻的研究，从而分出各种情形，并分别进行处理。

③ 复盘的必要性。通过复盘，我们发现了自己写的 AI 种种愚蠢的行为，进而分析其背后深层的算法控制因素，进而产生修改意见与新的想法。在复盘中，我们还能找到程序的 bug 并加以修正。同时，由于估值函数预设的主观性与可调性，更需要在实际棋局中观察表现，产生并录入启发式规则，对参数进行细致调节。此外，复盘还会有意外收获，如下图中的棋局是我们产生了后手封堵先手逃跑路线的后手策略。（关注第二行第一列的 2，封堵了对方 512 的逃跑路线）



4.3 建议与设想

我们组认为这个大作业真正的达到了锻炼编程能力和团队协作能力的效果，陈斌老师、助教团队和技术组团队呕心沥血，为我们的竞赛搭建了优良的平台，提供了一系列实用的工具。一个小建议是希望可以调用的功能函数能在算法放出来之前就能确定下来，不断对可用函数的更新有可能导致对已完成代码的大幅度改动。另外，希望大作业的时间能够长一点，更有利于我们进行研究和讨论，实现更优良的算法。
我们想对学弟学妹们说：陈斌老师真的好棒好棒，助教学长们又帅又厉害，大作业一定要好好做哦！
如果还有后续时间，我们想咨询一些更加优秀的组，了解一下他们是怎么压缩时间以进行更深的迭代的。对于目前的深度优先算法，可以进一步改成广度优先算法或者 IDDFS 算法，以达到中途 halt 的目的，实现用时间来控制深度，或者用 hash 表来储存已经出现的结果，通过查表压缩时间复杂度。

5 致谢

感谢陈斌老师一个学期以来兢兢业业的教学，通过您的课我们真的受益匪浅。
感谢助教老师们的辛勤工作，为我们搭建了作战平台，回答了不计其数的问题。
感谢技术组成员们的付出与奉献，夜以继日地修改和维护比赛。
感谢小组每一位成员对本次实习作业付出的辛劳与汗水，我们相互碰撞，共同努力，实现了并不完美却足够珍贵的 AI 算法。
感谢那一个个熬过的夜，一次次键盘与鼠标的敲击，一次次思维和想法的碰撞，带给我们进步与成长。

6 参考文献

Algorithms Explained – minimax and alpha-beta pruning 视频- BlueConvert.com

数据结构与算法课程实习作业报告

组长：郑云帆 组员：刘贤雨 吴佩鸿 张弛

【摘要】：本算法基于棋局的多方面分析和分类讨论进行下棋、移动。算法利用列表等基础结构，定义了多种评估函数、策略函数，组成了复杂的判断结构。
【关键字】：落子、移动、子算法、探测函数、评估函数、算法思维方式

1 算法思想.....1
1.1 总体思路.....1
1.2 算法流程图.....2
1.3 算法运行时间复杂度分析.....4
2 程序代码说明.....4
2.1 数据结构说明.....4
2.2 函数说明.....5
2.2.1 Output 方法的落子部分.....5
2.2.2 output 方法的移动部分.....6
2.2.3 探测类方法.....7
2.2.4 评估类方法.....8
2.2.5 落子、移动方法.....10
2.3 程序限制.....13
3 实验结果.....14
3.1 测试数据.....14
3.2 结果分析.....14
4 实习过程总结.....15
4.1 分工与合作.....15
4.2 经验与教训.....15
4.3 建议与设想.....15
5 致谢.....15
6 参考文献.....15

1 算法思想

1.1 总体思路

1.1.1 算法总体构架

在总体上，本算法的思路是定义若干子算法，并根据当前情况，结合多种方法分析应使用何者. 由于子算法和分析方法较复杂，故本部分介绍大体构架，其后再继续详细讨论. 本介绍将分为落子、移动两个方面.

落子方面，本算法具有 put_1、put_2、put_3、put_4、put_5、put_6 六个子算法，其

中前三者针对先手，后三者针对后手. 算法主要考虑了对方空位数、己方空位数、安全系数等方面. 分析过程可概括为：首先，算法判断对方空位数，若为一，则可在对方区域下子，造成堵塞. 若不行，则判断先手、后手，并定义安全系数，评估当前局势. 以先手为例，若安全系数低，则采用 put_1；若较高，则计算己方空位数. 若空位数较多，则采用 put_2；若较少，则采用 put_3. 对于后手的情况，只需将 1、2、3 分别改为 4、5、6 即可.

在移动方面，本算法的特点是分为成长模式、攻击模式和收官模式三个模式。移动算法具有 move_1.1、move_1.2、move_2 等子算法，并主要考虑了回合数、己方空位数、己敌方间谍情况、能否发起侵略等方面。下面将根据不同模式进行讨论。

1. 若局数小于 200，且己方空位数大于 4，则进入成长模式。本模式首先判断己方是否存在敌方间谍，并判断能否吃之。若能，则吃之；若不满足，则根据先后手，分别选择 move_1.1 方法和 move_1.2 方法。
2. 若局数大于等于 200，且己方空位数大于 4，则进入攻击模式。本模式首先判断己方是否存在敌方间谍，并判断能否吃之。若能，则吃之；若不满足，则判断是否存在可使己方间谍吃掉敌方棋子的移动方式。若存在，则向该方向移动；若不存在，则判断是否可侵略。若可，则侵略；反之，则采用 move_2 方法。
3. 若己方空位数小于 4，则进入收官模式。本模式首先判断己方是否存在敌方间谍，并判断能否吃之。若能，则吃之；若不满足，则采用 move_2 方法。

1.1.2 主要数据结构和方法

本部分给出主要数据结构和方法，并列出上部分中出现的子方法。本算法主要在 player 类中定义方法，除__init__方法和 output 方法外，本处定义了以下三类方法：

第一类：探测类方法，分别有 detector、antiSpy、antiSpying、Spy、Spying、spysafe 方法。

第二类：评估类方法，分别有 evaluate1、expect、move_assess、anotherlessexpect、dangermode、op_expect、remove_assess、evaluation、safe_grade 方法。

第三类：落子、移动方法，分别有前文提到的 put_1、put_2、put_3、put_4、put_5、put_6，和 move_1.1、move_1.2、move_2 方法。

此外，本处为对单个棋子的估值而定义了 value 函数。以上各个方法的详细介绍将在代码介绍部分中给出。

1.1.3 原始算法简介及改进过程

本组在开始制作一周内给出了原始算法，其主要思路和上述相近。相对于原始算法，本算法主要进行了以下几点改进：

1. 原始算法对落子的考虑较粗糙，未将对方棋盘、落子的后果等方面纳入考虑，且未考虑先后手，只有三个落子手段。在后期，本组对此进行了大幅改进。
2. 原始算法过多看中自身合成的效率，忽视了进攻的作用。在测试中，本算法常因对方攻击而被堵塞，自身合成的效率也随之下降。改进中，本算法放弃了一味在角落中合成的思路，采取了积极进攻+灵活合成的方针。
3. 原始算法的局势判断过于简单。为此，本组接连设计了棋盘复杂度、间谍估值等多方面的判别函数，加深了对棋盘的分析程度。

1.2 算法流程图

本处分为落子流程图、移动流程图：

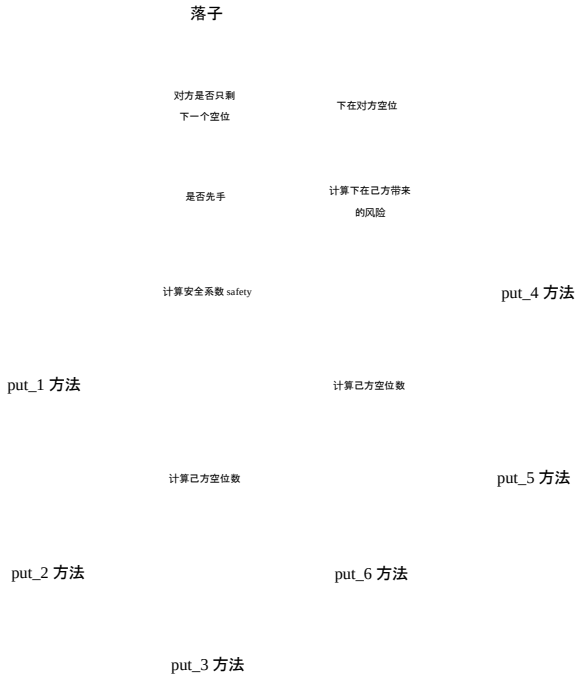


图 1 落子流程图

图 2 移动流程图

1.3 算法运行时间复杂度分析

由于本算法未使用树等结构，故时间复杂度主要和估值函数有关。在估值函数中，由于随对战的进行，局面棋子数将逐渐变化，故理论上运行时间应和局数正相关。然而，考虑到在数十局后局面即已达到近饱和状态，故此后的运行时间几乎不随局数变化。可以认为，本算法一局的运行时间随局数变化的时间复杂度近似为$O(1)$.

在实际对战中，本算法未出现超时现象，且各局运行时间基本控制在一秒之内。树结构的缺乏可能是时间未能充分利用的一大原因。

2 程序代码说明

2.1 数据结构说明

申明：本处代码为优化代码，与对战代码存在结构差别，不存在功能差别。

本算法基本使用列表、棋盘等基础数据类型，未使用树、图等结构。如前文所述，算法在 player 类中定义了以下三类方法：

第一类：探测类方法，分别有 detector、antiSpy、antiSpying、Spy、Spying、spysafe 方法。

第二类：评估类方法，分别有 evaluate1、expect、move_assess、anotherlessexpect、dangermode、op_expect、remove_assess、evaluation、safe_grade 方法。

第三类：落子、移动方法，分别有前文提到的 put_1、put_2、put_3、put_4、put_5、put_6，和 move_1.1、move_1.2、move_2 方法。

此外，本处为对单个棋子的估值而定义了 value 函数。下部分将给出方法的详细说明。

2.2 函数说明

2.2.1 Output 方法的落子部分

如图为落子分析的主代码。图中首先获取下一步己方落子位置，并获取对方空位列表。如果对方空位数为 1，则在对方处下子。余下的部分分为先手、后手判断，并分别求出了安全等级函数（$safety$、$dangerindex$），结合自身空位数来判断应选取何算法。

```
else:  # 输出己方合并的方向
    directionlist = [0, 1, 2, 3]
    self.oklist = []  # 调整可移动位置列表self.oklist

    for direction in directionlist:
        chessboard = board.copy()
        if copy.move(self.isFirst, direction):
            self.oklist.append(direction)

    if self.oklist != []:
        self.safelist = []
        for i in self.oklist:
            if self.move_assess(board, i) == 0:
                self.safelist.append(i)

    if self.oklist == []:
        self.safelist = []
    self.good_direction = []

    if self.safelist != []:
        for i in self.safelist:
            if self.evaluate1[board,i] <= 10:
                self.good_direction.append(i)

    self.exup = self.expect(board, 0)  # 上下左右，上左上右，下左下右的合并值
    self.exdown = self.expect(board, 1)
    self.exleft = self.expect(board, 2)
    self.exright = self.expect(board, 3)
    self.dangerup = self.move_assess(board, 0)
    self.dangerdown = self.move_assess(board, 1)
    self.dangerleft = self.move_assess(board, 2)
    self.dangerright = self.move_assess(board, 3)
    self.emptynum = len(board.getNone(self.isFirst))
    self.AnemySpies = self.antiSpy(board)  # 己方棋盘上的对方棋子坐标
    if currentRound < 200 and self.emptynum >= 4:
        if self.AnemySpies != []:  # 有敌方棋子则检测是否可吃
            destroy = self.antiSpying(board, self.AnemySpies)
            if destroy != -1 and - self.op_expect(board, destroy) >= self.move_assess(board, destroy):
                return destroy
        if self.isFirst:  # 先手
            return self.move_1_1()
        else:
            return self.move_1_2[currentRound]
        elif self.emptynum >= 4:  # 攻击模式
            if self.AnemySpies != []:  # 有敌方棋子则检测是否可吃
                destroy = self.antiSpying(board, self.AnemySpies)
                if destroy != -1 and - self.op_expect(board, destroy) >= self.move_assess(board, destroy):
                    return destroy
            spylist = self.Spy(board)  # 敌方区域是否有我方棋子，我方棋子是否可以进行吃掉
            if spylist != []:
                spydirect = self.Spying(board, spylist)
                if spydirect != -1 and spydirect in self.safelist:
                    return spydirect
                if spydirect == -1:
                    if not self.spysafe(board, spylist):
                        if self.isFirst:
                            if 2 in self.oklist: return 2
                        else:
                            if 3 in self.oklist: return 3
            attackable = self.detector(board)  # 是否可以进攻
            if attackable:
                if self.isFirst:  # 是否近攻
                    if 3 in self.safelist:
                        return 3
                else:
                    if 2 in self.safelist:
                        return 2  # 无法攻击则

                return self.move_2()

            elif self.self.emptynum < 4:  # 收官模式
                if self.AnemySpies != []:  # 有敌方棋子则检测是否可吃
                    destroy = self.antiSpying(board, self.AnemySpies)
                    if destroy != -1 and - self.op_expect(board, destroy) >= self.move_assess(board, destroy):
                        return destroy

                return self.move_2()
```

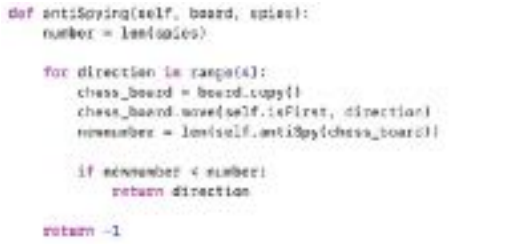
2.2.2 output 方法的移动部分

如上是 output 中移动分析的主要代码。图中首先运用多个列表，判断出移动后具有个方面较好性质的各方向列表，随后生成了各方向移动后的预测合并值、预测风险值。接着，代码分为了成长模式、攻击模式和收官模式，结合多方面因素选取子算法。

2.2.3 探测类方法

（a） detector 方法，用于检测是否可发起攻击。算法首先初始化棋盘，并定义己方总价值函数。通过左右移动后价值和原来进行比较，可判断是否可侵略。

（b） antiSpy 方法，用于判断己方区域是否有敌方间谍。算法根据先后手，分别在区域内查找地方棋子。最后返回所有敌方间谍的列表。



（c）antiSpying 方法，用于判断是否存在可吃敌方间谍的移动方向。函数输入敌方间谍列表，通过模拟下一次移动判断间谍数是否变化。若变小，则返回该方向；反之，则返回-1。

（d）Spy 方法，用于判断敌方区域是否有我方间谍。方法思路和 antiSpy 类似。

```
def Spying(self, board, spies):
    strength_of_spy = 0
    dilist = [0, 0, 0, 0]

    for spyer in spies:
        strength_of_spy += board.getValue(spyer)

    for direction in range(4):
        chess_board = board.copy()
        chess_board.move(self.isFirst, direction)
        for j in self.Spy(chess_board):
            dilist[direction] += chess_board.getValue(j)

    strength_of_our_spy = max(dilist[0], dilist[1], dilist[2], dilist[3])

    if strength_of_our_spy > strength_of_spy:
        for i in range(4):
            if dilist[i] == strength_of_our_spy:
                return i
    else:
        return -1
```

（e）Spying 方法，用于判断：是否存在移动方向，使得移动后我方间谍强于对方。

```
def spysafe(self, board, spylist):
    num = len(spylist)

    for i in range(4):
        qipan = board.copy()
        qipan.move(not self.isFirst, i)

        if num > len(self.Spy(qipan)):
            return False

    return True
```

（f）spysafe 方法。用于判断敌方移动后我方间谍是否会减少。

2.2.4 评估类方法

```
def evaluate1(self, board, direction):
    chess_board_1 = board.copy()
    chess_board_1.move(self.isFirst, direction)
    evaluate_1 = self.evaluation(chess_board_1, self.isFirst)

    for i in range(4):
        chess_board_2 = ...copy()
        chess_board_2.move(...isFirst, i)
        evaluate_2 = min(self.evaluation(chess_board_2, self.isFirst), ....)

    return ... + evaluate_1 + evaluate_2
```

（a） evaluate1 方法，用于评估己方往某方向移动后，是否对后续移动有利。

```
def expect(self, bd, direction):
    steal = - op_expect(self, bd, direction)
    new_value_list = [0, 0, 0, 0]

    for direct in range(4):
        chess_board = bd.copy()
        chess_board.move(self.isFirst, direct)
        eatlist = chess_board.getSpace(self.isFirst)

        for index in eatlist:
            new_value_list[i] += value[index]

    step2 = max(new_value_list[0], new_value_list[1], new_value_list[2], new_value_list[3]) - new_value
    return steal - 0.6 * step2 + 0.4
```

（b） expect 方法，用于往某方向移动后，继续移动后所产生的最高综合价值。

```
def move_assess(self, board, di):
    chess_board = board.copy()
    chess_board.move(self.isFirst, di)
    dangerpreb = max(self.op_expect(chess_board, 0), self.op_expect(chess_board, 1), self.op_expect(chess_board, 2), self.op_expect(chess_board, 3))

    if dangerpreb > 10:
        return dangerpreb
    else:
        return 1000 * dangerpreb
```

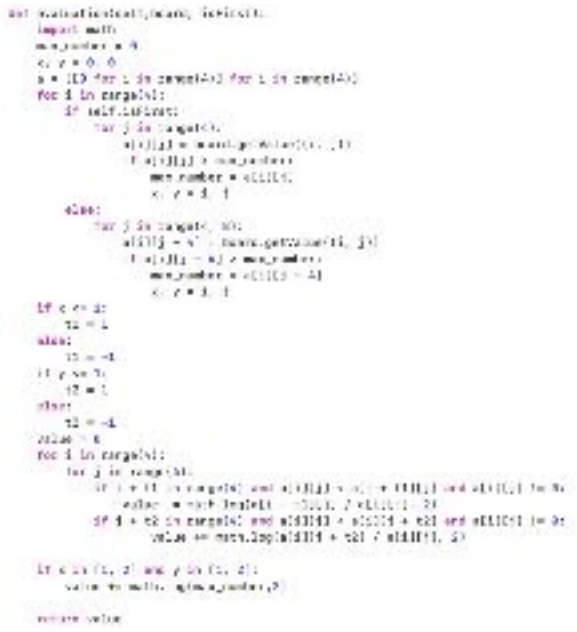
（c） move_assess 方法，用于评估向某方向移动后是否会产生危险。如果危险过大，则将危险系数扩大 1000 倍返回。

（d） anotherlessexpect 方法，用于评估对方移动后的价值。

（e） dangermode 方法，用于评估往某方向移动后自身价值的减少值。

（f） op_expect 方法，返回某方向移动后所产生的价值的相反数。

（g） remove_assess 方法，用于评估向某方向移动的危险程度。



（h） evaluation 方法，用于评估当前己方的混乱程度。

（i） safe_grade 方法，用于评估安全值。

2.2.5 落子、移动方法

```
def put_1(self, board):
    enemy_space = board.getNone(not self.isFirst) #获取对方空白位置
    index = 99999 #设置较大值

    if enemy_space != []:#计算落子后对对面是否安全决策最小值
        for space in enemy_space:      #这里是对空白位置进行落子安全评估遍历
            chess_board = board.copy()
            chess_board.add(self.isFirst,space)
            index = min(index,self.dangermode(board_2,2))

        for i in range(len(enemy_space)):#再次遍历列表寻找哪一个位置落子对应的是最小风险值
            chess_board = board.copy()
            chess_board.add(self.isFirst,enemy_space[i])
            if index == self.dangermode(board_2,2):
                return enemy_space[i]#找到了就返回

    else:    #如果对方对下列表是空集时选
        if next_chess != []:  #如果next_chess不为 []
            return next_chess
        else:
            return None
```

（a）put_1 方法。本方法在对方区域下棋，寻找在对方区域内下棋的最优方法。

```
def put_2(self, board, next_chess):
    chess_board_1 = board.copy()#创建模拟棋盘
    chess_board_2 = board_1.copy()#再创建一个
    chess_board_1.add(self.isFirst,next_chess)#board_1落子己方

    def xiaozhu(b):#定义简单函数
        return max(self.expect(b,0),self.expect(b,1),self.expect(b,2),self.expect(b,3))

    if xiaozhu(chess_board_1) <= xiaozhu(chess_board_2):#如果落子己方较敌方较小则第2列表
        def opvalue1(b): #对方一步能达到的值
            return max(self.anotherlessexpect(b,0),self.anotherlessexpect(b,1),self.anotherlessexpect(b,2),self.anotherlessexpect(b,3))
        enemy_space = board.getNone(not self.isFirst)#获取空位
        index = 99999
        if enemy_space != []: #模拟对面空位列表
            for i in range(len(enemy_space)):
                chess_board_3 = board.copy()
                chess_board_3.add(self.isFirst,enemy_space[i])
                index = min(index,opvalue1(chess_board_3))
            for i in range(len(enemy_space)): #再次遍历以寻找最小对对应的位置
                chess_board_3 = board.copy()
                chess_board_3.add(self.isFirst,enemy_space[i])
                if index == opvalue1(chess_board_3):
                    return enemy_space[i]
        if enemy_space == []:#如果对面列表是空列表
            if next_chess != () return next_chess
            return None
    else:#如果落子收益足够
        return next_chess
```

（b）put_2 方法。

```
def put_3(self, board, next_chess):
    chess_board_1 = board.copy()      #模拟一份棋盘
    def opvalue1(b):#对方一步能达到的值
        return max(self.anotherlessexpect(b,0),self.anotherlessexpect(b,1),self.anotherlessexpect(b,2),self.anotherlessexpect(b,3))
    enemy_space = board.getNone(not self.isFirst)#获取空位设置大值
    index = 99999
    if opvalue1(chess_board_1) > 8:#如果对面可以合成大致
        if enemy_space != []:#如果列表不为空
            for i in range(len(enemy_space)):
                chess_board_2 = board.copy()
                chess_board_2.add(self.isFirst,enemy_space[i])
                index = min(index,opvalue1(chess_board_2))#落子后计算风险值最小值
            for i in range(len(enemy_space)):#再次遍历以寻找最小的对应的位置
                chess_board_2 = board.copy()
                chess_board_2.add(self.isFirst,enemy_space[i])
                if next_chess != () and index == opvalue1(chess_board_2) and index <= opvalue1(b4) - 8:
                        return enemy_space[i]
                elif index == opvalue1(chess_board_2):
                        return enemy_space[i]
        if next_chess != []:#如果列表是空列表
            return next_chess
    else:#如果对面无法合成大致
        if next_chess != []:#先无法则自己的下一步值
            return next_chess
        elif enemy_space != []:
            from random import choice
            return choice(enemy_space)
```

（c）put_3 方法。

```
def put_4(self, board):
    def dangerindex(b):
        return max(self.dangermode(b,0),self.dangermode(b,1),self.dangermode(b,2),self.dangermode(b,3))
    enemy_space = board.getNone(not self.isFirst)#获取对方空白位置
    index = 99999#设置较大值

    if enemy_space != []:#如果列表不为空，对对方空白位置进行遍历
        for i in range(len(enemy_space)):
            chess_board = board.copy()
            chess_board.add(self.isFirst,enemy_space[i])#在空白位置落子
            index = min(index,dangerindex(chess_board))#计算落子后的可能返回的安全风险最小值
        for i in range(len(enemy_space)):#再次遍历列表寻找哪一个位置落子对应的最最小风险值
            chess_board = board.copy()
            chess_board.add(self.isFirst,enemy_space[i])
            if index == dangerindex(chess_board):
                return enemy_space[i]
    if enemy_space == []:#空列表则优先刷后自己位置落子
        if next_chess != ():
            return next_chess
        else:
            return None
```

（d）put_4 方法。

```
def put_5(self, board, next_chess):
    chess_board_1 = board.copy()
    chess_board_2 = board_1.copy()
    chess_board_1.add(self.isFirst,next_chess)
    def xiaoshu(b):
        return max(self.expect(b,0),self.expect(b,1),self.expect(b,2),self.expect(b,3))
    if xiaoshu(chess_board_1) <= xiaoshu(chess_board_2):
        def opvalue1(b):
            return max(self.anotherlessexpect(b,2),self.anotherlessexpect(b,1),self.anotherlessexpect(b,2),self.anotherlessexpect(b,3))
        enemy_space = board.getNone(not self.isFirst)
        index = 99999
        if enemy_space != []:
            for i in range(len(enemy_space)):
                chess_board_3 = board.copy()
                chess_board_3.add(self.isFirst,enemy_space[i])
                index=min(index,opvalue1(b3))
            for i in range(len(enemy_space)):
                chess_board_3 = board.copy()
                chess_board_3.add(self.isFirst,enemy_space[i])
                if index == opvalue1(b3):
                    return enemy_space[i]
        if enemy_space == []:
            if next_chess != ():return next_chess
            return None
    else:
        return next_chess
```

（e）put_5 方法。

```
def put_6(self, board, next_chess):
    chess_board_1 = board.copy()
    def opvalue1(b):
        return max(self.anotherlessexpect(b,0),self.anotherlessexpect(b,1),self.anotherlessexpect(b,2),self.anotherlessexpect(b,3))
    enemy_space = board.getNone(not self.isFirst)
    index = 99999
    if opvalue1(chess_board_1)>0:
        if enemy_space!=[]:
            for i in range(len(enemy_space)):
                chess_board_2 = board.copy()
                chess_board_2.add(self.isFirst,enemy_space[i])
                index=min(index,opvalue1(chess_board_2))
            for i in range(len(enemy_space)):
                chess_board_2 = board.copy()
                chess_board_2.add(self.isFirst,enemy_space[i])
                if next_chess!=() and index==opvalue1(b4) and index<=opvalue1(b5):
                    return enemy_space[i]
                elif and index==opvalue1(b5):
                    return enemy_space[i]
        if next_chess != ():
            return next_chess
    else:
        if next_chess != ():
            return next_chess
        if next_chess == ():
            if enemy_space != []:
                from random import choice
                return choice(enemy_space)
```

（f）put_6 方法。

```
def move_1_1(self):
    if 3 in self.safelist:
        if self.detector(board):
            self.memory.append(3)
            return 3
    Direction = {self.exup - 4 * self.dangerup, self.exdown - 4 * self.dangerdown, self.exleft - 4 * self.dangerleft,
                 self.exright - 4 * self.dangerright}  # 选取上下左右最大向并输向声
    maxscore = max(self.exup - 4 * self.dangerup, self.exdown - 4 * self.dangerdown, self.exleft - 4 * self.dangerleft,
                   self.exright - 4 * self.dangerright)

    max1 = 0
    for direction in self.good_direction:
        max1 = max(max1, Direction[i])
    for direction in [3, 0, 1, 2]:
        if Direction[direction] == max1 and direction in self.good_direction: return direction
    if len(self.good_direction) == 0:
        for direction in [3, 0, 1, 2]:
            if Direction[direction] == maxscore and direction in self.safelist: return direction
    for direction in [3, 0, 1, 2]:
        if direction in self.eklist: return direction
```

（g）move_1_1 方法。

```
def move_1_2(self,currentRound):
    if self.memory[-1] != 6:  # 如果上一步也占了到这里
        self.memory.pop()
        if self.safelist == [3]:
            return 3
    if 2 in self.safelist:
        if self.detector(board):
            self.memory.append(2)
            return 2

    Direction = [self.exup - 4 * self.dangerup, self.exdown - 4 * self.dangerdown, self.exleft - 4 * self.dangerleft,
                 self.exright - 4 * self.dangerright]  # 返回上下左右最大合并值合并
    maxscore = max(self.exup - 4 * self.dangerup, self.exdown - 4 * self.dangerdown, self.exleft - 4 * self.dangerleft,
                   self.exright - 4 * self.dangerright)
    max1 = 0

    for direction in self.good_direction:
        max1=max(max1,Direction[i])
    if currentRound < 26:
        for direction in [3, 0, 1, 2]:
            if Direction[i] == maxscore and direction in self.safelist: return direction
    if currentRound >= 25:
        for direction in [2, 0, 1, 3]:
            if Direction[i] == max1 and direction in self.good_direction: return direction
    if len(self.good_direction)==0:
        for direction in [3, 0, 1, 2]:
            if Direction[i] == maxscore and direction in self.safelist: return direction
    for direction in [3, 0, 1, 2]:
        if direction in self.oklist: return direction
```

（h）move_1_2 方法。

```
def move_2(self):
    Direction = [self.exup - 4 * self.dangerup, self.exdown - 4 * self.dangerdown, self.exleft - 4 * self.dangerleft,
                 self.exright - 4 * self.dangerright]  # 返回上下左右最大合并值合并
    maxscore = max(self.exup - 4 * self.dangerup, self.exdown - 4 * self.dangerdown, self.exleft - 4 * self.dangerleft,
                   self.exright - 4 * self.dangerright)

    if self.isFirst:
        for i in [3, 2, 1, 2]:
            if Direction[i] == maxscore and i in self.safelist: return i
        for i in [2, 2, 1, 3]:
            if Direction[i] == maxscore and i in self.oklist: return i
        for i in [2, 2, 1, 3]:
            if i in self.oklist: return i

    else:
        if currentRound < 25:
            for i in [3, 0, 1, 2]:
                if Direction[i] == maxscore and i in self.safelist: return i
        if currentRound >= 25:
            for i in [2, 0, 1, 3]:
                if Direction[i] == maxscore and i in self.safelist: return i
        for i in [3, 2, 1, 2]:
            if Direction[i] == maxscore and i in self.oklist: return i
        for i in [3, 2, 1, 2]:
            if i in self.oklist: return i
```

（i）move_2 方法。

2.3 程序限制

大量测试表明：

1. 由于本算法已考虑到各情况的发生，故实际中暂未出现报错。
2. 如前文已述，本算法的耗时较短，实际中也未出现过超时。

3 实验结果

3.1 测试数据

实验环境说明：

- 硬件配置：（2.4 GHz Intel Core i5 / 8 GB 2133 MHz LPDDR3）
- 操作系统：（macOS Mojave/10.14.6）
- Python 版本：（3.7.6）

1. 正式设计算法前，本组测试了其单人模式部分移动效果。测试共进行四千次，其中约两千次结束于 512，约两百次结束于 1024，并有 30 次结束于 2048。
2. 在正式设计算法时，本算法首先和 Robot-D、C、B 进行对战，其中前两者战绩维持在 10:0 至 9:1 左右，但后者战绩保持于 0:10 至 1：9。过程中，本组曾设计基于树结构的算法测试，虽然以 6:4 至 8:2 击败了原先算法，但对于其他算法无明显优势。最后，本组在原算法基础上添加了大量决策函数，性能优于先前两者，以 7：3 击败树结构算法，最后经调试后成为最终算法。

3.2 结果分析

重要的算法及其效果主要有下列几点：

1. 设置了安全系数，在小程度上防止了在己方下子后被攻击等情况。
2. 如果自身空位较少，则落子于敌方。此算法在小程度上缓解了堵塞问题，但难以抵消由于移动算法劣势造成的棋局混乱；同时，某些情况下可能会给予对方发育优势。
3. 在对方能合成大数时进行干扰，效果在短期内显著，但长期内仍然难以避免合并。
4. 通过自身未来两步的移动方式分析移动方向。此方法在实践中效果并不明显，其根本原因在于本算法思路缺少结构性，造成了各方面未能有效协调考虑，且未能进行未来多步骤的预测。此缺陷也是造成棋局混乱的主要因素。
5. 为了完善估值函数，张弛同学加入了混乱度的判断函数。由于对混乱度影响因素很多，在实际对战中该函数的作用一直不明显。后续虽然进行了改良，但胜率提高仍然不明显。没有应用决策树导致这个判断只考虑了部分特殊情况或因素，从结果来看并不完善。

综合来讲，本算法考虑点较多，但各个方面均未能完全达到预期效果，其根本原因来自于思维方式的原因。由于本算法预测程度浅，故运行时间短。开销主要发生在未来棋局的预测上。

4 实习过程总结

4.1 分工与合作

组长：郑云帆
策略：刘贤雨、吴佩鸿、张弛、郑云帆
代码：吴佩鸿、张弛
对战：吴佩鸿、郑云帆
报告：刘贤雨、郑云帆

4.2 经验与教训

经验：

1. 要充分让各个组员参与到各项工作中。如果采取过于明确的、单人化的分工，则很容易造成衔接不良、功能片面等缺陷。
2. 在各个组员水平较平均的情况下，分工过程应依据各个组员间思维方式的差异。

教训：

1. 应趁早采用较好的大致思路。原始代码采用了在单人版之上改进的思路，到后期尝试树结构时，发现时间不足。
2. 本组的部分评估函数中，可能存在特殊情况的盲点，从而给对手带来了机会。由于特殊情况难以被全部考虑，故应采取思维方式上的转化。（例如采用树结构）

4.3 建议与设想

1. 在天梯中向指定代码发起进攻时，可以考虑加入搜索功能
2. 可以考虑加入实时观战功能？

5 致谢

感谢陈斌老师一学期的辛勤付出！
感谢各位助教和卷王们一学期来给予的帮助！
感谢每一位组员的努力！

6 参考文献

https://www.zhihu.com/question/23029850

数据结构与算法课程实习作业报告

马超凡* 吴熙楠 高玮伯 金佳伟 易佳怡

摘要：我们的算法总体策略是通过决策树来分析接下来移动后棋盘的情况，然后通过不同时期的估值函数对棋盘情况进行分析评分，对特殊情况有特殊的处理办法，在决策树中我们使用了多种方法来增加搜索深度。在热身赛上不会出现报错，尽管总体胜率一直不会很高。

关键字：大小，位置，特殊情况

1 算法思想.....1
1.1 总体思路.....1
1.2 算法流程图.....3
1.3 算法运行时间复杂度分析.....4
2 程序代码说明.....5
2.1 数据结构说明.....5
2.2 函数说明.....6
2.3 程序限制.....18
3 实验结果.....18
3.1 测试数据.....18
3.2 结果分析.....19
3.3 经典战局（可选）.....20
4 实习过程总结.....21
4.1 分工与合作.....21
4.2 经验与教训.....22
4.3 建议与设想.....23
5 致谢.....23
6 参考文献.....23

1 算法思想

1.1 总体思路

利用我方对棋盘局面定义之估值函数评估棋盘分数，并使用 Minimax 演算法(最大最小值法)，假定双方都会选择对己方最有利的情况，即为选择对己方分数最高之棋局，使用深度优先搜索，搜索一定深度内的可能移动，在递归调用完成时同时更新节点之分数，每个节点分为终止节点、min 节点和 max 节点，终止节点代表深度达到最大深度之节点，其值即为该局面之分数，min 节点代表此时由敌方做决策，该节点分数为所有子节点分数之最小值，max 节点代表此时由我方做决策，该节点分数为所有子节点分数之最小值。最后在父节点找

出对我方分数最高的移动并输出，同时搭配 alpha-beta 剪枝，并使用启发式规则，在前期考虑落子时不考虑在敌方领域落子，其余时期在 minimax 中若选择落子在敌方领域，要求此时分数需高于操作前之棋局分数，否则不考虑此移动。

前期估值函数（在合成等级为 4 之前定义为前期）：

首先对棋盘内基本棋子赋分求和，并满足等级高一级，分数增加一倍还要多，来达到优先合成等级高的数的目的。同时先手时实现尽量多吃对方；后手时保守一点，保我方发育。总体采用的思路为见缝插针，“吃了就跑”的游击战术。

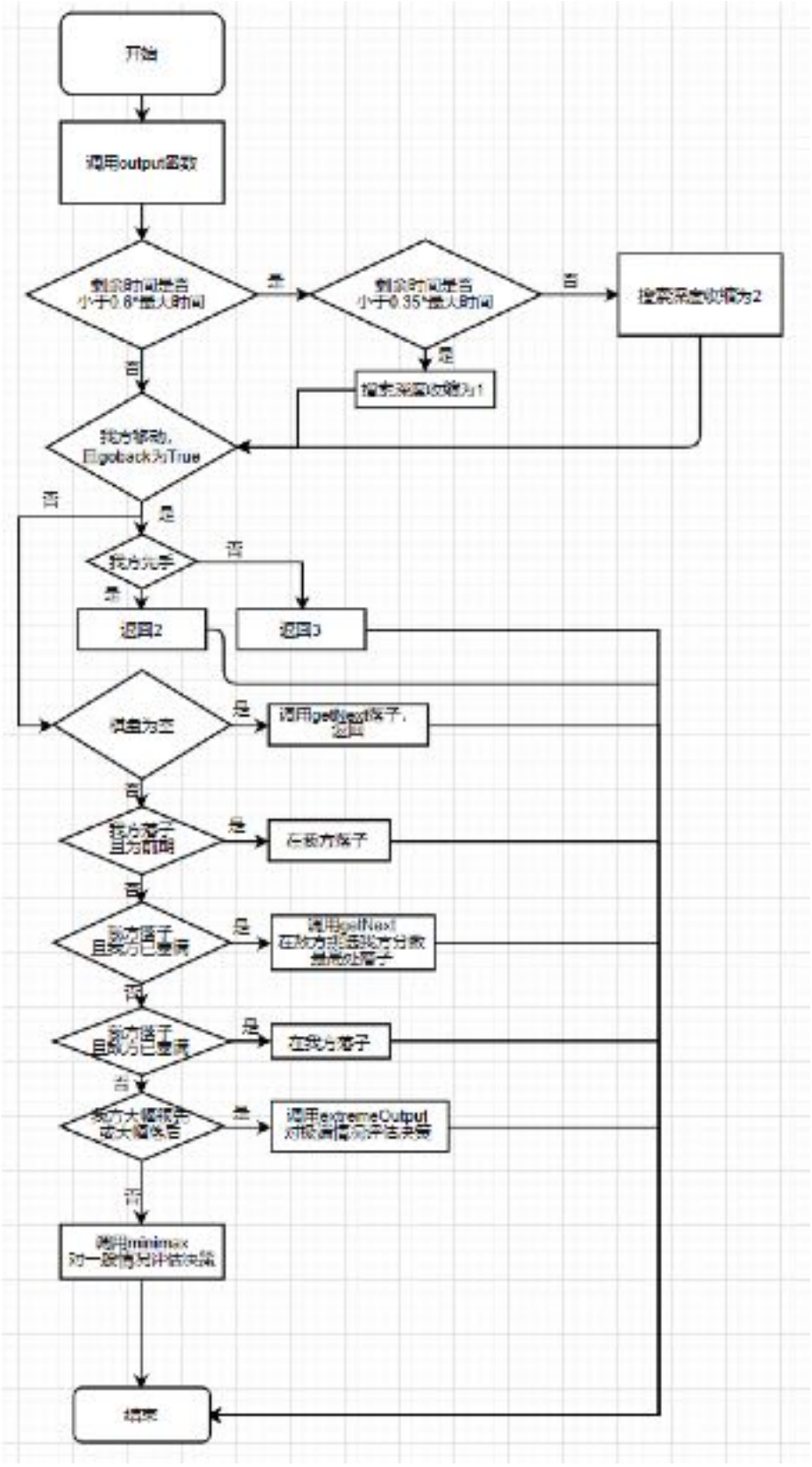
中期估值函数（在合成等级为 4 之后进入中期）：

中期估值函数需要满足以下要求，由棋子等级决定总体上的棋盘得分值，再由位置信息对棋盘得分进行调整。开始我们设想在边角位置养出一个非常大的数字，在距离边界线最远的地盘上按升序排列，摆好阵形来配合最大棋子的合成，因此当时的位置依赖主要是在最边界的那一列为单调排列会获得一定程度的加分，若最大棋子在预定好的位置则获得决定性的加分，如果棋子在对面棋盘则减去相应分数，因为这样会破坏已经形成的规则棋盘，经过实战检验后发现，这样做限制了棋盘 bot 的灵活性，一味防守通常是让对面进攻我们到死，因此我们更改了实现方式。

最终的中期估值函数也是保持原有的基础分方式，在位置依赖关系上进行了调整，改为如果棋子附近存在与他大小相近的棋子（只看比他大的最小的和比他小的最大的，这两个）那么会获得一定的加分，并且棋子等级越高加分越高，如果差距过大会失去分数。另外，如果我方棋子有被对面棋子吃掉的风险，那么也会失去很多分数。最后，为了实现阻碍对方合成的落子，还有一部分是遍历对面棋子所有等级为一的，如果他附近有许多等级比他高的，那么棋盘得分也会有所增加。

特殊情况函数，考虑三类较特殊情况：①我们处于领先优势的时候（然而似乎这个很少被用到），比如我方率先合成 7 级以上的子，且对方未合成与我们同样大小的子，我们将转移视线开始扰乱对面合成。②我们处于落后劣势的时候，比如我方落后对面两级以上的时候，我们将会抓住时机往对面塞不容易合成可以干扰对面的子，使得对方快速达到不能走的状态，然后乘机发育自己。③当我们的落子不太妙的时候，并且刚好有一个 2 落在边界线上，我们考虑往对面落一个 2，然后下一步移动吃掉，再返回。

1.2 算法流程图


开始
调用output函数
剩余时间是否
小于0.8*整人时间
是
剩余时间是否
小于0.35*整人时间
是
搜索深度设置为2
是
搜索深度设置为1
否
我方移动，
且goback为True
否
是
我方先手
否
是
返回2
返回3
调用getBest()落子，
返回
是
否
是
否
是
否
是
否
是
否
结束

1.3 算法运行时间复杂度分析

设棋盘共 r 行，c 列：

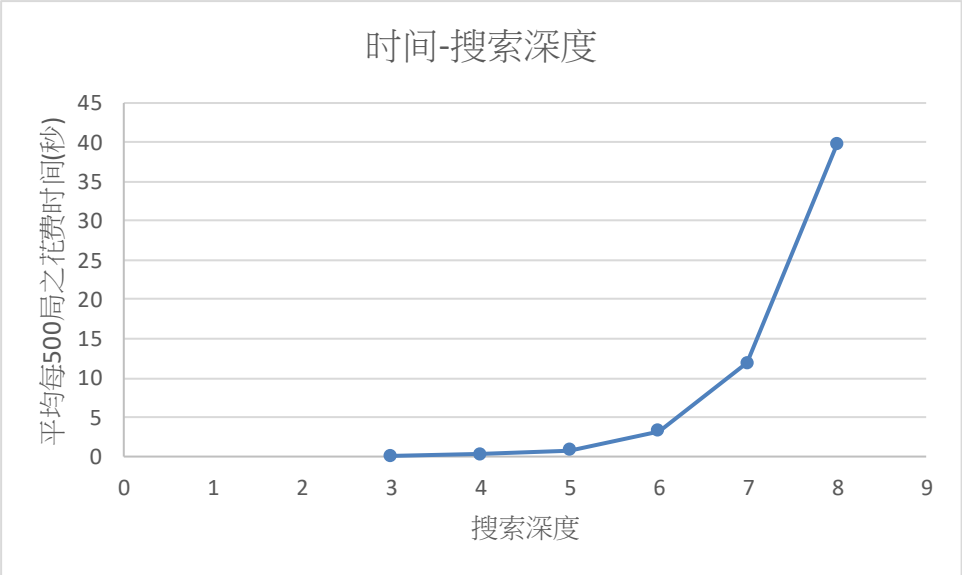
基础分计算的复杂度为 O（r*c），遍历棋子得到位置关系的复杂度为 O（r*c），遍历对方棋子部分的复杂度为 O（r*c），判断是否被吃的部分复杂度为（r*c）。

故总的算法复杂度为 O（r*c），但是在实际开销中，由于循环较多，计算步骤较复杂，时间开销较大，这限制了决策树的深度，但是里面关于一些位置依赖的算法相当于一些更大的基础分的征兆，一定意义上可以理解为增加决策树的深度。

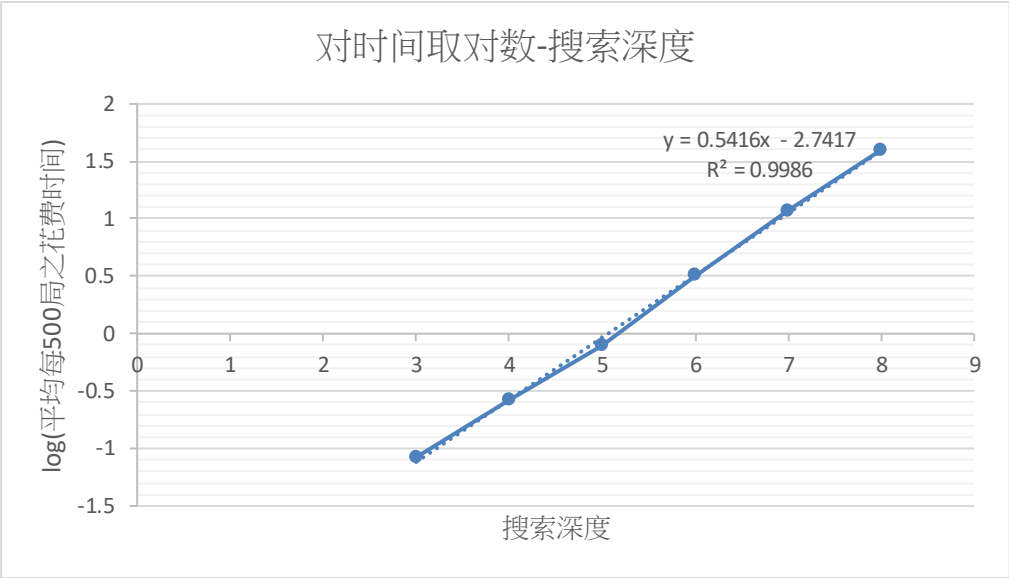
Minimax 演算法加上 alpha-beta 剪枝和启发式规则减少考虑在敌方领域落子之时间复杂度:

若无剪枝，则 Minimax 演算法时间复杂度与深度 d 为指数关系，约为 O(e^d)，因为每个子节点大约都有数量接近的子节点。考虑剪枝后，情况较为复杂，因此使用实验来评估。在本地使用非常简单为 O(1)的计分函数来评估棋盘，并利用此演算法对不同棋局重复运行多次且计算花费之平均时间以得出对搜索深度的时间复杂度。

搜索深度	3	4	5	6	7	8
平均每 500 局之花费时间(秒)	0.084	0.2625	0.7879	3.222	11.9	39.75



搜索深度	3	4	5	6	7	8
log(平均每 500 局之花费时间)	-1.07521	-0.58087	-0.10352	0.508224	1.075547	1.599337



由以上图表可看出，花费时间与深度为指数关系，若深度为 d，时间复杂度约为 O(e^0.54d)，经过有效地剪枝后，时间复查度由未剪枝前的 O(e^d)优化为 O(e^0.54d)

2 程序代码说明

2.1 数据结构说明

Minimax 演算法采用树的基本数据结构实现，树的基本单位为节点 Node，并自定义了一个类 Node。

相较于课上所学的树，此节点上存储信息更多，并且可链接多个子节点，以棋局做为标志 key，将深度、mode(下棋或移动)、父节点、子节点和 min 或 max 节点均存储于该节点，以方便调用，该节点的深度若为最大深度则该节点为终止节点，其值为该棋盘的分数，否则试其为 min 或 max 节点将值设为正或负无穷，同时存储其 alpha 值和 beta 值以利 alpha-beta 剪枝。

```
43      def addChild(self, key):  # 
44          if key not in self.child:
45              self.child.append(key)
46
47      def updateValue(self, newval):
48          self.value = newval
49
50      def updateAlpha(self, newAlpha):
51          self.alpha = newAlpha
52
53      def updateBeta(self, newBeta):
54          self.beta = newBeta
```

```
class Node:  # 结点节点
    def __init__(self, key, minormax: bool, depth, mode, parent, isFirst, early, searchDepth):  # minormax = True 表示min
        self.key = key  # 棋盘状况key
        self.depth = depth  # 深度
        self.mode = mode
        self.parent = parent
        self.isFirst = isFirst
        if self.depth == searchDepth:  # 到达搜索最深层数返回棋盘分数
            self.value = chessboardPoint(key, isFirst, early)
        else:  # 还没到最大深度 若为min 先令其数为正无穷 max为负无穷 随着搜索过程调整
            self.value = infinity if minormax else -1 * infinity
        self.minormax = minormax  # 我方视作为max(False) 我方视作为min(True)
        self.child = []
        self.alpha = -1 * infinity  # alpha 初始为负无穷
        self.beta = infinity  # beta 初始为正无穷
```

Player 类：
在 Player 类中，除了先后手及随机序列为，增加了搜索树最大深度和 goback，goback 为 True 代表上一局向中间移动攻击对手，这一局需退回来以确保安全。

```
class Player:
    def __init__(self, isFirst: bool, array) -> None:
        self.isFirst = isFirst
        self.array = array
        self.searchDepth = 3  # 初始搜索深度 (start from 0) (必奇数)
        self.goback = False
```

2.2 函数说明

possibleMove 函数：

①在 Node 类底下定义此函数，输入当前回合数、先后手和是否为前期，输出一个列表

②功能：输出所有需要考虑的下一步。

③具体操作为：若 mode 为移动，则考虑四个方向是否能移动，若能移动则添加移动后棋局至列表，最后输出列表。若 mode 为落子，分前期和非前期讨论，为前期则输出列表仅有落子在己方后的棋盘，非前期时，如果能在己方落子，将落子在己方后的棋盘添加至列表后，考虑敌方所有空位，若落子在该空位能使棋局分数比落子前高，则将落子在该空位后的棋局添加至列表，最后输出列表。

```
def possibleMove(self, currentRound, isFirst, early) -> list:  # 输出下一步的所有可能
    output = []
    if self.mode == "position":  # 一层
        if self.winormax:  # 我方一层
            if early:  # 假如只考虑在自己的区域一层
                copyBoard = self.key.copy()
                if len(copyBoard.getNone(not isFirst)) == 0:
                    pass
                else:
                    copyBoard.add(not isFirst, self.key.getNext(not isFirst, currentRound))
                    output.append(copyBoard)
                return output

            elif self.key.getNext(not isFirst, currentRound) != ():  # 当我方可在我方区域下棋则返回
                copyBoard = self.key.copy()  # 复制棋盘并添加下一步位置
                copyBoard.add(not isFirst, self.key.getNext(not isFirst, currentRound))
                output.append(copyBoard)
            for position in self.key.getNone(isFirst):  # 我方落位
                copyBoard = self.key.copy()
                copyBoard.add(isFirst, position)
                if chessboardPoint(board=copyBoard, isFirst=not isFirst, early=early) > chessboardPoint(
                        board=self.key, isFirst=not isFirst, early=early):
                    output.append(copyBoard)  # 我方在我方区域各个 分数要比对方才考虑
```

```
else:  # 我方下棋
    if early is True:  # 假如只考虑在自己的区域一层
        copyBoard = self.key.copy()  # 复制棋盘并添加下一步位置
        copyBoard.add(isFirst, self.key.getNext(isFirst, currentRound))
        output.append(copyBoard)
        return output

    elif self.key.getNext(isFirst, currentRound) != ():  # 当我方可在我方区域下棋则添加
        copyBoard = self.key.copy()  # 复制棋盘并添加下一步位置
        copyBoard.add(isFirst, self.key.getNext(isFirst, currentRound))
        output.append(copyBoard)
    for position in self.key.getNone(not isFirst):  # 敌方落位
        copyBoard = self.key.copy()
        copyBoard.add(not isFirst, position)
        if chessboardPoint(board=copyBoard, isFirst=isFirst, early=early) > chessboardPoint(board=self.key,
                                                                                            isFirst=isFirst,
                                                                                            early=early):
            output.append(copyBoard)  # 我方在我方区域各个 分数要比对方才考虑
        output.append(copyBoard)
```

```
else:  # 移动
    if self.minormax:  # 敌方移动
        for direction in range(4):  # 上下左右
            copyBoard = self.key.copy()
            if copyBoard.move(not isFirst, direction):  # 如果可移动 则添加到output
                output.append(copyBoard)
    else:  # 我方移动
        for direction in range(4):  # 上下左右
            copyBoard = self.key.copy()
            if copyBoard.move(isFirst, direction):  # 如果可移动 则添加到output
                output.append(copyBoard)
return output
```

newmode 函数:

①输入 mode、我方先后手、minormax(相当于我方或敌方下棋)，输出字串 direction 或 position 代表下一步之模式。

②功能：返回下轮决策是落子或移动。

③具体操作为：若 mode、我方先后手、minormax 处于某些状态则返回 direction，反之则返回 position。

```
122    def newmode(mode, isFirst, minormax) -> str:  # 返回下一模式
123        if (mode, isFirst, minormax) in [('position', True, True), ('direction', True, False),
124                                         ('direction', False, True), ('position', False, False)]:
125            return "direction"
126        else:
127            return 'position'
```

minimax 函数:

①输入棋局、当前回合数、minormax(相当于我方或敌方下棋)、mode、先后手、父节点、当前深度、是否为前期和最大搜索深度，输出决策(移动方向或落子位置)。

②功能：利用 minimax 演算法搭配剪枝，返回对我方最优的决策。

③具体操作为：利用递归调用实现深度优先搜索，若深度为最大搜索深度，则为终止节点，为递归函数结束条件，则以该棋局为 node 新增节点，并返回分数。若深度为 0 即创建父节点，考虑所有移动或落子可能(排除落子在敌方后分数随即降低之情况)，对所有可能之棋局调用 minimax 函数后，找出输出分数最高的操作，输出该操作。若深度不等于 0 或最大搜索深度，则先创立以该棋局为 key 之节点并继承父节点之 alpha 和 beta 值，若此时没有下一步可移动或落子则这一步不操作，那么直接对此棋局调用 minimax 函数，但是改为另一方动作，并更新此节点之 alpha 或 beta 值为该 minimax 函数之输出值并输出。若可操作，则逐一进行可能操作，并对操作后棋局调用 minimax 函数，将此节点之 alpha 或 beta 值(min 节点对应 beta,max 节点对应 alpha)更新为 minimax 函数之输出值,若 beta 值小于 alpha 值，则停止循环，循环结束后输出 alpha 或 beta 值(即为该节点之分数值)。

```
def minimax(newBoard, currentRound, minormax: bool, node, isFirst, parent, depth,
            early, searchDepth):  # minimax tree, depth starts from 0
    newRound = currentRound + 1 if newmode(node, isFirst,
                                           minormax) == 'position' and node is not 'position' else currentRound
    # 计算下一轮比分变化
    if depth == searchDepth:  # 搜索到头 直接返回估值结点
        node = Node(key=newBoard, depth=depth, minormax=minormax, node=node, parent=parent, isFirst=isFirst,
                    early=early, searchDepth=searchDepth)
        if minormax:  # 上层是min结点，父结点为max，更新beta
            node.updateBeta(node.value)
        else:  # 本身（下层）是 父结点为min，更新alpha
            node.updateAlpha(node.value)
        return node.value

    elif depth == 0:  # 叶结节点  为了给父结点 更新值域 返回估值结点
        point = -1 * infinity
        output = ()
        node = Node(key=newBoard, depth=0, minormax=minormax, node=node, parent=None, isFirst=isFirst,
                    early=early, searchDepth=searchDepth)  # 生成节点


    elif depth == 6:  # 深度大点  为了加速考虑 这样后面下层也能有初始估值
        point = -1 * infinity
        output = ()
        node = Node(key=newBoard, depth=N, minormax=minormax, node=node, parent=None, isFirst=isFirst,
                    early=early, searchDepth=searchDepth)  # 生成节点
        if node == 'position':  # 非叶节点
            if node.key.getNext(isFirst, currentRound) != ():  # 检查下面还有没有棋可以走
                copyBoard = newBoard.copy()  # 复制给定状态的一份副本
                copyBoard.add(isFirst, node.key.getNext(isFirst, currentRound))
                point = minimax(newBoard=copyBoard, currentRound=newRound, minormax=not minormax, isFirst=isFirst,
                                node=newnode(node, isFirst, minormax), depth=1, parent=node, early=early,
                                searchDepth=searchDepth)
                output = node.key.getNext(isFirst, currentRound)

            for position in newBoard.getNone(not isFirst):  # 为了剪枝
                copyBoard = newBoard.copy()
                copyBoard.add(not isFirst, position)
                if chessboardPoint(board=copyBoard, isFirst=isFirst, early=early) > chessboardPoint(newBoard,
                                                                                                    isFirst, early):
                    # 检查这个落点是否有一定优势，再下手它

                    if minimax(newBoard=copyBoard, currentRound=newRound, minormax=not minormax, isFirst=isFirst,
                               node=newnode(node, isFirst, minormax), depth=1, parent=node,
                               early=early, searchDepth=searchDepth) > point:
                        point = minimax(newBoard=copyBoard, currentRound=newRound, minormax=not minormax,
                                        isFirst=isFirst,
                                        node=newnode(node, isFirst, minormax), depth=1, parent=node,
                                        early=early, searchDepth=searchDepth)
                        output = position
            return output

        else:  # 己方结点
            if len(node.possibleMove(currentRound=currentRound, early=early, isFirst=isFirst)) == 1:  # 只有一个可能的点
                for direction in range(4):
                    copyBoard = newBoard.copy()
                    if copyBoard.move(isFirst, direction):
                        return direction
```

```
        for direction in range(4):  # 上下左右
            copyBoard = newBoard.copy()
            if copyBoard.move(isFirst, direction) and minimax(newBoard=copyBoard, currentRound=newRound,
                                                              minormax=not minormax, isFirst=isFirst,
                                                              node=newnode(node, isFirst, minormax), depth=1,
                                                              parent=node,
                                                              early=early, searchDepth=searchDepth) > point:
                point = minimax(newBoard=copyBoard, currentRound=newRound, minormax=not minormax, isFirst=isFirst,
                                node=newnode(node, isFirst, minormax), depth=1, parent=node,
                                early=early, searchDepth=searchDepth)
                output = direction  # 记录最优方向
        return output
```

```
    else:  # 不是叶子节点，递归计算子节点的 alpha 或 beta 值
        node = Node(key=newBoard, depth=depth, minormax=minormax, node=node, parent=parent, isFirst=isFirst,
                    early=early, searchDepth=searchDepth)  # 创建节点
        node.updateAlpha(node.parent.alpha)
        node.updateBeta(node.parent.beta)
        if len(node.possibleMove(currentRound=currentRound, isFirst=isFirst, early=early)) is not 0:  # 检查下一步是否有可行的移动
            if minormax:  # min 节点的操作
                for i in node.possibleMove(currentRound=currentRound, isFirst=isFirst, early=early):
                    node.updateBeta(min(node.beta, minimax(newBoard=i, depth=depth - 1, minormax=not minormax,
                                                           node=newnode(node, isFirst, minormax), isFirst=isFirst,
                                                           currentRound=newRound, parent=node, early=early,
                                                           searchDepth=searchDepth)))
                    if node.beta <= node.alpha:  # beta 小于等于 alpha 时剪枝
                        break
                return node.beta
            else:  # max 节点的操作
                for i in node.possibleMove(currentRound=currentRound, isFirst=isFirst, early=early):
                    node.updateAlpha(max(node.alpha, minimax(newBoard=i, depth=depth - 1, minormax=not minormax,
                                                             node=newnode(node, isFirst, minormax), isFirst=isFirst,
                                                             currentRound=newRound, parent=node, early=early,
                                                             searchDepth=searchDepth)))
                    if node.beta <= node.alpha:
                        break
                return node.alpha
        else:  # 没有可行的移动
            if minormax:  # min
                node.updateBeta(minimax(newBoard=newBoard.copy(), depth=depth - 1, currentRound=newRound,
                                        minormax=not minormax, node=newnode(node, isFirst, minormax), isFirst=isFirst,
                                        parent=node, early=early, searchDepth=searchDepth))
                return node.beta
            else:  # max
                node.updateAlpha(minimax(newBoard=newBoard.copy(), depth=depth - 1, currentRound=newRound,
                                         minormax=not minormax, node=newnode(node, isFirst, minormax), isFirst=isFirst,
                                         parent=node, early=early, searchDepth=searchDepth))
                return node.alpha
```

前期：

两个辅助函数：

Ⅰ，getourpower 函数：参数为当前棋盘状况 board，是否先手 isFirst，以及行数 row。功能：找出第 row 行我方最靠近对方的数。实现方法：由于前期棋盘状况较简单，并且采用见缝插针，吃了就跑的战术，简化了计算量，从对方分界线的一列开始遍历，利用 board.getBelong 函数，返回第一个我方棋子。称为“攻击棋”

Ⅱ，getdanger 函数：与 getourpower 函数相反来求第 row 行对方最靠近我方的数。详情参

见Ⅰ。得到危险棋。

Ⅲ，评价棋盘函数：ee（early evaluate）：输入棋盘 board，与 isFirst 先后手，返回一个得分（int）。

具体实现：遍历场面上所有我方棋子，累加（2**等级 - 1），这样赋分可以实现等级高一级，分数增加不止一倍，此为基本分。利用前面的两个辅助函数，在我方攻击棋有机会吞并对方处于分界线上的棋子时，额外加上对应的基本分；在我方已经吞并了对方棋子时，即对方棋盘上（主要是分界线上）有我方的棋子时，额外加上 2 倍的对应基本分。反过来危险的时候，即对方达成上述条件时，额外扣相应的分。另外，考虑我方最大等级的棋子是否在棋盘的角落处，如果是，则额外加基本分的 3/4，来达到尽量进入中期时提供好一点的发育环境。

```
def getourpower(board, isFirst, row):
    board1 = board.copy()
    if isFirst:
        for i in range(5):
            if board1.getValue((row, 4 - i)) != 0 and board1.getBelong((row, 4 - i)):
                return board1.getValue((row, 4 - i))
        else:
            return 0
    else:
        for i in range(5):
            if board1.getValue((row, 3 + i)) != 0 and not board1.getBelong((row, 3 + i)):
                return board1.getValue((row, 3 + i))
        else:
            return 0


def getdanger(board, isFirst, row):
    board1 = board.copy()
    if not isFirst:
        for i in range(5):
            if board1.getValue((row, 4 - i)) != 0 and board1.getBelong((row, 4 - i)):
                return board1.getValue((row, 4 - i))
        else:
            return 0
    else:
        for i in range(5):
            if board1.getValue((row, 3 + i)) != 0 and not board1.getBelong((row, 3 + i)):
                return board1.getValue((row, 3 + i))
        else:
            return 0
```

中期估值函数：

①输入棋盘 Chessboard，与 isFirst 先后手，返回一个得分（int）。

②功能：对棋盘的场面局势进行分析，对 isFirst 更有利则返回更高的分数。

算法策略：遍历棋盘，分析处理棋盘。中期估值函数通过对数字得分的精妙设计，实现了优先和成大子，其次注意单调性排序，注意防止我方棋子被吃，和在对方棋盘落子能取得一定优势后选择向对方落子。

③具体操作为：遍历场面上所有我方棋子，把 4**等级数（以确保优先合成等级高的棋子） 加起来得到基础分。如果棋子在对方棋盘获得则计算时等级提升 1.1 并且加 40 分（在不耽误大子合成的情况下，优先向对面棋盘落子）。然后遍历每个棋子四周比他大的最小的和比他小的最大的，若和他本身的相差小于等于二，加上他本身一半的等级分，否则减去他本身一半的等级分（在不耽误大子合成的情况下，优先棋子单调排列）。遍历场面上所有对方等级为 1 的棋子，若四周平均等级比他高 2，则总得分加上 64。（不耽误大子合成的情况下在可以造成一定阻碍时向对面棋盘落子）遍历场面上所有我方棋子，通过 menace 函数检测他是否有被吃掉的危险，如果有则减去他的等级分。（在不耽误大子合成的情况下避免我方棋子被吃掉）返回经过这一系列操作之后的总得分，即为中期棋盘估值。

特殊情况函数：

①isin 函数：判断某棋子是否处于己方的棋盘内，输入棋子位置，输出 bool 值。

②menace 函数：判断此处的我方棋子是否有被敌方棋子吃掉的危险，输入棋子的级别和

位置，若有危险，返回威胁棋子位置组成列表[[row,column]]；否则返回[]。

③ast 函数：为确保函数在遍历的时候不会出现行数或者列数超出范围的情况，进行检验，返回 bool 值。

④search 函数：给定搜索级别值，从给定行列出发逐行搜索，返回搜索到第一个符合的棋子位置(row,column)，否则返回空的元组。其中使用了 ast 函数来确保搜索不会超出范围。

⑤trouble1 函数：给对面捣乱的函数，在领先情况下使用，若对方存在两个较大级别的棋子快要合并的情况，在其中一个棋子旁边插入一个 2 搅乱计划，返回插入 2 的位置(cow,column)。其中使用 search 函数以及 menace 函数来确保搜寻结果以及不会被反吃的情况。

⑥trouble2 函数：同样是给对面捣乱的函数，在落后情况下使用，在 trouble1 可行的情况下优先使用 trouble1,否则，对于落后情况尽量牵制对方，往对方扔 2，返回 2 的位置(cow,column)，尽量达到将对面堵死然后发育自己的计划。

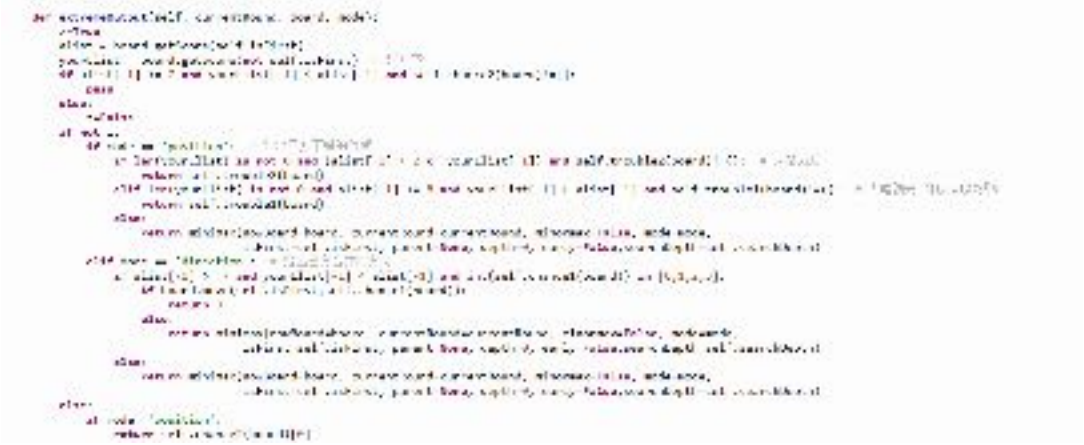
⑦search2 函数：给定某棋子的位置以及级别，查看周围是否有可合并项，返回可合并项位置及方向组成的列表[[cow,column,diraction]]。

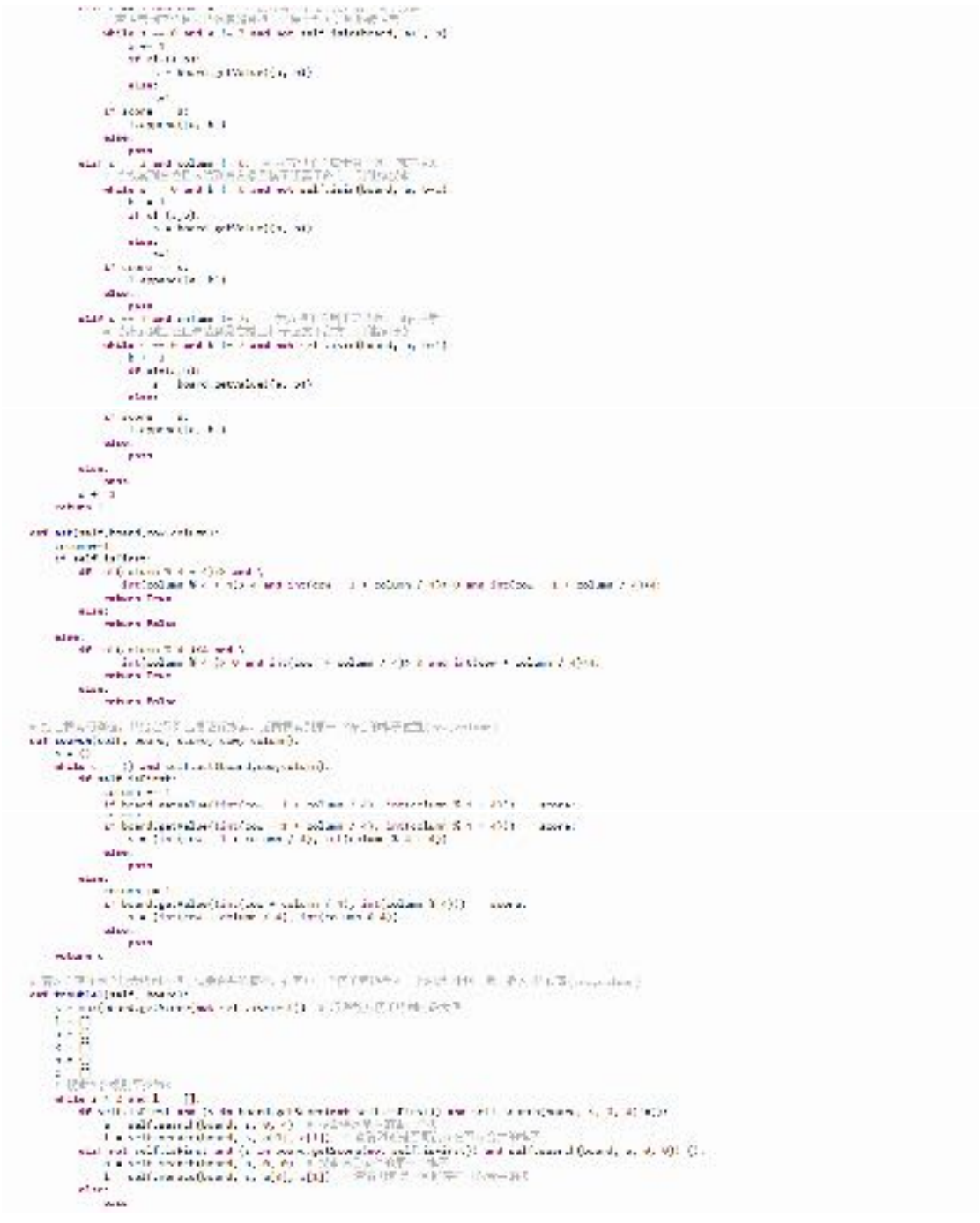
⑧total 函数：给定棋子位置，返回除开空位周围四个棋子的级别之和。

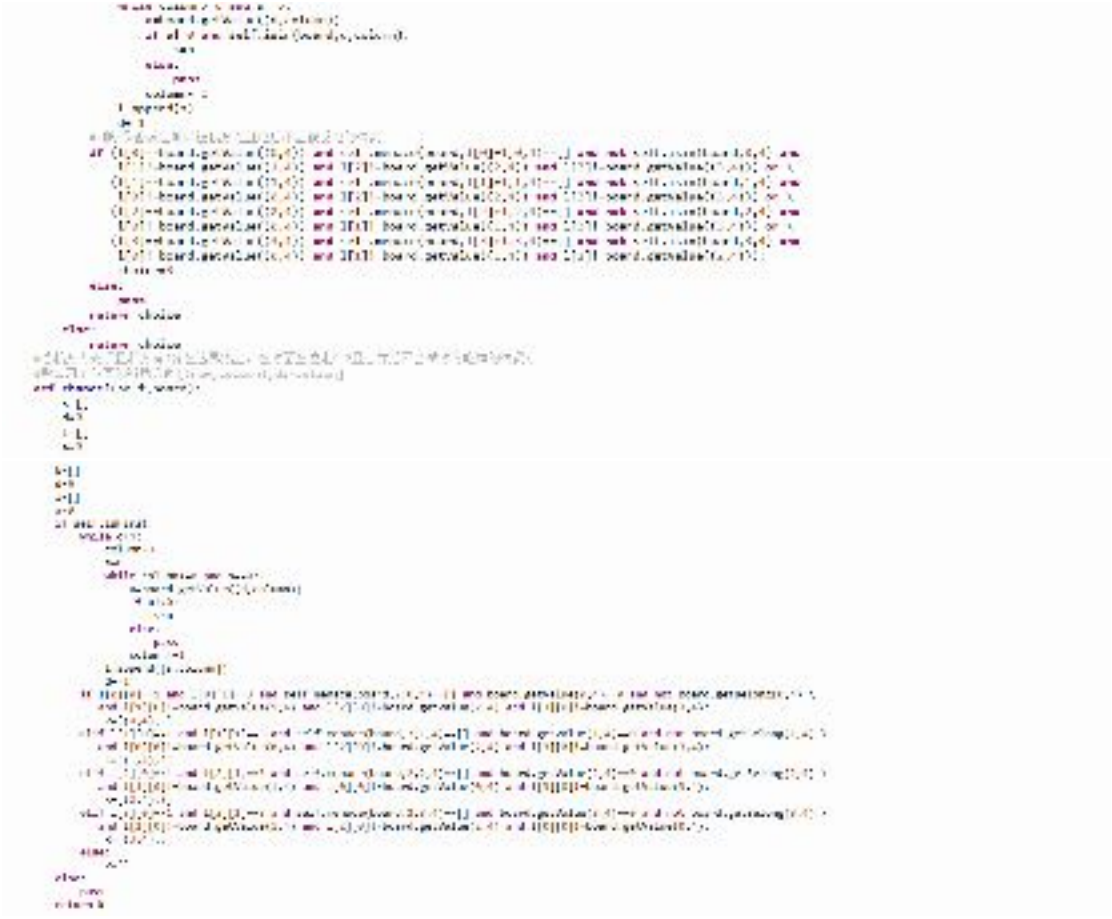
⑨chance1 函数：对于可能存在的对面棋子位于边界线并且刚好可以被我方棋子吃掉不被反吃的情况，返回方向 choice;否则返回 choice=-1。其中调用了 menace 函数来确保不会有被反吃的情况。

⑩chance2 函数：若我方为先手且我方有 2 落在边界线上，在对面空位走个 2 且合并后不会被对方吃掉的情况，返回落子位置和移动方向[(row,column),diraction]。

⑪extremeOutput 函数：对于落后领先的特殊情况特殊处理，先判断是否满足领先以及可以使用 chance2 的情况，如果是，则打包带走落子与移动方向;如果不是的话，那么就判断是否为领先情况，如果是且可以使用 trouble1 则调用，否则调用中期函数，落后情况调用 trouble2 同理，其余情况还是按照一般情况处理。







Output 函数：

①在 Player 类底下定义，输入回合数、棋局、mode，输出落子位置或移动方向

②功能：对棋盘局面进行分析，直接输出决策或是调用 minimax 函数或 extremeOutput 函数返回决策

③具体操作为：若剩余时间过少，则减少最大搜索深度以避免超时，若我方棋盘为空，在我方落子。若 goback 是 True 则先手返回 2，后手返回 3(上局吃掉边界上的棋子，要返回原位)。若 mode 为落子且为前期，在我方落子。若 mode 为落子且敌方已被塞满，在我方落子。若 mode 为落子且我方被塞满，考虑在敌方所有空位落子，选取其中分数最高的落子。对于其余情况，分为四种可能:前期、大幅领先(我方最大等级超过 7 且对方最大等级小于我方)、大幅落后(我方最大等级小于敌方最大等级-1)、一般情况。若大幅领先或落后，调用 extremeOutput 函数返回决策。若为前期或一般情况，调用 minimax 函数返回决策。

```
def output(self, currentRound: int, board, mode: str):
    if board.getTime(self.isFirst) <= 0.6 * maxTime:
        self.searchDepth = 2
    elif board.getTime(self.isFirst) <= 0.35 * maxTime:
        self.searchDepth = 1
    alist = board.getScore(self.isFirst)
    yourLlist = board.getScore(not self.isFirst)  # 敌方手列

    if len(alist) is 0:  # 我应该看
        return board.getNext(self.isFirst, currentRound)

    elif mode is 'direction' and self.goback:  # 我应该返回去
        if self.isFirst and board.move(True, 2):  # 先手
            return 2
        elif not self.isFirst and board.move(False, 3):  # 后手
            return 3
```

```
    elif mode is 'position' and alist[-1] <= 4 and len(alist) != 0 and len(
            board.getNone(self.isFirst)) is not 0:  # 我的对手
        return board.getNext(self.isFirst, currentRound)

    elif mode is 'position' and len(board.getNone(not self.isFirst)) == 0:  # 敌方已经检查满
        return board.getNext(self.isFirst, currentRound)

    elif mode is 'position' and len(board.getNone(self.isFirst)) == 0:  # 我方先检查 敌方还没有
        if len(yourLlist) == 0:
            return board.getNone()[0]
        pointDict = {}
        for position in board.getNone(not self.isFirst):  # 敌方走位
            copyBoard = board.copy()
            copyBoard.add(not self.isFirst, position)
            pointDict[chessboardPoint(copyBoard, self.isFirst, early=False)] = position
            return pointDict[max(pointDict.keys())]  # 返回最高的分数的下棋位置
```

```
if mode in ['position', 'direction']:
    if len(alist) is 0:  # 我应该看
        pass
    else:  # 我的对应
        if alist[-1] < 4 and len(alist) != 0:  # 防御
            return minimax(nowBoard=board, currentRound=currentRound, minormax=False, mode=mode,
                           isFirst=self.isFirst, parent=None, depth=0, early=True, searchDepth=self.searchDepth)
        elif len(yourLlist) is not 0 and (alist[-1] + 2 <= yourLlist[-1]):  # 大幅领先
            return self.extremeOutput(currentRound=currentRound, board=board, mode=mode)
        elif len(yourLlist) is not 0 and alist[-1] >= 7 and yourLlist[-1] < alist[-1]:  # 大幅领先 可以考虑7+8
            return self.extremeOutput(currentRound=currentRound, board=board, mode=mode)
        else:  # 领先不
            return minimax(nowBoard=board, currentRound=currentRound, minormax=False, mode=mode,
                           isFirst=self.isFirst, parent=None, depth=0, early=False,
                           searchDepth=self.searchDepth)
else:  # 一个没得的
    return None
```

2.3 程序限制

程序运行基本不会出错，但是如果进一步加深搜索深度，判断条件过多可能超时。
如果在前期，我方几乎被塞满导致接近无法落子可能会出错，在 minimax 函数中，前期决策树并没有考虑在敌方落子，因为一般情况下，前期双方领域都很空旷所以几乎不可能发生被卡死的情况，因此在设计时似乎没有考虑这个问题，但是在实战中也没有发生过。

3 实验结果

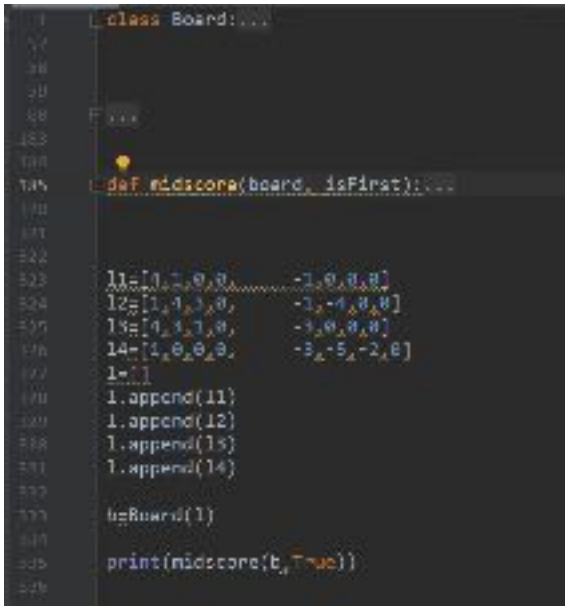
3.1 测试数据

实验环境说明：

- 硬件配置：intel® Core(TM) i7-8565U CPU @ 1.80GHz 1.99GHz　内存 8.00GB
- 操作系统：Win10 家庭版
- Python 版本：Python3.82

测试方式：

1、 对中期函数采用了样例输入使其输出分值的情况来调整参数。
2、 在提交热身赛之前组员们由于设备问题远程操纵进行 debug。
3、 在提交到热身赛后通过观察局势复盘来发现问题，对各种函数与参数不断进行改进优化。



上图为中期测试的函数，Board 类为自己写的简略的 Chessboard，通过这种方式对中期函数的参数进行调整，最终确定了最适合比赛的参数结果。

中期函数初始版参加了 370 场比赛，其中胜出 67 场，失败 303 场。

中期函数最终版参加的 410 场比赛中，胜出 87 场，失败 320 场，平局 3 场。比赛均为先后手交替，先手胜出 47 场，后手胜出 40 场。在失败的比赛中，只有 7 场是因为超时而失败。

测试结果：代码运行正常，稳定性也很好，就是竞争力不强。

3.2 结果分析

中期函数初始版：在我们复盘的比赛中，发现一直保守的行为并不能有效获得比赛胜利，有时候一个小棋子占领了最大棋子应该在的地方，导致堵塞，逐渐死亡，而且也很难招架住对方的进攻。这造成了我们应该保护的棋子没有在应该的位置，出现两个较大的相同等级的棋子之后不能有效的进行合成，反倒成为一种阻碍。经过组员讨论，我们放弃了这种猥琐发育，不去进攻，发育也发育不好的策略。

特殊情况处理初始版：最开始的时候人脑并没有考虑出太多的特殊情况，也并没有考虑到多种情况并存的情况，在利用天梯调试的时候不断发现出问题，然后改进，最初没有 ast 函数，然后时常在调用特殊函数时出现超出范围的情况。当时还设计了一种专门深入对面内部的战法，但最终经过讨论决定还是稍微保守一点，于是舍弃了深入敌方内部的函数，并且删除了一些无用的判断。

中期函数最终版：经过改进的中期函数胜率有了提升，其可以实现在变化移动中逐渐壮大自己，但依旧存在把自己堵死的状况，对对面的进攻也较为有效，在有好机会的适合也能实现向对面棋盘落子阻挡对面发育。这一种相对比较激进的算法的稳定发挥也让我们的代码可以去打比赛了。

特殊函数最终版：删除了深入敌方内部的函数，删去了许多无用或者可以被简化的判断，删去了后手时偷吃对面的情况，尽可能做到简化，同时增加了对于多种情况并存的判

断，在面对更复杂情况时也能做出判断。

算法的运行时间可以控制，基本不会超时，中期函数会花费大量的时间，特殊情况处理还不够完善，冗杂的判断条件还是过多。

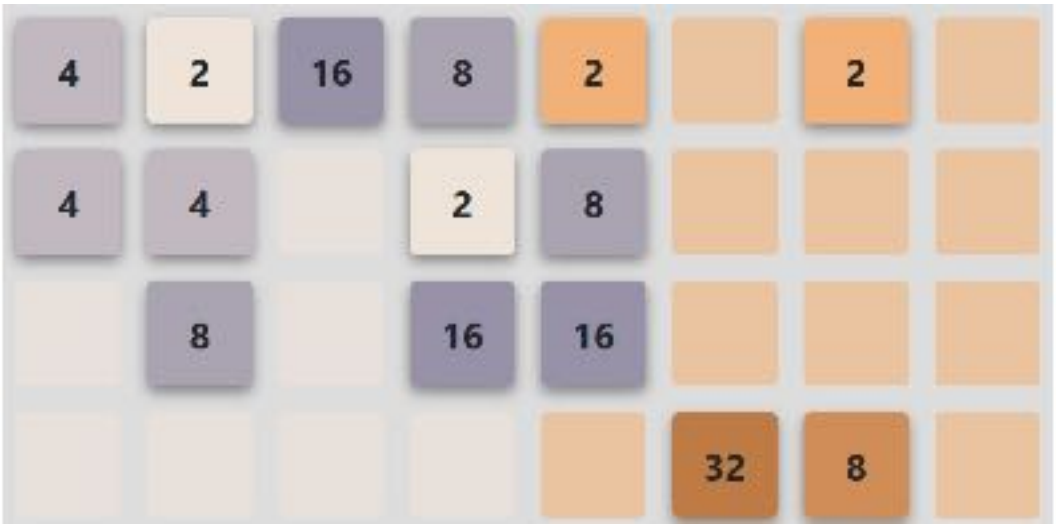
3.3 经典战局（可选）

在这一局我方为先手，上演一把欲擒故纵。

1、暴露破绽，在坐标（2，3）处的 4 去吸引对方来吃

2、对方向左移动，吃掉我方诱饵 4

3、我方露出獠牙，向上移动，将对方两个 8 全部吃掉。真是一出欲擒故纵的好戏！



4 实习过程总结

4.1 分工与合作

小组分工：
决策树的实现：高玮伯，易佳怡
前期评估函数：金佳伟
中期评估函数：马超凡
特殊情况处理：吴熙楠

小组交流的方式主要为腾讯会议与微信群，由于篇幅问题，只列出两次会议的截图，微信群聊天记录不再列出。

第一次会议在 5 月 14 日晚，大家在会议之前准备了自己的想法方案，准备好在会议上进行讨论。会议决定了我们的总体方略，会议之后要求同学们对各个部分进行细化，以方便进行下一次会议的分工完成。

第二次会议于 5 月 16 日晚，大家都拿出了自己的细化方案，在会议中我们进行讨论交流，在各个方面选出来大家认为最好的细化策略。在会议上大家进行了分工，在接下来一个星期的时间里去完成各自的代码。

第三次会议于 5 月 23 日晚，大家都拿出来各自的代码，由于设备原因开始远程 dubug，将各自的代码修改完毕后组合成我们最开始的总代码。

之后在微信群里，大家不断发现问题，解决问题，讨论交流，改进各自部分的代码。在天梯赛结束后，大家在微信群里讨论共同完成实习报告。

4.2 经验与教训

1、 这次我们的天梯赛没有取得一个比较好的成绩，现在反思下来应该是我们的估值函数过于复杂，导致花费的时间过多，使得决策树的层数非常浅。这是我们最开始的理解和讨论没有充分建立在实验的基础上造成的，希望小组成员都能够吸取这个教训。

2、 我们这次大作业实行了比较多的线上会议，效果还是非常好的，交流是必要的，群策群力也是必要的，这样我们才能够有比较好并且全面的方案实施，在进行充分的讨论交流之后我们才能更好的开展接下来的工作。

3、 另外，这次中期函数依赖于数值的选取，需要把数值设置的非常精妙，这虽然看起来比较简单，但在实际操作的适合会有很多问题，数值往往难以抉择。不要过分依赖数值的选取是一个教训。

4、 考虑情况较少，对于复杂战局墨守陈规，有大家任务都重的原因，但应该考虑情况更细致。

5、 Minimax 函数每次调用时都要重新制造节点 Node，可能会太过消耗时间，也许可以在制造节点时保存更完善的信息，让下一回合时，能够继续使用其中的部份节点。

4.3 建议与设想

这次实习作业的规则公平性做的非常好，基本上保证了随机因素的最小。但是实习作业临近期末，大家都比较忙，可能没有过多的时间去精致打磨这个实习作业，建议就是希望能够早点开展这个竞赛，尽量不要放在最后。

此外，这学期的课程都是线上授课非常可惜，虽然线上上课时间和空间上相对比较自由，但是却没能体会到良好的课堂气氛，尤其是最后的决赛，如果能在现场看两方代码激烈对战，一定非常刺激且欢乐，希望下学期能继续举办大作业决赛，让我们能去参观。

希望以后的学弟学妹们要有扎实的数理基础。陈斌老师的数算课虽然任务量较大，但是一学期下来是非常有收获的，一定要认真去学，积极参与每一次的活动。

5 致谢

感谢陈斌老师这一学期的指导！
感谢各位助教和技术组的成员为我们做的工作！
感谢我们组的每一位同学对组长的忍耐！
感谢所有组员们都有在 DDL 前完成自己份内的工作！
感谢所有参赛的同学，你们的参与让我们看到了自己的不足及能改进的地方！
感谢我们的电脑，他在这几周不眠不休的努力，非常辛苦！
感谢互联网，让我们即使在五个不同城市也能合作，也让课程能在线上顺利进行！
感谢各位大佬的指导！

6 参考文献

课程网站：http://gis4g.pku.edu.cn/course/pythonds/
https://en.wikipedia.org/wiki/Minimax
https://en.wikipedia.org/wiki/Alpha%E2%80%93beta_pruning
http://programmermagazine.github.io/201407/htm/focus3.html
https://fu-sheng-wang.blogspot.com/2017/02/ai-14-adversarial-search.html
http://blog.codinglabs.org/articles/2048-ai-analysis.html

数据结构与算法课程实习作业报告

杨城冉* 李子涵 卢韬 石霆章 张建学 （组员按名称首字母排列）

【摘要】本文是F19-996组在SESSDSA’20课程的大作业上的实习报告。本组编写的AI采用了minimax和alpha-beta剪枝的算法（采用决策树实现，搜索深度最大为9层），辅以启发式规则。由于对2048的策略等了解不足，最终的小组成果并不十分理想。

【关键字】决策树 minimax alpha-beta

目录

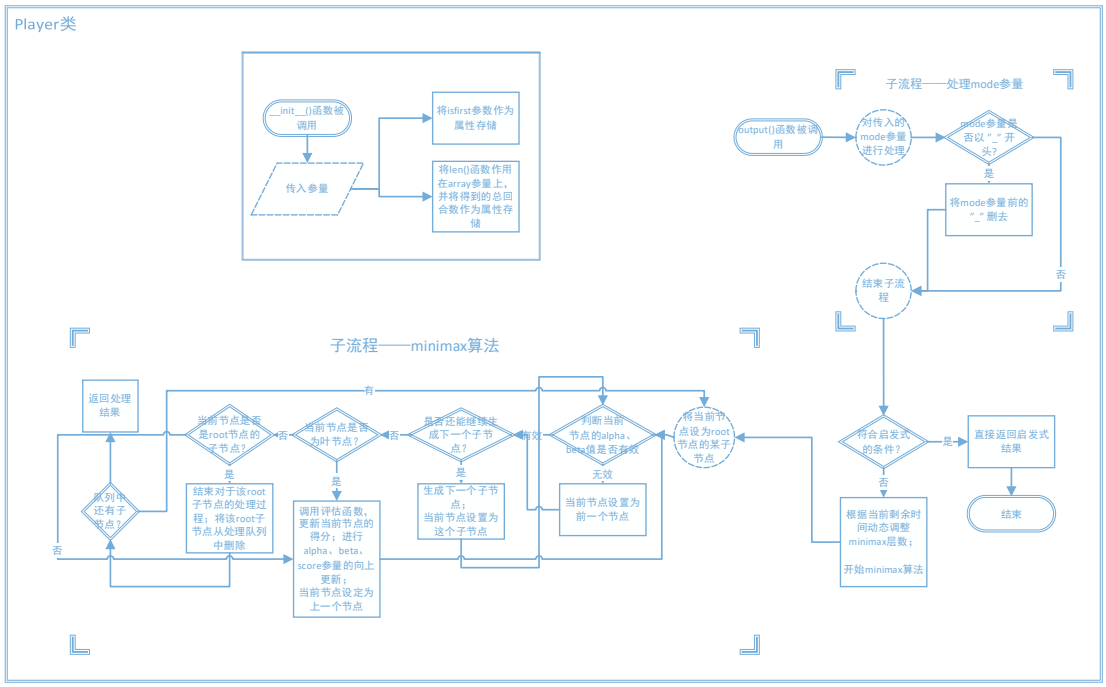
1 算法思想.....2
1.1 总体思路.....2
1.2 算法流程图.....2
1.3 算法运行时间复杂度分析.....2
2 程序代码说明.....3
2.1 数据结构说明.....3
2.2 函数说明.....3
2.3 程序限制.....3
3 实验结果.....4
3.1 测试数据.....4
3.2 结果分析.....5
3.3 经典战局（可选）.....6
4 实习过程总结.....7
4.1 分工与合作.....7
4.2 经验与教训.....9
4.3 建议与设想.....10
5 致谢.....10
6 参考文献.....10

1　算法思想

1.1 总体思路

本次大作业的特性是：每一步可以走的方式十分复杂，大多数行动的行为价值较低，但对于每一个给定的状态（array、currentplayer、currentround、mode、board），都有确定的发展路线；不存在过程中的偶然性（array 是状态参量，而非过程中的偶然性）。基于这些分析，我们决定采取 minimax 算法，同时配合 alpha-beta 剪枝来完成决策。同时，由于 5 秒的总决策时间限制，在过程中我们会根据当前剩余时间来动态调整 minimax 算法中决策树的遍历层数。

1.2 算法流程图



（若过于模糊，在“照片与流程图”文件夹下有 .vsdx 以及 .svg 格式的流程图文件）

1.3 算法运行时间复杂度分析

由于本程序采用了 alpha-beta 剪枝以及每个回合的参量的不确定性，此程序的时间复杂度没有确切的数值。但是，在最坏的情况，即 alpha-beta 完全失效，而且启发式规则未匹配时，时间复杂度大约为 O(8n)（测试方法：注释掉起 alpha-beta 剪枝和启发式规则功能的代码，与随机 AI 测试，并记录生成的总节点数）。在通常情况下，经过测试，平均时间复杂度为 O(2n)（测试方法同上）。在启发式规则起作用的情况下，时间复杂度为 O(1)。其

中，n 为搜索深度。在正式天梯上，默认的搜索深度为 8。

2 程序代码说明

2.1 数据结构说明

本代码的天梯提交版本没有自定义类，只有一个 Player 类；所有节点信息都存储在一个大字典中。但是，为了保证组内快速开发的能力，本小组还维护了一个包含决策树类、节点类的代码（以下称为快速开发代码）。在经过大量测试，保证该具有快速开发能力的代码基本没有问题的情况下，我们经过翻译，通过一个字典存储了所有的节点，以加快执行速度（以下称为快速执行代码）。

2.2 函数说明

快速执行代码：
①output：通过接收当前轮数，当前棋盘和模式，给出本代码的相应决策
②inspire：启发式规则函数
③add_node：通过接收当前棋盘、上一次操作者、模式、当前轮数、上一次操作等参量，向决策树中添加节点并返回节点编号
④evaluate：通过接收当前轮数、棋盘、所属、当前是否为自己操作，给出本轮棋盘估值，主要依据为棋盘的棋子大小和分布特征
⑤gen_next：返回和 add_node 添加的数据结构列表前五个参量一致的列表（其实就是生成新节点的内容），是个迭代器
⑥recur_for_min_node：流程图中的 minimax 算法对 min 节点的作用函数
⑦recur_for_max_node：流程图中的 minimax 算法对 max 节点的作用函数

2.3 程序限制

本程序中进行了大量多态的处理，不会因为最大回合、无可用操作等因素而抛出异常。但是，本程序的动态深度功能是针对偶数层的深度而测试的；对奇数层的深度可能支持能力不够。因此，当层次为奇数时，可能会出现某些未知的问题。

3 实验结果

3.1 测试数据

实验环境说明：

环境 1（高性能环境）

硬件配置：AMD Ryzen 5 3500U with Radeon Vega Mobile Gfx 2.10GHz，内存 8.00GB （4G DDR4 2400MHz x2）

操作系统：Windows 10 专业版，64-bit

Python 版本：3.8.1_x64，3.7.6_x64

环境 2（低性能环境）

硬件配置：AMD Ryzen 7 2700X Eight-Core Processor 3.70GHz，内存 16.0GB （8G DDR4 2400MHz x2）

操作系统：Windows 10 专业工作站版，64-bit

Python 版本：3.8.1_x64

测试方法：

1、本地使用 userinterface.py 与默认 AI 对战，并打印出每次决策消耗的时间；同时，打印出字典长度，来确定总生成节点数。

2、热身赛上与官方 AI 和随机对手自由对战，不断调整参数，优化性能。观看对局情况，寻找好的启发式算法、估值函数。

（表 1：热身赛情况）

版本名称	最大稳定遍历层数	热身赛测试局数	胜场数	胜率	与 B 级 AI 胜率
普通决策树版本	2 层	167	34	20.3%	0%
决策树+剪枝+启发式版本	5 层	196	42	21.4%	10%~20%
调参优化版本	8 层	230	106	46%	80%左右

（表 2：比赛结果统计）

版本名称	比赛结果
普通决策树版本	超时（36%）、下子违规（5%）、正常结束比赛
决策树+剪枝+启发式版本	正常结束比赛
调参优化后	正常结束比赛

3.2 结果分析

本算法的原始版本为普通决策树版本，稳定遍历层数为 2 层，由于使用了较为复杂的估值函数（计算棋盘棋子的单调性、平滑性），导致每次估值消耗大量时间。再加上决策树并未有效利用剪枝，导致决策时间容易超出预期，经常出现超时的情况。搜索深度较少，导致做出的决策并不十分优秀，胜率也很低。

经过翻译后的快速执行版本有效利用了 alpha-beta 剪枝，将搜索迭代深度提升到 5 层。估值函数总体思路不变，在数据结构上做了一些小小的优化，如手动求方差。主要的运行时间开销仍发生在对棋盘的估值上。引入了启发式算法：在 direction 模式下首先判断能否直接吞并对方棋子，若可以则直接吞并，不再调用搜索树；如果不能直接吞并对方棋子，则判断是否需要防御边界，避免对方棋子的吞并，如果需要则返回相应的防御方向，不再调用搜索树。改进后的 AI 胜率有所提升，不再有超时现象，但面对较强对手时仍然惨败。在本版算法的对局分析当中，我们发现了两大问题：

估值函数过于注重单调性和平滑性，影响了棋子本身的权重，导致出现能合并但不选择合并的情况。

启发式算法基于贪心思想，在直接吞并对方棋子时，经常选择的并不是最优的方向，没有对对方棋盘的基本形式进行判断，导致出现“螳螂捕蝉，黄雀在后”的局面，损失大子，得不偿失。

最后版本的 AI 引入了动态设置搜索深度的功能：对于当前棋局先判断搜索需要的时间，再决定搜索的最大层数，有效地利用了时间资源，将最大搜索深度提升到 8~9 层。对于搜索进行了进一步优化：在 position 模式下搜索对方棋盘的落子位置，通过先验经验判断落子的合理性，再将该棋盘加入搜索树，极大程度地减少了许多不必要的搜索分支。对估值函数进行了优化，部分舍弃了单调性和平滑性的判断，增加了对棋子等级判断的权重（因为搜索层数足够深，单调性和平滑性对最终结果的影响并不大）。不必要的落子和原本较为复杂的估值函数的时间开销大幅减少，主要时间开销发生在搜索树上，本版算法胜率明显提高。在启发式算法的策略上，舍弃贪心策略，因为在搜索层数足够多时，贪心策略已经没有必要，且得出的决策不一定最优，很大程度上影响本方得分。启发式算法主要为在将对方棋盘落子加入搜索树前的先验判断，以减少搜索分支个数。

在估值函数的优化上，采取总分估值法，将本方棋子得分设为正，取 3 的次幂，敌方棋子得分设为负，取 2 的次幂。为了体现大子的重要性，分别将己方与敌方第一大子和第二大子再次进行运算，第一大子取 4 的次幂，第二大子取 3 的次幂，使得最大子尽量大。测试结果符合预期。在对局结束阶段，多数情况下我方最大子等级与对方最大子等级相当。估值函数的优化有效提高了胜率。

经过天梯实战发现，本算法仍有很大的提升空间。本 AI 在实战中表现出的进攻、防守、发育能力均衡，但面对攻击性较强的算法往往会败下阵来。在原算法的框架上，如果继续增加搜索层数，使用更有效的启发式算法，增强 AI 的进攻性，则 AI 的决策应当可以进一步优化。

1：动态设置搜索层数

```
if board.getTime(self.isfirst) < 0.4:
    max_layer = 2
elif board.getTime(self.isfirst) < 0.8:
    max_layer = 6
else:
    max_layer = 9
```

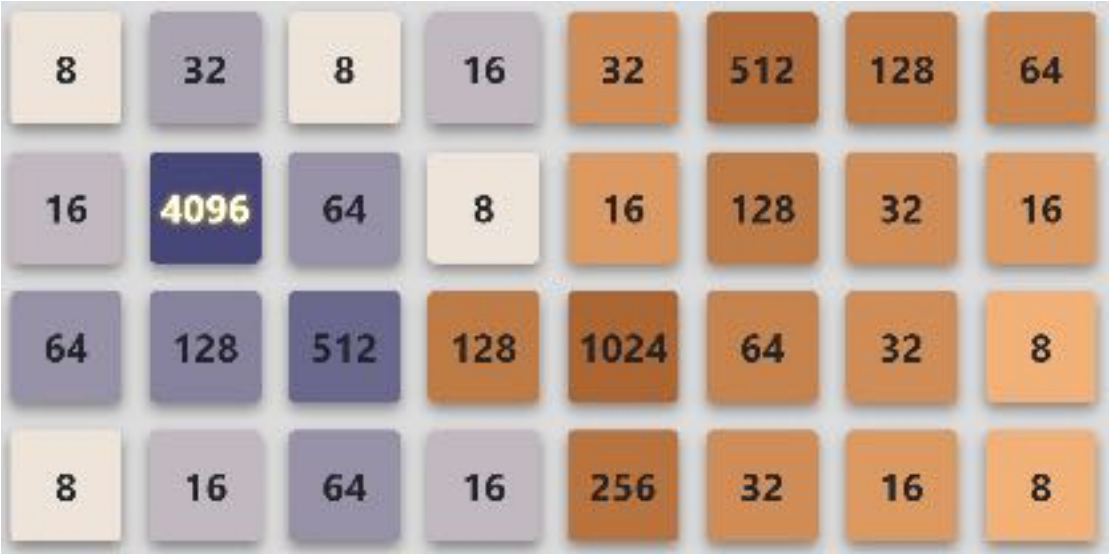
2：优化后的估值函数

```
def evaluate(round, board, belong, is_our_round):
    self_score = board.getScore(belong)
    enemy_score = board.getScore(not belong)
    result = 0
    for i in self_score:
        result += (3 ** i) ** 1.5
    for i in enemy_score:
        result -= (2 ** i) ** 1.8
    if len(self_score) > 0:
        result += (4 ** self_score[-1])
    if len(enemy_score) > 0:
        result -= (3 ** enemy_score[-1])
    if len(self_score) > 1:
        result += (3 ** self_score[-2])
    if len(enemy_score) > 1:
        result -= (3 ** enemy_score[-2])
    return result
```

3.3 经典战局

搜索层数加深 + evaluate 函数优化后：有效增加终局的大子等级

例：与官方 AI，2048Bot_B 胜局(紫色为本方)



4 实习过程总结

4.1 分工与合作

小组分工：

杨城冉：代码基础框架的设计和编写、快速开发代码向快速执行代码的翻译、报告算法部分的撰写

石霆章：估值函数的编写、代码参数调节

卢韬：算法性能测试、报告测试部分的撰写、启发式算法开发、代码调参

李子涵：算法思想的完善

张建学：报告代码说明部分的撰写、实习过程记录

交流方式：

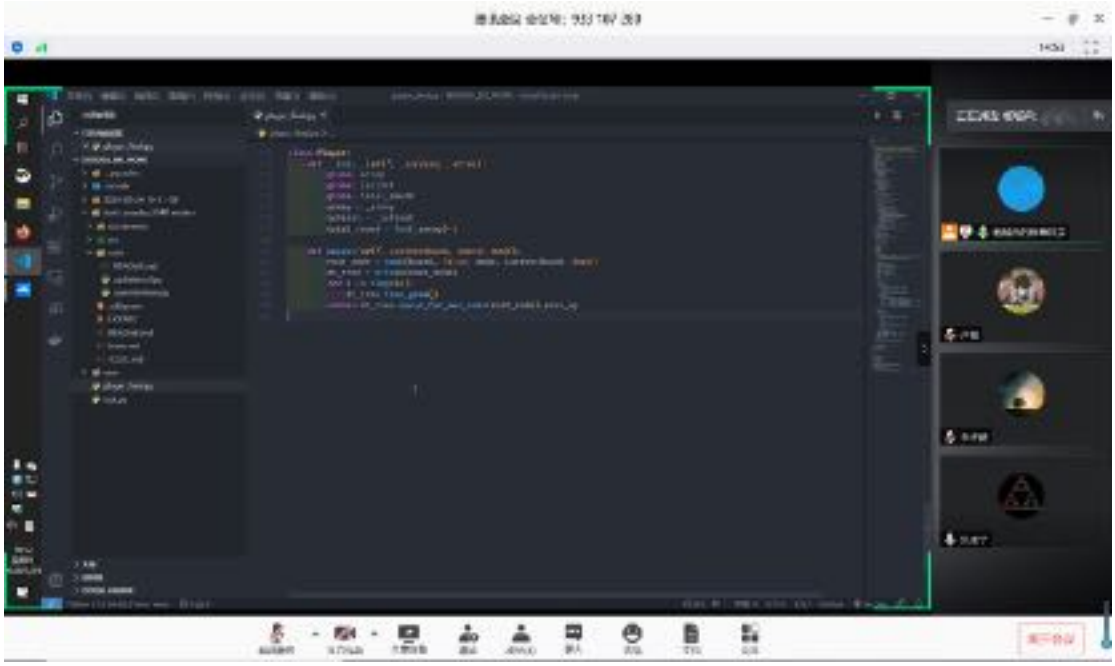
腾讯会议、微信语音通话

历次组会记录：

第一次组会：代码总体架构分析与分工

第二次组会：对代码总体架构进行了进一步讨论，进行了一定地修改

第三次组会：对最新架构的代码进行了分析，提出了一些优化方案。同时，对大作业报告进行了分工



第四次组会：代码性能测验、优化分析

4.2 经验与教训

杨城冉：

在接触一种新的算法的过程中，普通人（不是 dalao 的那种）会或多或少的会犯类似于架构不清、盲目编写代码的错误。不过，要想克服这些问题并不是不可能的。通过不断地实习各种项目并积累经验，一个人可以对自己正在写什么、自己应该采取什么样的架构有一个更清晰的认识。本次 AI 编写过程中，我们五个人之前都没有这种博弈算法的编写经验，因此走了不少弯路。比如在当初规划整体框架的时候，我认为只用一个类是不可能实现 alpha-beta 剪枝的；必须有很复杂的架构才能实现功能。最后，经过几次重构之后，我对这个算法有了比较清晰的认识。当时我就反思：是不是架构必须这么复杂呢？经过调整和删改，从现在看来，只用一个字典实现 alpha-beta 剪枝是能做到的。

不过，我对某个我定下的决策产生的效果持怀疑态度。在开放天梯后，我们组的表现不佳；当时我对接下来的决策是：停止进一步的开（nei）发（juan）调（lian）参（dan），并让我们组的成员去进行其它课的任务（比如通选课、英语课的论文等等）。当时我的判定依据是：第一，作为一个学生，不可能不把成绩放在心上，这个大作业对成绩促进作用的效费比可能太低了，并且好像组内对 2048 这个游戏的热情并不高，我们用爱发电并不太现实；第二，我们已经基本完成了预先定下的任务：遍历层次达到 8 层，并且有可接受的执行速度，各位继续调参的动力应当不是太大了。但是，我在做这个决策的时候其实太过急切了，只询问了部分组员的意见，掺杂了一些个人主观观念。这是我对这个决策产生的效果持怀疑态度的原因。

有一点是十分肯定的：这次大作业的确对我有着巨大的锻炼作用。通过这一次难能宝贵的组内协调和程序编写工作，我对程序开发和小组合作有了更深刻的认识；我一定会珍视这种可遇不可求的经验的。

张建学：

进行 coding 之前一定要事先对大作业题目进行充分地分析，对代码整体架构进行充分的思考，不要急于开展代码的编写。本组在作业期间就曾经修改过一次代码总体架构，其原因就是没有对大作业的算法进行很好地思考。导致前期大部分代码被弃用，因此浪费了很多时间。大作业代码参数调节需要耐心（虽然确实很迷幻），经过组员的耐心调整和建议，我们终于把相应的参数调整基本完善。

一项大作业的完成需要每一位成员的努力，这时候小组分工就显得尤为重要。只有让每个人的长处在小组中都得到发挥，每一个人才会在这次作业中有所收获，整个小组也就会变得更

强大。

4.3 建议与设想

张建学：

由于时间比较紧张，大作业代码完善得还不足。比如最大搜索深度还有待提高，启发式算法开发得还不完善。因此，强烈建议数算大作业的准备时间可以多一些，这样大家都可以在其中收获更多的知识和经验。

组队机制不妨可以考虑随机组队，不然各组水平不一，有失公平。

寄语：地空数算课是非常有意思的一门课，你在这里不仅可以从身边的同学们收获很多知识，还可以在实践（大作业、开源硬件）中锻炼自己。

杨城冉：

数算课把大作业的竞赛分占比降低，确实是良苦用心。在这一过程中我们积累了许多的经验，而不是单纯的为了分而卷。虽然说最近课改风波比较大，但还是希望老师能继续把这个形式的大作业坚持下去！

5 致谢

感谢为我们大作业倾注了太多心血的组长和每一位付出的组员，尽管结果不尽如人意，但我们都为之努力过！

6 参考文献

[1] Minimax with Alpha Beta Pruning，http://web.cs.ucla.edu/~rosen/161/notes/alphabeta.html

数据结构与算法课程实习作业报告

牛沛然*、朱新宇、张植竣、赵轩、刘新龙

【摘要】如何在快速合并的同时尽可能多的占领对方的地盘，是二〇四八对战的时候的一个关键。我们组算法的总体思路是：落子只考虑本轮最优，然后对每一个可行的局面用估值函数进行估值并部分结合α-β剪枝建立搜索树，来确定合并方向，其中估值函数的设置中加入了一些启发式规则。为了防止超时，我们也对比赛中不同时期的策略进行了调整。实验数据结果显示，输掉比赛的情况多是因为抢占地盘的能力不及对方。

【关键词】搜索树，剪枝，估值函数，进攻，占领地盘

1 算法思想.....1
1.1 总体思路.....1
1.2 算法流程图.....3
1.3 算法运行时间复杂度分析.....4
2 程序代码说明.....4
2.1 数据结构说明.....4
2.2 函数说明.....5
2.3 程序限制.....5
3 实验结果.....5
3.1 测试数据.....5
3.2 结果分析.....12
3.3 经典战局.....13
4 实习过程总结.....15
4.1 分工与合作.....15
4.2 经验与教训.....17
4.3 建议与设想.....18
5 致谢.....18
6 参考文献.....18

1 算法思想

1.1 总体思路

我们对二〇四八对战游戏的理解是：前期双方的棋子等级都比较低，此时最重要的是快速合并棋子，同时防御对方，防止己方较大棋子被对方吃掉，到后期双方棋子等级和数量都会变高，此时在尽量保证合并速度的同时，还要考虑做好防御以及积极进攻占领对方地盘。我们认为如果能占领对方地盘，则能在一定时间内形成优势，使对手的落子和合并受到极大

限制，尤其是如果能使对方处于塞满的状态，对手在后面的一定回合中无法进行合并和在己方区域落子，从而使己方获得相对优势。因此，前期的合并速度以及在后期如何更高效的对对方展开进攻和如何更有效的防御对方的进攻，会对比赛的形势造成较大的影响，从而影响比赛的结果。

基于这样的理解，我们的算法的总体思路是：落子只考虑本轮最优，“最优”的意思是用一个“算分”的函数分别计算自己和对手初始和落子并合并一步后的分值，然后计算这个分值的增量，落子位置要使得己方的增量尽可能最大同时对方的增量尽可能最小；下棋过程中最关键的是合并方向的选择，我们的总体思路是采用估值函数对所有可行局面进行估值来创建搜索树以确定合并方向，并借助α-β剪枝增加搜索层数（虽然其剪枝效果也不是太理想）。估值函数是算法最重要的一部分，在计算时我们对棋盘上每一列上的棋子都赋予了一个权重，越靠近对方的列上的棋子的权重越高。这样边线附近的棋子重要程度高于其他列的同等级棋子，以实现损失在一定范围内优先向对方进攻的策略。

并且在比赛的不同时期，我们对算法进行了调整：

前期：首先是落子不考虑对方棋盘上的空位，只在己方棋盘下棋，因为在对方棋盘落子的原因是为了扰乱对方合并的过程，阻止对方较大的棋子发生合并，以降低对方的收益，但前期双方棋子的等级都比较低，这时候如果在对方棋盘下棋，不但起不到很好的阻碍作用，反而会增加对方的合并速度，使己方陷入相对劣势。其次由于前期空位较多，可行局面的总数较大，为了不超时适当减小搜索的层数，同时此时主要进行合并，不考虑进攻，所以对估值函数也进行了调整。

后期：后期扩大了落子范围，落子位置通过计算落子并移动后己方和对方的“分值”来确定。适当增加了搜索层数，同时对估值函数进行了调整，给不同的列上的棋子赋予了权重，加上了优先向对方进攻的策略，以实现对对方地盘的侵占。

那么现在就谈一下合并的思路，其实也就是我们代码的核心思路，并讲述一下这个思路逐渐演化成熟的过程。第一代算法是采用了经典 2048 的玩法，不考虑进攻和防御等，专心在己方领地合并棋子，将较大的棋子都堆放在一个角落。这一代算法比较简陋，虽然对战随机 AI 有一些优势，但遇到进攻型的就会被打得很惨。当我们将代码放到天梯上的时候，就体会到了这个游戏的竞技性和对抗性；完全不考虑对手的情况，只自顾自地合并是不可取的，因为在对手堵子（就是在对方棋盘落子的操作）以及对手侵略地盘的干扰下，会严重阻碍到我们自身的发育，导致自己的分数严重低于对面。

受到了社会的毒打之后的我们，在第二代版本中加入了一个大的改动，那就是防御机制。这个防御机制我们现在称之为“怂到极致的防御”，就是在原来的合并规则基础上，判断这一步是否会有被对面吞子或者是被侵占地盘的风险，如果有，那么就改变我们的合并方向。这是一个完全不考虑后果的防御机制，某种意义上就有点当年的清王朝闭关锁国的意味。而在实战的初期，我们的这个防御机制竟然取得了一定的成效，这让我们受到了很大的鼓舞。而现在再回想这个机制能够起效的原因，也许就在于当时的竞技场整体水平还不高，那么在这样大家都不太会进攻和防御的环境下，我们防御力点满的战略自然就不会吃亏，甚至还能高人一等。但问题也就出现在了这里：一旦遇到一个擅长攻击且同时还能正常自我发育的人，那么我们的战略就无疑是故步自封了；因为一味地躲避让步，会导致我们在己方棋盘被占领后就逐渐失去自我发育的空间，从而输掉比赛。除此之外，我们也加入了一定程度上积极进攻的功能机制，但实际上并没有起多大的作用，就是因为我们主要的精力还在防守上。

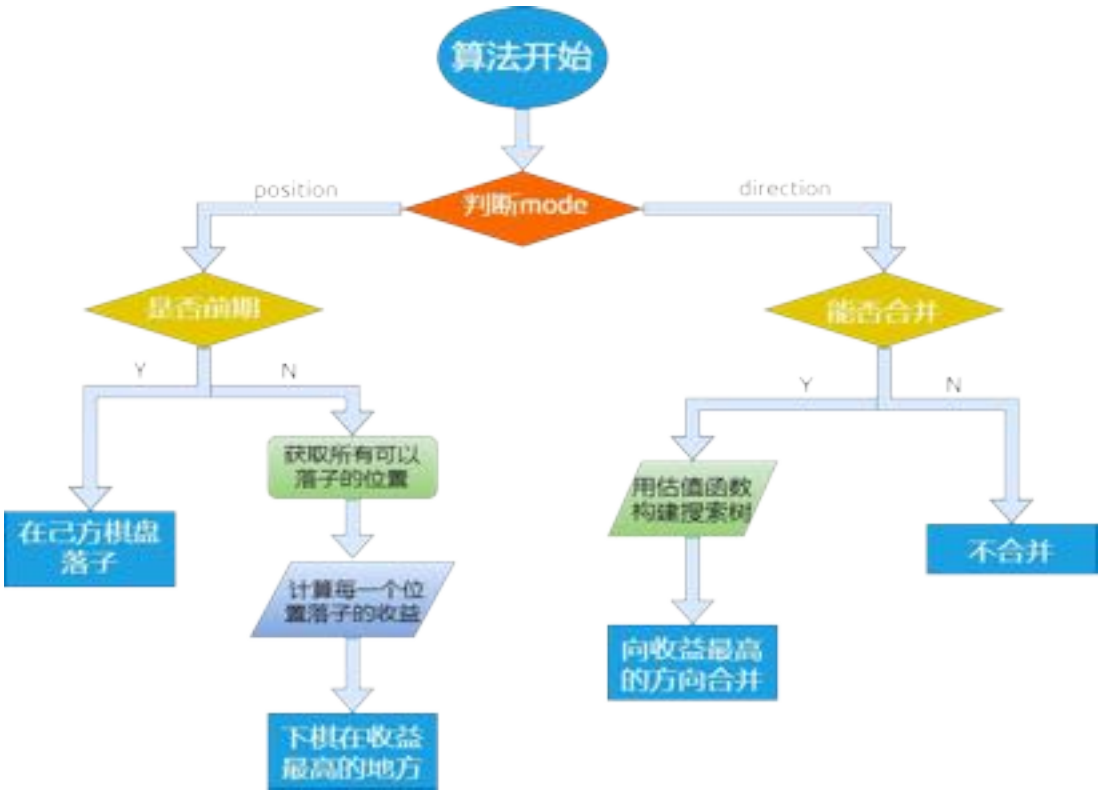
第三代算法对第二代的防御机制做了优化，同时优化了最后一步操作会违规的问题，然后加入了二叉搜索功能，积极进攻的功能仍然保留。但测试出现了 bug，经过一段时间调试后解决，算法略有提升。最后在这个方案上传没到两天，就出现了一个和预想中一样，既善

于自我发育，同时还能积极进攻的对手。在毫无悬念地被吊打之后，我们开始尝试改进代码，经历了几天的讨论后，我们确定了几个基本的关键点。首先，参考老师上课讲授的关于象棋的内容，经我们小组讨论后，我们认定在这个竞技对抗游戏中，每一个位置上不同的棋子都有不同的价值，因此存在一个随位置和棋子大小变化的估值函数，来帮助我们判断战局情况；其次，这个游戏的核心在于对抗，本质在于 2048，因此在积极进攻可以将我们对对抗的理解优势发挥到最大化，而通过树的搜索将己方分数抬高，保证对局过程中良好发育则为至关重要，而这样理解的话，落子就成为了辅助己方发育和干扰对面发育的关键；再者，由于对方存在着侵占棋盘、“吞子”合并和添堵加塞的干扰，我们对于边界上可能存在的问题也应该有所重视，包括防“吞子”和在边界堆大子防止入侵的策略，这里涉及到了我们要设计函数鼓励大子向边界移动；最后，就是保留多种对策以防止时间超时的问题。

第四代算法加入了统计对方得分的功能，进一步增强算法的进攻性，同时加入己方落子防止被吃的功能，调整了估值函数。并且经过几番测试发现，估值函数中不同列的权重不能设的相差过大；第五代算法加入连招功能，以及在某些情况下禁用回撤方向，同时用剪枝的方法把搜索树加到了五层，然后加入了在对方棋盘落子来堵对方的功能。算法比之前有一些提升；第六代算法在搜索树中加入了落子的判断，然后进一步调试了估值函数中的不同列的权重这一参数。这一代是最终的版本，经过测试发现，估值函数中这一参数的调整并未给算法带来太大的提升。

我们的算法中最关键的还是估值函数，我们设定的估值函数仍然很粗略，无法对一些情况进行精细的刻画，如没有刻画出棋盘上相近等级的棋子的位置关系，没有考虑棋盘的“平滑性”，因此在这一方面还有很大的提升空间，必然可以找到更为精确的估值函数。再就是我们由于水平限制，有很多想象中很厉害的功能都没能完全如愿实现，这也算是我们小组最后代码的一个遗憾之处。

1.2 算法流程图



1.3 算法运行时间复杂度分析

设官方的接口 board.move()的时间开销近似为常数 m，board.add()的时间开销近似为常数 a，board.getNext()的时间开销近似为常数 n1，board.getNone()的时间开销近似为常数 n2，board.getScore()的时间开销近似为常数 n3，board.copy()时间复杂度为 c.

我们的算法花费时间的主要有两部分：position 模式下计算每一个可落子位置落子后的收益并找出收益最高的位置，direction 模式下用估值函数估值建立搜索树并找出合并后收益最高的方向。

m : board.move()

a : board.add()

n1 : board.getNext()

n2 : board.getNone()

n3 : board.getScore()

c : board.copy()

_sumMyScore: 时间开销应为常数，因为只是 getScore 然后进行 4*8 个位置的运算。设为 s1。

_sumOpScore：同上。设为 s2。

sumMyScore：虽然相较于_sumMyScore 多运行了 32 次 self.weight 和 getBelong，但仍为常数。设为 s3。

sumOpScore：同上。设为 s4。

addHelper：如果当前对方有 n 个空位，则 addHelper 方法运行一次的时间开销为 n1+n2+s1+s2+(n+1)*a*4*(c+m+s1+s2)。总体上来说因为 n 有上界，所以时间开销被限定在一定范围内。时间复杂度大致为 O(n)。设其运行一次时间开销为 a2。

moveHelper :（搜索三层（即合并三次，中间也有放子过程））:大部分情况下先手运行一次时间开销约为 4*{c+m+4*[c+a2+a+m+4*(c+m+s3+s4)]}+其余常数级别列表 minmax 运算开销以及最后的防撤回判断开销；后手类似。时间主要消耗在搜索上，最大情况要(4+16+64)次 copy+move,中间在第一层或第二层还要有 addHelper,先手最大为 16 次 addHelper，后手最大为 4 次。由于三层的时间开销并不很大，所以未在上面做 alpha-beta 剪枝。

搜索四层的时间主要部分开销约为 (4+16+64+256)*(c+m), 五层为 (4+16+64+256+1024)*(c+m)，由于有剪枝功能，所以平均意义下时间开销应为前面计算的一半，即分别为 0.5*(4+16+64+256)*(c+m) + 0.5*(4+16+64+256+1024)*(c+m)

easyCanMove : 4(c+m)

getBoarder : 4*board.getBelong()

同时，我们对超时的可能也做了防范，在前期适当减少了搜索的层次，因此在实际测试过程中，很少出现超时的情况，并且在出现的时候也都及时做了处理。

2 程序代码说明

2.1 数据结构说明

在我们的算法中，只有一个 Player 类，所有的函数均写在这一个类中。在“分值”的计

算和估值函数计算收益中，用的都是列表等数据结构，在确定合并方向的时候，用到了搜索树，并对所构建的更深层的搜索树进行了α–β剪枝以增加搜索树的层次。

2.2 函数说明

addHelper：用于确定落子位置的函数，算法的核心函数之一，其主要策略是最大化己方“分值”以及最小化对方“分值”。
moveHelper：用于确定合并方向的函数，算法的核心函数之一，其主要策略是用估值函数构建搜索树并通过剪枝增加树的深度。
sumMyScore：计算己方带加权的“分值”。
sumOpScore：计算对方带加权的“分值”。
weight：计算某一列上棋子的权重。
_sumMyScore：计算己方不带权重的“分值”。
_sumOpScore：计算对方不带权重的“分值”。
easyCanMove：用于判断能否进行合并，即有无合法的合并方向。
getBoarder：得到边线上棋子的归属。

2.3 程序限制

我们的整体算法比较保守，可以说在每一次革新的时候，最先考虑到的就是是否会出现可能的错误和超时问题。而在这数次的测试和最终天梯的比赛中，我们的程序也并没有出现超时或者是错误的情况，也算是符合预期了。

3 实验结果

3.1 测试数据

实验环境说明：

- 硬件配置：（CPU/内存）Intel Core i5 / 8G
- 操作系统：（名称/版本）Mac / 10.15.4
- Python 版本：（版本号）3.7.4

如下是我们小组组长牛沛然进行的线下测试之一，由于我们小组按照不同的想法写出了不同的版本，每个版本的更新迭代次数也很多，最后选取的就是其中效果比较好的那个版本，因此这里只展示出一组测试实验结果。

但实际上我们对于估值函数进行了比较多的测试：

最初先尝试了按行给出估值权重的变化，第 2，3 行权重高，想着让子向 2，3 行聚集，可能会合并更快一些，尝试后发现这样比之前的战斗力差很多，分析原因可能是这样破坏了正常的搜索。

然后尝试按列给出估值的权重，

即把这个

```
    def sumMyScore(self,board):
        sums = 0
        for n in range(8):
            if self.isFirst:
                weight = 1 if n <= 3 else 3
                for m in range(4):
                    if board.getBelong((m,n)):
                        sums += weight*4**board.getValue((m,n))
                        #可注释掉取消连招功能
                        if board.getValue((m,n)) > 5 and self.getTheSame(board,m,n):
                            sums += weight*4**board.getValue((m,n))*self.getTheSame(board,m,n)
            else:
                weight = 1 if n >= 4 else 3
                for m in range(4):
                    if not board.getBelong((m,n)):
                        sums += weight*4**board.getValue((m,n))
                        #可注释掉取消连招功能
                        if board.getValue((m,n)) > 5 and self.getTheSame(board,m,n):
                            sums += weight*4**board.getValue((m,n))*self.getTheSame(board,m,n)
        return sums
```

改成

```
    def sumMyScore(self,board):
        sums = 0
        for n in range(8):
            if self.isFirst:
                if n in [0,1]:
                    weight = 1
                elif n in [2,3]:
                    weight = 1.5
                else:
                    weight = 2.0
                for m in range(4):
                    if board.getBelong((m,n)):
                        sums += weight*3**board.getValue((m,n))
                        #可注释掉取消连招功能
                        if board.getValue((m,n)) > 5 and self.getTheSame(board,m,n):
                            sums += weight*3**board.getValue((m,n))*self.getTheSame(board,m,n)
            else:
                if n in [6,7]:
                    weight = 1
                elif n in [4,5]:
                    weight = 1.5
                else:
                    weight = 2.0
                for m in range(4):
                    if not board.getBelong((m,n)):
                        sums += weight*3**board.getValue((m,n))
                        #可注释掉取消连招功能
                        if board.getValue((m,n)) > 5 and self.getTheSame(board,m,n):
                            sums += weight*3**board.getValue((m,n))*self.getTheSame(board,m,n)
        return sums
```

在本地测试中比之前的略佳，即先后手分别测试 40 或 50 轮，胜率高于 55%
后来把 1，1.5，2.0 的组合更改为 1，1.1，1.6; 1，1.2，1.8; 1，1.3，1.9 有一定的差异
#1，1.2，1.8--vs--最初的 1,3 :28：22；31：19

但放在天梯上进行对战的时候，发现后面这个组合打官方的 S 效果还不如最初的，所以最后也没有采用。

我们也更改过指数的底 尝试过 3，2，2.5
#底为 3-vs-4:先后手均为 27：23，应该可以认为略好；
#底为 3-vs-2:先手 27：23，后手 22：28，优劣无法判定；
2-vs-4:先手 22：28，后手 31：19，优劣无法判定
(均为只改动己方底，敌方底仍为 4，并且设的估值函数是最初的和采用上述按列）
#2.5--vs--3: 28：22；30：20

但用 2.5 被 S 级打爆了，用最开始的 4 是平局；而且 2.5 对上 4 好像也差不多是平局

因而至此对于本地测试产生了极大地怀疑，并且并未在最终的天梯赛上采用这样的估值方法

之后就开始在网站上进行测试

参考了 https://www.zhihu.com/question/54217135

尝试了这样的估值函数

在 sumMyScore(self,board) 中进行了这样的改造

保守型 1：

```
if self.isFirst:
    scores = {(0,0):7,(1,0):10,(2,0):14,(3,0):16,
              (0,1):6,(1,1):9,(2,1):13,(3,1):15,
              (0,2):5,(1,2):8,(2,2):11,(3,2):12,
              (0,3):1,(1,3):2,(2,3):3,(3,3):4,
              (0,4):14,(1,4):14,(2,4):14,(3,4):14,
              (0,5):14,(1,5):14,(2,5):14,(3,5):14,
              (0,6):14,(1,6):14,(2,6):14,(3,6):14,
              (0,7):14,(1,7):14,(2,7):14,(3,7):14}
else:
    scores = {(0,0):14,(1,0):14,(2,0):14,(3,0):14,
              (0,1):14,(1,1):14,(2,1):14,(3,1):14,
              (0,2):14,(1,2):14,(2,2):14,(3,2):14,
              (0,3):14,(1,3):14,(2,3):14,(3,3):14,
              (0,4):1,(1,4):2,(2,4):3,(3,4):4,
              (0,5):5,(1,5):8,(2,5):11,(3,5):12,
              (0,6):6,(1,6):9,(2,6):13,(3,6):15,
              (0,7):7,(1,7):10,(2,7):14,(3,7):16}
```

进攻型 1：

```
if self.isFirst:
    scores = {(0,0):4,(1,0):3,(2,0):2,(3,0):1,
              (0,1):14,(1,1):11,(2,1):8,(3,1):5,
              (0,2):15,(1,2):12,(2,2):9,(3,2):6,
              (0,3):16,(1,3):13,(2,3):10,(3,3):7,
```

```
                (0,4):14,(1,4):14,(2,4):14,(3,4):14,
                (0,5):14,(1,5):14,(2,5):14,(3,5):14,
                (0,6):14,(1,6):14,(2,6):14,(3,6):14,
                (0,7):14,(1,7):14,(2,7):14,(3,7):14}
    else:
        scores = {(0,0):14,(1,0):14,(2,0):14,(3,0):14,
                  (0,1):14,(1,1):14,(2,1):14,(3,1):14,
                  (0,2):14,(1,2):14,(2,2):14,(3,2):14,
                  (0,3):14,(1,3):14,(2,3):14,(3,3):14,
                  (0,4):16,(1,4):13,(2,4):10,(3,4):7,
                  (0,5):15,(1,5):12,(2,5):9,(3,5):6,
                  (0,6):14,(1,6):11,(2,6):8,(3,6):5,
                  (0,7):4,(1,7):3,(2,7):2,(3,7):1}
```

保守型 2：

```
    if self.isFirst:
        scores = {(0,0):7,(1,0):10,(2,0):13,(3,0):16,
                  (0,1):6,(1,1):9,(2,1):12,(3,1):15,
                  (0,2):5,(1,2):8,(2,2):11,(3,2):14,
                  (0,3):1,(1,3):2,(2,3):3,(3,3):4,
                  (0,4):14,(1,4):14,(2,4):14,(3,4):14,
                  (0,5):14,(1,5):14,(2,5):14,(3,5):14,
                  (0,6):14,(1,6):14,(2,6):14,(3,6):14,
                   (0,7):14,(1,7):14,(2,7):14,(3,7):14}
    else:
        scores = {(0,0):14,(1,0):14,(2,0):14,(3,0):14,
                  (0,1):14,(1,1):14,(2,1):14,(3,1):14,
                  (0,2):14,(1,2):14,(2,2):14,(3,2):14,
                  (0,3):14,(1,3):14,(2,3):14,(3,3):14,
                  (0,4):1,(1,4):2,(2,4):3,(3,4):4,
                  (0,5):5,(1,5):8,(2,5):11,(3,5):14,
                  (0,6):6,(1,6):9,(2,6):12,(3,6):15,
                  (0,7):7,(1,7):10,(2,7):13,(3,7):16}
```

保守型 3：

```
    if self.isFirst:
        scores = {(0,0):13,(1,0):14,(2,0):15,(3,0):16,
                  (0,1):12,(1,1):11,(2,1):10,(3,1):9,
                  (0,2):5,(1,2):6,(2,2):7,(3,2):8,
                  (0,3):4,(1,3):3,(2,3):2,(3,3):1,
                  (0,4):14,(1,4):14,(2,4):14,(3,4):14,
                  (0,5):14,(1,5):14,(2,5):14,(3,5):14,
                  (0,6):14,(1,6):14,(2,6):14,(3,6):14,
                  (0,7):14,(1,7):14,(2,7):14,(3,7):14}
```

```
    else:
        scores = {(0,0):14,(1,0):14,(2,0):14,(3,0):14,
                  (0,1):14,(1,1):14,(2,1):14,(3,1):14,
                  (0,2):14,(1,2):14,(2,2):14,(3,2):14,
                  (0,3):14,(1,3):14,(2,3):14,(3,3):14,
                  (0,4):4,(1,4):3,(2,4):2,(3,4):1,
                  (0,5):5,(1,5):6,(2,5):7,(3,5):8,
                  (0,6):12,(1,6):11,(2,6):10,(3,6):9,
                  (0,7):13,(1,7):14,(2,7):15,(3,7):16}
```

参考网站上共有三种估值函数，每种估值函数都改造成了进攻型和保守型两种，这几种之间差别不大

也尝试改变对方区域的权重，发现太小不行，太大也不行，总体来说 12-16 没有太大区别(有时候有区别，但不太稳定，所以对于测试这种事情就不太相信)，选择 14。

测试过程中发现如果全部回合都采用进攻估值函数，因为大子在边线的一个角落，合并的确很高效，但有时候大子会被吃掉，因为估值最高的一角权重确实很大，导致不会提前几个回合规避被吃的风险。而且全回合进攻和最初没有改变估值函数的进行对打也只是 55 开之后改成了前一定回合用进攻估值函数，也没有明显区别。之后就开始想着用保守态势，全回合保守估值函数肯定不行，因为前期工作经验告诉我们进攻收益更大。因而想着前一定回合用保守估值函数来保证前期合并效率，之后仍采用原来的估值函数。最后经过几番调试也只不过能和最开始的 55 开。就没有在最终的天梯赛中放上去。(就是这样，我们最后测试的所有东西基本都没有明显的效果，也是感觉非常遗憾)

#其中一种大致就是这个样子：

```
    def sumMyScore(self,board,currentRound):

        if currentRound < 70:
            sums = 0
            if self.isFirst:
                scores = {(0,0):13,(1,0):14,(2,0):15,(3,0):16,
                          (0,1):12,(1,1):11,(2,1):10,(3,1):9,
                          (0,2):5,(1,2):6,(2,2):7,(3,2):8,
                          (0,3):4,(1,3):3,(2,3):2,(3,3):1,
                          (0,4):14,(1,4):14,(2,4):14,(3,4):14,
                          (0,5):14,(1,5):14,(2,5):14,(3,5):14,
                          (0,6):14,(1,6):14,(2,6):14,(3,6):14,
                          (0,7):14,(1,7):14,(2,7):14,(3,7):14}
            else:
                scores = {(0,0):14,(1,0):14,(2,0):14,(3,0):14,
                          (0,1):14,(1,1):14,(2,1):14,(3,1):14,
                          (0,2):14,(1,2):14,(2,2):14,(3,2):14,
                          (0,3):14,(1,3):14,(2,3):14,(3,3):14,
                          (0,4):4,(1,4):3,(2,4):2,(3,4):1,
                          (0,5):5,(1,5):6,(2,5):7,(3,5):8,
```

```
                        (0,6):12,(1,6):11,(2,6):10,(3,6):9,
                        (0,7):13,(1,7):14,(2,7):15,(3,7):16}
            for n in range(8):
                for m in range(4):
                    weight = scores[(m,n)]
                    if board.getBelong((m,n)) == self.isFirst:
                        a = weight*3**board.getValue((m,n))/10
                        sums += a
            return sums
        else:
            sums = 0
            for n in range(8):
                weight = self.weight(n,self.isFirst)
                for m in range(4):
                    if board.getBelong((m,n)) == self.isFirst:
                        a = weight*3**board.getValue((m,n))
                        sums += a
            return sums

    def sumOpScore(self,board):
        sums = 0
        for n in range(8):
            weight = self.weight(n,not self.isFirst)
            for m in range(4):
                if board.getBelong((m,n)) != self.isFirst:
                    a = weight*3**board.getValue((m,n))/10
                    sums += a
        return sums

    def weight(self,n,p):
        if p:
            if n < 4:
                return 8
            elif 3 < n < 6:
                return 12
            else:
                return 14
        else:
            if n > 3:
                return 8
            elif 1 < n < 4:
                return 12
```

```
            else:
                return 14

    #算分
    def _sumMyScore(self,board):
        lst = board.getScore(self.isFirst)
        sums = 0
        for rank in lst:
            sums += 4**rank/10
        return sums

    def _sumOpScore(self,board):
        lst = board.getScore(not self.isFirst)
        sums = 0
        for rank in lst:
            sums += 4**rank/10
        return sums
```

其实调试估值函数的过程中伴随着对于搜索树的更新，最初的搜索树没有放子，因而觉得可能这样会影响搜索，之后就在搜索树里加入放子，起到了立竿见影的效果。但因为放子都是采用的定义的放子函数，时间复杂度较高，只在 3 层搜索中使用(即搜索三次合并)，之后就商量能不能在四层五层中加入放子，但即使在四层五层的搜索树中采用最开始随机 ai 的放子也没有明显的效果。在最终的天梯赛中也就只采用三层搜索。

3.2 结果分析

总体来说，结合在天梯上的表现，我们在不断地测试和对比中发现，我们小组所采取进攻策略有一定的效果。这是在算法开发第一周结束的时候根据 RBQ 的进攻策略而思考改进出来的。算法开发前期大家地算法进本都还不够复杂，一定情况下如果单纯进攻会把用搜索树来回移动的人堵死，那么之后就会进入我方单方面屠杀的境地；但随着大家地算法都越来越复杂，有时候进攻太猛烈只会葬送掉自己，而且我们一直没有找到可以迅速占领对方地盘并且让占领的子迅速增值来避免被合并的方法，因而完全的进攻并不是我们能选择中的最好的。

最开始有考虑过直接在对方前线放子来让我方进攻时大子不会有太大风险，但这样可能太过复杂而且局势多变可能也不会有很好的效果，而且本身单纯搜索就能够在关键时候在对方地界放子来降低我放大子被吃的风险。

也考虑了阻止对方大子合并的功能，有时候会有一定效果。

但缺陷也很多，在快速堆子发育方面的功能有时候则显得不尽如人意。有的时候我们的策略并不能在优势局面下扩大自己的优势，而在劣势局面下也并不能为自己取得翻盘的主动权，这与我们单一的搜索树的思路是有关的。在鼓励进攻这一方面，实际效果有时候也不是很明显，这说明我们在估值函数方面还做的不够好。

总体运行时间还算是尽如人意的，我们在其中加入了关于是否会超时的判断，因此我们的算法是不容易超时的。

开销主体在于我们的 addHelper，moveHelper 中的决策搜索树上，尤其是如果采用 5 层的搜索会很耗时间，最多只能运行 50 轮就要进行策略的更换。

3.3 经典战局

这一局的对战充分体现了我们算法的策略思想，从一开始就积极进攻，夺取对方地盘。下面是比赛进行到第 67 轮时的盘面，左边先手方是 Romeo，右边是我方：

虽然从图中看到这时我方的合并速度落后于对方，对方已经合并到 512 了，我们这边最大的才是 128，但好在我方已经将对方边界处的三个位置占领，从后面的比赛过程中将会看到，这一优势在后期将会发挥巨大的作用。比赛继续进行，下面是比赛进行到第 227 轮的盘面：

可以发现到这一轮已经将对方塞爆了，对方边界上的四个位置全部被我方占领，使得对方在一段时间内无法进行合并，而在这段时间里面我方一直在不断进行合并，逐渐超越对方，占据优势。对方最大的棋子仍然停留在 512，我方已经合并出 1024 了。并且从盘面信息可以看出我方棋子的位置整体比较规整，基本符合“平滑性”的要求，这也使得我方的合并速度能一直保持在一个较快的水平，同时不会自己把自己塞爆。然后下面几张图所展示比赛过程是这局比赛的一个亮点：

在第 228 轮时我方在边界处看似是出现了一个缺口，使对方乘虚而入，实则是我方布下的一个陷阱。因为在第 228 轮时对方可以进行合并了，并且只有向右这一个合法方向，于是对方只能向右进行合并，吃掉我方的 512，然而接下来对方合成的 1024 就被紧挨着的我方的 1024 吃掉，正所谓“欲擒故纵”，不可谓不妙。实际上在最终比赛直播的时候我们也在比赛中看到过这一招（具体是哪两位之间的比赛记不清了）。接下来的比赛过程我方一直占据

着对方边界上的三个位置，保持着绝对的优势，并且在第 288 轮时再次将对方塞爆，最终以绝对优势取得胜利：

4 实习过程总结

4.1 分工与合作

我们小组的分工是牛沛然、朱新宇、张植竣负责代码的编写和测试工作，赵轩和刘新龙负责编写实验报告，同时组内的每个人都思考并提出策略。小组自组建以来，一共进行了三次组会，均是在腾讯会议上进行的。

第一次组会，是在任务下发后不久，会上组长牛沛然和朱新宇分别介绍了自己的初代算法，提出了各自的一些想法。然后大家一起讨论了对战的策略，说出了各自对二〇四八对战游戏的理解和想法。

第二次组会，组长牛沛然和朱新宇介绍了各自经过改进后的算法，经过测试排除了一些 bug，并对每个人的策略思考和想法进行了一次汇总。在会议的最后，由张植竣同学共享屏幕，在天梯平台上一起观看了和其他组算法的对战，思考策略，期待从中受到启发。

第三次组会，是在天梯赛即将结束的时候，组长牛沛然和朱新宇先介绍了各自对代码的改动和测试，主要是调整并测试了估值函数中的一些参数，但发现参数调整后并没有带来很

大的提升。然后赵轩和刘新龙介绍了写实验报告的情况并一起做了一些讨论，会议最后仍然由张植竣同学共享屏幕然后观看对战的情况，从中继续思考策略。

下图是我们小组在 Github 上的合作平台截图，大家在经过意见商讨后会把代码在这里

进行统一更新。

这是我们小组在微信群里的讨论截图，有的时候会一直讨论到半夜

4.2 经验与教训

经验：在尝试解决问题的过程中组员的编程水平和思考问题的能力都得到了训练，并且通过合作交流锻炼了团队协作的能力。以及我们很多的策略都是通过对战从对手的对战策略的学到的，说明观看对战过程并思考策略是很有效果的，要学会多从对手的身上学习对自己有用的东西。

教训：前期的时候分工不是很明确，没有很具体的规划，所以在效率上有些不足。以及在实习过程中，组员之间的交流不是很充分，组会的次数不是很多。还有一点是前期由两个同学分别写一套代码，导致了一些重复的工作；并且其中有大量的代码属于没有用的，造成了很大的时间损失。

4.3 建议与设想

建议：（1）建议在最后比赛的时候可以在双方比分接近（比如得分差在两分以内）的时候多加几轮比赛，因为在比赛直播的时候看到一些算法其实是处于势均力敌的状态，输掉的一方往往以一分之差输掉比赛，非常可惜，所以可以适当增加几轮比赛，这样既可以保证公平性也能让比赛更激烈、更具观赏性。

（2）大作业的提交时间安排在期末季，这和一些课（比如一些通选课）的考试时间有冲突，建议可以适当提前一点，这样可以让期末 ddl 较多的组员投入更多的精力，也能减轻他们期末复习的压力。

寄语：数算课是一门比较累也很有趣的课，平时作业、大作业活动都很多，但只要投入时间，你们会发现能在一学期的时间里学到很多东西，希望你们能不留遗憾，收获很多。

设想：可以在大作业结束后将助教评选出的优秀作业的代码和实验报告放到课程网站上，供后续还有意改进自己算法的同学参考，也可以供明年选课的学弟学妹们参考。

5 致谢

感谢陈斌老师这一学期以来对我们的引导和支持。
感谢各位助教和技术组的同学构想生动有趣的大作业竞赛、不辞辛劳地搭建对战平台。
感谢我们每一个组员的在实习过程中的付出。
感谢天梯上的各位大佬，激励我们不断思考、改进算法。

6 参考文献

Github 代码库 https://github.com/pkulab409/sessdsa.2048
天梯对战平台 http://gis4g.pku.edu.cn/ai_arena/game/4/（参考的是天梯上大佬们的打法）
知乎的估值函数贴 https://www.zhihu.com/question/54217135

数据结构与算法课程实习作业报告

陈克发 胡苏麟 梁天宇 张桐铱 蒋志超*

摘要：在 2048 博弈版中，通过深度搜索，alpha-beta 剪枝，优先进攻策略设计的 AI，在最终天梯赛中第九名出线。

关键字：深度搜索，剪枝，2048

1 算法思想 2
1.1 总体思路 2
1.2 算法流程图 2
1.3 算法运行时间复杂度分析 4
2 程序代码说明 5
2.1 数据结构说明 5
2.2 函数说明 6
2.3 程序限制 10
3 实验结果 11
3.1 测试数据 11
3.2 结果分析 12
4 实习过程总结 12
4.1 分工与合作 12
4.2 经验与教训 13
4.3 建议与设想 13
5 致谢 13
6 参考文献 13

1 算法思想

1.1 总体思路

二〇四八是一款博弈游戏，游戏中玩家依次执行下棋和移动两种行为，游戏的最终胜利

条件是在双方都无法移动前或者在一定回合内合成比对方更大的数。为取得胜利，玩家会在不妨碍移动的情况下尽可能地合成较大数，或者使用妨碍对方的合成的手段，最好结果是使得对方无法移动。我们认为，游戏中的双方都会采取使局面变得对自己有利的行动。

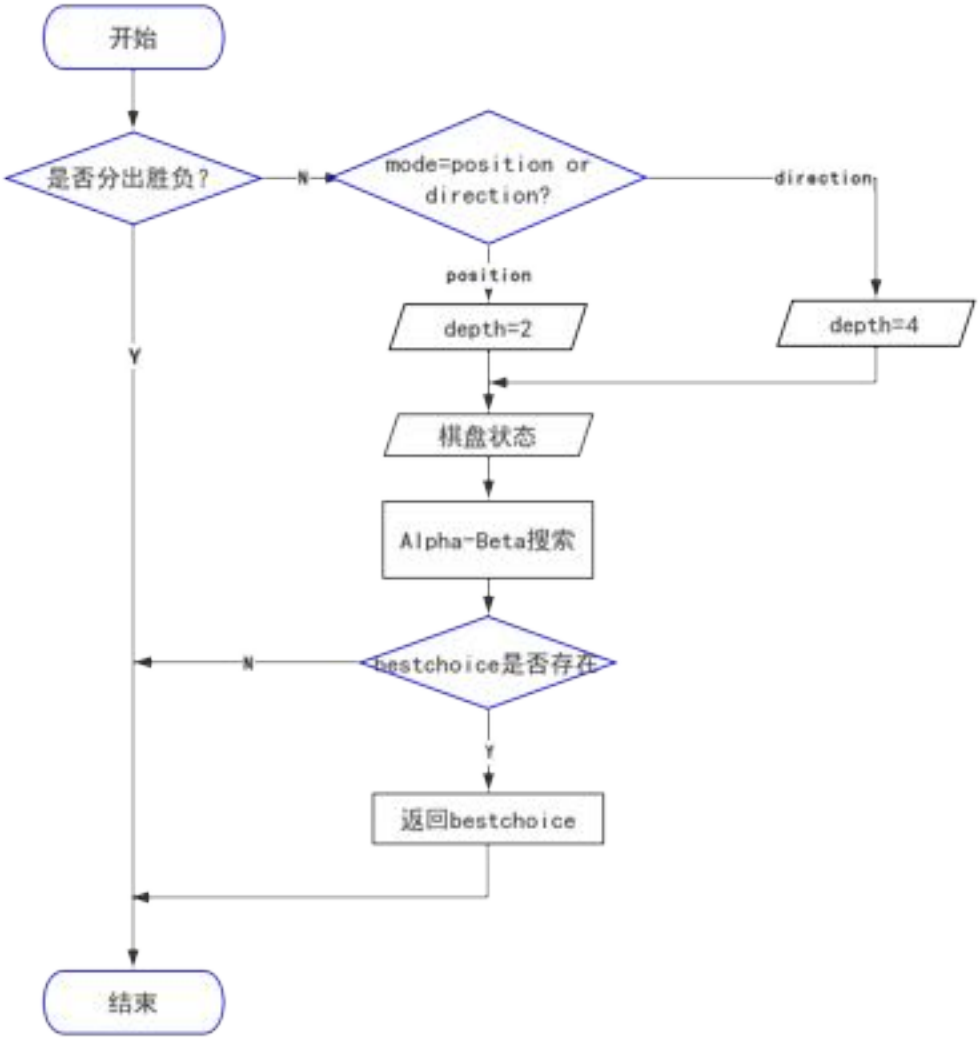
我们的最初设想是：首先让 AI 在行动前对模拟后的棋盘状态进行评分估值，有利于我方的、不利于对方的局面分数高，不利于我方，利于对方的则分数低。完成估值后，AI 会根据分数比较采取不同行动后的利弊，选择出最佳方案。考虑到这一策略实际上依赖于对静态棋盘的评价函数，但以回合数和棋盘状况作为参数的静态评分函数实现起来较为困难且没有明确的修正方向，并不理想，因此，我们保留了简单的代价函数而决定改用 DFS 搜索。

在代码的正式编写中，我们查阅了资料，最终采用了 Alpha-Beta 算法。Alpha-Beta 算法是在 minmax 算法的基础上进行剪枝的算法。一般来说，我方会采取最利于我方的行动，而对手则会想办法让我方进入最坏结局。minmax 算法是一种找出失败的最大可能性中的最小值的算法，简单来说就是在对手的选择让我方处在不利的前提下选出最优解的算法。其具体实现是以当前情形作为根节点，将所有情形列举出来构成一棵树，根据底层节点的分数推演出对我方最有利的一种情形。因为穷举会耗费大量资源，造成不必要的浪费，甚至引起超时。所以我们使用 Alpha-Beta 算法减去不必要的分支：让 AI 为在树中搜索到的每个节点都设置 alpha 和 beta 值，其中 alpha 表示最好选择的下限，beta 表示最差选择的上限，当某节点的 alpha 超过或等于 beta 时，说明从该子节点后的结果都低于已知的最优解，就可以将这一节点的分支剪去。最终用于比赛的代码采用递归调用的方式建立深度优先搜索树并进行剪枝。因为下棋的情况要比移动的更为复杂，经过 一定的测试后，我们最终将下棋的搜索深度设为 2，而将移动的搜索深度设为 4。

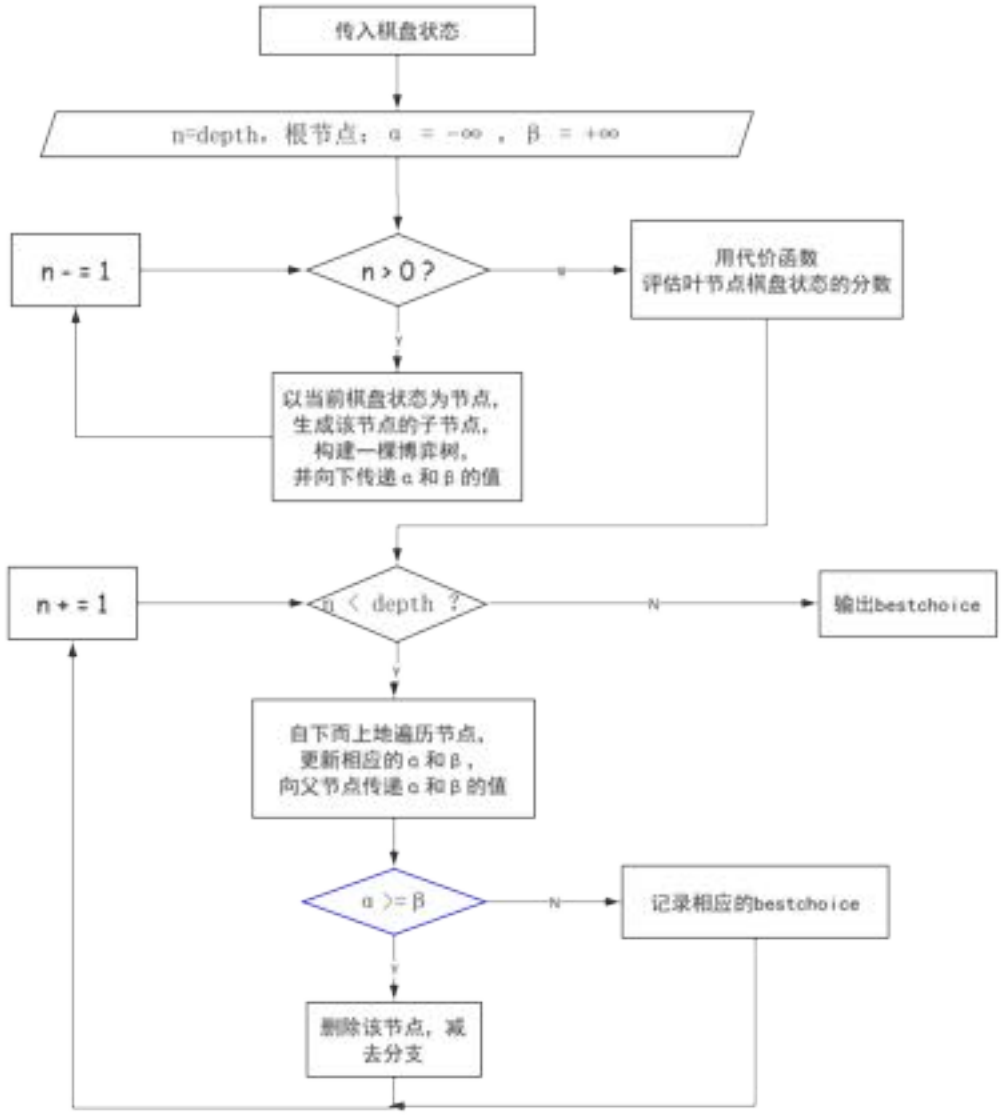
对棋盘状态分数的计算，我们将其分成棋子分，占领分以及形势分三种。棋子分是与棋子本身的分值相关，对不同的棋子会有不同的估值，大致来说，我方的棋子为正分，而对手的为负。计算占领分的基本思路为：若我方棋子占领了对手的棋盘，则加分；若对手的棋子占领了我方棋盘，则减分。我们通过调高对方棋子的占领分来鼓励 AI 合并我方棋盘上的对手棋子。而形势分则是考虑了对方吃我方子的可能以及我方吃对方子的必要性。分析我方的棋盘，若不存在吃子返回零分，若对方有吃子可能或我方可吃子可能但对方有靠山则返回负分，若我方可吃子可能且对方无靠山则返回正分，分析对方的棋盘则正好相反。实际运行中，AI 依次遍历我方的棋盘和对方的棋盘，一个个格子地完成相应的判断与计算。

1.2 算法流程图

总图：



Alpha-Beta 搜索图：



1.3 算法运行时间复杂度分析

我们的算法主要由经过 Alpha-Beta 剪枝的深度优先搜索树和代价函数两部分构成。其中代价函数的运行时间以参数的计算为主，并不存在数据的嵌套。理想情况下 Alpha-Beta 算法的复杂度为 O(nlogn)，但 Alpha-Beta 剪枝严重依赖于寻找顺序，若总是先去搜索最坏的情形，Beta 剪枝就不会发生，最坏情况下，算法的时间复杂度可达到 O(n^2)级别。综上，估计算法的时间复杂度约为 O(nlogn)。

2 程序代码说明

2.1 数据结构说明

代码中定义了两个大类，分别是 Vertex 类代表棋盘状况：

```
class Vertex:
    '''将每个棋盘状况做成节点，包括当前轮到谁的信息等等。'''

    def __init__(self, board, belong, player, currentRound=None, mode=None,
parent=None, action=None):
        self.board = board  # 棋盘
        self.belong = belong  # 当前棋局（即将）行动方
        self.player = player  # 玩家自己（即 Player 类）
        self.round = currentRound  # 当前轮数
        self.mode = mode  # 当前模式
        self.parent = parent  # 父节点
        self.child = {}  # 子节点的字典：{做出的行动:子节点}
        self.action = action  # 用于存放到达该节点的行动
        self.goal = 0  # goal,aj_value 用来保存父节点落子在对面的原因，目前是为
后手方准备的属性
        self.aj_value = 0
        self.alpha = float('-inf')  # 下界
        self.beta = float('inf')  # 上界
        self.boardValue = None  # 静态估值函数，先用这个凑合一下，要更改函数只
需要改此一处。
        self.searchValue = None  # 搜索后估值
        self.bestchoice = None  # （行动方）最佳策略
        self.winner = None  # 胜利方
```

以及 player 类包含所有估值函数以及对棋盘操作的输出：

```
class Player:
    def __init__(self, isFirst: bool, array: list) -> None:
        self.belong = isFirst
        self.array = array
        self.height = 4
        self.width = 8
        self.half_width = 4
        self.depth_search = False  # 保存先手阶段的搜索深度信息
        self.basic_list = [int(pow(2, n)) for n in range(11)]
        self.value_list = [n * self.basic_list[n] for n in range(11)]  # 预先维
护估分表
```

```
        self.basic_list.append(int(pow(2, 20)))  # 两个会用到的分值
        self.basic_list.append(int(pow(2, 21)))
```

在 Vertex 类中主要进行的工作是构建了棋盘的 max-min 树，这里我们小组经过测试，决定对先后手下棋分别构建两个不同的 max-min 树，在针对先手模式的 expand 函数,选择在对方的落子位置时没有像反手那样详细讨论，因为先手模式己方率先移动，有主动权，同时也省下一些时间，便于在特殊情况深度搜索到 3。然后在不同情况下的 max-min 树下进行相应的 alpha-beta 剪枝。

在 max-min 树中我们创建了一个字典存储子节点，以便深度搜索时快速查找调用。在剪枝的时候顺次调用。同时我们对不同的搜索深度相应写了 7 个不同的深度搜索函数，对搜索剪枝方法进行了完善。

在 Player 类里我们用到的技巧是维护了一个估分表，因为估值函数均为 2 的指数，所以直接调用表中求得的数据会缩短运行时间。

2.2 函数说明

一、 Vertex 类中的函数以及接口：

Vertex 的初始化内容在上一节的注释中已经说明。

1、**getWinner**(self)：

这个函数是判断胜负，平局也算获胜，简单调用盘面分值然后返回一个布尔值。

2、**getNone**(self):

返回场上对手方所有的空位,这里运用了一个小技巧是优先考虑返回中间的格子（见下方例子），因为这样做的概率更高。

代码细节： `return [(i, j) for j in [4, 7, 5, 6] for i in range(4) if not self.board.getValue((i, j))]`

3、**setRange**(self, alpha, beta)：

设置节点的 alpha 和 beta 值

4、**tru_expand**(self):

这是先手方创建 max-min 树的函数，将模式分为 position 和 direction 创建树，每个子节点都是调用了价值函数后得到的结果

5、**fal_expand**(self):

这是后手方创建 max-min 树的函数，与先手方类似，这里比较详细的讨论了在对方棋盘下棋子的情形，同时对棋子的威胁进行了讨论，不同之处如下：

先手方：（对应存在己方棋子受到威胁,取受威胁棋子分最大的落子点）

```
if all_threaten_chess:
    k = max(all_threaten_chess, key=lambda y: y[0])
    bestPos = k[1]
```

后手方（如果后手落子到对面而保护自己棋子可以加分）：

```
vertex.goal = 3   # 表明落子在对面的原因是为了保护己方的值为 aj_value 的棋子
```

6、**fal_dir_depth_0**(self):

这是后手移动阶段搜索深度为 0 的特殊情况讨论，这里存在的细节是讨论如果可以先吃对面棋子则吃棋子优先（保证“安全”的前提下）。返回的布尔值来确定剪枝的情况

代码细节：`if prospects[1] > self.player.startValue`

7、**tru_dir_depth_0**(self):

这是先手移动阶段搜索深度为 0 的讨论，返回布尔值

8、**fal_pos_depth_0**(self):

后手落子阶段搜索深度为 0 的函数，这里与前面 vertex.goal 对应的出现了一个讨论，即加重了此类情况的权重。，返回布尔值

代码细节：`if self.parent.goal == 3:`

9、**tru_pos_depth_0**(self):

先手落子搜索深度为 0 的函数。

10、**tru_search_1**(self, bestPos):

先手搜索深度为 3 的函数，仅仅为了方便而加。

11、**tru_search_3**(self, depth):

先手搜索阶段一般的深度搜索，运用了剪枝。

12、**fal_search**(self, depth):

后手搜索阶段一般的深度搜索，与先手类似。

二、 Player 类的函数及接口：

1、 **judge_threaten**(self, pos, now_chess):

这是判断先手落子深搜是否应该为 3 的函数，返回布尔值，先手模式倾向将子落在对方位置(因为估值函数中设置了一个很大的威胁分)，但实际上存在落子在对面后对面吃掉而不划算的情况(因为搜索深度为 2 的话只能搜到该自己移动的局面，到自己移动时，AI 可能发现吃掉自己刚刚落在对面的 2 不划算，因为有可能移过去后自己的棋会被对面吃),这样就白白地送分了

代码细节（判断对手的 2 是否能被吃）：

```
if pos[1] == 4:
    nex_pos = (pos[0], pos[1] - 1)
    while nex_pos[1] >= 0 and now_chess[nex_pos[0]][nex_pos[1]] == (0,
self.belong):
        nex_pos = (nex_pos[0], nex_pos[1] - 1)
    if nex_pos[1] < 0 or now_chess[nex_pos[0]][nex_pos[1]][1] !=
self.belong:
        return False
    else:
```

```
            if now_chess[nex_pos[0]][nex_pos[1]][0] == 1:
                return True
            else:
                return False
    else:
        return False
```

2、 **output**(self, currentRound, board, mode):

这是赛制要求的输出函数，其中整合了对深搜不同层数的讨论并给出最后的结果，一般情况下设置的为移动阶段深度为 4，落子阶段深度为 2 以保证不会超时

```
    if mode == "position" or mode == "direction":
            depth = 4 if mode == "direction" else 2
```

3、 **get_chess_score**(self, board):

返回整数，代表双方棋盘得分之差

4、 **new_cost_1**(self, now_chess, position: tuple, belong: bool):

这是下子时的代价函数，返回值为(score,value,pos)，这里搜索了上下左右的棋子并代入函数求出返回值

```
    up = self.get_neighbor_2(now_chess, position, 0)
    down = self.get_neighbor_2(now_chess, position, 1)
    left = self.get_neighbor_2(now_chess, position, 2)
    right = self.get_neighbor_2(now_chess, position, 3)
    return self.new_put_chess(up, down, left, right)
```

5、 **new_cost_function**(self, chessboard, dep=**False**):

考虑更加全面的落子代价函数，使得效率更高，返回的两个值一个代表估值函数计算的分数，另一个代表两个棋盘的分数差（为了计算吃掉棋子是否真正产生了收益）里面新增了一个参数是为了讨论当深搜深度为 3 时的情况。主要是对整个棋盘遍历，并依照情况计算分数。

典型的例子有：

```
    if now_chess[position[0]][position[1]][1] == self.belong:
            score[0] += 2 * b if self.belong else b
            score[0] += b * a  # 自己棋子分
            another_score += a
```

这是对方棋盘被本方占领的时候计算的代价函数

6、**new_situation_score**(self, score, position, direction, my_position_dict, now_chess, neighbor_value, has_add, dep=**False**)

这是对盘面分析的代价函数，主要考虑的因素有相同棋子合并的可能，被对方棋子吃掉的风险，吃掉对方棋子的可能，已经防止对方大子合并的可能。

典型的例子有：

```
    if now_chess[next_position[0]][next_position[1]][0] + 1 in lst:
            x = now_chess[position[0]][position[1]][0]
```

```
            score[0] -= x + 2
            return
```

这里是本方如果吃对面，对面却可以反吃自己的情况，返回的代价是负值，同时也考虑到存在自己无法移动而可以通过落子堵住对面

```
    if now_chess.count((0, self.belong)) <= 1:
        if has_add[next_position] and has_add[position]:
                has_add[next_position] = has_add[position] = False
                score[0] +=
                            self.basic_list[now_chess[next_position[0]][next_
                            position[1]][0] + 1] - 1
                return
```

这里是如果空位很少的情况下，则加大相同相邻棋子的权重，以防止棋盘被卡满的情况

这个函数是代码的核心函数之一，由于有很多的讨论以及算法，这里只能给出两个典型的例子，具体的全部代码可以查看源代码以及相应的注释。

7、**get_neighbor_2**(self, now_chess, position: tuple, direction: int):

这是落子方是 not self.belong,分析将棋子落在 self.belong 时的局势。返回一个布尔值。如果探测到的该棋子属于落子方,返回 True，否则返回 False。

8、**get_neighbor_3**(self, now_chess, position: tuple, direction: int):

这是与上一函数先后手交换的镜像函数，其余类似。

9、**get_neighbor_value**(self, score, pos, direction, my_position_dict, now_chess, neibor_value, has_add):

返回一个整数，为指定方向上相邻棋子的代价函数值。

10、**judge_is_safe**(self, pos, direction, now_chess, neibor_value):

Pos 是对面棋子的一个位置，函数返回一个整数，用来判断对方是否存在陷阱（诱导我方吃对方棋子然后被吃）

11、**mychess_is_safe**(self, pos, belong, now_chess):

这是为 fal_expand 函数写的判断棋子是否有靠山的辅助函数，返回一个 bool 值。

12、**return_pos**(self, belong):

返回当前方遍历的所有格子，返回值为 x 与 y 坐标。

13、**is_in_chess**(self, position):

返回布尔值，判断该位置是否在棋盘中。

14、**is_mine**(self, position):

返回布尔值，判断该棋子所属方。如果是本方返回 True。

15、**my_chessman_score**(self, value):

返回整数，对应己方每个值对应多少得分，封装在函数中便于修改，在移动模式的代价函数中调用。

16、rival_chessman_score(self, value: int):
返回对方每个值对应多少得分，封装在函数中便于修改，在移动模式的代价函数中调用。这里相比己方进行了权重下调，使得 ai 更倾向本方代价变大。
代码：return -self.value_list[value] // 2

17、be_occupied_score(self, value: int):
在己方棋盘被对方占据的情况返回的分数，返回的是一个负分，鼓励 AI 吃掉对方在自己位置的棋子封装在函数中便于修改，在移动模式的代价函数中调用。

18、occupy_score(value: int):
在对方棋盘被己方占据的情况返回的分数，返回整数 2，降低 AI 将棋子移回本方的可能性。

19、threaten_score(self, value: int):
返回一个整数，对应的是棋子存在吃对方和被对方吃的可能性时返回的分数，封装在函数中便于修改，在 get_neighbor 函数中调用

20、new_put_chess(self, value_1, value_2, value_3, value_4):
这里 value 参数:(value,bool,pos) or False，返回值为(score,value,pos)。函数主要是返回当棋子下在对面阻断合并，或者防止对方吃己方等情况的代价函数，主要是防守的策略
典型代码：

```
if value_1 and value_2:
        if (value_1[1] != value_2[1]) and value_1[0] == value_2[0]:
            return (float("inf"), value_1[0], value_1[2]) if value_1[1] else
                   (float("inf"), value_2[0], value_2[2])
```

这是落子在分值相同的己方棋和对方棋之间，返回棋子的值，属于落子方的棋子的位置

21、update_position(position, direction):
更新位置的函数，将棋子向指定方向移动一格，返回移动后棋子的坐标。

22、get_direction(direction):
new_situation_score 的辅助函数，返回整数值，对方向顺序进行交换

2.3 程序限制

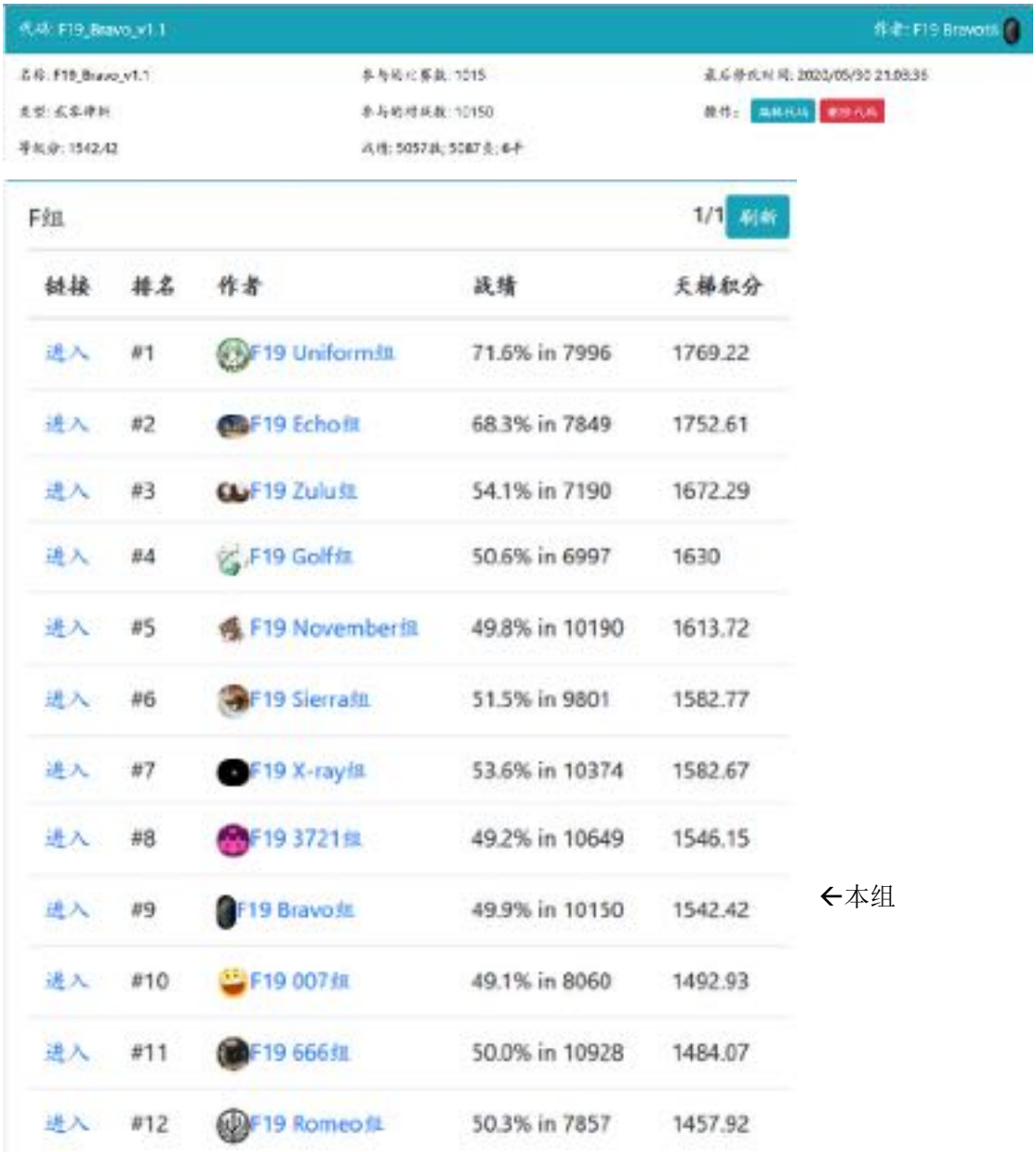
目前最终上传的代码在测试中并没有发现报错，整体稳定性很高

3 实验结果

3.1 测试数据

实验环境说明：

- 硬件配置： Intel® Core™ i5-8265U/16G
- 操作系统：windows 10
- Python 版本：3.7



F组 1/1 刷新

链接	排名	作者	战绩	天梯积分	
进入	#1	F19 Uniform组	71.6% in 7996	1769.22	
进入	#2	F19 Echo组	68.3% in 7849	1752.61	
进入	#3	F19 Zulu组	54.1% in 7190	1672.29	
进入	#4	F19 Golf组	50.6% in 6997	1630	
进入	#5	F19 November组	49.8% in 10190	1613.72	
进入	#6	F19 Sierra组	51.5% in 9801	1582.77	
进入	#7	F19 X-ray组	53.6% in 10374	1582.67	
进入	#8	F19 3721组	49.2% in 10649	1546.15	
进入	#9	F19 Bravo组	49.9% in 10150	1542.42	←本组
进入	#10	F19 007组	49.1% in 8060	1492.93	
进入	#11	F19 666组	50.0% in 10928	1484.07	
进入	#12	F19 Romeo组	50.3% in 7857	1457.92	

3.2 结果分析

我们组的进攻策略偏向于进攻，这使得在防御（防止被塞爆）方面做的很差，虽然最终对策上有所修改但是敌不过以塞爆为制胜手段的其他选手，未能进入八强。这里显然是初始思路出了问题。程序的运行时间是最大优化的，用满了所给的所有时间，计算效率还算不错，主要花费在深度搜索中。

4 实习过程总结

4.1 分工与合作

陈克发和胡苏麟为开发组（主程序），蒋志超，梁天宇和张桐铱为技术组（主测试和报告）

微信群和腾讯会议

4.2 经验与教训

时间分配要合理，这次就是没时间去完善防御部分的代码了。

4.3 建议与设想

建议最终的代码比赛可以动态实时可视化。虽然这门课大概率是要被取代了，但由衷感慨，这课很有趣但太‘卷’了。程序对于今后所有的工作都有巨大的帮助

5 致谢

感谢所有组员的辛勤劳动，着重感谢陈克发同学的贡献。

6 参考文献

http://blog.codinglabs.org/articles/2048-ai-analysis.html

数据结构与算法课程实习作业报告

陈湘、李子涵*、默文婷、孙久雯、文雨涵

摘要：本组算法将启发式规则与决策树相结合：落子部分运用一定启发式规则，编写函数决策落子；合并部分通过 minmax 算法模拟双方操作，进行深度搜索，并通过 alpha-beta 算法增加搜索深度。代码中主要涉及树的结构与算法、递归算法、minmax 算法、alpha-beta 算法等。最终本组在天梯赛中胜率为 49.4%，对战技术组 bot_D、bot_C 能取得压倒性的胜利。

关键字：决策树；递归算法；minmax 算法；alpha-beta 算法；

1 算法思想.....1
1.1 总体思路.....1
1.2 算法流程图.....2
1.3 算法运行时间复杂度分析.....3
2 程序代码说明.....3
2.1 数据结构说明.....3
2.2 函数说明.....3
2.3 程序限制.....9
3 实验结果.....9
3.1 测试数据.....9
3.2 结果分析.....11
3.3 经典战局.....12
4 实习过程总结.....13
4.1 分工与合作.....13
4.2 经验与教训.....14
4.3 建议与设想.....15
5 致谢.....13
6 参考文献.....15

1 算法思想

1.1 总体思路

此次实习作业“贰零肆捌”是从经典单机游戏 2048 演变进化设计而成的对抗类游戏。这就自然地为我们的思考提供了两个维度，一是如何完成 2048 基础游戏，得到尽可能好的成绩；二是如何对游戏中的对抗性进行处理，是顺其自然还是有意为之。

作为一个零和博弈游戏，对抗性的分析与利用是很重要的。这一游戏中的对抗性主要体现在两个方面——落子位置与边界合并。如果落子到对方地界，是提供棋子还是构成阻碍。对于边界区域的合并，是顺其自然、刻意进攻还是注重防守。经过分析，我们发现边界合并的进攻存在程度限制，即一方最多可以进攻得到 4 个格子且容易形成陷入对方包围的局面，收益并不明显，因此我们对这一点总体上保持着顺其自然的态度。而通过在对方地界合适位置落子实现阻碍，我们认为是十分可行的，但这一点在后续的测试中效果并不明显，后来出于对限时的考虑而删去。

问题的核心还是回到了对 2048 游戏本身的分析与处理。可以看到，一方在一轮游戏需要完成两个动作——落子与选择合并方向。其中，落子阶段里判断是否落到对方地界形成阻碍（如果进行）与后来的选择合并方向阶段都离不开对棋面整体形势与局部特征的认识与评估。设计合理高效的评估方法与策略至关重要。在这一点上，我们讨论得到了两种思路：

- 第一种策略参考了 2048 游戏官方积分版的积分规则，2 与 2 合并积 4 分，4 与 4 合并积 8 分，以此类推，将这一规则用于评分评估棋盘的整体局势，在此基础上引入一些游戏过程中得到的游戏技巧与规律（即启发式规则），用以评估棋盘的局部特征。
- 第二种策略则是将整体形势与局部特征结合，引入四种评估标准——最大数、空格数、光滑度、单调性，最大数用于指示当前成绩（部分），空格数用于指示发展潜力，光滑度与单调性则可以指示理想状态的完成度与完成的可能，并通过设置不同的权重系数实现侧重，使估值更加合理。

通过后续的分析与测试，我们发现第二种策略更加贴合实际情况，能够涵盖更加多维度的可能，在测试中的表现也更好，因而最终选择了第二种策略。

以上几个关键点已经确定，接下来要进行的是数据结构与算法的整合与构建。落子部分主要是由多项简单的条件判断组成选择结构，尽量落子在己方区域和双方交界处。合并部分以基本的树结构为基础，将所有可能的格局组织成一棵树，用确定的评估标准对每一格局进行估值，引入 minmax 算法，递归搜索模拟每一步的选择。为了实现更好的效果，我们又引入 alpha-beta 算法进行剪枝，增加搜索深度，使其更加“高瞻远瞩”（具体分析与实现见后文）。

1.2 算法流程图

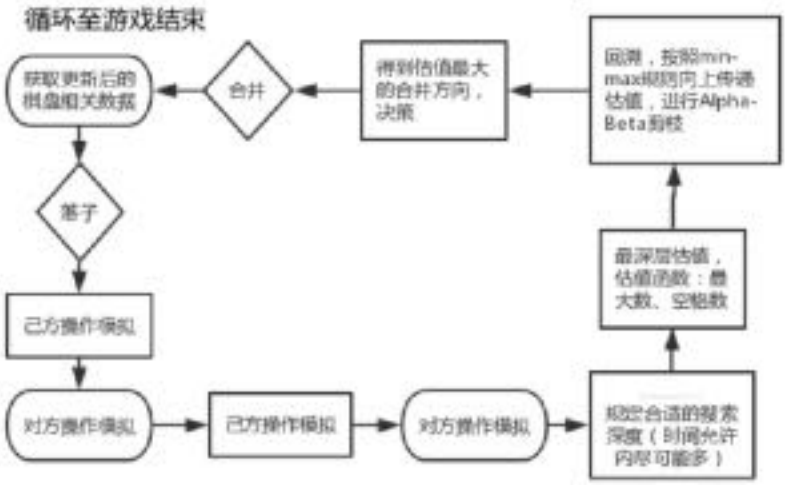


图 1：算法主体流程图

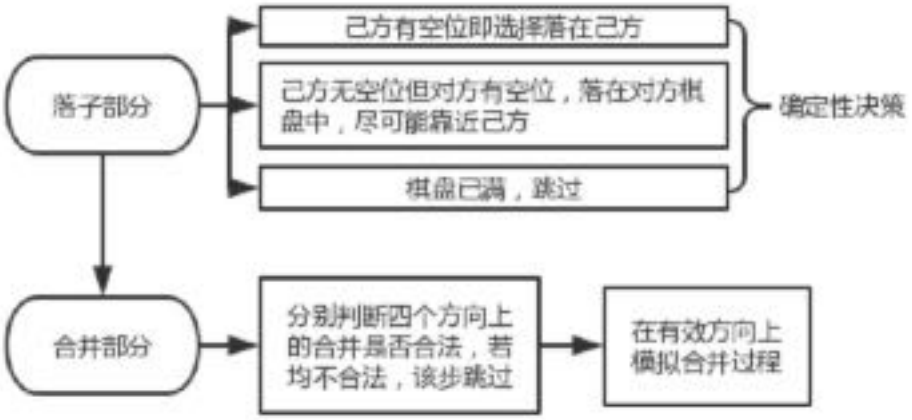


图 2：己方/对方操作模拟（均站在当前方视角）

1.3 算法运行时间复杂度分析

从时间复杂度上来看，本算法中的大部分操作均可由条件判断与确定项数的遍历实现，时间复杂度最高的部分在于对决策树的搜索。由于本算法中的落子部分的决策是确定性的，不参与决策树的构建，而合并部分每一种格局可以向 4 个方向演变，那么如果将搜索深度视为问题的规模 n，时间复杂度即为 O（4^n）。由于在游戏过程中有些方向的演变是无效的（可通过条件判断）且后期引入了 alpha-beta 剪枝算法，实际的时间复杂度会小一些，但总体上仍是指数级。

出乎意料的是，在实际操作中，我们发现，虽然算法中其他部分的时间复杂度远低于 O（4^n），但由于大量重复操作的出现或部分操作自身所需时间较长，除决策树搜索以外的部分耗散的时间同样不容忽视。减少重复操作，精进算法，节约时间以增加搜索深度，增强结果可靠性，是我们一直在努力的方向。从结果上看，算法最终实现了较为理想的搜索深度。

2 程序代码说明

2.1 数据结构说明

本算法落子部分主要是由多项简单的条件判断组成的选择结构，合并部分主要采用的数据结构为树，同时运用了 Python 内置的列表、元组、字典，没有自定义类。

算法合并部分决策树以基本的树结构为基础，进行了一定改造。将所有可能的格局组织成一棵树，树的每一个节点表示一种格局，由父格局经过一步可以到达子格局。我们使用

minmax 算法，通过对以当前格局为根的格局树搜索来确定下一步的选择。进行每一步搜索时，我们都假设我方每一步都会选择能使我方当前理论价值最大的格局方向，对手每一步都会选择能使我方当前理论价值最小的格局方向，即对手具有完美决策能力。这里的理论价值由我们编写的评估函数来衡量。

单纯的 minmax 算法面临的主要问题就是时间复杂度。由于所需搜索的节点数随最大深度增加呈指数膨胀，而算法的效果往往和深度相关，因此会极大限制算法的效果。因此我们使用 alpha-beta 剪枝对 minmax 算法进行补充和改进，增加最大搜索深度：在构造过程中，如果发现当前格局再往下不能找到更好的解，我们就停止在这个格局及以下的搜索，也就是剪枝。操作的基本方法是从根节点开始采用深度优先的方式构造格局树，在构造每个节点时，都会读取此节点的 alpha 和 beta 两个值，其中 alpha 表示搜索到当前节点时已知的最好选择的下界，而 beta 表示从这个节点往下搜索最坏结局的上界。由于我们假设对手会将局势引入最坏结局之一，因此当 beta 小于 alpha 时，就意味着从此处开始不论最终结局是哪一个，其上限价值也要低于已知的最优解，也就是说已经不可能从此处向下找到更好的解，于是就进行“剪枝”，不再搜索下去。

列表主要被用来存储可能的尝试路径和尝试结果，如可移动方向列表 directionList、各方向模拟移动后的价值列表 valueList 和 guessValueList、对方的可落子位置 oppoAvailableList 等等，并用到了列表排序等操作以帮助进行启发式决策。元组主要用来表示棋盘上某一位置的坐标，即（x，y）中的 x 表示行号，y 代表列号。字典用在 valuePosition 函数中，记录存储各坐标上的棋子价值。

2.2 函数说明

代码主体部分分为落子与移动两个方面的策略，落子优先考虑己方落子点，移动部分采用决策树思想进行实现，并有相关的评估函数来进行局面的分析。

一、主干结构

1、落子：整个棋盘已满时返回 None；当己方未满时落在己方；当己方已满而对方未满时，尽量将棋子下在靠近双方交界处的地方。

```
if mode == 'position':#给出己方下棋的位置
    selfAvailable=board.getNext(self.isFirst, currentRound)#己方的允许落子点，元组
    oppoAvailableList=board.getNone(not self.isFirst)#对方的允许落子点，列表
    if not selfAvailable and not oppoAvailableList:#整个棋盘已满
        return None
    elif not oppoAvailableList:#己方未满而对方已满
        return selfAvailable
    elif not selfAvailable:#己方已满而对方未满
        #按列号排序，尽量将棋子下在靠近双方交界处的地方，先后手需分类
        oppoAvailableList.sort(key=lambda x:x[1],reverse=False if self.isFirst else True)
        return oppoAvailableList[0]
    else:
        return selfAvailable
```

2、移动：调用 searchMoveResult 函数搜索最佳合并方向，并设置搜索深度。

```
if mode == 'direction':#给出己方合并的方向
    #此处设置分类讨论的原因是由于我们搜索方式的设置，先后手搜索相同深度所用时间有差异，因此分类讨论，便于调试
    if self.isFirst == True:
        return self.searchMoveResult(currentRound,board,2,-1*0xffffffff,1*0xffffffff)
    else:
        return self.searchMoveResult(currentRound,board,2,-1*0xffffffff,1*0xffffffff)
```

二、主体函数

searchMoveResult 与 minmax 是本程序的核心函数，采用的是决策树与 alpha-beta 剪枝算法。以下将会详细阐述 searchMoveResult 函数与 minmax 函数的实现过程，并展示出它们的代码截图。其余函数将只针对它们的接口功能给出简单的说明。

函数名称	功能	输入	输出
searchMoveResult	给出最佳合并方向	回合数 currentRound 棋盘 board、深度_depth 目前所有可能解中的最大下界 a 目前所有可能解中的最小上界 b	移动方向 int 类型
minmax	求使己方最大/对方最小的估值	所属方 nowcharacter 回合数 currentRound 棋盘 board、深度 depth 目前所有可能解中的最大下界 a 目前所有可能解中的最小上界 b	移动估值 浮点数类型
novalidMove	判断是否有合法移动方向	棋盘 board	bool 类型 (无合法移动时返回 True)
fallPiece	模拟落子	棋盘 board 回合数 currentRound	落子坐标 元组类型
valuePosition	返回某个数所在的位置坐标	棋盘 board、数值 value 所属方 character	所有位置坐标 列表类型

1、searchMoveResult 函数:

当没有合法移动方向时，函数返回 None。当至少有 1 个合法移动方向时，依次进行合法移动后进行盘面估值，返回估值最大的移动方向：若回合数大于 495，盘面估值为己方棋子最大价值*己方价值最大的棋子个数；若回合数小于 495，盘面估值为移动后的盘面估值（调用 minmax，递归到数轮后返回的盘面估值）。注意，如果己方是先手，直接开始探测；如果己方是后手，需要先进行两次模拟落子后再探测。

2、**minmax 函数**

先手部分函数说明：

① 己方是先手，当前轮到己方合并：

- 如果回合数大于 495：

  没有合法移动方向时，直接返回当前己方棋子最大价值*己方价值最大的棋子个数。如果有合法移动方向，则依次探测：移动方向合法时，尝试在模拟盘面合并后，记录己方棋子最大价值*己方价值最大的棋子个数；如果移动方向不合法，记录一个很小的负值-1*0xfffffffff。探测完成后返回记录的四个值中的最大值。

- 如果回合数小于 495：
  - 当 depth 为 0 时：

    如果没有合法移动方向，则返回当前局面己方评估函数值。如果有合法移动方向，则依次探测：移动方向合法时，尝试在模拟盘面合并后，记录模拟的合并后局面己方评估函数值；如果移动方向不合法，记录一个很小的负值-1*0xfffffffff。探测完成后返回记录的四个值中的最大值。

  - 当 depth 不为 0 时：

    依次探测四个方向。如果是合法移动方向，在模拟盘面 boardtmp 尝试合并后，对模拟的合并后局面调用 minmax 函数估值（注意传入参数为 not self.isFirst，currentRound，boardtmp，depth-0.5，a，b）并记录，若记录的这个值大于 a，则令 a=记录值；如果此时 a>b，则返回 b，成功实现剪枝；这时如果函数尚未返回某值，则继续探测下一个合并方向。如果不是合法移动方向，则不进行操作，相当于这条路截断不再搜索。全部探测完成后，若函数还没有运行结束，给上一个 min 节点返回 a。

② 己方是后手，当前轮到对方合并：

- 如果回合数大于 495：

  没有合法移动方向时，直接返回当前己方棋子最大价值*己方价值最大的棋子个数。如果有合法移动方向，则依次探测：移动方向合法时，尝试在模拟盘面合并后，记录己方棋子最大价值*己方价值最大的棋子个数；如果移动方向不合法，记录一个很大的正值 1*0xfffffffff。探测完成后返回记录的四个值中的最小值。

- 如果回合数小于 495：
  - 当 depth 为 0 时：

    如果没有合法移动方向，则返回当前局面己方评估函数值。如果有合法移动方向，则依次探测：移动方向合法时，尝试在模拟盘面合并后，记录模拟的合并后局面己方评估函数值；如果移动方向不合法，记录一个很大的正值

1*0xfffffffff。探测完成后返回记录的四个值中的最小值。

✧ 当 depth 不为 0 时：

依次探测四个方向。如果是合法移动方向，在模拟盘面 boardtmp 尝试合并后，对模拟的合并后局面调用 minmax 函数估值（注意传入参数为 not self.isFirst，currentRound，boardtmp，depth-0.5，a，b）并记录，若记录的这个值小于 b，则令 b=记录值；如果此时 b≦a，则返回 a，成功实现剪枝；这时如果函数尚未返回某值，则继续探测下一个合并方向。如果不是合法移动方向，则不进行操作，相当于这条路截断不再搜索。全部探测完成后，若函数还没有运行结束，给上一个 max 节点返回 b。

后手部分函数说明：由于先手与后手部分代码大体类似，因此仅对后手部分与先手部分有区别的地方加以说明，且下方仅给出先手部分的代码截图。

③ 己方是先手，当前轮到对方合并：

与上述情况②的处理基本相同，区别在于当 depth 不为 0 时，对合并后的模拟盘面还需进行先后两次模拟落子（用模拟落子函数 fallPiece 确定模拟落子位置，fallPiece 的基本策略与我们自己的真实落子策略一致），之后再调用 minmax 函数估值，传入的参数为 not self.isFirst，currentRound+1，boardtmp，depth-0.5，a，b。

④ 己方是后手，当前轮到己方合并：

与上述情况①的处理基本相同，区别在于当 depth 不为 0 时，对合并后的模拟盘面还需进行先后两次模拟落子（用模拟落子函数 fallPiece 确定模拟落子位置，fallPiece 的基本策略与我们自己的真实落子策略一致），之后再调用 minmax 函数估值，传入的参数为 not self.isFirst，currentRound+1，boardtmp，depth-0.5，a，b。

```
def minmax(self,newcharacter,currentRound,board,depth,a,b):
    annoyingPlayer = Player(not self.isFirst,self.array)#对手Player类
    if newcharacter == True:#我的轮到先手
        directionList=[3,0,1,2]#左上下右
        if self.isFirst == True:#己方是先手，当前轮到己方合并
            #此部分是 我们决定我们的合并方向
            if currentRound < 495:
                guessValueList=[0,0,0,0]
                flagtmp=False
                for direction in directionList:
                    boardtmp = board.copy()#在copy后的棋盘上尝试四个方向的合并操作，不会改变原棋盘
                    if boardtmp.move(self.isFirst,direction):#如果是合法移动方向，尝试合并后，记录己方局面得分
                        guessValueList[direction]=self.biggest(boardtmp,self.isFirst)*len(self.valuelist)+self.h(boardtmp,self.biggest(boardtmp,self.isFirst),self.isFirst)
                        flagtmp=True
                    else:#如果不是合法移动方向，这条路截断不再探索
                        guessValueList[direction]=-1*0xfffffffff
                if flagtmp == True:
                    return max(guessValueList[0],guessValueList[1],guessValueList[2],guessValueList[3])
                else:#没有合法移动方向时，，返回当前自己局面得分
                    return self.biggest(board,self.isFirst)*len(self.valuelist)+self.h(board,self.biggest(board,self.isFirst),self.isFirst)
```

```
            #此处不管是第一轮，还是我们决定的合并方向
            if depth == 0:#如果深度为0
                guessValueList=[0,0,0,0]
                flagtmp=False
                for direction in directionList:
                    boardtmp = board.copy()#在copy后的棋盘上尝试四个方向的合并操作，不会改变原棋盘
                    if boardtmp.move(self.isFirst,direction):#如果是合法移动方向，尝试合并后，记录当前局面己方评估的数值
                        guessValueList[direction]=self.evaluate(boardtmp)
                        flagtmp=True
                    else:#如果不是合法移动方向，这条路截断不再探索
                        guessValueList[direction]=-1*0xfffffffff
                if flagtmp == True:
                    return max(guessValueList[0],guessValueList[1],guessValueList[2],guessValueList[3])
                else:#没有合法移动方向时，返回当前局面己方评估函数值
                    return self.evaluate(board)
            else:#如果深度不为0
                guessValueList=[0,0,0,0]
                for direction in directionList:
                    boardtmp = board.copy()#在copy后的棋盘上尝试四个方向的合并操作，不会改变原棋盘
                    if boardtmp.move(self.isFirst,direction):#如果是合法移动方向，尝试合并后，depth减0.5，currentRound不变
```

```
            guessValueList[direction]=self.minmax(not self.isFirst,currentRound,boardtmp,depth-0.5,a,b)
            if guessValueList[direction] > a:
              a = guessValueList[direction]
            if a > b:
              return b#返回最大值，剪枝
          else:#如果不是合法移动方向，这条搜索树不再搜索
            guessValueList[direction]=-1*0xffffffff
        return a#除上一个max节点返回最小值
      else:#己方是先手，当前轮到对方合并
        #轮数+0.5，表达探测可能的合并方向
        if currentRound > 495:
          guessValueList=[0,0,0,0]
          flagtmp=False
          for direction in directionList:
            boardtmp = board.copy()#在copy出的棋盘上尝试各个方向的合并操作，不会改变原棋盘
            if boardtmp.move(not self.isFirst,direction):#如果是合法移动方向，尝试合并后，记录己方局面得分
              guessValueList[direction]=self.biggest(boardtmp,self.isFirst)*10+(self.valuePosition(boardtmp,self.biggest(boardtmp,self.isFirst),self.isFirst)
              flagtmp=True
            else:#如果不是合法移动方向，这条路径不再搜索
              guessValueList[direction]=1*0xffffffff
          if flagtmp == True:
            return min(guessValueList[0],guessValueList[1],guessValueList[2],guessValueList[3])


        else:#对方没有合法移动方向时，返回当前自己局面得分
          return self.biggest(board,self.isFirst)*10+(self.valuePosition(board,self.biggest(board,self.isFirst),self.isFirst))
      #轮数+0.5，表达探测可能的合并方向
      if depth == 0:#搜索深度为0
        guessValueList=[0,0,0,0]
        flagtmp=False
        for direction in directionList:
          boardtmp = board.copy()#在copy出的棋盘上尝试各个方向的合并操作，不会改变原棋盘
          if boardtmp.move(not self.isFirst,direction):#如果是合法移动方向，尝试合并后，记录当前局面己方评估函数值
            guessValueList[direction]=self.evaluate(boardtmp)
            flagtmp=True
          else:#如果不是合法移动方向，这条搜索树不再搜索
            guessValueList[direction]=-1*0xffffffff
        if flagtmp == True:
          return min(guessValueList[0],guessValueList[1],guessValueList[2],guessValueList[3])
        else:#对方没有合法移动方向时，返回当前局面己方评估函数值
          return self.evaluate(board)
      else:#搜索深度不为0
        guessValueList=[0,0,0,0]
        for direction in directionList:
          boardtmp = board.copy()#在copy出的棋盘上尝试各个方向的合并操作，不会改变原棋盘
          if boardtmp.move(not self.isFirst,direction):#如果是合法移动方向，尝试合并后，depth减0.5，currentRound不变

            guessValueList[direction]=self.minmax(self.isFirst,currentRound,boardtmp,depth-0.5,a,b)
            if guessValueList[direction] < b:
              b=guessValueList[direction]
            if b <= a:
              return a#返回最小值，剪枝
          else:#如果不是合法移动方向，这条路径不再搜索
            guessValueList[direction]=1*0xffffffff
        return b#除上一个max节点返回最大值
```

三、评估函数部分

函数名称	功能	输入	输出
biggest	求棋盘上某方棋子的最大值	棋盘 board 所属方 character	最大数 int 类型。棋盘为空时返回 0
getEmptyNum	求己方空格数	棋盘 board	空格数 int 类型
smoothness	对棋盘光滑度进行估值	棋盘 board	对光滑度的估值，int 类型
monotonicity	对棋盘单调性进行估值	棋盘 board	对单调性的估值，int 类型
evaluate	赋予各个部分权重，进行总评分	棋盘 board	赋予权重后各项之和，浮点数类型

其中，光滑度用于测量网格的平滑程度，这些块的值可以形象地解释为海拔。相邻两个方块的值差异越小，格局就越平滑。单调性用于衡量沿某一行/列的不同棋子价值变化的单调程度，我们希望在向左/向右和向上/向下的方向，方块的值都是尽量递增或递减的。

我们最终的版本的评估函数中没有用到光滑度与单调性的估值，仅将最大值与空格数纳入考虑。其中，最大值权重系数为 10，空格数权重系数为 1。

2.3 程序限制

一、可能出现的错误

经过初期排查调试，程序在各次测试中已经表现良好，不出现报错，除超时以外基本排除了其他意外终止运行的可能。后期天梯更新后出现了数组越界的报错（500 回时），已经予以及时修正。合理规定最大深度，就可以把运行 500 回合后的思考时间限制在 5 秒以内，规避超时的发生。

二、程序本身的缺陷

程序缺乏攻击性，即不擅长在边界吞并对方棋子，而我方边界棋子很容易被对方吞并。可能原因是没有合适的启发式规则，以及评估函数没有将己方和对方分开考虑对。AI 缺乏冒险性和攻击性使得分数增长有时很缓慢，在遇到特定对手时会出现大量白给的情况，输赢有一定运气成分，在面对实力较为强劲或极具攻击性的对手时获胜的可能性极小。尝试优先考虑向交界处合并，收效一般。

另外，程序的时间管理尚待改进。所用最大思考时间仅为 2 秒多，然而深度再增加即超时，因而实力也受到了一定限制。

三、程序调试的不足

程序中的参数大多凭经验进行初始赋值，后期经过 5～10 轮的调整以尽可能达到最优值。但由于调整范围过大、考虑尚有不周等等，很难让程序达到尽善尽美，充分发挥功能。由于时间有限，没有用蒙特卡洛算法或基因算法等进行尝试、帮助调参，最终代码仍有很大的改善空间。

3 实验结果

3.1 测试数据

一、实验环境说明：

- 硬件配置：（CPU/内存）1.4 GHz Intel Core i5 / 8 GB
- 操作系统：（名称/版本）macOS Catalina 10.15.3
- Python 版本：（版本号）3.7.6

二、测试方法：

1、利用 SESSDSA 测试平台找出代码中的 bug 并修改，确保我方代码运行时不会发生错误或产生非法输出。

2、利用 SESSDSA 测试平台与技术组 bot 自由对战、与其他选手进行匹配赛

a) 观察每次比赛用时，逐步调整搜索深度，确保对战时不会超时；

b) 观察战局及失败原因，改进代码算法、策略，修改权重系数等，争取获得更高的分数。

三、测试结果：

版本	对战	平均胜率	搜索深度	出现问题	改进方法
1	D	0.00%	2+0/1+0	1.搜索深度小，且极易超时 2.实力弱，对战 D 级时比分完全没有优势	1.删去在模拟合并时部分重复调用的 board.copy()操作，以节省时间
2	D	6.67%	4+2/1+0	1.搜索深度仍较小，我方先手时极易超时 2.我方后手时实力仍然较弱，因为搜索深度较小	1.稍微增加后手深度，以提高整体实力
3	D	16.7%	4+2/3+2	1.整体实力没有大的变化 2.仍然存在超时问题	1.修改落子函数，放弃启发式规则的决策，以节约时间 2.评估函数中尝试删去耗时较长的平滑性与单调性检测 3.代码中减少对 noValidMove 函数的调用 4.尽量加深搜索深度
4	D	93.3%	6+4/3+2	1.现有落子策略比较简单 2.现有搜索方式是按照 1 个完整回合为单位的，导致难以充分利用时间	1.改进搜索结束条件，使搜索深度单位为半个回合，从而调整深度更为灵活
	C	83.3%			
5	C	83.3%	5+4/4+4	1.胜率没有显著改进，需调整权重系数、改进落子策略 2.调参中发现深度增加胜率反而降低，需改进评估函数	1.调整权重系数： 空格数：2.7→1 最大数：1.0→10
6	C	90.0%	5+4/4+4	1.对战 B 我方为后手时，出现超时的问题	1.更新落子策略，采用一些启发式规则
	B	0.00%			
7	D	66.7%		1.可搜索深度大大降低，并且出现一些报错，实力降低	1.放弃启发式规则，对原有落子策略稍作改动 2.更改评估函数，极大放大棋子价值的占比 3.摸索调整搜索深度
	C	50.0%			
8	C	100%	5+4/4+4	1.没有什么大的问题，实力稍有起色，继续改进	1.增加 alpha-beta 剪枝用于增加搜索深度 2.相应的修改评估函数
	B	10.0%			
9	C	100%	6+4/6+6	最终版的问题的改进方向详见 4.4.2 经验与教训	
	B	16.7%			

注 1：对战指与技术组 bot 自由对战

注 2：平均胜率指 30 局中：胜利局数/总局数×100%

注 3：搜索深度指的是：

(我方先手)模拟合并次数+模拟落子次数/(我方后手)模拟合并次数+模拟落子次数

其中合并次数与落子次数都是我方与对方的总次数

图 3：最终版本与 bot_C 和 bot_B 的对战

3.2 结果分析

一、策略及作用效果

本组的代码大致可分为三个部分：落子函数，评估函数，决策树。在不断的调试中，我们发现搜索深度往往是影响整体实力的关键。因此，三个函数的改进绝大部分是围绕减少时间复杂度，为增加搜索深度提供足够时间展开的。

1、落子函数

落子函数主要起判断落子位置的作用。前期本组的落子函数条件判断较多、运行时间较长，后根据观察对战不断修改简化，代码整体性能前后变化不大，但为决策树增加搜索深度提供较宽裕的时间。整体上来说，有利于增强本组的实力。

2、评估函数

评估函数包括四种评估标准：最大数、空格数、光滑度、单调性，用于评估当前回合的局面的好坏。由于光滑度与单调性检测耗时较长，选择将两个函数删去；对于最大数和空格数的评估，不断修改权重系数。总的来说，虽然评估性能有所降低，但一定程度上减少了所

需时间，并且最大数和空格数的评估得到了优化。

3、决策树

决策树是代码的主体部分，通过模拟我方与对方的落子和合并方向，判断本局我方的合并方向。minmax 算法主要用于模拟我方最佳落子与合并方向以及对方最佳落子与合并方向；后增加的 alpha-beta 算法主要用于剪枝以增加搜索深度。最终，在最后一版的代码中，我们最好能够做到我方先手时进行 8 次模拟合并及 6 次模拟落子，我方后手时进行 8 次模拟合并及 8 次模拟落子。

二、时间开销

超时一直是我们组面临的一个比较大的问题，在搜索深度很小的情况下，仍然会出现超时的问题。后经过我们不断总结，发现以下一些地方所需时间开销较大：

- 模拟合并时多次出现重复调用 board.copy()操作
- 落子函数采用启发式规则时由于条件判断较复杂，时间开销较大
- 代码中平滑性检测与单调性检测花费时间较长
- 出现 noValidMove 函数的不必要的重复调用
- 搜索深度的最小单位还比较大，导致增加 1 层超时，减少 1 层时间利用率不高

通过改进这些问题，并且加上 alpha-beta 算法剪枝后，算法整体的时间在预期之内。

3.3 经典战局

我们组的算法在策略上并没有采用太多启发式规则，比较中规中矩，因此在对战中没有特别对战效果。以下是我们选取的最后一版代码与 bot_C 对战时，一些全面胜利的局面。

图 4：我方为左边一方

图 5：我方为左边一方

4 实习过程总结

4.1 分工与合作

一、小组分工

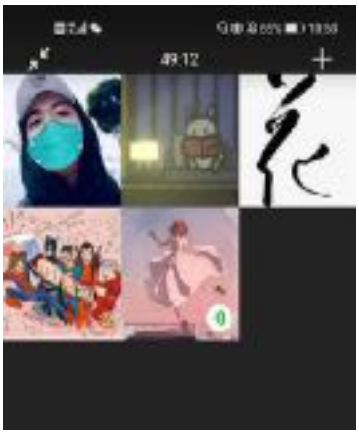
- 编写代码以及后续算法开发调整：李子涵，陈湘
- 算法思路：文雨涵，孙久雯，默文婷
- 报告（算法思路部分）：孙久雯
- 报告（程序代码说明部分）：陈湘，李子涵
- 报告（实验结果及摘要部分）：文雨涵
- 报告（实习过程总结及致谢部分）：默文婷

二、合作与交流

参考老师的意见以及其他组的情况，我们认为三个同学讨论思路、两个同学负责代码的分配比较合理。在第一次全员开会中，我们进行了分工，李子涵和陈湘两位同学负责代码部分，文雨涵、孙久雯和默文婷三位同学负责算法思路部分。

负责算法思路的三位同学，进行了两次大型的语音通话交流。第一次每个人分享整体的思路结构，通过讨论决定共同的思路结构，并进行分工。第二次大家完善了自己负责部分的细节，把思路用类似伪代码的形式表现出来。

负责代码的两位同学，一位负责 minmax 函数的实现，另一位负责评估函数，由于代码组只有两位同学，她们主要使用微信私信交流。


第一次语音会议

算法思路组第一次讨论

算法思路组第二次讨论

4.2 经验与教训

一、积累的经验

在本次实习作业中，我们小组分工比较明确，让每个同学尽可能地发挥自己的长处。组内氛围轻松融洽，成员们相互沟通、合作，彼此之间产生了积极的影响。

通过本次实习活动，我们每个人都收获了很多。首先，我们锻炼了合作能力，在交换思路、分享灵感的过程中，每个人的思维都得到了拓展。其次，在绞尽脑汁思考的过程中，我们对于算法、数据结构的理解在也不断加深，编写代码的过程更是强化了编程技术。此外，这也是我们第一次尝试去实现人工智能，用代码、数据结构去模仿人类的决策，这也锻炼了我们用算法思维思考问题。还有很重要的一点是，这次实习作业不同于平时的作业，它没有一个衡量是否通过的指标，我们能做的只是不断地改进，不断地向更智能、更完美的策略靠近。我们根据以往的对战记录，一遍一遍地调试、修改，这个过程不仅锻炼了分析的能力，更催化出一种臻于至善的精神，为我们将来面对更大型、更具挑战性的任务，打下了基础。

二、教训与改进

本次实习作业，我们也有很多值得改进的地方。

首先，负责算法思路的同学具备的数据结构课外知识比较有限，因此前两次讨论的算法比较基于自己玩游戏时的经验，即启发式规则，以条件讨论为主。而几天后直播课上老师对于决策树的讲解让我们意识到自己算法在结构性的欠缺。在我们改进的过程中，起初只是把 minmax 函数融入算法的一部分，之后慢慢才让决策树成为判断合并方向的结构，这也使得我们的决策树和其他组的相比不够成熟。

此外，我们的评估函数和参数设置都是根据人脑经验直觉和对实验记录的主观判断来编写调整的，难免有些局限。后续的完善中，期望能融入蒙特卡洛树算法和基因遗传算法，促进机器学习，完善启发式体系。

4.3 建议与设想

一、小小的建议

本次实习作业整体上已经十分完善，规则清晰，对战方式设计的合理，同时技术组还向我们提供了稳定的、可视化的平台。有一点建议是，大作业的战线可以拉的长一点，感觉对于算法思路讨论的时间比较有限，所以在后期修改的规模比较大，老师对于决策树的讲解可以再提前一些。决赛上，同学们的对决非常精彩，也期待能够看到他们对于自己策略结构的分析，让我们认识到自己的不足。

二、一点设想

本次实习作业有点像普通版本的 2048 与球球大作战的结合体，未来也期待课程组设计更多的由两个大家都熟悉的游戏结合的新游戏。大胆猜测，双人对战的 2048 游戏也可以推广到四人对战（每名参赛者都可以向相邻的两个对手的棋盘发起进攻）。

5 致谢

感谢陈斌老师用幽默的课堂为我们打开了数据结构的大门，丰富了我们的思维，激发了我们对于算法的兴趣，也激励我们在此次实习作业中灵活地开动脑筋。

感谢技术组的成员们，集思广益设计出这样贴近生活又有趣味的比赛，把日常的游戏增添创意后搬进课堂。同时，感谢他们开发出了稳定的、可视化的对战平台，为我们提供了测试、改进的优良环境。

当然也要感谢组内的五位同学。我们互相促进、合作，较为圆满地完成了任务，一起拥有了这段有趣的经历。

6 参考文献

[1] https://blog.csdn.net/weixin_42165981/article/details/103263211
参考内容：minmax 算法和 alpha-beta 剪枝详解
[2]https://blog.csdn.net/qq_32767041/article/details/80634244
参考内容：四要素评估函数的设计
[3]https://gitee.com/chbpku/sessdsa.2048/tree/1d87dfc4f374493b9c9d29bd108730a480cd756c/user
参考内容：本地测试与调试环境的使用

数据结构与算法课程实习作业报告

许喆鹏* 王宇航 钱渠成 王方瞳 施童耀

摘要：在对弈过程中通过构建决策树对每一步操作可能产生的结果进行深度搜索，并使用估值函数对涉及到的局面进行估值，选择最有利的操作。在搜索深度和时间优化方面，采用了 alpha-beta 剪枝。估值函数对局面的单调性，平滑性，空格数，棋子数字总和，最大值等性质进行估算并线性加总。决策树采用了最大最小算法策略，假想对手最优的可能操作来进行多层搜索，最多可搜索到 4 层。

关键字：决策树 alpha-beta 剪枝 估值函数 最大最小策略

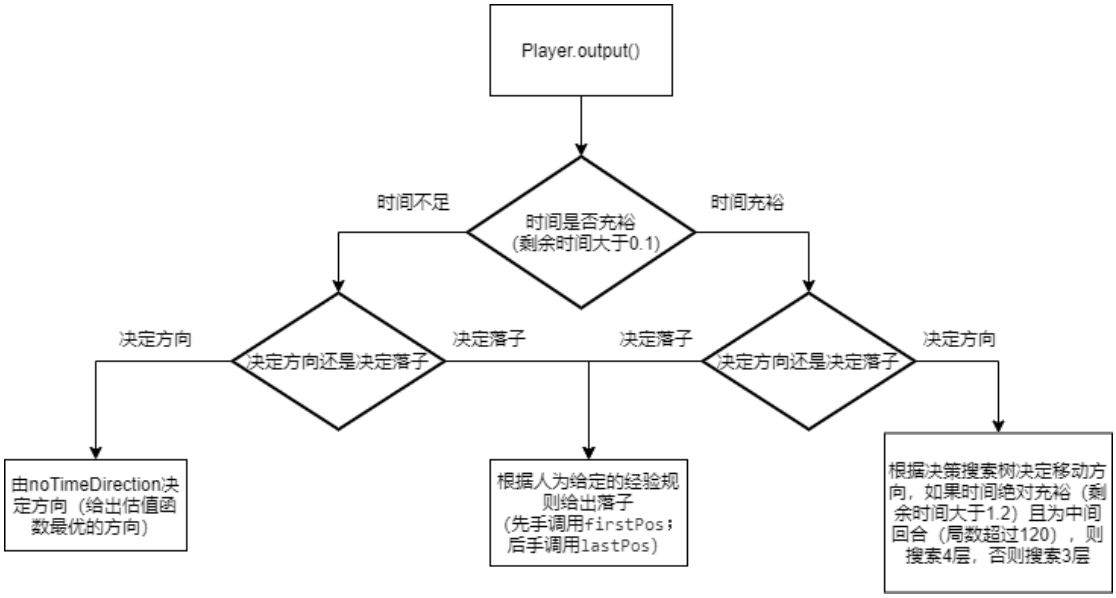
1 算法思想 1
1.1 总体思路 1
1.2 算法流程图 2
1.3 算法运行时间复杂度分析 2
2 程序代码说明 2
2.1 数据结构说明 2
2.2 函数说明 2
2.3 程序限制 3
3 实验结果 3
3.1 测试数据 3
3.2 结果分析 6
3.3 经典战局（可选） 6
4 实习过程总结 7
4.1 分工与合作 7
4.2 经验与教训 9
4.3 建议与设想 10
5 致谢 10
6 参考文献 10

1 算法思想

1.1 总体思路

本算法为使用人为给出的估值函数的 alpha-beta 剪枝算法，通过决策树决定下棋方式，算法最初决定有王宇航负责决策树部分；施童耀，钱渠成和许喆鹏分别独自给出一套估值函数，在每个人都给出估值函数之后开会讨论，充分交流各自的思路，最后由钱渠成同学负责汇总估值函数。在实际运行中，发现决策树很耗费时间，所以只有决定移动方向时候使用了决策树，在决定落字时采取经验规则。

1.2 算法流程图



1.3 算法运行时间复杂度分析

算法运行时间主要由决策树消耗，时间复杂度为 $O(2^n)$,n 为决策树搜索深度。
估值函数 smoothness()与 monotonic()均遍历所有者棋盘的每一个空格，时间复杂度约为 O（n）,n 为所有者拥有的棋盘格数。

2 程序代码说明

2.1 数据结构说明

算法中所用到的主要的数据结构为决策树。

2.2 函数说明

1.落子函数

a.先手落子—firstPos

主要思想为：如果对面即将吃掉我方棋子且无法闪避，则下子阻碍对方吃子；实现回收（即在对方下子后己方立即将其吃掉）；除此之外尽量在我方下子。

b.后手落子—lastPos

比先手落子增加一条规则，即防止自己新下的子被对方吃掉

2.noTimeDirection

时间不够时决定移动方向的函数，选择移动后估值函数最优的方向

3.估值函数—evaluation

为了量化棋局，我们根据博客（https://blog.codinglabs.org/articles/2048-ai-analysis.html）所提供的思路，定义了反应局势的几个函数—敌我局面的单调性（反应棋子在行列排列上的单调程度），敌我局面的光滑性（反应棋子大小分布是否光滑，即相邻棋子间的大小梯度），敌我的空格数，我方最大的棋子，我方最大的三个棋子，敌方所有棋子数值之和和我方所有棋子数值之和。并用这些参数人为的给出加权平均得到估值函数，并且以 350 局为界更换估值函数。

4.决策树搜索函数 search

采用 alpha-beta 剪枝的决策树，根据估值函数实现决策。其中用到了函数 imagineEnemy 用于实现对对方落子的模拟。

2.3 程序限制

无

3 实验结果

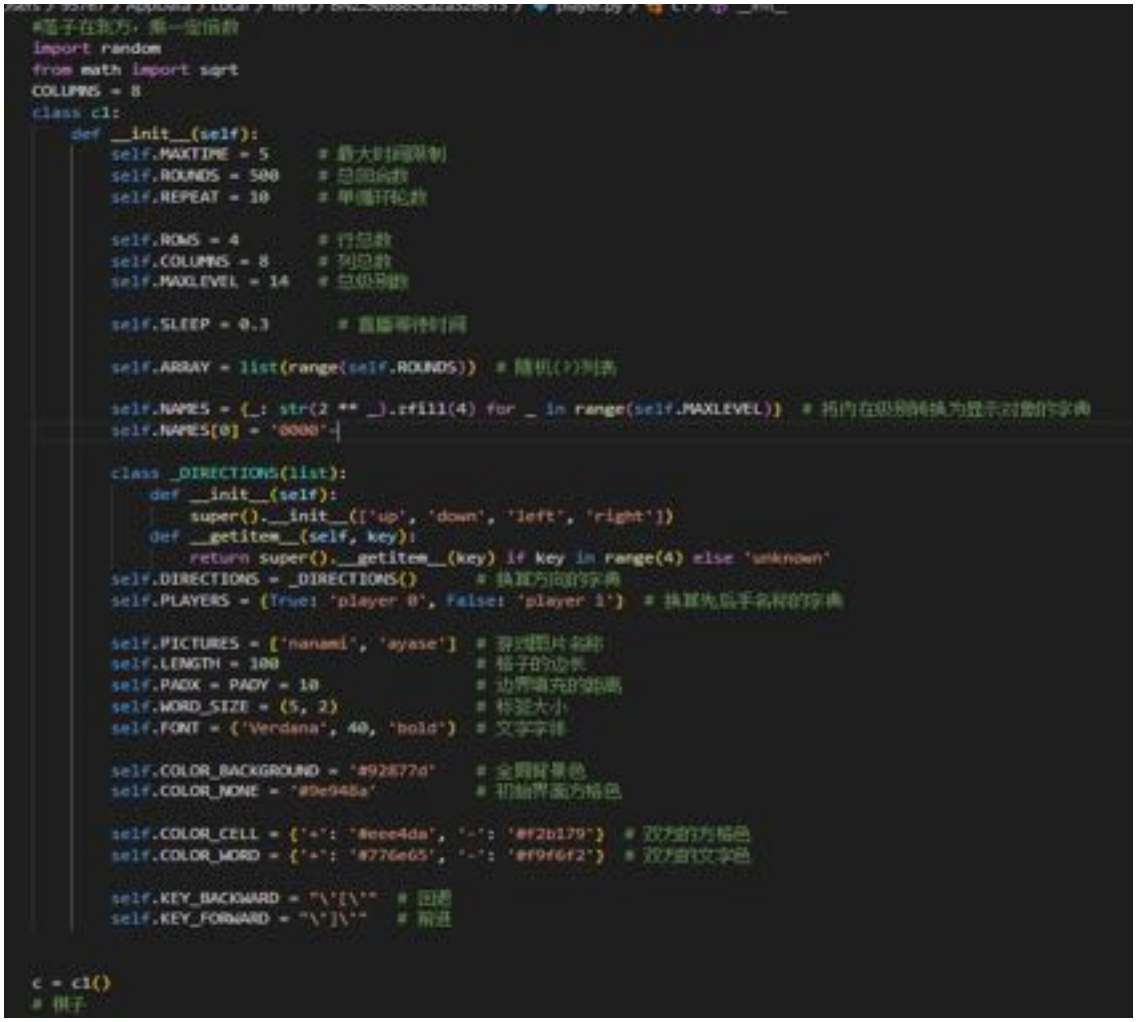
3.1 测试数据

实验环境说明：

- 硬件配置：（CPU/内存） i5-9300H,内存 8G
- 操作系统：（名称/版本）Windows10 专业版 版本号：18363.836
- Python 版本：（版本号）python 3.8

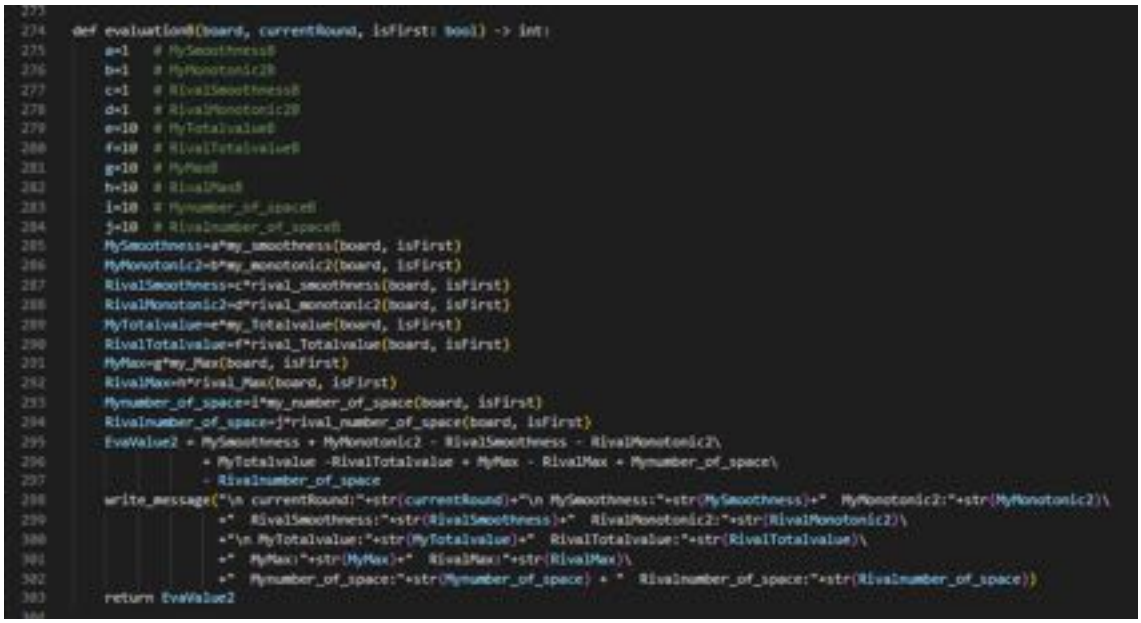
测试主要分为线下测试和线上测试，线下测试用于基本构造层面的 debug，以及参数合意性的主观估计。构造了可以在本地对战过程中将参数变化的细节写进文档用于调节参数的函数以及通过列表生成指定棋盘的函数。后期进行了大量线上对战，通过天梯匹配对战和对指定代码对战积累了经验，并不断修改参数，优化决策树。

（一）线下测试的所用测试程序的局部截图（太长了 23333）



```
#箱子在我方，单一型函数
import random
from math import sqrt
COLUMNS = 8
class c1:
    def __init__(self):
        self.MAXTIME = 5      # 最大时间限制
        self.ROUNDS = 500     # 总回合数
        self.REPEAT = 10      # 单循环轮数

        self.ROWS = 4         # 行总数
        self.COLUMNS = 8      # 列总数
        self.MAXLEVEL = 14    # 总级别数

        self.SLEEP = 0.3      # 直播等待时间

        self.ARRAY = list(range(self.ROUNDS))  # 随机()列表

        self.NAMES = {_: str(2 ** _).rfill(4) for _ in range(self.MAXLEVEL)}  # 将内存级别转换为显示对象的字典
        self.NAMES[0] = '0000'

        class _DIRECTIONS(list):
            def __init__(self):
                super().__init__(['up', 'down', 'left', 'right'])
            def __getitem__(self, key):
                return super().__getitem__(key) if key in range(4) else 'unknown'
        self.DIRECTIONS = _DIRECTIONS()      # 换算方向的字典
        self.PLAYERS = {True: 'player 0', False: 'player 1'}  # 换算先后手名称的字典

        self.PICTURES = ['nanami', 'ayase']  # 资源图片名称
        self.LENGTH = 100                    # 格子的边长
        self.PADX = PADY = 10                # 边界填充的距离
        self.WORD_SIZE = (5, 2)              # 标签大小
        self.FONT = ('Verdana', 40, 'bold')  # 文字字体

        self.COLOR_BACKGROUND = '#92877d'    # 全局背景色
        self.COLOR_NONE = '#9e948a'          # 初始界面方格色

        self.COLOR_CELL = {'+': '#eee4da', '-': '#f2b179'}  # 双方的方格色
        self.COLOR_WORD = {'+': '#776e65', '-': '#f9f6f2'}  # 双方的文字色

        self.KEY_BACKWARD = "\'[\'"  # 回退
        self.KEY_FORWARD = "\']\'"   # 前进


c = c1()
# 例子
```

（主体采用了开发者文档中的底层运行文件）



```
def evaluation0(board, currentRound, isFirst: bool) -> int:
    a=1   # MySmoothness0
    b=1   # MyMonotonic20
    c=1   # RivalSmoothness0
    d=1   # RivalMonotonic20
    e=10  # MyTotalvalue0
    f=10  # RivalTotalvalue0
    g=10  # MyMax0
    h=10  # RivalMax0
    i=10  # Mynumber_of_space0
    j=10  # Rivalnumber_of_space0
    MySmoothness=a*my_smoothness(board, isFirst)
    MyMonotonic2=b*my_monotonic2(board, isFirst)
    RivalSmoothness=c*rival_smoothness(board, isFirst)
    RivalMonotonic2=d*rival_monotonic2(board, isFirst)
    MyTotalvalue=e*my_Totalvalue(board, isFirst)
    RivalTotalvalue=f*rival_Totalvalue(board, isFirst)
    MyMax=g*my_Max(board, isFirst)
    RivalMax=h*rival_Max(board, isFirst)
    Mynumber_of_space=i*my_number_of_space(board, isFirst)
    Rivalnumber_of_space=j*rival_number_of_space(board, isFirst)
    EvaValue2 = MySmoothness + MyMonotonic2 - RivalSmoothness - RivalMonotonic2\
                + MyTotalvalue -RivalTotalvalue + MyMax - RivalMax + Mynumber_of_space\
                - Rivalnumber_of_space
    write_message("\n currentRound:"+str(currentRound)+"\n MySmoothness:"+str(MySmoothness)+"  MyMonotonic2:"+str(MyMonotonic2)\
                  +"  RivalSmoothness:"+str(RivalSmoothness)+"  RivalMonotonic2:"+str(RivalMonotonic2)\
                  +"\n MyTotalvalue:"+str(MyTotalvalue)+"  RivalTotalvalue:"+str(RivalTotalvalue)\
                  +"  MyMax:"+str(MyMax)+"  RivalMax:"+str(RivalMax)\
                  +"  Mynumber_of_space:"+str(Mynumber_of_space) + "  Rivalnumber_of_space:"+str(Rivalnumber_of_space))
    return EvaValue2
```

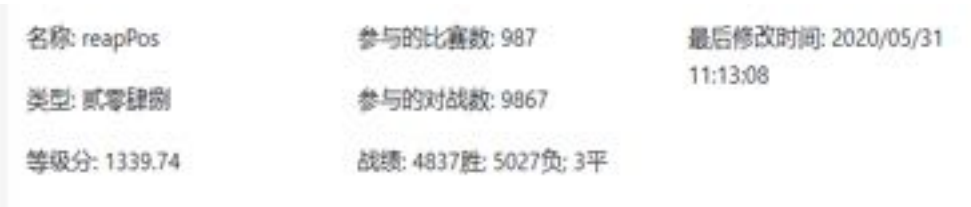
（程序中 player 函数的参数调整框架，可将估值细节导入文档）

```python
def make_chessboard_from_list(sample_list:list,i_am_right:bool)->Chessboard:
    sample_chessboard=Chessboard('qwq')
    for i in range(32):
        (x,y)=divmod(i,8)
        # position=(y,x)
        position=(x,y)
        value=sample_list[i]
        if sample_list[i]>0:
            sample_chessboard.belongs[i_am_right].append(position)
            belong=i_am_right
        elif sample_list[i]<0:
            sample_chessboard.belongs[not i_am_right].append(position)
            belong=not i_am_right
        else:
            if y<4:
                sample_chessboard.belongs[i_am_right].append(position)
                belong=i_am_right
            else:
                sample_chessboard.belongs[not i_am_right].append(position)
                belong=not i_am_right
        sample_chessboard.board[position]=Chessman(belong, position, value)
    return sample_chessboard

def judge_chessboard(sample_list:list)->bool:
    for row in range(4):
        div=-1
        for col in range(8):
            position=col+row*8
            if sample_list[position]>0:
                if col-div==1:
                    div+=1
                else:
                    return False
    return True

sample_list=[ 1, 1, 1, 1,-1,-1,-1,-1,
              0, 0,-1,-1, 0, 0,-3, 0,
              0, 0, 0, 0, 0, 0, 0, 0,
              0 ,0 ,0, 0, 0, 0, 0, 0]

# print(judge_chessboard(sample_list))
print(make_chessboard_from_list(sample_list,True))
```

（用列表生成棋盘的函数）

（二）线上测试数据

小组对战总参与场次 9867 场，胜率 49%。



在线上平台除了随机匹配其他小组代码进行对抗外，小组内也通过一套体系下不同参数的代码相互对抗，“手动炼丹”。

helptry3	贰零肆捌	2赛20战- 18胜2负0平	1006.09
helptry—plus	贰零肆捌	5赛50战- 19胜31负0平	976.72
helptry—plustry	贰零肆捌	3赛30战- 10胜20负0平	1001.96

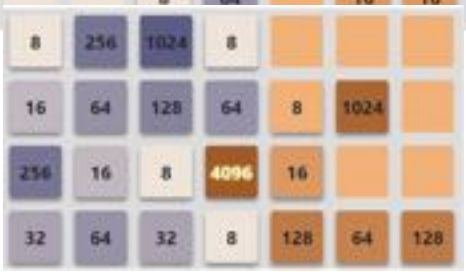
3.2 结果分析

最初小组内负责代码的同学分别写出了三套不同的估值函数，通过测试和热身赛合并出了一套效率比较高的函数。起初由于参数设置过于偏激，算法进攻性过强，经常为了吃子而一去不回（见下图先手方），且合并能力不强。后通过修改参数赋值比重，将算法的重心放

在了最优合并上，在前期和后期使用两种不同的取场面最大值的函数。此外，在算法构造初期超时现象严重，后通过有效剪枝和时间控制成功地避免了超时问题。主要的时间开销花费在决策树的搜索上，由于树的搜索是指数级时间复杂度，运行的时间开销主要发生在对决策数的搜索中。在各小组普遍使用决策树的情况下，搜到的层数更多者有着压倒性的优势。后期发现适度牺牲估值函数准确性将时间分配到搜索更多的层数中更为有用，并对估值函数进行了精简。

3.3 经典战局（可选）

在早期的一场比赛中，由于参数设置不够均衡，造成了一些匪夷所思的送人头操作，例如在一场天梯赛中为了进行一些并不决定胜负的移动而将自己的最大子送予敌手，直接输掉了比赛（我方为先手）：



由对局细节可看出我方为了合并两个 16 而送掉了最大值 2048。针对这一问题，我们适度调高的最大值重要性，并降低了合并的赋值，在后续的对战中，这一现象得到了明显的改善。

后期在天梯中的一场与名为 alpha 的代码比赛的过程中，我们的代码呈现出了均衡的决策评估和较强的进攻性，在一枚值为 256 的棋子突入对方领地后与我方外围棋子内应外合，连续吃掉对方许多较大的棋子并在对方境内达到难以被吃掉的数字。比赛截图如下

4 实习过程总结

4.1 分工与合作

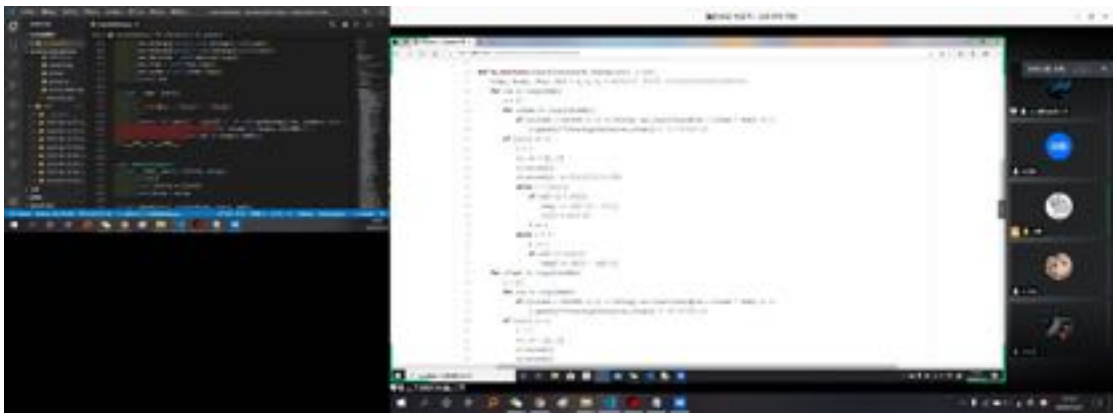
4.1.1 分工

许喆鹏，钱渠成，施童耀负责最初版本的估值函数 smoothness(),monotonic(),space() 的编写，王宇航负责决策数的构建和算法的整合，王方瞳和许喆鹏撰写报告，在后期各组员共同参与了对参数的调试，王宇航和钱渠成完成了绝大部分的调试和比赛（感激）。

4.1.2 合作

组会主要以腾讯会议的形式在线上召开，在会议中对前一阶段的工作进行总结报告并对接下来几天的任务进行分工。在第一次会议主要确定了算法的基本框架，组内任务的基本分工。第二次组会分享了三种不同思路的估值函数，并就每个思路进行了评价和实验。后期调试过程的讨论均在微信群中进行。

第一次组会腾讯会议号：230979760 ， 会议时间 2020.5.19



会议主要内容：

- 提出剪枝决策树和蒙特卡洛算法可行性
- 先写一个函数然后通过对战慢慢调试
- 先后手要分不同的策略
- 如何构建和搜索树
- 建立对局面分析的函数
- 对每个人进行了运行环境的指导
- 确定了码云的使用，在码云上建立了自己的分支并学习了如何修改
- 分享了 markdown 的打开方法
- 决定了一个人整合代码、算法，一个人写最后的文档，三个人写代码
  - 施童耀、钱渠成、许喆鹏写代码
  - 王方瞳写文档
  - 王宇航整合代码、算法
- 文档包括测试过程、分工、会议记录、代码思路
- 下次会议是明天下午 3：00~5：00
- 查看资料后，确定一下算法的框架

（截图）

第二次组会腾讯会议号：759315544 会议时间 2020.5.20

（截图）

会议主要内容：

- 在剪枝和蒙特卡洛中，目前倾向于更加易于实现的剪枝
- 代码组分工
  - 一个人写 alpha-beta 剪枝
  - 两个人负责根据盘面生成数据
- 如何根据局面生成一个数字
- 可以先把 player 类分成两个大部分
  - 第一个大部分是一个决策树，决策树传给第二部分的是各种盘面
  - 第二个大部分是生成盘面的参数
  - 每个人搞一些小参数，小参数按权重合成总参数传回去
- 原始版的特征变量
  - 被吃掉的风险、吃别人的可能性
  - 平滑性
  - 单调性
  - 空格数（也可以运用到对面的盘面上）
  - 能不能阻碍对方下棋
  - 启发式规则*
  - 判断在对方盘面落子看是否会被合并
  - 整数的取值为 1~？尽量不选择 1，1 是最小值

微信群讨论截图：

4.2 经验与教训

由于无法线下进行讨论的，线上的交流存在一定的不便，在任务分配和推进中效率不太高。且由于线上交流的局限性，测试只能在在线下个人的 pc 端完成并汇总情况，这也带来了后期调试有所脱节，将调试的很多过程留给了王宇航和钱渠成（深表歉意）。

在刚开始时提出了很多新颖的想法，但在后期实践过程中发现难度比想象中要高很多。并且大家发现理论预期中的效果可能现实存在一定的差距，因此后期的调节中也进行了很多反直觉的调整，取得了比原来更好的效果。

4.3 建议与设想

4.3.1 建议

- 适度将实习大作业的时间提前，与通选课冲突后将会造成课业压力的增大
- 或许可以除了课程大群以外开一个专门用来答疑的不水的群，目前想要得到有用的信息不得不群中翻几十上百条消息，造成了效率的损失

4.3.2 寄语

- 想对明年选修课程的学弟学妹们说，尽管数算是一门比较艰涩的算法课程，但陈老师的 mooc 对于萌新超友好，各系列作业虽然有一定的难度但老师也会耐心地解答（甚至直接贴出代码）。从数据结构与算法的学习中得到的不仅是编程技术和对数据结构的熟练运用，跟重要的是这些从实际生活中抽象出来事物的特征并加以分析利用的思维。希望学弟学妹们能在各类作业和小组合作中收获难忘的回忆。

  （ps. 没有学习过 python 的同学建议在选择课程之前的假期内自学陈老师的 python 语言基础的 mooc，更容易适应学习的节奏）

5 致谢

感谢陈老师和助教们精彩的创意和工作热情，感谢小黑的友情出演（
感谢小组的所有成员，特别鸣谢王宇航同学和钱渠成同学在后期调试中的付出。

6 参考文献

https://blog.csdn.net/kimylrong/article/details/21967869
https://blog.codinglabs.org/articles/2048-ai-analysis.html

数据结构与算法课程实习作业报告

包涛尼　陈炜轩　董益铭 *　王子铭　余疏桐

F-19 Echo 组

摘　　要

本代码涉及的主要算法为 Minimax 算法，并进行 α-β 剪枝，涉及的主要数据结构为树结构。

落子的决策情况如下：通常情况下在我方落子；如果出现对方棋子经移动将吞并我方棋子的情况，我方会在合并路径上落子保护我方棋子；如果我方空格数小于等于 3，会考虑在对方棋盘落子。

移动的决策情况如下：调用决策树进行模拟，采用评估函数对局面评估。评估函数考虑的因素有棋盘剩余空格数、棋盘棋子面值的总和及棋子合并带来的收益。

在编写过程的实验测试中，我们主要进行时间测试，测试各函数耗时从而确定改进方向；对 AI 的强度测试，采取的方式有在本地与原版本对战以及与天梯榜的前列代码对战，统计对局情况从而确定版本强度。

总体而言，本 AI 属于和平型 AI，擅长合并我方棋子，吞并对方棋子，不会刻意侵占对方领地，压迫对方边境。

关键词： 决策树，Minimax 算法，α-β 剪枝

目录

1　算法思想　2

1.1　总体思路 2

1.2　算法流程图 4

1.3　算法运行时间复杂度分析 4

2　程序代码说明　6

2.1　数据结构说明 6

2.2　函数说明 6

2.3　程序限制 11

3　实验结果　11

3.1　测试数据 11

3.2　结果分析 18

3.3　经典战局 19

4 实习过程总结 20
4.1 分工与合作 20
4.1.1 小组分工 20
4.1.2 工作流程 20
4.1.3 交流方式与组会记录 21
4.2 经验与教训 24
4.3 建议与设想 25

5 致谢 26

参考文献 26

1 算法思想

1.1 总体思路

Position（落子）步骤：

1. 如果我方最大棋子等级小于等于 5，默认在我方落子;
2. 运行 new_emergencyblock 函数，如果我方有 5 级以上的棋子将在移动步骤被吞并，或者对方两个等级大于等于 7 的棋子间有空位，则将棋子落于此处用于阻挡和干扰。否则仍落子在我方;
3. 如果我方空格数小于等于 3（包括无空格），将落子于敌方所有空位 block 值最大处，block 值由 block 函数计算。

Move（移动）步骤：

1. 由 Decision 函数构建决策树并进行剪枝，模拟接下来的对局，层数 Depth = 11（这里一局有 4 步，每一步算一层）;
2. 叶节点估值 Score 由 numberscore，stealscore 和 blankscore 相加构成，具体规则见“程序代码说明”部分;
3. 在构建决策树时，需要模拟双方落子和移动。移动四个方向均模拟并生枝。然而落子步骤可能性太多，进行人工剪枝：我方默认下在自己棋盘。对方落子时，运行 emergencyblock 函数，它只判断我方是否有 5 级以上的棋子将在移动步骤被吞并，如果有，则假设对方也可能会落子阻挡，这样多生一根树枝，加上对方也可能在己方落子，于是共两根树枝。如果没有则只考虑对方在自己区域落子。然而由于树内将多次调用此函数，时间限制下此函数又做了一些修改：a. 只能判断水平方向是否将被吞并。b. 如果出现多生枝情况，多生的树枝也不再继续向下搜索，直接作为叶节点结算，并额外返回估值 E_2 = -300（因为对方成功阻挡）;
4. 递归结束条件除了达到层数外，还有终局判断函数：如果双方都无法移动，则进入终局。若我方胜利，则直接估值 M = 1000000，反之估值-M。

时间管理与战略优化：

由 output 函数实现，只在 Move 步骤进行调用（时间测试 Position 步骤相对几乎不耗时间）基本思路如下：

1. 如果时间小于 t，则降低搜索层数 2 层以防止超时;
2. 如果当前轮数小于 N，则降低搜索层数 2 层，因为前期搜索过深没有太大意义，而且移动方向多，耗费算力严重;
3. 均匀分配深度搜索。如果搜索树是稳定的（换言之，对方和我方的实际落子与决策树的模拟相同），那么可以一定程度重复利用树。比如上一轮的移动步骤搜索到 n+4 层选出来最好结果，本轮只需要搜索 n 层就可以得到上轮的结果。部分深搜部分浅搜是层数无法深入下去时候的无奈之举。至于如何分配深搜，实际测试中均匀分配深搜确实比集中深搜要有效;
4. 激进型与保守型。试想我们的搜索层数为 4n+1，这样我们叶节点对应的操作将是我方移动，这样没有考虑下一步的对方反噬，可能会造成得分偏高，相应的会积极进攻，为激进型；如果是 4n+2，那么最后一次操作是对方移动，这就偏向防守，为保守型。叶节点是落子步骤的也类似。然而，实际战斗中，可能是游戏机制的原因，发现同样的条件下，激进型比保守型更有可能获胜。于是在 11 层的基础上，均匀分配搜索层数为 13 和 9，最终效果也确实比 12 和 10、甚至 12 和 11 的效果要好;
5. 先后手问题。实践证明，在相同的条件下运行相同回合，后手比先手往往少花费 0.5-1s 的时间。这样时间管理就与先后手有关：先手 t = 1.2s，N = 80；后手 t = 0.7s，N = 60。这里采用较为粗略的 t 与 N 值主要是考虑到平台性能的波动，把时间足够精细的耗尽反而不稳定，于是采用了比较粗略和保守的估计，保证耗时都在 4.6s 左右。

算法实现及改进过程：

为了方便进行更改与整理，我们将不同系列的代码命名，每当有大的结构性改动和提高时，便会取一个新系列名字。这里借助系列名字来说明整体代码算法结构的改进（参数调整一直存在，具体见“实验数据”部分）。

Whisper：

最初期的版本，落子步骤计算落在对方所有位置的 block 值，并与落在我方相比较做出决策。移动步骤没有构建决策树，评估函数为初期的 contractionscore（对每次移动计算合并得分），boundscore（计算边界得失得分）和 weightscore（计算大子压制边界得分），仅仅考虑一层反噬。

Beep：

落子与移动均构建决策树（5 层），未进行 α-β 剪枝。由于 contractionscore 对每次移动计算，不是态函数，暂时未进树，而采用较为简单的 numberscore（直接估算我方棋子的面值并减去对方的）代替。增加了评估函数 blankscore（空格数得分），endscore（终局判断），monotonicityscore（单调性和光滑性得分）。

Bell：

进行 α-β 剪枝。对各个函数的运行时间进行了测试，删去了较为耗时的 weightscore 和 mono-

tonicityscore，层数增加至 6。

Symphony：

进一步优化各个函数性能。增加 emergencyblock 函数（只能计算水平方向），在树内对我方调用，如果成功 emergencyblock 则估值增加 E = 100。剪枝顺序调整。层数增加至 9。

Thunder：

contractionscore 进入树结构，由于每一个节点都要计算，导致层数降低至 8 层。

Music：

大胆尝试，删去了几乎所有的估值函数，仅保留 contractionscore（必要）和 blankscore（最快）。层数恢复到 9 层。

Niconiconi：

继续优化树结构，舍弃极为复杂的 contractionscore，重定义 numberscore 和 stealscore 作为代替。numberscore 采用了原版 2048 的评分机制（等级 n 的得分为 $(n-1)\times 2^n$）。stealscore 则计算双方棋子面值差，以此为基础评价吞并对方棋子的效果。引入时间管理。

Makimakima：

注意到落子步骤没有必要建树，仅仅考虑 emergencyblock 就可以达到几乎相同的效果，这样减少了一半的树，层数加深至 11 层。

Bibi：

将 emergencyblock 挪至树外后，落子步骤有充足的时间来模拟所有情况，升级为 new_ emergencyblock 函数。另一方面，树内 emergencyblock 也进行了调整，由对我方调用转为对方调用，原因有二：a. 由于 emergencyblock 评估的相当于是解围的情况，如果出现了成功阻挡评估值便会极大增加（Makimakima 版本甚至为了保证一定会选择解围将 E 设置为 10000）。然而则可能出现“骗保”的情况，即为了营造 emergencyblock 发挥作用的局面故意向不好的方向移动。无论如何，emergencyblock 只是几乎要被吞并时的无奈之举，不能有意让局面向这个方向发展。b. 对对方的调用是有必要的。比较智能的算法一定也会有类似 emergencyblock 的机制，实测中也确实出现了这样的局面：我方向某一个方向移动后，会对方被吞并 32，但在下一轮移动中可以吞并对方的 128。然而下一轮中对方使用 emergencyblock 阻挡我方合并。这说明考虑对方的 emergencyblock 是有意义的。由于树内多次调用，emergencyblock 只遍历了水平方向的所有函数。

Echo：

最终版本。Bibi 的时间性能已经几乎无法优化，结构也几乎达到了最简。仅有的改动是将深搜层数分散开来，达到了 13-9 的效果。另一方面参数也最终确定。

1.2 算法流程图

见下页。

1.3 算法运行时间复杂度分析

n 为一方空格数，m 为一方棋子数。

output 函数有两种模式，position 和 direction 模式。

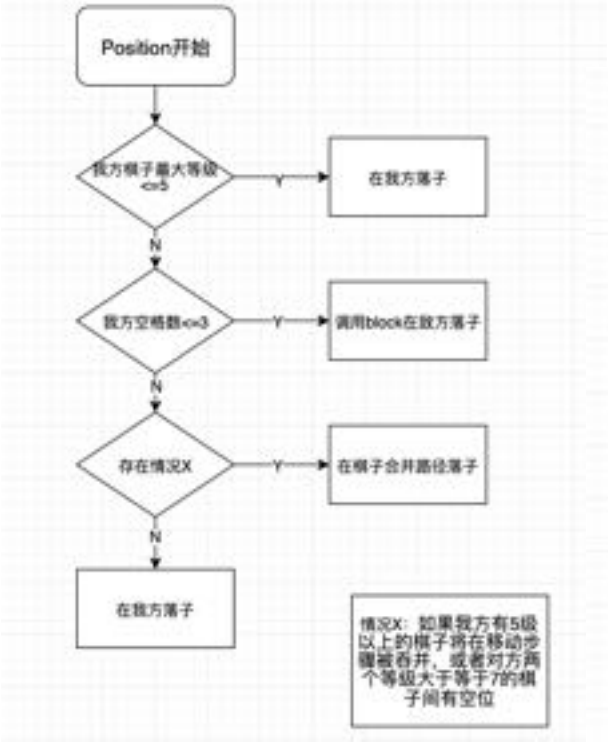


图 1: 落子流程图

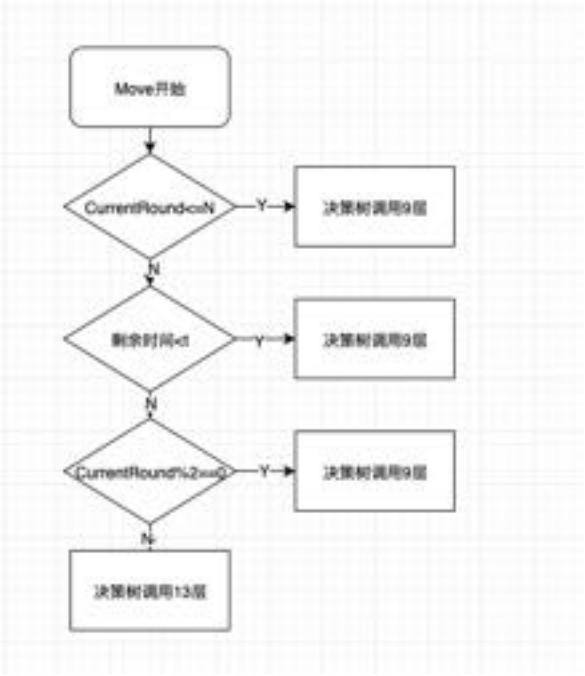


图 2: 移动流程图

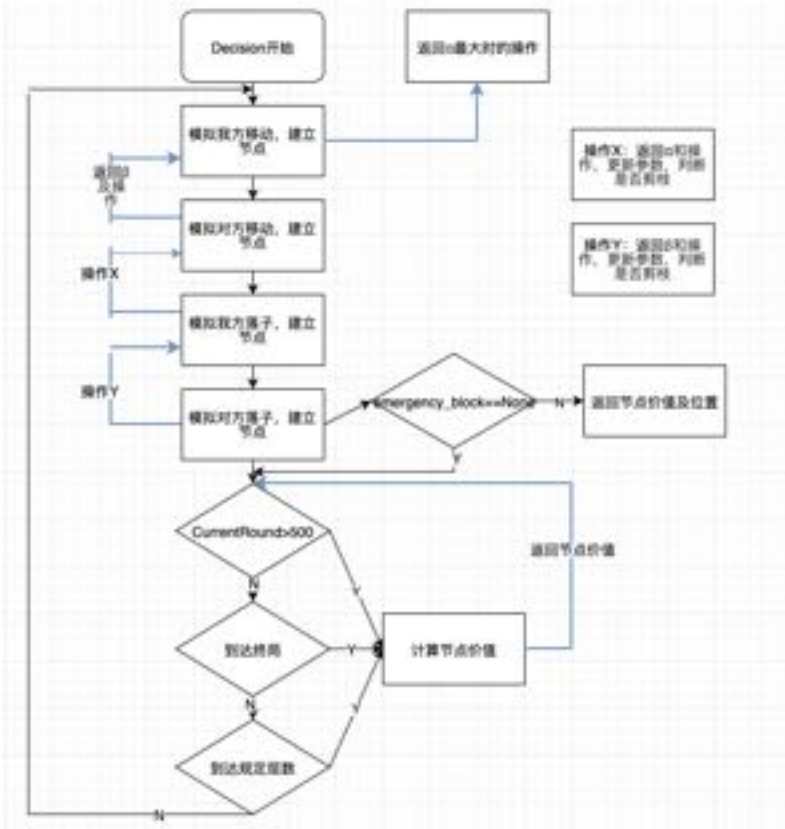


图 3: 决策树流程图（以先手为例）

position 模式：最坏情况是调用了 new_emergencyblock 函数，当中对棋盘某方所有空位进行了遍历检验，时间复杂度为 $O(n)$。

direction 模式：调用了 Decision 函数建立决策树，Decision 函数对进行 position 操作都只建立 1 个分支，对 direction 操作最多建立 4 个分支，最大深度是 11（第一层子节点为深度 1）。因此最坏情况为每次的 direction 操作都建立 4 个分支，最后对每个叶节点进行调用 Score 函数评估分数。其最大叶节点数为 4^6，每次 Score 函数的调用会遍历己方空格与两次双方棋子，因此时间复杂度应为 $O\left(4^6 \times (n+2m)\right)$。

上述两个模式中，direction 模式的时间复杂度明显要更复杂，因此 output 时间复杂度应为 $O\left(4^6 \times (n+2m)\right)$，简化后为 $O\left(n+2m\right)$。实际情况中应该较难遇到最坏情况，因此实际时间复杂度经过剪枝后应该会更低一些。经过多次测试，最终实战时并未出现超时的情况，因此时间复杂度还是让人满意的。

2 程序代码说明

2.1 数据结构说明

Node 类：

作为决策树的节点。每个节点携带的信息包括：

player：当前节点需要给出行动的玩家。以布尔值表示先后手。

mode：当前节点需要给出的行动为落子'position' 或移动'direction'。

depth：当前节点为决策树的第几层，以作为结束递归的条件之一。

lastAct：记录上一步进行了什么行动后导致当前节点的盘面。落子记录落子位置的元组，移动记录表示方向的整数。

alpha：记录 α-β 剪枝时的 α 值，表示我方最大收益。

beta：记录 α-β 剪枝时的 β 值，表示我方最小收益。

决策树：

与课程所学的搜索树相比，增加了在叶节点评估节点对应棋盘的分数并向上传递，以及进行 α-β 剪枝以减少分支的产生。具体实现方式为进行函数递归，并未实际储存一个树的数据结构。

栈：

endscore 中判断胜方的操作实际上用到了栈的线性结构。比较两栈栈顶元素的大小，相等则同时出栈，继续比较新的栈顶元素，不相等则可以判断胜负。一方变为空栈一方栈非空也可判断胜负。

2.2 函数说明

Decision 函数：

功能：

以递归的方式构架决策树，并最终返回本轮应进行的动作。

传入参数：

node：Node 类，表示当前节点，即根节点。

board：Chessboard 类，表示当前节点对应的盘面。

currentRound：整数，表示当前节点对应的回合数。

局域参数：

Depth：整数，表示决策树的最大递归深度。

E_2：整数，表示发现当前盘面会导致己方最大棋子被对方吞噬时，造成的损失。在盘面评估分数上扣除此损失以尽量避免选择会导致此等境地的决策。

递归结束条件包括：

已达到最大递归深度，回合数已超出 500，盘面拥有会被对方吞噬的己方棋子，盘面已无法落子或移动。

numberscore 函数：

功能：

体现棋子合并带来的收益。

传入参数：

board：Chessboard 类，表示当前节点对应的盘面。

局域参数：

k: 整数，为合并参数。

list0：[0, 0, 4, 16, 48, 128, 320, 768, 1792, 4096, 9126]

价值序列说明：当两个 2 合并时，分数贡献为一个 4 的分数减去两个 2 的分数，即为 4。当两个 4 合并时，分数贡献为一个 8 的分数减去两个 4 的分数，即为 8。即合成一个棋子，就得到该棋子大小的合并分数。

返回：

浮点数，表征棋子合并的收益。

实现：

将棋子通过价值序列映射为合并分数，返回我方棋子的合并分数总和 – 对方棋子合并分数总和。需要注意的是，真正体现合并分数的数据为末态棋盘的合并分数 – 初态棋盘的合并分数，然而考虑到棋盘的初态均为决策树的根节点棋盘状态，不同叶节点的末态合并分数足以体现合并带来的价值。

stealscore 函数：

功能：

体现吞并棋子带来的收益。

传入参数：

board：Chessboard 类，表示当前节点对应的盘面。

局域参数：

N：整数，为吞并参数。

list1：[0, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024]

价值序列说明：此序列将棋子的等级映射为棋子的大小。

返回：

整数，表征棋子吞并的收益。

实现：

将棋子等级通过价值序列映射为棋子面值，返回我方棋子的面值总和 − 对方棋子面值总和。同样需要注意的是，真正体现吞并分数的数据为末态棋盘的棋子的面值总和 − 初态棋盘的棋子的面值总和，然而考虑到棋盘的初态均为决策树的根节点棋盘状态，不同叶节点的末态合并分数足以体现吞并带来的价值。

blankscore 函数：

功能：

评估因己方棋盘上有空位而产生的分数，总体上剩余的空格数越多，我方操作的自由空间越大，产生的分数越大。

传入参数：

board：Chessboard 类，表示当前节点对应的盘面。

currentRound：整数，表示当前节点对应的回合数。

局域参数：

$k7$：整数，为空格数参数。

返回：

整数，表征因空格数产生的分数。

实现：

使用 Chessboard 类的 getNone() 方法，获取我方空位数。当前节点回合数 <250 时，$k7$=15；当前节点回合数 >=250 时，空位数对比赛的影响较为关键，$k7$=24。函数返回 $k7$ 与空位数的乘积作为空位数分数。

block 函数：

功能：

计算某位置的阻塞程度对应的值。

传入参数：

board：Chessboard 类，表示当前节点对应的盘面。

pos：元组，表示某一位置在棋盘上的坐标。

返回：

整数，表征下子的阻挡程度。

实现：

1. 阻挡两个大数 a, b 合并：当 $\min(a, b) \geq 32$，并且 a, b 间可以落子。aa 型：此时下在两个大数之间会阻挡合并，记阻挡度为 $x1 = a$；ab 型：此时下在两个大数之间会破坏对方阵型，记阻挡度为 $x2 = 0.8 \max(a, b)$。

2. 小子合并：下上 2 之后，垂直方向有 2 可以与它合并，这时阻挡程度相对差一些，记阻挡度贡献为 $x3 \cdot x1$ (或$x2$)，$x3 = 0.8$。
3. 棋盘无法移动：触发此项的判断条件为棋盘的空格数小于等于 3，这时下 2 可能会使棋盘卡死的风险增大。具体操作方式为，对 3 个格进行枚举，某个格子下 2 不能被立即消掉，就说此项贡献成立，在已经计算得到的 block 值额外增加 $x = 30/$当前空格数；如果三个格子下 2 都能被立刻消掉，此项无贡献。

emergencyblock 函数：

功能：

计算紧急在某方下子的位置，并且只在树内建立子节点时使用，因此功能不完善以节省计算量。

传入参数：

board：Chessboard 类，表示当前节点对应的盘面。

belong：可缺省参数，表征对哪一方的棋盘调用，缺省时默认为对手方。

返回：

元组。若达不到标准，返回空元组；否则返回位置。

实现：

内部定义有函数 get_chess(pos, direction)，direction 从 0-3 分别对应上、下、左、右，可以获取 pos 位置上下左右最近的“可触及棋子”。这里“可触及棋子”表示可以在'direction' 模式下通过移动棋盘触及的棋子，例如压在敌我边界上的棋子；而对方棋盘深处的棋子便不是“可触及棋子”。当某一方向有“可触及棋子”，返回该棋子的 (级别, 归属) 二元组；找不到“可触及棋子”时，返回空元组。此不完整的紧急堵截函数只会判断水平方向上是否有对面的大子可以吞噬我方的大子，其中这里“大子”被定义为级别 >=4 的棋子。若存在这种情况，函数返回对方棋盘上一个堵截位置，防止对方通过移动合并吞噬我方大子；否则返回空元组。

new_emergencyblock 函数：

功能：

计算紧急在某方下子的位置，并且直接在树外决策，功能完善。

传入参数：

board：Chessboard 类，表示当前节点对应的盘面。

belong：可缺省参数，表征对哪一方的棋盘调用，缺省时默认为对手方。

返回：

元组。若达不到标准，返回空元组；否则返回位置。

实现：

内部定义有函数 get_chess(pos, direction, dic)，direction 为'horizontal' 代表水平，'vertical' 代表竖直，可以获取 pos 位置水平或竖直最近的“可触及棋子”二元组。当某一方向有“可触及棋子”，返回上下/左右棋子属性的二元组，形式为 ((级别, 归属), (级别, 归属))；找不到“可触及棋子”时，返回 pos 位置处的棋子信息。此完整的紧急堵截函数会判断水平方向和竖直方向上

是否有对面的大子可以吞噬我方的大子以及对方在某一方向上是否有两个相同大子合并的情形，其中这里“大子”被定义为：若是对方吞噬我方棋子，级别 >=5；若是对方相同棋子合并，级别 >=7。同时为了加速计算速度和减少重复计算，每个位置被设定了附加属性 [bool, bool]，前者表示该位置是否拥有水平搜索历史，后者表示该位置是否拥有竖直搜索历史；若遍历时发现已有某方向上的搜索历史，则直接跳过该位置遍历下一个。若存在可以堵截的情况，函数返回对方棋盘上一个堵截位置，防止对方通过移动合并吞噬我方大子或自己合成更大的棋子；否则返回空元组。

output 函数：

功能：

调用其他函数搜寻并返回最佳操作。

传入参数：

currentRound：整数，表示当前节点对应的回合数。

board：Chessboard 类，表示当前节点对应的盘面。

mode：字符串，反映当前决策内容，若为’position’ 则应得出位置，若为’direction’ 则应得出合并方向。

返回：

mode 若为’position’ 则返回位置（元组），若为’direction’ 则返回合并方向（整数）。

实现：

若应输出位置，调用 new_emergencyblock 函数判断是否符合紧急在对方下子的条件，判断己方是否适合下子（己方空格数小于 3），若不适合下子则寻找对对方下子阻碍最大处。若应输出移动方向，使用决策树搜索。输出移动方向时因为建立决策树耗时较多而加入了时间控制模块，通过判断先后手信息、轮数和剩余时间调整初始情况的根节点层数来调整搜索深度，在对局前期压低算力，中期扩大算力，时间不足时收缩算力。

isEnd 函数：

功能：

判断当前局面是否为终局。

传入参数：

board：Chessboard 类，表示当前节点对应的盘面。

返回：

bool 值：True 表示到达了终局，False 表示未到达终局。

实现：

到达终局仅仅与双方是否均不能移动有关，与下子无关。判断终局的函数使用了 move 函数返回的 bool 值，利用 and 实现仅当 move 函数在双方四个方向移动均返回 False 时返回 True。

endscore 函数：

功能：

计算由于到达终局产生的额外分数。

传入参数：

board：Chessboard 类，表示当前节点对应的盘面。

局域参数：

M：大整数，加快到达胜局，避免到达败局。

返回：

整数，表示终局分数。

实现：

比较 getScore 函数返回的两方棋子级别列表，自大至小依次对比直至出现某一方棋子大于另一方的情况，则判断棋子更大的一方为胜者，若某一方棋子列表先被全部遍历而另一方仍有剩余则有剩余棋子方为胜方。若己方为胜方则返回 M（大数），为负方则返回-M，平手返回 0。

2.3 程序限制

在一些极端情形或面对某些对手时，存在超时的可能性。

3 实验结果

3.1 测试数据

实验环境说明：

硬件配置：

CPU：1.8 GHz Intel Core i5

内存：8 GB 1600 MHz DDR3

操作系统：

macOS Mojave 版本 10.14.6

Python 版本：3.7

我们对算法的测试集中在两个方面，一是算法的强度测试，二是算法的时间测试。算法的强度测试的目的是为了测试算法更新后的效果以及确定调参的方向；算法的时间测试的目的是为了测试各函数的用时，从而决定决策树的层数和算法优化的方向，以及便于后期进行算法的时间管理。根据我们的经验，优化算法结构，增大决策树的搜索层数以及良好的时间管理带来的收益是远大于精细调参的，所以测试的重心放在了时间测试上。

算法的强度测试：

前期我们测试版本强度的方式是使原版本与新版本在本地对战，根据不同版本之间的对局结果来评估算法的强度。

但在后期，根据版本更新积累的经验，我们发现新版本仅在本地与原版本对战不能很好地体现新版本的实力。也确实出现了虽然新版本可以大比分压制原版本，但是在天梯中实战效果

却不如原版本的情况，所以在后期，我们不仅在本地测试新版本与原版本对战情况，还会对天梯榜的前几名发起挑战，根据与对手的对局情况，判断新版本的实力。

图 4: 新版本与原版本的对战情况

makimakima 与 Echo 对战的平均情况为第 9 栏，Echo 每 10 局胜 5.75 局。

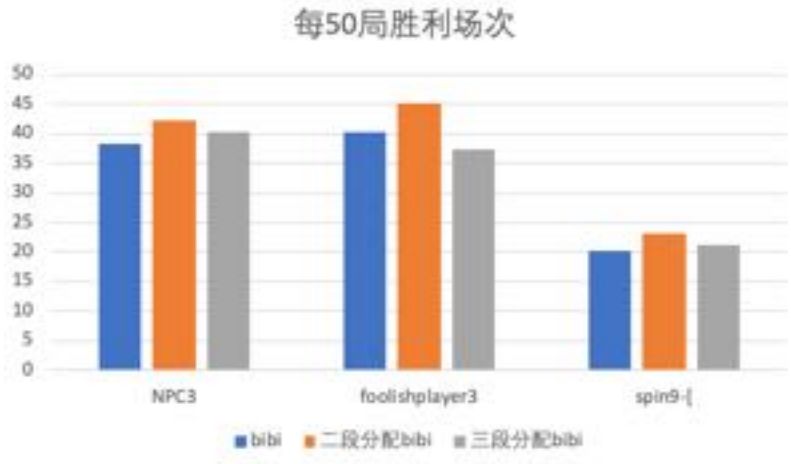


图 5: 不同版本对战天梯前几名以及隐藏高手的对局结果

bibi：即上述提到的版本。

二段分配 bibi：代码改进后期，我们时间管理尝试使用深搜加浅搜交替的方法，bibi 原版在时间足够的情况下，决策树的层数不变；二段分配 bibi 在时间足够的情况下，会一次深搜一次浅搜交替进行，三段分配 bibi 在时间足够的情况下，会一次超深搜，一次深搜，一次浅搜交替。根据与天梯榜前列 AI 以及 spin9-[的对战结果，决定了 Echo 的雏形为二段分配 bibi。

算法的调参说明：

本 AI 涉及的主要参数如下：

1. numberscore（合并评估函数）参数

价值序列：list0 = [0, 0, 4, 16, 48, 128, 320, 768, 1792, 4096, 9126]

合并系数：$k = 0.3$

价值序列确定：源自原版 2048 的评分体系，实现合成某个棋子，就得到该棋子面值的分数。

合并系数确定：系数确定的主要方法在于观摩对局。实战中发现，如果不加此合并系数，即 $k = 1$，则会出现 AI 合并优先级过高的问题。举个例子，在实战中发现，如果 $k = 1$，AI 会倾向于合并两个 128，而忽视 16 被对方吞并，这种处理方式显然不合理，从而我们决定降低合并分数的权重。

2. stealscore（吞并评估函数）参数

价值序列：list1 = [0, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024]

吞并系数：$k = 20$

价值序列的确定：即棋子对应面值的大小。

吞并系数的确定：吞并系数与合并系数是联动确定的，原因同上，意在加大吞并分数的权重。

3. blankscore（空格数评估函数）参数

250 轮之前 $k = 15$。

250 轮之后 $k = 24$。

之所以前后不同，原因有二。我们是先确定的合并函数以及吞并函数的评分体系才决定的空格数函数的评分体系，以上两个函数的评分体系，会在后期分数膨胀，所以势必加大参数与膨胀的分数体系适配；其次，后期更容易被塞满无法移动，所以空格数的价值更大，考虑两方面因素，增大后期系数。

4. 各函数系数的确定

根据我们的经验，结构性优化带来的收益远远大于调参带来的收益。前期我们进行过调参测试，发现精细调参之后，AI 的强度几乎不变或强度提升不大，远不如版本更替带来的强度提升大。所以后期的参数没有经过细致的调节，比如，我们不纠结合并系数是 $k = 0.3$ 或 $k = 0.31$ 哪个效果更好，我们只关注 $k = 0.3$ 左右与总体的分数评估体系更适配，不纠结具体的精细调参。

上述系数的说明：添加系数是为了整体的评分体系的合理，以及评分体系与我们的想法相一致。

我们希望吞并以及防止被吞并是第一优先级，因为吞并的机会转瞬既逝，被吞并造成的损失不可挽回。虽然比赛最终的对局是看最大棋子的大小，貌似是合并更重要一些，但是根据经验，合并的机会不像吞并的机会那么重要，这一步放弃了合并，之后还会有很多合并的机会，所以我们最终决定，把吞并与防被吞并设为最高优先级。经过计算，赋值吞并 $k = 20$，合并 $k = 0.3$，（实际两个参数可以合并为一个，但由于历史因素，最终决定设置两个）。达成的效果是，吞并 16 以及更大的棋子，分值达到 320 及以上，最高优先级；吞并 8 棋子，与合并 512 的重要性基本持平；吞并 4 棋子，与合成 256 棋子基本持平；吞并 2 棋子，与合成 128 棋子基本持平。虽然赋分体系看起来不太合理，但是实际不然，因为吞并的机会少，合并的机会多，虽然这一步我们放弃了合并 128 去吞并一个 4，但是接下来的几步，又会有 128 可以合并的机会，这个时候 128 再进行合并，实现吞并、合并两不误。

空格数系数：进一步引导合并，防止无法移动。前期一个空格数的价值，接近合并出 64 的价值，后期一个空格数的价值，略低于合并出 128 的价值。这样设置，引导 AI 移动时尽可能一次移动带来多个合并，产生更多空格数，增强 AI 的灵活性，防止出现无法移动的状况。

算法的时间测试：

前期的算法时间测试是在本地进行的，利用 time 函数库对函数的用时进行测试。（代码如下，为使时间测试程序更加清晰，删去了内部 board 和函数结构体的实现。）

```
import time
def fakerun(x): #计算除了执行真正函数的其他所有操作的时间
    t = time.time()
    for _ in range(x):
        import random

        class Board:
        #具体实现略去
        def weightscore(board):
        #具体实现略去
        board = Board()
    return time.time()-t

def realrun(x): #计算所有操作总时间
    t = time.time()
    for _ in range(x):
        import random

        class Board:
        #具体实现略去
        def weightscore(board):
        #具体实现略去
        board = Board()
        weightscore(board)
    return time.time()-t
for __ in range(10000,110000,10000):
    print("运行次数：",__,'运行时间：',realrun(__)-fakerun(__))
#二者直接相减，这里不选择“在程序内部待测函数运行前后计时相减”的办法是为了减少time
                                        调用次数，减少误差
```

测试结果如下图 6 所示。

weightscore 作用：如果大子在边界上，那么可以对应一定分数。设计该函数的目的是，引导大子往边界上移动，压迫对方边境，增强 AI 的进攻性。

根据测试结果我们发现，运行一次 weightscore 函数用时 8.6×10^{-5}s，相对其他评估函数而言很慢，加入 weightscore 之后，我们的决策树搜索层数无法超过 6 层，最终舍弃了这个函数。

后期天梯安装了加速库，我们除了在本地进行时间测试外，也会在天梯上进行时间测试验证本地时间测试的效果，防止电脑因素带来的时间测量偏差。

```
运行次数:  10000 运行时间:  0.8606247901916504
运行次数:  20000 运行时间:  1.8906762599945068
运行次数:  30000 运行时间:  2.7495453357696533
运行次数:  40000 运行时间:  2.8042171001434326
运行次数:  50000 运行时间:  4.5612733364105225
运行次数:  60000 运行时间:  5.946057319641113
运行次数:  70000 运行时间:  6.733816862106323
运行次数:  80000 运行时间:  7.569849967956543
运行次数:  90000 运行时间:  7.690307140350342
运行次数:  100000 运行时间:  9.366135120391846
```

图 6: 测试结果

天梯测试的具体方法为，在白痴 AI 中加入 test 函数，test 函数会反复运行某个待测函数，根据白痴 AI 超时时的轮数，可以判读出待测函数在规定的时间内运行了多少次，从而估计函数的用时。

具体代码如下：

```
class Player:
    def __init__(self, isFirst, array):
        # 初始化
        self.isFirst = isFirst
        self.array = array

    def monotonicityscore(self, board, belong):
        #函数具体实现略去

    def test(self,currentRound,board):
        for i in range(1000):
            self.monotonicityscore(board,True)

    def output(self, currentRound, board, mode):
        if mode == 'position':  # 给出己方下棋的位置
            if currentRound>=40:
                self.test(currentRound,board)
            another = board.getNext(self.isFirst, currentRound)  # 己方的允许落子点
            if another != (): return another

            available = board.getNone(not self.isFirst)  # 对方的允许落子点
            if not available:   # 整个棋盘已满
                return None
            else:
                from random import choice
                return choice(available)
            elif mode == 'direction':  # 给出己方合并的方向
                from random import shuffle
                directionList = [0, 1, 2, 3]
                shuffle(directionList)
                for direction in directionList:
```

```
                if board.move(self.isFirst, direction): return direction
        else:
            return
```

代码说明：当轮数 >=40 时，每次落子之前会调用 test 函数，test 函数会调用 1000 次 monotonicityscore 函数，根据白痴 AI 在第几轮超时，可以获取在 5s 内函数可以运行多少次，从而获取函数在天梯上的运行时间。

测试结果：

测试一例：从 50 轮开始，每次调用 103 次。可运行 75-100 轮，记 7.5-10 $\times 10^4$ 次。多次平均之后，获取每次运行占用时间 5×10^{-5}-1.1×10^{-4}s。

编号	先手方	比赛长度	胜者	胜利原因	补充说明	比赛过程
#1	test25	461	大总包 (接收方, 后手)	超时	player 0 time out	查看
#2	大总包	442	大总包 (接收方, 先手)	超时	player 1 time out	查看
#3	test25	427	大总包 (接收方, 后手)	超时	player 0 time out	查看
#4	大总包	438	大总包 (接收方, 先手)	超时	player 1 time out	查看
#5	test25	430	大总包 (接收方, 后手)	超时	player 0 time out	查看
#6	大总包	471	大总包 (接收方, 先手)	超时	player 1 time out	查看
#7	test25	502	大总包 (接收方, 后手)	超时	player 0 time out	查看
#8	大总包	481	大总包 (接收方, 先手)	超时	player 1 time out	查看
#9	test25	448	大总包 (接收方, 后手)	超时	player 0 time out	查看
#10	大总包	394	大总包 (接收方, 先手)	超时	player 1 time out	查看

图 7: 测试一例

最后，经过两个方法的测试，得出主要函数的时间测试图：

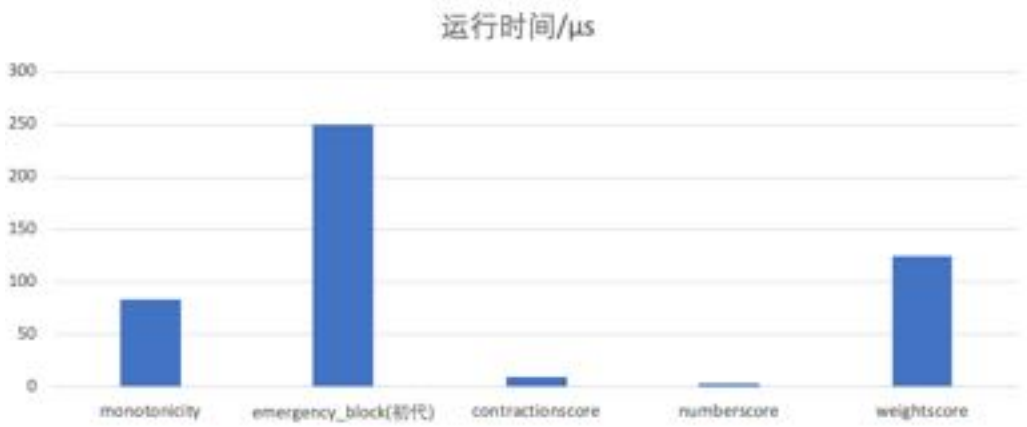


图 8: 主要函数运行时间

根据测试，我们发现单调性函数 monotonicity 以及控制大子往边界移动的 weight 函数耗时巨大，最终版本把这两个函数都舍弃了。

emergency_block（落子函数）耗时也很大，后期改进的一个大方向是优化落子函数。

最后，我们对决策树的调用深度进行优化，需要测试决策树深入不同层数的时间，方法同

测试函数函数用时，即在白痴 AI 中反复建树，统计超时的轮数。

战绩：

热身赛战绩：

最高得分曾突破过 2000 分，后来发现，是天梯莫名其妙给我们加了 500 分……

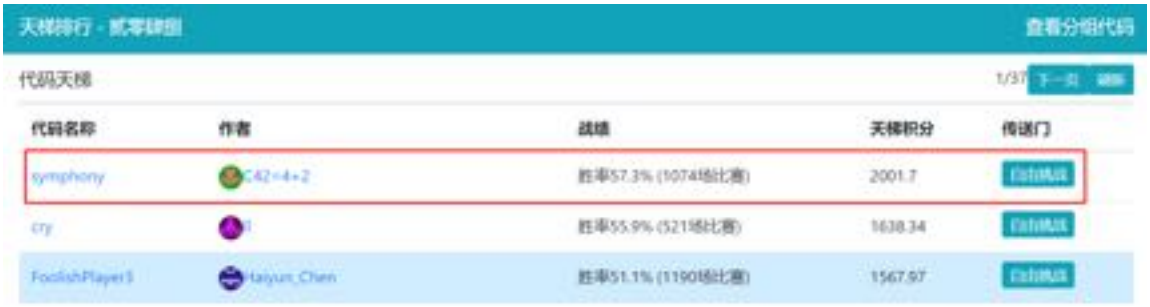


图 9: 天梯（因为 bug）突破 2000 分

回到正题。

在热身赛最后阶段，我们并没有集中地对天梯榜上代码发起过挑战，所以用天梯赛的胜率代替热身赛的结果。

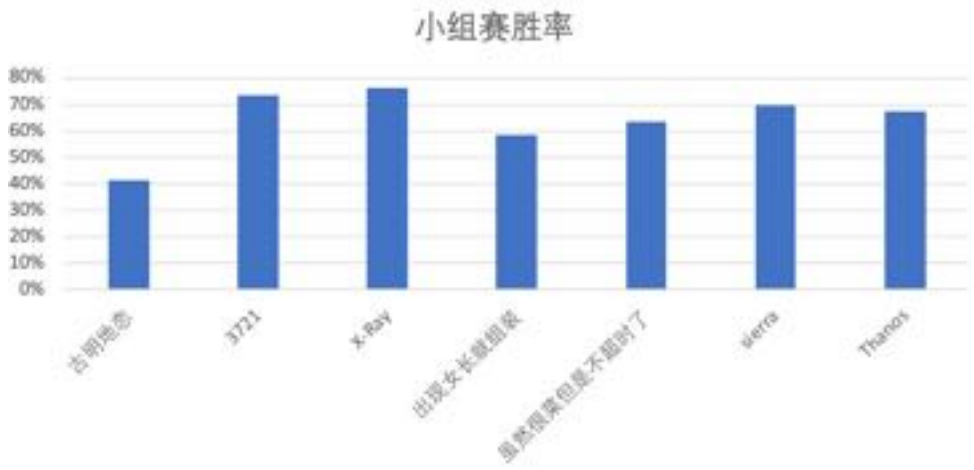


图 10: 小组赛胜率统计

最后附加一张我们组在天梯赛上到达最高分的截图纪念。

图 11: 天梯赛最高分纪念

3.2 结果分析

策略 1：emgerency_block：

此函数是我们组落子函数的核心部分，它的作用为，当出现移动无法改变我方被吞并棋子的情况时，紧急在我方棋子和敌方棋子中间落子，保护我方棋子。

比如以下局面，我方为先手，可以看到，第 3 排我方的 256 无法吞并对方的 256，但是对方的 256 可以吞并我方的 256，并且移动无法改变我方 256 被吞并的结局。在引入 emgerency_block 之前，我方的 256 就会被对方无情吞并，令我们非常心痛，于是加急写出 emergency_block 函数，在两个 256 中间落子，保护我方 256，解决了该问题。

图 12: 引入 emgerency_block 的契机

策略 2：深搜浅搜交替进行：

此改动是我们组最后一次结构性的改动。在最后的 bibi 版本中，凡是耗时长的函数都被砍掉了，在不改变根本性的决策树算法的基础上，层数优化已经到达极限。这时我们采取了深搜浅搜交替进行的一种策略。我们期望，在深搜的过程中，尽可能看得更远，找到到达最优解的路径，走上正确的分支，之后浅搜，在正确的分支上继续往前走。虽然没有做过严格的理论证明，但在实战过程中，确实比均匀搜索效果略好。

策略 3：放弃 AI 的攻击性，换取更深的层次与灵活性：

在大作业的开始阶段，我方 AI 也是具有比较强的进攻性，倾向于侵占对方棋盘，倾向于把大子放在边界，形成压迫之势。不过在这样的评估函数耗时较多，导致我们的决策树层数停留在 8 步左右。

最后经过考虑，我们把 boundscore（边界函数，即侵占对方的格子数越多，得分越高）以及 weightscore（倾向于把大的棋子放在边界）两个耗时较多的函数删掉，放弃了 AI 的攻击性。

放弃 AI 的攻击性带来的好处主要有两点，一点为我方 AI 的决策树层数深下去了，与对方 AI 进行吞并博弈时，看得相对长远，因此往往占据优势。第二点为我方 AI 的灵活性很强，以先手为例，如果 AI 倾向于压迫边境，那么 AI 基本不会向左移动，也就是回撤，那么它就仅有 3 个方向可供选择。而我们的 AI 放弃了攻击性，允许回撤，也就是有 4 个方向可供选择，灵活性得到不小的加强。

在实战中，如果对方不是进攻型 AI，那么我方的 AI 就会有很好的发挥；如果对方的进攻性不够强，没有办法很快把我方压迫到无法移动，我方的吞并能力以及合并能力往往比对方要强，

那么我方的 AI 基本可以击败对方；如果对方的 AI 进攻性非常强，那么我方就会很快被压迫到无法移动，从而负于对方。一言以概，我们牺牲了攻击性而换取灵活性。这导致我们面对进攻性非常强 AI（如 Uniform 组）时便会受到较严重的克制。

时间开销：

在前期，经过时间测试发现，我们的时间开销主要在 monotonicityscore 函数，boundscore 函数，weightscore 函数上，emergency_block（初代）上。

在后期，我们放弃攻击性，把上述耗时较长的评估函数全部删掉，同时改进了 emergency_block，从而我方评估函数耗时很短，时间开销主要在决策树上。

由于决策树占据时间开销较大，我们对决策树进行了多次改进。早期我们使用 contraction 函数计算合并得分，每个树节点都要维护棋子情况，耗时较多；后期我们改用 numberscore 函数，仅考虑末态即可完成评估。除 α-β 剪枝外，我们也考虑了一些人工剪枝的方法，比如 emergency_block 函数几乎剪掉了所有在对方棋盘落子的情况，比如后期在建立决策树时只有 move 过程生枝，position 过程不生枝，这些方案起到了一定的优化效果。经过我们的不懈努力，最终决策树探索深度不断加大，最后到达了平均 11 层（1 层指一次操作）的效果。

3.3 经典战局

心痛瞬间：

图 13: 我方为先手，注意第二排的两个 512 的位置

图 14: 我方为先手，注意第二排的两个 256 的位置

在我方引入 emergency_block 之前，我方不会在中间落子阻止对方 512 吞并我方的 512，经常出现被吞并 512、256 的情况，令我们非常心痛。

加入 emergency_block 之后，才基本解决此问题。

超时大战：

我方 AI 存在使对方 AI 更容易超时的特性，曾经在与官方 SSS_bot、SS_bot 作战时，把对方多次打成超时。

4 实习过程总结

4.1 分工与合作

4.1.1 小组分工

总的来讲，董益铭和王子铭负责算法，而包涛尼，陈炜轩，余疏桐负责具体代码。具体分工如下：

董益铭：组长，负责组织小组活动，分配任务，制定总体计划，监督督促组员。同时进行各个评估函数的参数调整与各个版本代码的性能测试。

王子铭：负责算法流程和评估函数，并且进行代码结构、估值函数的优化和时间管理。同时负责维护小组资料，进行最新消息更新和通知。

陈炜轩：主要负责决策树的构建。进行本地调试，保证历代版本的运行稳定性。同时负责少量评估函数的编写。

包涛尼：编写大量决策函数，实现接口，进行决策优化。同时负责后期代码的各版本修改与性能测试。

余疏桐：编写大量估值函数，进行估值优化并整合所有代码。同时在后期对树结构进行改造与提高，并进行各个版本的性能测试。

最终实习报告以董益铭和王子铭为主的同时由大家共同撰写，并由包涛尼负责排版为 LaTeX 版。

4.1.2 工作流程

初期阶段（5 月 14 日—5 月 25 日）：进行基本代码的编写与整合，形成早期版本便于进行测试。

算法与思路：王子铭、董益铭。王子铭负责构建评估函数和基本流程。董益铭负责进一步细化为具体算法与编写思路。

代码编写：陈炜轩、余疏桐、包涛尼。余疏桐和陈炜轩负责移动步骤决策树的构建与 α-β 剪枝，包涛尼负责落子步骤评估与决策函数的编写。决策树内评估函数由三人内部分配任务后共同编写。为了方便检验，三人均编写了随机测试函数来保证评估函数的正确性。

测试与提高阶段（5 月 26 日—5 月 31 日）：形成各个系列代码，同时不断测试提高。

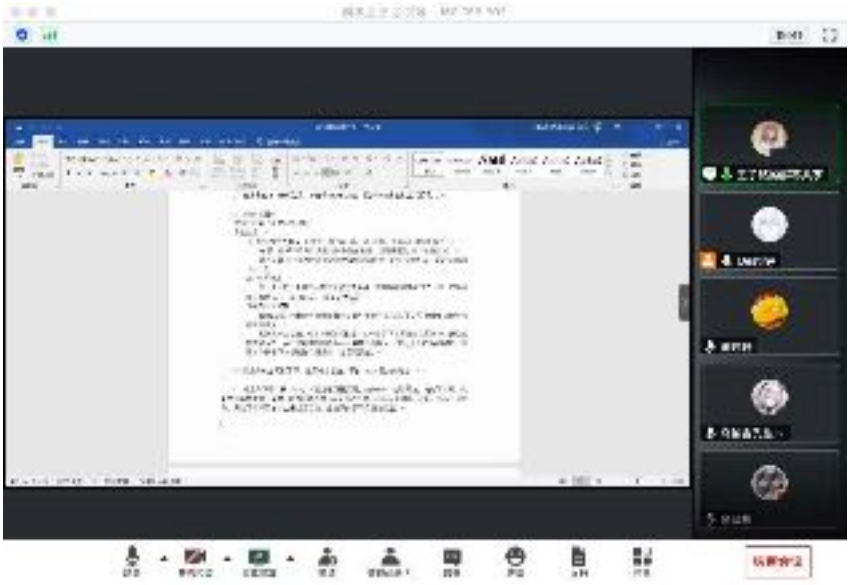
性能优化：董益铭。编写时间测试函数，进行代码的时间复杂度测试，并且根据实际运行时间进行估值函数的选择与优化。

实战测试提高：董益铭、王子铭、包涛尼、余疏桐。董益铭等将各个版本在本地和热身天梯上运行，观察实战效果并统计。王子铭根据实战情况，在时间约束下进行各种可能的尝试与提高，经讨论后由包涛尼与余疏桐实现，继续进行测试。

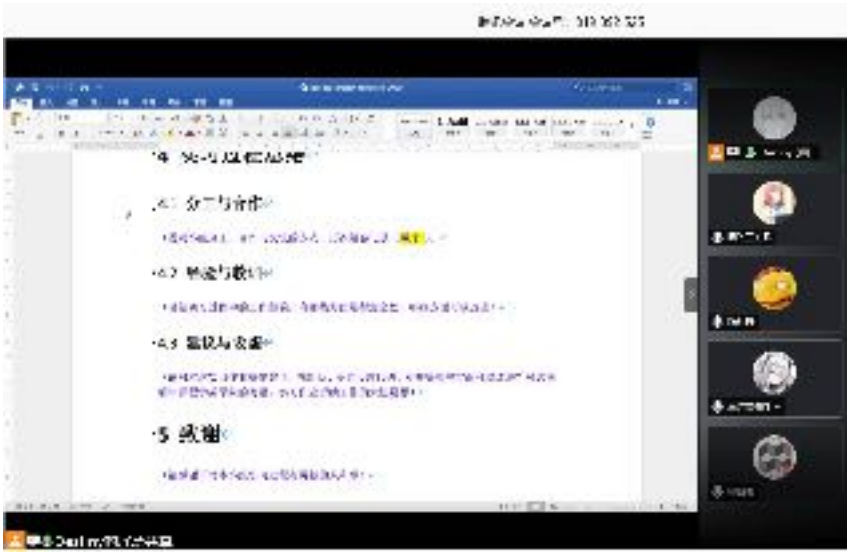
整体稳定性：陈炜轩。陈炜轩负责关注各种新的更新，进行各种可能的调整与稳定性测试。

4.1.3 交流方式与组会记录

交流主要在腾讯会议和微信进行。有大的讨论与任务分配时采用腾讯会议，平时各种交流和建议在微信群里进行。部分截图如下：



(a) 讨论初期算法

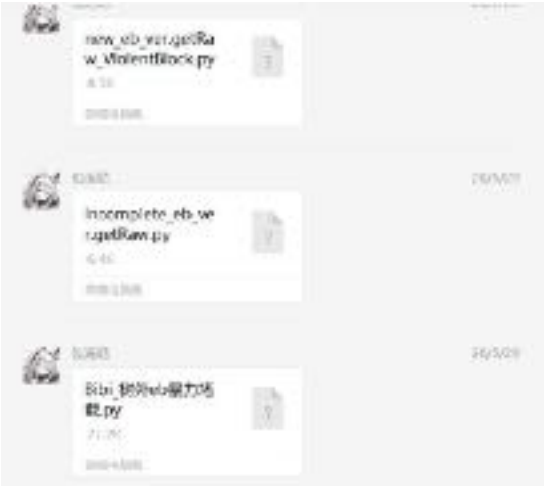


(b) 讨论报告分工

图 15: 小组会议截图

(a) 截图 1

(b) 截图 2



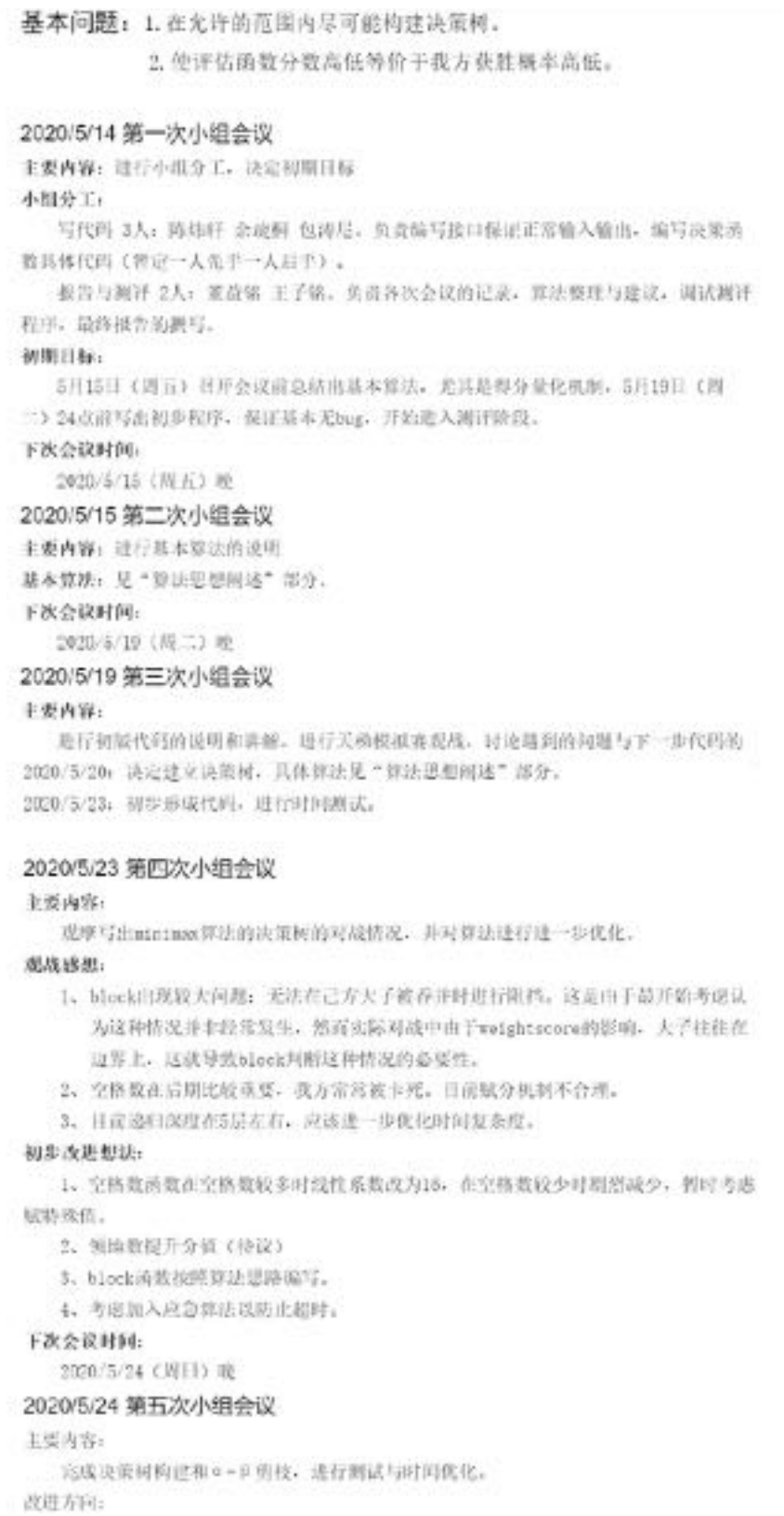
(c) 截图 3

(d) 截图 4

图 16: 微信交流截图

各种会议记录和杂项都记录在腾讯文档中，代码保存在北大网盘中，由王子铭维护，大家共同修改。

腾讯文档：



基本问题：1. 在允许的范围内尽可能构建决策树。

2. 使评估函数分数高低等价于我方获胜概率高低。

2020/5/14 第一次小组会议

主要内容：进行小组分工，决定初期目标

小组分工：

写代码 3人：陈炜轩 余晓桐 包涛尼。负责编写接口保证正常输入输出，编写决策函数具体代码（暂定一人先手一人后手）。

报告与测评 2人：董益铭 王子铭。负责各次会议的记录，算法整理与建议，调试测评程序，最终报告的撰写。

初期目标：

5月15日（周五）召开会议前总结出基本算法，尤其是得分量化机制，5月19日（周二）24点前写出初步程序，保证基本无bug，开始进入测评阶段。

下次会议时间：

2020/5/15（周五）晚

2020/5/15 第二次小组会议

主要内容：进行基本算法的说明

基本算法：见“算法思想阐述”部分。

下次会议时间：

2020/5/19（周二）晚

2020/5/19 第三次小组会议

主要内容：

进行初版代码的说明和讲解。进行人机模拟对战对战，讨论遇到的问题与下一步代码的

2020/5/20：决定建立决策树，具体算法见“算法思想阐述”部分。

2020/5/23：初步形成代码，进行时间测试。

2020/5/23 第四次小组会议

主要内容：

观摩写出minimax算法的决策树的对战情况，并对算法进行进一步优化。

对战感想：

1、block出现较大问题：无法在己方大子被吞掉时进行阻挡。这是由于最开始考虑认为这种情况非常经常发生，然而实际对战中由于weightscore的影响，大子往往在边界上，这就导致block判断这种情况的必要性。

2、空格数在后期比较重要，我方常常被卡死，目前赋分机制不合理。

3、目前递归深度在5层左右，应该进一步优化时间复杂度。

初步改进想法：

1、空格数函数在空格数较多时线性系数改为18，在空格数较少时indexing减少，暂时考虑赋特殊值。

2、领域数提升分值（待议）

3、block函数按照算法思路编写。

4、考虑加入应急算法以防止超时。

下次会议时间：

2020/5/24（周日）晚

2020/5/24 第五次小组会议

主要内容：

完成决策树构建和α-β剪枝，进行测试与时间优化。

改进方向：

图 17: 小组会议记录

北大网盘：

图 18: 主要目录

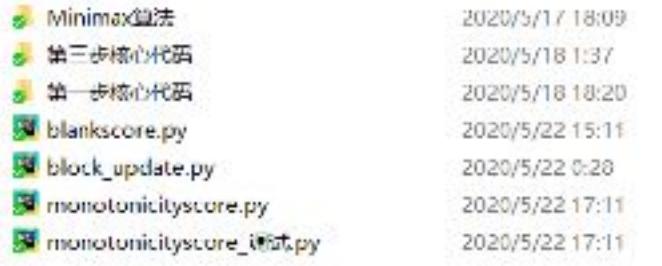


图 19: 核心代码

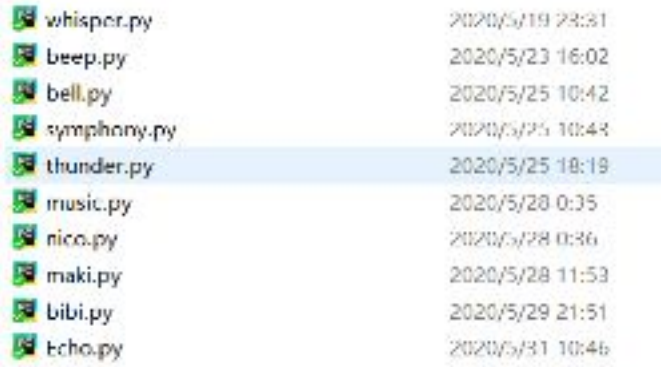


图 20: 各种版本

4.2 经验与教训

经验：

1. 足够的交流。在整个实习过程中，小组成员进行了充分和有效的交流。除了每周召开两次左右小组会议外，大家常常在微信群内交流各种想法，并且积极参与到整个实习流程的各个阶段。这保证整个实习过程中一直有足够的人手，各个计划得以按时完成。
2. 群体 debug 与观战。在初期代码的编写中，难免会出现各种 bug 难以发现，而仅凭个人的力量无法解决。这时让编写代码的同学给其他同学讲述整个代码，一方面有助于所有人对

代码各部分的理解，另一方面别人的想法与自己不同，往往也能找到错误，甚至能给出新的算法优化与提高。在后期代码测试优化阶段，大家共同观看代码在本地或者天梯上的运行结果，取出典型情况并与代码具体估值函数相对应，进行估值和决策函数的优化。实际上，如果提高优化部分仅仅由一个人来做，则往往由于个人难以考虑全面而进入错误的方向或者死胡同。

不足：

1. 对各种资源的利用不足。技术组其实提供了足够的调试资源和信息，如爬虫。但是可能由于时间、组员能力等限制，在整个实习过程中并未充分利用，在一定程度上影响了代码的优化进程与最终质量。
2. 本地调试略少。在基本代码编写阶段，大家还都在本地运行，编写各种函数来测试代码的性能。然而天梯开放后虽然也有进行，整个调试的重心却更偏向天梯实战测试，由于天梯的算力限制和其本身性质，本来就不是为了各种测试调参而生。这导致本地的算力没有充分利用，各种参数的调整和优化只停留在比较初级的阶段。
3. 启发式规则不够。在整个代码编写的后期，基本的结构已经确定，在这个基础上虽然进行了较多的结构性优化，但是对于整体算法的效率仍然有限。虽然有一定的思路和改进方式，但是并不能化成具体的规则判断。导致最后一段时间并没有什么大的提高。

4.3 建议与设想

建议：

本次对战版 2048 非常有趣，在原来规则上进行创新，对战性也较强。然而存在一个可能的问题——代码克制。虽然代码风格和水平确实有些不同，然而在相同水平的代码下，常常出现各种比较严重的循环克制。所以一方面或许可以进一步优化规则来避免这种情况，另一方面可以一定程度改进赛制（比如采用循环积分赛等），这样或许可以让最终排名更加贴近实际水平。

寄语：

请一定重视团队分工与合作，合理的任务分配与计划比个人能力更重要。本组成员并没有十分拔尖出色的所谓“大佬”，但是大家都热心参与，积极工作，最终取得了不错的成绩。另一方面，在最终竞赛结果之外，加强队友之间的交流也可以增进友谊，互相学习，这甚至可能比最终成绩更有意义。

后续设想：

我们将会与其他组进行代码交流，分享实习经验，并且进一步优化代码。

5 致谢

首先，感谢陈斌老师能给予同学们这个大作业对战的机会，同时感谢整个实习过程中技术组的大力支持与帮助。如果没有他们提供的平台和资源，我们无法顺利完成本次大作业。

其次，我们感谢在整个过程中与我们一直保持友好联系的几个小组（如 X-ray 组，520 组）。这些小组打破了“互相竞争”的障壁，组长或组员与本组成员进行适当的交流，并且互相鼓励，是我们在实习过程中的精神慰藉。

再者，感谢我们在天体热身赛和最终现场比赛遇到的各个对手。与他们的各场比赛帮助我们找到算法思路的问题，指明了改进的方向。天梯有一些同学（由于昵称和代码更改的原因只能回忆起一部分，如“枪兵”，“114”，“晚上好”，“无中生友的 bili”等同学）非常热衷于与我们对战交流，每次我们上传或者更新代码便会来挑战我们，这些对战经验在我们的提高过程中起了重要的作用。他们是我们在提高过程中的好对手，更是好朋友。另外再尤其感谢 Uniform 组，该小组在天梯上稳坐第一把交椅，与我们进行多次对决，也是我们一直追赶的目标。我们有幸最终在 F19 的决赛中与 Uniform 组对战，虽然最后告负，但我们心服口服，是我们尊敬的对手。

最后，感谢我们的家人和邻居对我们的理解和支持。我们常常开会至深夜，讨论的又比较激烈，有可能会对家人和邻居造成干扰。感谢他们的理解和帮助，让我们拥有一个相对自由的交流环境。

再次向以上各位致以衷心的感谢！

参考文献

[1] 2048 智能算法讨论 [EB/OL].https://blog.csdn.net/sky_j123/article/details/37832803,2014-7-16.

[2] AI 玩 2048[EB/OL].https://tourist.blog.csdn.net/article/details/84147363,2018-11-16.

[3] 蒙特卡洛树搜索（新手教程）[EB/OL].https://blog.csdn.net/qq_16137569/article/details/83543641,2018-11-01.

数据结构与算法课程实习作业报告

殷荣章* 陈赛 刘行远

摘要：算法的策略，在下子阶段，由于在对方下子的效率很低，远不如在己方下子的收益大，因而只要己方可以下子，便在己方棋盘下子。在合并阶段，主要是利用 minimax 算法，构建搜索树，利用估值函数计算最深层次的局面估值，配合 α - β 剪枝进行对局搜索，找到可选方向中传递上来的估值函数最大的一项，作为最终决策方向输出。

关键字：minimax 算法，搜索树， α - β 剪枝，估值函数

1 算法思想 2

1.1 总体思路 2

1.2 算法流程图 3

1.3 算法运行时间复杂度分析 3

2 程序代码说明 4

2.1 数据结构说明 4

2.2 函数说明 4

2.3 程序限制 7
3 实验结果 7
3.1 测试数据 7
3.2 结果分析 8
3.3 经典战局（可选） 8
4 实习过程总结 9
4.1 分工与合作 9
4.2 经验与教训 9
4.3 建议与设想 9
5 致谢 10
6 参考文献 10

1 算法思想

1.1 总体思路

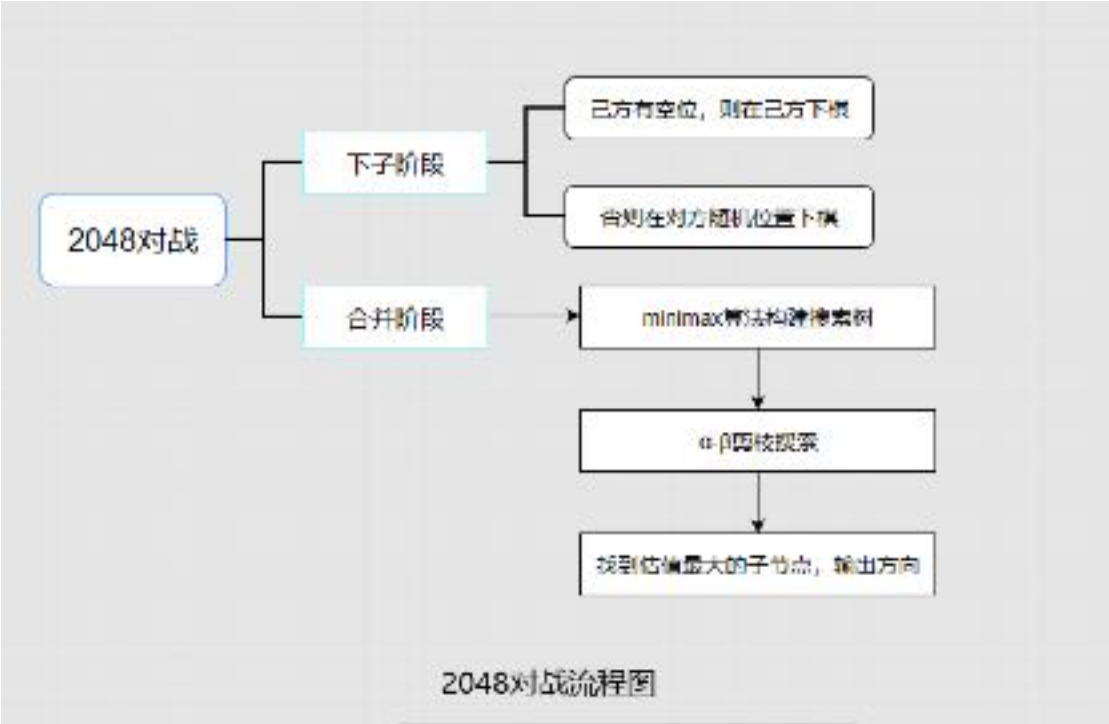
算法的总体思路主要集中在合并阶段，因为在下子阶段，由于在对方下子的效率很低（很难找到在对方的大子即将合并，并且中间有空隙的时机下插入一个子，而且对方完全可以通过绕路合并的过程完成对大子的合并，因此干扰对方合并的策略往往不会给对方造成有效阻碍，反而会增加对方的总体分数，更有利于对方合并更大的子），远不如在己方下子的收益大（在对方下子意味着放弃在己方棋盘内增加分数的机会），因而我们的下子阶段策略是只要己方可以下子，便在己方棋盘下子，如果己方棋盘已满，便在对方棋盘随机位置下子。

合并阶段的算法是主要过程，思路主要是利用 minimax 算法，构建搜索树（搜索树的建立意味着需要预估己方和对方的下子阶段的下子位置，由前面的分析，在下子阶段己方和敌方都采取下在自己方棋盘的位置），利用估值函数计算最深层次的局面估值，配合 α - β 剪枝进行对局搜索，向上传递估值后，找

到可选方向中估值函数最大的方向，作为最终决策方向输出。

由于每个局面下子后往往会有四个分支，递归深度增加，使得搜索树复杂程度指数增加，由于估值函数花费的时间较长，实测递归到第三层可以勉强挤进 5s 内（到 4 层的话几乎每次都超时，虽然考虑过使用启发式算法减少搜索分支，但是没有找到好的启发算法，实验的效果很差，因此没有使用启发式算法，而是暴力建树）。因为考虑到 2048 对战游戏需要预估对面的下子和合并方向，每次预测不一定准确，所以递归深度越深反而可能会让结果的偏差值越大，因此写完这样一个代码框架后，后期的调整工作主要是在估值函数调整上。

1.2 算法流程图



2048对战流程图

1.3 算法运行时间复杂度分析

算法运行时间的消耗主要是在合并阶段的两个过程：minimax 算法构建搜索树和 α - β 剪枝搜索。

设递归深度为 n。

minimax 算法构建搜索树时，树的大小和递归深度 n 有关，且构建树节点时间复杂度为 O(4^n)，对叶节点需要调用估值函数，估值函数是一个对整个棋盘状态的估值，基本是个定值，设为 t，而叶节点数位 4^n，故该阶段消耗总时间为 t*4^n+O(4^n)；

α - β 剪枝搜索的时间也主要和递归深度有关，假设平均剪枝剪掉一半的分支，那么消耗的时间就是 1/2*O(4^n)。

故总的时间复杂度为 O(4^n)，n 为递归深度。

实际测试过程中，取 n>3 时经常超时，故选取 n=3；这时的时间花费主要是在估值函数上（主要的计算过程在估值函数上，构建树和搜索树的过程不涉及过多计算，因而不太消耗时间），因为每次构建搜索树后会产生 4^3=64 个子节点，因此需要调用 64 次估值函数。

2 程序代码说明

2.1 数据结构说明

算法中定义了一个树节点类 TreeNode，以此实现树的数据结构（四叉树结构），此外还使用了列表数据类型作为局面棋子位置估值的权重。

2.2 函数说明

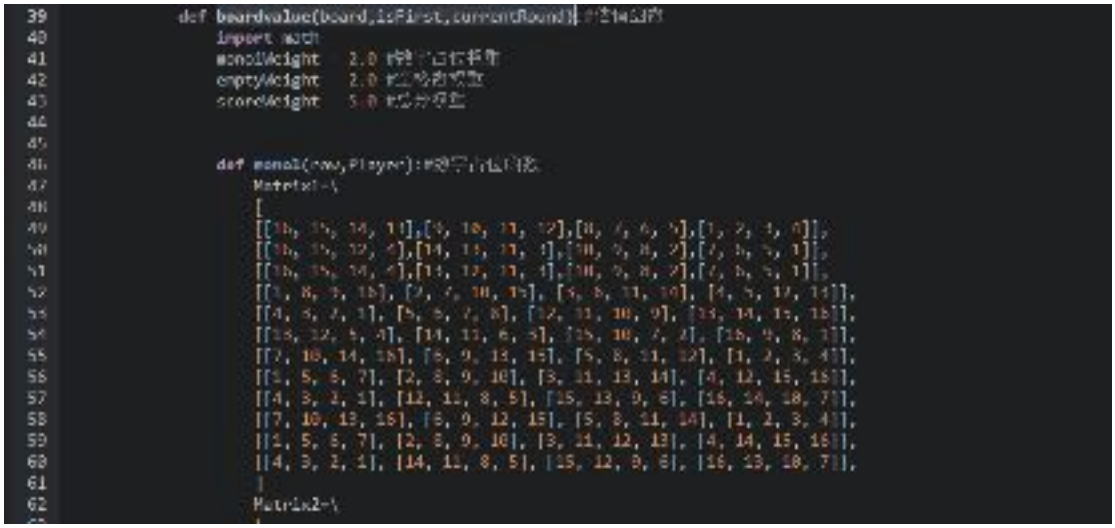
1、minimax(isFirst,Max_depth, board, currentRound)

minimax 函数为整个程序的框架函数，包括调用构建 minimax 决策树函数 tree（调用估值函数 boardvalue），调用 alphabeta 函数实现 α - β 剪枝搜索，以及最终决策方向的输出。

2、alphabeta(node, depth, Player)

alphabeta 函数功能为实现 α - β 剪枝搜索决策树，剪枝搜索后将局面估值向上传递到根节点。

3、boardvalue(board,isFirst,currentRound)



```
                1
                ||1, 2, 3, 4], |8, 7, 6, 5], |9, 10, 11, 12], |16, 15, 14, 13||,
                [[7, 6, 5, 1], [10, 9, 8, 2], [14, 13, 11, 3], [16, 15, 12, 4]],
                [[7, 6, 5, 1], [10, 9, 8, 2], [11, 12, 11, 4], [16, 15, 14, 4]],
                [[4, 3, 12, 13], [3, 6, 11, 14], [2, 7, 10, 15], [1, 8, 9, 16]],
                ||13, 14, 15, 16], |12, 11, 10, 9], |5, 6, 7, 8], |4, 3, 2, 1]],
                [[16, 9, 8, 1], [15, 10, 7, 2], [14, 11, 6, 3], [13, 12, 5, 4]],
                [[1, 2, 7, 8], [7, 8, 11, 12], [6, 9, 13, 15], [7, 10, 14, 16]],
                ||4, 12, 15, 16], |3, 11, 13, 14], |2, 8, 9, 10], |1, 5, 6, 7]],
                [[16, 14, 10, 7], [15, 13, 9, 6], [12, 11, 8, 5], [4, 3, 2, 1]],
                [[1, 2, 7, 8], [7, 8, 11, 16], [6, 9, 12, 15], [7, 10, 14, 16]],
                [[4, 14, 15, 16], [3, 11, 12, 13], [2, 8, 9, 10], [1, 5, 6, 7]],
                ||16, 13, 10, 7], |15, 12, 9, 6], |14, 11, 8, 5], |4, 3, 2, 1]|
                ]
                mon 0
                lisnum=[0]*12
                if Player:
                    for i in range(1):
                        for j in range(4):
                            if raw[i][j][1]==Player:
                                for x in range(12):
                                    lisnum[x]+=Matrix1[x][i][j]*raw[i][j][0]
                            else:
                                for x in range(12):
                                    lisnum[x]-=Matrix1[x][i][j]*raw[i][j][0]

                else:
```

```
                                for x in range(12):
                                    lisnum[x]-=Matrix1[x][i][j]*raw[i][j][0]

                else:
                    for i in range(4):
                        for j in range(4,8):
                            if raw[i][j][1]==Player:
                                for x in range(12):
                                    lisnum[x]+=Matrix2[x][i][j-4]*raw[i][j][0]
                            else:
                                for x in range(12):
                                    lisnum[x]-=Matrix2[x][i][j-4]*raw[i][j][0]

                return math.log(abs(max(lisnum))+1)
            #此方估值
            value1= \
                mono1(board.getRaw(),isFirst) * monoWeight + \
                math.log(len(board.getNone(isFirst))+1) * emptyWeight + \
                numcount(board,isFirst) * scoreWeight
            #对方估值
            value2= \
                mono1(board.getRaw(),not isFirst) * monoWeight + \
                math.log(len(board.getNone(not isFirst))+1) * emptyWeight + \
                numcount(board,not isFirst) * scoreWeight

            return value1-value2
```

```
        def numcount(board,Player):#求分数
            lisnum=board.getScore(Player)
            count=0
            for i in lisnum:
                count+=2**i
            return count
```

boardvalue 函数功能为对传入的局面进行估值，返回估值结果传给叶节点。

其包括三个部分：mono1(raw,Player)某方数字占位优劣估值函数，numcount(board,Player)某方总分计算，以及某方剩余空格数计算 math.log(len(board.getNone(isFirst))+1)。

对应三个权重：

```
41                  mono1Weight = 2.0 #数字占位权重
42                  emptyWeight = 2.0 #空格数权重
43                  scoreWeight = 5.0 #总分权重
```

其中：

mono1(raw,Player)某方数字占位优劣估值函数，采用估值矩阵作为权重与对应数字乘积之和作为数字占位的优劣值，使用了 12 个估值矩阵（数字排列考虑了数字间的平滑性和整体的单调性）依次与某方棋盘中的数值进行相乘计算求加和（其中，如过己方棋盘出现对方的棋子，则会减去该位置权重与棋子的乘积），取 12 个结果中的最大值作为返回值（数字越大，代表局面越好）。

numcount(board,Player)某方总分计算函数，直接计算某方所拥有棋子的分数之和，分数越高，优势越大。

math.log(len(board.getNone(isFirst))+1)为某方空格数的计算，空格数越多，说明该方局面灵活度大，有优势。

估值的结果为：

己方各函数返回值与对应权重的乘积之和 - 对方各函数返回值与对应权重的乘积之和，

即：

```
100             #己方估值
101             value1 = \
102                   mono1(board.getNow(),isFirst) * mono1Weight + \
103                   math.log(len(board.getNone(isFirst))+1) * emptyWeight + \
104                   numcount(board,isFirst) * scoreWeight
105             #对方估值
106             value2 = \
107                   mono1(board.getNow(),not isFirst) * mono1Weight + \
108                   math.log(len(board.getNone(not isFirst))+1) * emptyWeight + \
109                   numcount(board,not isFirst) * scoreWeight
110
111             return value1-value2
112
```

2.3 程序限制

程序的时间卡的比较紧，结束时经常出现 4.5s 以上的用时，因此如果对局中运气不太好，搜索树经常铺满，或者 α - β 剪枝的效果经常很差，可能会导致超时现象。

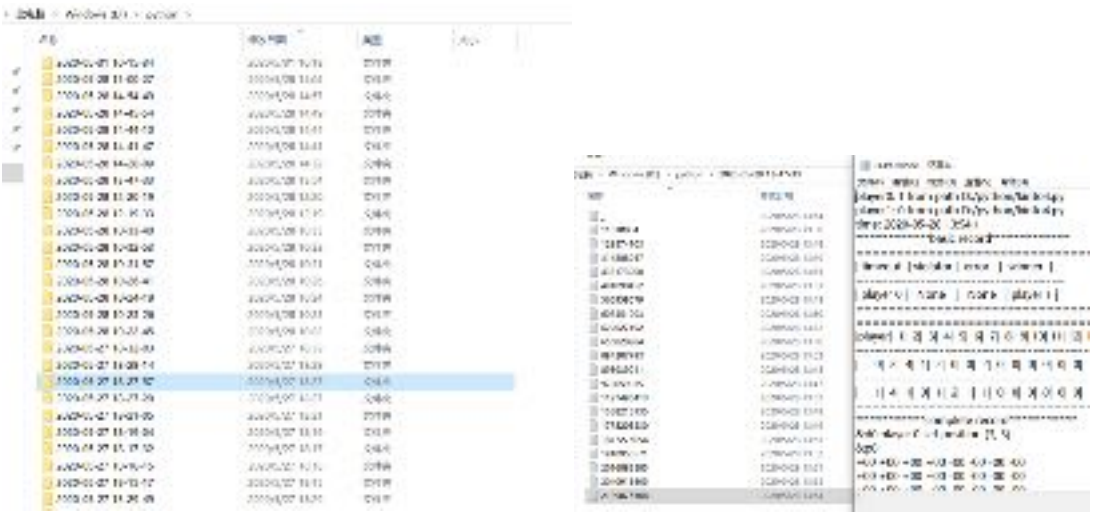
3 实验结果

3.1 测试数据

实验环境说明：

- 硬件配置：（CPU/内存）：Intel(R) Core(TM) i5-7200U CPU @ 2.50GHz 2.70GHz 内存：7.83GB
- 操作系统：（名称/版本）：win10 ， 64 位
- Python 版本：（版本号）：3.7

1、利用 github 里的集成开发环境与不同参数下的估值函数进行模拟对战，统计结果，计算胜率，调整估值函数



本地对战记录

2、天梯上的自由对战，观看对战过程，找到局势转折点，分析原因，调整估值函数

3.2 结果分析

测试结果主要用来对估值函数进行改进。其实，刚开始设置了 5 个估值考虑函数，还包括额外的平滑度估值（利用每个棋子和周围棋子差值的绝对值之和来判断）和单调性估值（从上到下，从左到右，前面棋子小于等于后面棋子就将结果值加上这两个棋子的和，否则减去这两个棋子的和），结果发现直接用棋子占位函数估值更加准确，还能减少不少计算量，就把这两个函数舍去了。因此，后面测试主要是对估值函数中 3 个权重因子的调整。

时间上，因为估值函数的调整符合预期值（比较极限），开销也主要在估值函数上，涉及比较多的计算。

3.3 经典战局（可选）

4 实习过程总结

4.1 分工与合作

说来惭愧，由于本来成员就仅有 4 个人，组队后，又有一位同学退课了，使得我们小组仅有 3 个人。成员讨论的方式是微信讨论，但是另外两位同学没有怎么参与讨论（可能在划水或者有其他要事吧，身为组长也有责任），于是编写代码的工作就只能由组长一人来做了。

4.2 经验与教训

大作业的工作量确实挺大的，设计思路，查阅资料，编写代码，调整参数，实际对战，

花了挺长一段时间，但是收获还是挺大的，学习了很多博弈方面的算法和思路（虽然大部分都没有用上）。因为是大作业，所以小组合作显得尤为重要，也希望能和不同的人进行算法方面的交流，提供不同的思路改进代码，但是情况比较无奈（疫情影响，只能线上搞也没办法），没有组织好讨论工作，组长也有一部分责任，所以希望下次能更好地参与讨论吧。

4.3 建议与设想

数算大作业每年都挺有意思的，建议的话就今年来说，由于本地搭建的模拟平台因为没法直观看到对局中的细节，会使调试工作会变得困难（本身电脑跑的慢，调试时间就长）。

关于组队机制，由于很多有些同学可能没有什么认识的人，拖到最后便随机去了一个陌生的小组，而导致参与度很少，而大佬抱团现象也很严重，相对的使得新手同学聚团，甚至划水同学聚团，平衡性差别很大。因此可以考虑在自由组队时间截至后，采用随机分配的方式组队。

寄语的话，整体这门数算课还是很有意思的，希望各位学弟学妹们可以纷纷选报，多多参与周围人或群里的讨论，把功夫花在平时，打下扎实的基础后，参与数算的各种活动也将会变得更加有兴致。

5 致谢

不管怎样，都要感谢我们全组的成员们，相遇即是缘分，虽然最后的结果可能不尽如人意。也要感谢老师的辛苦授课、助教们的耐心解答和技术组的同学的辛勤付出。

6 参考文献

[1]https://stackoverflow.com/questions/22342854/what-is-the-optimal-algorithm-for-the-game-2048

[2]https://github.com/ov3y/2048-AI

[3]http://www.flyingmachinestudios.com/programming/minimax/

[4]http://web.cs.ucla.edu/~rosen/161/notes/alphabeta.html

数据结构与算法课程实习作业报告

方静潮、王述冰、李春江、陈铮、娄宇*

摘要：本算法以估值函数和估值矩阵为核心，以最大最小搜索树为实现方法，总体来讲采用的是一些启发式规则和积极进攻的策略，另外，在时间的控制上，还采用了防超时的战略和通过剩余的时间决定搜索深度的方法，使得时间能够充分利用。这样仅仅依靠二百多行的代码，就取得了比赛的季军。

关键字：估值矩阵 搜索树 启发式规则 时间统筹

1 算法思想.....2
1.1 总体思路.....2
1.2 算法流程图.....3
1.3 算法运行时间复杂度分析.....4
2 程序代码说明.....4
2.1 数据结构说明.....4
2.2 函数说明.....4
2.3 程序限制.....8
3 实验结果.....9
3.1 测试数据.....9
3.2 结果分析.....10
3.3 经典战局（可选）.....10
4 实习过程总结.....11
4.1 分工与合作.....11
4.2 经验与教训.....12
4.3 建议与设想.....12
5 致谢.....12
6 参考文献.....12

1 算法思想

1.1 总体思路

我们的游戏策略分为两大部分——智慧搜索树决策和随机决策，随机决策是在时间严重不足时，对再次智慧搜索树可能带来的超时隐患而做出的必要补充。下面主要介绍智慧搜索树决策实现：

首先，我们的游戏策略主干为保守性的自我发育，因此初始化棋手的决策 choice 仅包含当前所允许的所有 direction 选择以及在己方棋盘的 position 下棋选择，在此基础上通过游戏进行过程中对方出现的破绽或危险而添加主动的进攻或制裁类型 choice 选项，最后通过对 choice 的迭代分析，对每个 choice 进行搜索树搜索，通过对最终棋盘的 score 估值评选出最佳 choice 并返回。

在设计第一代算法时，我们打算以单人二零四八的游戏规则进行设计来为此后的算法改进提供用于分析的基础，因此仅是考虑通过对己方较大棋子的重重保护，即通过设计己方较大棋子处于己方角落时能获取更大的棋盘评分，来避免己方棋子被对方棋子合并以及实现己方棋子的易于合并来进行游戏，因此棋手的 choice 在 position 模式最初只包含在己方棋盘下棋的决策，但对战结果并不理想，我们发现单纯的保护较大棋子容易被对方的反制策略针对而陷入死局，比如对我方即将合并的棋子进行堵塞，而受估值函数 score 影响下分布在双方棋盘边缘的小棋子也更容易被对方合并。在考虑了这个因素后我们对算法做出了第一次改进。

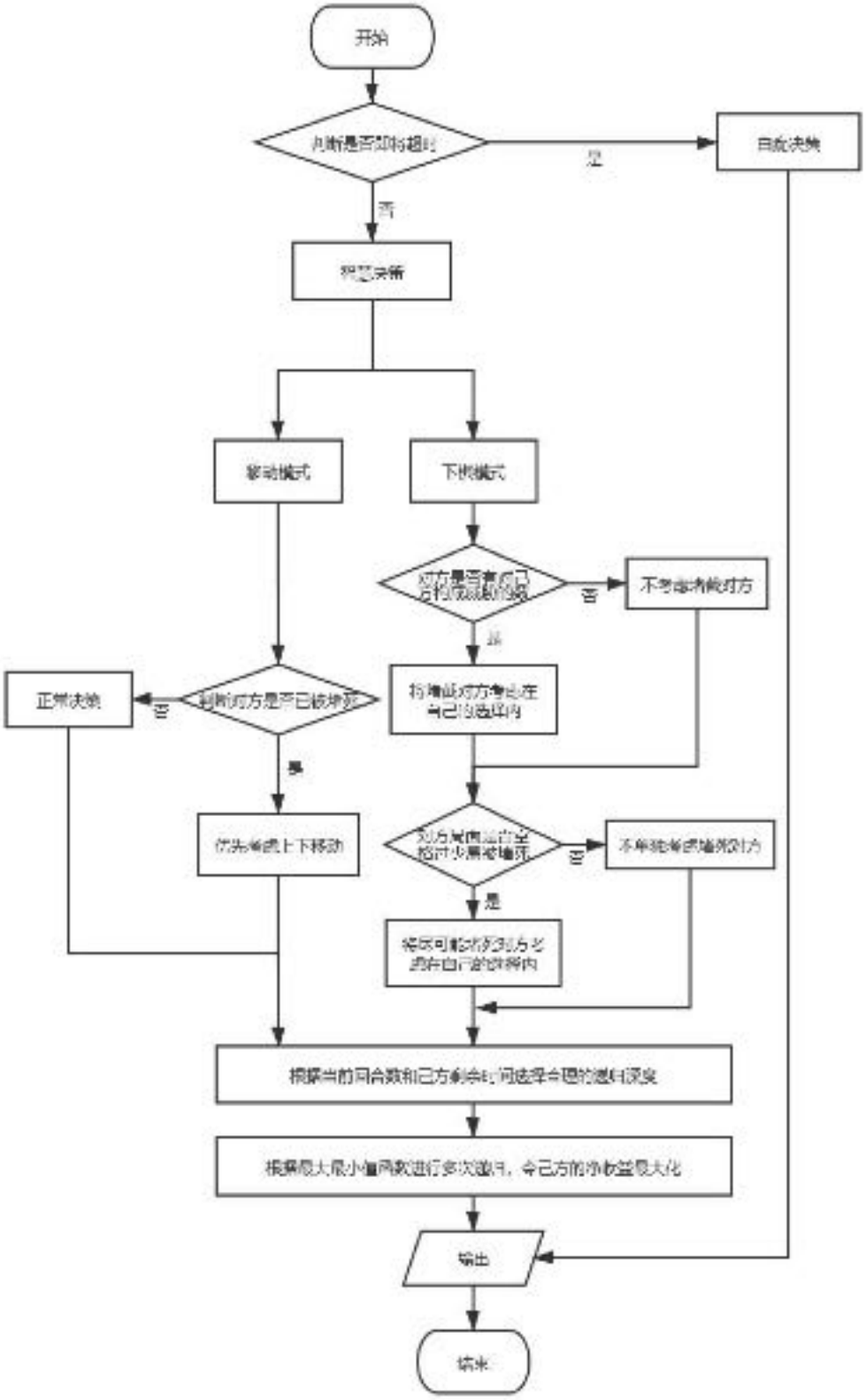
是的，对战游戏与怎么能少的了对对方的进攻呢？于是我们为第二代算法添加了一个用于进攻的函数 attack。在初始化 choice 之后，会对对方的棋盘进行扫描，通过获取对方棋盘棋子的级别和数量判断棋盘是否具有威胁棋子，如果具备，将对对方较大棋子即将可能出现的合并进行堵截，对战胜率得到了明显的提高。

在一次小组讨论中，我们发现天梯排名较高的很多算法都有着较强的进攻性，在他们的对战记录中有很多将对方堵死的情况(即在对方在 position 模式下自家棋盘无棋位可下，在 direction 模式下无法 move)，这给了我们很大的启发，此后我们为估值函数 score 增添了新的估值方法，即将能堵死对方的下棋方案设置为最高优先级，而可能被对方堵死的下棋方案为最低优先级，形成我们的第三代算法，此时我们的算法在天梯上已能和高排名算法一战。

第四代算法是对第三代算法的简单强化，主要内容为为当对方所剩棋位较少时，棋手 choice 优先填充对方棋位来实现堵死对方的目的。

不可否认的是，即使在进行重重改进之后，我们的算法依然存在很多缺陷，无法面对各式各样的情况，或者说我们的进攻性 choice 都有些许赌运气的成分，碍于时间与精力，我们此后最终的算法只是对第四代算法进行了参数的微调，并未再做出其它突破。

1.2 算法流程图


开始
是
否
被动模式
主动模式
否
是
正常决策
否
是
优先考虑上下移动
是
否
根据当前回合数和己方剩余时间选择合理的遍历深度
根据极大极小值网格进行多次遍历，令己方的净收益最大化
输出
结束

1.3 算法运行时间复杂度分析

首先整体来看，代码的主要时间开销第一在决策搜索树上面，深度越大，所用时间越长。其次，在具体实现上，因为 2048 合并的特性，不存在可以直接撤销操作的方法，每一次都要要复制整个棋盘，防止更改，这样一个不断复制的过程也耗费了大量的时间，是每一回合耗时的主要部分。

对于时间复杂度，如果对局定义回合数为 n，虽然每一步的递归深度都是动态的（用时不同），但是从平均来看（也是简单化的考虑），时间复杂度是 o（n）的。

2 程序代码说明

2.1 数据结构说明

算法主要采用了列表、递归和类的数据结构
列表：主要用于存储当前允许的下棋与移动选择
递归：用于实现对 choice 评分搜索树构建
类：定义了 Player 类，用于表示棋手

2.2 函数说明

算法接口一览：

估值函数矩阵 Player.myKey，是算法中 score 函数的核心部分，也是我们组经过历次调试得到的最优解。

```
class Player:
    def __init__(self, isFirst, array):
        # 初始化
        self.isFirst = isFirst
        self.array = array
        self.myKey=[8,8,8,8,6,4,2,1]    #估值矩阵
```

输出接口，根据 mode 模式以及当前的时间限制判断返回值，if else 的判断使得当我方时间小于一定数值时直接调用白痴 AI，为防止超时上一道保险：

```
    def output(self, currentRound, board, mode):
        if mode=='_position' or mode=='_direction':    #跳过此步
            return  None
        else:
            myTime=board.getTime(self.isFirst)
            if myTime>0.05:
                return self.smartDecision(currentRound, board, mode,myTime)
            else:
                return self.artificialIdiot(currentRound, board, mode)
```

算法主体部分，主要包括开始的认为对方棋子足够大对自己构成威胁就主动堵截和后来的根据比赛回合数以及剩余时间灵活控制递归深度两部分，最终根据最大最小值函数递归搜索，选择一个对己方最有利的局面：

```
def smartDecision(self, currentRound, board, mode,myTime):
    choice=self.getChoice(currentRound, board, mode,self.isFirst) ##初始化choice
    if mode=='position':
        available=board.getNone(not self.isFirst)
        if len(available)<3:
            choice+=available
        anamyChess=board.getScore(not self.isFirst)
        if len(anamyChess)>=2:
            if anamyChess[-1]>2 and anamyChess[-1]==anamyChess[-2]:
                dangerValue=anamyChess[-1]
                danger=self.getPosition(board,dangerValue)
                if len(danger)>=2:
                    myAttack=self.attack(board,danger,dangerValue)
                    if myAttack:
                        choice+=myAttack
        if not choice:
            choice=board.getNone(not self.isFirst)
```

在 smartDecision 中，首先初始化 choice 获取棋手当前允许的 move 以及己方棋盘空白棋位，在此 choice 的基础之上判断是否为棋手增添进攻方案，而判断方式有两个，一是对方棋盘空白棋位较少，此时进攻对方棋盘能促进堵死对方，而是判断对方是否即将完成较大数的合并，此时进攻对方棋盘能阻止对方棋子数值的增长。

```
    else:    #方向模式
        if self.isDead(board,not self.isFirst):
            temChoice=[]
            if 0 in choice:
                temChoice.append(0)
            if 1 in choice:
                temChoice.append(1)
            if temChoice:
                choice=temChoice
```

isdead 函数用于判断 direction 模式中对方是否已经被堵死，并会维持堵死对方的状态，在此处其实我们还有向对方 move 的方向选项，但在对战过程中，我们发现此举存在将原本堵死的对方棋盘释放的意外，因缺乏进一步判断商讨的以及对时间消耗的顾虑，故而舍弃

```
    if len(choice)==1:        #节能，返回单一选项
        return choice[0]
    else:                     #通过搜索树进行为所有choice打分，选出最佳choice并返回
        if currentRound<20:   #回合数较少时，影响不大，或许少考虑一些能够取得奇效
            deep=7
        elif myTime>0.5:      #时间较长，多搜索一层
            deep=8
        else:
            deep=7
        lst=[-1,-(float('inf'))]      #记录最大选择[choice,该choice的score]
        for mychoice in choice:       #对choice迭代搜索
            myboard=board.copy()
            mymode=mode #拷贝模式
            mycurrentRound=currentRound
            mymode,mycurrentRound=self.done(mychoice,myboard,mode,self.isFirst,mycurrentRound)
            tem=self.minimize(mycurrentRound,deep,-(float('inf')),float('inf'),myboard, mymode)
            if tem>lst[1]:   #记录最大选择
                lst=[mychoice,tem]
        return lst[0]
```

此后对 choice 进行迭代，运用递归进行棋盘的模拟实现虚拟对战从而评选出最优 choice

估值函数连同前面的估值矩阵，是算法的核心，也是代码调参的重要之处，我们组在小组讨论时多次发现，通过设计一个合理有效的估值函数，可以将诸多可以想到的策略交由估值函数来进行判断，一个好的估值函数是代码“智慧”高的重要保障。

我们设计 score 接口用于评价棋盘的优劣，在对 choice 的迭代估分中我们期望得到己方认为最有价值的棋盘，对方认为最没有价值的棋盘，此处 myScore 是模拟棋盘中己方的估值，anamyScore 是模拟棋盘中对方的估值，取两者之差作为最终估值。

```
    def score(self,board):
        myScore=0
        anamyScore=0
        myboard=board.getRaw()
        if self.isFirst:                    #先手方
            for x in range(4):                  #遍历全棋盘
                for y in range(8):
                    value=myboard[x][y][0]      #得到数值
                    if value:                               #非空
                        if myboard[x][y][1]==self.isFirst:  #己方棋子
                            myScore+=self.myKey[y]*(2.1**value)
                        else:     #对方棋子
                            anamyScore+=8*(2.1**value)
            return myScore-anamyScore
        else:     #后手方
            for x in range(4):                  #遍历全棋盘
                for y in range(8):
                    value=myboard[x][y][0]      #得到数值
                    if value:                               #非空
                        if myboard[x][y][1]==self.isFirst:  #己方棋子
                            myScore+=self.myKey[7-y]*(2.1**value)
                        else:     #对方棋子
                            anamyScore+=8*(2.1**value)
            return myScore-anamyScore
```

getChoice 获得我方本回合的选择(不含位置模式时对方的空格),用于初始化棋手 choice：

```
    def getChoice(self,currentRound, board, mode,player):
        if mode=='direction':
            lst=[]
            for x in range(4):
                myboard=board.copy()        #拷贝棋盘,避免更改
                if myboard.move(player,x):   #如果有移动
                    lst.append(x)
        elif mode=='position':      #下棋模式
            lst=[]
            tem=board.getNext(player,currentRound)
            if tem:
                lst.append(tem)  #加入可走的棋
        return lst
```

getPosition 接口用于返回某一级别棋子在己方或对方棋盘上所在的位置：

```
    def getPosition(self,board,value):          #返回对方在对方棋盘中某一个级别所有棋子的列表
        lst=[]
        if self.isFirst:
            y0=4
        else:
            y0=0
        for x in range(4):
            for y in range(y0,y0+4):
                if board.getBelong((x,y)) !=self.isFirst and board.getValue((x,y))==value:
                    lst.append((x,y))
        return lst
```

进攻函数，通过逐个遍历判断对方能够合并的棋子是否存在合并的条件，如果存在，则返回进攻可选空棋位，如果不存在，返回空白列表，取消本次进攻：

```
122     def attack(self,board,danger,dangerValue):
123         available=board.gethone(not self.isFirst)
124         lst=[]
125         if self.isFirst:
126             y0=4
127         else:
128             y0=0
129         for mydanger in danger:
130             x=mydanger[0]
131             y=mydanger[1]
132             if (x==0) and ((1,y) in available) and ((2,y) in danger):
133                 lst.append((1,y))
134             elif (x==0) and ((1,y) in available) and ((2,y) in available) and ((3,y) in danger):
135                 lst.append((1,y))
136                 lst.append((2,y))
137             elif (x==1) and ((2,y) in available) and ((3,y) in danger):
138                 lst.append((2,y))
139             y-=y0
140             if (y==0) and ((x,1+y0) in available) and ((x,2+y0) in danger):
141                 lst.append((x,1+y0))
142             elif (y==0) and ((x,1+y0) in available) and ((x,2+y0) in available) and ((x,3+y0) in danger):
143                 lst.append((x,1+y0))
144                 lst.append((x,2+y0))
145             elif (y==1) and ((x,2+y0) in available) and ((x,3+y0) in danger):
146                 lst.append((x,2+y0))
147         return lst
```

Done 接口用于返回进行一次 choice 操作后的模拟棋盘，为递归搜索提供基础：

```
161     def done(self,choice,board,mode,player,currentRound):      #操作
162         if mode == 'direction':                #位置模式，移动
163             board.move(player,choice)
164             if not player:   #后手方，更改模式
165                 mode='position'
166                 currentRound+=1
167         elif mode=='position':
168             board.add(player,choice)           #下棋
169             if not player:
170                 mode='direction'
171         return mode,currentRound
```

考虑到被堵死或者堵死对方有的时候是决定胜负的关键，新增了判断是否被堵死的接口，这一点，在最大最小值函数里面的极高和极低期望值，以及方向模式下，如果对方已经被堵死尽量不要放跑对面也有所体现。

```
250     def isDead(self,board,player):      #判断处于位置模式的玩家是否已经被堵死
251         available=board.getNone(player)
252         if not available:
253             lst=[]
254             for x in range(4):
255                 myboard=board.copy()          #拷贝棋盘，避免更改
256                 if myboard.move(player,x):    #如果有移动
257                     lst.append(x)
258             if not lst:
259                 return True
260             else:
261                 return False
262         else:
263             return False
```

最大值函数，除了实现了决策树的递归搜索和剪枝，其主体和 smartDesionl 类似，用于递归寻找能够实现己方评分最高的棋盘，因为观察到在对局中被堵死的一方胜算已经不大，

还增添了防堵死机制（即最后几行），也算是一种弃卒保车的策略：

```
    def maximize(self,currentRound,deep,alpha,beta,board, mode):
        if deep==0 or currentRound>len(self.array)-1:
            if mode=='position':
                if self.isDead(board,self.isFirst) and (not self.isDead(board,not self.isFirst)):
                    return -100000000
                else:
                    return self.score(board)
            else:
                return self.score(board)
```

```
        else:
            tem=-(float('inf'))
            choice=self.getChoice(currentRound,board, mode,self.isFirst)
            if choice:
                if mode=='position':
                    available=board.getNone(not self.isFirst)
                    if len(available)<3:
                        choice+=available
                for mychoice in choice:
                    myboard=board.copy()
                    mymode=mode
                    mycurrentRound=currentRound
                    mymode,mycurrentRound=self.done(mychoice,myboard,mode,self.isFirst,mycurrentRound)
                    choiceScore=self.minimize(mycurrentRound,deep-1,alpha,beta,myboard, mymode)
                    if choiceScore!=None:
                        tem=max(tem,choiceScore)
                        if tem>=beta:
                            return None
                        alpha=max(tem,alpha)
                return tem
```

```
            else:
                if mode=='position':
                    if self.isDead(board,self.isFirst) and (not self.isDead(board,not self.isFirst)):
                        return -100000000
                    else:
                        return self.score(board)
                else:
                    return -100000000
```

最小值函数与上面的最大值函数完全类似，但用于寻找实现对方评分最低的棋盘，防堵死机制也相应地变成了试图堵死对方，此处为节约篇幅故不列出。

2.3 程序限制

开始在自由对战调试时，没有在开始进行剩余时间的判断，因此程序在极端情况下（比如与参数略微改变的上代版本对战时）可能超时，但正式比赛时加入了这个判断，除了极特殊的情况（开始几回合就将时间消耗完导致即使后续完全依靠白痴 AI，决策时间也不足；或在时间将要耗尽时，递归不深的一步仍然用时 0.03 秒以上）会超时以外，理论上不存在，实际也没有发现其他限制。

另外，在游戏策略层面，我们组的算法更善于对战保守类的对手。而且由于估值函数相对单一，在面对进攻性的对手或较为复杂的情况时可能疲于应付，也相对容易被克制。

3 实验结果

3.1 测试数据

实验环境说明：

- 硬件配置：CPU:Intel(R) Core(TM) i5-8265U @ 1.60GHz 1.80GHz;RAM:8.00GB
- 操作系统：Windows 10；版本号：1809
- Python 版本：3.5.3

测试程序：主要通过天梯对战进行测试，另外在前期，还编写了白痴 AI 和复读机代码用于测试，天梯测试结果如下（部分）：

在代码修改（策略改变或参数调整）以后，为检验效果是否和预期一致，我们组还经常用其与之前版本的代码对战。下图是效果较好的一次版本修改与上一代的对战情况图：

3.2 结果分析

在进行测试的过程中，我们发现最初设计的 score 估值函数更多的起到了使己方合并更加容易地合并而不是避免被对手合并地作用，这意味着为除 score 估值函数外的优先堵塞对方、优先阻止对方合并的强制实行起到了很大的作用。

在算法设计时我们发现无论如何改变参数和剪枝，由于智慧决策递归搜索对时间和空间资源的巨大消耗，总是会出现己方超时而落败的情况，加入了白痴 AI 进行必要补充后，算法的运行时间得到了更好的控制。

3.3 经典战局

算法在初期不足 1100 等级分的情况下挑战 1500 分大佬的战局，虽然遗憾落败，但是每回合都能合成 2048 及以上，可谓打得难解难分，着实让人惊讶；甚至还有一局我方自始至终以绝对的优势获胜，可谓虽败犹荣。

4 实习过程总结

4.1 分工与合作

小组分工合作的任务安排：

算法策略：集思广益共同讨论

代码部分：方静潮、王述冰、李春江

论文部分：陈铮、娄宇*

合作交流的方式：微信群讨论、线上会议

右图是一次组会记录：

4.2 经验与教训

经验：首先是小组的分工，在这样一个特殊时期组员们无法面对面合作的时候体现的尤为重要。不能太宽泛也不能太具体，首先，如果太宽泛，可能使得每个人都无所适从不知道干什么好，无法形成有效的合作；反之，太具体可能使得组员的思维受到限制，不能集思广益，因此我们组采取了“办法一起想”、“代码和论文先由一人写框架，再由全组提意见和完善”的策略，取得了良好的效果。

教训：在普通的 2048 中有一些已有的经验和想到的策略可能效果并没有想象中的那样有效，只有实践才能区分是好的还是不好的决策。我们组开始采用的是让自己的大数居于角落希望最终形成梯度这样一种相对保守的策略，但是事实证明（也是我们组在研究天梯排名较高的组的决策后达成的一致意见）将大数排成一堵“墙”，用来堵截对方，这样一个富有进攻性的策略才是较好的策略，这样修改以后我们程序的强度也大大提高。但是由于开始走了许多弯路，调参时间略有不足，没有在参数上充分挖掘出算法的强度，这是比较遗憾的一点。

4.3 建议与设想

建议：首先对于游戏本身和天梯对战的方式，真的已经非常好了。另外一点建议就是：在决赛的赛制上，时间允许的前提下希望可以进行循环赛，因为代码本身有相互的克制关系，这样随机匹配的赛制可能使得运气成分增大。

对学弟学妹的寄语：数算课很硬核，一学期认真学下来能学到不少东西，大作业也是一段难忘的记忆，只是希望你们不要再卷了。

后续工作的大胆设想：来一场惊险刺激的亚马逊棋？

5 致谢

感谢陈斌老师一个学期以来的指导
感谢所有助教老师一学期为我们提供的帮助
感谢在决赛中与我们组展开精彩对战的 Uniform 组、X-Ray 组
感谢全组的成员们，我们为彼此而骄傲

6 参考文献

[1] 《自治系统 AI-CS814 分配——蒙特卡罗树搜索和平均模拟搜索应用于流行的网络游戏‘1048’》.弗兰克・米切尔.MSC 人工智能于应用

数据结构与算法课程实习作业报告

戴典 刘家正* 罗富昌 张健霖 张舒涛

摘要：本算法通过对 2048 棋盘估值并通过内部模拟下棋和合并过程以达到返回最优解的目的。在旧版本算法中，我们使用了基于 bytearray 和 list 的线性结构用于存储棋盘与估值权重并辅以较复杂的估值；在新版本算法中我们使用了基于 list 的线性结构存储双方棋子辅以较简单的估值。多次实验结果表明新版本在指定时间内显著优于旧版本算法。

关键字：DFS 剪枝算法 2048 估值

1 算法思想.....1
1.1 总体思路.....1
1.2 算法流程图.....2
1.3 算法运行时间复杂度分析.....2
2 程序代码说明.....3
2.1 数据结构说明.....3
2.2 函数说明.....3
2.3 程序限制.....6
3 实验结果.....7
3.1 测试数据.....7
3.2 结果分析.....12
3.3 经典战局（可选）.....15
4 实习过程总结.....16
4.1 分工与合作.....16
4.2 经验与教训.....18
4.3 建议与设想.....18
5 致谢.....19
6 参考文献.....19

1 算法思想

1.1 总体思路

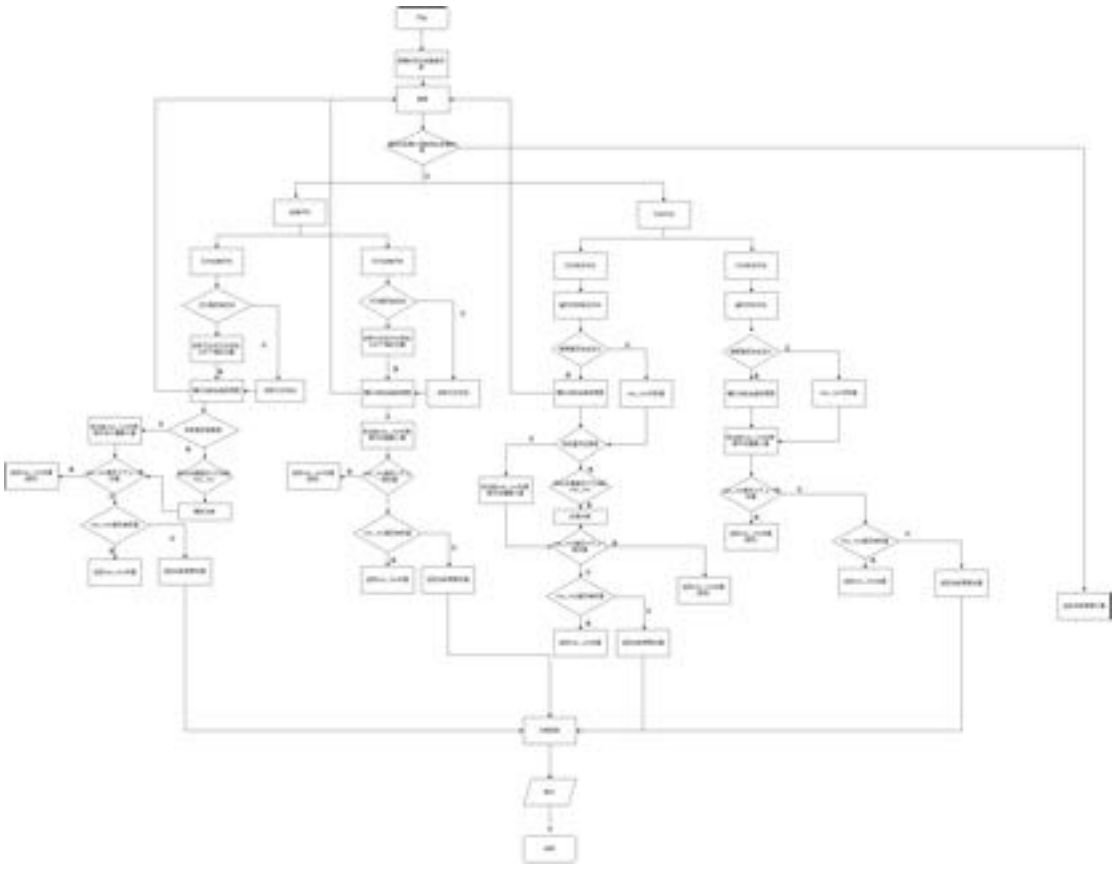
本组此次 2048 大作业所采用的算法主要为 min-max 剪枝，对于每一种可能决策进行局面的评估，最后选择估值最大的走棋方案。在主要搜索函数 move_search 中，分为四个阶段：我方落子，敌方落子，我方合并，敌方合并。在每一个步骤下都相应的进行 min-max 剪枝，最后综合选取最佳策略。

关于具体的局面估值函数则由 evl 函数给出。而其中的局面 value 的计算参数主要

包含我方拥有棋子的价值以及棋盘局面的自由度，二者与 value 均为正相关。由于 2048 游戏规则本身的特殊性，数值越大的棋子所代表的价值也相应地越高，故在将棋子价值计入 value 时我们也将其值进行非线性化处理以凸显数值更大的棋子所具有的更高价值。

本组算法再进行可能决策的搜索时所设定的最高搜索层数为 10 层，而考虑到在局面过于简单时搜索工程较为复杂，以及所耗的时间，具体搜索策略也将取决于走棋时的具体情况，即剩余时间以及当前回合数两个参数。在搜索过程中为了有效降低计算运行时间，我们的算法也进行了一定的剪枝算法，通过回溯对比的方式省去不必要的可能决策的运算。

1.2 算法流程图



1.3 算法运行时间复杂度分析

算法运行时间主要由对于决策树上每一种可能决策进行估值构成，考虑每层 n 种情况，总共 m 层，估算时间复杂度的大 O 数量级约为 $O(n^m)$。在实际运行中，由于剪枝算法的存在，时间开销可能低于估算值。

2 程序代码说明

2.1 数据结构说明

与本代码有关的数据结构有：

前期：

基于 bytearray 的线性结构——用于高效率处理棋盘

基于 list 的评估权重——用于代表对应位置的棋子权重

基于 DFS 的搜索树——用于搜索可能的步法

后期：

基于 list 的结构——用于存储双方已有的棋子

基于 DFS 的搜索树——用于搜索可能的步法

2.2 函数说明

前期：

1. Evl

   根据综合多个(共 6 个)矩阵给出的权重评估棋盘，先手方为正，后手方为负。矩阵四次旋转寻找最为合适的方向评估。并增加两个矩阵用于选择进攻/防守，调整比重可调整进攻/防守的趋向性。调整此函数可改变棋子的有序性。

2. Smooth_Evl

   光滑度评估函数，先手比后手光滑为正，后手比先手更光滑为负。等级越高的棋子权重越大，若相邻为空格则不计入光滑度计算。调整此函数可改变棋子的合并难度。

3. Free_Evl

   剩余空间评估函数，剩余空间越多，返回值越大。调整此函数可改变合并的激进程度。

4. Threat_Evl

   威胁度评估函数。评估下次合并之时，同一行的先手方有可能吃掉后手方棋子，抑或是同一行的后手方有可能吃掉先手方棋子。棋子越大权重越大。调整此函数可预防白给并帮助进攻。

5. Get_Evl

   将 Evl Smooth_Evl Free_Evl Threat_Evl 按权重相加并返回评估值

1. __init__

   初始化 Player 类，其中包括：

   self.isFirst 用于存储是否先手，先手为 1，后手为 0

   self.array 用于存储系统所给随机下棋位置

   self.currentRound 用于存储当前局数

   self.round_mode 用于存储需要作出的决策状态

   self.answer 用于存储经过搜索后得出的答案

   Ps：这其中的罗列实际上只是为了更清晰地体现数据内容，answer 值并不需要

在 init 函数中定义
self.end_depth 用于指定最大搜索深度
self.ROWS = 4 行数
self.COLLUMS = 8 列数
self.mode_offset 搜索模式偏移量，先手为 1 后手为-1
self.where self._where 为我方位置，1 说明我方在左，0 说明我方在右

Ps：注意到棋盘实现代码中的具体内容，是否在左和是否先手值相同，虽然这在用户文档中并未出现所以添加来方便兼容性，但实际上它们的存在并无必要意义。

6. able
   输入棋盘，返回己方可合并的方向，0、1、2、3 代表上下左右

7. able_attack
   返回己方可在对面边界放置棋子的位置

8. Beginning
   开局函数，通过在开局时尽可能地合并棋子，降低开局时的搜索时间并保证开局优势。

9. set_round_mode
   通过传入的参数判断当前的行动顺序并以 list 返回

10. get_next_mode
    通过传入的参数，根据 set_round_mode 判断下一步的行动模式并返回

11. get_available
    对于给定的 bitarray 棋盘，返回可以落子的所有位置，按编号以 list 类型返回

12. move_search
    以 DFS 为基础的搜索函数，通过搜索、剪枝返回指定深度的最佳结果

13. make_move_decision
    根据 time_remain 和 self.currentRound 决定搜索层数并存入 self.end_depth
    同时调用 move_search，进行 dfs 为基础的搜索

14. chessboard2mine
    将官方棋盘转换为以 bytearray 为基础的自定义棋盘，加快传参、copy 和合并的速度。

15. output
    输出函数，用于官方接口调用。（没错这只是个外壳）

16. switch
    传入 bytearray 棋盘、方向并返回移动后的棋盘

switch 实现过程：

传入参数：原版 bytearray 棋盘；移动方向；移动玩家（用真值表示该玩家是否先手）

切分：用切片操作，使用比较少的代码实现棋盘矩阵切分为横着的行或竖着的列

移动：对每一列（或行）调用对应方向的移动函数，使用字典结构调用函数省去名字，返回移动后的棋盘

合并：再次用切片操作合并分开的行列

~~自我感觉~~良好的方面：这个功能的全部实现（包括下面的辅助函数）使用了 120 行 似乎并不算非常多（看到过 400 行的实现方案）

17. move_row

可以移动一行棋子 ， 参数是棋盘，side 用来判断是否为我方领地 ，以及移动方是否为先手，作用：单元化地移动一行或一列棋子

move_row 实现过程：

使用四个条件判断语句囊括所有规则中提及的合并吞噬及移动规则

灵活使用了 bytearray 元素作为数的特性用以判断是否为友军

使用另设的列表来保护已经合并过的棋子不被合并

~~自我感觉~~良好的方面：将左移和上移用一个函数完成，减少代码书写量

18. move_up

参数：分割后的棋盘以及用于判断移动方是否为先手的真值，作用上移

关键是增加了一条判断语句用于判断移动方的领地位于何处，也就是 side 这一参量的确定

19. move_down

参数：分割后的棋盘以及用于判断移动方是否为先手的真值，作用下移

增加了一条判断语句用于判断移动方的领地位于何处，也就是 side 这一参量的确定

与 up 不同，下移函数因为需要改变移动方向，所以将列反转并上移以达到目的

20. move_left

参数：分割后的棋盘以及用于判断移动方是否为先手的真值，作用左移

增加了一条判断语句用于判断移动方的领地位于何处，也就是 side 这一参量的确定

21. move_right

参数：分割后的棋盘以及用于判断移动方是否为先手的真值，作用右移

增加了一条判断语句用于判断移动方的领地位于何处，也就是 side 这一参量的确定

与 left 不同，右移函数因为需要改变移动方向，所以将行反转并左移以达到目的

后期：

2. __init__ (self, isFirst, array)
   初始化 Player 类，其中包括：
   self.isFirst 用于存储是否先手，先手为 1，后手为 0
   self.array 用于存储系统所给随机下棋位置
   self.currentRound 用于存储当前局数
   self.answer 用于存储经过搜索后得出的答案
   self.end_depth 用于指定最大搜索深度
   self.mode_offset 搜索模式偏移量，先手为 1 后手为-1

3. output (self, currentRound, board, mode)
   传入当前局数，当前棋盘和当前行动模式，并调用 make_move_decision 进行搜索。对于传入 make_move_decision 的参数，move_mode 取数值 0，1，2，3 分别代表我方落子，敌方落子，我方合并，敌方合并

4. make_move_decision(self, chessboard, player_move_mode, time_left)
   根据 time_left 和 self.currentRound 决定搜索层数并存入 self.end_depth
   同时调用 move_search，进行 dfs 为基础的搜索

5. evl (self, board)
   根据自由度、棋子价值对己方棋盘作出评估。
   自由度越高棋局价值越高，己方拥有越大的棋子价值越高。

6. move_search (self, chessboard, player_move_mode, depth, last_max_min)
   从当前步数开始，使用 min-max 剪枝，对接下来的走棋方案搜索；
   若为先手，则按照 0123 的顺序搜索，若为后手则按照 0321 的顺序搜索，搜索到底层时返回评估值。
   此搜索不用调用 list 的 getitem，节省了时间。

2.3 程序限制

1. 搜索层数的限制

由于刚开局时可能性较多，因此若在开局时搜索过深会导致后程乏力，需要适当减小开局的搜索深度，加大棋局胶着时的搜索深度。同时，我们小组认为尽量增加己方优势是较好的处理方法，因此尽可能只搜索在己方落子的情况。

2. 搜索时间的限制

显然，平台对搜索时间有限制。因此，在接近超时的时候，我们大幅度降低搜索层数，此时搜索时间将大幅度降低，有效避免了超时的问题。

3. 评估函数的限制

由于规则较为简单，树的规模相对较小，因此，相比估值准确，搜索深度占有的价值更大。应当将评估函数控制在一定规模之下，以防止拖慢整体搜索速度，以便尽可能地加深搜索。

3 实验结果

3.1 测试数据

3.1.1 原始版本时间效率测试：

实验环境说明：

- 硬件配置：CPU:i7-9750H,RAM:8G
- 操作系统：Win10
- Python 版本：Python3.8
- 使用原始的 timeit 进行用时测试

【原始版本内用 bytearray 棋盘和官方棋盘用时测试】：

```
array=[random.randint(0,1000000) for _ in range(500)]

testplayer=Player(True,array)

testlist=[1,1,1,2,1,3,1,4]+[1,3,1,2,1,1,1,0]+[1,2,1,3,1,4,1,5]+[1,5,1,2,1,3,1,2]+[0,1,0,3,0,5,0,1]+[0

testbyte=bytearray(testlist)


testboard=Chessboard(array)
#index=4*x+y
for i in range(32):
    if testlist[2*i+1]!=0:
        if testlist[2*i]:
            testboard.board[(i%4,i//4)]=Chessman(True, (i%4,i//4), testlist[2*i+1])
            testboard.belongs[True].append((i%4,i//4))
        else:
            testboard.board[(i%4,i//4)]=Chessman(False, (i%4,i//4), testlist[2*i+1])
            testboard.belongs[False].append((i%4,i//4))
```

Pic.3.1.1-1 参数设置

```
def my_move(p):
    for i in range(4):
        p.switch(testbyte,i,True)

def board_move(b):
    for i in range(4):
        testboard.move(True,i)

mytime=timeit.Timer("my_move(testplayer)","from __main__ import Player,Chessboard,Chessman
oftime=timeit.Timer("board_move(testboard)","from __main__ import Player,Chessboard,Chessm

mt=mytime.timeit(number=100000)/100000
ot=oftime.timeit(number=100000)/100000
print("my move time used:",mt)
print("official move time used",ot)
```

Pic.3.1.1-2 move 方法用时测试代码

```
def my_copy():
    testboard.copy()

def of_copy():
    testboard.copy()

mctime=timeit.Timer("my_copy()","from __main__ import Player,Chessboard,Chessman,arr
octime=timeit.Timer("of_copy()","from __main__ import Player,Chessboard,Chessman,arr

mt=mctime.timeit(number=100000)/100000
ot=octime.timeit(number=100000)/100000
print("my copy time used:",mt)
print("official copy time used",ot)
```

Pic.3.1.1-3 copy 方法用时测试代码

```
def my_get_evl():
    Get_Evl(testbyte, 1)

mgtime=timeit.Timer("my_get_evl()","from __main__ import Playe
mgt=mgtime.timeit(number=100000)/100000
print("my matrix calcu time used:",mgt)
```

Pic.3.1.1-4 估值矩阵乘法用时

【多版本代码用时测试】

每次均从我方移动阶段开始搜索，每次轮数+10，时间-0.1s，一共搜 50 次

```
testlist=[1,1,1,2,1,3,1,4]+[1,3,1,2,1,1,1,0]+[1,2,1,3,1,4,1,5]+[1,5,1,2,1,3,1,2]+[0,1,0,3,0,5,0,1]+[0,3,0,5,0,1,0,1]+[
testplayers=[1:testplayer1,2:testplayer2,3:testplayer3,4:testplayer4,5:testplayer5,6:testplayer6,7:testplayer7,8:testp
testboard=Chessboard(array)
for i in range(32):
    if testlist[2*i+1]!=0:
        if testlist[2*i]:
            testboard.board[(i%4,i//4)]=Chessman(True, (i%4,i//4), testlist[2*i+1])
            testboard.belongs[True].append((i%4,i//4))
        else:
            testboard.board[(i%4,i//4)]=Chessman(False, (i%4,i//4), testlist[2*i+1])
            testboard.belongs[False].append((i%4,i//4))

def my_search(n):
    if n==1 or n==2:
        for i in range(50):
            testplayers[n].make_move_decision(testboard,"my_direction")
            testplayers[n].currentRound+=10
    elif n<7:
        for i in range(50):
            testplayers[n].make_move_decision(testboard,1,5.0-0.1*i)
            testplayers[n].currentRound+=10
    else:
        for i in range(50):
            testplayers[n].make_move_decision(testboard,2,5.0-0.1*i)
            testplayers[n].currentRound+=10
```

Pic.3.1.1-5 参数设置

```
print("-------test-------")
print("move_mode=my_direction or 3")
print("self.currentRound in range(0,500,10),time_left in range(5.0,0.0,-0.1)")

mstime1=timeit.Timer("my_search(1)","from __main__ import Player1,Player2,Chessboard,Chessma

mst1=mstime1.timeit(number=1)
print("my search time used totally(lfc version 1,first hand):",mst1)

mstime2=timeit.Timer("my_search(2)","from __main__ import Player1,Player2,Chessboard,Chessma

mst2=mstime2.timeit(number=1)
print("my search time used totally(lfc version 1,second hand):",mst2)

mstime3=timeit.Timer("my_search(3)","from __main__ import Player1,Player2,Chessboard,Chessma

mst3=mstime3.timeit(number=1)
print("my search time used totally(lfc version 2,first hand):",mst3)
```

Pic.3.1.1-6 部分测试代码

3.1.2 最终版本时间效率测试

• 实验环境说明

硬件配置: CPU:i5-8265U,RAM:8G

操作系统: Win10

Python 版本: Python3.6.2

• 测试一：算法时间复杂度

通过 ai 自己与自己对战测量用时，搜索层数从 1~12 进行递增，共进行 12 轮测试，记录数据并处理. (具体结果分析见 3.2.2 节)

测试代码:

```
import time
import random
from collections import namedtuple
Chessman = namedtuple('Chessman', 'belong position value')
ROWS = 4
COLUMNS = 8
class Chessboard:…
class Player:
    global level
…
    def make_move_decision(self, chessboard, player_move_mode, time_left):
        self.answer = None
        self.end_depth=levels
        self.move_search(chessboard, player_move_mode, 0, float('inf'))
…
time_list=[]
for i in range (12):
    levels=i
    random_list=[random.randint(1,100) for x in range(500)]
    player1 = Player(True, random_list)
    player2 = Player(False, random_list)
    board = Chessboard(random_list)
    start=time.time()
    for rd in range(500):
        board.add(True,player1.output(rd,board,'position'))
        board.add(False,player2.output(rd,board,'position'))
        board.move(True,player1.output(rd,board,'direction'))
        board.move(False,player2.output(rd,board,'direction'))
    time_list.append(time.time()-start)
print(time_list)
```

测试数据:

```
[0.04587912559509277,        0.8976101875305176,        2.001789093017578,
2.742339849472046, 3.606381893157959, 13.849903106689453, 25.19939160346985,
44.759679317474365, 67.12842035293579, 152.8799924850464, 473.7009701728821,
773.3973951339722]
```

• 测试二: 用时随轮数分配

通过最终版 ai 自己与自己对战测量用时，记录耗时随轮数的变化，共进行 10 轮测试取平均值，记录数据并处理. (具体结果分析见 3.2.2 节)

测试代码:

```
import time
import random
```

```
from collections import namedtuple
Chessman = namedtuple('Chessman', 'belong position value')
ROWS = 4
COLUMNS = 8
class Chessboard:…
class Player:…
random_list=[random.randint(1,100) for x in range(500)]
player1 = Player(True, random_list)
player2 = Player(False, random_list)
board = Chessboard(random_list)
time_used = []
start=time.time()
for rd in range(500):
    board.add(True,player1.output(rd,board,'position'))
    board.add(False,player2.output(rd,board,'position'))
    board.move(True,player1.output(rd,board,'direction'))
    board.move(False,player2.output(rd,board,'direction'))
    time_used.append(time.time()-start)
print(time_used)
```

3.1.3 最终版本估值函数测试

在最终版本中尝试采用过如下几个估值函数。

- 版本 1

```
    def evl(self,board):
        val=0
        my_sc=board.getScore(self.isFirst)
        your_sc=board.getScore(not self.isFirst)
        my_weight=2.5
        your_weight=2
        for me in my_sc:
            if len(my_sc)>0 and me==my_sc[-1]:
                val+=(my_weight**me)**2.5
            elif len(my_sc)>1 and me==my_sc[-2]:
                val+=(my_weight**me)**2
            elif len(my_sc)>2 and me==my_sc[-3]:
                val+=(my_weight**me)**1.6
            else:
                val+=(my_weight**me)**1.2
        for you in your_sc:
            if len(your_sc)>0 and you==your_sc[-1]:
                val-=(your_weight**you)**2.4
            elif len(your_sc)>1 and you==your_sc[-2]:
                val-=(your_weight**you)**1.9
            elif len(your_sc)>2 and you==your_sc[-3]:
                val-=(your_weight**you)**1.5
            else:
                val-=(your_weight**you)**1.1
        return val
```

此版本估值的思路为提升大子的权重、提升己方的权重（以使得 ai 以利己为主而不是以损人为主），在后续的测试中，我们发现此估值函数计算量相对较大，会导致后期搜索层数被迫减少。

- 版本 2

```
    def evl(self,board):
        val=0
        my_sc=board.getScore(self.isFirst)
        for me in my_sc:
```

```
            val += (2** me)** 2
        my_dof = 0
        for direction in range(4):
            next_board=board.copy()
            if next_board.move(self.isFirst, direction):
                my_dof += 1
        if my_dof == 0:
            val = val * 0.5
        elif my_dof == 1:
            val = val * 0.7
        return val
```

这个版本只保留了对己方棋子的计算以尽可能减少计算量，棋子基础估值采用棋子自身大小平方的方式，这样越大棋子的合并带来的估值增加就会越高。同时在与 Bot_SS 对战的过程中发现了“堵路”这种操作可能带来的影响，因此添加了合并自由度 dof 这个参数，在合并自由度小时，会相应的减少估值。

在天梯的测试中，版本 1 约达到 1100 分，版本 2 能达到近 1300 分，因此最后采用了版本 2.

3.2 结果分析

3.2.1 原始版本用时测试结果及分析：

各用时测试结果

```
my move time used: 0.000236627366
official move time used 0.00019758932099999999
```

Pic.3.2.1-1 move 用时测试结果

```
my copy time used: 2.5215519999999997e-06
official copy time used 2.3987630000000003e-06
```

Pic.3.2.1-2 copy 用时测试结果

```
my matrix calcu time used: 0.00014794560500000002
```

Pic.3.2.1-3 估值矩阵用时测试结果

```
-------test-------
move_mode=my_direction or 3
self.currentRound in range(0,500,10),time_left in range(5.0,0.0,-0.1)
my search time used totally(lfc version 1,first hand): 0.9624690999999999
my search time used totally(lfc version 1,second hand): 1.8412391000000001
my search time used totally(lfc version 2,first hand): 1.1195672
my search time used totally(lfc version 2,second hand): 1.4179532999999998
my search time used totally(ljz version 1,first hand): 1.0678425999999996
my search time used totally(ljz version 1,second hand): 1.4461649999999997
my search time used totally(ljz version 2,first hand): 1.0655798999999995
my search time used totally(ljz version 2,second hand): 1.3792466999999995
```

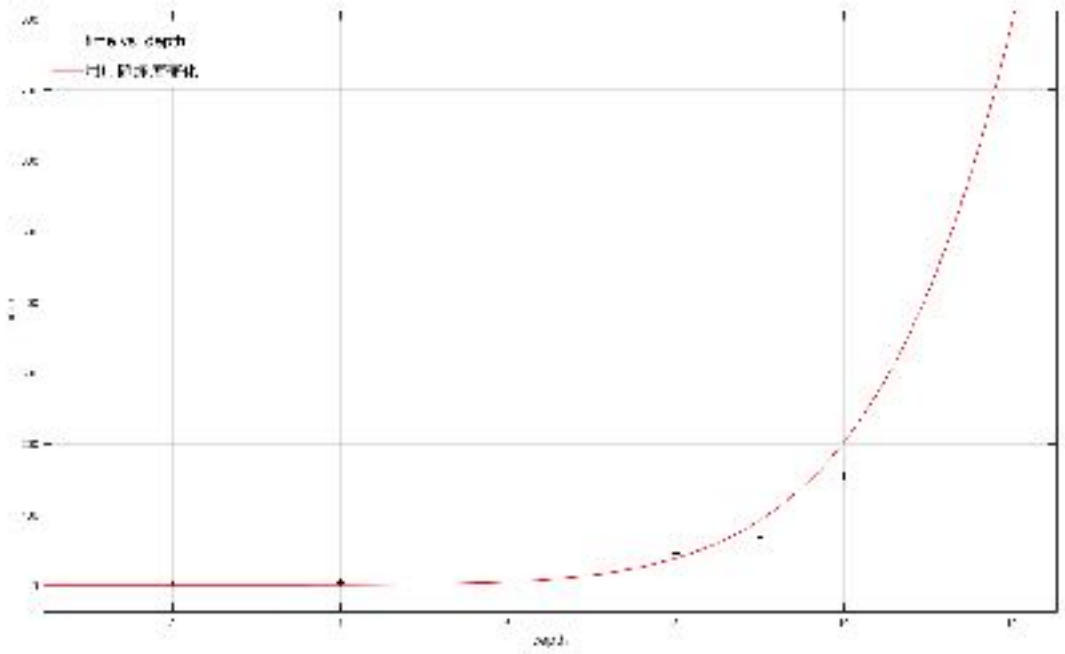
Pic.3.2.1-4 多版本搜索用时测试结果

首先我们发现我们自己内部采用的棋盘格式 bytearray 效果不佳，各种方法用时均比官方的 chessboard 慢一点点，且估值矩阵耗时甚多。在天梯里如果我们的代码不使用自己的棋盘而采用官方棋盘，我们可以搜到 8 层而不超时，但是如果我们采用自己的棋盘进行各种方法操作，至多只能搜到 5 层。考虑到我方代码在 position 阶段几乎不搜索，倾向于在己方趋于落子，于是每 4 层实际上有两层是不消耗算力的，而我方探索一个节点用时为（计算估值矩阵乘法用时+移动用时）≈0.0004s，而采用官方棋盘且不考虑估值矩阵计算的情况下探索一个节点用时约为 0.0002s。且考虑到我方有剪枝操作，多搜一次用时小于原来的 4 倍，并且认为 5→8 中间是两次 position，就可以说明为什么在天梯上我方初始代码只能搜 5 层，后来改用官方棋盘来操作能搜 8 层了。

3.2.2 最终版本时间效率测试结果分析

• 测试一（算法时间复杂度）结果分析

将数据使用 f(x) = a*x^b 的模型进行拟合，得到结果如下：

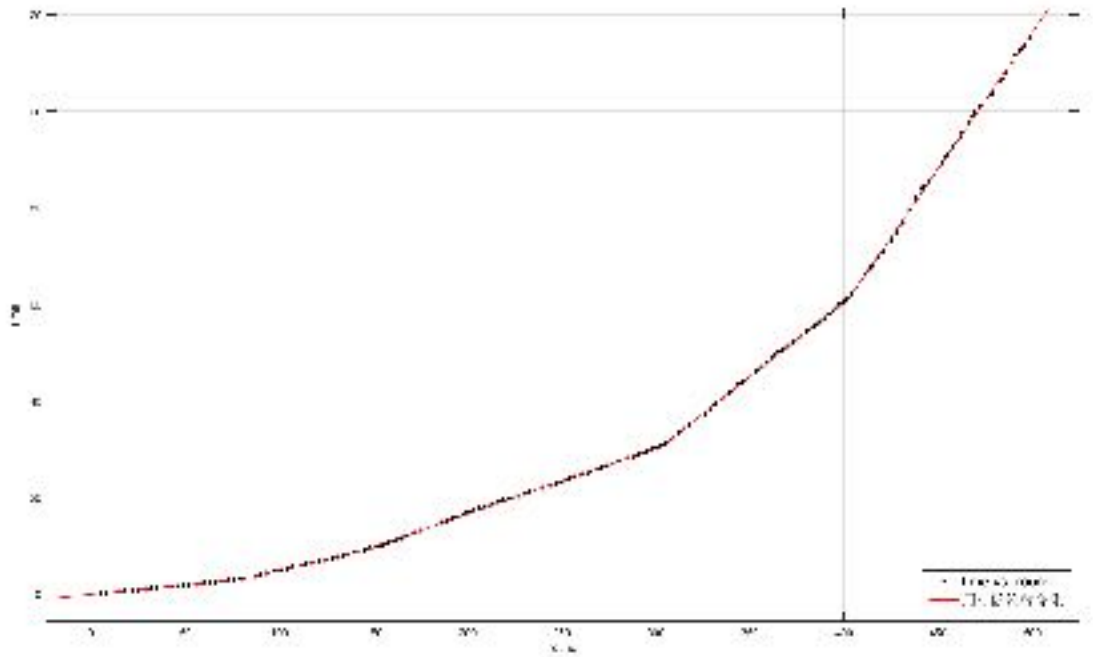


```
a =   6.437e-06  (-1.14e-05, 2.427e-05)
b =       7.496  (6.368, 8.624)
R-square: 0.9887
```

可见用时随深度变化基本符合指数函数增长规律，搜索时间复杂度为 $O(n^m)$.其中 n 为每层搜索的情况数，m 为层数

• 测试二(用时随轮数分配)结果分析

将数据使用 f(x) = (x<=80).*(a1*x+b1)+(80<x).*(x<=150).*(a2*x+b2)+(150<x).*(x<=300).*(a3*x+b3)+(300<x).*(x<=400).*(a4*x+b4)+(x>400).*(a5*x+b5)的模型进行拟合，得到结果如下：



```
a1 =     0.03954  (0.03729, 0.0418)
a2 =     0.09138  (0.08863, 0.09413)
a3 =      0.1363  (0.1355, 0.1372)
a4 =        0.31  (0.3084, 0.3116)
a5 =      0.5659  (0.5643, 0.5675)
R-square: 0.9999
```

可见用时在轮数分别为 0~80、81~150、150~300、300~400、400~500，即搜索深度分别为 6、7、8、9、10 时用时分段增加，最大的搜索层数 10 层耗时比最小的层数 6 层耗时多了约 14 倍。并且整个算法将大约一半的时间花在了 400 轮以后的搜索上，以尽量避免后期的失误葬送比赛。

在游戏的后期提高搜索深度也取得了一定的效果，对比一下均匀分配搜索时间（上）和随着游戏进程逐渐提高搜索深度（下）两种策略对战 2048Bot_S 的结果：

可见此策略的合理性。

3.3 经典战局（可选）

1. 在与 kilo 组的这场对战当中，我方为后手，吞并了对手的 2048.

可以看到我方的移动自由度在后期仍然保持的较好，在对手难以动弹时创造了这次吃棋机会，可以归功于估值函数中 dof 参数（说明见 3.1.3）的作用。

2. 在与 Bot_SS 的这场对战当中，我方为先手，成功在游戏快结束时合成 4096 赢下比赛.

这可以归功于搜索函数在 500 层出会直接返回估值，不作进一步搜索。因此在接近 500 层的时候 ai 会有尽快合并己方棋子的行为，避免出现虽然棋子总和大于对方但最高级棋子小于对方的情况。

4 实习过程总结

4.1 分工与合作

小组分工为：

刘家正（组长）：估值函数构造，优化代码速度，调参，实验报告 part2。

罗富昌：主体搜索函数，调参，carry，实验报告 part3(新)。

戴典：开局函数，实验报告 part1。

张舒涛：写内部 bytearray 棋盘相应方法，实验报告 part4+part5。

张健霖：整合函数，测试 Bug，测试代码耗时，实验报告 part3(旧)。

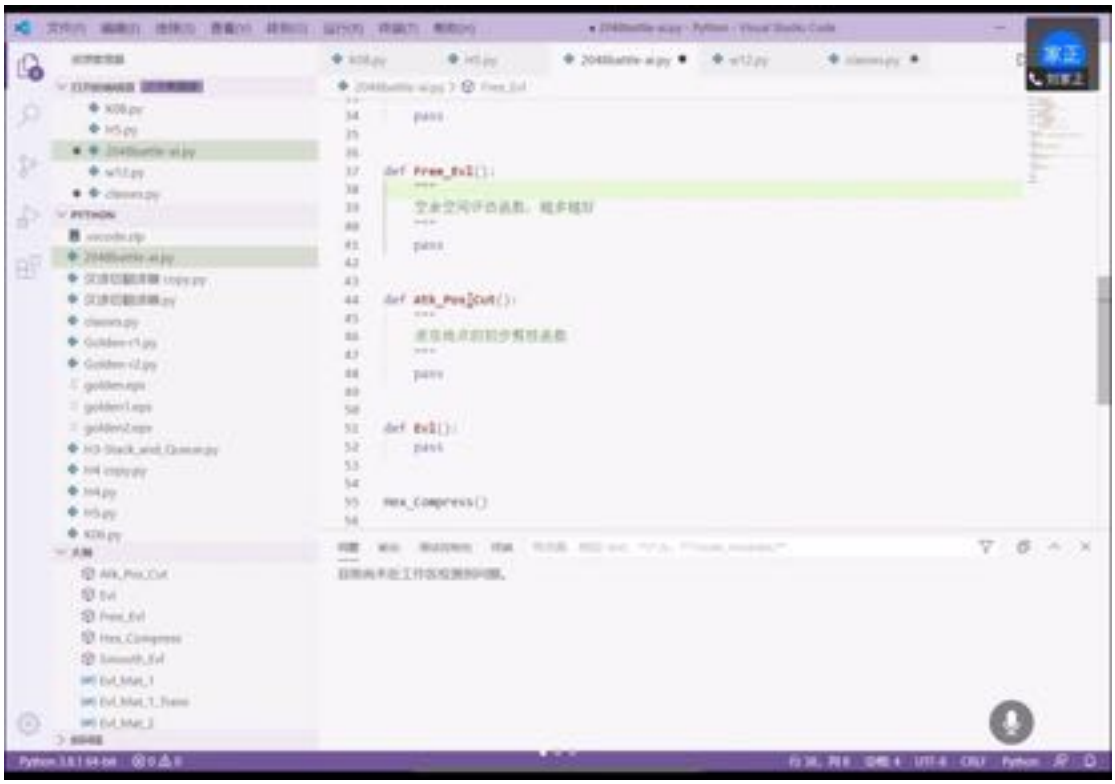
交流方式：

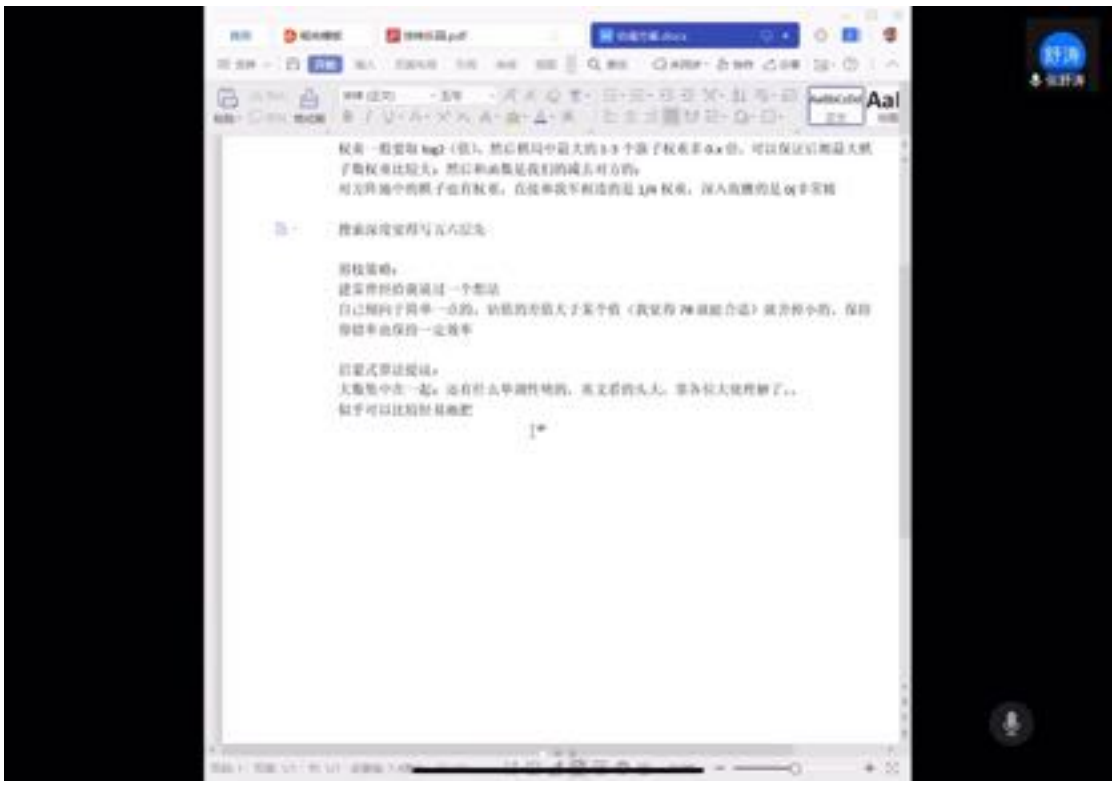
腾讯会议，微信群聊，私戳讨论。

腾讯会议两次：

第一次：确定策略函数框架，分工

第二次：统一规范格式





4.2 经验与教训

在最开始，我方同学均同意了刘家正同学提出了官方棋盘太慢了要自己内部写棋盘的观点，并且似乎大家都觉得这一点不需要证明，甚至内部棋盘可以比官方棋盘快很多，于是就有了写估值矩阵这样一件事的出现，并且张健霖同学下意识认为 4*4 的矩阵乘法不应该耗太多时间。于是经过长达一天的找 Bug 和 Debug 活动之后，我们的代码终于能正常跑了，但是在天梯上确是被吊着打，并且罗富昌和刘家正两位同学疯狂调参也无法改变被吊着打的结果。罗富昌同学用官方棋盘重写了一个搜索函数丢上天梯之后发现我们小组的代码不但更深了，还不超时了，在这一事实的冲击下，张健霖同学开始测试各种方法的时间，发现我们自己的棋盘方法每一个都比官方的慢，还有估值矩阵乘法这一个耗时大头，成功破案。

在看到 F19 冠军组的代码之后，我方觉得我们采用自己棋盘初始设想是对的，可以算是得意之处了，就是没有仔细考究效率差别以及如何提高效率，让内部自己实现棋盘这一操作变得有意义而不是在发现效率血慢后砍掉这一方案，屈服于官方方法。

实际上在早期的编程过程中就出现过大家认为的棋盘格式不匹配的情况，于是开了第二次腾讯会议来统一格式，这属于早期文档描述不清晰，预设不合理的锅，并且由于部分设置是组员间私戳讨论的，结果没有公开，于是就出现了分裂情况。这一教训告诉我们，公用的格式/接口的讨论应该放在公屏上，或者私戳完讨论决定后应当把结果告诉其他相应负责的人确定。

我方基本没有采用启发式算法，原因是后期内部达成了深度就是一切的共识，如果没有时间消耗十分之低且有奇效的启发式算法的话，不如改进自身算法效率来提升搜索深度，以及调参。（实际上是没有时间想好的启发式算法了，因为我方启动项目时间较晚，发现初始构想效率巨低后时间剩得已经不多了）

4.3 建议与设想

一些建议：

这次实习作业设置非常好，就是超时判负这一规定节目效果太足，建议以后继续使用，让后来人体会到同样的快乐。

基础设施代码建设很好，效率很高，自己难以写出效率高于官方棋盘的棋盘。

我觉得经过这次的尝试，一个重要的编程原则是灵活使用‘已有的’和‘熟悉的’数据结构，这样可以减少工作量，若希望自己进行优化，可不管少量使用的数据的空间时间优化（比如没必要把单个整型值改成字符类型以减少空间），应着重关注实现过程的时间复杂度。

寄语：

这门课有猫猫，任务又有趣，同学们都友好相处，共同进步，我们都超喜欢这里的。

我有一个大胆的想法：

在时间锁死为 5s 的情况下，复杂的启发式应该是起不到多大作用的（甚至会带来超时小惊喜），超深的搜索深度也很难做到，那就只能把希望放在玄学的炼丹上了！

5 致谢

在本次小组的通力合作中，尽管最后没有达到大家最期望的名次，但是大家都做出了自己的努力和贡献。大家在组会上各抒己见，积极发言（尽管本咸鱼可能在积极发“盐”，但）都做出了有建设性的，能够推动方案进一步完善的计划；还有深夜中，仍然沸腾的微信群。尽管大家在肝中借宿，甚至还轰走了一起住的头发，但大家无怨无悔（Doge）

我们可以看到刘家正组长积极地为程序的大方向确定了框架，做出了非常重要的提案，也获得了组员们的认同；罗富昌同学是代码提交前，最后的黑夜中肝到最后的勇士，深夜里还有他在群里广播着代码更新的实况令人敬佩。它们的工作都是非常重要的，在这里提出致谢。

6 参考文献

https://stackoverflow.com/questions/22342854/what-is-the-optimal-algorithm-for-the-game-2048

数据结构与算法课程实习作业报告

冷一飞* 夏罗生 卓睿 邓成志

摘要：我们使用博弈树来进行每一步的决策，采用负值极大搜索算法简化代码，alpha-beta 剪枝进行优化，根据胜利条件、最大值、空格数、单调性、总分数来确定估值。算法总体表现平稳，无超时情况。

关键字：博弈树；负值极大搜索；alpha-beta 剪枝

1 算法思想.....1
1.1 总体思路.....1
1.2 算法流程图.....2
1.3 算法运行时间复杂度分析.....2
2 程序代码说明.....3
2.1 数据结构说明.....3
2.2 函数说明.....3
2.3 程序限制.....9
3 实验结果.....9
3.1 测试数据.....9
3.2 结果分析.....9
4 实习过程总结.....10
4.1 分工与合作.....10
4.2 经验与教训.....11
4.3 建议与设想.....11
5 致谢.....11
6 参考文献.....11

1 算法思想

1.1 总体思路

概括的来说，算法可以分为两部分：放置棋子和倾斜棋盘。

放置棋子部分：考虑到胜负取决于双方的最大值比较，我们优先选择按照给出的随机序列落子在己方棋盘。当己方棋盘无处落子时，则需要在对方棋盘选择落子位置。经过实战考量，我们决定在对方棋盘随机选择位置。

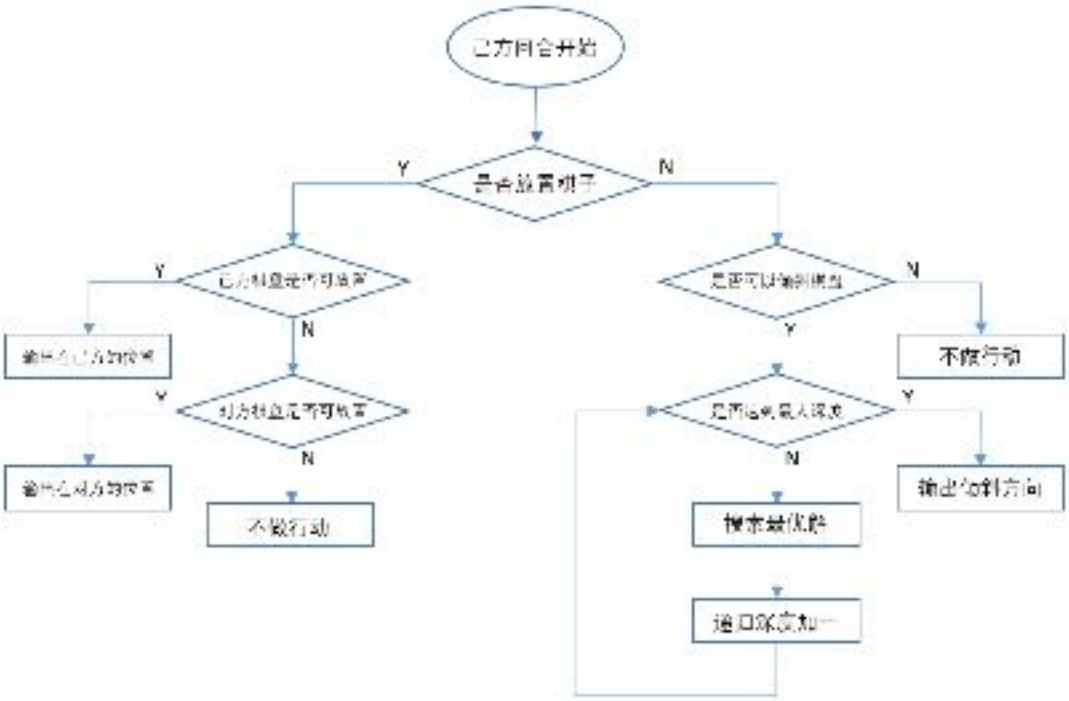
倾斜棋盘部分：首先设定一个参量标志倾斜后棋盘是否发生变化。若能发生变化，则对倾斜的方向展开深度优先搜索，寻找局面得分最大的行动，并采用 alpha-beta 剪枝进行优化。若所有方向均不能发生变化，则不采取行动。

我们设计的估值函数主要考虑以下几个因素：胜利条件、己方最大值、空格数、单调性、

总分数。给每一方面分别赋予一定的权重，其总和作为最终的得分。

为避免博弈树过于庞大，需要采用 alpha-beta 剪枝搜索优化。当某一节点需要从其子树中找寻极小值时，可通过已搜索的部分节点确定取值上限，倘若后续搜索节点大于该上限，则可直接将其所在子树剪去；反之亦然。这样一来，就减少了大量搜索时间，使得算法能够搜索更深。其中，我们还采用了一个小技巧，使极大值、极小值能够统一用一个函数求出，即“负值极大法”。其主要方法是，仅写一个求极大值的函数，同时可以应用于己方和对方，求出的分别是双方各自行动的极大值。只需将对手的极大值取相反数，便可以作为己方的极小值。这一想法来源于二元博弈的游戏性质。

1.2 算法流程图



1.3 算法运行时间复杂度分析

该算法的时间复杂度主要来源于搜索博弈树的开销，而 alpha-beta 剪枝可以对其时间复杂度进行很大的优化。即使是在节点排列最坏的情况下，时间复杂度也控制在 $O(B^D)$，其中 B 代表每一节点的子节点个数，D 代表搜索深度。随机排列的情况下，平均时间复杂度约为 $O\left(B^{\frac{3}{4}D}\right)$。在实际算法中，我们对搜索顺序进行了细致的调整，可将时间复杂度控制在 $O\left(B^{\frac{3}{4}D}\right)$ 以内，其中 $\bar{B}=3$，$D=8$，在实际运行过程中，不会出现超时的情况。

2 程序代码说明

2.1 数据结构说明

主要采用的是树的数据结构，每一个节点对应一个局面，每一条边对应一步操作，每一个父节点的子节点即对应当前局面模拟一步操作后所得到的新局面。每一个子节点都可以得到父节点一个可能的估值，而所有可能的估值中的最大值就是父节点的估值。

以初始局面为根节点，经过层层模拟得到的最终局面为叶节点。利用估值函数对叶节点估值后，逐层由子节点的估值得到其父节点的估值，并时刻注意剪枝以省去不必要的计算。最终得到根节点的估值，并得到当前所应选择的操作。

2.2 函数说明

2.2.0 给 player 类增加的属性值

```
class Player:
    def __init__(self, isFirst, array):
        # 初始化
        self.isFirst = isFirst
        self.array = array
        self.DEPTH=8 # 递归深度，由于算法的限制，深度应取值为偶数。
        self.next=None # 下一步操作
```

增加的属性值和其含义如图中注释

2.2.1 输出函数

```
def output(self, currentRound, board, mode):
    if mode=='position':
        own_pos=self.own_pos(self.isFirst,board,currentRound)
        eny_pos=self.eny_pos(self.isFirst,board)
        if own_pos!=0:
            self.next=own_pos
        elif eny_pos!=():
            self.next=eny_pos
        else:
            self.next=()
    if mode=='direction':
        self.negamax(self.isFirst,board,self.DEPTH,-99999999,99999999,mode,currentRound)
    return self.next
```

由实战经验可知，在大多数情况下，将棋子下在本方区域是最优的选择。并且由于我们可以下在对方区域的任意位置，所以如果我们要将棋子下在对方区域，则要进行多次搜索或者使用启发式规则。这将会较大程度地影响树的搜索时间，导致搜索深度变浅。

所以我们选择的策略为如果能在己方区域下棋，则优先在己方区域下棋，如果己方区域已满，则在对方区域下棋。至于如何决定在对方区域下棋的位置，我们曾经试图使用启发式规则来得到一个较好的位置，但是这样会使得搜索深度降低。经过实战权衡后，我们决定放弃启发式规则，在对方区域随机选取一个位置。

如果是要选择一个移动方向，则需要利用 negamax 函数进行递归搜索。

2.2.2 辅助函数

```
def own_pos(self,is_me,board,currentRound):
    # 返回一个在己方下棋的位置
    return board.getNext(is_me,currentRound)

def emy_pos(self,is_me,board):
    # 返回一个在对面下棋的位置
    blank_list=board.getNone(not is_me)
    if blank_list!=[]:
        return blank_list[0]
    else:
        return ()
```

2.2.3 主体函数 negamax

2.2.3.1：函数参数介绍

```
def negamax(self,is_me,board,depth,alpha,beta,mode,currentRound):
    # is_me取bool值，代表当前下棋方，is_me为True，则为先下的。
    # board即当前局面
    # depth为当前所搜索的深度，为0则说明到达叶节点。
    # alpha为当前局面估值的最小值，beta为当前局面估值的最大值。
```

函数的参数值已由注释介绍。

非常重要的是，此函数返回的是当前局面对于当前下棋方的估值。这样我们就可以用负值极大搜索算法进行递归。

2.2.3.2：基本结束条件

```
if mode=='position' and depth<self.DEPTH and is_me:
    # 此时递归进入了下一回合，回合数加一
    currentRound+=1

if depth==0 or currentRound>=len(self.array):
    # 递归结束条件：到达叶节点 或者 达到最大回合数
    return self.evaluation(board)
```

2.2.3.3：mode==position 时

```
if mode=='position':
    next_mode='position' if is_me else 'direction'
    # 决定下一个mode
    own_pos=self.own_pos(is_me,board,currentRound)
    emy_pos=self.emy_pos(is_me,board)

    if own_pos!=():
        next_step=own_pos
        copy_board=board.copy()
        copy_board.add(is_me,next_step)
        value=-self.negamax(not is_me,copy_board,depth-1,-beta,-alpha,next_mode,currentRound)
        if value>alpha:
            if depth==self.DEPTH:
                self.next=next_step
                # self.next即最终决定进行的操作
            if value>=beta:
                # 进行alpha-beta剪枝
                return beta
            alpha=value
```

首先根据当前下棋方的先后手决定下一个 mode，并由辅助函数得到可能的操作。如果己方棋盘未满，则优先在己方棋盘下棋，并使用复制的棋盘进行模拟操作得到新局面。

我们知道，每一个父节点的估值都是由其子节点的估值决定。故我们进行递归，对子节点使用 negamax 函数，非常重要的是，此时我们要在参数中进行一些变化。

（1） 更改操作方，此时函数的返回值也变为了新局面相对于新操作方的估值。

（2） 使用新局面。

（3） 搜索深度加一。

（4） 在父节点处继承 alpha-beta 值。由于操作方发生变化，故有 alpha（子节点）= -beta（父节点），beta（子节点）=-alpha（父节点）。

因为 negamax（子节点）函数返回值是相对于新操作方的估值，而 negamax（父节点）的返回值是相对于原操作方的估值。于是我们将子节点的返回值加一个负号，就变为相对于原操作方的估值 value，value 是父节点的一个可能的估值。

当 alpha<value<beta 时，value 则是一个可取的估值，并更新 alpha 值。

当 value>beta 时，说明父节点的一个可能估值就大于 beta 值，所以父节点的最终估值一定也大于 beta 值。所以这个父节点是无用的，此时进行剪枝，不再搜索这个父节点，直接返回 beta 值。

同时，当这个节点是根节点时，我们将当前操作更新到属性值 self.next 中。

```
elif any_pos!=():
    next_step=any_pos
    copy_board=board.copy()
    copy_board.add(is_me,next_step)
    value=-self.negamax(not is_me,copy_board,depth-1,-beta,-alpha,next_mode,currentRound)
    if value>alpha:
        if depth==self.DEPTH:
            self.next=next_step
            # self.next即最终决定下行的操作
        if value>=beta:
            # 进行alpha-beta剪枝
            return beta
        alpha=value
else:
    next_step=()
    value= self.negamax(not is_me,board,depth-1, beta, alpha,next_mode,currentRound)
    if value>alpha:
        if depth==self.DEPTH:
            self.next=next_step
            # self.next即最终决定进行的操作
        if value>=beta:
            # 进行alpha-beta剪枝
            return beta
        alpha=value

return alpha
```

当己方棋盘已满时，则选择在对方棋盘下棋。

当对方棋盘也满时，此时无法下棋，但游戏并未结束，继续向下递归。

2.2.3.4：mode==direction 时

```
if mode=='direction':
    next_mode='direction' if is_me else 'position'
    list2=[3,1,0,2] if self.isFirst else [2,1,0,3]
    has_moved=False

    for next_step in list2:
        copy_board=board.copy()
        if copy_board.move(is_me,next_step):
            has_moved=True
            value=-self.negamax(not is_me,copy_board,depth-1,-beta,-alpha,next_mode,currentRound)
            if value>alpha:
                if depth==self.DEPTH:
                    self.next=next_step
                if value>=beta:
                    return beta
                alpha=value

    if not has_moved:
        next_step=None
        value= self.negamax(not is_me,board,depth-1, beta, alpha,next_mode,currentRound)
        if value>alpha:
            if depth==self.DEPTH:
                self.next=next_step
            if value>=beta:
                return beta
            alpha=value

    return alpha
```

我们将可能的操作放在一个列表 list2 中。值得注意的是，我们在列表中将可能性更大的操作放在前面，此时可以促进剪枝，减少算法时间。

当四个方向都被堵住时，此时无法移动，但游戏并未结束，继续向下递归。

2.2.4 估值函数

```
def evaluation(self,board):
    # 估值函数
```

在负值极大值算法中，我们既要有要有对己方的估值，也要有对敌方的估值。但实际上，估值函数只需要应用于叶节点。当我们将搜索深度选取为偶数时，叶节点和根节点的操作方都为己方，所以我们只需要对于己方的估值函数即可。

下面分析估值函数中各部分的优缺点。

2.2.4.1

```
# 第一个部分：根据胜利条件，比较两方最大的棋子
# 如果最大棋子相同，则比较第二大的，直到比出胜负
# 最后取两棋子分数之差为price1
price1=0
own_list=board.getScore(self.isFirst)
emy_list=board.getScore(not self.isFirst)
while own_list!=[] and emy_list!=[] and own_list[-1]==emy_list[-1]:
    own_list.pop()
    emy_list.pop()
if own_list!=[] and emy_list!=[]:
    price1=2**own_list[-1]-2**emy_list[-1]
```

优点：在对战后期，虽然己方局面的单调性优于对手，但是没有将棋子进行合并，导致当前分数低于对手。这可以减少此类情况发生。

缺点：在对战前中期，过度合并，导致单调性降低。

2.2.4.2

```
# 第二个部分：最大值
price2=0
if board.getScore(self.isFirst)!=[]:
    price2=2**board.getScore(self.isFirst)[-1]
```

优点：促进高分棋子合并。

缺点：过度合并，导致单调性降低。

2.2.4.3

```
# 第三个部分：空格数
blanks=len(board.getNone(self.isFirst))
price3=blanks
```

优点：促进合并。让棋子进攻对方区域（这样也会给己方增加空格数）。

缺点：过度合并，导致单调性降低。阻止敌方区域的己方棋子撤回，导致己方棋子被吃。

2.2.4.4

```
# 第四个部分：单调性
price4=0
n=2
if self.isFirst:
    model=[[ 0, 1, 2, 3,4,n,n,n],
           [-1, 0, 1, 2,3,n,n,n],
           [-2,-1, 0, 1,2,n,n,n],
           [-3,-2,-1, 0,1,n,n,n]]

    for i in range(4):
        for j in range(8):
            if board.getBelong((i,j))==self.isFirst:
                price4+=2**board.getValue((i,j))*model[i][j]
        for j in range(4):
            if board.getBelong((i,j))!=self.isFirst:
                price4-=2**board.getValue((i,j))*model[i][j]

if not self.isFirst:
    model=[[n,n,n,4, 3, 2, 1, 0],
           [n,n,n,3, 2, 1, 0,-1],
           [n,n,n,2, 1, 0,-1,-2],
           [n,n,n,1, 0,-1,-2,-3]]
    for i in range(4):
        for j in range(8):
            if board.getBelong((i,j))==self.isFirst:
                price4+=2**board.getValue((i,j))*model[i][j]
        for j in range(4,8):
            if board.getBelong((i,j))!=self.isFirst:
                price4-=2**board.getValue((i,j))*model[i][j]
```

我们用二维表格来模拟整个棋盘，给每一个位置一个得分倍率。

单调性即让最大的棋子在角落，并且按照大小顺序环绕最大的棋子。

同时，我们奖励进入敌方阵营的棋子，所以将单调性扩展到敌方阵营内。但是当深入敌

方腹部后，很难接应己方棋子，非常容易被对方吃掉，故在敌方深处采用较低的倍率。

对于进入己方区域的敌方棋子，我们要对其进行惩罚，进行扣分。

2.2.4.5

```
#第五个部分：总分数
price5=0
score_list=board.getScore(self.isFirst)
while score_list!=[]:
    price5+=2**score_list.pop()
```

优点：由于总分数不会因为己方棋子合并变化，故可以保护己方棋子不被吃，并且吃掉对方棋子。

缺点：不促进合并，不促进单调性。

2.2.4.6

```
...
#第六个部分：平滑性
price6=0
if self.isFirst:
    for i in range(4):
        alist=[]
        # 空列表用来存储不为0的值
        for j in range(3,-1):
            if board.getValue((i,j))!=0:
                alist.append(board.getValue((i,j)))
        if len(alist)>=2:
            for k in range(0,len(alist)-1):
                price5-=abs(alist[k]-alist[k+1])
if not self.isFirst:
    for i in range(4):
        alist=[]
        # 空列表用来存储不为0的值
        for j in range(4,8):
            if board.getValue((i,j))!=0:
                alist.append(board.getValue((i,j)))
        if len(alist)>=2:
            for k in range(0,len(alist)-1):
                price5-=abs(alist[k]-alist[k+1])
...
```

由于平滑性需要耗费较多的时间，而且其影响较小，性价比较低，将其舍去。

2.2.4.7 加权得到总价值

```
price=price1*6+price2*2.5+price3*57+price4+price5*56
# 最终分数
return price
```

根据实战，分析失败的原因，并根据各个估值部分的优缺点进行改进。不断测试得到较好的加权值。

但是在天梯中，对战胜负偶然性较大。而且对于不同的对手策略，最佳的加权方式是不同的。可能对于下一个对手，这个加权方式就不适用了。这使得改进过程比较艰难。

2.3 程序限制

程序在对战中不会报错，不会出现超时。

3 实验结果

3.1 测试数据

实验环境说明：

- 硬件配置：i5-8300H/8G
- 操作系统：Windows 10
- Python 版本：Python 3.7

测试平台为代码竞技场。

历次热身赛结果如下表：

胜	负	平	胜率
1005	1803	2	35.8%
170	195	0	46.6%
357	163	0	68.7%
606	345	3	63.5%

3.2 结果分析

根据实战的结果，我们的算法在热身赛发挥平稳，不断提升，虽然没有特别亮眼的地方，却也没有非常大的缺点。

我们的估值函数中，空格数、单调性以及总分数三项影响效果最大，也是我们主要修改权重的参数。空格数确保了算法的进攻性，单调性促进合并并阻止己方棋子深入，总分数保护了己方棋子不被对方吃掉。早期代码中的平滑性一项也被实战证明会极大增加运行时间而影响较小，被我们舍弃。

但是由于测试时间略显不足，权重的选择不够完善，最终天梯赛的胜率在 50%左右。我们对于下在对方的棋盘内的操作也没有充分考虑，有时候自己会错失一些吃掉对方大棋子的良机；并且不会防范对方下在己方棋盘内的操作，导致己方大棋子被卡住，最终被吃。

4 实习过程总结

4.1 分工与合作

小组成员共四人，在紧张的学习生活中共进行了两次视频讨论：第一次一起试玩 2048 寻找策略灵感并查阅相关资料，确定算法的基本策略；第二次主要是针对已经写好的代码进行修改，同时分配实习报告撰写任务，邓成志同学因为临时有事未能参加。两次讨论期间我们在线上保持着算法的讨论与交流。

组长冷一飞同学负责组织讨论与汇总成果；代码主要由夏罗生同学编写，其他同学参与优化，其中卓睿同学对估值函数进行了关键的优化，极大提升了算法的性能；卓睿、邓成志同学负责进行网上测试，改进参数选择。实习报告由组内四人分工完成。

4.2 经验与教训

本次实习，虽然我们早早进行了线上视频讨论，但由于各位组员编程水平的限制，我们也没有充分重视本次实习，代码框架迟迟没有敲定。这也使得后期我们对代码的测试不够充分，最终提交比赛的代码也没有调试到很好的程度，这是需要反省的。

同时也是收获满满。四名小组成员在学习数据结构与算法课程前都没有 Python 基础，这给代码的初期编写阶段带来了不少的困难。我们积极地查找资料、在天梯上学习大佬的优秀算法，过程中也产生许多优秀的想法，不过很多都由于编程水平的限制而不得不放弃。我们最终的代码虽然没有打进淘汰赛，更比不上排名前列的大佬，但却是每一位组员辛勤工作的成果，已经突破了我们原有的编程水平，这对于我们而言已是最大的收获。

4.3 建议与设想

本次实习作业从组队、赛制到基础设施都安排得很好，可以看到陈斌老师和各位助教付出的心血。我们有一点小小的建议是希望实习结束后能开放参赛代码，给同学们学习优秀算法的机会，这对同学们的编程水平也是有提升的。

5 致谢

感谢陈斌老师和所有助教以及技术组的同学们！
感谢全组成员的辛勤努力！

6 参考文献

[1]https://kns.cnki.net/KCMS/detail/detail.aspx?dbcode=CMFD&dbname=CMFD201601&filename=1015376837.nh&v=MjIwMDRSN3FmWWVacUZDbmtVNzdPVkYyNkc3Qy9HTm5QcUpFYlBJUjhlWDFMdXhZUzdEaDFUM3FUcldNMUZyQ1U=
[2]https://www.jianshu.com/p/8376efe0782d

数据结构与算法课程实习作业报告

白纯迪 吕东澍 宋思潼 杨轩 刘行健*

摘要：利用评估函数对棋盘局面作出评估，通过负极大值算法对博弈树进行深度搜索，获取合法的最佳下棋决策和合并决策。运用 alpha-beta 剪枝提高搜索效率，进而增加搜索深度。通过本地和天梯比赛调整评估函数的评价指标、权重和搜索深度等参数，以提高胜率。

关键字：线性结构 负极大值算法 alpha-beta 剪枝 评估函数 面向对象

1 算法思想 2
1.1 总体思路 2
1.2 算法流程图 3
1.3 算法运行时间复杂度分析 6
2 程序代码说明 6
2.1 数据结构说明 8
2.2 函数说明 9
2.3 程序限制 13
3 实验结果 14
3.1 测试数据 14
3.2 结果分析 16
4 实习过程总结 17
4.1 分工与合作 17
4.2 经验与教训 21
4.3 建议与设想 32
5 致谢 32
6 参考文献 32

1 算法思想

1.1 总体思路

算法的总体思路是通过对棋局各方面性质的评价来估算双方棋盘各自的局面值，通过负极大值算法对博弈树进行深度优先搜索，在最优化对方决策的情况下，获得能使 n 步以后己方与对方局面值差值最大的己方决策。

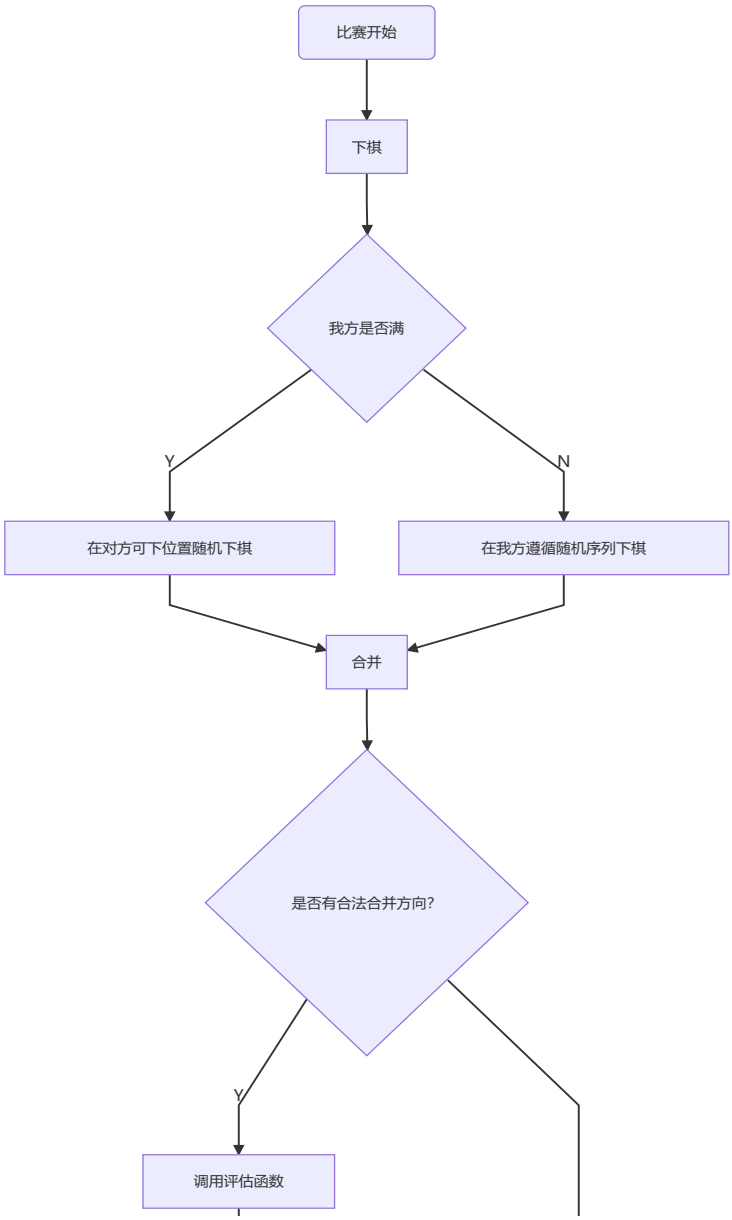
在数据结构上，采用栈来缓存做出变化前的棋盘（即搜索树父节点处的棋盘），采用列表来存储对各种可能的决策，并未用到“四叉树类”和“树节点”。总之，虽然算法本身是一种树形递归算法，但在代码实现时并没有采用树形结构，而是采用了简单的线性栈结构，原因在于，虽然一种局面下对应有四种决策分枝，但实际上可以用简单的列表遍历来实现子节点分枝的遍历，用栈来顺序存取父节点处的棋盘，所以实际上不需要创建树和节点类也可实现博弈树的功能。

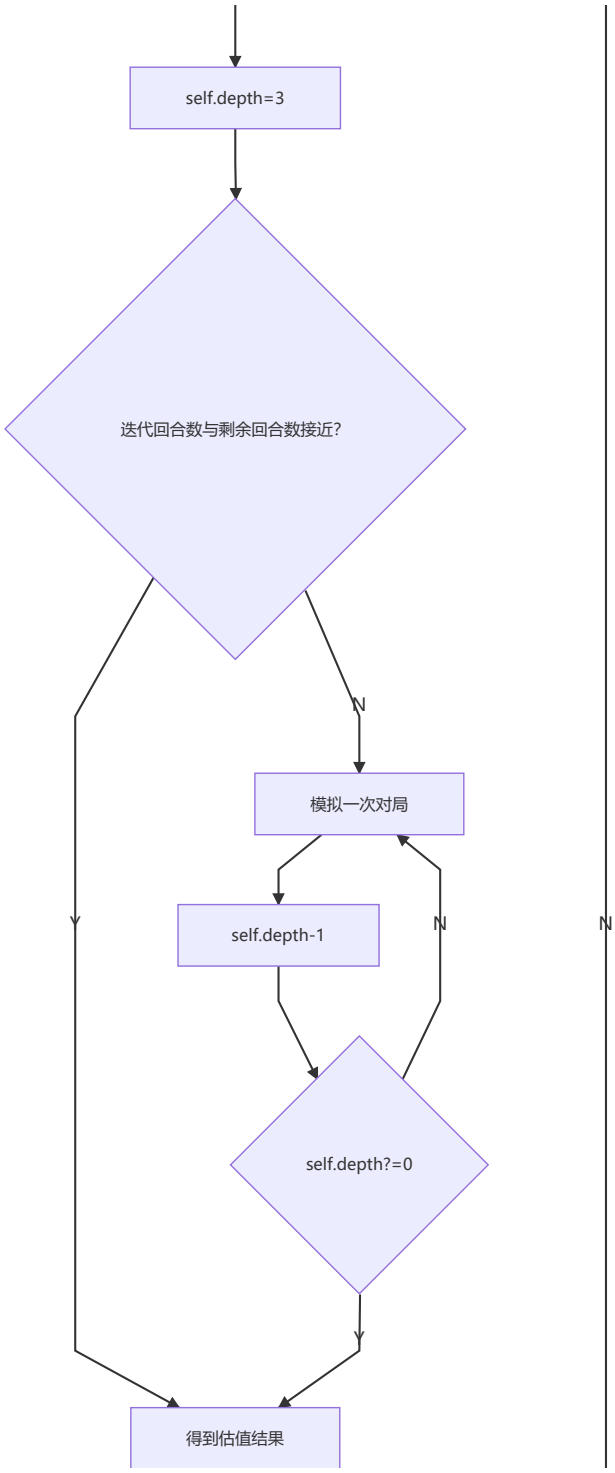
主要的算法策略是通过负极大值算法对不同决策分枝进行搜索，获取某一子层的双方局面值。搜索算法中应用 alpha-beta 剪枝以提高搜索效率。需要做出的决策分为两类：决策下棋位置和决策合并方向。

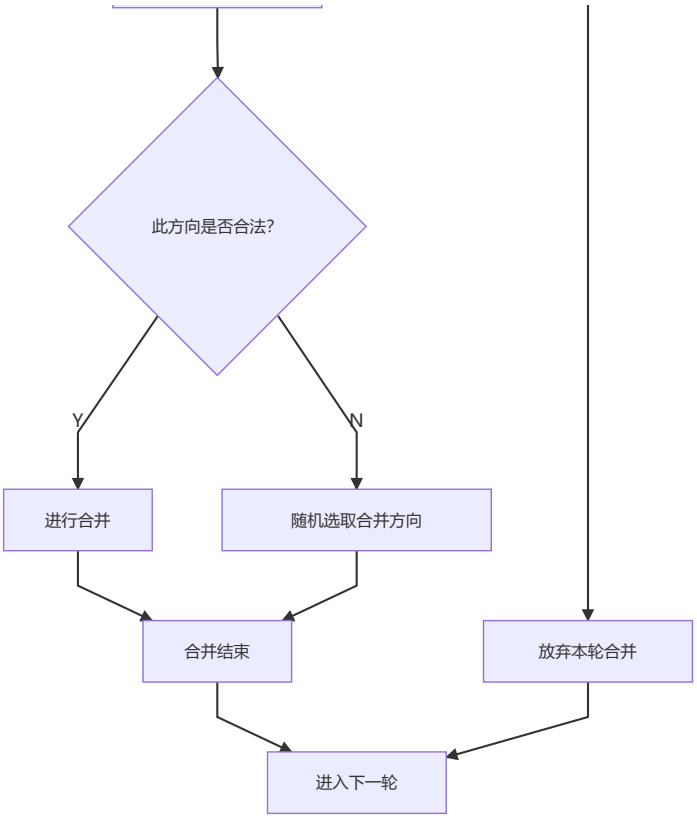
【1】 在选择下棋位置的决策时，我们采取较为防守的策略，只要我方棋盘未满，即在我方棋盘处的指定位置下棋，若我方棋盘已满，则随机选择对方棋盘处的空位下棋。这样选择的原因是，在大多数的情况下，选择在对方棋盘下棋的弊大于利；在少部分情况下，如“对方棋盘有较大同级别棋子有合并趋势”“对方棋盘将近填满”，我们也曾尝试编写函数来进攻对方棋盘以达到给对方添堵的目的，但由于函数的考虑并不足够周到，往往结果是为对方“送子”，或者并没有在最终达到“添堵”的目的，所以便放弃了这一想法

【2】 在选择合并方向的决策时，我们通过负极大值算法获取合并 3 步（包括对方合并）以后使得己方与对方局面值差值最大的合并方向。在搜索算法的实现中，会先验证该合并方向是否合法，若不合法则直接剪枝，遍历下一种决策。这样即可保证最终做出的决策一定是合法的。其中，合并 3 步是综合考虑搜索深度和超时因素得出的最优搜索深度。在后期优化的过程中，我们曾考虑过“即时优势”与“即时危机”的情况，编写相应的函数来进行即时进攻与防守。这种“即时决策”有别于深度搜索得出的最优决策，目的是为了防止通过深度搜索得出的最优决策使得当前的棋子陷于更大的危机或者放弃了当前更大的局面优势。但由于“即时决策”仍然面临着考虑不够周全的问题，其“掐断式”写法更是使得大部分情况下的递归搜索和评估函数失去了意义，算法变得目光短浅，胜率一直“居低不上”，所以最终还是去掉了即时决策的编写，回归最朴素的搜索算法。

1.2 算法流程图


比赛开始
下棋
我方是否满
Y
N
在对方可下位置随机下棋
在我方遵循随机序列下棋
合并
是否有合法合并方向?
Y
调用评估函数


self.depth=3
迭代回合数与剩余回合数接近?
N
模拟一次对局
self.depth-1
N
self.depth?=0
Y
Y
得到估值结果
N


此方向是否合法?
Y
N
进行合并
随机选取合并方向
合并结束
放弃本轮合并
进入下一轮

1.3 算法运行时间复杂度分析

由于场上的棋子数、空格数的数量级与棋盘总个数相等，因此可认为 N 为棋盘总格数。

1.figureValue 函数：

```
def figureValue(self, isFirst):  # 将对象方棋子的级别加权求和,得到棋子级别大小和数量提供的 value
        selfside = isFirst
        selfscore = self.board.getScore(selfside)
        figurevalue = 0
        for i in selfscore:
            if i >= 7:  # 级别太低也没有加的必要
                figurevalue += 3 ** i
            elif i >= 5:
                figurevalue += 2 ** i
            elif i >= 3:
                figurevalue += i * 2
        return figurevalue
```

计算每个棋子的 value 并加入到 figurevalue 中，时间复杂度为 O(N)。

因此总的时间复杂度为 O(N)。

2.emptyCellsValue 函数：

```
def emptyCellsValue(self, isFirst):  # 计算对象方空位数所提供的 value
    selfSide = isFirst
    selfEmptyCellsNums = len(self.board.getNone(selfSide))  # 对象方空位数
    emptyCellsValue = selfEmptyCellsNums**0.5
    return emptyCellsValue
```

取出对象方棋盘内空格数的时间复杂度为 O(1)，计算 emptyCellsvalue 的时间复杂度为 O(1)。

因此总的时间复杂度为 O(1)。

3.monotonicityValue 函数：

```
def monotonicityValue(self, isFirst): # 单调性 value 评估，仅考虑对象方棋盘内的棋子，且对方棋子入侵时，其所在的行与列的 value 贡献均为 0
        selfSide = isFirst
        monotonicityvalue = 0
        # 遍历对象方棋盘每一行
        for row in range(4):
          ......
        # 遍历对象方棋盘每一列，同上
        for col in range((1 - isFirst) * 4, (1 - isFirst) * 4 + 4):
          ......
        return monotonicityvalue
```

遍历对象方棋盘上的格子，时间复杂度为 O(N)。

因此总的时间复杂度为 O(N)。

对于总的评估函数：

```
def estimateValue(self, isFirst):  # 计算先手或后手方的棋面 value
      monotonicityWeight = 0.23  # 单调性权重
      emptyCellsWeight = 0.21  # 空格数权重
      figureWeight = 0.42  # 棋子级别权重 0.42

      value = monotonicityWeight * self.monotonicityValue(isFirst) + \
            emptyCellsWeight * self.emptyCellsValue(isFirst) + \
            figureWeight * self.figureValue(isFirst)
      return value
```

调用了函数 1～3，因此时间复杂度为其中最高的 O(N)。

由于 minmax 搜索与剪枝函数搜索深度为 3，每层搜索取四个方向，每个方向调用一次评估函数。因此，算法的总时间复杂度为 $O(N^3)$。

2 程序代码说明

2.1 数据结构说明

【1】 扩充了 Player 类的属性：

self.board:记录当前操作时的棋盘，为 Chessboard 类

self.best_position 和 self.best_direction：记录得出的最佳决策

self.depth：记录搜索深度

self.another：记录轮到己方下棋时在己方棋盘的指定下棋位置

self.current_round：记录当前回合数，在递归搜索时需要更新

self.prev_stack：记录“父棋盘”的存储栈

```
class Player:
    def __init__(self, isFirst, array):
        # 初始化
        self.isFirst = isFirst
        self.array = array
        self.board = None
        self.best_direction = None
        self.best_position = None
        self.depth = 3
        self.another = None  # 轮到己方下棋时，着下在己方棋盘则应该下的位置
        self.current_round = 0  # 表示当前进行的回合数
        self.prev_stack = Stack()  # 用于回溯之前的棋盘，即用于cancel move
```

将这些随递归搜索而改变的量存储为 Player 类的属性，既可以避免变量作用域的标注与分辨，又可以减少向功能函数传入的参数的数量，使得程序更加简洁明了。

【2】 应用栈来存取“父棋盘”：

在递归搜索时，由于调用 add 方法和 move 方法会改变棋盘的状态，所以每次在对棋盘做出相应改变前，都要将当前棋盘存入栈中。在此分枝搜索结束，遍历下一种决策前，pop 出栈顶的“父棋盘”并赋值给当前棋盘 self.board，以达到恢复棋盘状态（即“取消变化”）的目的。

```
for direction in directionList:
    current_board = self.board.copy()
    if self.board.move(isFirst, direction):  # 此合并方向合法时make move
        self.prev_stack.push(current_board)
        # 进行递归
        if isFirst:  # 本层player为先手方，即下一步是另一方选择方向进行合并
            value = -self.get_ChoiceValue(depth - 1, -beta, -alpha, -player, not isFirst, 'direction')
        else:  # 本层player为后手方，即下一步是另一方选择位置下棋
            self.current_round += 1  # 下一步就是下一回合了
            value = -self.get_ChoiceValue(depth - 1, -beta, -alpha, -player, not isFirst, 'position')

        self.board = self.prev_stack.pop()  # cancel move
```

2.2 函数说明

【1】choose 接口：为了尽量简化 output 接口里的代码，设置两个 choose 接口分别返回最终的下棋位置决策和合并方向决策。

```
def choose_position(self):
    # 优先进攻，其次下在自己这里，其次下在对方那里（具体下在哪个位置？需要决策！）
    self.best_position = self.another
    if self.best_position != ():
        return self.best_position
    else:  # self.another可能为()（己方棋盘满了）
        available = self.board.getNone(not self.isFirst)  # 对方的允许落子点
        if not available:  # 整个棋盘已满
            return None
        else:
            from random import choice
            return choice(available)

def choose_direction(self):
    self.best_direction = None
    self.get_ChoiceValue(self.depth, -1000000000, 1000000000, -1, self.isFirst, 'direction')
    if self.best_direction is not None:
        return self.best_direction
    # 若迭代深搜得出的最佳方向是非法的，则随机选择
    from random import shuffle
    directionList = [0, 1, 2, 3]
    shuffle(directionList)
    for direction in directionList:
        if self.board.move(self.isFirst, direction):
            return direction
```

【2】get_ChoiceValue 接口：递归搜索树。

通过调用该函数，为 self.best_direction 赋值；函数返回本层决策的最大 value 值。

```
# 运用negamax算法和alpha-beta剪枝获取这一步决策的value值
def get_ChoiceValue(self, depth, alpha, beta, player, isFirst, mode):
```

player 参数标注当前玩家：-1 为己方，1 为对方

isFirst 参数标注当前玩家是否为先手：True 为先手，False 为后手

由于递归搜索时为双方交替决策，故下一层递归调用时需取为-player 和 not isFirst

• 递归主体：按当前需要做出决策的类型（mode）进行分类

1. 当这一步需要做出的决策是合并方向时：

   Step 1: 构造列表存放所有可能的方向进行遍历。

   这里方向的存放顺序与先后手有关，保证在遍历得到的局面值相同时优先选择向己方棋盘的“后方”合并，以达到防守的目的。

```
if isFirst:
    directionList = [0, 1, 3, 2]
else:
    directionList = [0, 1, 2, 3]

for direction in directionList:
```

   Step 2: 判断该 direction 是否合法，以保证后续得到的最优方向是合法方向中的最优决策。

```
for direction in directionList:
    current_board = self.board.copy()
    if self.board.move(isFirst, direction):  # 此合并方向合法时make move
        self.prev_stack.push(current_board)
        # 进行递归
```

```
    else:  # 此合并方向非法时，继续尝试下一个方向
        self.board = current_board  # if判断时移动了盘面，应及时恢复盘面
        continue
```

Step 3：合并方向合法时，进行递归。根据本轮 player 是先手还是后手来决定下一层递归调用时的 mode 类型。由于为负极大值算法，本层取下一层的相反数作为本层该节点处的 value 值。

需要注意的是，若本轮 player 为后手方，则下一轮为下一回合的开始，需更新 self.current_round。

```
# 进行递归
if isFirst:  # 本层player为先手方，即下一步是另一方选择方向进行合并
    value = -self.get_ChoiceValue(depth - 1, -beta, -alpha, -player, not isFirst, 'direction')
else:  # 本层player为后手方，即下一步是另一方选择位置下棋
    self.current_round += 1  # 下一步就是下一回合了
    value = -self.get_ChoiceValue(depth - 1, -beta, -alpha, -player, not isFirst, 'position')
```

Step 4: 本分枝深度搜索结束后，恢复棋盘，并选出本层决策的最大 value 值返回。若本层为第一层，则 value 值最大的决策即为所选的最佳决策。

```
    self.board = self.prev_stack.pop()  # cancel move

    # 进行alpha-beta剪枝
    if value >= beta:
        if depth == self.depth:
            self.best_direction = direction
        return beta
    if value > alpha:
        if depth == self.depth:  # 回合数为现实回合时选择best direction
            self.best_direction = direction
        alpha = value
```

最终 return alpha

2. 当这一步需要做出的决策是下棋位置时：
   由于选择下棋位置的方法是以本方指定下棋位置为先，本方棋满后随机选择对方棋盘空位下棋，故直接将双方的决策各执行一遍后，又回到了选择合并方向的决策上。

```
    another = self.board.getNext(isFirst, self.current_round)
    if another != ():
        self.board.add(isFirst, another)
    else:
        available = self.board.getNone(not isFirst)
        if available:
            from random import choice
            self.board.add(isFirst, choice(available))
    isFirst = not isFirst  # 交换下棋方
    another = self.board.getNext(isFirst, self.current_round)
    if another != ():
        self.board.add(isFirst, another)
    else:
        available = self.board.getNone(not isFirst)
        if available:
            from random import choice
            self.board.add(isFirst, choice(available))
    isFirst = not isFirst  # 交换下棋方

    # 双方下棋结束，当前盘面为双方add后的盘面
```

之后代码部分与 mode=='direction'时完全相同。

- 递归结束条件：到达所定深度或当前回合数接近总回合数（500）
  当当前回合数接近总回合数剪掉搜索枝可以避免搜索到 500 回合以后的棋盘 array 缺失的 bug。

```
def get_ChoiceValue(self, depth, alpha, beta, player, isFirst, mode):  # isFirst表示当前player是否为先手(bool)
    if depth == 0:
        return self.get_BoardValue(player)
    if self.current_round >= len(self.array) - 3:  # 迭代到的回合数快超过总回合数时，剪掉搜索枝
        return self.get_BoardValue(player)
```

【3】 get_BoardValue 接口：返回某层的双方局面值之差（己方-对方）

```
def get_BoardValue(self, player):  # player=-1时为己方，player=1时为对方
    isFirst = self.isFirst if player == -1 else (not self.isFirst)  # 表示当前玩家是先手or后手
    selfvalue = self.estimateValue(isFirst)
    enemyvalue = self.estimateValue(not isFirst)
    return selfvalue - enemyvalue
```

【4】 estimateValue 接口：返回某方棋盘的局面值。isFirst 表是否为先手方，与是否为己方无关。

```
def estimateValue(self, isFirst):  # 计算先手或后手方的棋面value
    monotonicityWeight = 0.23  # 单调性权重
    emptyCellsWeight = 0.21  # 空格数权重
    figureWeight = 0.42  # 棋子级别权重0.42

    value = monotonicityWeight * self.monotonicityValue(isFirst) + \
            emptyCellsWeight * self.emptyCellsValue(isFirst) + \
            figureWeight * self.figureValue(isFirst)
    return value
```

【5】 三个评价指标接口：

1. monotonicityValue：单调性评价指标

```
# 单调性value评估，仅考虑对象方棋盘内的棋子，且对方棋子入侵时，其所在的行与列的value贡献均为0
def monotonicityValue(self, isFirst):
    selfSide = isFirst
    monotonicityvalue = 0
    # 遍历每一行
    for row in range(4):
        # 引入单调性因子，表征单行或者单列的单调性好坏
        monotonicityFactor = 0
        # 列数从对象方棋盘的最左列到最右列
        col = (1 - isFirst) * 4
        while col < (1 - isFirst) * 4 + 3:
            # 当前格子的级别
            currentValue = self.board.getValue((row, col))
            # 下一格的坐标
            nextrow = row
            nextcol = col + 1
            # 有内鬼，则终止交易，转至下一行
            if selfSide != self.board.getBelong((row, col)) or \
                    selfSide != self.board.getBelong((nextrow, nextcol)):
                monotonicityFactor = 0
                break  # 只是跳出while循环
            # 无内鬼，计算下一格的级别，单调性因子变化
            else:
                nextValue = self.board.getValue((nextrow, nextcol))
                '''此表达式可以更改'''
                monotonicityFactor += nextValue - currentValue
            # 列数加一，继续下一格
            col += 1
        # 此行的单调性因子计算完毕，加至value中
        '''此表达式可以更改'''
        monotonicityvalue += abs(monotonicityFactor)
```

```
    # 遍历对象方棋盘每一列
    for col in range((1 - isFirst) * 4, (1 - isFirst) * 4 + 4):
        monotonicityFactor = 0
        row = 0
        while row < 3:
            currentValue = self.board.getValue((row, col))
            nextrow = row + 1
            nextcol = col
            if selfSide != self.board.getBelong((row, col)) or \
                    selfSide != self.board.getBelong((nextrow, nextcol)):
                monotonicityFactor = 0
                break
            else:
                nextValue = self.board.getValue((nextrow, nextcol))
                monotonicityFactor += nextValue - currentValue
            row += 1
        monotonicityvalue += abs(monotonicityFactor)
    return monotonicityvalue
```

2. emptyCellsValue：空格数评价指标

```
def emptyCellsValue(self, isFirst):  # 计算对象方空位数value
    selfSide = isFirst
    selfEmptyCellsNums = len(self.board.getNone(selfSide))  # 对象方空位数
    '''此处计算式可更改，但是应有：在空格数较少时，value对空格数的导数较大，
       空格数较大时，value对空格数的导数较小'''
    emptyCellsvalue = selfEmptyCellsNums ** 0.5  # 更改为幂函数
    return emptyCellsvalue
```

3. figureValue：棋子级别评价指标

```
def figureValue(self, isFirst):  # 将对象方棋子的级别加权求和,得到棋子级别大小和数量提供的value
    selfside = isFirst
    selfscore = self.board.getScore(selfside)
    figurevalue = 0
    for i in selfscore:
        if i >= 7:  # 级别太低也没有加的必要
            figurevalue += 3 ** i
        elif i >= 5:
            figurevalue += 2 ** i
        elif i >= 3:
            figurevalue += i * 2
    return figurevalue
```

2.3 程序限制

目前没有发现程序报错或做出违法行为的现象。

在开发与优化过程中已被解决的 bug 有：

1. array 数组越界问题：原因是递归搜索树搜索到了 500 回合以后的棋盘状态。
   改进方法：剪掉 497 回合后的搜索枝

```
if self.current_round >= len(self.array) - 3:  # 递代到的回合数快超过总回合数时，剪掉搜索枝
    return self.get_BoardValue(player)
```

2. 棋盘状态复原的问题：只有深拷贝得到的对象才能在原对象发生改变后保持不变。

```
current_board = self.board.copy()
if self.board.move(isFirst, direction):  # 此合并方向合法时make move
    self.prev_stack.push(current_board)
```

   第一步 current_board 需要赋值为 self.board 的深拷贝对象，而不是 self.board,否则后续栈中的“父棋盘”均会随 self.board 的改变而改变，与 self.board 的指向是完全一致的。

3 实验结果

3.1 测试数据

实验环境说明：

- 硬件配置：Intel(R) Core(TM) i7-8580U CPU @1.8GHz　1.99GHz/8.00G（CPU/内存）
- 操作系统：Window10/64 位（名称/版本）
- Python 版本：3.8.2（版本号）

3.1.1 测试方法：

（1）关于迭代深度的测试

对于迭代深度的测试比较简单，即在确保不超时的情况下尽可能将其调为最大。

（2）关于评估函数的测试

对于评估函数考量参数及其权重的测试，本组主要采取控制变量法，即控制其他参数及其权重不变，调整其中某一个参数的权重，并与调整之前的代码进行热身赛，以此来确定最适合本组算法的评估函数参数及其权重。

（3）关于整体算法稳定性的测试

对于算法在运行过程中可能会出现的包括但不限于超时，调用非法等问题，需要对算法进行大量的实战测试，与不同的组的测试代码进行对战，以此来检测代码的稳定性与实战能力。

3.1.2 测试结果：

（1）迭代深度测试

注：由于代码最初有部分函数在 minimax 与 α - β 剪枝之前进行，得出的迭代深度实际上是伪迭代，所以会出现极其不可思议的数据。

迭代深度	6	7	8	9	10	11
测试结果	正常运行	正常运行	1.5%超时	78%超时	100%超时	100%超时

测试结论：最佳迭代深度为 depth=7

（2）评估函数测试

本组算法最初的评估函数共有 6 个参数，分别为平滑性，单调性，最大棋子级别，空格数，孤岛数，对方棋子在我方的评估。综合考量后，由于孤岛难以出现，且对整体算法贡献不大，故取消这一项。

根据参考文献，初步确定各项权重如下：

Smooth weight：0.02

Mono weight：0.26

Empty weight：0.54

Max weight：0.20

首先调节平滑性权重：（与调节之前的对战）

平滑性权重	胜场数	负场数	胜率
0.02	33	67	33%
0.04	31	69	31%
0.06	39	61	39%
0.08	49	51	49%
0.10	47	53	47%
0.12	58	42	58%
0.14	52	48	52%
0.16	50	50	50%
0.18	41	59	41%
0.20	38	62	38%
0.22	42	58	42%
0.24	45	55	45%
0.26	35	65	35%
0.28	29	71	29%
0.30	21	79	21%
0.32	25	75	25%
0.34	12	88	12%
0.36	13	87	13%
0.38	5	95	5%
0.40	8	92	8%

同样方法调节其他权重，最终得出最佳权重如下：

Smooth weight：0.12

Mono weight：0.23

Empty weight：0.22

Max weight：0.42

Enemy weight：0.17

（4）稳定性测试（包括热身赛的结果）：

在本地进行大量测试没有问题之后，我们转战天梯热身赛

与当时天梯热身赛排名前十的代码对战，全部以 0-10 战败，但没有报错

对战其他队伍的战况如下

我方	我方胜利	对方胜利	对方	胜利原因	失败原因
我方	3	97	Test1	得分统计	得分统计
我方	8	92	微波炉 4 号	得分统计	得分统计
我方	12	88	微波炉 2 号	得分统计	得分统计
我方	23	77	Emm	得分统计	得分统计
我方	19	81	京都人形	得分统计	得分统计
我方	100	0	Test2	对方报错	得分统计
我方	0	100	0.3.3.3	得分统计	得分统计
我方	15	85	Kiiii	得分统计	得分统计

我方	0	100	Player526	得分统计	得分统计
我方	77	23	四季平安	对方报错	得分统计
我方	19	81	D3	得分统计	得分统计
我方	0	100	对战试试	得分统计	得分统计
我方	0	100	二龙戏珠	得分统计	得分统计

经过测试发现，我方算法稳定性较好，但竞争力极差。

测试之后，我们在算法中加入了 contest 和 athena 函数来提高算法的竞争力，但效果并不明显，反而导致新增函数覆盖评估函数的情况，导致评估失效。

所以，我们对算法进行了返璞归真式的处理，重新编辑代码，只留下搜索和评估，同时重整了搜索的代码。接下来的测试，更多的使用了与官方的 bot 进行对战处理。

我方	我方胜利	对方胜利	对方	胜利原因	失败原因
大换血	20	0	Bot D	得分统计	
大换血	40	0	Bot C	得分统计	
大换血	7	23	Bot B	得分统计	得分统计
大换血	0	40	Bot A		得分统计
大换血	50	0	8848hh	得分统计	

经过这一次优化，我方算法竞争力明显提升，我们又针对评估函数的空格数一项进行了优化，使得其权重分配更加合理。最终版本战绩如下：

我方	我方胜利	对方胜利	对方	胜利原因	失败原因
最终版	12	8	大换血	得分统计	得分统计
最终版	19	41	Bot B	得分统计	得分统计
最终版	2	9	Bot A	得分统计	得分统计

最终版本稳定性较高，竞争力较之前有了显著提升。

3.2 结果分析

3.2.1 测试阶段

在测试阶段，评估函数被覆盖，导致无法对战况做出正确的判断，添加的撤回机制，进攻机制并不能根据形势做出正确的判断，竞争力较差。算法在迭代深度为 7 时稳定不超时，在迭代深度为 8 是不稳定偶尔超时。由于评估函数被覆盖，迭代均为伪迭代，时间开销主要用于各个函数以及权重计算。

3.2.2 热身赛阶段

此阶段添加了 contest 和 athena 函数增强了对我方的保护，但由于评估函数的问题并没有修改，效果仍然不理想，战绩较差。伪迭代深度甚至可以达到 12，时间开销仍然没有用于迭代评估。

3.2.3 天梯实战阶段

此阶段进行了大换血的更改，删除了没有用处的函数，更改了评估函数的迭代机制，效果理想，战绩有显著提升。此时经过测试发现，迭代深度为 3 时，算法稳定不超时，迭代深度超过 3 时，有超时风险。时间开销主要用于迭代计算评估局面，以做出正确的选择。

4 实习过程总结

4.1 分工与合作

【1】 分工情况：

刘行健——负责除评估函数接口以外的算法和代码编写；后期主要负责 debug 和评估函数的优化

杨轩——负责评估函数的研究和编写

宋思潼——负责分析攻守策略，编写相应的功能函数

吕东澍——负责程序测试和比赛结果的分析与记录，调整评估函数权重

白纯迪——负责搜索树算法的意见指导；后期负责分析攻守策略

【2】 工作阶段：

第一阶段：写出整体的代码框架：

DDL：5.20 晚(尽早，写完后及时共享)

【1】 负极大值搜索算法（博弈树）及 alpha-beta 剪枝 —— 刘行健，白纯迪

分工方法：两人同时进行，不断讨论，最后整合优化

【2】 评估函数 ——杨轩

初期任务：写出单人版 2048 的评估函数

第二阶段：优化算法 ——宋思潼，吕东澍（其他三人参与讨论，尤其是【2】）

DDL：初步定 5.28（左右）开始天梯赛竞技

（6.2 课上为冠亚军争夺战）

【1】 优化算法效率

【2】 关于进攻和防守的决策：评估函数的优化——添加更多条件分支及更改权重

进攻：在对方棋盘落棋的位置；有棋子在对方棋盘时的合并方向

防守：对方棋子在我方棋盘时的情况

【3】 检查添加【2】后的非法行为等异常情况

【4】 本地运行平台进行初步调试：

通过人机对战和人人对战来枚举特殊情况，优化评估函数

【5】 进行代码竞技场热身赛尝试（不是天梯赛）

【3】 合作方式

1. 阶段性组会与会议记录

（第一次组会）

（第二次组会）

2. gitee 合作

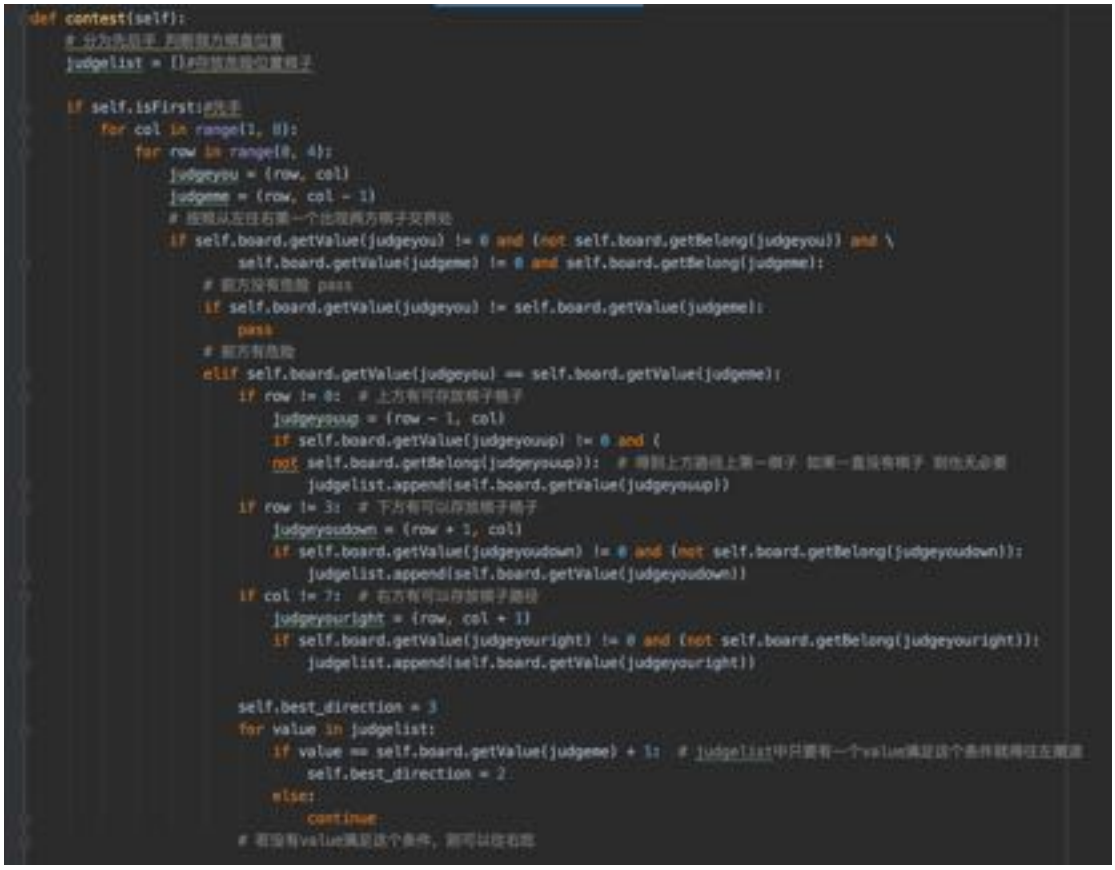
3. 微信群合作

数算大作业通知群（勿水）(5)
player 5.py
player6.py
https://blog.csdn.net/qq_32767041/article/details/80634241
@所有人
本群用于存放主群中的各种文件/链接
大作业提前
PC游戏编程-人机博弈.pdf
test11.py
大作业会议内容.docx
评估函数里的 isFirst 是 int 另
不是 bool？

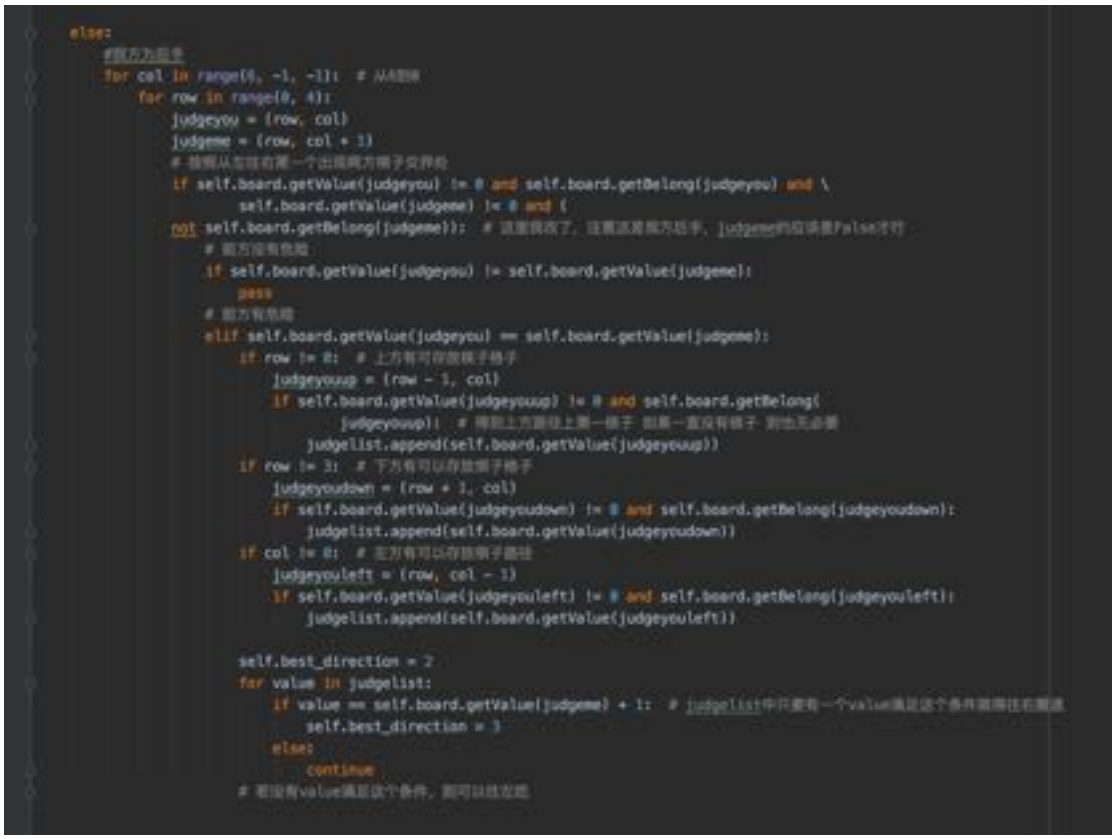
4.2 经验与教训

虽然说小组最终版的代码十分朴素，但其实我们组在开发的过程中有一段**痛心的血泪史。**
首先，在 1.1 中讲过，我们后期优化时重点侧重了对于“即时危机”和“即时优势”的细化，编写了一些掐断式的决策函数，却由于在有限的时间内未能将函数优化得十分周到，反而导致算法的目光变得短浅，胜率还不如以往的“简朴版”。在“繁琐版”天梯经过一段时间的低靡之后，我们决定“~~返璞归真~~”将代码更换为以前的简朴版，排名这才提高了不少。
以及，我们最初版的评估函数评估了棋盘状态的许多方面，但由于评估方面太多反而导致某些方面的权重为 0 时貌似最优，故删去了一些次要的评估方面，留下最关键的几个。
下面展示的是我们在后期优化过程中编写的函数以及编写函数时的心路历程。

4.2.1 contest 函数



```
def contest(self):
    # 分为先后手 判断双方棋盘位置
    judgelist = []存放先后位置棋子

    if self.isFirst:先手
        for col in range(3, 0):
            for row in range(0, 4):
                judgeyou = (row, col)
                judgeme = (row, col - 1)
                # 按照从左往右第一个出现两方棋子交界处
                if self.board.getValue(judgeyou) != 0 and (not self.board.getBelong(judgeyou)) and \
                        self.board.getValue(judgeme) != 0 and self.board.getBelong(judgeme):
                    # 前方没有危险 pass
                    if self.board.getValue(judgeyou) != self.board.getValue(judgeme):
                        pass
                    # 前方有危险
                    elif self.board.getValue(judgeyou) == self.board.getValue(judgeme):
                        if row != 0:  # 上方有可存放棋子格子
                            judgeyouup = (row - 1, col)
                            if self.board.getValue(judgeyouup) != 0 and (
                            not self.board.getBelong(judgeyouup)):  # 得到上方路径上第一棋子 如果一直没有棋子 则也无必要
                                judgelist.append(self.board.getValue(judgeyouup))
                        if row != 3:  # 下方有可以存放棋子格子
                            judgeyoudown = (row + 1, col)
                            if self.board.getValue(judgeyoudown) != 0 and (not self.board.getBelong(judgeyoudown)):
                                judgelist.append(self.board.getValue(judgeyoudown))
                        if col != 7:  # 右方有可以存放棋子路径
                            judgeyouright = (row, col + 1)
                            if self.board.getValue(judgeyouright) != 0 and (not self.board.getBelong(judgeyouright)):
                                judgelist.append(self.board.getValue(judgeyouright))

                        self.best_direction = 3
                        for value in judgelist:
                            if value == self.board.getValue(judgeme) + 1:  # judgelist中只要有一个value满足这个条件就得往左搬运
                                self.best_direction = 2
                            else:
                                continue
                        # 若没有value满足这个条件，则可以往右跑
```



```
    else:
        #前方为后手
        for col in range(6, -1, -1):  # 从6到0
            for row in range(0, 4):
                judgeyou = (row, col)
                judgeme = (row, col + 1)
                # 按照从左往右第一个出现两方棋子交界处
                if self.board.getValue(judgeyou) != 0 and self.board.getBelong(judgeyou) and \
                        self.board.getValue(judgeme) != 0 and (
                not self.board.getBelong(judgeme)):  # 这里我改了，注意这是前方后手，judgeme的应该是False才对
                    # 前方没有危险
                    if self.board.getValue(judgeyou) != self.board.getValue(judgeme):
                        pass
                    # 前方有危险
                    elif self.board.getValue(judgeyou) == self.board.getValue(judgeme):
                        if row != 0:  # 上方有可存放棋子格子
                            judgeyouup = (row - 1, col)
                            if self.board.getValue(judgeyouup) != 0 and self.board.getBelong(
                                    judgeyouup):  # 得到上方路径上第一棋子 如果一直没有棋子 则也无必要
                                judgelist.append(self.board.getValue(judgeyouup))
                        if row != 3:  # 下方有可以存放棋子格子
                            judgeyoudown = (row + 1, col)
                            if self.board.getValue(judgeyoudown) != 0 and self.board.getBelong(judgeyoudown):
                                judgelist.append(self.board.getValue(judgeyoudown))
                        if col != 0:  # 左方有可以存放棋子路径
                            judgeyouleft = (row, col - 1)
                            if self.board.getValue(judgeyouleft) != 0 and self.board.getBelong(judgeyouleft):
                                judgelist.append(self.board.getValue(judgeyouleft))

                        self.best_direction = 2
                        for value in judgelist:
                            if value == self.board.getValue(judgeme) + 1:  # judgelist中只要有一个value满足这个条件就得往右搬运
                                self.best_direction = 3
                            else:
                                continue
                        # 若没有value满足这个条件，则可以往左跑
```

4.2.1.1 心路历程

1.函数作用：

在修改完评估函数后复盘战局，我们发现常常会出现这样的情况：我方棋子当与对方接壤且级别一致时常常会不顾一切地向前进攻，而这样的结果往往是“螳螂捕蝉，另一只蝉在背后”，为了解决这样的问题，我编写了 contest 函数，企图先通过 contest 的判断避免 AI 执行评估函数得出来的“愚蠢结果”。

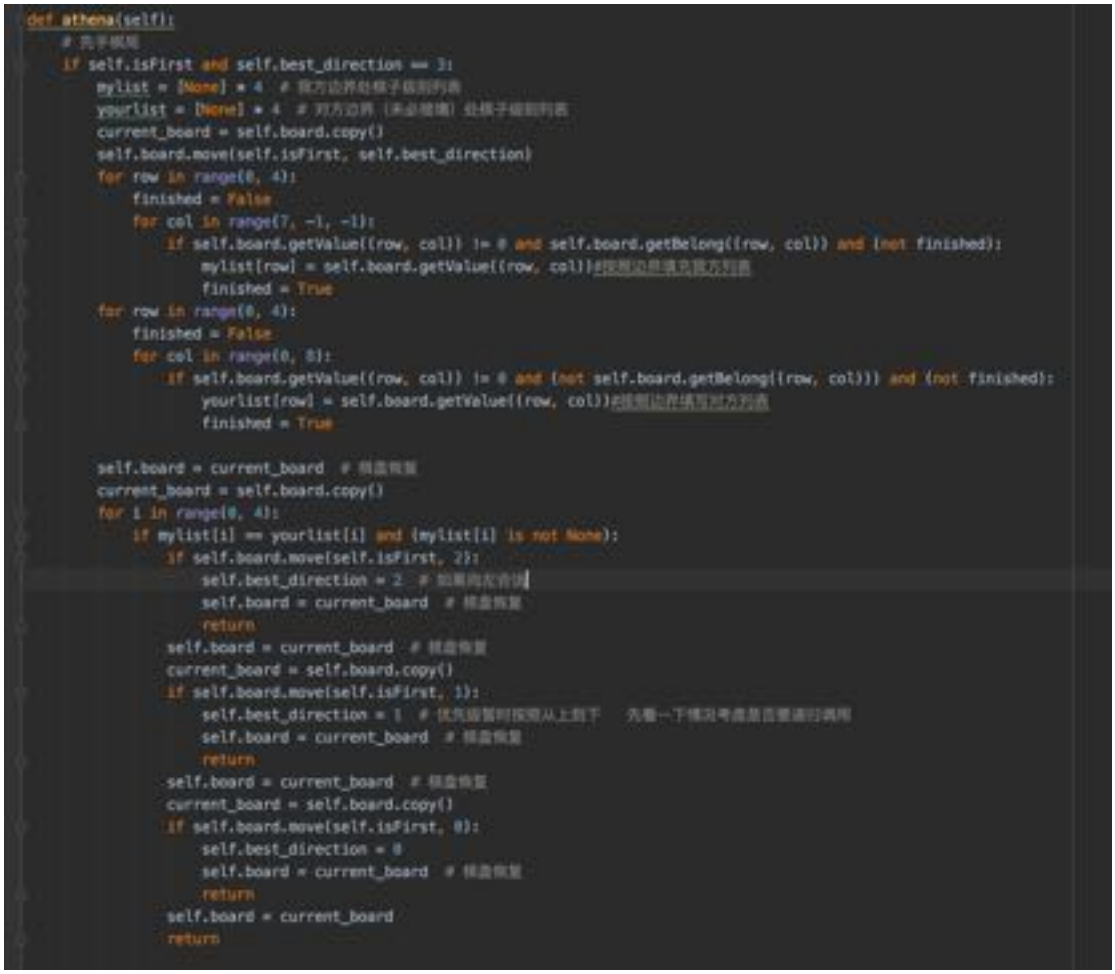
2.函数实现：

初版 contest 函数的优先级高于评估函数，首先根据先后手不同选择不同的遍历顺序找到接壤处，如果存在交锋的“危险”，将继续判断对方棋子相邻三个方向的棋子级别是否会对我们进攻造成威胁，如果有威胁，则选择后退；如果前方尚且没有危险则迅速出击。

在对增加了 contest 的 AI 进行复盘的时候，虽然胜率略有增加，但是这个策略仅在后期起着比较大的作用。而前期双方接壤时，对我们造成威胁的棋子不仅仅是相邻三个方向上的，而是三个方向路径上的，针对这种情况，我对代码进行了修改，得到了第二版的 contest 函数（如上 ）。

再次复盘后我们发现，的确避免了这种情况。但是修改 contest 前后，AI 的策略发生了极大的变化，从相对积极转变为了相对保守，在这样的情况下，我们的胜率并没有明显的提升。我们最终版本的代码也没有选用 contest 函数，再次修改评估函数之后，我们放弃了 contest 函数。现在看来，contest 函数编写的背景是在我们评估函数存在瑕疵的情况下，并且这种人眼的“短视”存在极大的“鼠目寸光“的危险，不如效能比较好的 AI 的多步棋局评估。

4.2.2 athena 函数



```
def athena(self):
    # 先手模板
    if self.isFirst and self.best_direction == 3:
        mylist = [None] * 4  # 我方边界处棋子编到列表
        yourlist = [None] * 4  # 对方边界（未必模板）处棋子编到列表
        current_board = self.board.copy()
        self.board.move(self.isFirst, self.best_direction)
        for row in range(0, 4):
            finished = False
            for col in range(7, -1, -1):
                if self.board.getValue((row, col)) != 0 and self.board.getBelong((row, col)) and (not finished):
                    mylist[row] = self.board.getValue((row, col))#根据边界填充我方列表
                    finished = True
        for row in range(0, 4):
            finished = False
            for col in range(0, 8):
                if self.board.getValue((row, col)) != 0 and (not self.board.getBelong((row, col))) and (not finished):
                    yourlist[row] = self.board.getValue((row, col))#根据边界填写对方列表
                    finished = True

        self.board = current_board  # 棋盘恢复
        current_board = self.board.copy()
        for i in range(0, 4):
            if mylist[i] == yourlist[i] and (mylist[i] is not None):
                if self.board.move(self.isFirst, 2):
                    self.best_direction = 2  # 如果向左吞掉
                    self.board = current_board  # 棋盘恢复
                    return
                self.board = current_board  # 棋盘恢复
                current_board = self.board.copy()
                if self.board.move(self.isFirst, 1):
                    self.best_direction = 1  # 优先级暂时按照从上到下   先看一下情况考虑是否要递归调用
                    self.board = current_board  # 棋盘恢复
                    return
                self.board = current_board  # 棋盘恢复
                current_board = self.board.copy()
                if self.board.move(self.isFirst, 0):
                    self.best_direction = 0
                    self.board = current_board  # 棋盘恢复
                    return
                self.board = current_board
                return
```

4.2.2.1 心路历程

1.函数说明：

评估函数做出的决定常常出现”送子观音“的情况，为了避免我方辛辛苦苦合成的棋子成为对方的间谍，开发 athena（处女神）函数，判断评估函数做出决定后的棋局是否存在成为”送子观音“的危险，进行有效规避。

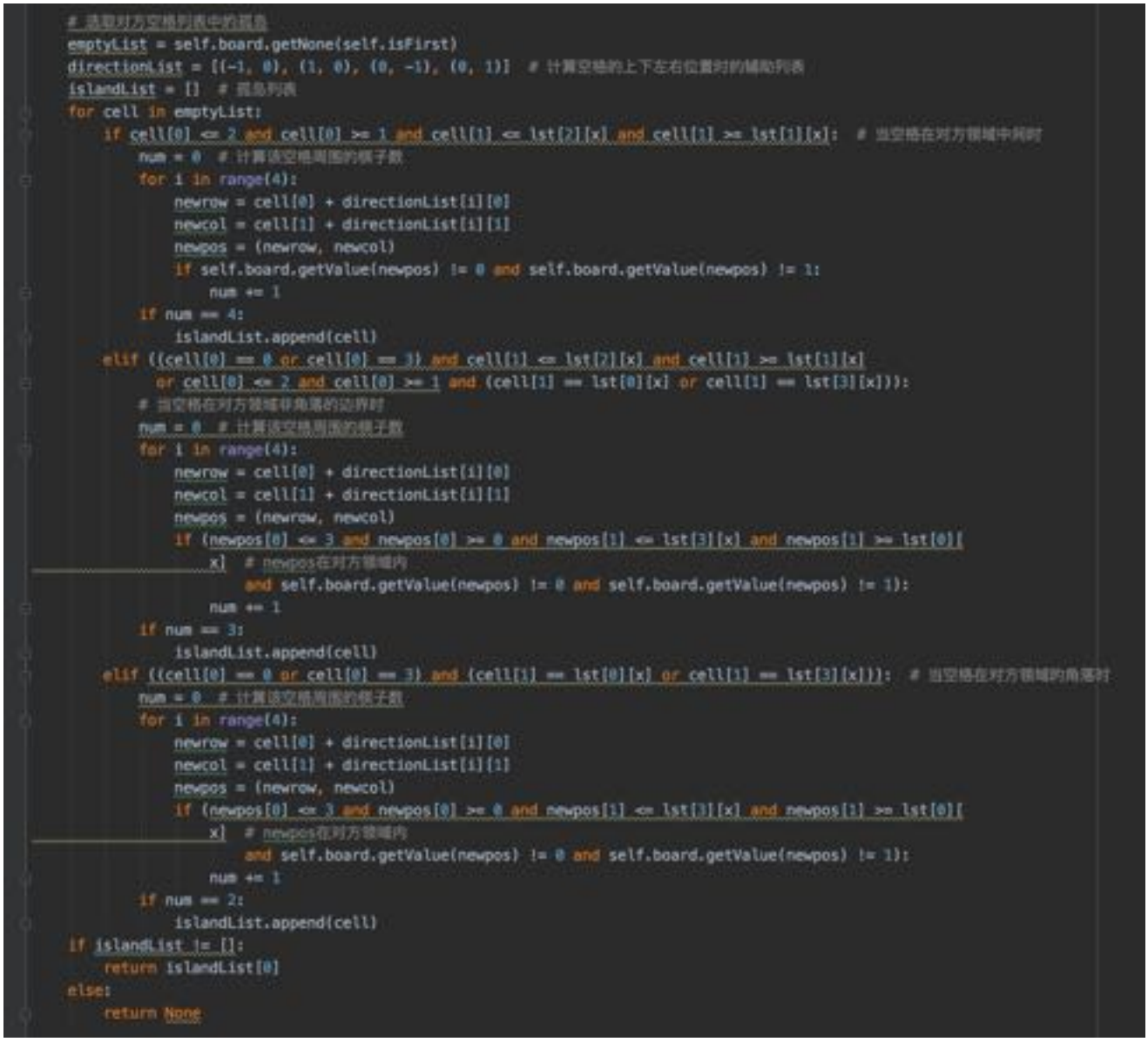
2.函数实现：

在此仅以先手方为例，代码如上，根据评估函数做出决定后的棋局分别得到我方和对方的边界棋子列表，一旦出现成为”送子观音“的风险，则选择其他合法方向进行合并。

athena 函数其实和 contest 函数的核心思想基本一致，都是希望避免当前的局面对我方不利，我们希望达成一个当前局面的优先级大于迭代几局的结果，但是事后反思发现评估函数的作用正是在此，突破人脑的局限，对多层棋局进行模拟，所以这也是我们最终版代码并没有选用 athena 函数的原因。

4.2.3 invade 函数

```
def invade(self):
    lst = [(0, 4), (1, 5), (2, 6), (3, 7)]  # 用于转换先后手时判断不同的辅助列表
    # 对方即将合成较大棋子时若可以阻挠则选取其路径上的空格
    value = 5  # 用于判断棋子级别是否达到标准
    position = None
    x = 1 if self.isFirst else 0  # 用于选取对方棋盘
    for row in range(4):  # 遍历对方的每一行
        valueList = [self.board.getValue((row, column)) for column in range(lst[0][x], lst[3][x] + 1)]
        # 若对方某一行中有两个等级大于等于5的棋子，且中间为空格，则取出该空格的位置
        # 若存在不止一个这样的位置，则选取等级更大时的空格位置
        if valueList[0] == valueList[2] and valueList[1] == 0 and valueList[0] >= value:
            value = valueList[0]
            position = (row, lst[1][x])
        elif valueList[1] == valueList[3] and valueList[2] == 0 and valueList[1] >= value:
            value = valueList[0]
            position = (row, lst[2][x])
        elif valueList[0] == valueList[3] and valueList[1] == 0 and valueList[2] == 0 and valueList[0] >= value:
            value = valueList[0]
            position = (row, lst[1][x])
    for column in range(lst[0][x], lst[3][x] + 1):  # 遍历对方的每一列
        valueList = [self.board.getValue((row, column)) for row in range(4)]
        # 若对方某一列中有两个等级大于等于5的棋子，且中间为空格，则取出该空格的位置
        # 若存在不止一个这样的位置，则选取等级更大时的空格位置
        if valueList[0] == valueList[2] and valueList[1] == 0 and valueList[0] >= value:
            value = valueList[0]
            position = (1, column)
        elif valueList[1] == valueList[3] and valueList[2] == 0 and valueList[1] >= value:
            value = valueList[0]
            position = (2, column)
        elif valueList[0] == valueList[3] and valueList[1] == 0 and valueList[2] == 0 and valueList[0] >= value:
            value = valueList[0]
            position = (1, column)
    if position != None:
        return position
```



```
    # 选取对方空格列表中的首选
    emptyList = self.board.getNone(self.isFirst)
    directionList = [(-1, 0), (1, 0), (0, -1), (0, 1)]  # 计算空格的上下左右位置时的辅助列表
    islandList = []  # 孤岛列表
    for cell in emptyList:
        if cell[0] <= 2 and cell[0] >= 1 and cell[1] <= lst[2][x] and cell[1] >= lst[1][x]:  # 当空格在对方领域中间时
            num = 0  # 计算该空格周围的棋子数
            for i in range(4):
                newrow = cell[0] + directionList[i][0]
                newcol = cell[1] + directionList[i][1]
                newpos = (newrow, newcol)
                if self.board.getValue(newpos) != 0 and self.board.getValue(newpos) != 1:
                    num += 1
            if num == 4:
                islandList.append(cell)
        elif ((cell[0] == 0 or cell[0] == 3) and cell[1] <= lst[2][x] and cell[1] >= lst[1][x]
              or cell[0] <= 2 and cell[0] >= 1 and (cell[1] == lst[0][x] or cell[1] == lst[3][x])):
            # 当空格在对方领域非角落的边界时
            num = 0  # 计算该空格周围的棋子数
            for i in range(4):
                newrow = cell[0] + directionList[i][0]
                newcol = cell[1] + directionList[i][1]
                newpos = (newrow, newcol)
                if (newpos[0] <= 3 and newpos[0] >= 0 and newpos[1] <= lst[3][x] and newpos[1] >= lst[0][
                    x]  # newpos在对方领域内
                        and self.board.getValue(newpos) != 0 and self.board.getValue(newpos) != 1):
                    num += 1
            if num == 3:
                islandList.append(cell)
        elif ((cell[0] == 0 or cell[0] == 3) and (cell[1] == lst[0][x] or cell[1] == lst[3][x])):  # 当空格在对方领域的角落时
            num = 0  # 计算该空格周围的棋子数
            for i in range(4):
                newrow = cell[0] + directionList[i][0]
                newcol = cell[1] + directionList[i][1]
                newpos = (newrow, newcol)
                if (newpos[0] <= 3 and newpos[0] >= 0 and newpos[1] <= lst[3][x] and newpos[1] >= lst[0][
                    x]  # newpos在对方领域内
                        and self.board.getValue(newpos) != 0 and self.board.getValue(newpos) != 1):
                    num += 1
            if num == 2:
                islandList.append(cell)
    if islandList != []:
        return islandList[0]
    else:
        return None
```

4.2.3.1 心路历程

在优化的过程中，我们认为，当对方有较大的棋子即将合并而中间有空格时，下在这个空额可以阻碍对方合成更大的棋子；以及当对方有一个孤岛（周围均为对方棋子的空格）时，如果下在这个空格的位置，无论对方接下来往哪个方向进行合并，都能保持我们的棋子在一定时间内在其内部，从而对对方的合并造成阻碍。Invade 函数是对我们一开始持续了很久的相对保守的策略的进行的一次尝试，加入了 invade 函数之后，大部分时候决策仍然继承先前的保守，但在能对对方造成较大阻碍时则采取了进攻的姿态。然而写出 invade 函数之后，我们发现其有效的时刻并不是很多，尤其是孤岛的情形，很少出现。对当前局面进行预判的方式在后来也被证明是不如搜索树评估的，于是最后我们放弃了 invade 函数。

4.2.4 最初版评估函数

此函数为客观评价函数，即输入参数 阵营(isFirst) ，输出该阵营在此棋面下的 value，不附带有主观性，敌我均可用。 将 isFirst 代表的阵营称为对象方，另一方则称为对方（这也深刻体现了该函数的客观性）。isFirst 为 bool 值，isFirst = True 相当于 isFirst = 1，isFirst = False 相当于 isFirst = -1

4.2.4.1 总函数：estimate

```
def estimateValue(self, isFirst):  # 计算先手或后手方的棋面value
    smoothnessWeight = 0.1  ####平滑性权重
    monotonicityWeight = 0.1  ####单调性权重
    emptyCellsWeight = 0.1  ####空格数权重
    islandsWeight = 0.1  ####孤岛数权重
    figureWeight = 0.1  ####棋子级别权重
    invadeWeight = 0  ####入侵对方阵的权重
    value = smoothnessWeight * self.smoothnessValue(isFirst) + \
            monotonicityWeight * self.monotonicityValue(isFirst) + \
            emptyCellsWeight * self.emptyCellsValue(isFirst) + \
            islandsWeight * self.islandsValue(isFirst) + \
            figureWeight * self.figureValue(isFirst) + \
            invadeWeight * self.invadeValue(isFirst) ####总value
    return value
```

4.2.4.1.1 构建思路：

计算当前棋盘上各种参数（如空格数、棋子级别等）贡献的 value，加权求和得到最终的 value

4.2.4.2 棋子级别函数：figurevalue

```
def figureValue(self, isFirst):  # 将对象方棋子的级别加权求和 得到 棋子级别大小和数量提供的value
    import math
    selfSide = isFirst
    selfChessmanList = self.board.getScore(selfSide)  # 对象方所有棋子级别按升序排成的列表
    figurevalue = 0
    for level in selfChessmanList:  # 对每个棋子计算其value并加入到figurevalue中
        '''此处计算式可更改 但是应有value(2*level)>2*value(level)'''
        figurevalue += math.pow(2.2, level)
    return figurevalue
```

4.2.4.2.1 构建思路：

考虑对象方棋子本身贡献的 value 时，要同时考虑棋子的最高级别是多少和各级别棋子的数量。由于高级别的棋子贡献更大，所以在计算 value 时，应保证一个高级别的棋子的 value 应比多个低级别棋子的 value 要大。因此选择每个棋子贡献的 value 为棋子级别 level 的 n 次方。（n>2）

4.2.4.3 空格数函数：emptyCellsValue

```
def emptyCellsValue(self, isFirst):  # 计算对象方空位数所提供的value
    import math
    selfSide = isFirst
    selfEmptyCellsNums = len(self.board.getNone(selfSide))  # 对象方空位数
    '''此处计算式可更改，但是应有：在空格数较少时，value对空格数的导数较大，
       空格数较大时，value对空格数的导数较小'''
    if selfEmptyCellsNums != 0:
        emptyCellsvalue = math.log(selfEmptyCellsNums, 2)  # 这里有定义域问题，如果selfEmptyCellsNums=0则会报错，故增加一种情况，
        return emptyCellsvalue
    return 0
```

4.2.4.3.1 构建思路：

对象方棋盘上的空格数越多，对象方棋子的机动性就越大，越不容易被卡位。当空位较多时，增加一个空位带来的 value 增量应该比较小，因为整体的机动性变化不明显。而当空位较少时，增加一个空位带来的 value 增量应该比较大，因为这时候每个空位都很关键。因此选择对数函数，使得空位数较少时，value 对空位数的导数较大，空位数较大时，value 对空位数的导数较小。

4.2.4.4 孤岛函数：islandValue

```
def islandsValue(self, isFirst):
    import math
    selfSide = isFirst
    selfEmptyCellsPosList = self.board.getNone(selfSide)  # 获得对象方所有空格位置的列表
    islandNums = 0  # 孤岛数计数
    directionList = [(-1, 0), (1, 0), (0, -1), (0, 1)]  # 计算空格的上下左右位置时的辅助列表
    for cell in selfEmptyCellsPosList:  # 考虑每一个对象方空格
        isIsland = True  # 用于判断空格是否为孤岛
        i = 0  # 计数器
        # 遍历此空格上下左右的格子
        while i <= 3 and isIsland:
            newrow = cell[0] + directionList[i][0]  # 空格周围格子所在的行
            newcol = cell[1] + directionList[i][1]  # 空格周围格子所在的列
            newpos = (newrow, newcol)  # 空格周围格子的坐标

            if newrow < 0 or newrow > 3:
                # 行数不在对象方棋盘范围内
                pass
            elif (newcol - (1 - isFirst) * 4) < 0 or (newcol - (1 - isFirst) * 4) > 3:
                # 列数不在对象方棋盘范围内
                pass
            elif self.board.getValue(newpos) <= 1:  # 若 空格旁的这个格子 级别小于等于1，则其必定是对象方的'2'或者空格
                isIsland = False

            i += 1

        if isIsland:
            islandNums += 1
    '''此处可更改表达式，但应保证value为负值'''
    islandsvalue = - math.pow(islandNums, 1.2)
    return islandsvalue
```

4.2.4.4.1 构建思路：

对于对象方棋盘，若一个空格 周围没有 其他空格 或者 级别为 1 的棋子 时，将其称为孤岛，它卡在高级别棋子当中，当级别为 1 的棋子落在其上时，会对整体的平滑性与单调性造成较大影响。因此它的存在会使得总 value 下降。

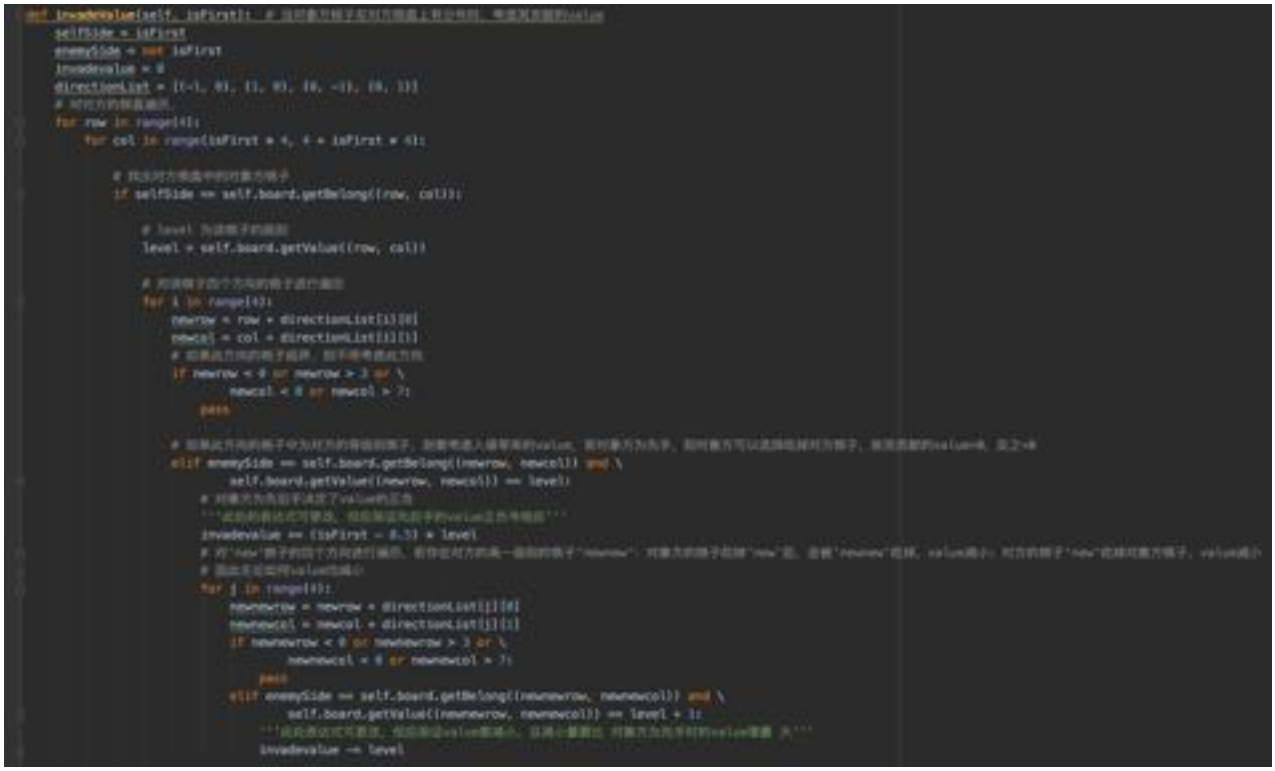
4.2.4.5 平滑性函数：smoothnessValue

```
def smoothnessValue(self, isFirst):  # 计算对象方棋子的平滑性，即统一方格上相邻棋子级别差异是否明显，不平滑度低的value应为负值
    selfSide = isFirst
    smoothnessvalue = 0  # 初始化平滑性value
    directionList = [(-1, 0), (1, 0), (0, -1), (0, 1)]
    # 遍历棋盘上每一个格子（因为对象方棋子可能在对方棋盘内有分布）
    for row in range(0, 4):
        for col in range(0, 8):
            # 如果此格为对象方的棋子，则要考虑其与邻近的对象方棋子贡献的平滑性value
            if selfSide == self.board.getBelong((row, col)) and \
                    self.board.getValue((row, col)) != 0:
                # level代表此格棋子的级别
                level = self.board.getValue((row, col))
                # 遍历此格棋子上下左右的格子
                for i in range(4):
                    newrow = row + directionList[i][0]
                    newcol = col + directionList[i][1]
                    # 如果此方向的格子超界，则不用考虑此方向
                    if (newrow) < 0 or (newrow) > 3 or \
                            (newcol) < 0 or (newcol) > 7:
                        pass
                    # 如果此方向的格子上为对象方棋子，则要考虑平滑性影响
                    elif selfSide == self.board.getBelong((newrow, newcol)) and \
                            self.board.getValue((newrow, newcol)) != 0:
                        newLevel = self.board.getValue((newrow, newcol))
                        # deltaLevel代表两棋子的级别差
                        deltaLevel = abs(level - newLevel)
                        '''此处表达式可以修改，但应保证smoothnessvalue非正'''
                        smoothnessvalue -= deltaLevel ** 2

    return smoothnessvalue
```

4.2.4.5.1 构建思路：

在对象方棋盘内，考虑一行或者一列，当相邻的棋子级别之差较小时，它们就较容易合并，此时这一行或者一列的平滑性较好。因此两相邻的棋子级别差越大，它俩贡献的 value 就越小。

4.2.4.6 进攻函数：invadeValue



```
def invadeValue(self, isFirst):  # 当对象方棋子在对方棋盘上有分布时，考虑其贡献的value
    selfSide = isFirst
    enemySide = not isFirst
    invadevalue = 0
    directionList = [[-1, 0], [1, 0], [0, -1], [0, 1]]
    # 对对方的棋盘遍历
    for row in range(4):
        for col in range(isFirst * 4, 4 + isFirst * 4):

            # 找出对方棋盘中的对象方棋子
            if selfSide == self.board.getBelong(row, col):

                # level 为该棋子的级别
                level = self.board.getValue(row, col)

                # 对该棋子四个方向的棋子进行遍历
                for i in range(4):
                    newrow = row + directionList[i][0]
                    newcol = col + directionList[i][1]
                    # 如果此方向的棋子超界，则不用考虑此方向
                    if newrow < 0 or newrow > 3 or \
                            newcol < 0 or newcol > 7:
                        pass

                    # 如果此方向的棋子中为对方的等级别棋子，则要考虑入侵带来的value，若对象方为先手，则对象方可以选择吃掉对方棋子，故贡献的value>0，反之<0
                    elif enemySide == self.board.getBelong(newrow, newcol) and \
                            self.board.getValue(newrow, newcol) == level:
                        # 对象方为先后手决定了value的正负
                        '''此处的表达式可修改，但应保证先后手的value正负号相反'''
                        invadevalue += (isFirst - 0.5) * level
                        # 对'new'棋子的四个方向进行遍历，若存在对方的高一级别的棋子'newnew'，对象方的棋子在除'new'后，会被'newnew'吃掉，value减小；对方的棋子'new'吃掉对象方棋子，value减小
                        # 因此不论如何value均减小
                        for j in range(4):
                            newnewrow = newrow + directionList[j][0]
                            newnewcol = newcol + directionList[j][1]
                            if newnewrow < 0 or newnewrow > 3 or \
                                    newnewcol < 0 or newnewcol > 7:
                                pass
                            elif enemySide == self.board.getBelong(newnewrow, newnewcol) and \
                                    self.board.getValue(newnewrow, newnewcol) == level + 1:
                                '''此处表达式可修改，但应保证value都减小，且减小量最后 对象方为先手时的value要 大'''
                                invadevalue -= level
```

```
                    # 如果此方向为对方空格，则对方的等级别棋子可能会隔着空格击杀，此时value会减小
                    elif enemySide == self.board.getBelong(newrow, newcol) and \
                            self.board.getValue(newrow, newcol) == 0:
                        stop = False
                        # 沿该方向前进，若出界，则停下；若碰到对方空格，则继续前进；若碰到对方等级别棋子，则停下并且value减小；其他情况下均停下
                        while not stop:
                            newrow = newrow + directionList[i][0]
                            newcol = newcol + directionList[i][1]
                            # 出界
                            if newrow < 0 or newrow > 3 or \
                                    newcol < 0 or newcol > 7:
                                stop = True
                            # 碰到对方等级别棋子
                            elif enemySide == self.board.getBelong(newrow, newcol) and \
                                    self.board.getValue(newrow, newcol) == level:
                                '''此处的表达式可修改，但应保证value减小'''
                                invadevalue -= 0.5 * level
                                stop = True
                            # 碰到对方空格，前进四
                            elif enemySide == self.board.getBelong(newrow, newcol) and \
                                    self.board.getValue(newrow, newcol) == 0:
                                stop = False
                            # 其他情况，停下
                            else:
                                stop = True
    return invadevalue
```

4.2.4.6.1 构建思路：

当对象方棋子在对方棋盘内有分布时，依据情况的不同来计算其贡献的 value:（1）其周围存在有对方的等级别棋子时，若对象方为先手，则对象方可以选择吃掉对方棋子，故而贡献的 value>0,反之<0。(2）其他的一些特殊情况另外考虑。

4.2.4.7 单调性函数：monotonicityValue

```
def monotonicityValue(self, isFirst):  # 单调性value评估，仅考虑对象方棋盘内的棋子，且对方棋子入侵时，可认为其所在的行与列的value贡献为0
    selfSide = isFirst
    monotonicityvalue = 0
    # 遍历对象方棋盘每一行
    for row in range(4):
        # 引入单调性因子，表征单行或者单列的单调性好坏
        monotonicityFactor = 0
        # 列数从对象方棋盘的最左列到最右列遍历，即遍历这一行的所有对象方格子
        col = (1 - isFirst) * 4
        while col < (1 - isFirst) * 4 + 3:
            # 当前格子的级别
            currentValue = self.board.getValue((row, col))
            # 下一格的坐标
            nextrow = row
            nextcol = col + 1
            # 有内鬼，则终止交易，转至下一行
            if selfSide != self.board.getBelong((row, col)) or \
                    selfSide != self.board.getBelong((nextrow, nextcol)):
                monotonicityFactor = 0
                break  # 只是跳出while循环
            # 无内鬼，计算下一格的级别，单调性因子变化
            else:
                nextValue = self.board.getValue((nextrow, nextcol))
                '''此表达式可以更改'''
                monotonicityFactor += nextValue - currentValue
            # 列数加一，继续下一格
            col += 1
        # 此行的单调性因子计算完毕，加至value中
        '''此表达式可以更改'''
        monotonicityvalue += abs(monotonicityFactor)
```

```
    # 遍历对象方棋盘每一列，同上一种情况
    for col in range((1 - isFirst) * 4, (1 - isFirst) * 4 + 4):
        monotonicityFactor = 0
        row = 0
        while row < 3:
            currentValue = self.board.getValue((row, col))
            nextrow = row + 1
            nextcol = col
            if selfSide != self.board.getBelong((row, col)) or \
                    selfSide != self.board.getBelong((nextrow, nextcol)):
                monotonicityFactor = 0
                break
            else:
                nextValue = self.board.getValue((nextrow, nextcol))
                monotonicityFactor += nextValue - currentValue
            row += 1
        monotonicityvalue += abs(monotonicityFactor)
    return monotonicityvalue
```

4.2.4.7.1 构建思路：

当一行或者一列内的棋子级别是单调增加或者减少时，其越容易进行合并，因此会贡献 value。对于对象方棋盘内的每一行和每一列进行单调性评估，仅当该列或行的棋子均为对象方棋子且级别单调变化时，才计算 value 贡献。

4.3 建议与设想

建议：增大最大思考时间，比如从 5s→10s，这样可以使得决策更优化，比赛的观赏性更强。

寄语：希望选修这门课的学弟学妹们，多在数算上下功夫，就一定能获得更多的知识与回报，课程虽卷，但最终一定会收获满满。

大作业设想：参考贪吃蛇大作战进行智能对战。

5 致谢

感谢陈斌老师一学期以来干货满满的教授，让我们对 python 语言的掌握更加熟练，对算法有了更深刻的理解。
感谢各位助教一学期以来不断为我们答疑解惑，提供新知。
感谢技术组的各位巨佬，为我们提供了人性化的平台和接口，为大作业比赛的建设辛勤付出。
感谢你群各位大佬，从你们的交流中我们学到了很多新的知识，获得思路的启发。
感谢远在元培的数科大佬，向我们传授了麻将程设的经验，并提供了珍贵的参考文献。
感谢在另一个小组里的室友，我们虽然是竞争对手，但却可以进行深入的探讨，做到双赢。
感谢腾讯为我们提供了交流合作的平台。
感谢**让我们能够获取更多的海外学术知识（危）。
感谢 python 之父。
感谢北大。

6 参考文献

https://www.cnblogs.com/mumuxinfei/p/4415352.html
https://blog.csdn.net/wenjianmuran/article/details/90633418
https://blog.csdn.net/qq_32767041/article/details/80634244
https://blog.csdn.net/wsh596823919/article/details/80753087
https://blog.csdn.net/qq_44516149/article/details/106399289
https://www.cnblogs.com/mumuxinfei/p/4415352.html
https://www.zhihu.com/question/54217135
参考书籍：《快乐写游戏 轻松学编程：PC 游戏编程》by 陈其

数据结构与算法课程实习作业报告

吴嘉晟* 曹朗 韩非 刘升葳 张亦弛

摘要：

我们的算法以calculate方法这一递归函数作为主要架构，其运行原理可以抽象为以函数调用栈实现的minimax决策树（进行了α-β剪枝）。最终版本的代码估值函数只考虑了棋子等级，搜索深度达到5层，在天梯赛中显现出许多出乎意料的优秀品质，最终成绩离出线仅一名之差。对比组内其他代码，最终版本的成功启示我们本次大作业中搜索深度或许比估值函数的复杂度更为重要。

关键字：

minimax决策树；α-β剪枝；递归

1. 算法思想 2
1.1. 总体思路 2
1.2. 算法流程图 2
1.3. 算法运行时间复杂度分析 3
2. 程序代码说明 4
2.1. 数据结构说明 4
2.2. 函数说明 4
2.3. 程序限制 5
3. 实验结果 5
3.1. 测试数据 5
3.2. 结果分析 6
3.3. 经典战局（可选） 7
4. 实习过程总结 10
4.1. 分工与合作 10
4.2. 经验与教训 11
4.3. 建议与设想 12
5. 致谢 12
6. 参考文献 12

1. 算法思想

1.1. 总体思路

我们的算法以进行α-β剪枝的minimax决策树作为主体，决策树的功能由递归函数与估值函数实现。

我们在算法策略中的前提假设为对手AI的决策方法与我方AI的决策方法一致。

在天梯赛中，各组都在与对手的不断比赛中调整自己的算法，因而如果我们只采用单一的估值函数则可能产生被克制的情况，故为最小化这一损失，我们分别在前后期使用两种估值函数，前期的估值函数A考虑棋子等级和盘面状态（即棋子分布，此处应用了组内最初集中讨论所得的策略如边角处大棋子的摆放等）两项，时间复杂度较高，搜索层数较低；后期的估值函数B仅考虑棋子等级一项，降低了时间复杂度，提高了搜索层数。经调试，代码组认为使用时间复杂度较低的估值函数B搭配较深的搜索深度的代码（5.27版）效果较优。

我们的算法经历了如下的改进过程：

根据普通2048策略写出的代码→未剪枝的minimax决策树（搜索深度最大为3层，采用估值函数A）（5.25版）→α-β剪枝的minimax决策树（搜索深度最大为4层，采用估值函数A）（5.26版）→α-β剪枝的minimax决策树（搜索深度最大为5层，采用估值函数B）（5.27版）

1.2. 算法流程图

最终版本（5.27版）：

1.3. 算法运行时间复杂度分析

算法运行时间应当主要由估值函数的计算时间与minimax决策树的搜索时间（calculate方法的递归调用）构成，二者所占比重由估值函数的复杂度与minimax决策树的最大搜索深度决定。

在最终版本（5.27版）中：

assessValue方法对各棋子的级别进行指数运算并相加（或相减），由于级别是整数，故根据Python中指数运算符"**"的相关实现，该方法的时间复杂度为O(log(x))（x为场上最高的棋子级别）。当对局数量有上限时，x有上限，设该上限为m，故此时可以认为时间复杂度是O(log(m))的。assess方法的时间复杂度则为O(1)，且assess方法至多调用一次assessValue方法，故估值函数部分整体的时间复杂度为O(log(m))。

对于决策树部分，设双方在position阶段平均有n个位置可以下棋，最大搜索深度为h。若当前阶段是position阶段，则当前阶段在决策树中对应的节点平均有n个子节点，存储所有决策结果并找出其中的最值的时间复杂度为O(n)；若当前阶段是direction阶段，则当前阶

段在决策树中对应的节点最多有4个子节点，存储所有决策结果并找出其中的最值的时间复杂度为O(1)。若不考虑剪枝效果则最终平均时间复杂度为$O((4n)^{0.5h})$，若假设剪枝率为50%则最终平均时间复杂度为$O((n)^{0.5h})$。

综上，在最大搜索深度为5时，我们的算法时间复杂度为$O(n^2\log(m))$。（因为叶节点为估值函数）其中：n为双方在position阶段可以下棋的位置的平均数量；m为有限的对局数量下，场上可能产生的最高的棋子级别。

2. 程序代码说明

2.1. 数据结构说明

算法按照平台要求定义了Player类。

算法以calculate方法这一递归函数作为主要架构，其运行原理可以抽象为以函数调用栈实现的minimax决策树（进行了α-β剪枝）。此外也使用了列表、字典等Python自带的数据结构。

2.2. 函数说明

我们的算法在Player类中定义了以下方法：

__init__方法与output方法：

- 按平台规范要求进行定义。

assess方法与assessValue方法：

- 传入参数为board，belong，返回一个在区间[-100,100]之间的值；
- 二者共同构成估值函数，根据传入的board参数为belong参数代表的一方返回一个在区间[-100,100]之间的值，值越大则对belong参数代表的一方越有利；
- 估值方法基本能保证优先合并级别高的棋子。

function方法：

- 传入参数为currentRound，board，mode，返回值与output方法要求的返回值一致；
- 根据传入的board参数与mode参数，调用calculate方法进行决策（若mode为“_position”或“_direction”则不调用calculate方法，直接返回空）并返回所有返回值中最大值对应的操作；
- 为满足currentRound≤499的边界条件，调整了临近终局时的搜索深度；
- 可以认为function方法是在为calculate方法实现的minimax决策树定义根节点。

calculate方法：

- 传入参数为currentRound，board，belong，mode，depth，before，返回值为一个

在区间[-100,100]之间的值；

- 通过递归实现minimax决策树的功能并进行α-β剪枝；
- 递归的基本结束条件为达到最大搜索深度maxDepth（可视作达到叶节点），同时返回估值函数的估值；
- 我方剩余时间反映了我方超时的可能性，我方剩余时间与对方剩余时间的差值反映了我方比对方先超时的可能性（0.01s为某方每局平均可用时间），每次调用calculate方法时由上述二者确定最大搜索深度maxDepth，将maxDepth与代表当前搜索深度的depth参数对比可知是否满足基本结束条件；
- before参数可确定本次调用进入的节点与其父节点分别是min节点还是max节点。

2.3. 程序限制

当前提交至天梯的最终版本（5.27版）运行良好，尚未发现bug；但由于最大搜索深度为人工调试所得，故在时间限制等边界条件改变的情况下需要重新调试设置。

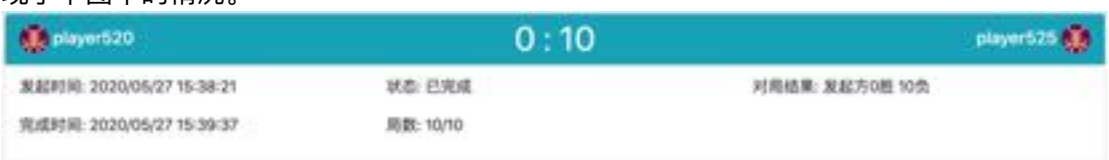
3. 实验结果

3.1. 测试数据

实验环境说明：

- 硬件配置：CPU 1.6 GHz 双核Intel Core i5 内存 8 GB 2133 MHz LPDDR3
- 操作系统：macOs Catalina 版本 10.15
- Python版本：3.7

我们选择了直接使用代码竞技场对我们的代码进行测评。由于使用了老师在课上讲到的搜索+剪枝+估值函数的方式，我们在初期构想的诸如将最大块限制在角落、“蛇形”按大小顺序摆放等的策略被抛弃，并且被证明是完全在使用人类的思维方式限制拥有大量算力的计算机程序。在一次用搜索+剪枝+估值函数的后来的程序与原本“以人度机”的程序的对局中，比分是惊人的10-0，并且出现了**2048! 24回合塞爆! 3级压胜!** 的惊人场面，并一度出现了下图中的情况。



在确定了使用搜索+剪枝+估值函数的策略之后，由于完全抛弃了启发式规则，我们对于程序的发挥完全没有底气，对于程序中的各种参数也是两眼一抹黑——我们不知道使用的搜索深度会因为太深而超时，或是因为太浅而决策很弱；我们也不知道估值函数会表现如何，而且更加让人头疼的是，并不像搜索深度仅仅是一个参数，估值函数有着无限的可能性，而且其中的参数也更加多元。

在估值函数和搜索深度调试的过程中，另一位小组成员将较为复杂的剪枝部分程序写了出来，加入到程序中后，继续进行测试，在对搜索深度和估值函数做了改变和初步测试之后，累计在竞技场平台上提交了三种参数的程序，分别为三层搜索+未剪枝+复杂的估值函数（5.25版）、四层搜索+剪枝+复杂的估值函数（5.26版）、五层搜索+剪枝+简单的估值函数（5.27版），其中，复杂的估值函数同时考虑了棋子和棋盘的状况，而简单的估值函数仅仅考虑了棋子的状况，最后在相互间以及与官方bot的测试结果如下（胜/负）：

	SSS	SS	S	A	B	C	D	5.25	5.26	5.27
5.25	0/9	2/7	3/22	5/5	7/18	10/0	5/5		34/26	0/10
5.26	0/7	2/15	14/21	12/32	17/18	7/3	16/4	26/34		13/14
5.27	1/13	8/20	30/26	25/26	41/14			10/0	14/13	

此外，在竞技场的总体表现中，三个版本分别得等级分1112.18、1021.95、1205.22。结合以上数据，我们得到了5.27版本最优的结论。我们发现，考虑了棋盘状况的估值函数并没有更优，反而是拖累了计算的速度，因此，回归简单的估值函数之后，程序发挥出了更加优秀的能力。

3.2. 结果分析

能够明显发现的是，再加入了剪枝并且将搜索深度加深到五层时，程序发挥出了比原本好得多的效果。利用老师上课讲过的minimax决策树，与程序表现关系最直接的毋庸置疑应该是搜索深度，因此在没有特别优的策略的情况下应该将算力更多的留给搜索，而不是浪费在估值函数上。值得一提的是，我们小组的另一名成员曾经尝试过寻找并运用更好的复杂的估值函数，但是最后因为估值函数占用的时间太长而导致不仅搜索层数并不尽如人意，甚至还会出现超时的状况，当然，走过这样的弯路使得我们对于算法有了更深刻的了

解。事实上，复杂的估值函数并非能够更加好地反映局面的状况，不同的权重设置和计算方法都会对估值函数本身有着非常大的影响，哪怕是一点点的改变都很容易使得估值函数发生天翻地覆的变化，因此，在没有非常好的数学方法能够得到近乎完美的估值函数的情况下，我们小组最终使用简单估值函数以换取搜索深度的方法得到了不错的成效。此外，对于搜索能力有明显提升的，是采用的剪枝策略，大大减少了决策树搜索时多余的计算，能够用多余出来的时间充分利用于更深的搜索，使得算法有了更高的提升。

在天梯赛中，我们小组的成绩虽然不能和最高的几位比，但是依然稳扎稳打地取得了还算过得去的成绩，在开局不利仅仅排在17位左右的情况下，曾一度杀进了前十，但是最终还是没能出线，在离出线只有区区三分的情况下，在胜率高于前面五队的情况下，遗憾无缘出线。

F组					1/1 刷新
链接	排名	作者	战绩	天梯积分	
进入	#1	F19 Echo组	70.4% in 5369	1763.61	
进入	#2	F19 Uniform组	74.9% in 5546	1721.12	
进入	#3	F19 Zulu组	51.3% in 4330	1676.65	
进入	#4	F19 Golf组	49.9% in 3970	1653.81	
进入	#5	F19 November组	49.0% in 7230	1617.47	
进入	#6	F19 X-ray组	56.0% in 7264	1567.56	
进入	#7	F19 Bravo组	50.7% in 6460	1557.7	
进入	#8	F19 Sierra组	52.2% in 6734	1532.63	
进入	#9	F19 Tango组	46.0% in 7357	1405.06	
进入	#10	F19 Kilo组	50.6% in 3747	1392.51	

进入	#4	F19 Golf组	50.6% in 6997	1630
进入	#5	F19 November组	49.8% in 10190	1613.72
进入	#6	F19 Sierra组	51.5% in 9801	1582.77
进入	#7	F19 X-ray组	53.6% in 10374	1582.67
进入	#8	F19 3721组	49.2% in 10649	1546.15
进入	#9	F19 Bravo组	49.9% in 10150	1542.42
进入	#10	F19 007组	49.1% in 8060	1492.93
进入	#11	F19 666组	50.0% in 10928	1484.07
进入	#12	F19 Romeo组	50.3% in 7857	1457.92
进入	#13	F19 Kilo组	50.4% in 7137	1454.76

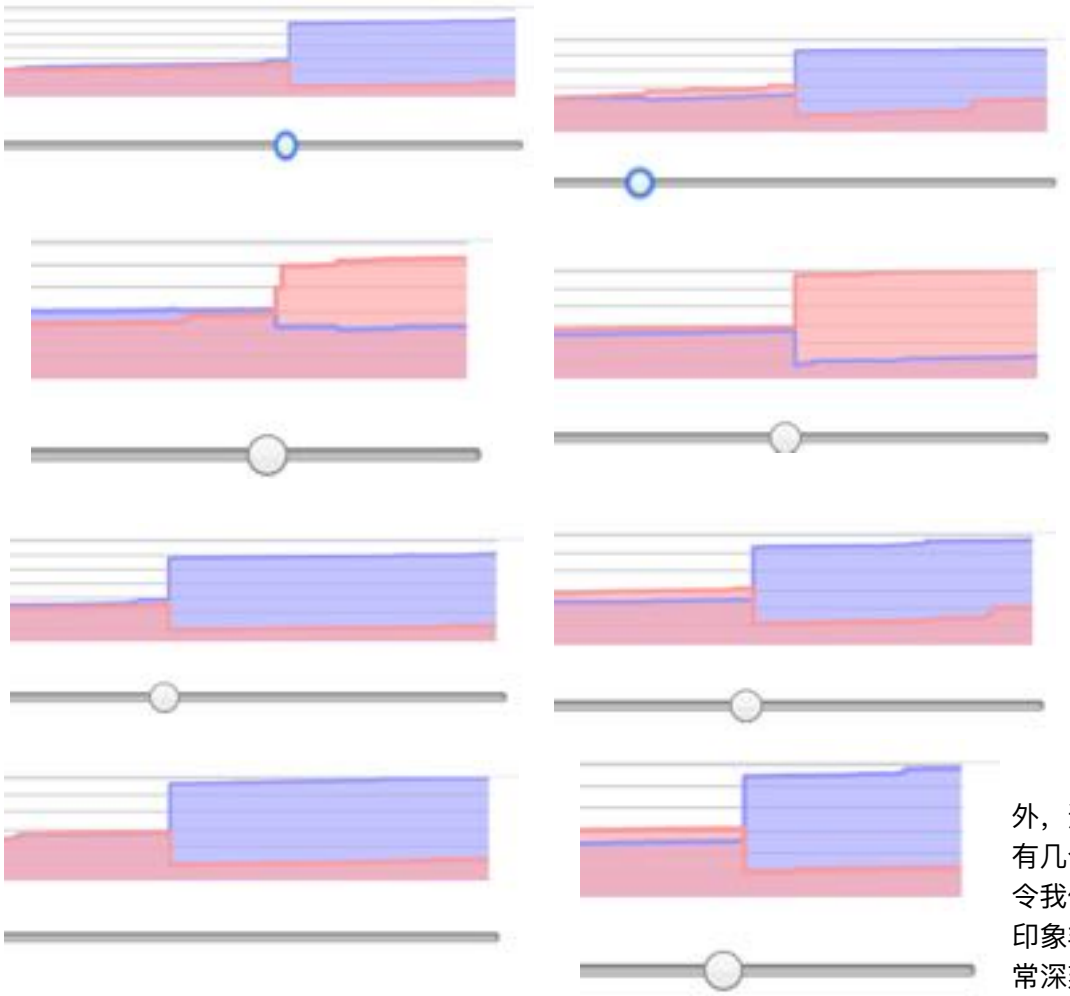
左：巅峰时刻，右：最终排名

简单抽取了几个实战的例子，我们能够发现，在实战测试中，我们小组的算法表现出了许多优秀的没有预料到的品质，比如会通过吃对方的棋子逐渐"深入敌境"限制对方操作的空间，也会通过牺牲较小点数的棋子去吃对方较大点数的棋子，经常能够通过吃掉1024或是2048取得胜利（按照天梯从8算起的话）。

3.3. 经典战局（可选）

在比赛中我们的算法出现了许多惊人的战局——找准时机吞并对方的1024，甚至是2048，都已经不是什么新鲜事。

还怪能吃的

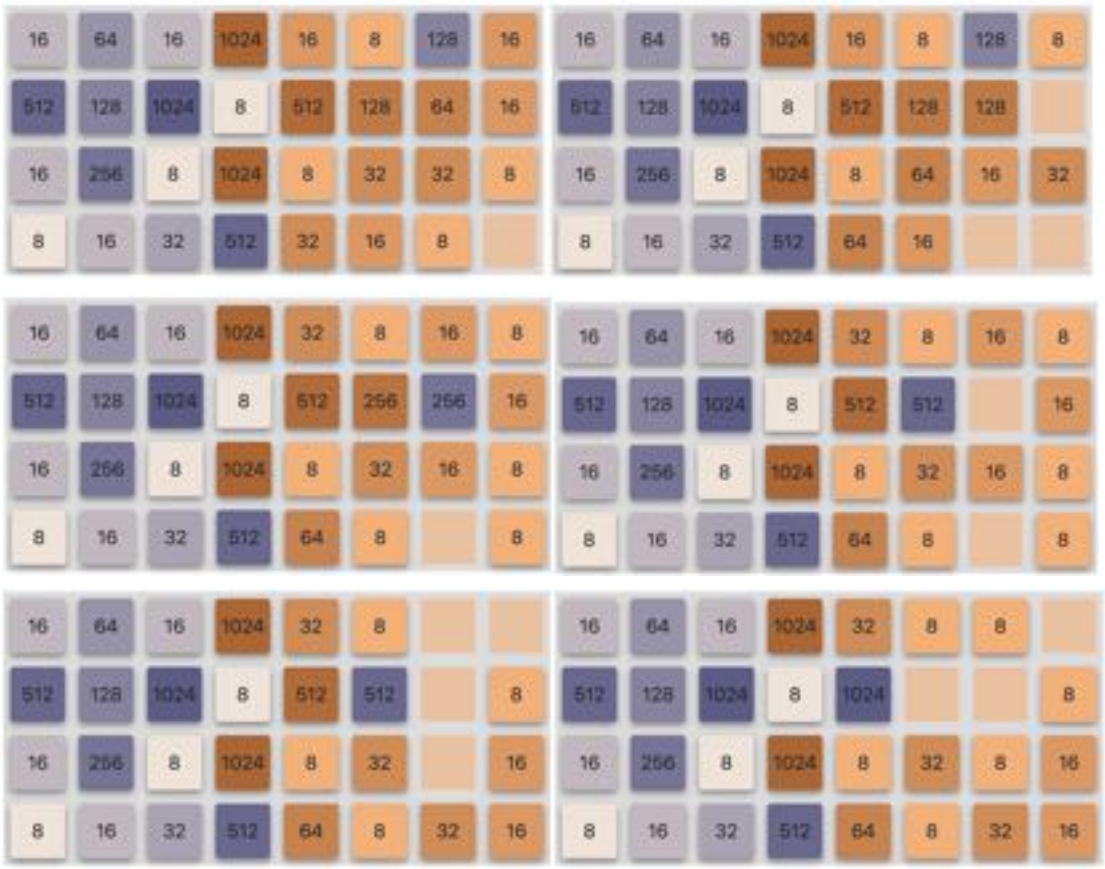


此外，还有几个令我们印象非常深刻的经典情况，也非常有趣。

首先是这样的一局（我方为先手），我们的算法在已经将两个棋子深入对方的棋盘的时候，再将我方的512送到了对方的512嘴边，本来就因为两个位置被侵占即将塞爆的对方自然笑纳了这份大礼，于是，等在512后面的1024立刻将对方新的1024归为己有，让对方陷入塞爆的境地，还顺势合成了4096，取得胜利。

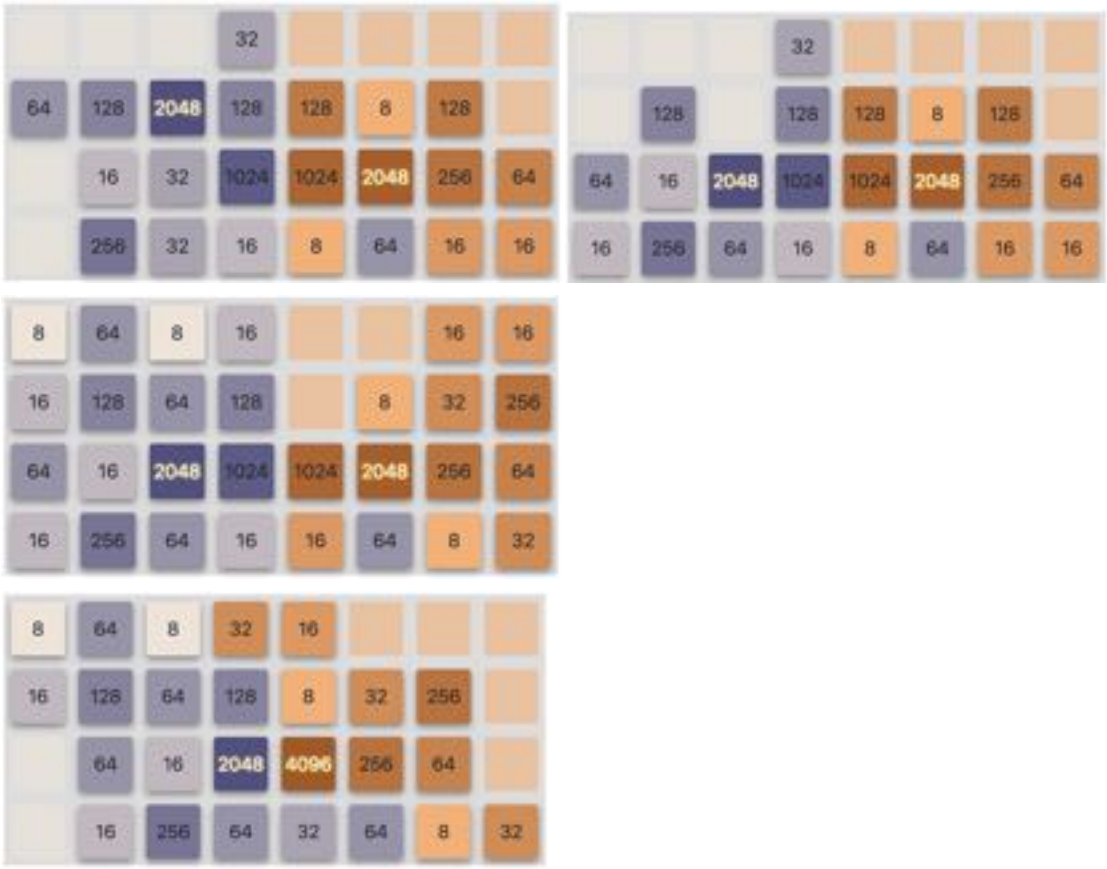
螳螂捕蝉

然后是这样让人摸不着头脑的一局（我方为先手）——对方的算法咄咄逼人，比赛还未过半，就已经让我方达到了塞爆的尴尬境地。在随机落子的帮助下，我方苟延残喘着又获得了机会，却仅仅又落得了被塞爆的境地。在对方已经取得了明显的优势的情况下，哪怕是任由自己被塞爆，对方也能取胜。然而，对方鬼使神差地把一个128送到了我方在敌阵中的128旁，于是我方笑纳，吃成了256。哪怕到了此时塞爆，对于对方也是必胜的结果，然而偏偏对方又把一个256送上门来，作为唯一能够合并的方向，我方算法自然吃掉了256。但是对方就在我方现在的512旁边的512却匪夷所思地并没有吃掉我方的512，而是朝下合并，最终给了我方可乘之机，顺势吃掉了对方的512，吃出了两个1024，虽然仍然处于塞爆的状态，但是对方已经没有空间造出两个512来与我方抗衡了，自然也就落败了。



在这局中，我们发现，游戏算法很重要的一点是大局，而不是眼前的利益，这样的大局只有靠搜索的深度才能得到。如果对方没有这么强的攻击性，如果对方多预测两步，或许就能够发现被吃掉的命运，而从一开始就不选择做愚蠢的决定，而导致自己在开局非常顺利的情况下被以一种甚至有一丝荒谬的方式逆转，但或许这就是二零四八的魅力吧。

最后一个令我们大呼过瘾的例子，则完全是高手的较量（我方为后手）。在双方都已经有了一个2048和一个1024之后，首先我方将1024和2048一起送到了对方1024的旁边，拿1024作为诱饵，如果对方吃掉我方的1024，我方立马可以将生成的2048吃掉。然而，对方不仅不为所动，还把本方的2048挪到1024旁进行防守，进入了僵持的状态。接下来，便完全是我方发挥的时刻。这时，我方以两个64微微落后，但似乎是铁了心想要通过这个方式取胜似的，我方的算法放弃了自己的成长，一旦对方棋盘中存在空格就向对方棋盘中落子，最终将对方逼到了无路可走不得不吃掉我方1024的地步，于是我方2048顺理成章地将对方新的2048吞并，最终取得了胜利。



高手过招

在多层搜索的帮助下，我们的算法的确远见了得，知道要追求最大的利益，而非安于现状，知道放弃短期自身的增加转而着眼逼着对方送给自己1024并且最终取得成功。

4. 实习过程总结

4.1. 分工与合作

我们小组一共五人：吴嘉晟（组长）、曹朗、韩非、刘升葳、张亦弛。其中，韩非和刘升葳负责文档撰写，吴嘉晟、曹朗和张亦弛负责代码编写和测试，具体地，曹朗负责搜索和剪枝部分的代码，吴嘉晟和张亦弛负责估值函数的代码和测试。

小组在创建过程中就经历了不少的困难，在经历了原先组员退课跑路等挑战后，最终完成了组建。由于正好遇上了COVID-19大流行，疫情期间大家都被关在家里，无法在现场进行交流，甚至无法返校或是现场上课，小组成员们甚至到现在都没有彼此都没有见过。然而，这并没有阻碍成员们合作的热情和努力。在5月16号时，小组便召开了会议，安排了任务，大家积极参与出谋划策。

此后，小组没有再召开正式的会议，但是小组成员们非常积极，一直在群内实时交流并且更新成果，信息交换非常流畅，按时完成了各项任务，配合默契，分工明确，也取得了还算不错的成果。当然，这样合作的效率不能和面对面开会相比，但是在疫情期间，能有这样的合作完成度，已经是不错的结果了。

4.2. 经验与教训

计算机的运算能力跟人类根本不在一个水平，在程序的考虑时，如果忽视这样的差异，纯粹以人类的思维方式编写程序，会大大局限代码所可能完成的高度。计算机最强大的便是其强大的计算能力，我们就是要利用它超强的计算能力，去完成人类所做不到的大量计算和分析，以此方法得到人类智慧所分析不出的优秀决策。如果使用人类的思维方式去局限计算机程序，那么计算机程序永远超不过人类，只有打破思维定式，利用全新的眼光去审视问题，才有可能能够突破人类的极限，真正做到充分利用计算机的算力。在这次的实习过程中，我们小组首先编写了一份根据平时2048游戏的程序，代码长度高达500多行，但是成绩并不理想。在了解了上课老师讲解的minimax决策树之后，我们小组改用了这种方法，最终仅用了不到250行的代码写出了比原来强大数倍的程序。

此外就是，如果使用搜索的方式，那么最重要的，就是搜索深度。如果能够提升搜索深度，那么别的一切都没有这点来的重要。我们小组花了大量的时间和精力改进我们的估值函数，最后估值函数有没有变强不知道，但是搜索深度显然是变浅了，而且还时不时超时判负。这样的尝试并不是坏事，至少帮助我们得到了关键的道理——深度为王。最后我们并没有采用花了大量时间精力写出来的估值函数，而是还是使用了简单的估值函数，最后通过简单的估值函数以及剪枝使得搜索深度能够有所提升，从而能够使得整个程序得到更优的结果。

4.3. 建议与设想

本次实习作业，对于我们组来说，基本还是算顺利的，在过程中也基本没有遇到什么太大的困难或者是阻挠。能够改进的其中一点自然是由于线上上课，沟通自然不如线下上课那样足，可能会导致少量的滞后或者是不畅，但是基本上，还是保证了效率和效果。此次作业包括平台的搭建、游戏的设计，相信老师助教和技术组的同学们都付出了大量的努力，也可以看到，最终呈现的成果非常完善。作为差一名就能够出线（多整整两分！）的队伍，自然是觉得有点遗憾的，要是能够加一点出线名额就好了，当然，不管怎么加都是会有差一名出线的队伍的，而如果直接开始淘汰赛，又未免随机性太强，所以权当是在一个很不错的系统中的倒霉蛋吧！

我当时选陈老师的数算是跨院系选课来的，一开始是因为听说给分好云云选择了这门课，依稀记得第一节课的时候老师就说，我们这门课的特点是任务量非常大，而且会很杂，当时没有概念，硬着头皮上，害，能有多忙。结果第一次作业便傻了眼，然后经过一次又一次H的洗礼，经过了无数个日落和无数个日出，有过怀疑有过抱怨，但是最终还是觉得跟着陈老师虽然很累，但是真的有学到很多，完全值得。所以如果说要给学弟学妹什么建议，那估计就是告诉他们课程的确会比较累，但是选了绝对不会后悔。

5. 致谢

首先感谢陈老师和助教们一个学期的辛勤付出，给老师助教们添麻烦了。感谢技术组的同学们，为我们搭建了这么好的平台，也感谢所有别的组的对手同学们，一起奉献了这么一次精彩的大作业体验。感谢选课网让我饭也没吃坚守着电脑在卡了一个半小时之后终于选上了这门课，感谢腾讯公司为我们小组提供了交流的平台，感谢小黑一直配合地上镜让我们都过了一把云撸猫的瘾头，感谢所有为了这门课付出过的人，包括以前的助教们同学们，以及为我们提供了美丽的校园环境的叔叔阿姨们，谢谢大家了！

6. 参考文献

[1] http://gis4g.pku.edu.cn/course/pythonds/
[2] https://github.com/pkulab409/sessdsa.2048/

数据结构与算法课程实习作业报告

董天宇、董自豪、符凯蓬、李鹤轩、要昌雨*

摘要：本代码是运用了 minimax 搜索方法，采用了α - β剪枝，给出当前棋盘较好决策的贰零肆捌 AI 程序。α - β剪枝是对自定义的节点类进行的搜索，并且利用了储存功能来节约时间。而比较依据是自定义的评分函数，其中考虑了所有棋子和己方领域大小以及双发子力不平权的影响。评分函数需要用到三个 magic number，这些数据由一个自动化的对战程序实现。

关键字：minimax 算法； α - β剪枝；数据结构——树

1 算法思想.....2
1.1 总体思路.....2
1.1.1 搜索树的节点：.....2
1.1.2 剪枝.....2
1.1.3 评分.....2
1.2 算法流程图.....3
1.3 算法运行时间复杂度分析.....4
2 程序代码说明.....5
2.1 数据结构.....5
2.1.1 节点 Node.....5
2.1.2 player 类：.....7
2.2 函数说明.....8
2.2.1 评分函数.....8
2.2.2 α-β剪枝.....9
2.3 程序限制.....10
2.3.1 错误和超时.....10
2.3.2 局限性.....10
3 实验结果.....10
3.1 测试数据.....10
3.2 结果分析.....13
3.2.1 本地运行结果（主要目的：比较参数的影响）.....13
3.2.2 天梯测试结果（主要目的：控制时间、发现重大漏洞）.....13
3.2.3 算法效果：.....13
3.3 经典战局.....13
4 实习过程总结.....14
4.1 分工与合作.....14
4.2 经验与教训.....15
4.3 建议与设想.....15
5 致谢.....16
6 参考文献.....16

1 算法思想

1.1 总体思路

在本次的大作业中，我们组的代码总体的思路是：对每个棋局评分+搜索树；在搜索的过程中使用 alpha-beta 剪枝以减少用时。我们小组采用了启发式的规则设计评分函数，使之更倾向于在比赛过程中在对方棋盘下子以阻碍对方的合并步骤。在剪枝时，也参考了一些启发性的规则，人工排除掉了一些不必考虑的情况。

1.1.1 搜索树的节点：

我们小组定义了搜索树的节点，即 Node 类，它记录了当前棋局的状态。它拥有的属性如下：1.isFirst（我方是否先手），2.nowFirst，3.nowMove，（分别用来记录下一步轮到哪一方行动，行动为落子还是移动），4.board（记录当前的棋盘），5.value（记录棋盘的评分），6.currentRound（当前的轮数），7.stop（记录比赛是否已经停止，用来防止报错），8.need，9.done（分别记录是否已经生成了子节点列表和是否已经打过分）10.subs（即子节点列表）

树节点具有三个方法:span（生成子节点的列表），getValue（返回本节点的棋盘的估值）以及 getOpr（返回本节点相对其父节点进行的操作）。在 Span 方法中，我们定义了 span_subs 函数，用来根据 nowMove 和 nowFirst，以及当前的棋盘状况（可能没有可行的方向或位置）来生成子节点列表。如果是移动（nowMove）的话，同时还要根据其得分进行排序，这样可以更快的搜索到较优解。

为了减少生成节点的时间，我们小组每次在 player 类里面缓存了根节点。因为根节点的子节点有很大概率已经遍历过了，那么这样就可以很大程度上节点的创建过程，从而节约了时间。

1.1.2 剪枝

我们小组采用的是常规的 alpha-beta 剪枝算法，但是为了优化搜索树的性能，我们组对于每个节点只搜索前六个子节点。因为在己方落子是子列表的第一个，而且只有当棋子比较多的情况下才会用到封堵对面的操作，所以这样的搜索方法漏掉最优解的概率非常小，可以认为这样的算法是合理的。

1.1.3 评分

评分的函数总体上有三个版本，第一个版本考虑到了棋局的平滑性，两边的棋子的大小，空格数以及相互侵占的情况，但是事实证明，因为考虑平滑性而花费的时间过多，导致搜索层数过少，ai 变得非常菜，所以我们很快放弃了这种想法

第二个版本的评分函数改成了

```
def evaluate(isFirst,board):
    return sum(map(evalf,board.getScore(isFirst)))-sum(map(evalf,board.getScore(not isFirst)))
```

之后，为了增强我们的算法的进攻性，我们又将评分函数改成了

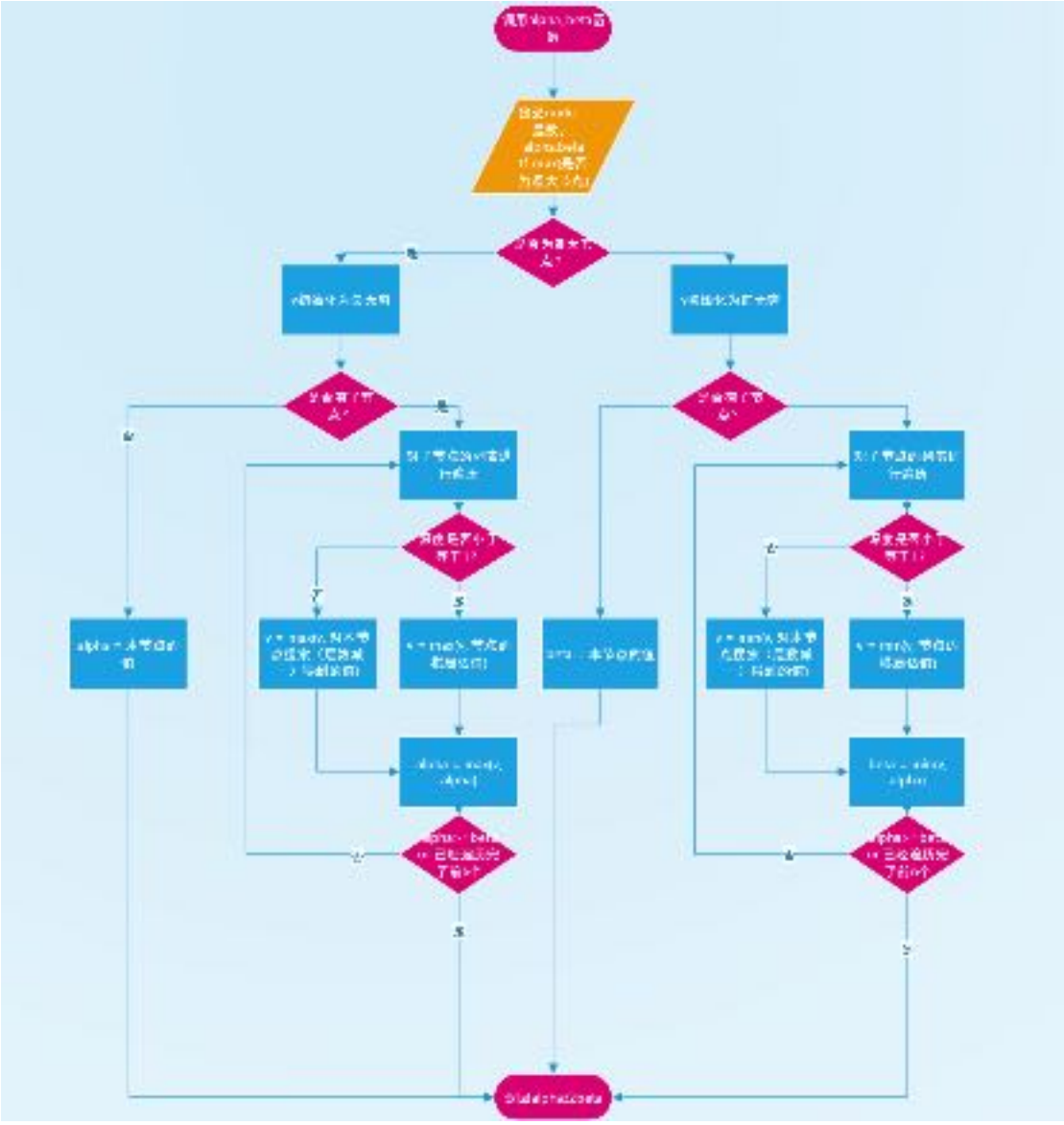
```
def evaluate(isFirst,board):
    return 0.05*sum(map(evalf,board.getScore(isFirst)))-sum(map(evalf,board.getScore(not isFirst)))
```

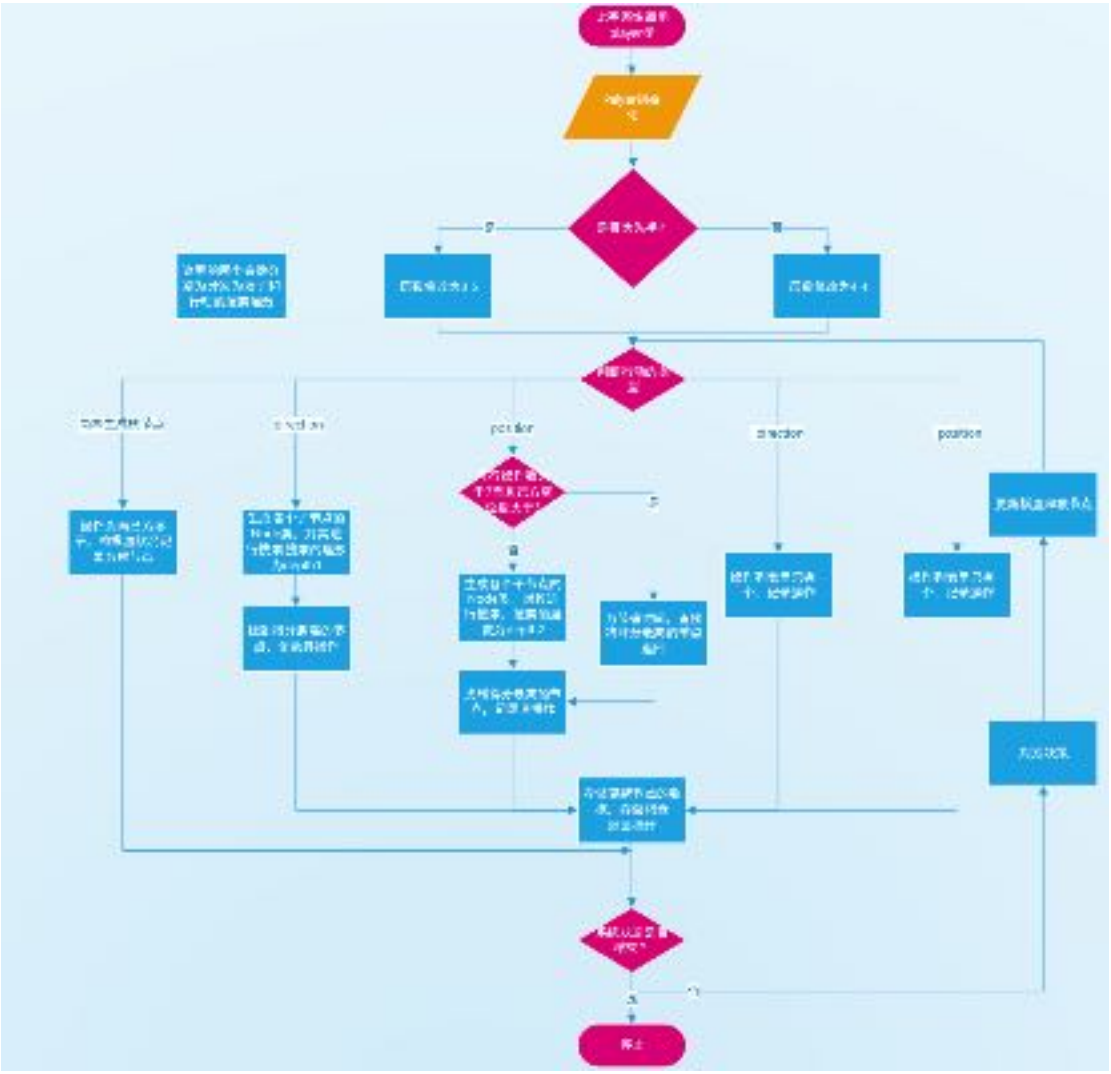
这样搜索的深度大大的增加，由原来的 2-3 层增加到 5-6 层左右，实力有了明显的飞跃，而且有了一定的进攻性。但是在面对一些有启发式规则的 AI 时，我们仍然以较大的劣势败北（比如 RBQ4），于是我们改进代码，有了下面的第三个版本。

第三个版本的估值考略了我方的领地数，即我方棋子列表和空位列表的长度和，对于棋盘的评分将随着领地数的增加而迅速增加，这样我们的 AI 就会倾向于封堵对面。若占据了对方的空位的话，也倾向于继续占领。

1.2 算法流程图

由于总图过大，所以对 α - β 的流程图和 output 的流程图分开。





1.3 算法运行时间复杂度分析

程序运行时间主要用来构建搜索树以及对棋盘进行评分，生成每个节点的时间主要用于棋盘的复制和对棋盘评分，棋盘的位置数是固定的，所以棋盘复制是 O(n)的，而评分是 O（n）的，n 为己方的棋子的个数，但是由于棋子数上限为 32 ，比较小，所以可以把进行一次评分的时间看作 O（1）的。这样得出构建一个节点的时间可以看作是常数，但是这个时间实际上比较长，因此拖延了层数的加深。

如果下一步是移动的话，节点的子节点数至多为 4，而落子的话则可行的区域则至多有 17 个，但是由于有搜索时有截断的机制存在，实际上在落子时大部分的节点被创建但并不会继续深入搜索，所以子节点数平均来看仍然不会超过 6。

通过以上的分析，可以看出，搜索树的时间复杂度是搜索层数的函数，复杂度应该介于 O（4^n）到 O（6^n）之间。虽然对于子节点进行排序会是 O（$n\ ln(n)$）的，但是这样也会使剪枝进行的比较顺利，从而减少时间的占用。理想中，如果最优解在搜索到的第一个位置出现的话，时间复杂度将是线性的，并且如果对手的操作也符合预期的话，由于储存功能甚至会使只要是搜索的节点都被用到了没有任何浪费。但是实际上由于不清楚在运行过程中剪枝的多少，我们现在无法给出程序的时间复杂度的特别精确的估计。

2 程序代码说明

2.1 数据结构

2.1.1 节点 Node

基于数据结构：树
相比普通的树，我们改进的地方是，所有的子节点都是以最适合探索的方式排好序的。
没有建立从子节点到父节点的连接，只保留了从父节点到子节点的操作（节约内存）

2.1.1.1 初始化：

```
class Node:

    def __init__(self,opr=None,\
                 isFirst=False,\
                 nowFirst=True,\
                 nowMove=False,
                 board=None,\
                 currentRound=0,
                 stop=False):
        self.opr = opr    #记录上一步的操作，用于维护和储存
        self.isFirst = isFirst
        self.nowFirst = nowFirst
        self.nowMove = nowMove
        self.board = board
        self.currentRound=currentRound
        self.stop=stop    #是否已经结束，避免报错
        self.need=True    #是否需要span，避免掉多次调用耽误时间
        self.Done=False   #是否评过分，避免多次调用耽误时间
```

定义了棋盘所需要的所有参数：与父节点相连的操作、己方先后手、棋盘所属先后手、棋盘状态（该移动还是落子）、棋盘、当前轮数、截止条件。

初始化：未曾评分、需要探索子节点。

2.1.1.2 span 方法：

返回对应的**所有**的子节点列表（已按最适合排序的方法排号序了）

```
def span_subs(board, isFirst, nowFirst, nowMove, currentRound):
    #生成所有可能的子节点并按照较为容易剪枝的顺序排列
    #返回列表
    result=[]

    if nowMove:

        #修改盘面的基本信息
        if nowFirst:
            isStop=False
        else:
            currentRound=currentRound+1
            isStop=currentRound==MAX_ROUND

        #生成合法输出
        for i in range(4):
            newboard=board.copy()
            if newboard.move(nowFirst, i):
                newNode=Node(opr=i, \
                             isFirst=isFirst, \
                             nowFirst=not nowFirst, \
                             nowMove=nowFirst, \
                             board=newboard, \
                             currentRound=currentRound, \
                             stop=isStop)
                result.append(newNode)

        #没有可合并方向直接更新盘面信息
        if result==[]:
            return [Node(opr=(), \
                         isFirst=isFirst, \
                         nowFirst=not nowFirst, \
                         nowMove=nowFirst, \
                         board=board, \
                         currentRound=currentRound, \
                         stop=isStop)]

        #返回排序后的列表（自己轮分高优先，对手轮分低优先）
        return sorted(result, key=sort_key, reverse=isFirst==nowFirst)

    else:
        #获得可用位置
        a=board.getNext(nowFirst, currentRound)
        b=board.getNone(not nowFirst)
        available_pos=[a] + b if a else b

        if available_pos:
            for i in available_pos:
                newboard=board.copy()
                newboard.add(nowFirst, i)
                newNode=Node(opr=i, \
                             isFirst=isFirst, \
                             nowFirst=not nowFirst, \
                             nowMove=not nowFirst, \
                             board=newboard, \
                             currentRound=currentRound)
                result.append(newNode)
            #生成的列表已经是按评分函数的己方高分优先顺序排列好，无需排序耽误时间
            return result

        #无可用位置直接更新信息
        else:
            return [Node(opr=(), \
                         isFirst=isFirst, \
                         nowFirst=not nowFirst, \
                         nowMove=not nowFirst, \
                         board=board, \
                         currentRound=currentRound)]
```

如果是第一次调用此方法，则会在返回之前根据该节点对应的一系列属性调用 span_subs 函数探索**所有合法操作对应的**子节点，并且对不存在合法操作（output 对应'_'系列）的情况进行处理——返回列表仅含更新先后手等基本属性，但继承棋盘的节点——来防止报错。

2.1.1.3 getValue 方法：

返回节点对应棋盘的估值。
如果是第一次调用，则在返回前调用 evaluate 函数获得盘面估值。（evaluate 见 2.2.1）

2.1.1.4 getOpr 方法：

获得父节点到此节点的操作。
用于 player 类中对节点的保存、维护。

2.1.2 player 类：

2.1.2.1 初始化方法：

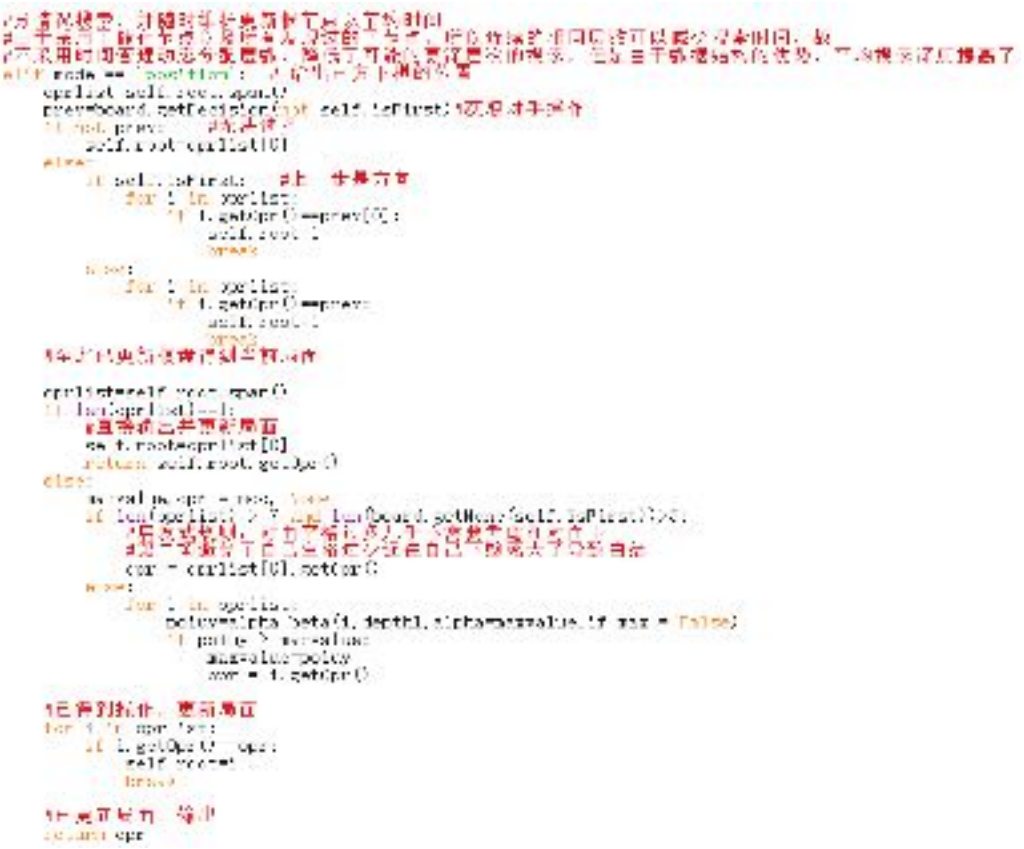
定义所需参数：是否先后手。
初始化：未定义根；根据先后手调整搜索深度：先手 6 层后手 5 层。

2.1.2.2 output 方法：

如果是第一次调用，则建立根节点，直接输出在己方落子。

```
def output(self, currentRound, board, mode):
    #全新棋盘则直接在己方落子，并初始化节点（设为根节点）
    if self.notDone:
        self.notDone=False
        newboard=board.copy()
        pos=newboard.getNext(self.isFirst,currentRound)
        newboard.add(True,pos)
        self.root=Node(opr=pos,\
                       board=newboard,\
                       isFirst=self.isFirst,\
                       nowMove=not self.isFirst,\
                       nowFirst=not self.isFirst)
        self.root.span()
        return pos
```

如果不是，则先获取对手操作，更新根节点，然后对更新后的根节点的子节点展开，对每一个子节点调用 α - β 剪枝函数（这就是为什么虽然 depth=5，但是一直都在说搜索深度是 6 的原因），α 值最高的节点对应的操作即为本局的输出。但是在输出前需要先更新根节点。



2.2 函数说明

2.2.1 评分函数

```
base=3.49

value_list = [base**i for i in range(0)]+[1*base**0]+[10*(base**i) for i in range(9,12)]
#评分函数用，以大于1024，2048，4096的影响，使得存在这样大的指标下能够执行了合成大了
def eval*(n):
    return value_list[n]

def evaluate(isFirst,board):  #评分函数
    list1 = board.getScore(isFirst)
    list2 = board.getNone(isFirst)
    a1 = sum(map(eval1,list1))
    a2 = len(list1) + len(list2) -16

    return 0.35*a1-sum(map(eval1,board.getScore(not isFirst)))*(1-0.18*a2)
```

2.2.1.1 传入参数：棋盘，棋盘归属

2.2.1.2 常数以及思路

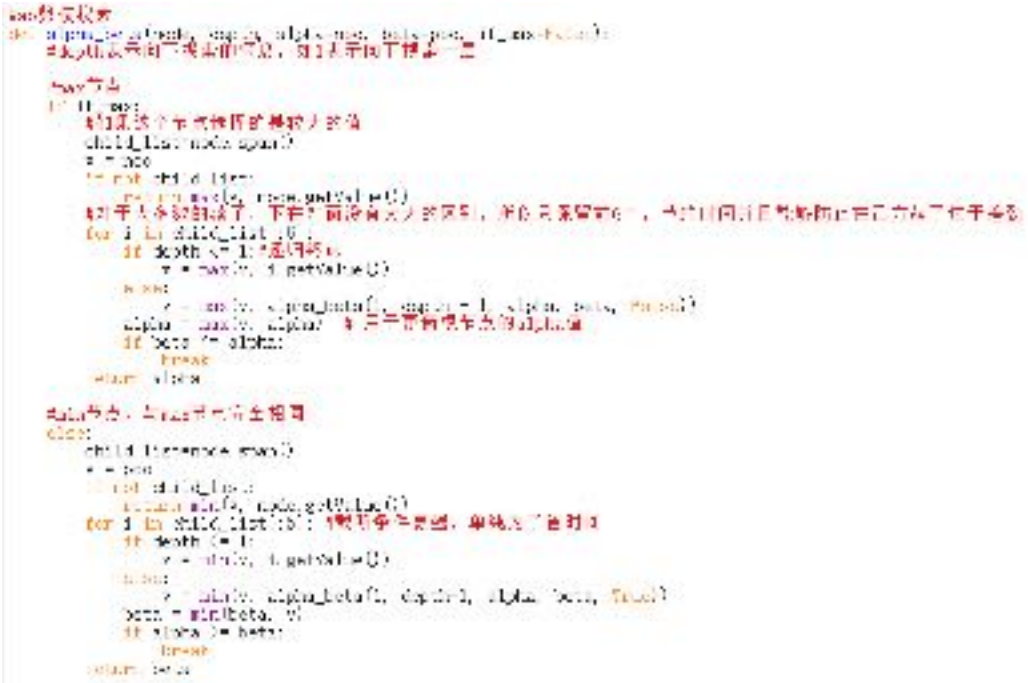
Base∈（2,4）保证了合成时的分数增加，但又不至于太高以至于程序过分贪心。大于 2 是显然的，至于为什么最后选用了 3.49，这主要归功于强大的“本地的自动化测试程序”（见第三部分）。

对于等级 8 及以上的棋子的评分相对其他进行了上调，这就使得在后期，程序将倾向于合成较大的棋子建立优势。

评分时还用到了两个 magic number 即 0.35 和 0.18，0.35 在 1/base 左右，即对方的一个棋子评分比己方高一级的棋子评分还要高，因而程序会倾向于吞并对方的棋子。a2 则记录了我方占据对方的领域的个数，这样使得当我方封死对面的合并步骤时评分会非常高，因此我们组的战局大部分依靠“塞爆”对手获胜。

2.2.1.3 返回值：当前棋盘的价值

2.2.2 α - β 剪枝



2.2.2.1 传入参数：搜索节点，深度，已有 α 、β 值，节点类型（min/max）

传入节点类使得此函数可以递归调用。

2.2.2.2 思路（可参见流程图-α-β剪枝）：

函数的实现依靠的是递归调用：当深度不足 1 的时候返回评分值，其余的节点（手动砍断某些不重要节点后）的每一个子节点都进行相同的递归调用，并且每次调用的递归深度都降低 1，这时，我们相当于把函数返回的结果当做了这个决策后棋盘将取得的评分。

α 记录了之前在己方（也就是每次搜索最大值的一方）最优决策的情况下取得的最小的值，也就是我方得分的下界，而 beta 则相反，记录的是在对方最优决策的情况下我方取得的最大值，也就是我方得分的上界，在任何一次递归之中如果出现 α > β 的情形，那么说明当前决策一定劣于之前考虑过的决策，不可能得到更好的结果，所以可以放弃对当前的树枝的搜索，

2.2.2.3 返回值

max 节点则返回 α （好情况中的最差值）；min 节点返回 β （坏情况中的最好值）。

对于 player 里的 output，返回的是展开一层之后所有节点的 β 值。

2.3 程序限制

2.3.1 错误和超时

从第一个版本开始就没有报过错。经过改进后，后来的代码也不超时了。

2.3.2 局限性

它的搜索程度不是很深，导致很早之前插入对面的棋子过了很久之后发现不能够拔出来，导致直接送给对面合成（孤军深入的另说，即使是白给也会严重影响对面）。这个问题过于难以解决，因为一般白给需要溯源到很早以前，至今未能解决。

评分函数未能考虑平滑性；将评分函数直接写入自定的 board 类应该会快很多，但是考虑到投入产出比，似乎不值得且能力不够重写一个性能更好的 board。

3 实验结果

3.1 测试数据

实验环境说明：

- 硬件配置：董自豪：CPU：Intel（R）Core（TM）i7-8565U CPU @ 1.80GHz 1.99GHz
- 内存：16GB（15.8GB 可用）
- 董天宇：CPU：Intel（R）Core（TM）i7-7700 CPU @ 3.60GHz

- 内存：16GB（11.4GB 可用）
-
- 操作系统：董自豪：Windows10
- 董天宇：Windows10
- Python 版本：董自豪：Python 3.7.6
- 董天宇：Python 3.7.6

主要的测量方法有两种：一是在本地进行参数调试，即利用测试代码生成大量仅不同参数的代码以确定最佳参数的大致范围；二是在天梯上与天梯榜前几位的优秀代码进行手动对战，用于对算法进行优化以及调整参数。第一种方法的思路与部分测试程序代码如下：

```
5  #文件个数
6  div1=0#第一个量所需等分的间隔
7  div2=0#第二个量需等分的间隔
8  num1=10#第一个量所等分成的个数
9  num2=2#第二个量所等分成的个数
10 base0=0#第一个量所等分的起始值
11 aggr0=0.17#第一个量等分成后的值
12 for i in range(num1):
13     from_file=open('C:\\python\\resources\\examples\\py\\Player.py','r',encoding='utf-8')#导入现有代码文件
14     content = from_file.read()
15
16     mark=0
17     if num2!=0:
18         while mark<num2:#对每个i生成多个不同的文件
19             f = open('player%d_%d.py' % (i,mark), 'w',encoding="utf-8")
20             f.write("constA="+str(base0+i*div1))
21             f.write('\n')
22             f.write("const0="+str(aggr0+i*div2))
23             f.write('\n')
24             for x in content:
25                 f.write(x)
26             mark+=1
27     else:
28         f = open('player%d.py' % i, 'w',encoding="utf-8")
29         f.write("constA="+str(base0+i*div1))
30         f.write('\n')
31         f.write("const0="+str(aggr0))
32         for x in content:
33             f.write(x)
34     from_file.close()
35     f.close()
```

基本思路是，先用某种规律生成大量参数不同的文件。上面的代码是将两个待调整参数在一定范围内等间隔分布；也可以改变第 20、22 行，使参数在一定范围内随机生成或按其他规律（如间隔按幂次增大）生成。

接下来需要生成 round_match 文件函数中的参数 playerlist。代码如右，由于生成的文件与测试文件在同一文件夹内，故读取文件夹中 py 文件的文件名并生成列表，再将其中的 player 文件切片即可获得 playerlist。

```
37 #列表生成
38 filelist=os.listdir()
39 l = len(filelist)
40 _l = 0
41 while l-_l:
42     l = len(filelist)
43     for file in filelist:#将同一文件夹中的py文件名循环生成列表
44         if '.py' not in file:
45             filelist.remove(file)
46     _l = len(filelist)
47 print(filelist)
48 total_playerList=filelist[4:14]#取需要进行对战的文件
```

测试函数如右。考虑到单次将大量 player 两两比较耗时过长，故每次比较确定数量的 player，并选出性能最好的一到两个保留在 playerlist 中，每次显示选出的 player 名，以及对战胜率的列表。如此反复，直到选出性能最好的 player。

下面是对 round_match 文件中内容的修改部分，这可以使得在测试函数中我们可以直接得到各个 player 在对战中的胜率列表。

3.2 结果分析

3.2.1 本地运行结果（主要目的：比较参数的影响）

某次测试的结果如下：

```
['83.33333333333333+50.0', '0.0+0.0', '75.0+50.0', '75.0+66.66666666666667']
player2 0.py
['75.0+91.66666666666667', '75.0+91.66666666666667', '33.333333333333336+0.0', '33.333333333333336+0.0']
player2 0.py
['player0 0.py', 'player2 1.py', 'player4 1.py', 'player2 0.py']
['75.0+75.0', '33.333333333333336+0.0', '33.333333333333336+0.0', '91.66666666666667+91.66666666666667']
```

前两个列表是在 playerlist 中随机取的几个 player 两两对战后统计的胜率，具体的对应关系可以再生成的文件夹的 txt 文件中读取。显示的 py 文件名即为列表中胜率最高、性能最好的 player。文件名列表即为筛选后，最终比较的 player 列表。最后的胜率列表意义与前述类似。

算法在运行时间上基本在预期之中。在本地的测试主要是通过内部对战，找到性能较好的参数，然后再天梯赛上再和其他代码对打观察效果。与此同时，再根据最优参数的范围对代码算法进行优化。

3.2.2 天梯测试结果（主要目的：控制时间、发现重大漏洞）

首先，在天梯上调整了截断条件（α - β 剪枝中的[:6][:5]，在本地使用的[:4][:4]）。由于下一个质的飞跃的层数使先手 10 层后手 9 层，需要优化太多，即使是天梯的高性能也无法实现。

其次，发现了会给对方送子的重大问题，于是在决定 position 的时候加强了先前直接在自己方落子的条件：如果己方空间很小仍然己方落子很可能让大子无路可走，不能及时撤退。

3.2.3 算法效果：

此评分函数具有极强的进攻性，与预期符合的相当不错（基本不是被“塞爆”的时候都可以“塞爆”对面），并且大部分情况下合成效率还不错（很少会出现 3 个相同的大子）

3.3 经典战局

高光时刻：

图表 1：远古版本代码五五开霸榜代码 RBQ_4

势均力敌：

4 实习过程总结

4.1 分工与合作

小组分工（按照姓名拼音排序）：

董天宇：

初代代码的时间管理函数（尽管由于及其难以控制+后期采用了储存树弃用了）；
代码自动测试（魔改版的 round_match）内战工具的开发工作和维护；
启发式规则的建议。

董自豪：

代码自动测试（魔改版的 round_match）内战工具的开发和维护；
大部分的参数内战工作；
得到重要参数（废除某评分函数，并导致了只用 getScore 的评分函数的思路起源）。

符凯蓬：

测试关键大子的生成轮数用于时间管理（另一个功能的 round_match）（尽管随着时间管理的废弃被遗弃了）。

李鹤轩：

α - β 剪枝；
大部分启发式规则；
搜索树的优化；
节点关于时间复杂度的优化；
评分函数引入了空格的影响以及大子分值提升。

要昌雨：

Node 类；
α - β 剪枝 debug；
指数函数求和型评分函数的原型；
双方分值权重不一样的提出；
Player 的储存功能（尽管这导致了到最后放弃了时间管理）。

4.2 经验与教训

在实习过程中，组员之间的分工合作十分重要。正是因为各个组员能力不同、擅长之处不同，分工合作才更显得重要。如果让五个人的某个或某几个人完成所有工作，即便他们能力再强，能想到的东西可能还是不如合理分工后来得多，效率可能也会不够高。这一点，我们做得良好但也有需要改进的地方。另外，优化代码的过程中应该抓住每一阶段的主要问题——是参数的影响更大，还是算法的影响更大。这会决定组员们在该阶段的工作重心，避免劳动力在相对不重要的地方花费过多精力。而对主要问题的判断，也是需要花时间、花精力做的。这一方面，可能是我们对于代码这一类似于黑箱的体系缺乏了解（尤其是对于估值函数的优劣），做得欠佳，导致了一些劳动力的损耗。

4.3 建议与设想

在组队方面，还是尽量避免五个人能力都极强的情况，这样可以让资源最大利用，既不至于让同组的这五个人中的某几位无事可做（毕竟大作业的工作对这一部分优秀的同学而

言，可能挑战性还是小了些），又可以让基础稍弱的同学有机会向这些优秀的同学学习。在基础设施代码方面，是否可以考虑修改平台脚本，使得原本在本地进行的大量测试，能够在平台上进行。这样一来，代码的测试会更高效、更精确（毕竟最后的对战是在平台上进行的）。竞赛方面，也可以考虑循环赛，这样的赛制会更公平。当然，现有的淘汰赛机制也很不错，比赛现场十分刺激。

对于选修明年课程的学弟学妹，我想说的是，要对这门课程做好充足的心理准备。既要对课程本身的难点以及陈老师对课程独特的理解和教学方式做好准备，也要对参加课程的同辈做好准备。能够迎接这些挑战并且努力克服，你们一定能在这门课程中收获满满。或许，这也是陈老师的数据结构与算法课近些年来吸引一批又一批优秀学生的原因。

关于实习作业后续工作，一方面希望助教团队能够实现人机对战（难度应该不大）（甚至可以考虑推出一款产品），另一方面，我们可以有更多的时间优化算法，考虑更多可能的情况，从而优化估值函数；也可以学习更多强力的算法，对程序进行脱胎换骨的优化。

5 致谢

感谢 N19_Sierra 组、F19_X-Ray 组、N19_404 组的代码作为我们组代码的专业陪练；
感谢戴天翔（技术组）的标杆代码；感谢技术组提供的平台及服务；
感谢陈老师和助教们一学期以来的付出；
感谢所有组员的努力！

6 参考文献

2048-AI 程序算法分析：http://blog.codinglabs.org/articles/2048-ai-analysis.html
算法的唯一的参考文献，但是提供了总体的 α - β 剪枝的原理（核心）以及初代版本中评分函数的建立（尽管被废弃）

数据结构与算法课程实习作业报告

F19 Oscar 组

智伟峰* 田力 马德伟 张容嘉 吴泽宇

摘要： 我们的算法整体上是通过对于局面价值 value 的评估计算，来判断收益最高的走法。主要涉及到的数据结构有列表（存贮合并方向）和树，主要涉及到的算法就是 alpha-beta 剪枝算法，以减少不必要的搜索，节省时间。从实验数据结果来看，我们的算法的战斗力主要依赖于评估函数中的搜索程度，实验中经常出现由于搜索层度的不够导致算法犯了一些明显错误的现象，因此我们在讨论中的问题主要是针对评估函数的优化。最后看来，在不超时的前提下，我们的搜索层度最多能做到 4/6 层，虽然小组赛未出线，但是结果还是令人满意的。

关键字：搜索树、价值评估函数、alpha-beta 剪枝

1 算法思想.....1
1.1 总体思路.....1
1.2 算法流程图.....2
1.3 算法运行时间复杂度分析.....3
2 程序代码说明.....3
2.1 数据结构说明.....3
2.2 函数说明.....3
2.3 程序限制.....4
3 实验结果.....5
3.1 测试数据.....5
3.2 结果分析.....9
3.3 经典战局（可选）.....9
4 实习过程总结.....9
4.1 分工与合作.....9
4.2 经验与教训.....12
4.3 建议与设想.....12
5 致谢.....12
6 参考文献.....13

1 算法思想

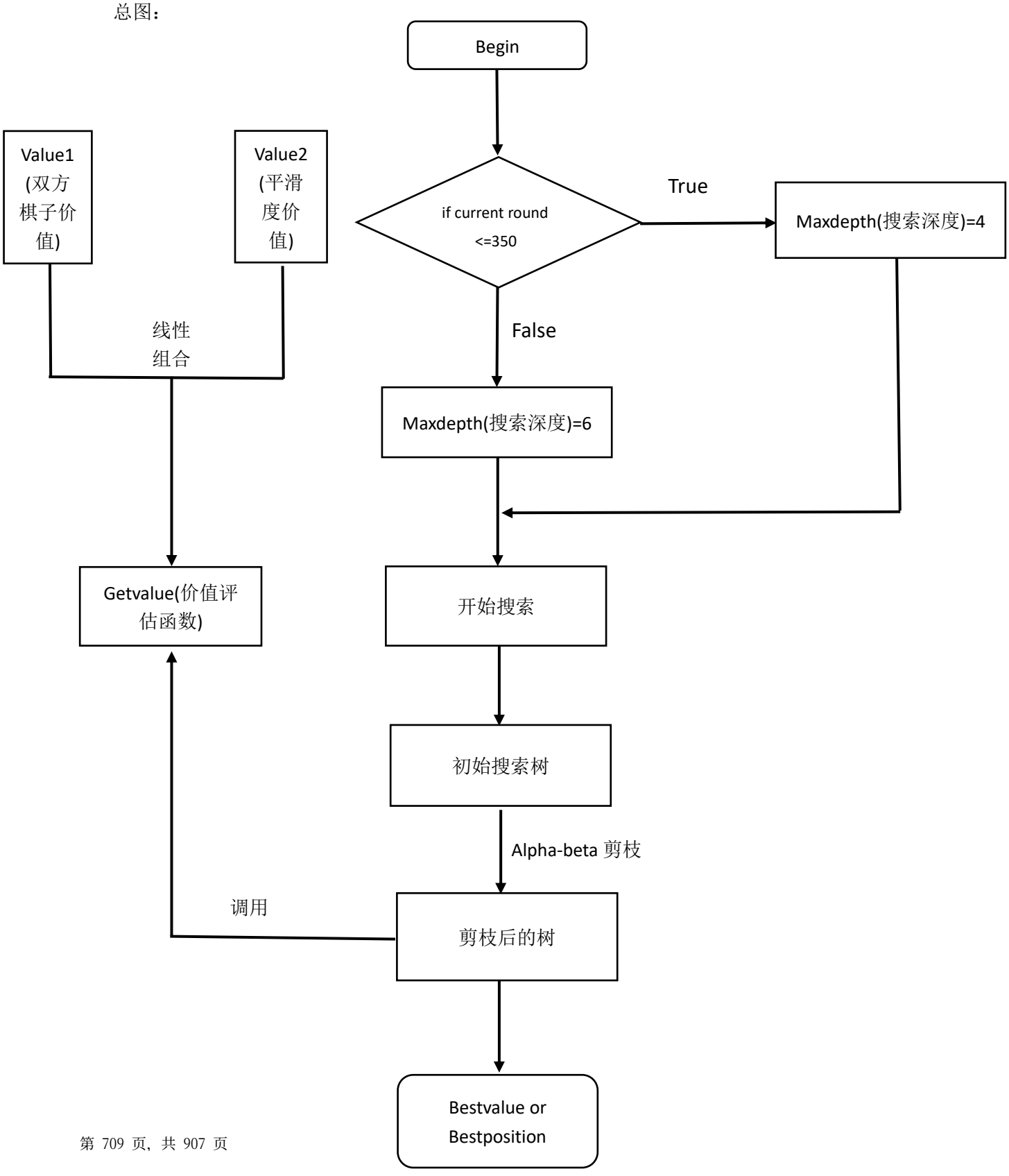
1.1 总体思路

算法的总体思路就是类似于一般棋类游戏的 AI 思路，不过较为简单。
第一步是局面价值评估函数的确定，计算出当前局面的价值；第二步是建立搜索树，搜索出自己能走的所有可能的走棋，然后对下面几步的所有的情况进行搜索，最后计算在下面几步能够获取的最大利益的落子地方和合并方向

采用的数据结构与算法是列表和树、alpha-beta 剪枝算法
算法的思路从最初开始就是很明确的，这种思路在棋类游戏中是很通行的，想把算法改进的更好，主要难点就在于增大搜索层数和评估函数的确定，所以大家的精力主要放在了对搜索方式的优化，和调整评估函数的参数上。

1.2 算法流程图

总图：



1.3 算法运行时间复杂度分析

本算法的时间主要消耗在价值评估和搜索树上。
1.价值评估部分,计算 value1 是对双方棋子进行遍历，所以的时间复杂度为 O(n)级别,计算 value2 是对每一行每一列进行计算,时间复杂度也为 O(n)级别。
2.搜索树部分,搜索层数为 k,则时间复杂度为指数量级的,记为 O(2^k)。
所以总的时间复杂度即为 O(n)*O(2^k)
可见,随着搜索深度的增加,时间复杂度迅速增加,为了在保证不超时的情况下增加搜索层数，我们对搜索树进行了剪枝。

2 程序代码说明

2.1 数据结构说明

算法中用到的主要数据结构是深度优先**搜索树**，通过对之后步骤的预测生成搜索树的分支。在这里处理在**对方区域**落子时，为了避免大量的无用步骤，直接调用 JudgePos 函数找到所谓最佳落子点，对我方是让对方收益最小的点，对对方是我方收益最小的点。这样关于在对方区域里下棋就简化为了一个分支。对每一个节点（局面），调用估值函数得到估值。接下来按照设定好的搜索深度，搜索得到最优策略并返回。

2.2 函数说明

1._init_方法：
对 player 类进行初始化，包含是否先手 isFirst 和随机落子的序列 array
2.output 方法：
在其中按照已经进行的局数和剩余的时间给出搜索层数，在防止超时的基础上尽量多搜索几层树。具体来说就是 300 轮以内搜 4 层，之后按照剩余时间搜 6~（500-currentRound）层。其返回结果在后面给出（**注：后面的函数都是 output 下的子函数**）
3.在对面落子的估值函数 JudgePos：
函数的返回值是对对方的干扰程度，具体流程如下：对每一个可以落子的点，向四个方向寻找对方第一个不为零的点（如果没有找到这个方向（水平/竖直）直接返回零） 。以左右为例，假设向左第一个子的值为 L，向右第一个子的值为 R，那么得到的估值为 2**L+2**R。对上下两个方向也是同理。找到干扰程度最大的一个点，返回它的位置。
4.估值函数 getValue
getValue 是整个 player 类的核心之一，它对整个局面进行评估，并返回一定的数值。在修改讨论的过程中，我们写了很多的估值函数。在经过天梯测试和出于运行时间上的考虑，我们

保留了 value_2，value_3 两个估值模块，还有 k_2，k_3 两个模块用于根据不同的情况生成两个 value 函数的系数。

（1） value_2：

这个函数返回的是棋盘上所有棋子的线性组合，但是由于胜负的评判标准是先看最大的子，我们把每个字的数值乘以其级别求和，（比如 16 的贡献是 4*16=64）保证大子的贡献更加显著。

（2） value_3：

此函数分析局面的“单调性”，返回“行增值”，“行减值”，“列增值”，“列减值”的最大值。以行增值为例：是所有行中左子减右子的和，如果左子小于右子，这行的值记为零，求和。其余三个值同理。value_3 返回这四个值中的最大值。

（3） k_2：

在 100 回合以内，大部分棋子的值较小，所以此项的系数相应也小一些，为 3。随局数增加，大的子的数量会增多，越到最后，数值大的子越重要。于是 100~300 回合这个值为 4,300~500 回合这个值为 5

（4） k_3：

我们在最初调试的时候发现后期在各种数值的子都多了的时候，合并过程不够“丝滑”，于是我们经过调整，把最初的“单调性”和“光滑性”合并到了 value_3 中，并调整系数为前 100 轮为 2,100~300 轮为 4.5,300~500 轮为 6。

最后返回值为 k_3*value_3 + k_2*value_2

5.alphabet 函数，建立搜索树与剪枝：

这个函数是另外一个核心。包括建立搜索树与搜索剪枝的过程。

首先是递归结束条件，如果搜索层数减到 0，就调用 getValue 对局面评估，并根据是我方还是对方返回+-getValue。

之后是两步操作，下棋与合并。**下棋时**判断是否为己方，copy 棋盘，并在这个拷贝棋盘上落子，递归调用得到 value。根据 value 的大小更新 alpha 和 beta 值。这里在对方下棋的那种选择情况较多，我们选择用 JudgePos 函数得到最大干扰位置，直接选择该位置，避免大量无效的搜索。**合并时**对四个方向进行递归调用得到 value，更新 alpha，beta 值。在一些不能下棋或者不能合并的情况（简称为卡住），都有相应的处理。

接下来为了确定当前的阶段，把第一次搜索单独拿出来，开始逐层回溯并剪枝，选取最有利我方的操作返回。搜索的层数由由回合数与剩余时间决定（output 方法）

2.3 程序限制

整个调试过程出现的最大问题就是搜索层数，估值函数与时间的冲突。最初的估值函数比较多，可以更好刻画局面的形势。但是在实战过程中，出现了自己在边界上较大的棋子被对方吃掉的悲惨后果，这就是搜索层数不够的问题，于是我们增加了搜索层数。再次调试又出现了了超时的问题。经过权衡，我们觉得估值函数的影响不如搜索层数。于是就将估值函数缩减，注重搜索层数，同时保证不超时。所以程序的限制主要是时间控制和估值函数不够全面。

3 实验结果

3.1 测试数据

实验环境说明：

- 硬件配置：（CPU/内存）Intel(R) Core(TM) i7-9750H CPU @ 2.60GHz
- 操作系统：（名称/版本）Windows 10 家庭中文版 64 位操作系统，基于 x64 的处理器
- Python 版本：（版本号）python3.8.3

测试算法采用“左右互搏”操作、人机对战操作、和一次短暂的天梯热身，即让新版代码对战旧版代码，或者让玩家对战程序。

对于代码间的互搏：我们记录了最终版“new”与之前部分版本的对战。

（1） new 与 player2：**共进行两次，均为 6:0**

对战结果大概是这样的

这里就看出之前提到的老版本的问题：player2 的搜索层数显然不够导致自己合成的大子在边界上被吃掉。边界上的控制也不好。而且本身稳定性不够，在干扰下“光滑性”降低了很多。不过这只是“傻瓜一号”的改进版，刚刚建立搜索树，评估函数也不成熟，可以原谅。（收获测试人员评价：player2 是真的辣鸡）

（2） new 和 player5 **共进行两次，分别为 5:1 和 6:0，这里只列举 5:1**

分别选取 new 和 player5 获胜的场次看一下：

new 获胜：

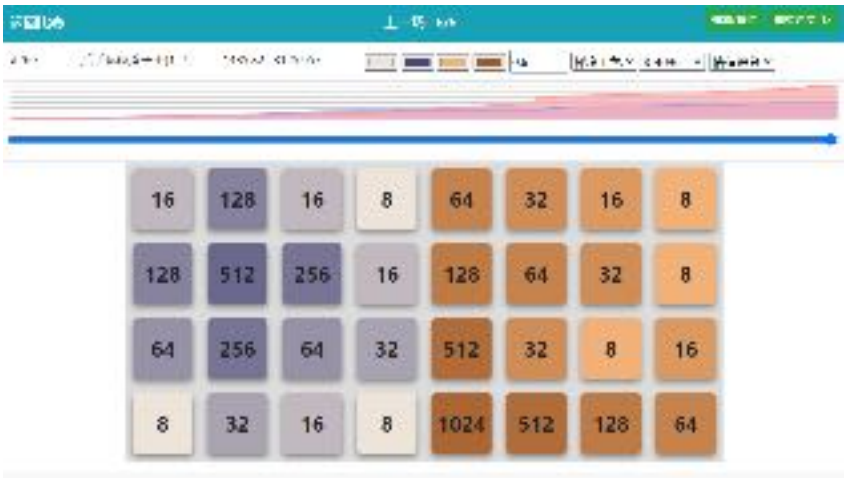
player5 获胜：

player5 是增加了 alpha-beta 剪枝和有了一些评估函数的版本，战绩比之前的 player2 好了不少，至少赢了一局（赢得不多）可以看到它已经努力的在增大平滑性方便合并了。可以看到它的棋盘上相同数值的棋子尽量按照斜线排列，也尽量让最大的子靠到角上。但是仍有搜索深度不足的问题，这将在接下来的版本中改进。

（3） new 和 player6 **共进行了两次，均为 4:2，列举其中一次的结果**

分别选取 new 和 player6 获胜的场次看一下：

new 获胜：



player6 获胜：

player6 又进步了不少就算输了差距也不是那么大，具体表现为在边界上的表现更好，因为 player6 的版本里添加了专门的边界上评估函数。而 new 因为搜索层数牺牲了一部分评估函数。通过对比可以看出，增加搜索层数的效果要优于一个好的评估函数。**正是当时对 player6 的修改让小组决定最大限度增大搜索层数，尽量压缩评估函数的复杂度，简化其评估过程。**（这里特别感谢技术组新加入的 getRaw）

（4）new 和 player7 测试了一局结果是 5:1

player7 是调整参数的一个版本，对评估函数的组合系数进行了调整，性能不是很稳定，具体表现为对战最后的 new 有时可以大比分获胜，有时却可以大比分输掉比赛。而且进攻性比较强，局面略有混乱。截取的两场比赛结果如下：

new 胜

Player7 胜

在采用人机对战中（人机对战没有记录），测试人员尝试了各种风格下棋方法，包括保守的自身发育、在边界不时吃子骚扰、在对方棋盘内搅局等各种情况。最后调整了估值函数为前面提到的总和与单调性。算法的整体偏向保守，但是在边界上出现机会的时候可以及时抓住。并且有较强的抗干扰能力。即可以在对方入侵时保持光滑性和合并的流畅性。

在天梯热身赛时，小组测试成员主要挑选了两个 ID 对代码进行调整，这两个 **ID** 是：**小恶魔、图书管理员润之同志**。主要对战方是“小恶魔”（**此次测试是在会议上由田力同学直播**）

在和**“小恶魔”**的对战中，小组选用了当时的 player6。最开始在不断调整评估函数以及系数。在发现效果不理想之后，代码组修改了策略，加深了搜索层数，稍微简化了评估函数，收获奇效，具体情况如下：

我方：对方

修改方案前

1:9	2:8	2:8	3:7	1:9

改为增加搜索深度后：

10:0	10:0

而在于“**图书管理员润之同志**”对战中，我方多次出现超时情况。一共三次，战绩如下：

1:9（有 8 次超时）	2:8（8 次全超时）	1:9（有七次超时）

当时并未留下截图，比较遗憾。在超时的多次中，有相当一部分局面是我方占优。于是我们明确了接下来的改进方式，用不同的搜索深度控制不超时，并尽量简化评估函数，取精去糟。

3.2 结果分析

测试中：主要是看不同的评估函数，评估函数的系数，还有不同的搜索深度，以及剪枝的方法。重要流程如下：开发了原始代码->增加了搜索树->增加了剪枝->增加了最初的估值函数：总和与最大值->增加搜索深度->增加有关单调性和边界的评估函数
起到重要作用的策略为剪枝（小组必备），单调性评估函数（应该是最利于合并的函数）

人机对战中：算法稳定性的提升。之前在写评估函数时没有仔细考虑对方侵略性很强的情况，导致在干扰下算法的表现明显逊于正常情况。于是小组改进了评估函数，重点考虑了对方棋子在单调性中的作用，以及如何尽量简便地围剿这种孤军深入的子。这种策略也起到了很重要的作用。

天梯热身中：主要的调整是调参以及提高搜索层数的地位，这是一个很重要的突破，事实证明搜索层数可以大大弥补评估函数的缺陷。每提高一层都是碾压性的。但是限于时间等因素，小组最多把层数提到了 6。而另一个重要的事情就是利用读取剩余的时间解决了超时的问题，这保证不会出现在大好局势下输掉比赛的情况。

这样算法的时间得到了最大程度上的应用，也最大程度保证了搜索深度。最后代码主要的运行时间开销现在在于搜索树上，因为估值函数已经大大简化。

3.3 经典战局（可选）

最经典的战局要数上面提到的“反杀小恶魔”。再次重申，这是一个很重要的转折，让小组将代码的主要时间花费在搜索树上。

4 实习过程总结

4.1 分工与合作

本小组由 5 位成员组成，其中智伟峰担任组长总体统筹协调，田力和吴泽宇主要负责程序的设计和代码的编写，智伟峰、马德伟和张容嘉主要负责实习作业报告的撰写。五位成员

均给出了估值函数的建议或者测试版代码。小组成员们主要通过在腾讯会议平台上召开组会进行合作交流，共召开了四次组会。

历次组会记录：

5.19 21:30 第一次组会

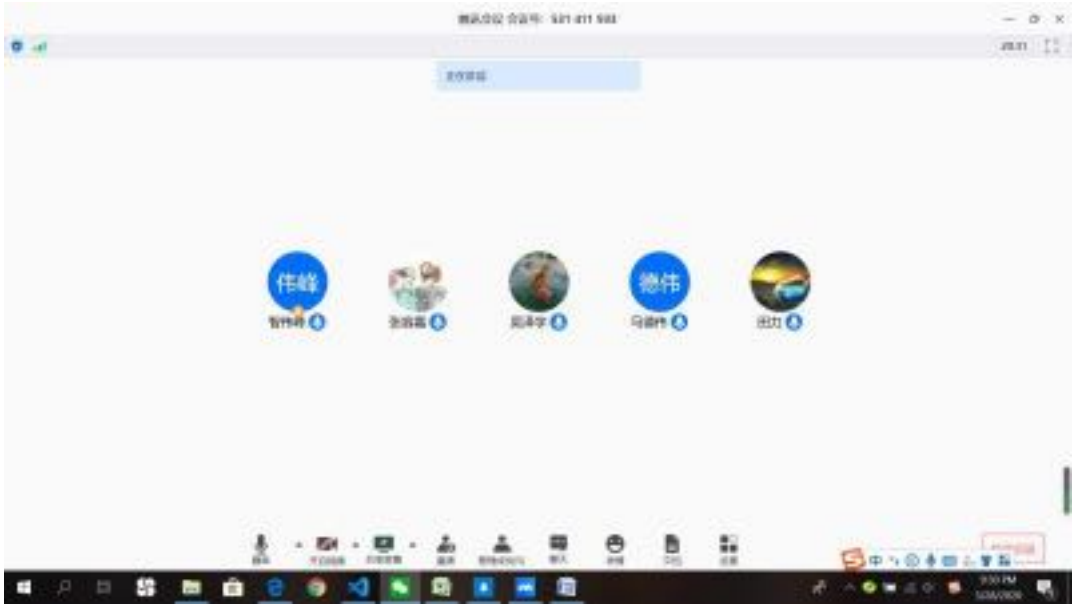
会议内容：大家在组长智伟峰的带领下对分工和时间安排进行讨论。首先将实习作业拆解成三个主要部分：策略的制定（程序中估值函数的编写）、代码的编写和测试、实习报告的撰写。最后商定得到如下安排：田力和吴泽宇先在几天内写出一个可以投入对战的简单版本的代码，同时 5 个人一起思考如何制定策略编写估值函数，等到敲定估值函数后再由田力和吴泽宇进行代码的修改和调试，在天梯开放后放上天梯进行对战，总结对战结果不断进行修改完善，直至确定最终代码后由智伟峰、马德伟和张容嘉着手撰写实习报告。会议进行了 30min，大家都明确了自己的任务。

5.24 21:30 第二次组会

会议内容：田力和吴泽宇展示了已经编写好的简单版本 AI“傻瓜一号”的代码，大家一起阅读了代码，进行了一些交流。田力表示“傻瓜一号”已经基本完成了第一阶段预定目标，可以投入对战，下一步就是尽快想出策略，补充估值函数。组长智伟峰明确了下一阶段的任务：大家继续思考对战策略，编写估值函数，择优采用。会议进行了 45min。

5.28 21:30 第三次组会

会议内容：田力和吴泽宇已经将新确定的估值函数写进程序中，并将 AI 放上天梯进行对战。他们展示了一些对战结果。对战结果不甚理想，大家一起观看了几局游戏的可视化复盘，分析代码存在的问题和劣势，经过对比和讨论，找到了搜索深度不够、缺乏防守战术、超时等问题，也提出了一些改进的方案。下一阶段的任务就是不断进行测试和对战，同时不断优化程序，提高 AI 的实力。会议进行了 60min。



6.2 21:30 第四次组会

会议内容：二零四八竞赛已经结束，接下来的主要任务就是撰写实习报告。本次会议主要由 3 位负责实习报告的成员参与，讨论了实习报告的结构和分工安排，最后确定由马德伟负责“算法思想”部分，由智伟峰负责“程序代码说明”部分，由张容嘉负责“实习过程总结”部分，三人一起进行实验结果的采集、统计和分析。会议进行了 20min。

4.2 经验与教训

在实习过程中，本组成员都能按时参加组会，积极配合整组的安排分工，认真完成各自的任务，形成了良好的合作交流氛围，同时我们制定了合理的工作计划，有条不紊地按时完成了实习工作。大家认真的工作态度、良好的合作态度和较高的工作效率都起到了正面作用。

但是由于经验不足，本次实习工作中也存在一些可以改进的地方，比如在开始设计程序之前没有做好足够的功课和较细致的理论准备，导致最后使用的算法和策略相对粗糙简单，对战效果不算理想；还有在思考策略阶段一开始考虑得不够完整，值得反思。

4.3 建议与设想

对本次实习作业的建议：

组队方面建议可以通过某种机制让各队的水平更为平均一些，比如预先用测试将所有人的相关能力水平分级，然后将不同级别的人员按照一定比例分配到一组，这样不光能使所有组的综合水平均衡一些从而形成更激烈的竞争氛围调动大家的积极性，还能促进本来不太熟悉的同学或者来自不同学科背景的同学相互交流相互学习，计算机方面较弱的同学也能在较强的同学的带动下收获更大的提高。竞赛方面建议改进一下赛程设计，让更多的人有参与感，提高竞赛的趣味性。

对选修明年课程的学弟学妹的寄语：

希望选修明年课程的学弟学妹们能在实习作业中做好充足的准备，学以致用，勤于思考，合作互助，能把这份大作业当作一个需要认真对待的项目，积极参与竞赛，在付出中收获知识、能力、快乐和成就感。

实习作业后续工作的大胆设想：

本次实习作业其实可以发展成一款人机对战的小游戏。跟据游戏体验和面向用户的实用性改进游戏规则，设计一个中等水平的 AI，并将游戏界面实现可视化即可。

5 致谢

感谢陈斌老师和各位助教老师的教导！
感谢技术组同学和助教对竞赛平台的开发！
感谢我们组全体组员的共同努力使得本次大作业能够顺利完成！

6 参考文献

[1]https://blog.csdn.net/pilipala6868/article/details/79598641
[2]https://blog.csdn.net/pilipala6868/article/details/79598641
[3]https://blog.csdn.net/qq_32767041/article/details/80634244?utm_medium=distribute.pc_relevant.none-task-blog-BlogCommendFromMachineLearnPai2-2.nonecase&depth_1-utm_source=distribute.pc_relevant.none-task-blog-BlogCommendFromMachineLearnPai2-2.nonecase
https://www.cnblogs.com/baiyuer/p/9606773.html

数据结构与算法课程实习作业报告

陈泓睿*，石英彤，王昱钦，李一萌，管宸炅

摘要：本算法主要通过估值和搜索两部分函数，运用迭代及 α-β 剪枝技巧，决定当前情况下最优策略。本算法并不复杂，主要运用线性数据结构如列表和非线性数据结构如搜索树等。通过不断测试和修改，我们对搜索部分引入了一定的启发式规则。搜索部分的算法较为理想，但遗憾的是，由于时间限制，未能给出更强力的估值函数，只能使用测试版。本报告简要介绍算法函数，通过测试进行分析和改进，提出了反思和建议。

关键字：迭代、决策树、深度优先搜索、α-β 剪枝、启发式规则、线性表

1 算法思想.....2
1.1 总体思路.....2
1.2 算法流程图.....2
1.3 算法运行时间复杂度分析.....3
2 程序代码说明.....3
2.1 数据结构说明.....3
2.2 函数说明.....3
2.3 程序限制.....4
3 实验结果.....4
3.1 测试数据.....4
3.1.1 实验环境说明：.....4
3.1.2 AI 版本说明：.....5
3.1.3 测试方法与测试数据.....6
3.2 结果分析.....6
3.3 经典战局.....7
3.3.1 第一个案例.....7
3.3.2 第二个案例.....7
4 实习过程总结.....8
4.1 分工与合作.....8
4.2 经验与教训.....9
4.2.1 收获与经验.....9
4.2.2 教训与反思.....10
4.3 建议与设想.....10
5 致谢.....10
6 参考文献.....10

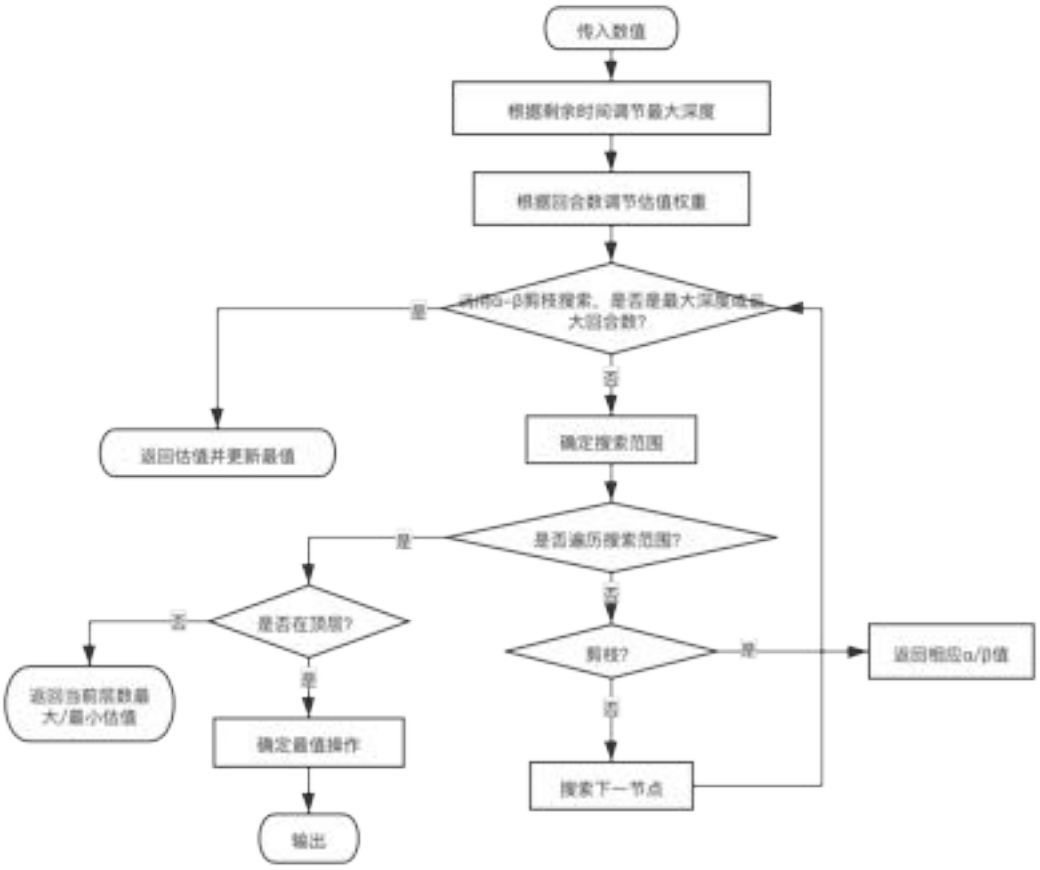
1 算法思想

1.1 总体思路

本算法参考了传统 2048 游戏的算法，分为两大部分，即搜索和估值。其中估值部分是基于当前棋盘情况，将棋子的分布通过单调性、连续性、最大值分数等数值进行估计。（详情在 1.2 具体说明）搜索部分是基于迭代算法，预判己方与对方落子对棋盘的影响，并通过 α-β 剪枝减少迭代次数。考虑到竞争因素，加入对棋盘边界的估计。

程序运行时，首先根据剩余时间和回合数分别调节最大深度和估值函数各项数值权重，再开始调用 α-β 剪枝搜索。建立搜索树。若已到达最大深度或最大回合数，直接返回数值。否则，将确定搜索范围并通过迭代逐层遍历，同时更新最大最小值节点。完成搜索即返回顶层时，选择估值最大操作作为输出。具体流程见 1.2。

1.2 算法流程图



1.3 算法运行时间复杂度分析

我们设搜索深度(self.depth)为 N（对应决策树的层数为 2N），棋盘的大小为 m*n。

本算法的运行时间主要消耗在 alpha-beta 剪枝搜索部分。根据我们定义的每一步搜索范围，决策树每个节点的子节点个数一般不超过 4。搜索到下一步该下棋的节点时，需要调用确定搜索范围的函数 positionList，执行该函数的运算次数约为 2mn。搜索到叶节点时，需要调用评估函数，执行评估函数的运算次数约为 4mn。由于 m,n 都很小，可以将它们视为常数，从而我们在计算时间复杂度时可以忽略确定搜索范围的函数和评估函数的调用次数，只考虑遍历的节点个数。

在最坏情况下，即不考虑剪枝（即必须遍历完 2N 层中所有节点），且每个节点的子节点个数均为 4。此时时间复杂度为 O(4^2N)

而在更多的情况下，对下一步该下棋的节点我们只考虑了在己方空位内下棋的走法。从而相当于一半的层数中节点的子节点个数为 1。此时时间复杂度不超过 O(4^N)

在一些极端情况下，我们的时间复杂度为 O(1)，包括以下几种情形：

1) 此时该我方下棋，而 positionList 函数返回的搜索范围中只有一个节点。
2) 此时该我方合并，只有一个合法合并方向
3) 剩余时间极短，我们直接随机输出

2 程序代码说明

2.1 数据结构说明

- 树：决策树的极大极小搜索和 alpha-beta 剪枝采用了树的数据结构。将某一局面下可选择的操作导致的新局面作为子节点连接，并考虑对方决策理智的情况下，选择最终估值最好的情况为决策。考虑到树的结点呈指数增长，不能搜索到终局，故限制搜索深度。返回 minimax 值时使用了递归，考虑到树的分支可以看作子树，可以对这些子树嵌套调用 minimax。
- 无序表：在确定搜索范围的 positionList 函数和评估函数的单调性部分使用了双指针遍历无序表的算法思想。

2.2 函数说明

- def setDepth(self, time, currentRound, mode) -> None:
  本函数根据剩余时间控制搜索深度(改变 player 类的属性 depth)。注意在本程序中每个单位深度代表决策树的两层，例如设定搜索深度为 2 表示完整地搜索一个回合。另外，我们规定设定搜索深度为 0 表示不按正常方式进行决策，而是直接用默认 AI 的输出方式进行随机输出。值得注意的是，由于在大多数情况下合并对局面带来的影响比下棋大，因此在时间不够充足的情况下我们尽可能让搜索覆盖更多的合并操作。
- def positionList(self,belong,currentRound,board) -> list:
  每次对下棋位置进行搜索时，通过该函数确定搜索范围。本函数具体的含义依赖于 AI

的版本。源代码中给出的是最终版本算法。具体我们在 3.1 中进行了详细说明。

- def defalutoutput(self,currentRound,board,mode) -> [Tuple[int, int], int]:
  当剩余时间特别少时，直接调用随机决策，防止超时
- def minimax(belong,mode,board,last,depth,currentRound,max_depth) -> float:
  设定 player 类的属性 self.ans 作为最佳输出，返回叶节点的估值函数值。本函数是搜索部分的主要框架。其主要部分是通过树的深度优先搜索的方法实现最大-最小搜索。同时还包含了对每一步确定搜索范围的调用、alpha-beta 剪枝的操作、对叶节点的估值函数调用。
- def setWeight(self,currentRound) -> None:
  在比赛的不同时期，各方面的重要性不同，故各方面的权值会随时间改变。因此，在决策之前，先根据当前的回合数设定评估函数中各个部分的权重。
- def evaluate(self, board,currentRound): -> float:
  评估函数，定义了如下几个子函数：
  1) def smooth_func(self, board,currentRound): -> float:
     评估平滑性的函数：评价相邻棋子的大小差异，差异越大，越难合并，分值越低。
  2) def empty_func(self, board,currentRound): -> float:
     评估空格数的函数：空位越多，可操作空间就越多，也更不容易因己方棋盘充满而无法操作，空格数越多，分值越高。
  3) def mono_func(self, board,currentRound): -> float:
     评估单调性的函数：在某一方向上，如果棋子呈单调递增或递减的趋势，则更容易进行多次合并，且容易得到更大的数，若某一方向上棋子数值单调排列，分数更高。
  4) def score_func(self, board,currentRound): -> float:
     评估双方棋子大小的函数：取定一个下凸函数 f，计算双方各自棋子等级的 f 值之差。在这里，我们取 f(x) = 3^x

2.3 程序限制

在实际的运行过程中，我们并没有发现出错的情况，除了由于程序本身的问题，在某次估值时，若程序发现无论如何都会失败，便会自动报错。我们认为实践操作中，这种报错不会影响对局的输赢情况，因此并没有进行修改。

3 实验结果

3.1 测试数据

3.1.1 实验环境说明：

- 硬件配置：2.3 GHz Intel Core i5/8 GB 2133 MHz LPDDR3
- 操作系统：macOS Mojave 10.14.6
- Python 版本：python 3.7.0

3.1.2 AI 版本说明：

- **搜索部分：**

本部分考虑的是合理选取 alpha-beta 剪枝中每一步的搜索范围。在最初版本的 AI 中，我们将所有合法的下棋或合并方法都纳入搜索范围内。但结果是时间复杂度非常高，搜索深度为 2（4 次操作）时都超时严重。经过测试发现：在轮到某方下棋时，在对方空位内下棋的方法数可能多达十几种；而大多数情况下最优的走法都不是走在对方的空位内。这造成了极大的浪费。为了解决上述问题，我们进行了以下几种尝试：

- **版本 01234：**
  我们尝试将下棋方法的范围改成：只要己方有空位，就不将棋下在对方空位内。这样大大提高了搜索效率，从而提高了搜索深度。但这个版本的 AI 完全不考虑在对方空位下棋的情况，在特定局面中有明显的局限性（见 3.3 的案例一）。
- **版本-10123：**
  为了解决版本 01234 的 AI 完全不考虑在对方空位下棋的问题，我们想到每次对下棋方法搜索时，对每种下棋方法的好坏用较简单的方法进行初步评估，然后筛选出初步评估分数较高的若干个节点进行更深的搜索。这种初步评估的方式可以是直接调用估值函数，也可以是调用深度较浅的 alpha-beta 搜索。这种做法的问题是：我们对每一个节点的初步评估本身就有一定的时间复杂度，从而对降低搜索耗时的效果不明显；同时，将“初步评估”的方法简单化意味着这样的初步评估方法是非常“短视”的，我们很难保证像这样筛出一部分节点的做法的合理性。
- **版本 01234.1**
  在版本 01234 基础上的另一种优化是：根据人脑在已有棋局中获得的经验，对下棋方法的搜索引入启发式规则。通过对大佬们的战局的反复分析，我们发现需要在对方空位里下棋的情况主要有以下两种：
  1) 涉及棋子的合并，包括：
     - 阻碍对方棋子吃掉己方棋子
     - 阻碍对方内部两个较高等级棋子的合并
     - 防止阻碍己方吃掉对方棋子
     - 防止阻碍己方内部两个较高等级棋子合并
  2) 当某一方的空位数较少时，包括：
     - 对方的空位较少时，加速对方“塞爆“
     - 己方的空位较少时，减缓己方“塞爆“

  于是，我们在每一次对下棋方法的搜索中将己方空位（如果有的话）和满足以上两个条件之一的点纳入搜索范围中，如果这个范围仍然过大，则随机取出其中若干个作为搜索范围。这样的做法将搜索范围以外的节点全部舍弃掉，有效实现了搜索耗时的降低，同时尽量保留了好的走法。

- **估值部分：**

我们的**“完整版”估值函数**参考了原版 2048 游戏 AI 的估值方法，详情见参考文献[2]。该估值函数由平滑性、单调性、空位数、棋子大小这 4 个部分构成。为了测定不同权重分配带来的效果差异，我们设定了多种不同版本的权值分配方案。

这里只列出其中具有代表性的 5 个版本：

版本号	平滑性权重	空格数权重	单调性权重	棋子大小权重
1	0.03	0.63	0.31	0.03
2	0.02	0.53	0.25	0.20
3	0.02	0.44	0.21	0.33
4	0.20	0.53	0.25	0.02
5	0.02	0.85	0.11	0.02

3.1.3 测试方法与测试数据

- **搜索部分**

我们对这 3 个版本的代码进行测试。在这一部分的测试中，为了控制变量，我们均采用如下的**“测试版”估值函数：**即只保留完整版估值函数中的棋子大小部分，舍弃其余几个部分的分数。

由于搜索部分的测试涉及到的代码版本数量较少，且这 3 个版本的实力悬殊较大，我们直接用在天梯上反复进行随机匹配的结果作为测试数据。结果如下：

- **估值部分**

我们用开发组提供的本地调试工具，分别使用各个不同权值分配的**“完整版”估值函数**的 AI 与**“测试版”估值函数**的 AI 进行 100 次对战（搜索方法均使用版本 01234.1），统计相应的对战结果。

这里给出 3.1.1 中所列的 5 个版本的对战结果：

版本号	对战结果（胜：负：平）
1	44 : 56 : 0
2	19 : 81 : 0
3	22 : 78 : 0
4	33 : 67 : 0
5	29 : 71 : 0

3.2 结果分析

- **搜索部分**

根据测试数据，版本-10123 不如人意（甚至不如最初的版本 01234）；而版本 01234.1 的效果较好，明显优于另外两个版本。因此我们在比赛中使用了版本 01234.1

- 估值部分

根据对不同版本的**“完整版”估值函数**的对战结果，最后悲剧地发现，没有一个版本能够优于**“测试版”估值函数**！然而我们也没有充足的时间进行其他尝试，最终被迫在比赛中使用了非常简陋的**“测试版”估值函数**。这实在让人遗憾。

3.3 经典战局

3.3.1 第一个案例

此处搜索版本为 01234，估值版本为**“测试版”估值函数**）

我方为后手，现在轮到后手下棋。显然，此时我方正确的走法是下在两个 128 中间。然而这个版本的 AI 还不会考虑将棋下在对方空位的情况，于是我方的 128（相当于 512）被吃了。事实上，这个局面给我们后面的改进带来了很大的启发，在版本 01234.1 中，我们就顺利地解决了这个问题。

3.3.2 第二个案例

此处搜索版本为 01234.1，估值版本为**“完整版”估值函数中的版本 1**

我方是先手，此时对方已堵死。事实上，在前面的游戏中我方表现还不错，当前的棋子排布情况也非常好。可悲的是，此时我方 AI 选择了向下合并，导致自己也被堵死，从而因为棋子大小比不过对方而输掉比赛。我们认为，我方能将棋子排布成这个样子，也就说明了估值函数是有一定道理的。但排布好后不能正确地进行合并，则说明该估值函数在特定情况下有着致命的缺陷。但是，看到这个局面的我们已经没有时间再进行改进了……

4 实习过程总结

4.1 分工与合作

首次开会通过参考传统 2048AI，决定算法的框架，分为搜索和估值两部分，分别由两位同学负责，且需要一人编写对棋盘的各种指令（当时我们觉得开发组提供的 board 类可能会速度较慢，应该自己再写一个以方便搜索），并约定共同考虑更复杂的算法如利用规则进攻等。程序的调试由各组元共同完成。最初分组如下：

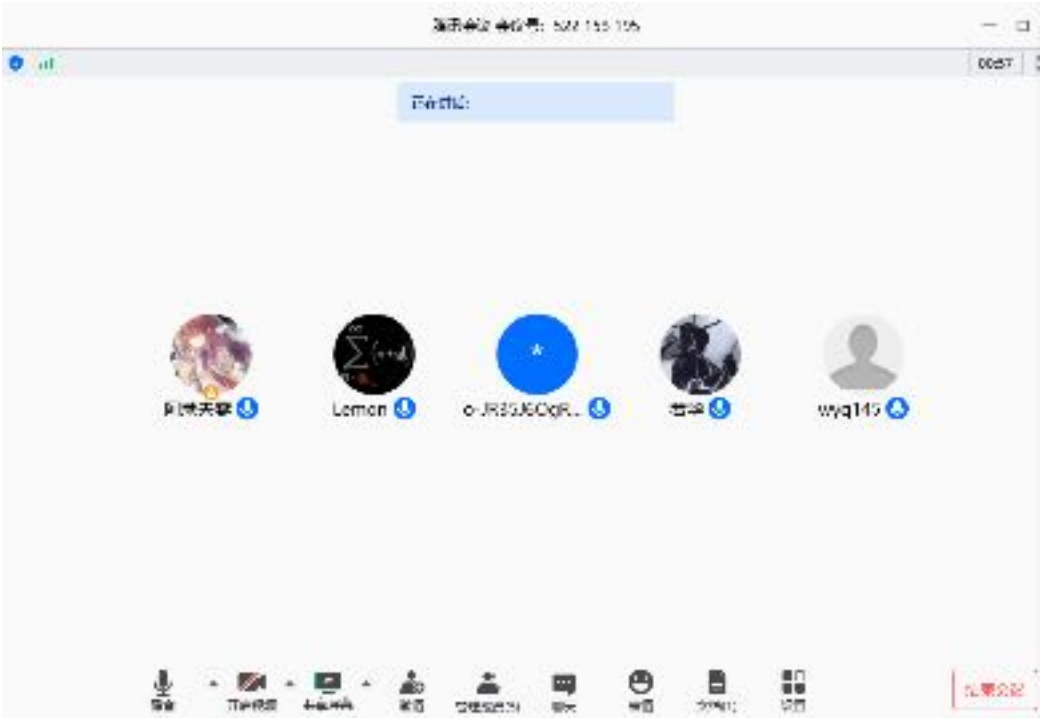
- 陈泓睿同学和李一萌同学负责算法的搜索部分和搜索部分的测试；
- 石英彤同学和王同学负责算法的估值部分和估值部分的测试；
- 管宸炅同学负责指令。
- 报告的主体部分由写相应部分算法或进行相应部分测试的同学撰写。

但后来 Chessboard 类利用了 cpp 加速，速度很难被超过，因此不需要我们自己来写指令。管宸炅同学转入撰写报告的主要部分。

交流方面，考虑到疫情影响，组员们不能在线下会面，直面交流缺失。但是，我们做到了及时线上通讯以及按时召开的组内会议。具体会议记录如下：

- 第一次会议：5 月 13 日 20：30　提出基本的算法思路，基本确定分工
- 第二次会议：5 月 17 日 20：30　明确搜索、估值算法的具体写法及参考材料
- 第三次会议：5 月 24 日 20：30　最初版 AI 诞生，讨论测试方法和细节改进事项
- 第四次会议：6 月 3 日 20：30　讨论报告的写法

附：腾讯会议截屏



4.2 经验与教训

4.2.1 收获与经验

本次实践小组分工合理，目标明确，从游戏策略的探讨、算法的实现和考虑及调试到报告撰写，能使各位队员发挥专长，也能互相合作提高效率。在天梯赛上，我们发现并解决了诸多漏洞，进一步提高了程序的性能。

各位组员也都积极参加讨论实践，无论是正式会议或平时讨论，组员们踊跃参与，创造了良好的实践氛围。编写代码中保持交流，利用天梯平台掌握本组程序与他组程序之间的差异，不断填补编写时不曾考虑的漏洞。另外，合作使得我们意识到代码注释的重要性。其一，简洁易读的代码和详细的注释加强了组员之间的合作与探讨，间接地创造了活跃的参与环境。其二，有效的注释降低了改动代码的难度，为调试改进提供了保障，提高了算法开发的效率，节省了不少用于思考策略的时间。

实际上，若按设想进行操作有充足时间进行调试与改进，但提交时间的提前两天，导致个人与团队的时间分配出现少许冲突，不得不说是一个小小的遗憾。相应地，我们意识到预留时间的重要性。由于本次实践临近期末，虽然少数课程没有期末考试的压力，但总体的安排使得实践在时间上略显局促。若是能在制定计划时考虑到可能出现的冲突和意外，相信我们可以更从容的完成任务。

最后，AI 的编写对于我们来说是一个全新的领域，对人工智能的高度神话使我们产生了畏惧心理。然而通过不断学习，我们从最初的摸索到后来的渐有眉目，从肇始的茫然到最后的详尽规划，每一步我们都经历了不小的考验与挫折，每一次实践都记录着我们进步的脚印。感谢本次实践，我们明白了学习现有理论知识的意义，体会到自身思考的局限。在解决问题之前，需要充分地分析问题并参考前人的策略，在学习的过程中不忘变通，对新难题、新困境下利用已知应对未知。

4.2.2 教训与反思

首先，我们未能及时关注信息，所出现的信息差使得他组在较为从容的情况下，我们仍未能完善最后的代码。这反映出本组在信息收集和获取上有所欠缺。同时，也能看出我们在规划过程中思想过于简单，未能给出应对突发事件的解决方案，以致在濒临提交的几天内略显匆忙。这种信息的缺失又体现在我们的首次讨论。面临问题时不习惯于全面采集信息，我们在制定分工时就出现了明显的错误。这种错误本可以通过仔细阅读已有文件避免，但遗憾的是，在场的五位组员均未能指出。

然后，对时间的准确把控的无力。在实践过程中，我们对时间的控制并不完全令人满意。这不仅体现在信息收集导致的提前中，更为重要的是，我们没能做到准时提醒组员完成任务。第一，在分配任务时，约定的提交底线较为模糊，这种不明确造成了实践的不紧凑，于是调试与改错阶段我们的精力略显单薄。一方面，考虑到期末的各科时间分配，希望对代码的编写尽量给出足够的预算，另一方面，组员在初次会晤时的陌生导致的权威缺少也可能是一个解释。第二，承袭之前的原因，在编写过程中，没能做到提醒组员按时完成，所以在进度方面，我们完成的不尽人意。如此的失误对我们都是不小的打击。

4.3 建议与设想

本次实践体验极佳，交流是提高的最好途径。从组队，发布任务，天梯赛预热到出线小组对抗直播，整个过程都非常有趣。我们通过团队协作共同完成任务，在充实和饱满中度过了一个难忘的期末。如果必须给出建议的话，我们希望大作业的布置能更提前，让同学们的时间更加充裕（并不一定要讲完全部课才布置大作业啊），同时，对 AI 的介绍也相应地前移几周，以便同学们对 AI 的认识能更加深刻。

对明年选修的同学们，我们也有不少提议：希望学弟学妹们吸取我们的教训，努力不留遗憾。但事实上，人类从历史中唯一学会的，便是人类没有从历史中学会任何东西。又或者，没有遗憾的人生才是真正的遗憾，无论过程如何，结果就在那里。从这方面看，不如像我们一样，把实践作为一次享受，成功和失败都有独特的滋味。

5 致谢

实名感谢本组各位成员！
感谢课程同学一学期的陪伴！
感谢开发组精彩的程序与设计！
感谢各位助教和陈老师一学期的教导和鼓励！

6 参考文献

[1]二〇四八开发文件网站：https://github.com/pkulab409/sessdsa.2048
[2]重要参考：
https://blog.csdn.net/qq_32767041/article/details/80634244?ops_request_misc=%257B%2522r

equest%255Fid%2522%253A%252215911902111972522241518 0%2522%252C%2522scm%2522%253A%252220140713.130102334.pc%255Fblog.%2522%257D&request_id=159119021119725222415180&biz_id=0&utm_medium=distribute.pc_search_result.none-task-blog-2~blog~first_rank_v2~rank_blog_default-1-80634244.pc_v2_rank_blog_default&utm_term=2048+python+AI

数据结构与算法课程实习作业报告

苏遨*　伍岳　马泓锐　杨棋　滕正

摘要：

F19_Quebec 组的二〇四八对抗算法主要分为两个部分：估值函数和决策函数。估值函数用以判断局面对己方的有利程度，估值越高，局面对己方越有利。决策函数用以推演棋局的发展情况，结合估值函数所给出的估值，选取最高估值的路径进行落子、合并。

关键词：

估值函数　决策树

目录

1　算法思想.....2

1.1　总体思路.....2

1.2　算法流程图.....3

1.3　算法运行时间复杂度分析.....4

2　程序代码说明.....4

2.1　数据结构说明.....4

2.2　函数说明.....5

2.3　程序限制.....8

3　实验结果.....9

3.1　测试数据.....9

3.2　结果分析.....9

3.3　经典战局（可选）.....11

4　实习过程总结.....12

4.1　分工与合作.....12

4.2　经验与教训.....15

4.3 建议与设想.....16
5 致谢.....17
6 参考文献.....17

1 算法思想

1.1 总体思路

经过第一次线上讨论，我们确定整个二〇四八对抗算法（以下简称 AI）应当包含两个部分：估值函数和决策函数。

估值函数用以判断局面对己方的有利程度。在前期的单人 2048 游戏及后期的对抗赛中，我们找到了一些规律。这些规律经具体化、定量化后便成为了估值函数的 6 个部分：单调性（monotonicity）、均一性（uniformity）、空位充足（abundance）、缺陷（defects）、高分奖励（bonus）和领域（territory & keep_away）。总估值等于这 6 个估值之和；总估值越高，AI 判定局面对己方越有利。

（1）单调性估值：如果在某方的棋盘上，所有棋子的等级能够在两个相互正交的方向上按照从高到低的方式排列，那么这一方便不会出现高、低级棋子相互交错的情况，从而能使 AI 在比赛过程中进行更有效的合并。

（2）均一性估值：如果在某方的棋盘上，等级相近的棋子位置也相近，那么这一方便不会出现两个高级棋子被其间的低级棋子所阻隔而无法合并的情况，从而能使 AI 在比赛过程中合成更高等级的棋子。

（3）空位估值：这是一个来自 2048 对抗模式的启发式估值函数。在对抗模式中，一旦某方被“塞爆”，这一方的棋局就基本失去了继续演化的能力，分数的增长随之停止。因此，被“塞爆”是需要 AI 在决策时刻意避免的。

（4）缺陷估值：这也是一个来自 2048 对抗模式的启发式估值函数。在对抗模式中，一旦某方的两个高等级棋子间出现了空位，那么对方即可向该空位中投掷低等级棋子，从而达到阻碍其合并的目的。因此，“在高等级棋子间产生空位”是需要 AI 在决策时刻意避免的。

（5）高分估值：按照对抗规则，合成更高等级棋子的一方胜出。因而有必要依照棋子的等级给与不同的奖励；等级越高，奖励越高，从而促使 AI 合成高等级棋子。

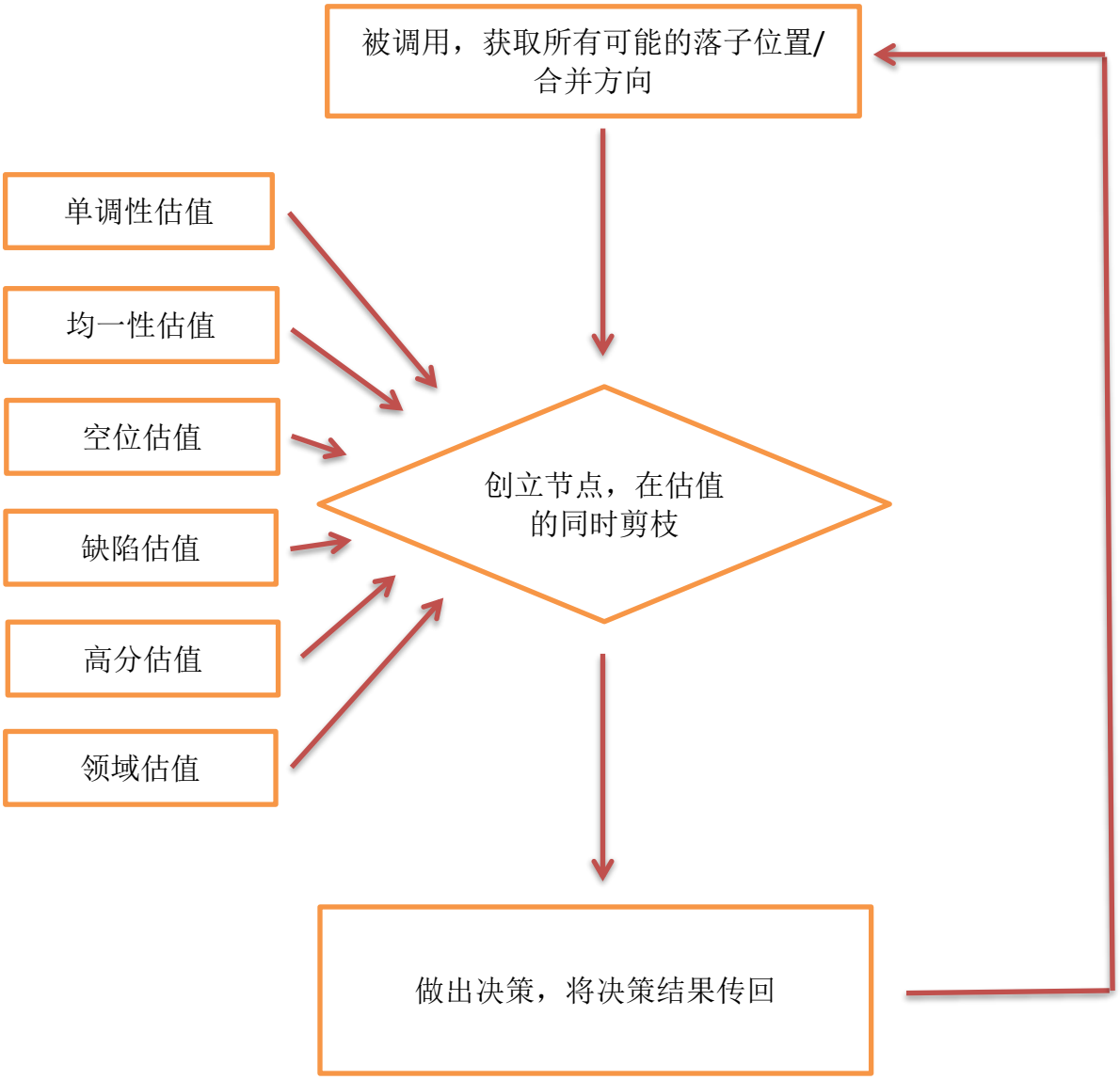
（6）领域估值：将己方的棋子打入对方棋盘内，不仅可以干扰对方的合并

操作，还可以拓展我方领域、压缩对方领域，使局面朝着有利于我方的方向发展。为此，有必要鼓励我方棋子打入对方。

决策函数用树结构来模拟对弈的进行。AI 首先取得当前情况下所有可能的落子位置或所有允许的合并方向；对于每种可能，建立一个节点以进行评估；同时依照 α - β 剪枝的基本原理，缩小候选范围，提高搜索效率；最终选择最有利的路径进行落子、合并。

在树结构中，每 4 层为一周期，分别对应先手落子、后手落子、先手合并、后手合并。每次创建节点时，该节点的层数等于（上一层节点层数+1）%4。

1.2 算法流程图



1.3 算法运行时间复杂度分析

我方 AI 在运行速度上表现不佳，主函数及各个分函数的时间复杂度如下：

函数名称	复杂度
moves	O(1)
evaSort	O(n)
explore	$O(e^n)$
value_list	O(n)
monotonicity	O(n)
uniformity	O(n)
abundance	O(1)
defects	O(n)
bonus	O(n)
territory	O(n)
keep_away	O(n)
attack_defense	$O(n^2)$
主函数	$O(n^2e^n)$

2 程序代码说明

2.1 数据结构说明

AI 中涉及的数据结构有节点、树、列表、二阶嵌套列表、元组等。

2.2 函数说明

现摘取函数的主要部分，以注释的形式进行说明：

（1）moves 函数实现本地落子与合并，以便估值函数进行估值

```
def moves(self):#1.返回允许的所有落子位置或所有合并方向

    if self.stage <= 1:#此时是落子阶段
        if self.stage==0:#先手落
            fiPosition=self.board.getNext(True,self.currentRound)
            #先手方此下在自己领域为的棋子位置，类型是turple
            sePosition=self.board.getNone(False)
            #可在后手方下的位置，类型是list，list元素是turple
            if fiPosition!=():
                self.explore(fiPosition)
            try:
                for i in sePosition:
                    self.explore(i)
            except:
                pass
        else:#后手落
            fiPosition=self.board.getNone(True)
            #后手方此下在先手方领域为的棋子位置，类型是list
            sePosition=self.board.getNext(False,self.currentRound)
            #此后手方在自己领域下的位置，类型是turple
            if sePosition!=():
                self.explore(sePosition)
            try:
                for i in fiPosition:
                    self.explore(i)  #0.5以来相关动
            except:
                pass

    else:
        if self.stage==2:#先手方
            listmove=[0,1,3,3]
            #决定方向为向左走，对于先手，顺序为上下左右（测试人员：这个顺序会影响结果吗，请测试）
            for i in range(4):
                self.explore(listmove[i])
        else:
            listmove=[0,1,3,2]#对于后手，上下右左
            for i in range(4):
                self.explore(listmove[i])

    if self.child==[]:
        self.explore('still')
```

（2）explore 函数记录局面的变化

```
def explore(self,move):#在move动作下棋盘改变后面，并生成相应节点
    a=True
    if self.stage%2==0:#先手方
        nextboard=self.board.copy()#复制棋局
        if move=='still':#如果走了或下或左右无法合并，保持棋局不变
            pass
        else:
            if self.stage == 0:#先手落子阶段
                nextboard.add(True,move)#先手落子
            else:#先手合并阶段
                a=nextboard.move(True,move)#先手合并
    else:#后手方
        nextboard=self.board.copy()
        if move=='still':
            pass
        else:
            if self.stage == 1:#落子阶段
                nextboard.add(False,move)
            else:#合并阶段
                a=nextboard.move(False,move)

    if a:
        nextnode=Node((self.stage+1)%4,(self.currentRound+(self.stage+1)//4)
                      ,nextboard,(-1*self.ismine),self.isFirst,self)
        #生成新节点并且成为对应节点，阶段为（父节点阶段+1）mod 4....
        self.child.append([nextnode,move,evaluate2(nextboard,self.isFirst)])
        #添加子节点与走动动作
```

（3）min_max_alpha_beta 函数实现搜索与剪枝

```
def min_max_alpha_beta(self,depth,stage):#搜索+剪枝


    if depth == 0 or (self.currentRound==500 and self.stage==4) :#搜索终止条件
        return evaluate(self.board,self.isFirst,stage)#返回局面评估函数

    if self.ismine == 1:#如果当前节点为我方max节点
        self.moves()
        sortmoves=self.evaSort(self.child)
        for nextnode_and_move in sortmoves:#在我方所有可移动步骤中遍历
            minnum=self.min#当前节点得分至少能达到的值，即之前探索过所有子节点中的得分最大值
            compare=nextnode_and_move[0].min_max_alpha_beta(depth-1,(stage+1)%4)#进行下
            if minnum < compare:#如果一个节点得分大于minnum，则当前节点实际得分增加
                self.min=compare#更新得分
                self.nextstep=nextnode_and_move[1]#更新行棋动作
            if self.min >= self.max:#剪枝
                break
        score=self.min
        return score
```

```
        else:
            self.moves()
            sortmoves=self.evaSort(self.child)
            for nextnode_and_move in sortmoves:#对对方所有可移动步骤中遍历
                maxnum=self.max
                compare=nextnode_and_move[0].min_max_alpha_beta(depth-1,(stage+1)%4)
                if maxnum > compare:
                    self.max=compare
                    self.nextstep=nextnode_and_move[1]
                if self.min >= self.max:
                    break
            score=self.max
            return score
```

（4）monotonicity 函数判断当前局面的单调性

```
def monotonicity (values):  #单调性估值函数
    left_right = 0
    up_down = 0  #估值函数的两个部分，分别对矩阵左右、上下两个方向的单调性
    i = 0
    j = 0
    k = 1
    while i != 4:
        while k != 4:
            if values[i][j] == 0:  #空位不参与计算
                j += 1
                k += 1
            elif values[i][j] != 0 and values[i][k] == 0:  #空位不参与计算
                k += 1
            else:
                left_right += (values[i][j] - values[i][k])  #前项减后项
                j = k
                k += 1
        j,k = 0,1
        i += 1
    i = 0
    j = 0
    k = 1
    while i != 4:
        while k != 4:
            if values[j][i] == 0:  #空位不参与计算
                j += 1
                k += 1
            elif values[j][i] != 0 and values[k][i] == 0:  #空位不参与计算
                k += 1
            else:
                up_down += (values[j][i] - values[k][i])  #前项减后项
                j = k
                k += 1
        j,k = 0,1
        i += 1
    total = abs(left_right) + abs(up_down)  #总的量化值是上下、左右两个方向的量化值之和
    return 0.25*total
```

（5）uniformity 函数判断当前局面的均一性

```
def uniformity (values):  #均一化估值函数
    left_right = 0
    up_down = 0  #估值函数分为2个部分，分别与矢量盘左右、上下2个方向的均一性
    i = 0
    j = 0
    k = 1
    while i != 4:
        while k != 4:
            if values[i][j] == 0:  #空位不参与计算
                j += 1
                k += 1
            elif values[i][j] != 0 and values[i][k] == 0:  #空位不参与计算
                k += 1
            else:
                left_right += abs(values[i][j] - values[i][k])  #大数减小数
                j = k
        j,k = 0,1
        i += 1
    i = 0
    j = 0
    k = 1
    while i != 4:
        while k != 4:
            if values[j][i] == 0:  #空位不参与计算
                j += 1
                k += 1
            elif values[j][i] != 0 and values[k][i] == 0:  #空位不参与计算
                k += 1
            else:
                up_down += abs(values[j][i] - values[k][i])  #大数减小数
                j = k
                k += 1
        j,k = 0,1
        i += 1
    total = - (left_right + up_down)  #总的均化值是上下、左右方向上均化值之和
    return 0.25*total
```

（6）abundance 函数对棋盘上的空位数量进行估值

```
def abundance (board,side,isFirst):  #空位估值函数
    Len = len(board.getNone(side))  #获取指定一方的空位数目
    scores = [1024,512,128,32,8,0,0,0,0,-2,-4,-6,-8,-12,-20,-32,-64]
    if side==isFirst:
        return scores[Len]  #此函数自版本号v0.91
    else:
        return int(0.25*scores[Len])
```

2.3 程序限制

我方 AI 所受到的最大制约，来自于高时间复杂度所导致的搜索深度不足（只能搜索 3 层左右）。这进一步导致了估值函数不能完全发挥其作用，且在对战过程中有一定概率（约 10%）超时。究其根本原因，是落子阶段时搜索分支过多，

既没有很好地利用 currentRound 来调整 AI 在开局、中局、残局的搜索策略，也没有充分利用估值函数来在节点生成之前剪枝所导致的。

另一方面，由于我方 AI 是基于 2048 的单人模式设计而成的，因此尽管在镜像模式（自己和自己对战）下表现接近预期，但在对抗方面有明显不足，结果往往是 AI 在 defects 函数的指引下很快将对方塞爆，随后对方的棋子逐步侵入我方（要归咎于 monotonicity 和 uniformity 函数给对方入侵棋子“铺路”），最终导致我方最高等级棋子被吃而满盘皆属。虽然后期加入了 attack_defense 函数以改善这一情况，但 AI 的表现仍不理想。

3 实验结果

3.1 测试环境

- 硬件配置与操作系统

- Python 版本：（版本号） Python 3.6

3.2 结果与分析

本地测试日志：

对手	先手胜利（共20场）	后手胜利（共20场）	分析及改进
defaultplayer	10	6	发现程序不能很好的保持整体的单调性，单调函数只作用于局部。大数喜欢相边界移动，易被吞并。同时搜索慢，容易超时。
defaultplayer	13	9	与上版相比，增加了keep_away函数，将我方位置加权，使大数待在后方。增加了启发式算法，搜索速度可以更深一层。但发现在中局倾向于在对方位置落子，而己方做无用移动。
defaultplayer	14	14	优化了空位估值函数，对我方与对方的空位分别估值。此时程序对空位的评估表现良好。但应对攻击能力弱，己方位置经常被吞并，会有无子可下的情况。
defaultplayer	19	19	与上版相比，将keep_away函数对我方位置加权改为对棋盘所有位置加权，对方位置权重更高，将大棋子放在边界增加了攻击性，我方位置不易被侵占。发现了合并成大数的倾向不高的问题。
defaultplayer	19	20	更新之后大体问题已经解决，具体到逐次行动，通过计算，略微修改了各个评估函数的参数。
defaultplayer	20	20	本地调试完成，进入天梯赛阶段。

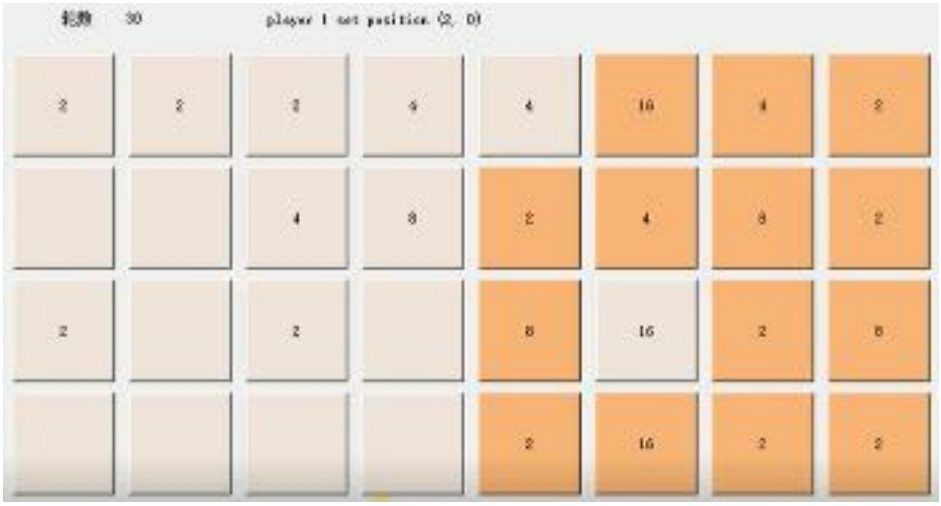
天梯热身赛测试日志：

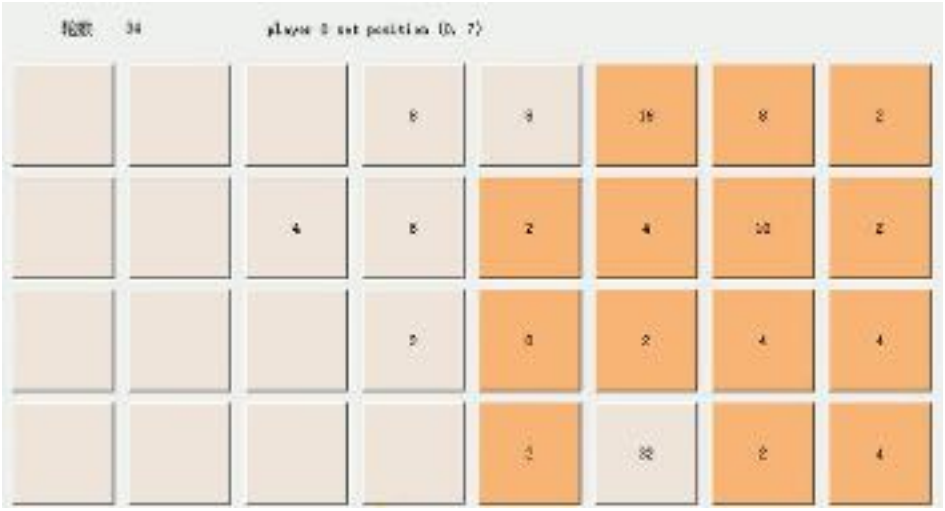
对手	先手胜利	后手胜利	分析及改进
2个随机对手	0/10	0/10	暴露出了最大棋子容易被对方合并的问题。
2个随机对手	4//10	2//10	与上版相比，增加了attack_defense函数，考察己方棋子的相邻位置与处在同一水平/竖直线上的对方棋子位置，防止被攻击，更容易创造机会攻击对方。但表现出大局观不足，尽管攻击力进步，但会出现突然将占领位置撤出的情况。
3个随机对手	6//15	6//15	与上版相比，加入了territory函数，用来刻画侵占对方的领土与被侵占领土的情况。但与攻防相关的函数过多，有关自身秩序与规律的评估函数作用被弱化。
同上的3个对手	9//15	10//15	全面协调优化了各个函数的参数。 大体行动表现合格，但受搜索能力限制，没有达到理想表现。

注：日志中只列举了重点的更新与测试结果。代码中出现的许多 bug 和对函数本身、参数设置的许多调整，内容相似的都归结到一条内。

3.3 经典战局

在设计之初，组员皆未认识到跨界吞并对方棋子的重要性，以为只要在远离对方的一边“闷声发大财”，即可凭借估值函数的优势战胜对手。但由于 bug，我方棋子非但没有聚集在远离对手的一边，反而扎堆在靠近对手的一边，以至于我方 AI 成了一个有力的侵略性 AI，往往能将高等级棋子打入对方内部，出奇制胜。





当时组员们都认为，这样的战绩的确不错，但将己方高等级棋子打入敌方内部的举动实在是过于冒险，于是经过 debug 和参数调整，将 AI 修复为“闷声发大财”模式，结果反倒不如以前。复盘结果显示，我方经常成为被侵略的一方，虽然能有效地积累高等级棋子，但最后都成了对方的盘中餐。

综合考虑，我方添加了 attack_defense 函数，即专门用来“吃掉对方”的函数，无论是在我方棋盘中还是在对方棋盘中。在我放棋盘中吃掉对方即“defense”，在对方棋盘中吃掉对方即“attack：。这样一来，攻防能力相对均衡，天梯胜率显著提升。

4 实习过程总结

4.1 分工与合作

分工：

苏遨*： 1. 负责会议的召开，规划时间、提出策略；

2. 负责估值函数的设计与编写；

3. 进行天梯实验，调整相关参数；

4. 修改、完善作业报告。

伍岳： 1. 提出程序的整体思路；

2. 负责决策函数的设计与编写；

3.进行天梯实验，消除 bug。

滕正： 1.和伍岳一同完成决策函数；

2.商讨估值策略。

马泓锐：1.主要负责天梯实验，及时反馈相关的 bug；

2.添加 AI 的对抗部分，使之更加适应天梯的对抗赛；

3.讨论策略。

杨棋： 1.负责记录每次会议的内容。

2. 负责实验报告的主体书写。

组员交流方式：腾讯会议、微信群讨论

会议次数	会议时间	会议内容
第一次会议	2020/5/18	本次讨论主要以大家对 2048 的游戏经验为主，最后确定目前的任务主要就是大家熟悉一下平台的各个接口，以及对战的各种具体规则，具体分工待定
第二次会议	2020/5/20	小组完成分工并定下近几天工作： 苏遨、吴岳、滕正：合作完成评估函数、决策函数的编写 杨棋：记录每次会议内容、完成报告的书写 马泓锐：负责后期程序的 debug
第三次会议	2020/5/23	大家各自汇报近几天的进度以及遇到的问题，一起讨论解决。然后确定下我们的总安排，最后分配接下来几天各自的主要任务。

照片：

Figure1:第一次腾讯会议现场（当时马泓锐同学尚未加入我组）

Figure2:第二次腾讯会议现场（这次有了马泓锐同学的加入）

Figure3:第三次腾讯会议现场

Figure4：微信群里的部分讨论记录

4.2 经验与教训

我们组实行分工合作，每个人都有自己主要的工作，这样首先我们不会在代码的编写上出现较大的偏差，主要就是苏遨和伍岳各自负责两部分函数的编写，这让我们在代码的编程过程中具有较大的便捷性，每位编程人员也对自己的那部分代码比较熟悉，再者，在调试时，哪个部分出现了 bug，那么我们可以直接反馈那部分的负责人，这样我们的工作就不会有重复的部分，各司其职，相互帮助，这的确是我对我们小组的最大的感受之一。当然这样还是会有一些问题出现，就是大家的工作时间有时会出现冲突，不过毕竟正值期末季，大家的任务都比较重，所有组都难以避免。

然后我们组隔几天会在腾讯会议上举行会议，各自汇报一下自己这几天的收获（都干了啥），说说自己最近遇到的问题，讨论接下来大家几天的任务，这样我们就会有一个明确的目标，也给我们一个衡量的标椎（我们离我们的最终目标还差多少）。这比我们在微信群里面聊天讨论会快很多，是一个效率高的方法。

当然，我们小组也存在一些问题，首当其冲的是我们的时间有点紧迫，在老师发布大作业之后没有立即开始行动，然后之后的日子编程的各位成员不可避免的成了“熬夜小王子”。印象最深的是我们组长苏遨同学有时肝一整晚，这是我

们应该吸取的一大教训。然后就是我们的具体分工也会有一些问题，比如我主要是负责写报告，当编程同学在群里提出问题时，我很难 get 到那个点，有时看群聊内容有点迷茫（之后索性不管了）。最后就是我们在临近 ddl 的时候代码还会有许多的 bug，当然这与我们的时间分配也有一定的关系，这些都是我们值得改进的地方。

在代码编写的过程中，由于全部代码作为一个整体被编写于一份.py 文件中，所以在后期调试的过程中，不同组员各自调试各自的，以至于发生了严重的代码冲突，使得原本就不充裕的时间显得更加紧迫。在这个问题上，我们得到的教训便是代码应当实行严格的分配制度，每个人负责一个代码包，彼此不应当有任何重叠之处。每个人修改好自己的代码包后都交由组内负责调试的同学进行调试，这样便可以很好地避免混乱修改所导致地版本冲突问题。

4.3 建议与设想

首先我想说，我们的大作业真的已经很好了，游戏设计非常有意思，天梯上各组的对战也非常的刺激。感谢精心设计这次大作业的各位老师与大佬，这里我就提几点小的建议：

对于组队的建议，我知道我们这次分为 19 级区和非 19 级区是更好一点，但我想到一种情况，如果两者在前期没有交手，在出线后，会不会出现某一方完全强一点的状况，这样我们可以考虑：起初时让双方进行少数的对战。

对于大作业的开放时间，我认为可以稍微往前提一点点，比如在期中考试结束后，我们就可以开始着手准备，因为那时候大家的时间都比较充裕一点，同时我们完成本次大作业也是相当的紧迫（这可能与我们处于特殊环境有关），在后期会有许多的 bug 和改进，但是时间真的不太够，我们组员这两周经常熬夜改代码，希望以后能够改善一点。

对于对战平台，我觉得有点慢了，群里也经常有人对此抱怨，希望以后的对战平台能够更快更好一点。

对明年学弟学妹的寄语：

1、选择这门课，前三周你们会很轻松，但到后来会越来越难，希望大家做好准备。

2、自己提前下载相关的书籍，学习编程的基础知识，很多东西老师在课堂上是不会详细讲解的，而且速度会有点快，你需要提前打好基础。

3、学习几周后，你们可能会觉得陈斌老师的数算课 任务量很大，是一堂很累的课（反正我是这样觉得）但是如果你认真对待，你会收获许许多多的快乐，你会

对于你能做到的事情感到无比的惊讶与刺激。

4、这门课会有很多大佬级别的存在每天“虐”你千百遍，你需要调整好自己的心理状态，不要为大佬所打倒。

对于实习作业后续工作：比赛时应该将各个组的代码与其他各组都比一次，自己感觉可能公平一点，还可以将这几届的大作业内容进行整合，让同学们在平时就可以进行大作业的练习。

5 致谢

感谢陈斌老师在这一个学期中对我们数据结构与算法课程尽心尽力的策划。虽然在疫情影响下，我们进行线上教学，但是老师尽心尽力之下，我们仍然学到了很多。

感谢这门课的助教老师们认真负责地批改我们千奇百怪的 K、H 作业。在微信群纷杂的聊天信息中找出我们的问题及时解答。

感谢为我们构想出这次生动有趣的大作业竞赛的各位老师、助教、同学。

感谢我们组的每一个组员在本次大作业实习过程中的付出。

6 参考文献

[1]https://blog.csdn.net/u014296991/article/details/105611011

[2]https://blog.csdn.net/weixin_42165981/article/details/103263211?fps=1&locationNum=2

[3]https://stackoverflow.com/questions/22342854/what-is-the-optimal-algorithm-for-the-game-2048

[4]https://github.com/lihongxun945/myblog/labels

[5]https://www.cnblogs.com/mumuxinfei/p/4396339.html

数据结构与算法课程实习作业报告

迟誉，李宸旭*，铁宸睿，邢景琦

摘要： 我们组的 AI 使用了 minimax 算法实现了博弈树以进行决策。在开发过程中着重进行了搜索深度的优化，通过主要变例搜索极大增加了搜索深度，从而在对局中获得较大优势。使用了子力值作为局面的评估函数，并通过遗传算法在参数空间中搜索最佳的子力值权重。此 AI 在小组赛中最终获得了出线资格。

关键字： 博弈树，minimax 算法，遗传算法，评估函数，主要变例搜索

1 算法思想.....2
1.1 总体思路.....2
1.2 算法流程图.....2
1.3 算法运行时间复杂度分析.....3
2 程序代码说明.....4
2.1 数据结构说明.....4
2.2 函数说明.....4
2.3 程序限制.....9
3 实验结果.....9
3.1 测试数据.....9
3.2 结果分析.....13
3.3 经典战局（可选）.....14
4 实习过程总结.....14
4.1 分工与合作.....14
4.2 经验与教训.....16
4.3 建议与设想.....17
5 致谢.....17
6 参考文献.....18

1 算法思想

1.1 总体思路

二〇四八是一个基于原版 2048 改编而来的对抗性游戏，很自然的，我们最初算法总体思路的设想建立在了对原版 2048 的理解之上，希望能以一种接近原版 2048 的方式，注重于在己方半场合成出更大的棋子，从硬实力上战胜对手。为做到这一点，在'position'模式下，我们优先考虑了在己方半场落子，因为在对方落子棋子归对方，会增强对方的总子力。在算法开发后期，出于进攻和防守的需要，在某些情况下也会考虑在对方落子的情况。

在算法的实现上，由于是完全信息博弈，我们通过构建博弈树来模拟双方的决策过程，并编写了评估函数来对搜索到的局面进行评估。我们用函数的递归调用实现了树这一数据结构，使用极大极小算法的改进算法——negamax 算法（命名仍为 minimax）来确定双方在每一步的最优选择。我们算法的一个特色是在'position'模式下引入了启发性规则来判断是否需要在对方半场落子，当对方的空格数足够少或己方的棋子受到威胁时会落在对方半场，否则只会考虑在己方落子。Alpha-Beta 剪枝的改进算法——主要变例搜索（PVS）被应用于剪枝操作中，以减少搜索的节点数，节约搜索时间。经过剪枝，在相同时间内搜索深度由原来的最多 4 层增加到了 10 层。

在评估函数的选择上，我们最初考虑了多种因素，包括子力值，单调性，平滑性，赋值矩阵等。调参工作主要是由遗传算法完成的，先根据经验人工地提供一组初始值再经过遗传变异、算法之间的交叉互换、相互竞赛，多代进化，以寻找最优的参数组合。经过遗传算法的进化和练习赛中的表现我们最终确定了以一组最优的子力值做为评估函数。

在练习赛的过程中，我们注意到了比赛中最为致命的情况是一方被卡死（也就是塞爆），无法落子也无法移动，这样只能眼睁睁地看着对方合成 2048、4096，可以说一旦被卡死几乎必输无疑，除非劣势很大且剩下的回合过少。为避免被塞爆和企图塞爆对方，我们的算法会对是否塞爆进行判断，如果塞爆会给评估函数加上一个 bonus，bonus 的值也是由遗传算法确定的。

考虑到天梯的运行环境有时会不稳定，使得原本不会超时的算法在比赛中会超时，为避免这一情况出现，我们设置了防超时模块，如果剩余时间低于预期，则最大搜索深度减少一层。增加了防超时模块后，再没有发现因为超时被判负。

1.2 算法流程图

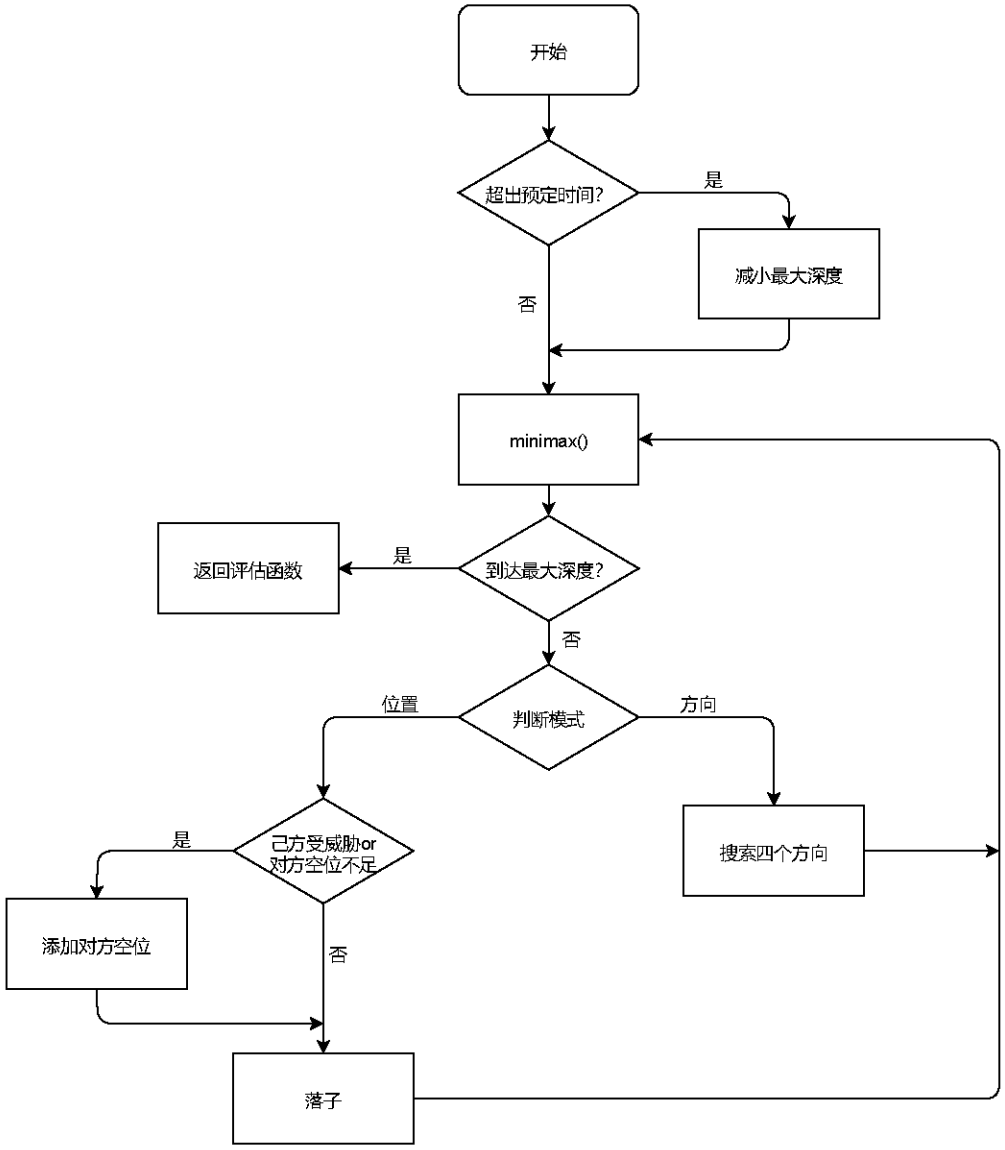


图 1 算法流程图

1.3 算法运行时间复杂度分析

我们算法的主要部分是用 minimax 算法实现的对博弈树的搜索，这占据了运行时间的绝大部分。对于一般的 Minimax（negamax）算法对树执行完整的深度优先探索，若树的最大深度为 m，每个点有 b 个合法有效的动作方法，则算法的时间复杂度为 $O(b^m)$。由于二〇四八这个游戏存在 position 和 direction 两个阶段，在 position 模式下我们的算法会给出一个确定的落子位置，子节点数为 1；而在 direction 模式下，如果不出现卡死，子节点数为 4。所以在不考虑剪枝的情况下算法的时间复杂度应为 $O(2^m)$。现在考虑 PVS 剪枝，经过测试，我们的算法平均剪去了 55%的节点（通过统计剪枝时未搜索的节点数），使得 direction 模式下

搜索的节点数大约减少了一半，这样整体的时间复杂度约为 $O(1.4^m)$。

由于我们采用了精简的基于静态的估值函数，在估值函数上面耗费的时间是 O(1)的。所以综合考虑，算法的时间复杂度约为 $O(1.4^m)$，m 为最大搜索深度。

在我们的实际运行中，如果讲搜索深度固定为 10，那么在一些局面中会出现超时。由于防超时模块的设计，我们的搜索深度在 9-10 之间，具体选择依之前的耗时决定。在天梯的运行环境下，我们发现此程序并没有出现任何超时的情况，因此可以得出结论：搜索深度为 10 时，程序运行 500 回合的平均时间大于 5s，搜索深度为 9 时，程序运行 500 回合的时间小于 5s。

2 程序代码说明

2.1 数据结构说明

算法中采用的主要数据结构是树，用以描述完全信息博弈中的决策过程。

考虑到自定义类可能会产生额外的调用成本，本组程序没有使用自定义类，而是通过函数的递归调用来实现，递归操作也同时完成了对树的深度优先搜索。

在本学期的数据结构与算法课程中介绍了基本的二叉树，本组程序中对它进行了以下拓展和改进：

1. 将二叉改为多叉，使每个节点的子节点数目不固定，对应真实决策过程中每个局面对应的允许操作数不固定。
2. 在深度优先搜索中引入了剪枝操作，可以在搜索过程中判断是否没有必要继续进行搜索，从而大大降低了 DFS 的时间开销。
3. 将子节点和着法进行了对应，这样搜索完成后，每个节点都储存着其所有子节点（可能着法）中最佳的那个。如何维护这样的结构将在下一节进行说明。

过程中具体的参数传递和剪枝过程见下。

2.2 函数说明

本程序中定义了两个函数：evaluation，minimax 和格式要求的 Player 类，以及全局参数若干。下面按照顺序进行说明：

```
# --------parameters--------
MAXDEPTH = 10  # 最大搜索深度
MAXROUND = 499  # 最大回合数
SAFETIME = 4.25  # 安全的最大消耗时间
orderList = [(True,"position"),(False,"position"),\
             (True,"direction"),(False,"direction")]  # 交替进行的四种情况
ef=[0.0, 2.6350325838864244, 4.02155890283844, \
    6.178770918954514,8.902744224582106, 58.46867202851946, \
    169.80587873023782, 404.4969938775643, \
    1197.5663037172623, 3049.2527163289797, 8517.213603995344, 995]  # 参数列表
```

全局参数有五个，分别为 MAXDEPTH：最大搜索深度；MAXROUND：最大回合数；SAFETIME：安全的最大消耗时间，用于防止程序运行超时；orderList：我们将游戏中一回合的四个步骤“先手落子”“后手落子”“先手移动”“后手移动”赋予相应的 order 0，1，2，

3， orderList 用来辅助这一过程；ef：用在局面评估函数中的参数列表。

1.evaluation(board: Chessboard, belong: bool, ef = ef) -> float

```
# --------evaluation--------
def evaluation(board, belong, ef=ef):
    res = 0
    score1 = board.getScore(belong)
    score2 = board.getScore(not belong)
    for piece in score1:
        res += piece*ef[piece]
    for piece in score2:
        res -= piece*ef[piece]
    return res
```

evaluation 是局面评价函数，其参数如下：board：一个 Chessboard 类对象，即当前局面；belong：一个布尔变量，格式与 Player.isFirst 相同；可选参数 ef 即全局变量中的参数列表。返回值为 float，其大小反应了传入棋盘对应的局面对 belong 方的优劣程度。即 evaluation(board, belong)数值越大，对 belong 方越有利。

出于运行时间等因素的考虑，评价函数只考虑了子力这一单一因素，而且没有加入启发项，仅对传入的静态局面进行评价。具体实现依靠 getScore 方法，将双方所有棋子等级与 ef 加权求和得到两方各自的子力值，己方子力值减去对方子力值便为局面对己方的评价值。

2.minimax(order: int, currentRound: int, board: Chessboard, depth: int, alpha = -1e10, beta = 1e10, ef = ef) -> Tuple[float, Union[None, int, Tuple[int, int]]]

minimax 是本程序的核心函数，在 output 方法中直接调用它获得每步决策。其参数如下：order：四个步骤的序号，在递归中进行 0，1，2，3 的循环；currentRound：当前的回合数；board：此刻的棋盘局面；depth：博弈树剩余的搜索深度，每次递归将其减一；可选参数 alpha 和 beta 为剪枝过程中的辅助变量，其含义将在下面的剪枝过程中予以说明。可选参数 ef 即全局变量中的参数列表 ef。

其返回值为一个 Tuple，描述了传入参数的条件下，当前操作方能够做出的最佳操作。第一项是一个 float，提供给父节点进行比较决策，而第二项有三种可能：如果当前条件下无法进行操作（棋盘没有空位/当前方被塞爆），则返回 None（因为递归过程中无法使用外部的_direction 和_position 模式所以需要自己判断何时无法进行操作），否则格式与 Chessboard 类中的 position 和 direction 相同，作为当前条件的最佳操作提供给 output 方法进行输出。

由于函数中为了优化算力，将逻辑处理得有些模糊，下面将不完全按照实际代码中的运行顺序，而是按照我们设计函数时的想法对应的逻辑顺序进行介绍。

```
nOrder = 0 if order == 3 else order + 1  # 下一层的模式编号
nRound = currentRound + 1 if nOrder == 0 else currentRound  # 下一层的回合数
```

nOrder 和 nRound 为下一层的模式编号以及回合数，在递归调用 minimax 中使用。

```
if depth == 0:
    locked = False  # 是否被塞爆
    if order <= 1:  # mode "position"
        belong = (order == 0)
    else:
        belong = (order == 2)
    util = evaluation(board, belong)
    if len(board.getNone(belong)) == 0:
        locked = True
        for direction in [3,2,1,0]:  # 尝试四个方向的移动
            boardCopy = board.copy()
            if boardCopy.move(belong, direction):
                locked = False
                break
    if locked:
        util = util - ef[-1]  # 对塞爆的惩罚
    return (util, None)
```

```
if currentRound > MAXROUND:
    util = evaluation(board, belong)
    return (util, None)
```

如图所示，函数的递归终止条件是 depth==0 或者 currentRound > MAXROUND，返回对此时局面的评估（区分先后手）。depth==0 时还加入了条件判断用以判定当前局面对于操作方是否处于被塞爆状态（即任意方向的移动都无法进行）。如果处于被塞爆状态，那么对当前的评估值减去 ef[-1]中的惩罚偏置，这样可以使决策树尽量规避被塞爆的情况，并尝试塞爆对手。另外，小组中有人提议在达到最大轮数时直接通过判断哪方获胜来返回正负无穷大，经过讨论和本地测试，还是决定返回正常的 evaluation。原因有二：首先如果只返回正负无穷大，AI 将无法从中判断大胜和险胜，导致局末战斗力下降，本地进行的左右互博实验也说明了这一点。其次，我们使用的评价函数对每个 level 的加权近似于 3 的幂次，所以我们采用的评价函数用来描述终局时的胜利条件是很合适的。

下面是当 order 为 0 或 1 时的情况（即位置模式）。

```
theirNone = board.getNone(not belong)

positions = [board.getNext(belong, currentRound)] + theirNone  # 可以下的位置
len(positions[0]) == 0 and positions.pop(0)

if len(positions) == 0:  # 没有空位，这回合
    boardCopy = board.copy()
    val = -(minimax(nOrder, nRound, boardCopy, depth - 1, -beta, -alpha)[0])
    return (val, None)
```

positions 是一个列表，存储当前局面允许落子的所有位置。如果没有空位，直接跳过回合，并将此步对应操作设为 None。

这里不同于正常的极小极大值算法，我们采取了负极大值算法 negamax 的写法来描述对 minimax 的递归调用。正常的极大极小值算法中，需要按奇偶层将节点分为极大值节点和极小值节点，写法上稍显累赘且不易维护。而在 negamax 中，在递归中返回时简单地加了负

号，以反应对弈双方的交替操作。这样每个节点都成为了极大值节点，目标都是取其子节点对应 val 中的最大值。

对于位置模式，我们注意到如果每次都将对方的所有空位加入子节点进行决策，决策树的规模将变得过于庞大，对于其层数的增加相当不利。而我们通过观察天梯上大量的对局以及自己进行大量 pve 对战后发现，对于一般情况，落子于己方区域总是最优解，只有在一些特殊情况下，落子于对方半场才能获得更大收益。因此我们对此采取了启发式规则，即在位置模式下对当前局势进行扫描，如果出现了这些特殊情况就直接固定落子位置，若没有则直接令 AI 在己方半场正常落子。我们设计了两个启发式规则：

```
if len(theirNone) > 0:  # 启发式规则1：试图塞爆对方
    num = 2 if belong else 1
    if len(theirNone) <= num:
        boardCopy = board.copy()
        boardCopy.add(belong, theirNone[-1])
        val = -(minimax(nOrder, nRound, boardCopy, depth - 1, -beta, -alpha)[0])
        return (val, theirNone[-1])
```

启发式规则 1 是为了试图塞爆对方。我们通过观察发现，如果对方的空位数非常少时，在对方空位中落子有很大可能将对方塞爆，从而获得相当大的优势。

```
# 启发式规则2：卡位保己方大子
raw = board.getRaw()
if belong:
    breakFlag = False
    for row in range(4):
        if breakFlag:
            break
        if raw[row][3][1] and raw[row][3][0] >= 5:
            if raw[row][4][0] == 0:
                for col in [5,6,7]:
                    if raw[row][col][0] == 0:
                        continue
                    elif raw[row][col][0] == raw[row][3][0] and not raw[row][col][1]:
                        positions = [(row, 4)]
                        breakFlag = True
                        break
                    else:
                        break
else:
    breakFlag = False
    for row in range(4):
        if breakFlag:
            break
        if not raw[row][4][1] and raw[row][4][0] >= 5:
            if raw[row][3][0] == 0:
                for col in [2,1,0]:
                    if raw[row][col][0] == 0:
                        continue
                    elif raw[row][col][0] == raw[row][4][0] and raw[row][col][1]:
                        positions = [(row, 3)]
                        breakFlag = True
                        break
                    else:
                        break
```

启发式规则 2 是针对己方大子（level 大于 5）进行的保护。其针对的是这样的情况：（TODO：补图）如果己方的大子位于边界线处，对方同样价值的棋子位于对方半场并能够在一步移动之后将己方棋子吃掉，那么此时在边界线处落子可以完成卡位效果，从而起到保护己方棋子的作用。

```
boardCopy = board.copy()
boardCopy.add(belong, positions[0])
val = -(minimax(nOrder, nRound, boardCopy, depth - 1, -beta, -alpha)[0])
return (val, positions[0])
```

如果没有触发启发式规则，那么直接选择在己方半场按照随机列表落子。这样的话，position 模式下子节点个数只会有一个，从而大大提高了搜索效率。

下面是 order 为 2 或 3 的情况（方向模式）：

```
belong = (order == 2)  # 己方布尔值
directions = [0,1,2,3] if belong else [1,0,3,2]
foundPV = False  # 是否找到主要节点
locked = True  # 是否被锁死
```

```
while len(directions) > 0:
    direction = directions.pop()
    boardCopy = board.copy()
    if boardCopy.move(belong, direction):

        if foundPV:
            val = -(minimax(nOrder, nRound, boardCopy, depth - 1, -alpha-1, -alpha)[0])
            if val > alpha and val < beta:  # 候选值已经失败
                val = -(minimax(nOrder, nRound, boardCopy, depth - 1, -beta, -alpha)[0])
        else:
            val = -(minimax(nOrder, nRound, boardCopy, depth - 1, -beta, -alpha)[0])

        _val = (val, direction)  # 用来记录移动的信息，_表示这是一个元组
        if locked:
            _best = _val  # 保证第一个节点都行

        if val >= beta:
            return (beta, direction)  # 直接返回搜索的节点

        if val > alpha:
            alpha = val
            foundPV = True
            _best = (alpha, direction)  # 更新最佳节点

        locked = False
```

在方向模式下，用四个方向作为四个子节点，如果该方向上可以进行移动，那么对此节点进行 DFS 搜索。在正常搜索过程中，常见的做法是通过 alpha-beta 剪枝来剪去不必要的搜索以减少博弈树规模。本程序中则使用了主要变例搜索（PVS）进行剪枝，这是在 alpha-beta 剪枝的基础上的改进算法，在（TODO：reference）中证明了其拥有比传统 alpha-beta 剪枝更高的剪枝效率。

先明确一些基本概念：每个 minimax 函数附带一个窗口（alpha，beta），作为对本节点返回值的估计。其中 alpha 为受对方制约，己方已知的最佳叶节点对应的分值；beta 则为受己方制约，对方已知的最佳叶节点对应的分值。在搜索的过程中不断更新 alpha，beta，使得窗口的大小不断减小。在搜索过程中，任何节点都属于以下三种类型：1.alpha 节点，所有子节点返回值都小于等于 alpha；2.beta 节点，存在一个子节点返回值大于等于 beta；3.主要变例节点（PV 节点），不为以上任何一种的节点，即存在返回值大于等于 alpha 的子节点（子节点对应的着法称为 PV 着法），但是没有着法返回大于等于 beta。在负极大值的框架下，我们需要找到返回值最大的节点，显然如果本节点成为了 beta 节点，那么对手将不会允许我们选择这里，故不必做剩余搜索，直接返回 beta，完成剪枝。这是传统 alpha-beta 剪枝的过程。

PVS 的核心假设是：如果在找到了一个 PV 着法，那么本节点就是一个 PV 节点，而且这一着法是全部着法中最好的那个，即剩余子节点的返回值都小于等于该 PV 着法的值。在这一假设下，对于第一个子节点，设其返回值为 val 且位于窗口之内，那么接下来用极小窗口（val，val+1）来搜索剩余子节点，如果假设成立，那么这些剩下的节点返回值都小于等于 val。那么极小窗口搜索不会对结果造成影响，而且还能加快剪枝的效率。当然如果不满足这一假设，那仍需用原来的窗口（alpha，beta）进行搜索。

```
if locked:  # 被堵塞了
    boardCopy = board.copy()
    val = -(minimax(nOrder, nRound, boardCopy, depth - 1, -beta, -alpha)[0])
    return (val, None)
else:
    return _best
```

另外还考虑了四个方向都没有被成功调用，也就是被塞爆的状态，此时直接空过当前操作即可。

```
# --------Player--------
class Player:
    def __init__(self, isFirst: bool, array: tuple) -> None:
        self.isFirst = isFirst
        self.array = array
        self.maxdepth = MAXDEPTH

    def output(self, currentRound: int, board, mode: str):
        if mode in ['position','direction']:
            order = orderList.index((self.isFirst, mode))
            if mode == 'position' and currentRound % 10 == 9:  # 
                self.maxdepth = MAXDEPTH
                remainTime = board.getTime(self.isFirst)
                scheduleTime = SAFETIME*(1 (currentRound/MAXROUND))  # 
                if remainTime > 5 - SAFETIME and remainTime + SAFETIME - 5 < 8.8*scheduleTime:
                    self.maxdepth -= 1
                if remainTime < 5 - SAFETIME:  # 
                    self.maxdepth = MAXDEPTH - 1
            return minimax(order, currentRound, board, self.maxdepth)[1]
        else:
            return
```

最后是格式要求的 Player 类。其主体调用了 minimax 函数进行决策，唯一不同之处是加上了防超时模块。大致思路是规定程序在 SAFETIME 左右完成一轮比赛，每过 10 回合评估剩余时间，如果剩余时间比计划中的剩余时间短上很多，那么在接下来的十个回合内减小一层最大搜索层数。如果超过 SAFETIME 仍未结束，那么同样减小一层搜索层数。这样的设置可以保证程序在各种环境下都不会出现超时错误。

2.3 程序限制

本程序在开发过程中进行了大量的本地以及线上测试，并参加了天梯小组赛，在上千场比赛后成功取得了出线资格，在此过程中并没有一次因为出现报错而输掉比赛。因此可以说：我们的程序是没有 bug 的。

此外，我们在本地调试过程中编写了额外的调试工具 debug.py,在下一节中将对其进行介绍。

3 实验结果

3.1 测试数据

实验环境说明：

- 硬件配置：i7-9750　GTX1650
- 操作系统：win10.0 17763

- Python 版本：3.8.1

本小组采用了遗传算法来对评估函数的参数进行调整。简单介绍如下：

1、 遗传算法(Genetic Algorithm , GA)

借用了生物遗传学的观点,通过选择、遗传、变异等作用机制，实现个体适应性的提高,体现了自然界中“物竞天择、适者生存”的进化过程，在机器学习中被广泛应用。

2、 适应度函数的计算——锦标赛算法

2048 游戏中，要想对评估函数的某组参数给出一个“适应度”并不是可以直接做到的事情，只能通过装配了这组参数的 AI 在对战中的表现来反映，因此我们采用了锦标赛算法，首先将整个种群分成十组，每组内两两对战，在相同适应度的初始状态下，赢了对适应度进行加运算(奖赏)，输了对适应度进行减运算(惩罚)，和了则不操作。最后挑出每个组的冠军（适应度最大的个体）作为下一代的亲本。

3、均匀交叉

由亲本基因确定子代基因的方法有很多，这里我们采用均匀交叉。从亲本中随机取出两个个体，两亲本每一个位置上的基因都有相等的概率（交叉率）进行交叉互换，从而产生两个子代。

4、变异

为增加基因的丰富性，一定程度上防止局部收敛，这里引入了变异机制。子代产生后，对每个子代个体的每一位基因，都有一定的概率变成一个在给定范围内的随机数

5、自适应遗传算法

在整个遗传算法实现的过程中，交叉率 pc 和变异率 pm 的选择是影响遗传算法行为和性能的关键所在，直接影响算法的收敛性。pc 越大，新个体产生的速度就越快，然而 pc 过大时遗传模式被破坏的可能性也越大，使得具有高适应度的个体结构很快就会被破坏;但是如果 pc 过小，会使搜索过程缓慢，以至停滞不前。对于变异率 pm，如果 pm 过小，就不易产生新的个体结构;如果 pm 取值过大，那么遗传算法就变成了纯粹的随机搜索算法。

因此本小组查阅资料，发现了一种“自适应遗传算法”，即可以通过现有种群的各种数据自动计算出交叉率和变异率。公式如下：

$$p_c = p_{c1} - \frac{(p_{c1}-p_{c2})(f'-f_{avg})}{f_{max}-f_{avg}} \qquad p_m = p_{m1} - \frac{(p_{m1}-p_{m2})(f_{max}-f)}{f_{max}-f_{avg}}$$

式中，p_{c1} =0.9 , p_{c2} =0.6 , p_{m1} =0.1 , p_{m2} =0.001。 f_{max} 为群体中最大的适应度值;f_{avg} 为每代群体的平均适应度值;f'为要交叉的两个个体中较大的适应度值;f 为要变异个体的适应度值。这种自适应的 p_c 和 p_m 能够提供相对某个解的最佳 p_c 和 p_m，在保持群体多样性的同时，保证了遗传算法的收敛性。

6、多进程

为了充分利用处理器的性能（充分剥削处理器的剩余价值），运行遗传算法时，使用了 python 中的 multiprocessing 库进行多进程运算，大大提高了炼丹效率（x）

直接放出代码

```
DNA_SIZE=12
POPULATION=40
N_GENERATIONS = 25
```

```
def tournament(A,B,q,index0,index1):#A,B是两组个体的函数与参数，两个组各个体成绩的参数外，记相同
    #而传递类似拷贝个体的作为 q获取上级比赛结果
    orderlist = [(True,"position"),(False,"position"),(True,"direction"),(False,"direction")]
    from __main__ import winisax,MAXDEPTH
    class Player1:  #引入两个类
        def __init__(self, isFirst: bool, array: tuple) -> None:
            self.isFirst = isFirst
            self.array = array

        def output(self, currentRound: int, board, mode: str):
            global MAXDEPTH
            safe = True
            if mode in ['position','direction']:
                order = orderlist.index((self.isFirst, mode))
                return winisax(order, currentRound, board,A,8)[1]
            else:
                return
    class Player2:
        def __init__(self, isFirst: bool, array: tuple) -> None:
            self.isFirst = isFirst
            self.array = array


    def output(self, currentRound: int, board, mode: str):
        global MAXDEPTH
        safe = True
        if mode in ['position','direction']:
            order = orderlist.index((self.isFirst, mode))
            return winisax(order, currentRound, board,B,8)[1]
        else:
            return
from race import main
[Afitness,Bfitness]=main([Player1,Player2])
print(Afitness,'\n',\
      A,'\n',
    Bfitness,'\n',
      B,'\n')
q.put((index0,index1,Afitness,Bfitness))  #多进程要求函数没有return 所以直接让队列里添加


def crossover(parent,fmax,fmin):#输入一个parent 列表，列表每一项是一个（chromosome,fitness）元组,Pc是交叉率
    children=[]
    for i in parent:  #把当上一代父代拷贝，为其交叉子代继续进化
        children.append(i)
    while len(children)<CNFOLMIDS:  #直到子代数量与种群规模一样大
        child=[]
        child2=[]
        a=random.randint(0,len(parent)-1)  #从父代中选择进行两个个体
        b=random.randint(0,len(parent)-1)
        pc=0.9 if fmax-0 else (0.4-0.9)*(max(parent[a][1],parent[b][1])-fmin)/(fmax)
        #下面以选择算法"轮盘赌法"给出的交叉率
        for j in range(DNA_SIZE):  #遍历基因，对每一个基因位点，都有一定概率交换
            twical= 0 if np.random.random_sample()>pc else 1
            if twical ==0:
                child.append(parent[a][0][j])
                child2.append(parent[b][0][j])
            else:
                child.append(parent[b][0][j])
                child2.append(parent[a][0][j])
        children.append((child,parent[a][1]))  #children中加入了杀父当未变的child以及其对应的父代fitness
        children.append((child2,parent[a][1]))
    return children
```

```
def mutate(children, fmax, favg):
    for child in children:
        pm=0.05 if fmax==0 else 0.1-0.09*(fmax-child[1])/fmax#"自适应算法"给出的变异率
        for i in range(DNA_SIZE):
            if rm.random.random_sample() < pm:  #实现变异，对某些位上的系数随机
                if i in [1,2,3,4,5,6,7,8,9,10]:
                    child[0][i] = random.uniform(of[i]*8,of[i]*1.2)
                elif i==11:
                    child[0][i]=random.randint(400,1000)
    return children
```

```
weknow=af    #weknow是基于人类给出设定的参数初值
pop=[  for i in range (POPULATION)]   #初始生成种群时标从正负死区上下波动30%
for j in range(POPULATION):
    for i in range (len(weknow)):
        pop[j].append(random.uniform(0.7*weknow[i],1.3*weknow[i]))
pop[0]=af #人工预值也放进种群中
#行数=POPULATION ，列=DNA_SIZE
for   in range (N_GENERATIONS):  #N代进化
    group=[[] for i in range(10)]
    fit=[0 for i in range (POPULATION)] #此行记录每个个体的适应程度
    for i in range (10):   #把种群分成10组，每组进行竞标赛
        for j in range (int(POPULATION/10)):
            group[i].append(pop[int(POPULATION/10)*i+j])
```

```
if __name__ == '__main__':
    for i in range(10):  #分别 组，每组随机取出了4个，所以每组4个个体间可两两对垒，代码采（同志最好采集的版）
        q=mp.Queue()
        p1=mp.Process(target=tournament,args=(group[i][0],group[i][1],q,0,1))
        p2=mp.Process(target=tournament,args=(group[i][0],group[i][2],q,0,2))
        p3=mp.Process(target=tournament,args=(group[i][0],group[i][3],q,0,3))
        p4=mp.Process(target=tournament,args=(group[i][1],group[i][2],q,1,2))
        p5=mp.Process(target=tournament,args=(group[i][1],group[i][3],q,1,3))
        p6=mp.Process(target=tournament,args=(group[i][2],group[i][3],q,2,3))
        p1.start()
        p2.start()
        p3.start()
        p4.start()
        p5.start()
        p6.start()
        p1.join()
        p2.join()
        p3.join()
        p4.join()
        p5.join()
        p6.join()
```

```
        while not q.empty():#统计对手胜场
            tp=q.get()
            score[int(tp[0])]+=tp[2]
            fit[int(tp[0]*POPULATION/10)+tp[1]]=tp[2] #tp[0]第一个个体在组内的序号
            #tp[2]第一个个体的胜场数
            fit[int(tp[1]*POPULATION/10)+tp[0]]=tp[3]

    best_index=[i[0] for i in sorted(enumerate(fit), key=lambda x:x[1],reverse=True)[:int(POPULATION/10)]]
    #获得最终出现最多人个体的序号（前10%）
    best_individual=pop[best_index[0]]  #最一个 代最好的个体
    parent=[pop[i],fit[i]] for  in best_index  #parent此为上一代最好的部分个体的适应度
    max=max(fit)  #这两个表现出为了新的适应度和变异率
    Log=open('log.txt')
    child=crossover(pop=pop,fmax,favg)  #交叉生成下一代种群
    child=mutate(child=pop,fmax favg)  #突变后的变化
    pop=[child=a[0] for a in range (len(child))]
    print('max_individual in generation',_,':',max_individual)
    fo = open("out.txt", "w")
    fo.write( str(best_individual))
    fo.write('\n')
    fo.close()
```

通过遗传算法，我们得到了（在内战中）表现更好的 AI

	使用遗传算法调参后	未使用遗传算法	调参后胜率
本地胜利局数	113	47	70.6%
竞技场胜利局数	58	30	65.9%

此外，为了在开发过程中更加精细地进行调试，以及逐步跟踪函数的调用过程，我们在技术组提供的本地调试工具之外额外编写了调试程序。将 plat.py 中的本地运行环境进行复现，并编写了从 str 到 Chessboard 的反解析函数。这样可以在任意给定的棋盘局面、当前轮数、模式下分析 AI 的行为。在程序的开发调试过程中起到了重要作用。

```
# 传入指定位置棋盘的str形式,str2board会将其解析为Chessboard类,实现断点调试功能
def str2board(board, astr):
    belongs = {'+':True, '-':False}
    info = astr.split()
    for i in range(len(info)):
        belong = belongs[info[i][:1]]
        position = divmod(i,COLUMNS)
        temp = info[i][1:]
        if temp[0] == '0':
            temp = temp[1]
        val = int(temp)
        val and board.add(belong, position, value = val)
```

反解析函数

```
r = 3
while r < 500:
    try:
        order = 0
        if_position(board, True, r) and board.add(True, player1.output(r, board, 'position'));order+=1
        if_position(board, False, r) and board.add(False, player2.output(r, board, 'position'));order+=1
        if_direction(board, True) and board.move(True, player1.output(r,board, 'direction'));order+=1
        if_direction(board, False) and board.move(False, player2.output(r,board,'direction'));order -=0
        r += 1
    except (IndexError,TypeError) as reason:
        print(str(reason))
        print(board)
        print(r)
        print(order)
        break
print('pass')
print(board)
```

测试一轮比赛

```
astr = ''
+00 +00 +00 +00 -00 -03 -00 -00
+04 +10 +03 +01 -03 -01 -00 -00
+09 +05 +06 +07 -01 +07 -01 -00
+02 +03 +01 +03 +02 -01 -05 -00
''
newBoard = Chessboard(ARRAY)
str2board(newBoard, astr)
print(player1.output(499,newBoard,'position'))
```

断点调试功能示例：astr 直接拷贝自 roundmatch 提供的报错信息，从而可以通过单步调试看到问题出在何处，成为了已有工具的有力辅助。

3.2 结果分析

经过小组成员反复的交流与讨论，我们发现，一个相对全面的评估函数在时间上的开销是显著的（即使只考虑子力值和每个位置的权重，这样的开销也十分明显）因此最终我们放弃了复杂的评估函数，采用了最为简易的评估函数，即只考虑子力值，用每个棋子的值乘上一个参数后线性组合形成评估函数，从而最大程度减小在这方面的开销，增强搜索树的算力。

这种做法的利弊都很明显。利处在于将所有的时间和算力集中在了搜索树上，从而可以尽可能搜得深一些，最大深度达到了 10 层，我们相信这一层数这在本次的所有 AI 中是名列前茅的。层数带来的优势使得我们的 AI 更加“目光长远”，在面对一些搜索树层数不如我们

深的 AI 时能产生较强的克制。体现为在无干扰情况下，我们的分数积累速度几乎总是快于对方。弊端在于一旦遇到搜索层数与我们相当，同时装配了较为合理的评估函数或启发性策略的 AI，由于决策树不认为对方策略是好的，就会导致决策出现失真，从而落入对方的策略之中，几乎没有还手之力。在淘汰赛中，我们在第一轮中碰到的对手就使用了优先塞爆对方的进攻型策略。在这样的策略下，由于缺乏应对措施，我们几乎每一轮都被塞爆大量回合，导致大比分落败从而被淘汰出局。

算法用时方面，由于装配了防超时功能，在剩余时间较少时自动减少搜索深度，时间掌控上还算合理，几乎不会出现濒临超时的现象。

主要的运行时间开销发生在搜到最深一层后对每个结点使用评估函数。所以搜索层数对时间开销带来的影响是指数级的。

3.3 经典战局

http://162.105.17.143:9580/match/rlMzRX38lp/1/ （策略的劣势下依靠搜索深度的优势取胜）

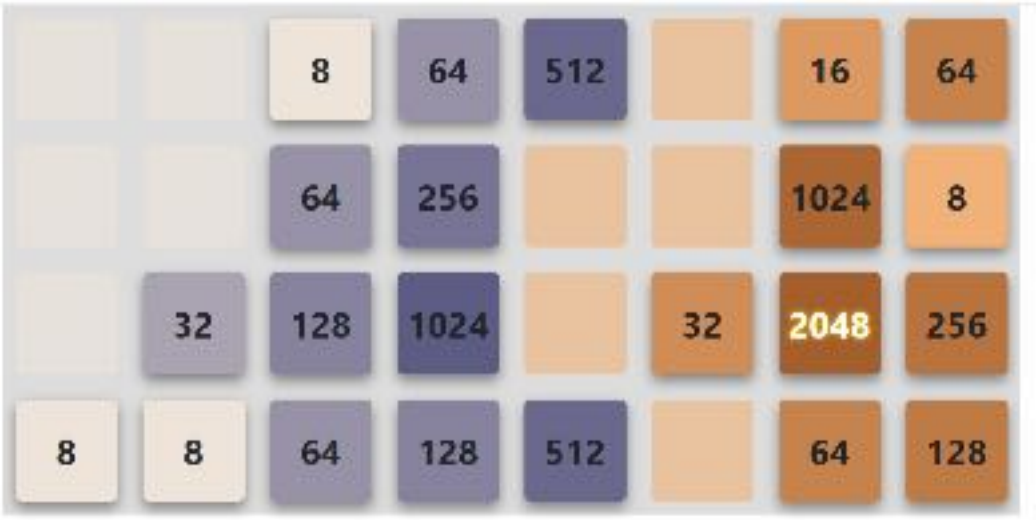


图 2 经典战局

这是练习赛中的一局比赛，我方为后手。对手方是一个进攻型 AI，从第 61 回合开始就牢牢的占据了我方半场的两个格子，直到对局的终止不曾后退一步，我方一直处于防守的状态。然而，虽然我方一直被压制，但依靠着搜索深度的优势，避开了那些可能被塞爆的情况，不断地在夹缝中求生存，242 回合合成出 2048，478 回合合成出 4096，二级压制获胜！

4 实习过程总结

4.1 分工与合作

在首次组会上，我们确定了 AI 设计的四个主要步骤：1.通过在线上和线下观察大量对局以获得策略上的启发，并学习其他小组的优秀策略；2.设计良好的决策树结构，使用剪枝等

方式进行优化以节省搜索时间；3.设计合理的评估函数，作为决策树的补充，为其提供来自人类的启发性智慧；4.用遗传算法等方式进行左右互搏以得到参数空间中的最优解，从而进一步增强 AI 的表现。整个过程中还伴随着大量的线上和线下测试。

正好我们小组有四个人，所以就主要按照这四个步骤进行了分工：迟誉同学负责观察对局，进行测试并总结策略，同时为其他成员提供建议和改进方案；李宸旭同学负责编写决策树结构；邢景琦同学负责编写评估函数；铁宸睿同学负责通过遗传算法进行相关的参数优化过程。

小组的合作交流方式主要有三个：分别是微信，腾讯会议和 Github。在微信上主要进行小文件和代码片段传递，以及在非组会时期快速交流想法和思路。历次的组会在腾讯会议上进行，平均每次组会时长在 2 小时左右。在组会上小组成员依次总结自己之前的成果，并对之后的发展方向进行热烈的讨论。Github 则用来管理历代程序，以防止出现版本上的混淆。我们为此建立了一个 repository 来存放为本次大作业编写的全部程序，并附有版本开发日志。

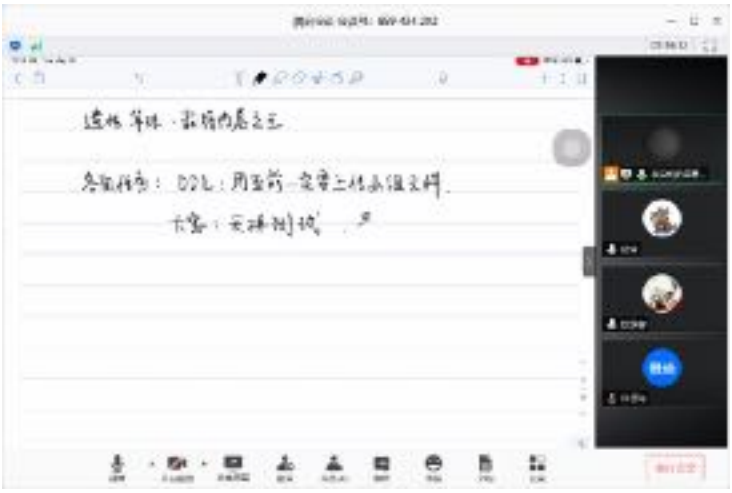


图 3 腾讯会议截图

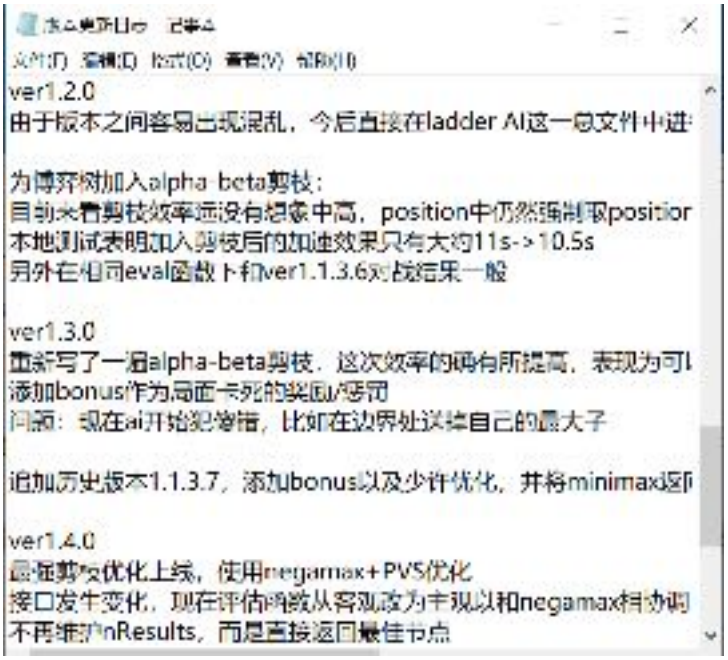


图 4 更新日志

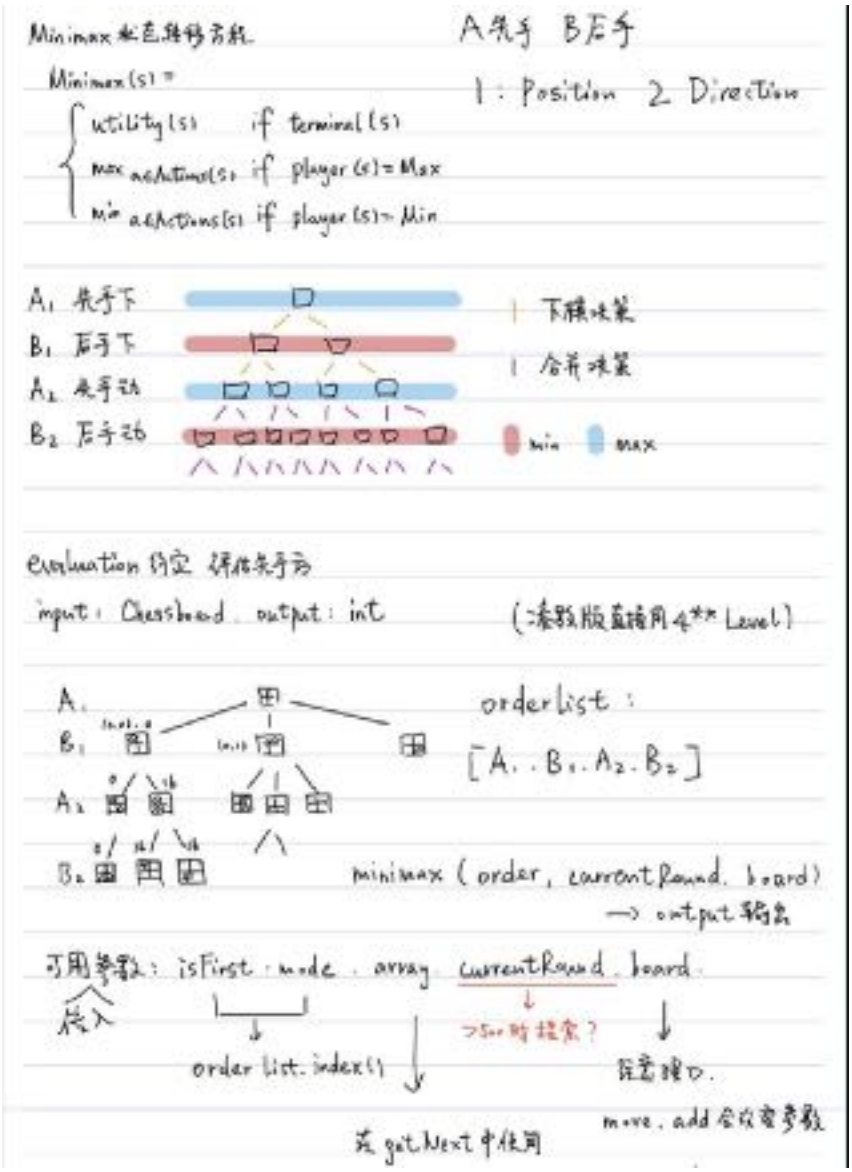


图 5 组会笔记

4.2 经验与教训

我们组最大的遗憾就是由组员精心设计的评价函数在最终版本中被放弃，而采用了最为简单的计算子力值作为局面评估。这也是导致我们在淘汰赛中被大比分淘汰的重要原因。在这个过程中也收获了一些宝贵的教训。

一个教训是理论要与实践相结合。我们组的分工中，由邢景琦同学负责涉及评估函数。他从理论角度考虑了很多全面的因素，比如进攻，防守等启发性的思路，并加入了子力值、局面有序性、局面平滑性、移动自由度等诸多因素。而在前期由于还没有进行调参，我们的测试的重点放在了博弈树结构的改进与优化上，而没有将评估函数加入到树中进行测试。这导致在开发过程的中后期我们才发现邢景琦同学设计的函数并不能与博弈树很好结合，因为在复制棋盘以及分多种情况讨论的过程当中消耗了大量的时间，再考虑到博弈树在每个叶节点都需要调用评估函数进行计算，形成了指数级的时间复杂度，在实际应用当中限制了搜索深度，阻碍了博弈树功能的发挥。另外设计的启发性项也没有发挥其应有的功能。这个教训告诉我们，完全将开发过程割裂开来而不进行理论和实践的结合，往往不会发挥出很好的效

果。

另一个收获是机器和人思考方式的差异。前期的评估函数设置主要按照人的思路进行模拟，希望通过评估函数的规范实现一种有序平滑的阵型，即子力整体按照大小顺序排列，并且持续把最大的棋子安排在棋盘的某个固定位置或者固定方向。但是在实际操作过程中发现，为了维持这样的阵型，我方子力总是相对分散，外围一层层的阵型最后并没有价值，大量浪费了子力。之所以会这样，很重要的一个原因就是没有考虑到人机思考方式的不同。对于 AI 来说，像人一样排出有序的阵型是不必要的。因为它掌握着生成棋子的随机序列，同时拥有着更深的搜索深度。以后如果有机会再进行 AI 开发，一定会尤其注意这一点。

还有一个教训是优化存在边际效应。我们在前期的开发过程中始终认为搜索深度是“一寸长一寸强”的，所以几乎所有对树结构的改进都是如何提速，如何增加搜索深度。在早期的天梯环境中也确实如此。当时我们的 1.1 版本代码以 8 层的搜索深度，在无启发的情况下成功在 5.20 左右的天梯中达到了第三名。这也在一定程度上干扰了我们后续的开发策略。接下来的重心仍然是对深度进行优化。通过更加细致的剪枝，以及干脆默认在己方半场落子，并忍痛放弃了组员编写的评估函数，我们又将搜索深度扩展到了 10 层。但是这时层数优化的边际效应出现了，当对手使用启发性策略时，由于采取了不同的评价标准，博弈树的思考过程出现了失真，从而使得单纯的深度增加无法破解对手的策略。这一问题我们在小组赛开始时才意识到，而时间已经不允许从头开发新的策略了。这一教训告诉我们单一方向的优化存在着边际效应，对于 AI 这种复杂系统更是如此。必须综合进行考虑，才能获得整体最优的表现。

4.3 建议与设想

本次大作业是一个基于单人版 2048 设计的对战游戏，让很多人没想到的是它最后的对抗性会那么的强，使得每一局游戏的对局都有很强的观赏性。从一个熟悉的小游戏开发出这样一个大作是非常了不起的，希望明年还有大作业并且还能这样精彩。

算法的开发不是一件简单事，很多时候我们会发现，当你把脑子里的设想实现出来后，结果可能不尽人意。这时，不要轻言放弃，可能还需要一些调试，可能还需要一小点的改进，只要你不停地尝试，在不经意间就能形成很大的突破。还有一个重要的点是大家之间要多多交流，就如同我们使用的遗传算法一样，思想的交融碰撞是进步的一大源泉，你的想法和我的想法结合起来会产生绝妙的效果。分工中也是同样的道理，每个人的算法结合在一起才是一个完整的算法，它能完成我们每个人单独无法完成的事情。期待学弟学妹们能在明年的大作业开发出更强有力的 AI 来。

5 致谢

感谢陈斌老师为我们提供这次大作业实习的机会，带我们领略算法的魅力。

感谢助教和技术组的同学搭建了天梯和线下对战平台，呈现了精彩丰富的比赛过程。

感谢助教老师们的辛勤付出，孜孜不倦地为我们解答问题。

最后感谢 kilo 组，通过一场非直接的对战把我们抬进了淘汰赛。

6 参考文献

1. http://www.xqbase.com/computer.htm
2. 《PC 游戏编程》王小春编著 重庆大学出版社
3. 《Artificial Intelligence： A Modern Approach》3rd edition by Stuart J.Russel and Peter Norvig

数据结构与算法课程实习作业报告

（陈可人* 袁子珉 虞晨煦 李卓逸 陈浩良）

摘要：如何评判移动方向和落子位置的优劣是二零四八的核心所在，我们的程序在使用极大极小方法以及 alpha-beta 剪枝完成决策树的构建后，通过给定价值函数的方式达到对移动方向和落子位置进行决策的目的。在采用偏向防御型的评估函数的同时，我们的放子操作通过在敌方缺陷处加子的方式在一定程度上增强了整体程序的攻击性。在第五代 sierra 程序中，我们把放子程序和评估函数共同放入了决策树，从而极大地增强了程序决策的整体性。在 sierra 系列程序中，简洁明了的评估函数以及极具针对性的放子函数使得程序在耗时较短的情况下取得了不错的成绩。

关键字：List, Dict, AlphaBeta pruning algorithm, value funnction

目录

1.算法思想 3

1.1.总体思路 3

1.2.算法流程图 3

1.3.算法运行时间复杂度分析 4

2. 程序代码说明 4

2.1.数据结构说明 4

2.2.函数说明 4

2.3.程序限制 11

3. 实验结果 12

3.1.测试数据 12

3.2.结果分析 12

4. 实习过程总结 13

4.1.分工与合作 13

4.2.经验与教训 14

4.3.建议与设想 14

5. 致谢 14

6. 参考文献 15

1.算法思想

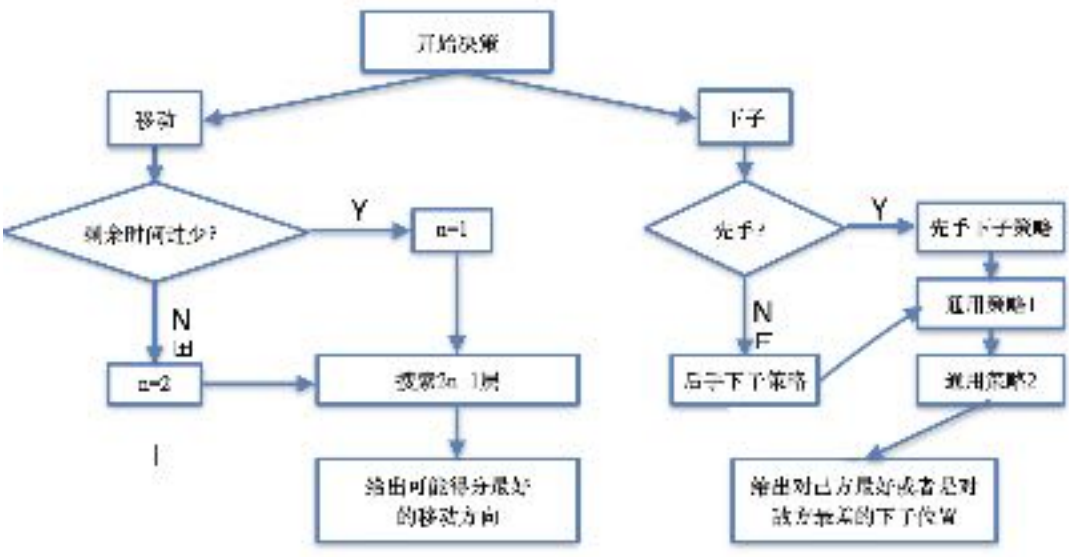
1.1.总体思路

算法的总体思路是利用价值函数给棋盘局面评分，并通过模拟接下来几回合双方的行动来得到往四个方向移动时棋盘得分情况。对敌方进行的模拟移动和放子是按照使我方得分最低的方向。我方移动则是穷举四个方向找到可能出现的分数最大值。

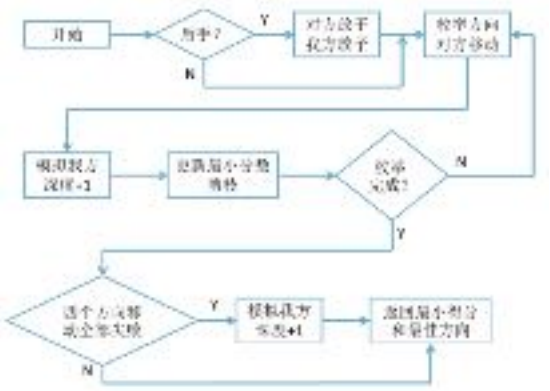
在最初的版本中，我们对价值函数的确定是依照参考资料中的平滑性单调性空格数以及最大子综合考虑的，因此每一层的搜索耗时导致最终只能往下搜索一层，同时也没有时间留给特定的落子算法。在天梯自由对战屡屡落败后小组成员更新了自己的想法，决定采用时间复杂度为 O（n）的价值函数，并且只考虑棋子的大小不考虑棋子的位置，平滑性的方面考虑采取更高的搜索层数来弥补。同时，在技术组更新了 getRaw 接口之后，我方的用时出现了大幅度减少，一度整盘比赛的总用时稳定在 1s 左右，这促使我们开始寻求更合理的时间利用方式。经过讨论之后，小组决定将用时花在落子程序上并且适当增加棋盘局面评估函数的用时。

最后的版本中落子函数占据了源代码将近一半的篇幅，原本我们是想模拟敌方的落子函数，但是考虑到不同代码对落子评分的可能性过于繁多，因此只用了随机落子程序模拟对方。在对价值函数的修改里，增添了对敌方侵入我方领域棋子的位置扣分以及相对移动前棋盘相比有棋子被吃情况下的惩罚。

1.2. 算法流程图



总流程图



扫描中轮到模拟对方行动

扫描中轮到模拟我方移动

1.3. 算法运行时间复杂度分析

算法运行时间主要用于每次搜索中的模拟行动、棋盘局势评分以及下子策略三个方面。

由于本组算法所需要的计算量与棋盘上的棋子数成线性关系，与回合数没有直接的关系，所以各回合的耗时基本相同，因此我们算法的时间复杂度是 O（n）级。

在实际运行中小组代码从未超时，往往整局比赛下来只用了规定时间的一半，可以说是非常“节能”的算法了。（折棒点赞.jpg）

2. 程序代码说明

2.1. 数据结构说明

在本小组所采用的算法中，主要采用的数据结构是列表以及字典。

列表的主要作用是存放 ChessBoard 的内容，配合 .getRaw 接口用以快速访问，而不用一次次调用棋盘的接口，达到减少用时的结果。同时将价值函数中的参数也存放在列表中方便取用。

字典的主要作用是通过 True 以及 False 两个 key 值，分别存放先手以及后手条件下的特殊参数。以及在 bestSolution 函数作出决策时返回一个字典，提升代码可读性。

2.2. 函数说明

2.2.1 __init__

作用：函数初始化，内置各参数及价值函数所需调用列表

成员：

1）self.isFirst：己方是否为先手

2）self.array：下子随机序列

3）self.alist：通过$alist_n = 2alist_{n-1} + n - 1$，$alist_0 = 0$，$alist_1 = 2$ 给出的列表，用以评价我方子的加分权重

4）self.liist：通过 $liist_{2i} = \frac{liist_{2i+1}}{\sqrt{2}}$，$liist_{2i+1} = alist_i$ 给出的列表，用以评价敌方子在我方棋盘范围内的减分权重。

5）self.blist： 通过 $\begin{cases} blist_i = 2^(i+1) & i > 0 \\ blist_i = 0 & i = 0 \end{cases}$ 给出的列表，对 alist 的补充参数

6）self.p：与该回合前棋盘上棋子总大小相比较，在有棋子被吃的情况价值函数的减分惩

7）self.clist：棋子的真实值列表

源代码：

```
def __init__(self, isFirst, array):
    # 初始化
    self.isFirst = isFirst
    self.array = array
    self.liist=[0, 1.4142135624746193, 2, 3.5355340618654822, 5, 8.485281748477156,\
                12, 19.091883934873684, 27, 41.81219511763959, 58, 85.55592429714466, \
                121, 175.36248946985279, 248, 355.67472062366875, 503, 717.0063077463197,\
                1014, 1440.3765768839973, 2037, 2887.8242217317256, 4084,\
                5783.426618399556, 8179, 11575.338518547589, 16370]
    #self.liist[2i+1]=self.alist[i],self.liist[2i]=self.liist[2i+1] /div /sqrt(2)
    self.alist=[0, 2, 5, 12, 27, 58, 121, 248, 503, 1014, 2037, 4084, 8179, 16370]
                                        #递推公式 a_n=a_[n-1]*2+n-1,n-1系数促进大子合成
    self.blist=[0,4,8,16,32,64,128,256,512,1024,2048,4096,8192]
    self.p=100#100: 被吃子减分
    self.clist=[1,2,4,8,16,32,64,128,256,512,1024,2048,4096]
    #a,b list:两个算分函数
```

2.2.2 getScore

作用：价值函数，用以判别场上局势，以采用相应的策略

参数：

1）cb 当前棋盘

2）side 判断是否先手

3）initsum 搜索前原始棋盘上棋子分值和

4）rounds 已经搜索的轮次

源代码以及评分策略：

```
emp=cb.getNone(side)#获取空格数
if len(emp)<=2:
    a-=(3-len(emp))*32
```

1）在空格数小于二的情况下扣分，保证不会轻易卡死

2）主棋盘评分程序

最后的总得分为 a+b

以自己为先手方时评分为例，对自己的棋子仅以大小计算得分，对敌方的棋子在大小基础上还考虑位置，同时对每个侵入我方的地方子给予 64 分的减分。

如果移动后棋盘上我方棋子总分数小于移动前，说明我方有棋子被吃，给予 100 分减分。

```
bo=cb.getRaw()
for i in range(4):
    for j in range(4):
        if bo[1][j][1]==side:
            a+=self.alist[bo[i][j][0]]
            b+=self.blist[bo[i][j][0]]
            c+=self.clist[bo[i][j][0]]
        else:
            a-=self.liist[bo[i][j][0]*2+3-j]+64#64：对我方占有格子的加分,
                                               #防止被吃子，3-j:格子j坐标的加成
            b-=self.blist[bo[i][j][0]]
    for j in range(4,8):
        if bo[i][j][1]==side:
            c+=self.clist[bo[i][j][0]]
            if j==4:
                a+=self.alist[bo[i][j][0]]+24#
                b+=self.blist[bo[i][j][0]]
        else:
            a-=self.liist[bo[i][j][0]*2+1]#形式f(n)*2**1/2
            b-=self.blist[bo[i][j][0]]
a=a+b
if c<initsum0:#判断分数是否减少
    a-=self.p
```

2.2.3 getPosition

作用：

获取双方都没有特殊的落子策略下（也就是说先在己方落子，己方满了在对方落子）的落子位置，用于搜索时模拟落子。

参数：side, board, currentRound——是否先手，当前棋盘，当前回合数

源代码：

```
def getPosition(self, side, board, currentRound):
    another = board.getNext(side, currentRound)  # 己方的允许落子点
    if another != (): return another

    available = board.getNone(not side)  # 对方的允许落子点
    if not available:    # 整个棋盘已满
        return None
    else:
        from random import choice
        return choice(available)
```

2.2.4 down2theSea

作用：运用下子策略判断落子位置

参数：side, board, currentRound——是否先手，当前棋盘，当前回合数

源代码及下子策略：

1）判断前的准备：

i）获取全棋盘上双方落子点列表，特殊情况直接输出落子结果

ii）扫描双方边界上的棋子，给出列表

```
from random import choice
boa = board.getRaw()
another = {}
available = {}
another[True] = board.getNext(True, currentRound)  # 随机落子点
another[False] = board.getNext(False, currentRound)
available[True] = board.getNone(True) # 允许落子点
available[False] = board.getNone(False)
if not available[True] and not available[False]:  # 整个棋盘已满
    return None
elif not available[not side]: # 对方棋盘已满
    return another[side]
```

2）落子策略所运用到的子函数

（由于代码行数过多，简要总结一下各子函数的作用）

i）isEatable： # 判断是否是可被吃,返回能吃它的子在哪个方向或 False

ii）canDefend： # 判定是否能通过下子防御,返回防御的位置或 False

iii）canAttack： # 判定是否能通过堵路吃敌方棋子（配合 isEatable()使用）

iv）badanother：# 判定在己方落子是否不利

```
border = {True:[], False:[]} # 边界棋子

# 扫描边界
maxValue = 2 # 预设一个阈值，超过才会记入

for i in range(4):
    if boa[i][4][1] == True and boa[i][4][0] > maxValue:
        border[True].append((i,4))
    elif boa[i][4][1] == False and boa[i][3][1] == True \
         and boa[i][3][0] > maxValue:
        border[True].append((i,3))
for i in range(4):
    if boa[i][3][1] == False and boa[i][3][0] > maxValue:
        border[False].append((i,3))
    elif boa[i][3][1] == True and boa[i][4][1] == False \
         and boa[i][4][0] > maxValue:
        border[False].append((i,4))
```

3）落子策略

大致分为四种情况：先手落子，后手落子，接近终局以及在己方随机落子不利。

i）后手落子情况下，考虑进攻以及防御

通过计算得分得到进攻和防御的优先性并得出结论，如果进攻和防御分数都为 0，进入通用策略。

```
# 后手落子策略
if not side:
    attScore = 0 # 进攻分数
    for rival in border[not side]:
        if isEatable(boa, rival, boa[rival[0]][rival[1]][0]):
            point = canAttack(boa,rival) # 进攻位置
            if isinstance(point,tuple) and boa[rival[0]][rival[1]][0] > attScore:
                attPoint = point
                attScore = boa[rival[0]][rival[1]][0]
    defScore = 0 # 防御分数
    for mine in border[side]:
        direction = isEatable(boa, mine, boa[mine[0]][mine[1]][0])
        if direction and boa[mine[0]][mine[1]][0] > defScore:
            point = canDefend(boa,mine,direction)
            if point:
                defPoint = point
                defScore = boa[mine[0]][mine[1]][0]

    if attScore == 0 and defScore == 0:
        # 若对方在下子在我方，且不为吃子或防御，认为对方随机位置对其不利
        lastDec = board.getDecision(not side)
        if len(lastDec) == 2 and boa[lastDec[0]][lastDec[1]][1] == side:
            return another[not side]

    elif attScore > defScore: return attPoint
    elif attScore <= defScore: return defPoint
```

同时在敌方于我方地盘落子的情况下，选择往敌方所放弃的落子位置下子。

ii）先手落子情况下，只考虑防御，如果防御分数为 0，进入通用策略

```
# 先手落子策略
else:
    defScore = 0 # 防御分数
    for mine in border[side]:
        direction = isEatable(boa, mine, boa[mine[0]][mine[1]][0])
        if direction and boa[mine[0]][mine[1]][0] > defScore:
            point = canDefend(boa,mine,direction)
            if point:
                defPoint = point
                defScore = boa[mine[0]][mine[1]][0]
    if defScore > 0: return defPoint
```

iii）近终局情况，在防守无果情况下，进入通用策略 1

```
# 通用策略：近终局时操作
judgeNum = 3 # 参数：空位数
if len(available[True]) <= judgeNum or len(available[False]) <= judgeNum:
    score = 0
    point = None
    for none in available[not side]:
        score_none = badAnother(boa, none)
        if score_none and score_none > score:
            point = none
            score = score_none
    if point: return point

    for none in available[not side]:
        if not isEatable(boa, none, 1): return none
```

iv）通用策略 1 无效进入通用策略 2

```
# 通用策略：己方随机落子不利
if another[side] != () and badAnother(boa,another[side]):
    score = 0
    point = None
    for none in available[not side]:
        score_none = badAnother(boa, none)
        if score_none and score_none > score:
            point = none
            score = score_none
    if point: return point
```

```
if another[side] != (): return another[side]
else: return choice(available[not side])
```

v）极端情况下的保底行为

2.2.5 bestSolution

同时以 getPosition 模拟敌方落子，以 down2thesea 模拟我方落子，通过$\alpha\beta$剪枝进行多层搜索及模拟，寻找出棋盘评分最大化的移动方向

参数：depth, alpha, beta, myTurn, board, currentRound, maxDepth, initsum

依次为：当前搜索深度，两个参数，当前是否我方，当前棋盘，当前回合数，最大深度，搜索前我方棋盘上我方棋子总分值。

i）模拟落子

我方行动时：如果我方是先手并且没达到最大搜索深度，模拟落子，先我方后敌方

```
if myTurn:  # 当前是direction

    if self.isFirst and depth != maxDepth:
        currentRound += 1
        if currentRound <= 499:
            if depth == maxDepth - 1 and self.isFirst==True:
                nextPosition = self.down2theSea(currentRound, board, True)
            else:
                nextPosition = self.getPosition(True, board, currentRound)
            if nextPosition != None:
                board.add(True, nextPosition)
            if depth == maxDepth - 1 and self.isFirst==False:
                nextPosition = self.down2theSea(currentRound, board, False)
            else:
                nextPosition = self.getPosition(False, board, currentRound)
            if nextPosition != None:
                board.add(False, nextPosition)
```

敌方行动时：如果敌方是先手，模拟落子，先敌方后我方

```
else:

    if not self.isFirst:
        currentRound += 1
        if currentRound <= 499:
            if depth == maxDepth - 1:
                nextPosition = self.down2theSea(currentRound, board, True)
            else:
                nextPosition = self.getPosition(True, board, currentRound)
            if nextPosition != None:
                board.add(True, nextPosition)
            if depth == maxDepth - 1:
                nextPosition = self.down2theSea(currentRound, board, False)
            else:
                nextPosition = self.getPosition(False, board, currentRound)
            if nextPosition != None:
                board.add(False, nextPosition)
```

ii）穷举移动方向并进行$\alpha\beta$剪枝

```
bestScore = alpha
bestDirection = -1
movable = False
for direction in [0, 1, 2, 3]:  # 穷举方向
    newBoard = board.copy()
    if newBoard.move(self.isFirst, direction):  # 如果该方向可以移动到棋楼
        movable = True
        if depth == 0:  # 递归结束，直接当前棋盘分数
            result = {'direction': direction, 'score': self.getScore(newBoard, self.isFirst, initsum, maxDepth-depth)}
        else:  # 向下一层递归，深度-1，下一步是模拟对手移动方向，myTurn改为False
            result = self.bestSolution(depth-1, bestScore, beta, False, newBoard, currentRound, maxDepth, initsum)
        if result['score'] > bestScore:  # 决定direction时目标是使分数最大
            bestScore = result['score']
            bestDirection = direction
        if bestScore >= beta:  # 剪枝
            return {'direction': -1, 'score': beta}
```

我方移动时，寻找我方分数最大的移动方向

```
bestScore = beta
bestDirection = -1
movable = False
for direction in [0, 1, 2, 3]:  # 四个方向
    newBoard = board.copy()
    if newBoard.move(not self.isFirst, direction):  # 记录这个方向可以移动，并移动
        movable = True
        result = self.bestSolution(depth, alpha, bestScore, True, newBoard, currentRound, maxDepth, initsum)
        if result['score'] < bestScore:
            bestScore = result['score']
            bestDirection = direction
        if alpha >= bestScore:  # 剪枝
            return {'direction': -1, 'score': alpha}
```

敌方移动时，寻找我方分数最小的移动方向

```
if not movable:
    result = self.bestSolution(depth, alpha, bestScore, True, board, currentRound, maxDepth, initsum)
    if result['score'] < bestScore:
        bestScore = result['score']
```

某一方卡死时不进行移动

2.2.6 output

作用：输出移动方向或者落子位置

参数： currentRound, board, mode——当前回合数，棋盘，模式

源代码及策略：

通过 down2thesea 选择下子位置

100 回合之前被吃子惩罚为 100，100 回合以后取消被吃惩罚

时间剩余较少时只搜两层

```
if mode == 'position':  # 给出己方下棋的位置

    return self.down2theSea(currentRound, board, self.isFirst)

elif mode == 'direction':  # 给出己方合并的方向
    initsum=0
    if currentRound>100:
        self.p=0
    for i in board.getScore(self.isFirst):
        initsum += 2**i
    if 5 - board.getTime(self.isFirst) > 4.9/500*currentRound:
        result = self.bestSolution(1, -114514, 114514, True, board, currentRound, 1, initsum)
    else:
        result = self.bestSolution(2, -114514, 114514, True, board, currentRound, 2, initsum)
    if result['direction'] == -1:
        result = self.bestSolution(0, -114514, 114514, True, board, currentRound, 0, initsum)
    return result['direction']

else:
    return
```

时间剩余较多时搜三层

2.3.程序限制

非常幸运的是我们的程序并没有 bug。

3. 实验结果

3.1.测试数据

实验环境说明：

- 硬件配置：2.3 GHz Intel Core i5；8 GB 2133 MHz LPDDR3
- 操作系统：macOS 10.14.6
- Python 版本：3.1

小组采用了内部版本直接互相对战以及在天梯上自由对战的方式进行测试，为了缓解测试压力，分配给小组各个成员一定的测试任务。测试数据整合如下

3.2.结果分析

在算法设计的早期，即 4.0 版本之前，经过对测试结果的分析，发现本算法落败的原因基本都是因为被敌方塞爆，因此增加了空格数的重要性权重，同时效仿敌方开辟

sierra内战以及天梯数据（计算左侧版本的胜率）

版本号	1.0	2.0	3.0	4.0	4.3	5.1	5.2	5.3	omega	天梯胜率
1.0										35%
2.0	90%									42%
3.0	90%	62.5%								38%
4.0	80%	80.3%	61.2%							52%
4.3	X	X	75.2%	54.0%						29%
5.1	X	X	80.0%	66.7%	85.0%					47%
5.2	X	X	74.5%	70.0%	82%	80.0%				58%
5.3	95%	90%	90%	75%	72%	63%	64%			65%
omega	X	X	X	X	X	X	X	60%		X
F-Sierra	95%	95%	85%	80.0%	75%	70%	70%	60%	57.2%	51.5%

了攻击性落子的策略，并小组讨论得出了防御性落子方法。

中后期，即 5 代版本时，我们发现 5.1 和 5.2 算法在天梯上的表现出现了两极分化，5.1 在比赛中获胜基本是因为前期优势累计，而 5.2 获胜则是由于苟活能力强以及能在后期对敌方大子进攻，因此我们整合两版代码的优势方面，编写了 5.3，在天梯上胜率一度很高。

在正式天梯赛开始后，我们对 5.3 版本的弱点特意编写了 omega 程序，用来调整参数，并通过在天梯上的表现，最终完成了 F-Sierra 的最终参赛函数。虽然最后参赛的优势程序大部分都具有极强的进攻性，但是我们小组的参赛函数却是以防守性为主，也算是另辟蹊径吧。最终在 8-4 时遇到了 yyds 代码 Uniform，无奈落败，但是相较天梯赛结果的第 3 与第 4 我们仍有一战之力，实属可惜。

另外值得关注的一点就是我们算法对时间的利用不够，在比赛中总是不能够用满 3s 以上的时间，这使得我们的 AI 思考的内容相较他人较少，也是弱势的一个原因吧。

4. 实习过程总结

4.1. 分工与合作

陈可人*：负责整体调度以及组织会议，参与棋盘评分以及落子策略部分的讨论及编写

袁子珉 ：负责代码主体编写，并提供策略接口以及迭代搜索

虞晨煦：主要参与棋盘评分部分的编写以及代码间对战测试并分析结果

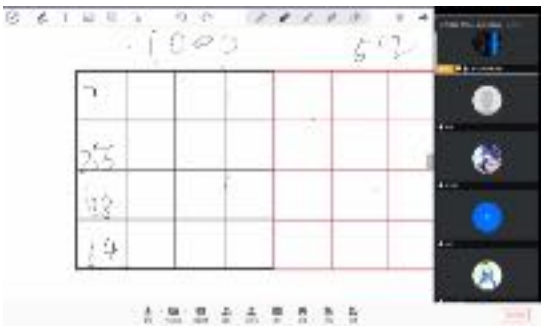
李卓逸：落子策略部分代码编写以及构思防御攻击思路

陈浩良：参与各部分代码讨论，编写实习报告，记录数据

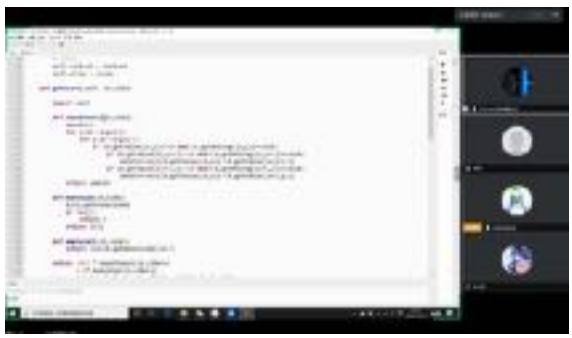
组内交流大体通过微信群进行，在微信群上进行中小问题的讨论以及分享代码共同编写。

在商讨核心算法时召开腾讯会议一同讨论，并由组长通过屏幕共享的方式交流战略。

会议截图



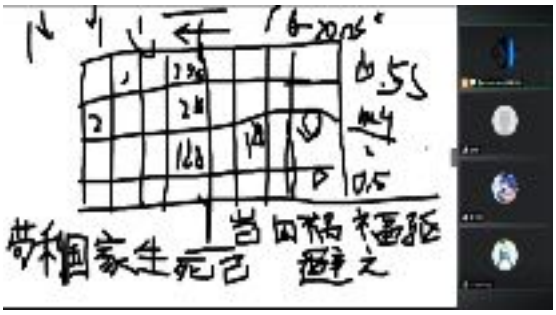
初次会议，sierra-1.0 的诞生



艰难的 debug 历程

放子算法的实现

放子算法的改进，从 3.0 到 4.0



5.0——sierra 之集大成者

4.2. 经验与教训

在本次实习过程中，值得得意的是我们小组的成员们配合良好，每个人都积极地在群里讨论，并揽下了自己的工作，这与之前笔者所经历的小组合作有天壤之别（这次实习里最划水的就是写报告的我了），这一点上组长功不可没，在初次会议时就分配了大家的工作，使实习有条不紊的进行。

教训是我们组对时间利用的还不够充分，算法浪费了可用时间，这一点希望以后能尽量用满给定的时间。归根结底还是算法有值得改进之处。

4.3. 建议与设想

对于这次数算课的大作业，我们组有如下建议:

1. 这次实习作业的主题具有一定的随机性，在代码强度相近的情况下，游戏的胜负主要依赖于随机队列，但是在很多比赛中出现了随机队列在比赛前中期就将某一方塞爆的情况，希望以后的作业中如果有这种随机数列的存在，能够考虑排除极差的随机性的存在，使比赛竞争更稳定（但是就没有节目效果了）
2. 在最终比赛的流程中，我们在赛制方面存在疑问。在八进四的比赛里由于直接遇到了血虐我们的 Uniform，直接被淘汰出局，其实遇到强手这对任何一个有机会争夺第三第四的队伍来说都不太公平，因此希望将来的赛制能够进行小组赛而不是淘汰赛（个人建议）

对选修明年课程的学弟学妹们，我们想表达的是：在陈老师的数算课上你们能学到的东西（juan，就硬 juan）比课程要求的会多得多，并且你会有很多的机会发挥自己的创意，并学到崭新的知识。当然，最重要的是，这门课的环境氛围很好，微信群里总是有大佬可以解决你的问题，老师非常帅，小黑很可爱，助教人很好，听我的，认真卷（bushi）就完事了。

5. 致谢

感谢陈斌老师在本学期中对我们尽心尽力的教导。

感谢这门课的助教老师们认真负责地批改我们 bug 遍是的作业，并在群里解答我们千奇百怪的问题。

感谢技术组的成员为我们带来了如此有趣的大作业。

感谢每一个组员在本次实习过程中的付出。

6. 参考文献

[1]http://gis4g.pku.edu.cn/ai_arena/game/4/
[2]https://blog.csdn.net/qq_32767041/article/details/80634244#53-己方格局评价函数实现
[3]https://gitee.com/chbpku/sessdsa.2048

数据结构与算法课程实习作业报告

马千一* 马宇琦 尹擎 刘嘉 张驰

【摘　要】本算法总体采用了带有α-β剪枝的 min-max 决策树算法。算法中构建了一棵先后手轮流行动的 min-max 决策树进行决策模拟，在模拟到游戏终局或最大深度时调用评价函数对局势进行评价，在向上返回时使用α-β剪枝减小运算量（实际上与α-β剪枝并不完全一样，有改进）。

在落子阶段由于遍历搜索不现实，且可能的策略并不多，所以主要使用启发式算法。本代码在天梯上可与 botS 战成平手，排名在 F18 赛区 9 到 15 名之间浮动，最终为 15 名未出线。

【关键词】min-max 决策树；α-β剪枝；启发式算法；MCTS

1 算法思想.....1
1.1 总体思路.....1
1.2 算法流程图.....5
1.3 算法运行时间复杂度分析.....6
2 程序代码说明.....7
2.1 数据结构说明.....7
2.2 函数说明.....7
2.3 程序限制.....13
3 实验结果.....14
3.1 测试数据.....14
3.2 结果分析.....14
3.3 经典战局（可选）.....15
4 实习过程总结.....17
4.1 分工与合作.....17
4.2 经验与教训.....20
4.3 建议与设想.....20
5 致谢.....20
6 参考文献.....21

1 算法思想

1.1 总体思路

1.1.1 带有α-β剪枝的 min-max 决策树

我们的算法总体采用了带有α-β剪枝的 min-max 决策树算法。此算法的基本思想如下：

(1) 构建一棵先后手轮流行动的 min-max 决策树

图 1.1.1-1 是一棵以己方先手为例的决策树。此树中，每一层代表某一方进行一次决策，该层的所有节点代表该方所有可能的决策情形。我们一致认为，“落子位置”的决策在对局中不起主要作用。虽然落子有很多的可能性，但落子决策的主要部分——决定落在己方还是

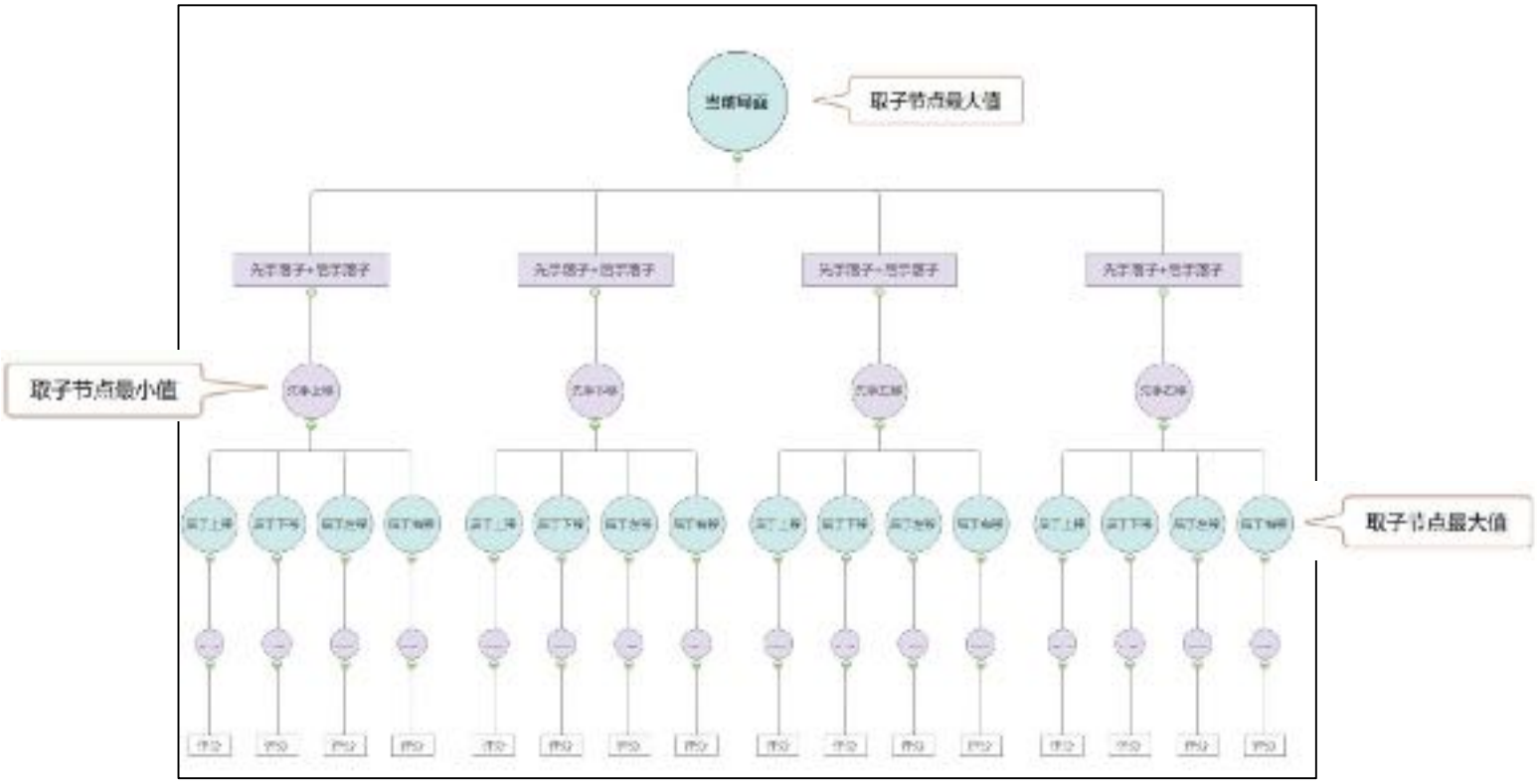


图 1.1.1-1-1 min-max 决策树

对方，可以用一个启发式算法描述。又考虑到对所有落在对方的情形进行列举需要大量时间花费，所以我们不进行列举，而是将双方的落子与先手方的移动合并成同一个操作以减少层数。在图中，一次颜色变更即为我们定义的一次“操作”。

(2) 对局面进行评分

当模拟到游戏终局或到达算法设计的最大深度时，停止继续深入，对当前局面使用一个局势评价函数进行评分（分数越大，对己方越有利）。再由叶节点开始向上回溯，由于假想对手能做出最聪明的决策，故在每一个己方决策层的节点都取子节点（即对方决策）的评分最小值，而在每一个对方决策层的节点都取子节点（即己方决策）的评分最大值。在对每个节点的子节点评价时，采用了列表的数据结构存储各个方向的评分。

(3) α-β剪枝

α剪枝如图 1.1-2，在从我方决策层向下搜索的过程中，将“目前搜索得到的评分最大值”设为α，传递进搜索函数中。作为例子，图 1.1-2 中的α为 100。在下一层的搜索中，若遇见有子节点的评分大于α，则说明我方决策层该节点的评分也必定大于α，即不会被上一层采用。因此可直接停止搜索，剪去红色框内的所有节点。β剪枝用于对方决策层的向下搜索，与α剪枝完全类似。因此，在具体程序中我们将二者合并以简化代码，评价值也因此采用了每回溯一层取一次相反数再取最大值的操作，具体可见 2.2 节。

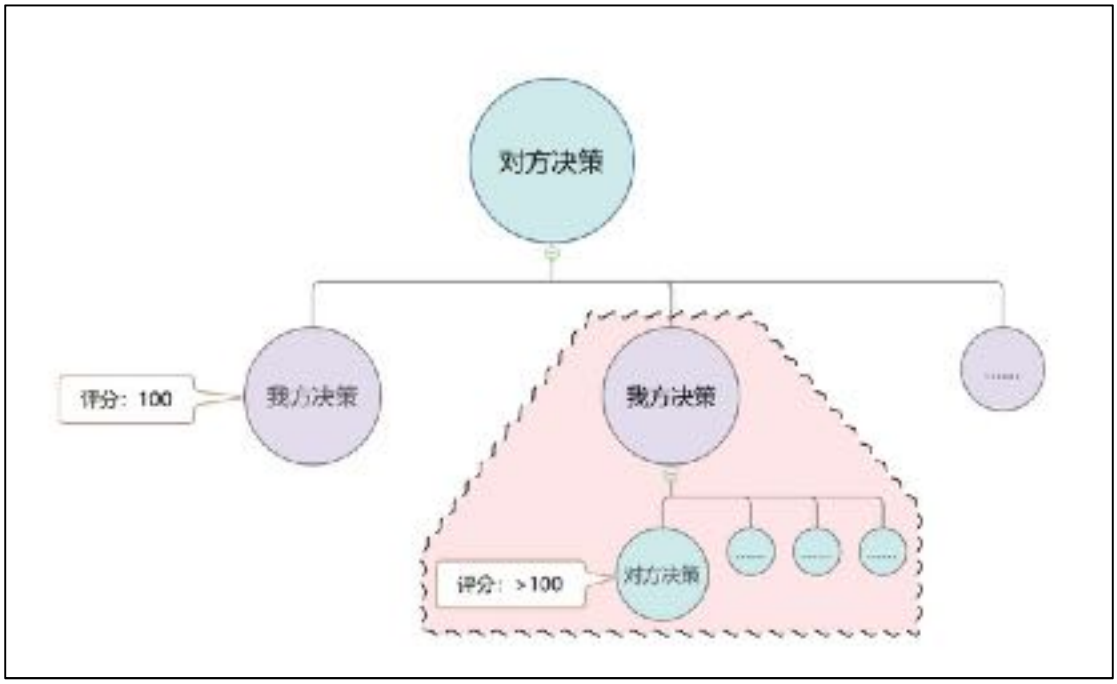


图 1.1.1-1-2 α剪枝

1.1.2 启发式的落子算法

在开发落子算法时，我们做了如下考虑：从盘面棋子总数值来看，一般来说每次将棋子落在己方总是一个好的选择，只有特殊情况下才考虑下在对方。特殊情况包括：己方无处可下、下在对方可以保护己方边界线上的大子、下在己方会干扰己方棋子的合并。经过优先级排序，我们确定了如下决策顺序：

(1) 只能下在己方时，下在己方；
(2) 保护己方大子；
(3) 下在己方不造成不利时，下在己方；
(4) 其余情况下在对方；

我们在树中对两方都使用了这个落子算法进行模拟。

1.1.3 算法的确立过程

开发工作开始时，出于试验和探索的目的，我们小组的成员分成三组，分别尝试着进行了纯启发式算法、min-max 决策树、蒙特卡罗树搜索（MCTS）的开发，开发出了几个可用的算法。我们在几天后的组会中进行了交流与算法之间的比较，经过权衡，最终选择了 min-max 决策树作为主要开发方向，其他算法也同时继续改进。

为说明算法具体的改进过程，直接将我们的各代码版本列举如下（其余失败的调参不列）：

代码名称	发布时间	发布者	改进点
terrible_I	5 月 9 日	马千一	极为简单的价值函数，无决策树，盲目合成大子
trial_1	5 月 16 日	马宇琦	贪婪吃子，优先苟
trial_2	5 月 17 日	马宇琦	尝试深搜，不告而终
trial_3	5 月 19 日	马宇琦	改进评价函数，包括紧密程度、贴边度、数值等多参数

terrible_II	5 月 20 日	马千一	带剪枝的 minmax 决策树，价值函数仍为盲目合成大子
babanbo	5 月 21 日	尹擎	进攻型启发式算法，加入了在先手时先塞一个 2 然后吃掉以迅速占地的申必小策略
Babanboo	5 月 23 日	尹擎	加入了危险判断和判断是否撤退的算法，不再只会硬冲
babanbooo	5 月 25 日	尹擎	将整个局面分成了三个阶段，前期苟，中期进攻堵塞对方，后期维持局势
terrible_VI	5 月 27 日	马千一	改进了价值函数，同时考虑合成、吃子等。加入了保护己方大子的落子决策。
terrible_VII	5 月 27 日	马千一	尝试加深搜索深度，无果
mcts_529	5 月 29 日	张驰（开发） 马宇琦（debug）	最终的 MCTS 算法，经过了十分不容易的开发后的产物。能吊打之前版本的 min-max 树，可惜打不过当时的。
terrible_X	5 月 30 日	马千一	尝试加入占地、围堵的决策，但是失败了，程序不能完全按照我们所期待的样子运行，战力不够。
terrible_XI	5 月 30 日	马千一	在修改 X 时对搜索树的部分做了一些改动，意外得到了一个战力提升很多的 AI，但是不是非常清楚它的运行机制。这一版本里，价值函数也强调了中间几格的重要性。我们用它更新了天梯上进行小组赛的代码，排位大幅上升。
terrible_ba	5 月 31 日	尹擎	尝试在决策树中遇到“吃子”情形时忽略层数上限限制，直至相互吃子停止，希望增强算法的对抗能力。然而实战效果不佳。
babanboooo	5 月 31 日	尹擎	最终的启发式算法，加入了预测，优化了前期苟法、危险判断、风险防范、合成堵塞，但是仍然打不过当时的 min-max 树。
terrible_VIIp	5 月 31 日	刘嘉	加入了集群函数，使得相同的棋子紧挨着能加分
terrible_XILp	6 月 1 日	刘嘉	刘嘉通宵在本地做实验调参的结果，战力稍有上升。是我们小组最终参赛代码。个别时候会超时，所以在临近提交截止时，我们小改了一下小组天梯上代码的时间管理部分。
terrible_VIFin	6 月 3 日	大家	6 月 3 日 赛后，我们痛定思痛，认为上传代码风格混乱且我们也不是很理解其机制的 XI 是不正确的行为。于是马千一决定去繁从简，在整理最终版代码时又参照了一下代码风格良好的 VI，对代码的落子部分进行了一次小改动，又有张驰、尹擎进一步完善代码风格，做出了最终用于撰写报告的代码版本。另外，这一版本的代码可以稳定地击败 VI 与 XILp。

1.2 算法流程图

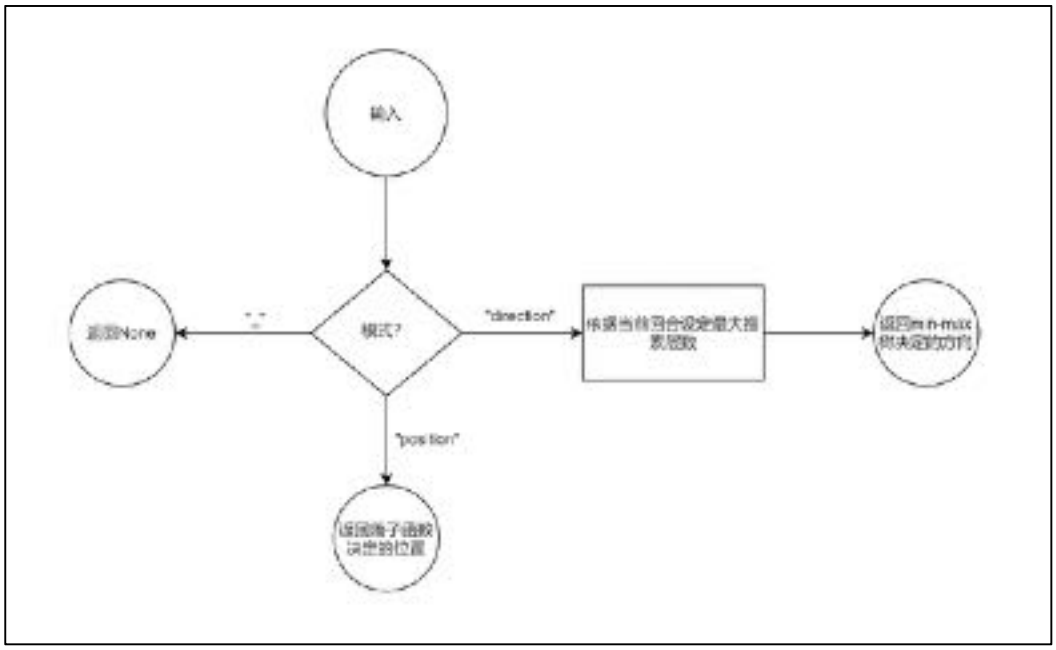


图 1.2-1 主流程图

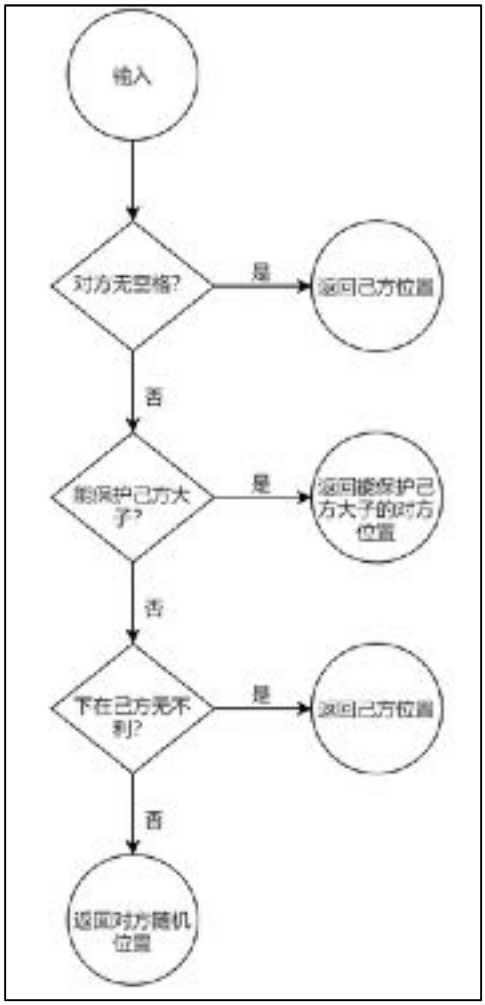


图 1.2-2 落子算法流程图

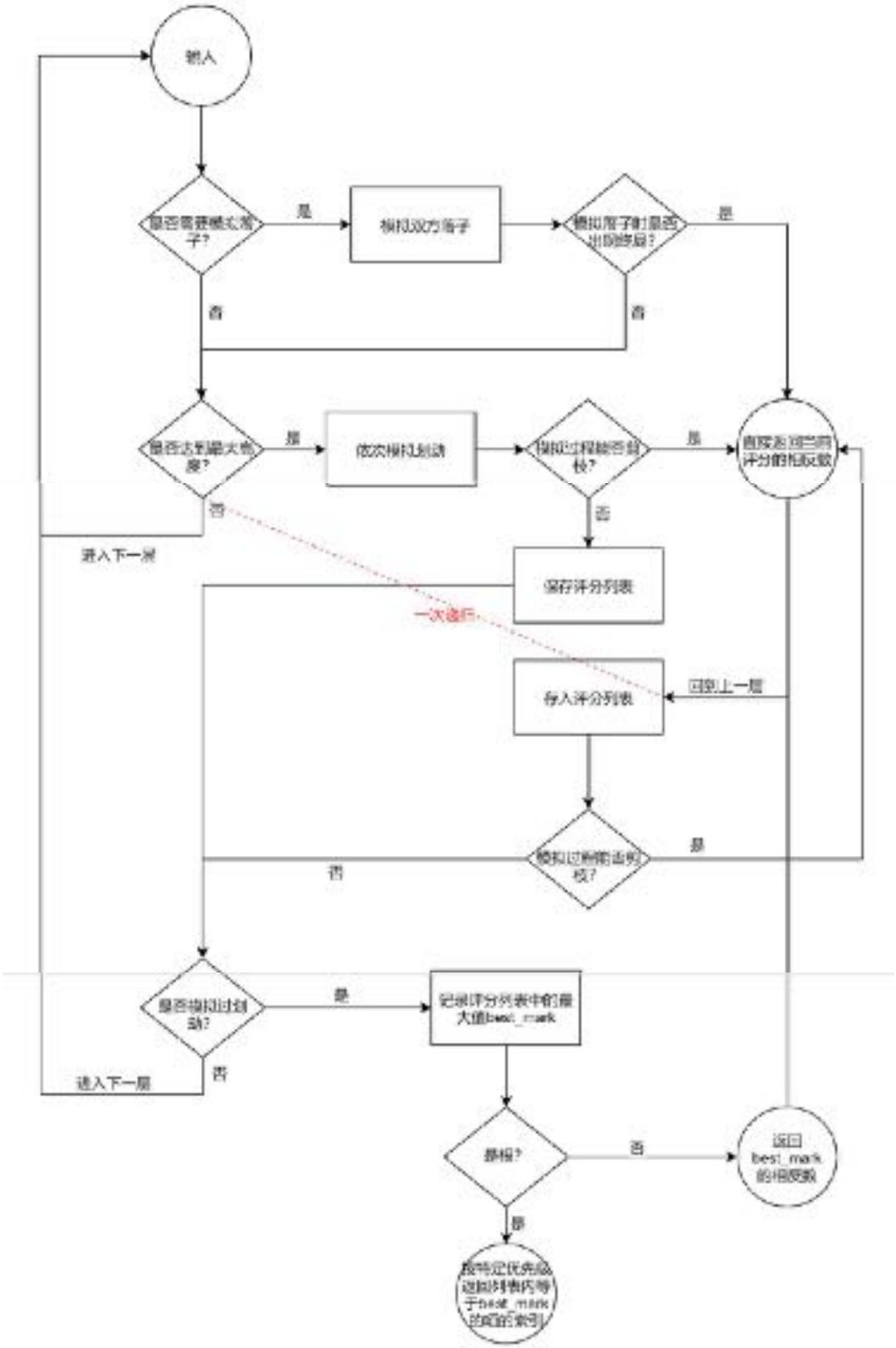


图 1.2-3 min-max 决策树流程图

1.3 算法运行时间复杂度分析

该算法的运行时间主要花费在展开树时的盘面操作与局面评价。在每一个节点，这样的时间花费是大致相同的，所以该算法的时间复杂度为树算法的 $O(2^n)$级别。剪枝可以减少算法的时间复杂度，然而剪枝的效率依局面不同产生较大差别，难以下定论。

在实际运行中，由于对最大层数进行了限制，我们的代码很少发生超时的情况。

2 程序代码说明

2.1 数据结构说明

我们算法的核心部分，其数据结构是树，即 min-max 决策树，同时还加入了合理的剪枝。但是我们并没有定义节点类和决策树类，而是使用了一个递归函数来实现决策树的功能。另外，在存储各种可能决策的评价值时，我们使用了线性表；为了加快程序运行的速度，我们用字典存储有关棋局的实时信息。

2.2 函数说明

这一版本的程序是在天梯赛截止后（惨遭淘汰后）的最终修改版，即 terrible_VIFin，有比较完备的注释。下面就给出一些主要函数的代码截图，并简要介绍它们的功能、采用的策略等。完整的代码详见压缩包内的 PY 文件。

首先是__init__()，在玩家类被创建时，我们为它增加了很多属性，这是为了方便在以后进行调用，减少函数间传递的各种参数。属性包括递归的深度（树的高度）、目前的落子/划动玩家、当前棋盘、存储局面信息的字典等。

```
1.  class Player:
2.      def __init__(self, isFirst, array):
3.          # 初始化 这些量一直存在类里面，这样在各种函数调用时不需要传太多参数
4.          self.isFirst = isFirst
5.          self.array = array
6.          # 一个方便移动位置的元组(见 protect 函数等)
7.          self.add_tuple = ((1, 0), (-1, 0), (0, 1), (0, -1))
8.          # 一个表示方位的字典
9.          self.dir_Dict = {'l': 2, 'u': 0, 'd': 1, 'r': 3}
10.
11.         # 下面是四个可以使程序加速的字典（注意这些所谓的“对方”“己方”在决策树发展的过程中是会变的）
12.         self.p_to_v_NowOther = {}  # 在当前棋盘上，按位置查找对方子的大小
13.         self.v_to_p_NowOther = {}  # 在当前棋盘上，按子的大小查找对方棋子的位置
14.         self.p_to_v_NowSelf = {}  # 己方的位置-棋子大小字典
15.         self.v_to_p_NowSelf = {}  # 己方的棋子大小-位置字典
16.
17.         self.nowSide = self.isFirst  # 用于在模拟过程中切换先后手
18.         self.nowBoard = None  # 当前棋盘（注意实时更新和保存！）
19.         self.num_NowSelf = 0  # 己方棋子个数（不算空格）
20.         self.num_NowOther = 0  # 对方棋子个数（不算空格）
```

```
        self.Biggest_NowOther = 0  # 对方最大的棋子
        self.Biggest_NowSelf = 0  # 己方最大的棋子

        # height 系列变量，表示搜索树的高度
        self.Height0 = 0  # 初始高度
        self.maxHeight = 0  # 最大高度
        self.height = 0  # 当前高度
        self.minmin = -2**50  # 一个有时会用到的最小值
```

然后是被对战平台调用的 output()，主要是对不同的调用情况作简单的分类，在需要给出划动策略时，会使用决策树，所以在这里设置了树的高度。节约篇幅，不做截图。

see_board()函数，是为了把目前的棋局信息使用字典记录下来，加快速度也便于访问。节约篇幅，不做截图。

give_position()函数，可以给出需要落子的位置。不仅仅在做出落子决策时需要调用它，在选择滑动方向时，决策树也会调用它来模拟双方玩家的落子。该函数会进一步调用两个具体实现落子决策的函数，其中 protect()用来判断可否保护己方的大子，check_self()用来检查是否要在己方领地内落子。为了节约篇幅，give_position()函数在这里不给出。

```
    def protect(self):
        """用以保护己方大子"""
        if not self.nowBoard.getNone(not self.nowSide):  # 对方无空格
            return

        currentLv = self.Biggest_NowSelf  # 正在检查的己方棋子的级别
        while currentLv > 3:
            # 字典：所有该级别的我方棋子，pieces：当前正在检查的我方棋子
            for pieces in self.v_to_p_NowSelf[currentLv]:
                checking = pieces  # checking：目前在检查的格子
                for each_direction in self.add_tuple:  # 观察各个方向
                    # 把眼光从这个格子往考察的方向上移动一格
                    checking = (pieces[0] + each_direction[0],
                                pieces[1] + each_direction[1])
                    # pieces[0]:横坐标,pieces[1]:纵坐标
                    while checking in self.p_to_v_NowOther:  # 当发现这里是对方领地
                        if self.p_to_v_NowOther[checking] == 0:
                            # 如果是对方的空格
                            checking = (checking[0] + each_direction[0],
                                        checking[1] + each_direction[1])
                            # 那就再往这个方向移动一格看看
                        elif self.p_to_v_NowOther[checking] == currentLv:
                            # 如果是对方的同级别棋子（有危险！）
                            protect_position = (checking[0] -
                                                each_direction[0],
                                                checking[1] -
                                                each_direction[1])
```

```
                                # 从该位置，往来的方向移动一格，就是我们的保护位置
                                if protect_position in self.p_to_v_NowOther:
                                    # 如果保护位置是对方的领地（则必为空格）
                                    return protect_position  # 返回保护位置（将会在此落子）
                                else:
                                    # 如果不是，说明我方棋子与对方同级别棋子紧邻，已经无法保护，结束搜索该方向
                                    break
                            else:
                                break  # 如果是对方领地却既不是空格也不是同级别棋子，则构不成威胁，结束搜索该方向
                    currentLv -= 1  # 遍历了这一级别的所有棋子后，进入较低的下一个级别

            # 全走了一遍还没有从 return protect_position 那里返回出来（即未找到保护位置），则返回 None
            return

        def check_self(self, position):
            """检查在己方落子是否有利"""
            if not position:  # 无法在己方落子的情况
                return False

            if self.num_NowSelf < 15:  # 己方棋子个数低于 15
                return True  # 可以放 2

            # 防止放入 2 会卡住自己
            for each_direction in self.add_tuple:  # add_tuple 里包含分别向上下左右移动一格的操作
                check = (position[0] + each_direction[0],
                         position[1] + each_direction[1])  # 检查 2 周围的格子
                if check in self.p_to_v_NowSelf and self.p_to_v_NowSelf[check]:
                    # 检查的格子属于己方且等于 2       self.p_to_v_NowSelf[check] == 1
                    return True  # 可以放 2
            return False  # 否则不放 2
```

下面就是核心算法 start_tree()了。这是一个递归函数，完成决策树的构建和剪枝工作。我们把先手划动的模拟与后手划动的模拟区分开来，在先手划动前加上对双方落子的模拟，这样子一个回合的游戏就对应于决策树的两个单位高度。另外，我们具体的实现方式与传统的 minmax 决策树不同，而是把操控权交替地在先手和后手玩家之间转移，这样一来 α 剪枝与 β 剪枝就变为了同一种情况。需要注意的是，在向父节点传递决策估值时，要加上一个负号。当然，棋盘的存储是重要工作，必须在每次模拟前执行存储。在我们所定义的 height 意义下，本算法能搜索 0-3 层的 height，在对战中则为两个完整的回合（全局共 500 回合）。

```
    # 一个单位的 height 是如下定义的：
```

```
    # 或者是先手落子+后手落子+先手划动，或者是仅有后手划动
    # 模拟的时候，对 height 分奇偶数讨论
    # start_tree()是递归的
    def start_tree(self, currentRound, alpha):  # alpha 是用来剪枝的，从上一层传下来
        # 如果是奇数，要在模拟划动前进行模拟落子，不过如果现在是第一次模拟，就没有落子过程
        if self.height % 2 == 1 and self.height != self.Height0:
            cannot_add = True  # 是否双方都死棋
            if self.nowBoard.getNext(self.nowSide, currentRound) \
                    or self.nowBoard.getNone(not self.nowSide):
                # 己方有子可下或者对方有空位
                p1 = self.give_position(currentRound, self.nowSide)  # 己方落子
                self.nowBoard.add(self.nowSide, p1)  # 局面改变
                cannot_add = False
            self.nowSide = not self.nowSide  # 模拟对方落子

            if self.nowBoard.getNext(self.nowSide, currentRound) \
                    or self.nowBoard.getNone(not self.nowSide):
                p2 = self.give_position(currentRound, self.nowSide)
                self.nowBoard.add(self.nowSide, p2)
                cannot_add = False
            self.nowSide = not self.nowSide  # 回到己方

            if cannot_add:  # 这种情况不会出现在树根上（树根的话不会运行以上这一大段）
                return -self.get_mark()  # 双方都不能落子了，直接返回终局评分

        save_board = self.nowBoard.copy()  # 模拟后的局面保存
        evaluation_list = [self.minmin] * 4  # 估值列表（分别是各方向移动后的估值）
        now_best = self.minmin  # 用于剪枝
        Nothing = True  # 记录是否不存在有效滑动

        # 剪枝决策树的搜索顺序
        # 经过试验，各个顺序的剪枝所花时间差别不大。这个方向顺序进攻性较强，往往稍快一些
        if self.nowSide:
            search_order = 'rudl'
        else:
            search_order = 'ludr'

        if self.height == self.maxHeight:
            # 决策树达到最大高度，只有一方行动，直接返回决策
            for letter in search_order:
                directions = self.dir_Dict[letter]  # 各个方向
                if self.nowBoard.move(self.nowSide, directions):  # 模拟每个方向的移
    动
                    this_choice = self.get_mark()  # 该滑动后的估值
```

```
                    Nothing = False
                    if this_choice >= alpha:  # 判断是否可以剪枝
                        return -this_choice  # 剪枝
                    evaluation_list[directions] = this_choice  # 不剪枝的话，记下评分
                    # 非常重要：每次划动都要用上一次未划动前存下的棋盘
                    self.nowBoard = save_board.copy()

        else:  # 未达到最大高度
            for letter in search_order:
                directions = self.dir_Dict[letter]
                if self.nowBoard.move(self.nowSide, directions):
                    self.nowSide = not self.nowSide  # 需双方行动，所以换另一方
                    self.height += 1  # 当前高度+1

                    # 获得该方向的估值，注意根据先手/后手，考虑轮数参量是否加一，还要传入本轮的目前最佳决策
                    i = 0 if self.height % 2 == 0 else 1
                    this_choice = self.start_tree(currentRound + i, -now_best)

                    Nothing = False
                    self.height -= 1
                    self.nowSide = not self.nowSide  # 回到目前棋手
                    if this_choice >= alpha:  # 考虑是否剪枝
                        return -this_choice
                    evaluation_list[directions] = this_choice  # 记下估值
                    self.nowBoard = save_board.copy()  # 回到移动前的棋盘
                    now_best = max(evaluation_list)  # 更新目前最佳决策

        if not Nothing:
            # 如果模拟成功
            best_mark = max(evaluation_list)  # 挑选最佳评估值
        else:
            # 无可滑动方向，接下来将操作权拱手相让，不过这种情况不会出现在树根上
            if self.height == self.maxHeight:
                # 如果已经达到最大高度，将当前局面评分返回
                return self.get_mark()
            else:
                # 若未达到最大高度
                self.nowSide = not self.nowSide  # 换另一方行动
                self.height += 1  # 当前高度+1
                # 获得该方向的估值，注意根据先手/后手，考虑轮数参量是否加一，还要传入本轮的目前最佳决策
                if self.height % 2 == 0:
                    this_value = self.start_tree(currentRound, -now_best)
```

```
                    # 此时-now_best 是 2 的 50 次方，所以其子节点不可能进行剪枝
                else:
                    this_value = self.start_tree(currentRound + 1, -now_best)
                self.height -= 1
                self.nowSide = not self.nowSide  # 回到目前棋手
                return -this_value  # 返回对方操作后的最高分（负号由切换棋手导致）

        if self.height == self.Height0:  # 是树根
            # 根据先后手，决定在相同的评分时，各个方向不同的录用顺序
            if self.nowSide:
                order = 'rudl'  # 由于相同评分的情况并不是很常见，顺序影响不大
            else:
                order = 'ludr'

            for i in order:
                # 事先约定的字典为程序编写带来方便，也增强可读性
                if best_mark == evaluation_list[self.dir_Dict[i]]:
                    # self.dir_Dict = {'l': 2, 'u': 0, 'd': 1, 'r': 3}
                    return self.dir_Dict[i]
        else:
            return -best_mark  # 返回最高价值（对对方最不利）
```

最后就是我们的评价函数和一个简单的 inboard()（用来检查一个坐标是否位于棋盘内，不截图了）。关于评价函数，我们的评价方案是分别为每一方的棋子给出评价值，然后把两方的得分相减。其中估值函数中的参数是通过前期的人为设置和后期的进一步调节得来的（未能学会自动调参，汗颜），但我们没有过多占用天梯资源，都是本地调参。

```
    def evaluate(self, scoreList):
        """评价一个棋子列表的价值"""
        # 承认一下错误，我们未学习自动调参的方法，参数都是自己想一个大概，然后通过本地测试调节的
        # 不过我们没有大规模占用过天梯的资源。一直是在本地调好后，放到天梯上打一些匹配赛看看效果
        # 一般在天梯赛上看到的效果与预期基本一致，所以就不使用天梯来调参了

        # value1 控制着己方对合成大子的重视程度
        value1 = 0
        for num in scoreList:
            value1 += (3**num)

        # value2 控制着对于吞食对方棋子的重视程度
        value2 = 0
        for pieces in scoreList:
            value2 += (2**pieces) * 50
            # 仅在自己这里合成大子，它是不会增大的，要吞食对方才能增大。
```

```
302.
303.         # value3 是一个防止深入敌方的第二列的评价值。
304.         # 当你尝试深入敌方时，value3 会给出一个负数。因为根据观察，大多数深入敌方的棋子最终会
     被吞食
305.         value3 = 0
306.         if self.nowSide:
307.             # 处于左方的情况
308.             for positions in [(0, 5), (1, 5), (2, 5), (3, 5), (0, 6), (1, 6),
309.                               (2, 6), (3, 6), (0, 7), (1, 7), (2, 7), (3, 7)]:
310.                 if self.nowBoard.getBelong(positions) == self.nowSide:
311.                     value3 += -(2**self.nowBoard.getValue(positions)) * 100
312.                     # 深入敌方的子越大，给出负值的绝对值也就越大
313.         else:
314.             # 处于右方的情况
315.             for positions in [(0, 2), (1, 2), (2, 2), (3, 2), (0, 1), (1, 1),
316.                               (2, 1), (3, 1), (0, 0), (1, 0), (2, 0), (3, 0)]:
317.                 if self.nowBoard.getBelong(positions) == self.nowSide:
318.                     value3 += -(2**self.nowBoard.getValue(positions)) * 100
319.
320.         value_total = value1 + value2 + value3  # 把三种价值相加
321.         return value_total
322.
323.     def get_mark(self):
324.         """评价一个局势，即比较谁的价值大，且大得更多"""
325.         eva_self = self.evaluate(self.nowBoard.getScore(self.nowSide))
326.         eva_enemy = self.evaluate(self.nowBoard.getScore(not self.nowSide))
327.
328.         # 将双方的价值函数相减
329.         return eva_self - eva_enemy
```

2.3 程序限制

最终参加天梯赛版本的代码 terrible_XILp 由于尽可能地用满了运行时间，在极个别情况下有超时现象（我们做了时间管理，但是仍有可能在最后几回合的某一次递归内超时）。而在报告中给出的这个版本经过大量的本地测试，从未超时。

在历次本地测试和天梯赛中，我们的程序从未报错（不管是哪一个版本）。而通过对程序本身的仔细分析，我们发现了这样一种极端情况：如果在决策树的模拟过程中，有一方的棋子被对方全部吃光，那么决策树将停止这一局面的继续演化并返回此刻的局势评分。这对于模拟来说自然是不够正确的，但是倒也不至于报错。实际的测试中，这种极端情况并未出现过。

3 实验结果

3.1 测试数据

实验环境说明：

- 硬件配置：（CPU/内存）
  - CPU: Intel(R) Core(TM) i7-9750H
  - 内存: 16GB DDR4 2666MHz
- 操作系统：（名称/版本）
  - Windows 10 Enterprise LTSC 1809
  - Ubuntu 18.04 LTS
- Python 版本：（版本号）
  - Python 3.8.1
- 代码托管服务
  - Coding 免费版 + git
- 其他
  - 单线程运行

本地使用了 Visual Studio Code 或 PyCharm 进行开发，调试工具使用了官方给出的 userinterface.py 和 round_match.py。本地测试共 50 轮对战（天梯 bot 对战为 6 轮）。和天梯上其他选手比赛的参考性不强，故不予列入。在改进时，主要通过加入新的判定和改进局面评价函数以实现棋力的增长。调试参数的过程基本是在本地通过与旧版代码对战的方式实现的。从 terrible_I 到 II，再到 VI 等，每个版本都有着巨大的飞跃。

玩家1	玩家2	比分	玩家1	玩家2	比分
terrible_II	terrible_I	50：0	terrible_XILp	botSSS	1：5
terrible_II	mcts_529	28：22	terrible_XILp	botSS	2：4
terrible_VI	terrible_II	48：2	terrible_XILp	botS	3：3
terrible_VI	mcts_529	49：1	terrible_XILp	botA	5：1
terrible_XILp	terrible_VI	玄学	terrible_XILp	botB	6：0
terrible_VIFin	terrible_XILp	28：22	terrible_XILp	botC	6：0
terrible_VIFin	terrible_VI	33：17	terrible_XILp	botD	6：0
terrible_XILp为比赛版本，terrible_VIFin为报告版本					

3.2 结果分析

3.2.1 效果

在实际比赛中，我们所设想的规则如保护己方大子、侧重深度优先、防止己方棋子过度移动导致无法返回等情况大都成功实现。然而，程序并不稳定。在某些情况下，规则并不能很好地约束程序的行为。这可以表现为无意义的白给大子的行为，或者错误的，使得棋盘混

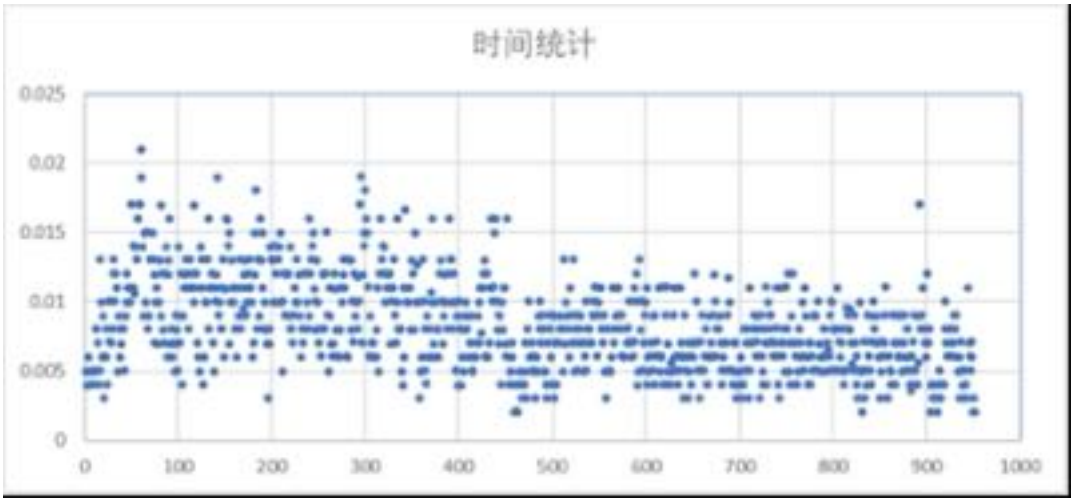
乱的合并方向，这常常将己方棋盘堵死。

通常情况下，当 terrible 面对其他敌人时，先手更加有利，这也符合游戏本身的特点。然而，在 terrible 系列左右互搏时，反而是后手占有更大的优势，后手胜率显著高于先手胜率。这可能是因为后手需要比先手多搜索一次。

3.2.2 时间开销

由于落子时选择的特殊性，落子函数并未使用剪枝树给出决定，而是使用了简单策略下的启发式搜索，因而耗时极短，500 轮总计耗时小于 0.05s，故可以忽略不计。当在需要给出位置时，启动 minmax 树进行搜索。耗费时间将随着树的最深层数呈几何级增长。

总体时长方面，经过较为合理的控制，程序能够在 5 秒内完成 500 回合下棋动作。时间的主要开销主要在于不断拓展树的深度时子树规模的大幅增长和程序递归时带来的时间开支。局面评价函数也有一定的运算量。先后手时间存在差异。图中，前半部分为后手单步移动耗时，后半部分为先手单步移动耗时。本地平均耗时在 0.08s 左右。从图中可以看出，后手所耗时间整体较先手更长，但差距并不明显，这是因为后手需要比先手多搜索一次导致的。



3.3 经典战局（可选）

1、我方　v.s　F19-Sierra

比赛选自 2020/6/01 00:33，我方（先手）对战 sierra（后手），比赛长度 1935，上图为第 213 回合，对方不断的尝试在我方放棋子并进行吞并，局势对我方十分不利，一有不慎就有被卡死的危险。

上图为第 281 回合，尽管对方不断的以自损八百伤敌一千的方式侵略我方的领地，但是我方以优秀的合成棋子技术成功地渡过了 68 回合合成了 512 解围。而对方也因为侵略我方被反杀和在我方放子拉大了双方差距，从而让我方处于了领先地位。我方也保持着这个优势直到比赛结束。

上图为比赛结束时的局面，这次比赛让我们发现了进攻是一把双刃剑，我们的程序具有一定的反侵略技巧和 2048 技术，能够成功地化解不太强势的进攻。这与我们代码中的评价函数相吻合，即优先合成大子和重视吞并对方棋子，可以说达到了我们的预先目的。

2、我方　v.s　F19-November

比赛选自 2020/06/01 05:26，我方（先手）对战 November（后手），比赛长度为 1905，上图为第 93 回合，可以看到，对方成功地侵入了我们的领地，但是我方利用对方的侵入营造出了一个良好的陷阱局面：对方的 16 若想吃掉我方 16 就会被 32 反吃，且我方的 32 能吃到对方的 32 并且对方的 64 如果反吃就会再次被 128 反吃。依靠着构造一个又一个的陷阱局面，我方不断地吃掉对方侵略过来的棋子从而拉大双方差距。

上图为双方局面的估值得分，蓝色为我方，可以看到虽然对方的进攻性很强，但是我方的程序善于利用对方进攻从而不断地反吃掉对方的棋子拉大差距。这再一次地验证了我们的程序具有一定的反侵略技巧，与我们的设计思路一致。

不过，许多的战局暴露出了程序所存在的问题：善于反侵略但不擅长侵略对方。另外我方先手时这个优点的确容易发挥，但如果我方后手时这个优点其实是不太容易发挥的，这也是我们的改进空间。

4 实习过程总结

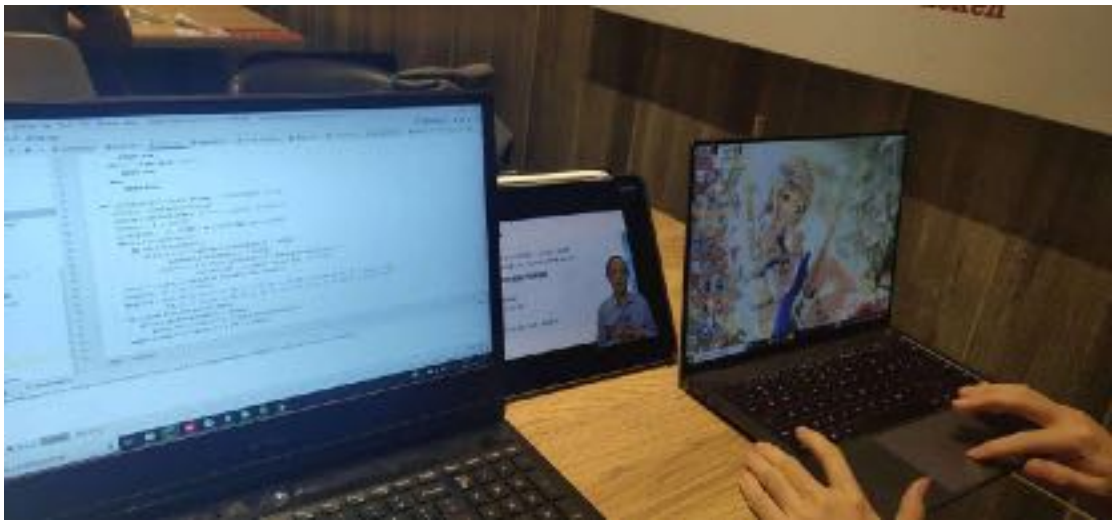
4.1 分工与合作

在大家一致推举马千一作为小组组长之后，我们小组于 5 月 14 日在微信群上进行了初步分工。初步分工的思路是将 5 人分为 221 三组，用不同的思路各自独立开发一款 AI。之后让各组 AI 之间相互对抗，在对抗中比较优劣，并相互借鉴。最终挑选出性能最为上乘的 AI，并将其他 AI 中值得借鉴的部分融合进去，锻炼出更强大的 AI。分组情况如下：

马千一：min-max 决策树

张驰，刘嘉：蒙特克罗树算法

尹擎，马宇琦：启发式算法



图为同城的尹擎和马宇琦线下面基，开发启发式算法

在此过程中，张驰于 5 月 14 日在 coding 平台上建立了我们小组的工作平台。从此我们小组便将开发过程中的代码都上传到 coding 上，一人一个代码文件夹，方便交流借鉴。

图为我们在 coding 上的工作平台

5 月 23 日，我们召开了线上组会，展示了前一阶段各个 AI 开发的成果，分享了各自的开发思路。同时由于开赛日期临近，我们对下一阶段重点攻关的任务进行了分配。任务分配结果如下：

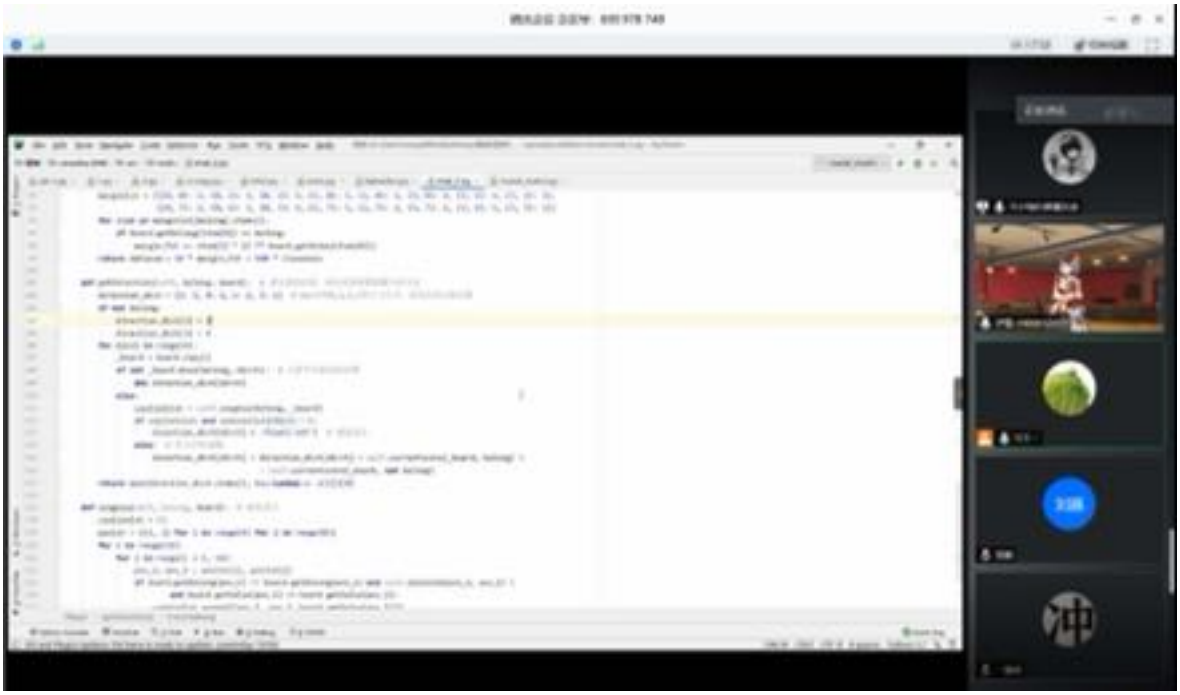
马千一：继续更新 min-max 决策树算法

张驰：继续更新 MCTS 算法

马宇琦：协助张驰对 MCTS 算法 debug

尹擎：继续更新启发式算法，总结出有用的启发式规律

刘嘉：改进评价函数



上两图为线上组会实录

5 月 28 日，MCTS 算法在历经艰辛之后终于能跑了！然而由于战力不佳，不得不被我们弃用。由于 ddl 迫在眉睫，无暇更近其他代码，我们最终选择马千一的 terrible 系列（min-max 决策树）作为参战 AI。在最后阶段，马千一继续开发 terrible 系列 AI，而其余四人协助改进 AI 的评价函数等代码，众志成城完成最后的 AI 提交。

4.2 经验与教训

经验：

1.科学有效的分工合作可以事半功倍。我们小组从开始开发到最后自始至终每个人都有明确的任务与目标，这得益于组长的领导和每个人的参与感、责任感。
2.遇到重难点应该积极向组员求助，共同攻坚克难。比如我们组的评价函数就是多人一同实验改进的，MCTS 算法开发过程中的 bug 也是其他组员协同解决的。
3.研究天梯上的顶尖 AI 对我们自己算法的开发非常重要。比如我们在启发式算法开发过程中参考研究了众多顶尖 AI 之间的棋局,由此获得灵感提高自己 AI 的战力。
4.代码有清晰的注释后可以方便其他组员阅读和提建议，观感极佳。

教训：

1.MCTS 算法因实战能力不佳最终被摒弃，我们都对此十分惋惜。如果我们在开发过程中能尽早认识到这个算法的工作量之大，及早做出调整或许结果会是另一个样。
2.作为 ddl 战士，我们虽然在 ddl 面前迸发出了强大的战斗力，但不免手忙脚乱，略显仓促。组长在比赛后整理代码时，增添了一点小改动结果 AI 战力大幅提高。如果我们不是压着 ddl 完成任务，而是预留出足够时间开发的话，或许就不用在出线边缘起起落落，最终失意。
3.赛后的情况证明，其实我们是有能力做出更简洁且强大的 AI 的，但是在 ddl 迫近时有些慌张，未能静下心来进行思考和更有效的改进，导致在天梯上参加比赛的代码版本比较混乱，战力受随机因素的影响波动较大，酿成未能出线的结果。

4.3 建议与设想

1.F19 赛区较 N19 赛区竞争更加激烈，如果能增加 F19 赛区出线名额或许更加合理。
2.直播过程中光看比分的变动略显乏味，如果能有那么几局的实时动画直播，或像天梯体平台上那样实时提供除比分外其他有趣信息或许观感会更佳。
3.或许可以在天梯上开发人类对战代码的功能，这样在开发 AI 时找灵感就变得更加容易。不过据说这个好像挺难做的是吧？（难做的话就当我们没说哈哈哈）

5 致谢

感谢陈斌老师一学期来辛勤的教学，耐心的讲解，组织丰富的活动
感谢各位助教的讲解答疑、作业批改、组织教学、默默付出
感谢技术组大佬们开发出如此 interesting 的对战项目，并搭建了完备的对战平台
感谢马千一组长自始至终积极理性的组织领导，以及开发了 terrible 系列高战力 AI
感谢张驰坚持不懈开发 MCTS 算法，以及在 Coding 上搭建工作平台
感谢尹擎在启发式算法上的突出贡献与奇思妙想
感谢刘嘉在 ddl 面前毫不畏惧，不舍昼夜完成提高战力的目标
感谢马宇琦在启发式算法上的贡献，以及攻克 MCTS 算法中的 bug
最后，感谢所有组员的辛勤付出，团结一心。结果固然重要，但沿途的风景或许更值得回味。
“多年之后，我们仍会想起，那个巨肝无比的夜晚……”

6 参考文献

[1]https://zhuanlan.zhihu.com/p/30458774
[2]https://zhuanlan.zhihu.com/p/25345778
[3]https://zhuanlan.zhihu.com/p/53948964
[4]https://youtu.be/UXW2yZndl7U

数据结构与算法课程实习作业报告

陈钇冰 *　胡嘉航　许兆晖　杨言智　黄雨衡

F19 Uniform 组

摘　要：本组提交的二〇四八竞赛算法主要基于博弈树中的 min-max 搜索和 α-β 剪枝算法，通过合理涉及盘面价值函数和尽可能提升代码运行效率来获得更高的智能。本组代码最终在天梯赛中位居榜首，并在淘汰赛中最终获得了 F19 组的冠军。在本次竞赛的实践过程中，我组始终位于领先地位。

关键词：博弈树；α-β 剪枝；启发式算法

目录

1　算法思想　2

1.1　总体思路　2

1.2　算法流程图　2

1.3　算法运行时间复杂度分析　3

2　程序代码说明　5

2.1　数据结构说明　5

2.2　函数说明　5

2.2.1　MyChessboard 模块　5

2.2.2　DecisionTreeNode 模块　9

2.2.3　DecisionTree 模块　10

2.2.4　Player 模块　11

2.3　程序限制　12

3　实验结果　12

3.1　测试数据　12

3.2　结果分析　14

3.3　经典战局　17

4　实习过程总结　17

4.1　分工与合作　17

4.2　经验与教训　19

4.3　建议与设想　19

5 致谢 19

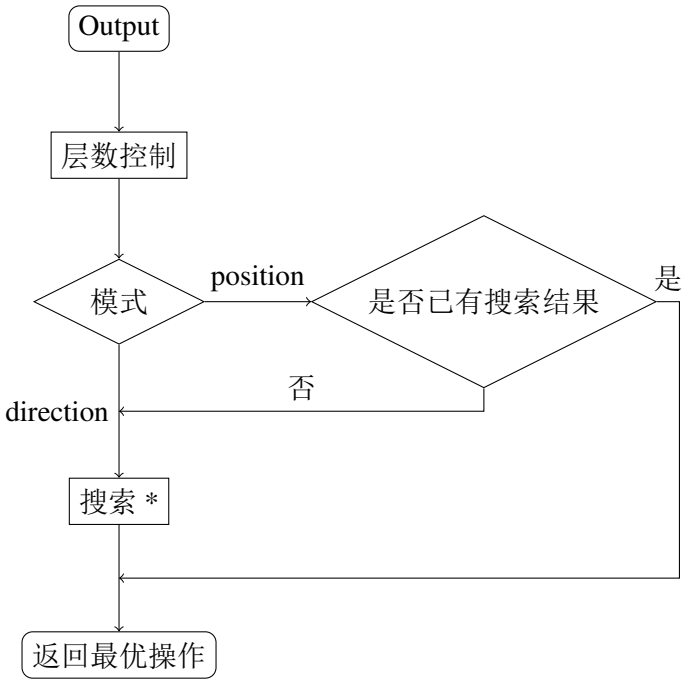
6 参考文献 19

1 算法思想

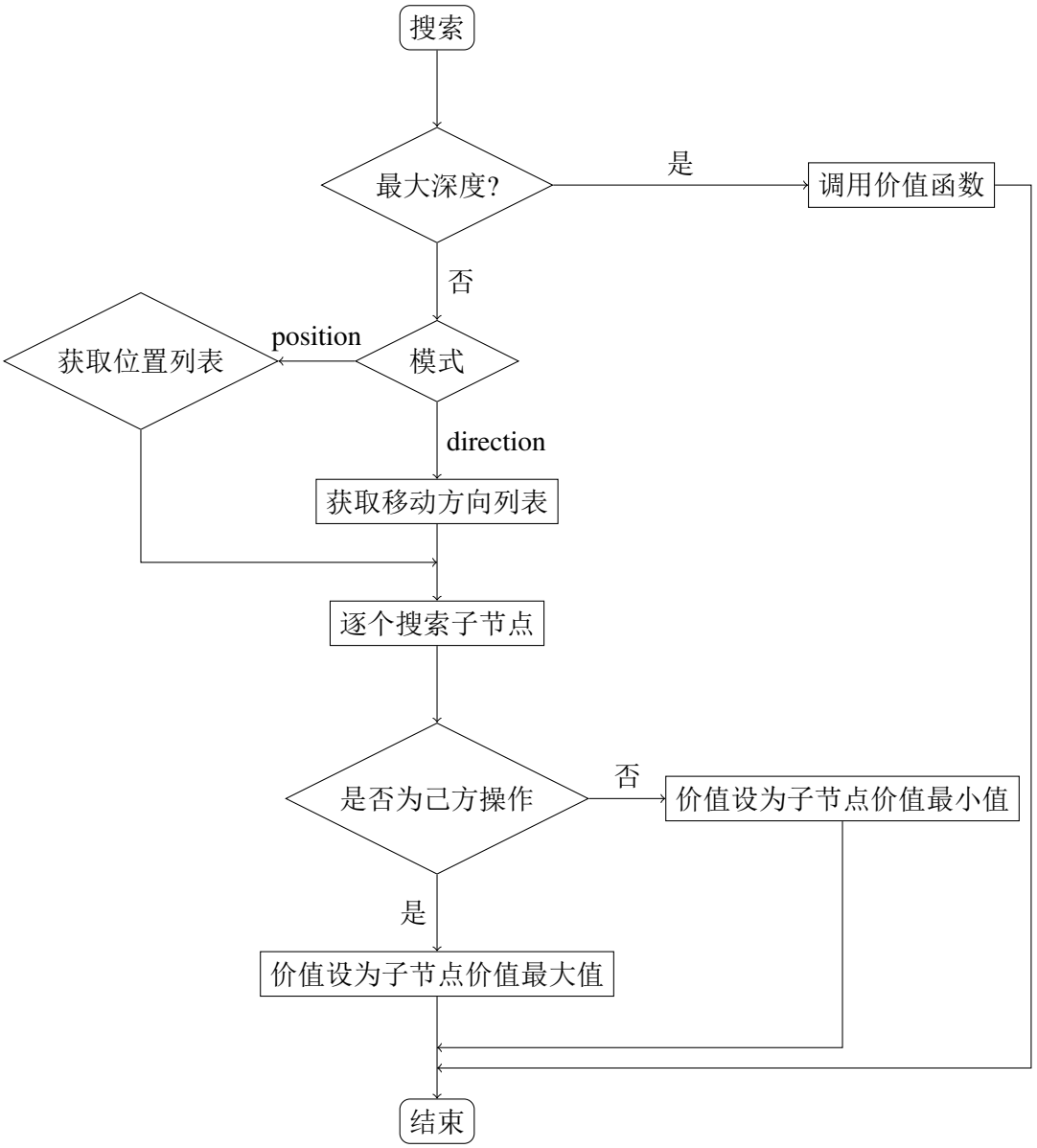
1.1 总体思路

本组代码基于博弈树中的 min-max 搜索和 α-β 剪枝算法，通过合理设计盘面价值函数以及尽可能提升代码运行效率以提升搜索层数、采用启发性的搜索策略来提升 AI 智能，具有相当强大的进攻性。我组成员在算法设计早期已意识到二〇四八算法竞赛获胜的关键因素为更多的搜索层数以及更优良的价值函数。经过中期对算法的完善以及后期的调参，本组算法在搜索层数、搜索方法和价值函数方面具有较明显的优势。

1.2 算法流程图



上图“搜索 *”部分流程如下图：



1.3 算法运行时间复杂度分析

程序主要的时间开销在于搜索，而搜索将会遍历一个多叉树的每个节点，时间复杂度应为指数级别。由于策略树每个节点数量可能改变，底数并非为常量。通过统计搜索过程中的节点数量可以直接获得时间复杂度有关信息。我们从本组 AI 参与的对局中随机抽取盘面作为样例棋盘，分别统计全搜索算法和本组 AI 搜索算法搜索过程中节点数量均值与层数的关系。

使用本组 AI 对多个随机棋盘进行搜索，统计平均策略树节点数和标准不确定度（S_A）:（从先手移动开始，下同，由于我们一般只在移动时进行搜索；由于相邻层节点数只相差约两倍，可以以先手情况的结果作为较好的估计）

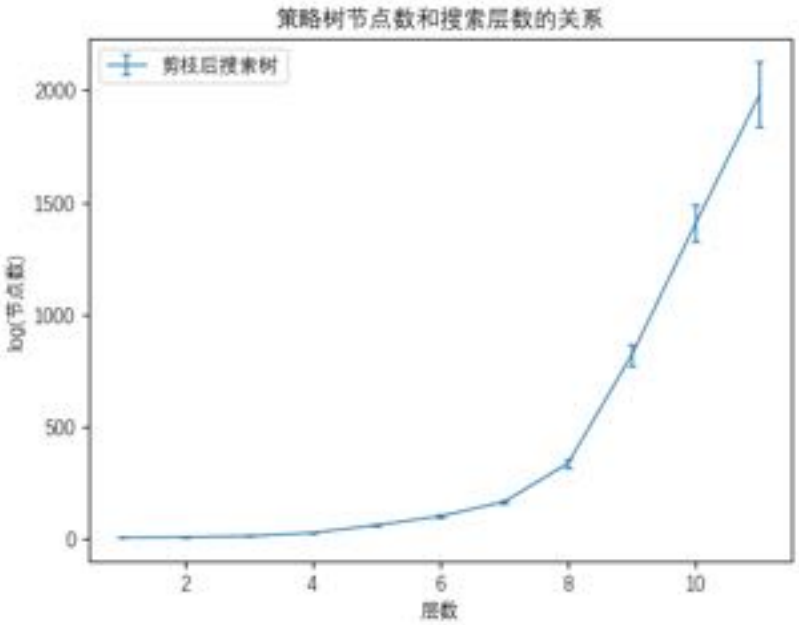


图 1: AI 策略树节点数和搜索层数的关系

统计得随机抽取棋盘可落子位置数及可移动方向数的期望值分别为 6.429,3.062。据此，用程序模拟出全搜索时节点数量和层数的关系，与 AI 策略树结果对比：

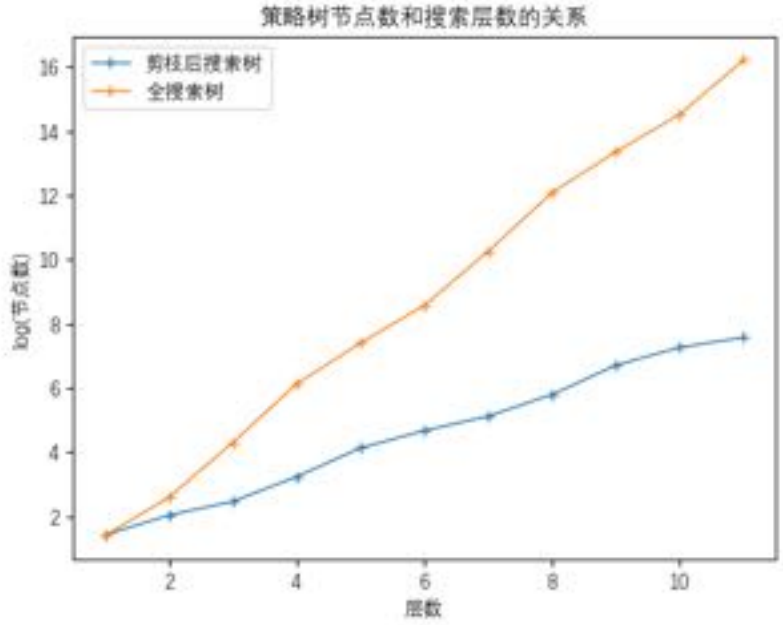


图 2: 策略树节点数和搜索层数的关系（对比）

线性较好，符合指数级时间复杂度的估计。数值拟合得 AI 策略树和全搜索时间复杂度为 $O(n^{1.9})$ 及 $O(n^{4.4})$，相对全搜索 AI 的搜索节省了巨量时间，主要是因为 α-β 剪枝以及有选择性地对对方落子位置进行搜索。

2 程序代码说明

2.1 数据结构说明

由于本组代码采用 min-max 树算法，主要的数据结构为节点类 DecisionTreeNode 和策略树类 DecisionTree，每个节点对象存储其深度以及盘面，并使用字典存储代表所有下一步可能情况的子节点。策略树类存储了代表当前棋局的根节点。

此外，由于前期官方 Chessboard 类方法效率较低，我们实现了棋盘类 MyChessboard，采用一维列表的结构存储所有棋子的归属和级别信息，并实现了和原有 Chessboard 接口基本相同的方法。

2.2 函数说明

2.2.1 MyChessboard 模块

初始化方法 接受随机位置序列（或预设棋盘）。

```
def __init__(self, array, board=None):
```

若初始棋盘不为空则转化为 MyChessboard 棋盘。类中使用 board 一维列表存储棋子信息，每个棋子以 (value,belong) 的二维元组存储。

```
if board is None:  # 空白棋盘
    self.board = [(0, x < 4) for y in range(4) for x in range(8)]
elif isinstance(board, MyChessboard):  # 读入已有的MyChessboard类
    self.board = board.board[:]
else:  # 读入标准棋盘类
    self.board = [(board.getValue((y, x)), board.getBelong((y, x))) for y in
        range(4) for x in range(8)]  # 棋盘所有棋子
```

move,add,getBelong,getValue,getNone,getNext 方法 与官方 python 版棋盘接口一致，但效率有显著提升。

oppoPosition 方法 接受以 bool 值表示的当前方，返回盘面中危险对立之间空格的位置列表。

```
def oppoPosition(self, currentPlayer) -> list:
```

其中危险对立情形指不同方棋子级别相同且至少为 4 并处在相同行或列，中间均为对方空格。

getPosList 方法 接受己方是否为先手、以 bool 值表示的当前方和当前轮数，返回当前方被纳入决策的下一步落子位置列表。

```
def getPosList(self, isFirst, currentPlayer, currentRound) -> list:
```

返回的位置列表基于一些启发式规则给出，避免全搜索带来的效率困境。

(1) 扫描棋盘中的危险对立，特别地当当前方为我方时发现危险对立直接返回，强迫 AI 落子在危险对立处防止被吞并。

```
posList = self.oppoPosition(currentPlayer)
if currentPlayer == isFirst and posList:
  return posList
```

(2) 寻找可能较优的在对方棋盘落子的位点并强迫落子。具体来说，当当前方为我方 AI 且有位点满足：其横向或竖向两组相邻格至少有一组满足为边界或级别至少为 5。其目的为阻塞对手，实验表明，考虑到搜索树的短视性此处确应为强迫落子。

```
Nones = self.getNone(not currentPlayer)
lenNonSelf = len(Nones)
if currentPlayer == isFirst and lenNonSelf < 6:
    for (y, x) in Nones:
        # 寻找能堵塞对方移动的位点
        if ((y == 0 or self.board[(y - 1) * 8 + x][0] > 4) and
            (y == 3 or self.board[(y + 1) * 8 + x][0] > 4)) or (
                (x == 0 or self.board[y * 8 + (x - 1)][0] > 4) and
                (x == 7 or self.board[y * 8 + (x + 1)][0] > 4)):
            return [(y, x)]
```

(3) 扫描我方空格，当空格数小于 3 时防御性强制往对面落子。

```
lenSelf = len(self.getNone(currentPlayer))
if lenNonSelf and lenSelf < 3:
    return [Nones[0]]
```

(4) 添加我方盘面落子点。

```
Next = self.getNext(currentPlayer, currentRound)
posList += [Next] if Next != () else []
```

(5) 对我方 AI 当对面空格小于等于 2 时将往对面落子加入考虑，否则，若当前方无子可落或我方空格小于 6 时，或即将生成的子恰位于对立处时，添加往对面落子情况。其中，即将生成的子恰位于对立处时添加往对面落子情况，主要由于实践中发现对手会通过往我方落子回避自然生成在对立处（而我方 AI 未考虑到这一点），从而吞并我方的棋子。

```
if currentPlayer == isFirst and 0 < lenNonself <= 2:
  posList.extend(Nones)
elif lenNonself:
  if not posList or lenSelf < 6:
    posList.append(Nones[0])
  elif Next != ():
    up, down, left, right = (-1, None), (-1, None), (-1, None), (-1, None)
    for y in range(Next[0] - 1, -1, -1):
      if self.board[y * 8 + Next[1]][0]:
        up = self.board[y * 8 + Next[1]]
        break
    for y in range(Next[0] + 1, 4):
      if self.board[y * 8 + Next[1]][0]:
```

```
14          down = self.board[y * 8 + Next[1]]
15          break
16      for x in range(Next[1] - 1, -1, -1):
17        if self.board[Next[0] * 8 + x][0]:
18          left = self.board[Next[0] * 8 + x]
19          break
20        elif self.board[Next[0] * 8 + x][1] != currentPlayer:
21          break
22      for x in range(Next[1] + 1, 8):
23        if self.board[Next[0] * 8 + x][0]:
24          right = self.board[Next[0] * 8 + x]
25          break
26        elif self.board[Next[0] * 8 + x][1] != currentPlayer:
27          break
28          if (up[0] != -1 and down[0] != -1 and up[0] == down[0] and up[1] != down
                [1]) or (
29              left[0] != -1 and right[0] != -1 and left[0] == right[0] and left[1]
                    != right[1]):
30        posList.append(Nones[0])
```

evaluate 方法　接受己方是否为先手、下一步最近的移动操作方，返回盘面价值。

```
1 def evaluate(self, isFirst, whoNext):
```

(1) 基准价值机制。对于每个棋子确定一个基准价值，以 valuePlus 记。valuePlus 的基础值仅由是否属于己方及其级别决定，在不属于己方时乘以 -1。

基准价值的绝对值和级别的关系参考变量 valueValue：

```
1 valueValue = [1.0, 2.5, 6.25, 15.625, 39.0625, 97.65625, 244.140625,
    610.3515625, 1525.87890625 * 1.25, 3814.697265625 * 1.5, 1145141919810]
```

其数值的具体含义见下面的表格：

棋子级别	基准价值
$n = 1$ 至 7	2.5^n
8	$2.5^8 \times 1.25$
9	$2.5^9 \times 1.5$
10	1145141919810

表 1: 棋子级别和基准价值

（当一方拥有级别为 10 的棋子时对方无法合成同样大小的棋子，该方必胜，因此基准价值设定为一个充分大的数）

(2) 为增强进攻性做出的价值调节。如果我方的入侵了对方领地，其价值变为原先的 5 倍。在这种机制下 AI 表现出强烈的进攻性，并且除非即将被对方吞并一般不会将棋子从边界移出。

以己方先手为例：

```
if x > 3:  # 当棋子入侵对方时增大价值，增强进攻性
    valuePlus *= 5
```

这种策略同时存在一定的问题：由于入侵对方的价值过高，AI 有时不愿意将棋子合并为更大的棋子。

(3) 带来接近层数增加效果的补偿设计。虽然搜索层数本身受到执行速度的限制无法增加，但可以通过价值函数使之在某种程度上达到与增加层数相似的效果：具体地，对于两个相邻的同所属方级别相同的棋子，其价值均调整为原先的 1.25 倍，从而其价值和与两棋子合并为一个棋子后的价值等同。

以己方先手为例：

```
if chessValue >= 2 and (
    (eightY != 0 and self.board[position - 8] == (chessValue, True))
    or (eightY != 24 and self.board[position + 8] == (chessValue, True))
    or (x != 0 and self.board[position - 1] == (chessValue, True))
    or (x != 7 and self.board[position + 1] == (chessValue, True))):
    valuePlus *= 1.25  # 己方相同子相邻加成，增强合并的倾向
```

另一个补偿设计则是对于两个相邻的不同所属方级别相同的棋子，其价值按照下一个做出移动的方 whoNext 决定。如果是对方，则视作该棋子被对方吞并，价值变为相反数。

以己方先手为例：

```
if whoNext is False and (
    (eightY != 0 and self.board[position - 8] == (chessValue, False))
    or (eightY != 24 and self.board[position + 8] == (chessValue, False))
    or (x != 0 and self.board[position - 1] == (chessValue, False))
    or (x != 7 and self.board[position + 1] == (chessValue, False))):
    valuePlus *= -1  # 下一次移动为对方移动时与对方相同子相邻减成，防止白给
```

这种设计与下一次在搜索树中移动后再 evaluate 得到的效果近乎等同（注：在某些特殊情况下存在差异：例如两对相邻棋子其一为上下相邻，其二为左右相邻），一定程度上提升 AI 的智能。

在 evaluate 函数中将所有的判断语句全部展开书写（inline）以免多次调用消耗时间，尽管一定程度上降低了代码的可读性和可维护性，但因为 evaluate 方法在运行中多次被调用，测试结果表明效率确实大约 10% 至 20% 的提升。

evaluate0 方法　代替 evaluate 方法，基准价值采用 16 为底数的幂次，同时附加判断条件仅保留了即将被吞并棋子价值取相反数。

```
valuePlus = (1 if chessBelong == isFirst else -1) * 16 ** chessValue
```

此方法在临近 500 回合时被调用，使得 AI 倾向于将低级别棋子合并成高级别棋子的倾向极大增加，保证结束时在高级别棋子上的数量优势。

evaluate1 方法　代替 evaluate 方法，基准价值采用 2.05 为底数的幂次，同时附加判断条件仅保留了即将被吞并棋子价值取相反数。

```
valuePlus = (1 if chessBelong == isFirst else -1) * 2.05 ** chessValue
```

此方法在前 150 回合时被调用，使得 AI 倾向于增加自身棋子数值之和，使得前期积累优势。由于前期双方空格较为充裕，不采用堵塞 AI 棋盘的策略。采用前 150 回合的原因是根据实验水平接近的 AI 双方空位在 150 至 200 回合时逐渐趋于紧缺，此时开始切换成入侵的策略侵占对方棋盘最优。

2.2.2 DecisionTreeNode 模块

初始化方法 接受存储当前盘面的 MyChessboard 对象、节点深度以及 AI 是否为先手。

```
def __init__(self, board, depth, isFirst):
```

其中初始棋盘对应的节点深度为 0，每一方进行一次操作（包括空操作）节点深度 +1，其他属性通过 depth 直接换算得来：

```
self.currentRound = depth // 4  # 当前轮数
self.currentPlayer = False if depth % 2 else True  # 下一步是否为先手执子
self.isCurrent = isFirst == self.currentPlayer  # 下一步是否为AI执子
self.mode = (depth % 4) // 2  # 下一步操作类型，0为落子，1为移动
```

使用这种传输方式可以大量减少传递参数的数量。

该节点所有对应下一步操作的子节点将存储在字典 self.children 中，其中键值代表操作（(x,y) 代表在棋盘该位置落子，0-3 代表向上、下、左、右移动，None 代表无法操作）。

search 方法 递归向下搜索可能的局面并更新最优/最劣操作及价值。

```
def search(self, maxDepth, abLimit=None):
```

当模式为落子时，所搜索的位置列表由当前棋盘对象的 getPosList 方法有选择地给出。

```
for pos in self.board.getPosList(self.isFirst, self.currentPlayer, self.
    currentRound):  # 所有决定搜索的落子位置
```

而当模式为移动时，搜索方向的顺序将为先向中线、后向两边、最后向上或下的顺序，使得在有相同价值时 AI 倾向于向中线移动以提高进攻性；实验表明这种搜索顺序将有明显的胜率优势。

```
for direction in ((3, 2, 1, 0) if self.currentPlayer else (2, 3, 0, 1)):
```

由于搜索树可能留存上一次搜索时产生的节点，当节点已存在时不重建节点而是直接对已有节点重新搜索；否则新建节点并将其加入 self.children。

搜索函数使用 α-β 剪枝以节省效率。当前层将维护一个 newAbLimit，它代表到目前搜索过子节点价值的最小值（目前为极大层）或最大值（目前为极小层）。该层价值及对应采取的最优操作将同时被维护：

```
if newAbLimit is None:
    newAbLimit = newNode.value
    self.value = newAbLimit
    self.bestOperation = pos
else:
    if (self.isCurrent and newNode.value > self.value) or (
            not self.isCurrent and newNode.value < self.value):
        self.value = newAbLimit = newNode.value
        self.bestOperation = pos
```

newAbLimit 将被作为搜索子节点时的 abLimit 参数。当目前搜索过子节点的价值中有小于（目前为极小层）/大于（目前为极大层）abLimit 时意味着当前节点不可能作为上层的最优选择，因此直接停止搜索以节省时间：

```
if abLimit is not None and ((self.isCurrent and newNode.value >= abLimit) or
                            (not self.isCurrent and newNode.value <= abLimit)):
    break
```

当搜索到最大层数时将会对当前节点盘面进行估值并向上传递。在不同的回合数时将采取不同的估值函数：前期采用注重棋子数值和的价值函数以保证前期积累足够的数值和；中期采用进攻性强的价值函数以增大选择吃子的概率，同时挤压对手生存空间，迫使对手无法移动或进行强制的不利移动；后期几回合将采取注重合成大子的价值函数，在终盘将优势转化为胜势。

```
if self.depth == maxDepth:  # 搜索到最大层数，计算价值并回溯
    if self.currentRound < 150:  # 前期采用保持棋子价值总和的价值函数，保证己方总价值
        self.value = self.board.evaluate1(self.isFirst, self.depth % 4 != 3)
    elif self.currentRound < 490:  # 中期采用进攻性强的价值函数，挤压对手生存空间
        self.value = self.board.evaluate(self.isFirst, self.depth % 4 != 3)
    else:  # 后期采用标准价值函数，保证尾盘达成游戏胜利条件
        self.value = self.board.evaluate0(self.isFirst, self.depth % 4 != 3)
```

2.2.3 DecisionTree 模块

初始化方法 接受 AI 是否为先手、初始位置序列。

```
def __init__(self, isFirst, array):
```

初始化中将同时生成一个节点作为 root 对象，采用初始的空棋盘。

search 方法 从根节点开始向下搜索若干层。

```
def search(self, depth):
    self.root.search(min(self.root.depth + depth, self.limitDepth))
```

在搜索树中实现独立的搜索方法目的是便于控制最大搜索深度。

applyOperation 方法 将根节点改为（先手或后手方）进行某步操作后的节点。

```
def applyOperation(self, operation, currentBoard):
```

当 root 没有对应该操作的子节点时（一般为对方在我方落子）将直接使用当前盘面重建根节点。

2.2.4 Player 模块

初始化方法 接受 AI 是否为先手、初始位置序列。

```
def __init__(self, isFirst, array):
```

初始化过程中建立默认的策略树并进行一次搜索，此后不再重建策略树以节省时间。

output 方法 比赛的标准接口，返回选择的操作。

```
def output(self, currentRound, board, mode):
```

在搜索前调整当前步的搜索层数。搜索过程的最后一步为移动才会对搜索结果产生影响，由于通常情况下只在模式为移动时搜索（见下），对先手（后手）而言有效的层数选择是 1/2/5/6/9/10 层（1/4/5/8/9 层）。由于本程序的性能支持的平均搜索层数大约在 6-7 层，因此采取先手全程搜 6 层、后手搜 5 或 8 层的策略。

```
if self.isFirst:
    self.searchDepth = 6
elif currentRound < 30:  # 前期压低算力
    self.searchDepth = 5
elif currentRound % 5 == 0:
    self.systemTime = board.getTime(self.isFirst)
    if self.searchDepth == 8:
        self.numEight += 1
        self.sumEight += self.lastTime - self.systemTime
    avgEight = self.sumEight / self.numEight if self.numEight else 0
    ratio = avgEight / ((self.systemTime - 0.1) * 5 / (500 - currentRound))
    if ratio < 0.87:
        self.searchDepth = 8
    else:
        self.searchDepth = 5
    # 即将超时时压低算力
    if self.systemTime < 0.2:
        self.searchDepth = 5
    self.lastTime = self.systemTime
```

在后手的情况，player 类将会维护 sumEight 和 numEight 作为历史搜索 8 层时间总和及历史搜索 8 层轮数（的 1/5）。每次搜索时将计算历史所搜 8 层的平均时间作为参考，若该时间比剩余平均每轮可用时间（假设游戏进行到 500 回合）小于一定比例（在天梯上实验的结果）则接下来五轮采用 8 层，否则采用 5 层。由于需要将算力运用到最重要的中期，前期一小段时间将会压低算力

到 5 层；当时间即将用尽时也采取 5 层。这种时间管理机制保证 AI 基本不会超时，并且对时间的利用较为充分。

当不是初始局面时将更新操作。由于两层前已经搜索较多的层数，当模式为落子时将不再重新搜索，直接返回以前搜索出的最优操作；当无最优操作（搜索树重建）或模式为移动时将进行一次搜索并返回最优操作。

2.3 程序限制

由于程序采取了自己实现的棋盘类，鲁棒性强。此外由于有较好的时间管理机制，在各种环境下运行基本不会超时，发挥稳定。

3 实验结果

3.1 测试数据

实验环境说明：

1. 硬件配置：Intel(R) Core(TM) i7-9750H CPU @ 2.60GHz
2. 操作系统：Microsoft Windows 10
3. Python 版本：Python 3.7

本组在技术组给出的本地比赛工具基础上编写了本地对战系统，实现了一键对战并自动统计结果，同时编写了自动棋局分析系统，使“养蛊”和“炼丹”成为可能。利用本地对战系统，我组通过先后手胜率，平均用时等数据对算法进行评价与改进。本地测试是我组在算法竞赛前中期使用的主要测试方式。

底数选择 在标准的 evaluate 中，棋子依据等级而被赋予基准价值，棋子的基准价值和等级之间呈指数关系 (见2.2.1)，此时底数的选择非常重要。以底数为 2.5 的代码为基准，在一定范围内分别生成 1000 个底数不同的代码与其对战（共 2000 场），得到结果如下（图中每个点值为长度 0.3 领域内平均值，下同）：

胜率在底数取 2.5 附近值时达到最高，因此选取 2.5 作为基准价值和棋子等级关系的底数。

进攻系数 在标准的 evaluate 函数中，为增强进攻性，入侵对方领地的棋子价值将乘以一定系数。(见2.2.1) 以进攻系数为 4 的代码为基准，使用无进攻性加成的代母与其对战 200 场，得到结果如下：

代码类型	先手胜率	后手胜率
基准	78%	76%
无进攻系数	24%	22%

可见相对于无进攻加成的代码，附加了进攻系数的代码取得了显著的优势。

进一步，以进攻系数为 4 的代码为基准，在一定范围内分别生成 1000 个底数不同的代码与其对战（共 2000 场），得到结果如下：可见进攻系数为 5 时先手胜率与后手胜率均略有提升，同

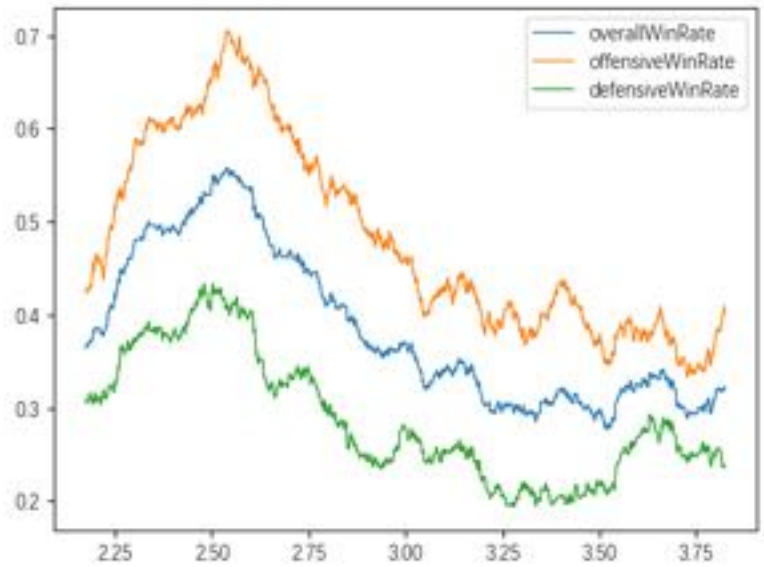


图 3: 胜率-底数关系图

图 4: 胜率-进攻系数关系图

时在实战中效果较好，因此在最终版本中选择采用 5 为进攻系数。

相邻加成 在标准的 evaluate 函数中，为增强合成棋子的倾向，己方相邻的两个级别相同的棋子价值将会得到加成 (见2.2.1)。以加成系数为 1.25 的代码为基准，使用无加成的代码与其对战 200 场，得到结果如下：

代码类型	先手胜率	后手胜率
基准	62%	65%
无相邻加成系数	35%	38%

可见相对无相邻加成的代码，附加了相邻加成的代码取得了显著的优势。

相邻减成 在标准的 evaluate 函数中，为防止搜索树未搜到下一步即将被吞并的情况，在对方即将移动前己方棋子若相邻对方级别相同的棋子将会得到减成 (见2.2.1)。以附加相邻减成代码为基准，使用无减成的代码与其对战 200 场，得到结果如下：

代码类型	先手胜率	后手胜率
基准	50%	56%
无相邻加成系数	44%	50%

相对无相邻减成的代码，附加了相邻减成的代码有较小的优势。实际对局中我们发现增加了这一功能的代码在与其他代码的对局中效果较好，因此采用这一策略。

3.2 结果分析

我们在早期便确定采取 $\alpha - \beta$ 剪枝的 min-max 树策略，由于我们一开始的版本采取完全不搜索对方落子点的方式，即使在没有优化时也能搜索 5 至 6 层，因此早期在天梯上很快登顶。由此，我们确定了优化搜索效率的方向，并且很早就对棋盘类进行了重写，在模拟棋盘上节省的时间使我们搜索深度能稳定维持 6 层左右并有时间剩余。

在优化效率达到瓶颈后我们从价值函数和启发式增加向对方落子节点两个角度进行考虑。中期有较长的时间我们都在进行价值函数的调整。在比较进攻性和防守性价值函数时，我们发现进攻性的 AI 在本地和线上都有很好的发挥，通过侵蚀地盘使得对方操作选择减少，在很早的局面就能将对方塞爆。因此，增强进攻性成为后续调整主要基调。同时我们对价值函数进行了一些其它调整。

在早期我们与其他 AI 的战斗经常能达到全胜，但我们发现同一些 AI 如大萝莉（后证实是 N19 组冠军 N19_Sierra 组）比赛时常会出现少数几盘输的情况，在不断调整价值函数后仍然没有改善。通过研究对局我们发现其经常在某方即将吞并对方棋子时在中间的空格处落子，使我方 AI 无法提前做出正确的选择。我们猜测对方采用全搜索树的策略，但全搜索会造成效率大幅度降低，因此我们专门针对两个大子面对的情况对对方空格进行搜索。经试验这种策略效果很好，使得新版本迅速在天梯中取得优势。

中期我们陷入了瓶颈期，我们计划使用本地的养蛊系统进行大规模的系数调试，但由于算力不足以及部分调整后天梯表现更差，总体收获较小（事实上我们初始采取的参数都是接近最优的）。

随着正式比赛的临近，各组 AI 水平都有大幅度提高。在同 F19_Echo 组（的小号）对战时我们的代码一度陷入劣势。通过本地搭建的分析系统我们仔细对棋局进行了分析，发现我们的代

图 5: 屠榜天梯 (图中所有代码均为我组代码的不同版本)

码在前中期往往陷入棋子数值和的劣势，而获胜的局一般是后期将对方塞爆翻盘造成的。

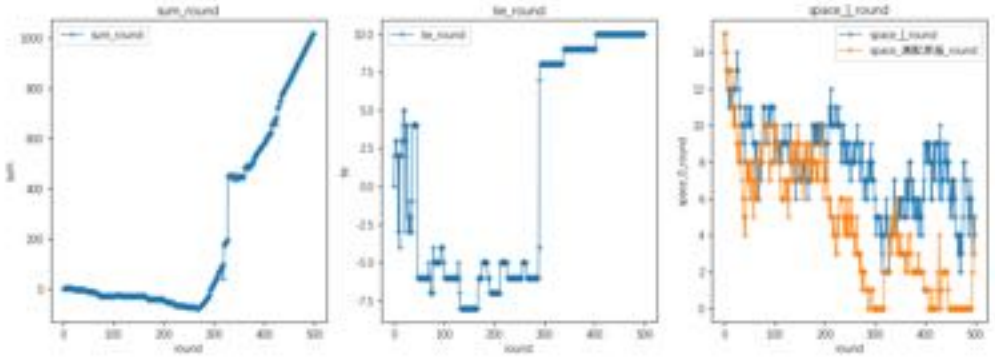


图 6: 与 Echo 组典型对局 1

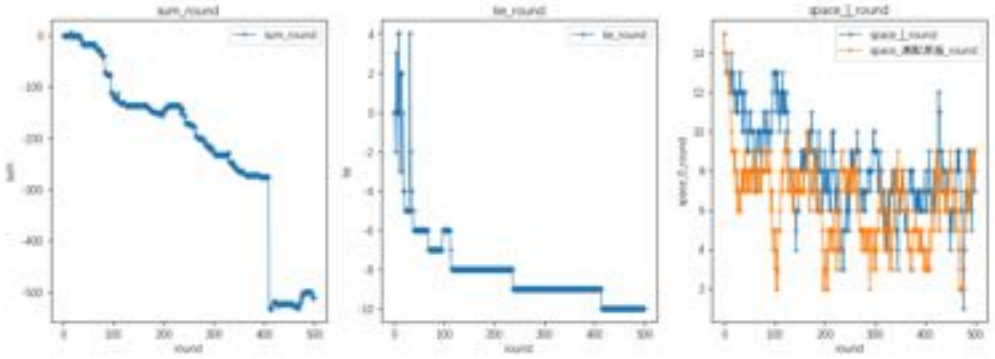


图 7: 与 Echo 组典型对局 2

（左、中、右分别为棋子数值和差距、棋子最大级别差距、双方空格数）以上是一些我们 ('[') 和

Echo 组小号 (' 满配原版') 比赛的分析图。对局 1 我方先手，前期陷入劣势，后期通过塞爆对方（直线段）翻盘；对局 2 则是我方全程劣势。更多的比赛记录中无一是我方前期数量和取得优势的。我们认为由于我们采取了进攻加成的策略，往往会牺牲一些棋子保证己方棋子入侵；然而从图中看出前 200 回合左右双方空格数都有较大盈余，前期的进攻并不能很好起到堵塞对手的作用。受此启发，我们将前 150 回合的价值函数调整为倾向数量和最大，并且取消进攻的加成。这个版本在与 Echo 组以及其他较强组的比赛中取得不小的优势。

在天梯赛的最后几天比赛进入白热化的阶段，我们发现许多 AI 的策略变得更加激进，向对方塞子的倾向大幅增加。典型的如 F19_Zulu 组，他们的代码在某次更新后经常通过连续塞子将我们堵死，使得两边比分经常不分上下：

图 8: 与 Thano 组天梯赛截图，全程塞爆

在危难之际，我们决定增加代码塞子的激进程度，在很多情况下会强制向对面塞子。虽然在本地效果一般，但在天梯赛中终于取得了较好的效果，并且将优势延续到了比赛结束。
（图为我们爬取 5.31 关闭提交代码通道到当天结束的数据后绘成的天梯赛部分组别 AI 克制关系图，可见我们的 AI 对几乎所有其他组在胜率上都有优势。）

综合来看，我组 AI 取得优势的因素主要有：

- 效率和复杂度的合理权衡。我们一方面在早期优化了搜索过程中大量调用模块的效率，保证搜索层数不会过低，另一方面节省出的时间分配在启发式搜索和估值函数上，将增加层数逐渐递减的边际效益转移到更为有效的地方。
- 合理的估值函数。通过本地实验和天梯实战我们发现并最终采取了进攻性的估值策略，相较于单纯提高合并子的能力带来总体上更大的效益。同时我们采取了一些微调使得估值函数一定程度上能反映未来一步甚至多步的优劣，相当于变相增加搜索层数。
- 启发式的搜索策略。考虑向对方落子的位置时我们经过一系列规则给出较优的位置选择，一方面规避了全搜索的极高时间复杂度，另一方面保留了塞子和防止被塞的功能。

我组 AI 的部分劣势有：

- 搜索层数仍不够高，相比于某些 11 层的 imba 组。由于增加层数的边际效益过低，后期我们没有把后期放在这里，但时间控制机制的实现对避免尾盘被迫降低到很低层数作用很大。

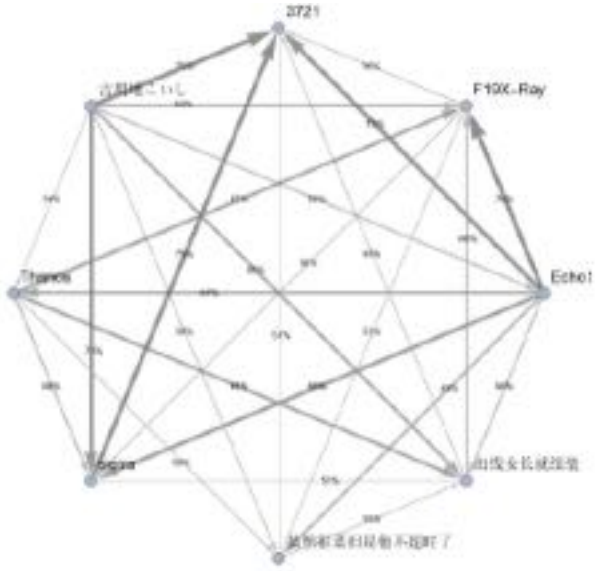


图 9: 天梯赛前八名克制关系图

- 单纯的进攻策略容易导致棋子无法合并，有时甚至被迫被吞并大子。这一点由于搜索层数无法得到突破性提升而难以解决，降低进攻性又会导致胜率的下降，因此一直成为难点。通过在即将被吞并时向对方落子以及后期调回价值函数稍微减少了此种情形的发生。
- 后期增加的塞子策略较为粗暴，由于比赛前不得不针对其他组的战略进行调整而做此选择。

3.3 经典战局

4 实习过程总结

4.1 分工与合作

在此次算法竞赛中，F19 Uniform 组员分工明确，通力合作，解决了大量技术困难，并全程关注算法竞赛各项更新。具体分工与合作情况如下：

姓名	分工
陈钇冰	独立的 move 函数、价值函数设计
胡嘉航	基础算法框架、对局分析系统、时间控制模块、报告数据
许兆晖	独立的 MyChessboard 类、价值函数设计
杨言智	本地对战系统、实习报告主编
黄雨衡	提供部分算力和数据

表 2: 分工情况

胡嘉航实现了基础的 min-max 搜索和 α-β 剪枝算法，编写及改进了功能强大的对局分析系统，实现了精精细化的对局分析。许兆晖和陈钇冰实现了完全独立于技术组给出的棋盘类 MyChessboard 以及全新的 move 函数，两人的工作大大提高了程序的运行效率，相比于原实现，运行时间平均缩短到原来的 33%。此外，以上三人反复改进价值函数，并在实质上实现了 AI 战力

的提升。杨言智编写了本地对战程序，为版本更新带来的增强提供了评价标准，并主要负责报告的编写。黄雨衡同学提供了部分算力，并且提供了部分分析数据。

除此之外，为实现对程序的全面监控和分析，我组还实现了基于爬虫的代码监控系统等多种工具。总体来说，F19 Uniform 组全体成员均在此次竞赛中倾注了大量心血，实现了良好的分工与合作。

我组的组会开展非常密集，几乎是一天一次，一次持续数小时。在算法竞赛期间，我组成员即使不在开会，也时刻关注天梯动向并努力寻找加强代码的方法。频繁的组会与大量的时间投入成为了我组代码持续更新加强的有力保障。组会截图与具体组会内容见下。

图 10: 组会截图 (图为组员观察具体比赛并讨论改进方向)

日期	组会内容
5.14	底层算法、AB 剪枝
5.15	调整方向优先级、确定价值函数底数、上传 RBQ
5.16	确定进攻 > 保守 > 混合的策略、增加时间保底机制、上传 RBQ_2
5.17	增加塞子与防塞策略
5.18	完成本地大规模测试平台、完成独立的 Chessboard 和 move
5.19	增加估值函数考虑相邻棋子耦合产生的影响 增加向对方塞子判断、增加决策树测试模块
5.20	上传 RBQ_4、搭建爬虫系统和分析系统
5.23	增加全局结束控制
5.26	发布了天梯代码古明地こいし
5.29	确定最终使用的时间调节机制, 确定搜索终止层数的深度
5.30	确定最终使用的塞子功能 (针对性调整)
6.1	最终版本定稿

表 3: 组会内容

4.2 经验与教训

我组成员在实践中认识到必须搭建一系列平台用于测试代码强度以及代码决策分析，于是搭建了本地大规模测试平台、自动分析系统以及爬虫系统。利用这些工具，对算法的调整便有迹可循，这是我组算法能够不断增强的重要倚仗。

在本次活动中，我组共解决了两个难题，即超时的解决和价值函数的编写。作为超时问题的解决，我组采用了重新编写内部实现的方法，使用重新编写的 MyChessboard 类和 move 函数减少了耗时 (见代码)。在此过程中我们看到，即便是技术组给出的内容，我们也能够加以修改和完善。对于价值函数的参数调整问题，我组利用自主开发的对战和分析平台，采用类似退火算法的方式，对各参数进行调整。虽然最终因为算力不足而放弃，但最终我们还是获得了一些比原来表现更好的参数。

此外，在实现全新的 Chessboard 类和 move 方法时，曾经出现过一处只在极少数情况下出现的 bug，因此在开始时我们并没有发现它。但在利用测试平台大规模测试时，我们通过统计数据发现了 bug，并通过具体的对局归纳出了 bug 的触发条件。由此，一方面，我组认识到在写底层或内部函数的时候，必须要严谨细致，否则出现的 bug 可能难以发现并延续到此后多个版本。另一方面，我组再一次认识到了大规模测试的必要性。

以上即为我组在此次竞赛活动中所累计的经验与教训。

4.3 建议与设想

对于此后的算法竞赛活动，基于此次算法竞赛的实际情况，我组提出以下建议：

在竞赛开始前，建议留出足够的时间完善竞赛规则和平台，避免发布后再对规则和平台进行修改。本次竞赛中，技术组对 Chessboard 的实现进行过一次调整，对自行实现 Chessboard 类的我组来说，这增加了重新对 Chessboard 类的不同实现进行测试和选取的工作量。

对于天梯比赛，建议取消中止比赛功能。参赛者通过中止比赛功能能够实现小范围的控分，而这样产生的分数差距对于实力相差不大的组来说是非常关键的。

5 致谢

截止到 2020 年 5 月 26 日天梯赛开始，我组代码共更新了 4 个大版本，数十个小版本。在编写各种测试程序时，我组成员一定程度上参考了技术组成员所编写的工具程序。在此特别感谢本次二〇四八算法竞赛的技术组成员。没有平台的支持，二〇四八算法竞赛开展不可能如此顺利，在此，F19 Uniform 组全体组员对于提供平台的北京大学新一代 GIS 实验室表示由衷的感谢。

此外，在讨论过程中，我组成员主要以腾讯会议为主要平台，在此，对腾讯会议提供的支持表示感谢。

6 参考文献

[1]TheFetter, 五子棋 AI 算法原理，博弈树、极大极小搜索、剪枝
https://blog.csdn.net/qq_42058018/article/details/81142881

数据结构与算法课程实习作业报告

谢添威* 陈十楚 李翔宇 荣天泽 喻翰

摘要：采用自身维护棋盘的方式降低基本操作所需时间，搜索算法基于决策树，通过迭代加深和 PVS 对其进行优化。落子时算法会判断对方是否存在吞并我方棋子的机会并尝试落子阻拦；合并时由估值函数和静态搜索决定方向。我们组的 AI 在天梯一度有较高胜率，多次经历了估值函数的调整，但最终没能出线。

关键字：Alpha-Beta 搜索、一维棋盘、主要变例

1 算法思想 2
1.1 总体思路 2
1.2 算法流程图 2
1.3 算法运行时间复杂度分析 3
2 程序代码说明 3
2.1 数据结构说明 3
2.2 函数说明 3
2.3 程序限制 6
3 实验结果 6
3.1 测试数据 6
3.2 结果分析 8
3.3 经典战局（可选） 8
4 实习过程总结 10
4.1 分工与合作 10
4.2 经验与教训 13
4.3 建议与设想 13
5 致谢 13
6 参考文献 13

1 算法思想

1.1 总体思路

总体思路：

初始化时实例化 ChessBoard 类，将棋盘转化为一维数组自身维护，每次决策前获取对方策略更新棋盘，每次决策后在返回前同样更新棋盘，较公用接口具有更高的效率。

在不超过时间限制的情况下，利用博弈树启发式搜索通过评估函数获得相应策略的分数估值，以此进行决策。算法的核心基于决策树。决策树深度搜索的初始层数为 3，即一轮的深度，通过 Alpha-Beta 搜索建立决策树的 max 和 min 节点，同时通过 PVS 剪枝将相似的节点剪除以优化搜索，每次搜索获取并储存主要变例在下次搜索时优先选择。最终通过估值函数与静态搜搜结合的方式获取局面的评估，并决定最后的合并方向。

与此同时，我们的算法还包括了通过计算对方是否存在吞并己方棋子的机会而选择在对方棋盘落子来阻挡。算法的后一部分起初占比不大，是在天梯赛中的几次失利后受到我们的重视，并加大了其在 AI 中的权重。

主要算法：

利用了 Alpha-Beta 搜索结合 PVS 剪枝构建博弈树

算法策略：

在己方下棋阶段：按照优先级的顺序——首要优先判断在领地交界处的边缘己方棋子是否有可能被对方吃掉且不存在逃跑的方向，如果有可能，则获取格挡位置在该处下棋；其次考虑己方允许下棋的位置；最后考虑对方领地空位随机落子。

在己方移动阶段：通过迭代加深，从仅考虑当前这一步开始逐渐加深，并在退出本次搜索储存主要变例——即该深度下最佳的路径，在下一次加深优先选择进行搜索，以达到尽可能的剪枝效果。由于每轮棋子数不同，相同层数用时不同，通过限制时间，尽可能搜索更多的步数，并在最后利用评估函数计算各个可能的决策的分数。再使用静态搜索对局面中的边界情况作评估，排查己方边界上可能被对方吃掉的棋子与撤退路径，在最后取最佳状况获取移动方向。

对于评估函数：

- 己方空位数越多，评分越高；对方空位数越少，评分越低。
- 己方空位越集中，评分越高。
- 己方合并的棋子数量越多、合并的棋子分值越大，评分越高。
- 己方棋子分值越大且对方分值较大棋子距离己方领地越近，评分越高；对方棋子分值越大且己方分值较大棋子距离对方领地越近，评分越低。
- 己方吃掉对方棋子数越多、分值越大，评分越高；对方吃掉己方棋子数越多、分值越大，评分越低。
- 在领地交界处，若己方棋子可能被吃掉且不存在可以保护的棋子，评分越低。

1.2 算法流程图

1.3 算法运行时间复杂度分析

算法的主要运算时间花费在棋盘类的 Move 函数以及估值上，算法时间复杂度为 O(depth^4)量级，但由于我们采用了 PVS 剪枝优化，算法实际时间复杂度是要低一些的。同时我们也通过对每层迭代设置时间上限以规避算法超时的错误。

2 程序代码说明

2.1 数据结构说明

（1）利用一维数组结构构建本地棋盘，并注意到 Belongs 删减少于访问，利用集合结构存储棋子的信息，提升了棋盘的访问速度。
（2）利用字典结构在同一个变量中同时存储先手方和后手方的数据。
（3）利用树结构构建决策树。

2.2 函数说明

getWeightedSum---根据位置及权重将所有分值加权

```
def getWeightedSum(self, belong: bool) -> int:
    """
    根据位置及权重将所有分值加权
    :param belong:
    :return:
    """
    weightedSum = 0
    for index in self.belongs[belong]:
        weightedSum += (self.board[index].value * ScoreList[belong][index]) ** self.sumPower
    return weightedSum
```

getWeightedEmpty---获取带权重的空格，空格越集中分值越高

```
def getWeightedEmpty(self, belong):
    """
    获取带权重的空格，空格越集中分值越高
    :param belong:
    :return:
    """
    weightedEmpty = 0
    myNones = self.getNone(belong, getPositionList=False)
    initCol = 0 if belong else 7
    endCol = 3 if belong else 4
    # 对先手方，进行向右和向下的计算，
    # 对后手方，进行向左和向上的计算
    searchDirections = [1, 3] if belong else [0, 2]
    for row in range(4):
        for col in range(initCol, endCol):
            position = getIndex((row, col))
            if position not in myNones:
                continue
            for direction in searchDirections:
                nextPosition = theNext(position, direction)
                while nextPosition in myNones:
                    nextPosition = theNext(nextPosition, direction)
                    weightedEmpty += 1  # 空格相连
    return weightedEmpty
```

evalForGetEaten---在本方的边缘搜索是否会被对方棋子吃掉，用于移动

```
def evalForGetEaten(self, belong) -> int:
    """
    在本方的边缘搜索是否会被对方棋子吃掉，用于移动
    :param belong:
    :return:
    """
    potentialEatCount = 0
    col = 3 if belong else 4
    searchOppDirection = 3 if belong else 2
    searchMyDirection = 2 if belong else 3
    for row in range(4):
        myPosition = getIndex((row, col))
        if myPosition in self.belongs[belong]:  # 存在本方棋子
            myChess = self.board[myPosition]
            nextPosition = self.findNearestChess(not belong, searchOppDirection, myPosition)
            # 在同一行存在对方棋子，且对方棋子数值相同
            if nextPosition and self.board[nextPosition].value == myChess.value:
                # 继续寻找这枚棋子是否存在保护的棋子
                nextPosition = self.findNearestChess(not belong, searchOppDirection, nextPosition)
                if nextPosition and self.board[nextPosition].value == myChess.value + 2:
                    # 存在该保护棋子，该局面扣分
                    potentialEatCount -= (myChess.value * 2) ** self.eatPower
                # 不存在保护棋子，在己方搜索即使被吃，存在可以反吃的棋子
                else:
                    findCanEatBack = False
                    for direction in [0, 1, searchMyDirection]:
                        nextPosition = self.findNearestChess(belong, direction, myPosition)
                        # 己方存在可以反吃的棋子
                        if nextPosition and self.board[nextPosition].value == myChess.value + 1:
                            findCanEatBack = True
                            potentialEatCount += (myChess.value * 2) ** self.eatPower
                            # 停止搜索
                            break
                    # 找不到可以反吃的棋子，扣分
                    if not findCanEatBack:
                        potentialEatCount -= (myChess.value * 2) ** self.eatPower
    return potentialEatCount
```

evalForEaten---与 evalForGetEaten 相同，为搜索在对方区域的己方棋子是否会被吃掉

```
def evalForEaten(self, belong) -> int:
    """
    与evalForGetEaten相同，为搜索在对方区域的己方棋子是否会被吃掉
    :param belong:
    :return:
    """
    potentialEatCount = 0
    col = 4 if belong else 3
    searchOppDirection = 3 if belong else 2
    searchMyDirection = 2 if belong else 3
    for row in range(4):
        myPosition = getIndex((row, col))
        if myPosition in self.belongs[belong]:
            myChess = self.board[myPosition]
            for direction in [0, 1, searchOppDirection]:
                nextPosition = self.findNearestChess(not belong, direction, myPosition)
                if nextPosition and self.board[nextPosition].value == myChess.value:
                    nextPosition = self.findNearestChess(belong, searchMyDirection, myPosition)
                    if nextPosition and self.board[nextPosition].value == myChess.value + 1:  # 己方存在保护棋子
                        break
                    potentialEatCount -= (myChess.value * 2) ** self.eatPower  # 己方不存在保护棋子
                    break
    return potentialEatCount
```

getAvoidPosition---在落子之前先遍历边缘的棋子，是否可能被吃掉,若可能被吃掉，但无法移动躲避，则在对方区域下棋以格挡.

```
def getAvoidPosition(self, belong):
    """
    在落子之前先遍历边缘的棋子，是否可能被吃掉，
    若可能被吃掉，但无法移动躲避，则在对方区域下棋以格挡.
    :param belong:
    :return:
    """
    col = 3 if belong else 4
    searchOppDirection = 3 if belong else 2
    searchMyDirection = 2 if belong else 3
    for row in range(4):
        myPosition = getIndex((row, col))
        if myPosition in self.belongs[belong]:  # 获取一枚本方棋子
            myChess = self.board[myPosition]
            if myChess.value < 4:  # 在数值大于一定级别才考虑指往对方的棋子
                continue
            nextPosition = self.findNearestChess(not belong, searchOppDirection, myPosition)
            if nextPosition and self.board[nextPosition].value == myChess.value:  # 对方存在可以吃掉己方的棋子
                findCanEatBack = False
                # 己方是否存在可以反吃的棋子?
                for direction in [0, 1, searchMyDirection]:
                    nextPosition = self.findNearestChess(belong, direction, myPosition)
                    if nextPosition and self.board[nextPosition].value == myChess.value + 1:  # 己方存在可以反吃的棋子
                        findCanEatBack = True
                        break
                # 不存在可以反吃的棋子
                if not findCanEatBack:
                    # 是否可以移动躲避?
                    for direction in [0, 1, searchMyDirection]:
                        canAvoid = False
                        newBoard = self.copy()
                        if newBoard.move(belong, direction):
                            # 移动之后棋子的位置发生改变
                            if myPosition not in newBoard.belongs[belong] or \
                                    (myPosition in newBoard.belongs[belong]
                                     and newBoard.board[myPosition].value != myChess.value):
                                canAvoid = True
                                break
                    # 无法移动躲避
                    if not canAvoid:
                        # 若可以下棋格挡，则返回格挡的位置
                        if theNext(myPosition, searchOppDirection) not in self.board:
                            return getPosition(theNext(myPosition, searchOppDirection))
    # 所有条件不成立，返回空元组
    return ()
```

MoveSearch---利用 AlphaBeta 搜索，结合 PVS 剪枝，移动方向的递归搜索
（算法核心部分，由于篇幅原因不在报告中展示，详情请见代码文件）

getBestMove---迭代加深，由于每次一层的速率不相同，故设置时间上限，从一层开始逐渐加深

```
def getBestMove(self, currentRound):
    """
    迭代加深，由于每次一层的速率不相同，故设置时间上限，从一层开始逐渐加深
    :param currentRound:
    :return:
    """
    depth = 3
    decisions = Decisions()
    startTime = time()
    while time() - startTime < self.maxTime and depth < self.maxDepth:
        if self.isFirst:
            self.MoveSearch(depth, -INFINITY, INFINITY, 0, currentRound, decisions)
        else:
            self.MoveSearch(depth, -INFINITY, INFINITY, 1, currentRound, decisions)
        depth += 1
    return decisions.getDecision()
```

2.3 程序限制

最终代码在测试及天梯对战时并未出现运行错误的情况。

3 实验结果

3.1 测试数据

实验环境说明：

- 硬件配置：Intel Core i5-4300U/4.00GB
- 操作系统：windows 8.1/专业版
- Python 版本：3.7

我们通过技术组提供的 userinterface 程序，在电脑本地，将不同参数（估值函数和搜索深度）的 AI 相互对战，挑选出不超时（本地个别对局的用时可略大于 5s，其平均用时仍在 5s 内）且胜率较高的 AI。而后对挑选出的 AI 继续调节参数，重复上述操作，选出在本地 AI 中胜率最高者。然后我们将 AI 放在对战平台上，通过热身赛进一步调整参数，并研究输掉的对局情形，考虑启发性算法。

在本地的测试中，我们挑选了具有代表性的两个 AI，1_4 版本和天梯版本，主要对他们运行时间的行为进行测量。（1_4 版本没有考虑阻挡策略，俩版本估值函数略有差别）两个 AI 进行了二十局对战（攻防交替），平均用时结果如下表所示。

AI 版本	进攻方用时 s	进攻方胜率%	防守方用时 s	防守方胜率%
1_4	4.457	90	4.070	50
天梯	4.221	50	3.721	10

天梯版本，经过优化用时减少，但胜率并不如先前版本。这是由于我们临近截止时才想到要增加阻挡的代码，仓促之下参数选取不理想。针对这两个代码表现的不同，我们对下子用时进行了比较，代码和结果如下（进行了十轮先后手交替对战）：

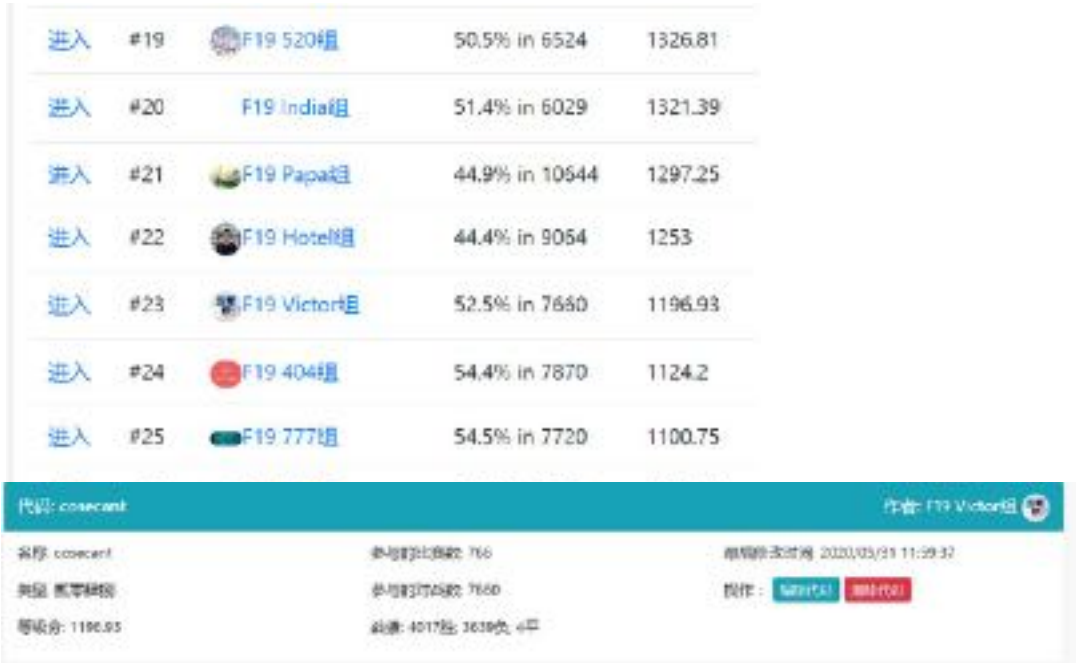
AI 版本	总用时 s	落子用时	胜率%
1_4	4.532	0.0156	60
天梯	4.286	0.0218	40

从上面数据可以看出，落子用时占全部算法用时的千分之四左右，可见大部分用时用在了移动的剪枝树中。针对天梯版本胜率并不如先前版本的问题，我们看到天梯版本增加了阻挡测略后，落子用时有大幅增加，但相较于全部用时仍只是很小的一部分。这说明落子策略对剪枝树所用时间挤占有限，胜率降低的原因可能来自阻挡策略会送给对方子，以及估值函数需要进一步优化。

下面我们以表格形式列出参加热身赛的结果。（AI 版本为热身赛中代码的名称，括号中表示其版本接近本地的哪一版，天梯上版本之间对战的结果未展示）

AI 版本	Bot_B	Bot_A	Bot_S	Bot_SS	总战绩
F1（近最初版）	4：2（66.7%）	34：53（39.1%）	6：4（60.0%）	1：19（0.5%）	192：200（48.9%）
F4（近 1_4）	3：3（50.0%）	14：14（50.0%）	20：17（58.8%）	未对战	146：96（60.3%）
F6（近天梯）	14：16（46.7%）	11：13（45.8%）	未对战	未对战	49：53（48.0%）

最后这里给出天梯排名。



3.2 结果分析

在测试、热身赛和实战中，算法的策略主要如下：算法在移动时，通过剪枝树进行多次多层搜索，估值函数评估，选择最优的移动方向；在落子时，通过判断是否可能被对方吃子，选择是否在吃子路上堵截。算法为了防止超时，在剪枝树中当搜索时间超过阈值时便会停止搜索。

这些策略有如下作用。估值函数中，对盘面棋子分值的估计使得选择趋向于生成更大的子，对空位的估计使得棋盘不易被填满，对合并度的估计使得选择趋向于更易合并，对吃子或被吃子的估计来创造有利条件或避免被吃，这些函数形式权重不同、互相取舍，最终选择更有利的结果。剪枝树与估值函数的设计，是算法的核心部分，也让我们在对局中有不错的表现（天梯胜率 52.5%），但有时会出现过快填满棋盘以致无法合并的情况。落子时的阻挡措施，能有效的防止被吃，但有时也会为对方送子。

我们的算法在运行时间上有很好的管控，使得算法基本不会超时。我们的实现方式如下：一方面我们通过自己实现的棋盘类，大大减少了基本操作的时间开销；另一方面，搜索算法会优先搜索有利的局面，在搜索时间开销大过阈值时停止搜索。

主要的运行时间开销发生在移动时的剪枝树中，随着搜索深度的增加时间开销呈几何级数增加。每一层搜索中，时间主要花费在计算估值函数上，因此我们也对估值函数进行了优化和割舍，舍去了开销时间极大的单调性的计算。

3.3 经典战局（可选）

上天梯前最后的版本胜率很不错，在 12-17 名间徘徊，荣天泽同学发现对局中这版本的 AI 会出现被人吃大子而输掉比赛的情形（下面的例子），于是他建议增加通过在对方棋盘下子阻挡对方吃子的代码。然而增加这部分代码后的 AI 胜率竟不如原 AI。下面给出让他提出此建议的棋局。

16 32 8 16 128 16
2048 2048 64 512 8
64 8 16 16 256 128
8 32 8 8 32 8

16 32 8 16 128 16
2048 2048 64 512 8
64 8 16 8 16 256 128
8 32 8 8 32 8

16 32 8 16 128 16
4096 64 512 8
64 8 16 8 16 256 128
8 32 8 8 32 8

4 实习过程总结

4.1 分工与合作

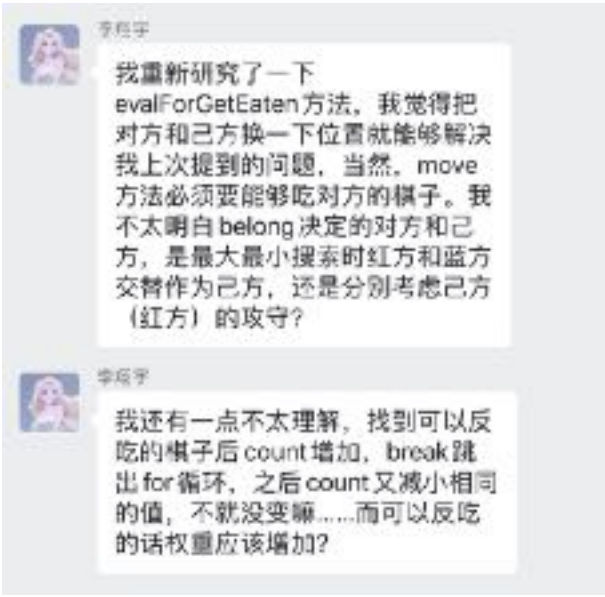
小组里的陈十楚主同学要负责算法的整体设计及代码的主要编写，李翔宇同学主要负责算法的补充设计及评估函数的修改编写，荣天泽同学主要负责代码的线上对局及参数修整，谢添威、喻翰同学主要负责实习报告的撰写。组员在完成自己任务的同时，也会互相帮助协作完成其他成员的任务。

由于疫情隔离在家的原因，小组成员只能通过微信进行线上交流。

【聊天记录里的照片展示】

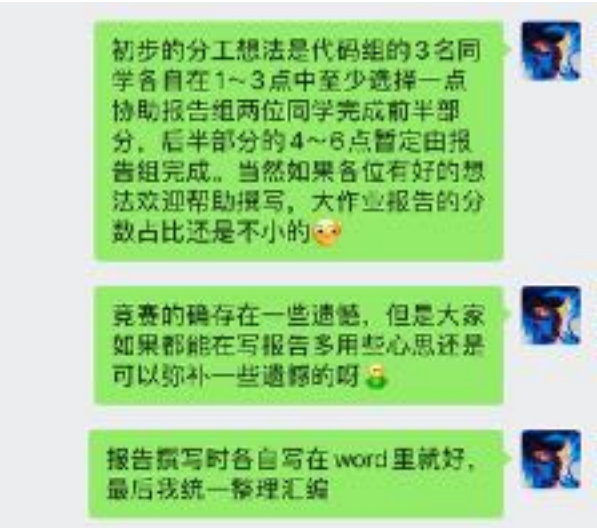
【不断更新迭代的代码文件】

【陈十楚同学编写的代码不断更新迭代】



【李翔宇同学针对算法提出相应的疑问与改进思路】

【代码组的同学进行讨论】



【组长进行任务分配】

4.2 经验与教训

1、实习中可取之处在于组员都参与到讨论中时出现了许多好的想法，对代码完善改进具有一定帮助。

2、小组讨论的高峰期距离截止时间较近，前期讨论不够积极，导致后续的代码改进较为匆忙，有些想法并未能成功实现。

3、在算法设计与代码编写部分主要由一人独立完成，如果能有更多组员参与或许可以使算法更加完善、进度加快，同时也可以减轻编写者的负担。

4. 基于迭代加深，未能有效处理缓存结果，即对已搜索层数结果进行缓存，下次进入不必再次使用 Move 或者 Add，但实际上由于比赛一次时间用时太少，导致复制与排序棋盘的速度并不明显低于操作的速度，故最终舍去. 若比赛时间延长，搜索层数加大，应是不错的选择。

4.3 建议与设想

希望天梯网站上计算积分的机制不受提交代码时间先后的影响，或者采用胜率排名机制。希望大作业的提交时间可以有所调整，避开期末考试的高峰期。

学弟学妹们一定要学好数算课的基础知识，认真对待每一次作业。大作业尽量不要往后拖延，否则后续改进真的手忙脚乱。

5 致谢

感谢陈斌老师本学期的悉心教导！
感谢各位助教的帮助与付出！
感谢技术组同学的辛勤付出！
感谢在群里热心答疑解惑的大佬们！
感谢所有组员为大作业作出的努力！

6 参考文献

1、https://stackoverflow.com/questions/22342854/what-is-the-optimal-algorithm-for-the-game-2048/22498940#22498940 ——单机 2048AI 算法

2、https://baike.baidu.com/item/Alpha-Beta 算法/19460348?fr=aladdin ——Alpha-Beta 搜索算法

3、https://baike.baidu.com/item/窗口搜索/19480022?fr=aladdin ——窗口搜索算法 PV 剪枝

数据结构与算法课程实习作业报告

（郭维豪 支啸渤 胡昊哲 秦明洋 袁源*）

摘要：本组算法利用广度优先策略构建决策树，利用深度优先策略进行剪枝并估值，估值时主要考虑了子力大小和吞并风险。其中加入了启发式规则灵活决定搜索深度、简化了树的构建等等。实际运行与对战中，估值所用时间相对较长，但整体时间把握适当；一些时候吃子进攻能力尚佳，但对平滑性把握较差，容易卡死。

关键字：决策树 剪枝 启发式规则 深度优先 广度优先

1 算法思想.....2
1.1 总体思路.....2
1.2 算法流程图.....2
1.3 算法运行时间复杂度分析.....2
2 程序代码说明.....3
2.1 数据结构说明.....3
2.2 函数说明.....6
2.3 程序限制.....11
3 实验结果.....11
3.1 测试数据.....11
3.2 结果分析.....12
3.3 经典战局（可选）.....13
4 实习过程总结.....14
4.1 分工与合作.....14
4.2 经验与教训.....15
4.3 建议与设想.....16
5 致谢.....17
6 参考文献.....17

1 算法思想

1.1 总体思路

算法的总体思路就是构建并利用决策树给出决策。

决策树的节点主要代表了一种棋面。决策树的功能是对节点（棋面）估值，并根据需要的决策类型给出正确的决策。

算法实现的关键主要是如何高效构建决策树和如何实现合理估值。构建决策树最初的方法是深度优先的迭代构建，缺点是包含了太多无用节点；故之后改为广度优先的迭代构建。估值最初考虑了子力大小、平滑性、单调性、吞并风险，缺点是各估值之间比例不好掌握，实际效果差；实战时只考虑了子力大小和吞并风险，实战效果有所改进。

1.2 算法流程图

1.3 算法运行时间复杂度分析

算法的核心运算部分在于三个函数（详见 2.2），即构建树的方法、估值、剪枝+更新所有节点值这三部分。下面分别考虑：

（1）构建树

先估算节点个数。

构建树的算法是广度优先的。在构建需要给出位置坐标的节点的子节点时，大约会有 5-10 个节点；在构建需要给出方向的节点的子节点时，最多只有 4 个。两层作为一个单位，以叶节点来计数，则 $DEPTH = 2N$ 时，节点构建的时间复杂度为 $O(40^N)$.

再考虑每个节点构建所需要的时间，这当然是与深度无关的，考虑为作 100 单位时间（主要是复制棋盘）。

故这一部分总的时间复杂度为 $O(100 \times 40^N)$.

（2）估值

主要是对叶节点估值，剪枝更新所有节点值时用到了这一函数。对单一节点估值时间当

然相对固定。

估值部分主要有几个函数： `self.updateAllAdjacent([], [False for _ in range(32)])`，`self.updateDanger(pos)` 和 `self.updateBorder()` 以及最终的子力估算函数，前三个只是用来分析棋盘的辅助函数。

但考虑到其中一个函数的迭代算法，估值会用固定但相对长的时间。考虑为：

$32\times3^{16}\approx10^9$ 单位时间。比较夸张，但确实是反映了这一部分的运算量。

（3）剪枝+更新所有节点值

叶节点上我们采用了估值。剪枝深度优先，部分减少了一些节点的计算，原则上会大幅缩减整体运算时间。

整体考虑为 $O(10^9\times10^{-3}\times40^N)=O(10^6\times40^N)$.

可以看到，总的时间复杂度集中在估值部分。

实际运行时间开销的分析详见 3.3，结论符合上面的预期，但估值和建树的时间比值并没有那么悬殊，大约是 100 倍（而不是 10000 倍）。

2 程序代码说明

2.1 数据结构说明

算法中主要采用的是**树**这一数据结构；为了实现广度优先的建树方法，又借用了**队列**这一基本数据结构。

自定义了**决策搜索树 decisionTree** 和**节点 TreeNode** 两个类，用来辅助实现 **Player 类**。

决策搜索树 decisionTree 是形式上实现决策的结构主体，代码如下：

```
class decisionTree:
    def __init__(self, isFirst: bool, array: list, DEPTH, mode, currentRound, stage, currentState):
        self.isFirst = isFirst # 自己是先手还是后手
        self.root = currentState # 树的根节点是由初始状态生成的节点，作为参数传入
        self.DEPTH = DEPTH  # 决策树目标深度（建完之后就是决策树的深度，根节点所处深度为 0）
        self.mode = mode # 决策类型
        searchQueue = [self.root] # 搜索队列
        self.createTree(currentRound, stage, searchQueue) # 初始化创建决策树
        self.valueTree() # 初始化对节点估值
    # 1. 开始构建决策树
    def createTree(self, currentRound, stage, searchQueue):
        self.root.createTreeNode(self.DEPTH, currentRound, stage, searchQueue)
    # 2. 将估值函数应用于叶子结点
```

```
    # 3. 深度优先搜索，传递并更新 α、β、结点值
    #  Max 结点更新 α 值(下限)
    #  Min 结点更新 β 值(上限)
    def valueTree(self):
        # 调用节点的 countValue 方法，按合理顺序遍历所有节点并标注值，过程中进行 α、β 剪枝，剪枝的部分当然不必再进行标注
        self.root.valueTreenode(-2147483647, 2147483647)

    # 4. 从根结点选择评估值最大的分支，作为行动策略
    def makeDecision(self):
        # 利用已经建构的决策树，给出本步的下子位置或合并方向
        bestChoice = None
        maxscore = -2147483647
        for node in self.root.children:
            if node.value > maxscore:
                bestChoice = node
                maxscore = node.value
        # 用 getDecision 的话要注意创建节点时 update 棋盘
        if self.mode == 'position':
            return bestChoice.board.getDecision(self.isFirst)
        else:
            return bestChoice.board.getDecision(self.isFirst)[0]
```

其中主要的方法实现了决策的整个过程：创建决策树→叶节点估值&α-β剪枝→给出决策。由于搜索树无需保持节点值的顺序排列、树的平衡等性质，故没有 BST 或 AVL 的相关方法。

节点 TreeNode 是实际实现决策树构建的对象，决策树中的方法实现基本上是靠根节点的方法。其部分代码如下：

```
class TreeNode:
    def __init__(self, board: Chessboard, isFirst, mode, currentRound):
        ROWS = 4  # 行总数
        COLUMNS = 8  # 列总数
        self.board = board # 节点储存的棋盘
        self.value = 0  # 节点估值
        self.depth = 0  # 节点在决策树中深度
        self.isFirst = isFirst  # 节点对应棋盘该哪一方操作
        self.mode = mode # 节点对应的操作方的操作类型
        # 先后手（哪一方）isFirst 和下一步所做操作 mode 共同决定了节点（及其估值）
        # 这是因为，对象不同，操作不同，所做的估值方法是不同的
        self.currentRound = currentRound # 棋盘对应轮数
        self.children = [] # 子节点
        self.parent = None # 父节点
        self.grids = [{'number': 0, 'belong': None, 'adjacent': 0, 'in_danger': []} for _ in range(ROWS * COLUMNS)]
```

```
16.         # 每个格子的信息(包括所含数字，归属，包括自身相连的格子数，以及吞并的危险来源（上下左右）),
17.         # 形如 {'number':16, 'belong':True, 'adjacent': 13, 'in_danger':['right']}
18.         self.border = []
19.         # 记录先后手方的边界格子，内部包含两个列表，分别记录每行先后手边界的列值
20.
21.     def addChild(self, node):  # 新增子节点 node
22.         self.children.append(node)
23.         node.parent = self
24.
25.     def getValue(self):
26.         return self.value
27.
28.     def createTreeNode(self, DEPTH, currentRound, stage, searchQueue):
29.         # 建构所需深度决策树（即所有节点）;
30.         # 建构时根据根节点的 mode 状态（下一步所作操作）和所需深度，依次添加做所有可能操作的子节点
31.         # 建构是广度优先搜索，因为这样可以更方便的剔除一些垃圾子节点
32.
33.     def searchAdjacentGrids(self, singlePosition, belong, memory, searched):
34.         # 递归的搜索目前格子所在连通分支的格子
35.         # 是 updateAllAdjacent 方法的辅助函数
36.
37.     def updateAllAdjacent(self, memory, searched):
38.         # 更新棋盘每个格子所在连通分支的大小
39.
40.     def updateDanger(self, pos):
41.         # 更新是否有被吞并的危险
42.
43.     def updateBorder(self):
44.         # 更新边界
45.         # 边界满足三个条件（以先手方为例）：1 是连通的 2 下一位属于后手方或到头 3 满足以上两点的最大值
46.
47.     def sumOfPositionValue(self, isFirst):
48.         # 空间分布估值（含单调性）（实际估值时因为时间原因没被使用）
49.
50.     def sumOfFlatnessValue(self, isFirst):
51.         # 平滑性估值  （实际估值时因为时间原因没被使用）
52.
53.     def sumOfChessValue(self, board, belong, currentRound):
54.         # belong 方目前子力大小估值
55.         # 如果某子落单或者要被吃掉就会子力削弱
```

```
56.
57.     def countValue(self, isFirst, mode):
58.         # 计算总估值
59.
60.     def valueTreenode(self, alpha, beta):
61.         # 对以此节点作为根的子树中所有节点估值
```

其中核心的方法只有三个：

1. createTreeNode：用来从此节点开始构建树；

2. countValue：用来给出节点棋盘的估值

3. valueTreenode：用来从此节点开始更新树中部分节点的估值，其中实现了剪枝（即部分节点估值不必计算）。

而其他的方法则主要是一些辅助方法：

1. searchAdjacentGrids：用于递归的搜索目前格子所在连通分支的格子，是 updateAllAdjacent 的辅助函数
2. updateAllAdjacent：更新所有格子所在连通分支的大小
3. updateDanger：更新当前所考虑的棋子是否有被吞并的危险
4. updateBorder：更新边界
5. sumOfPositionValue：计算某方的单调性函数
6. sumOfFlatnessValue：计算某方的平滑性函数
7. sumOfChessValue：计算某方的子力大小

其中 1-4 是更新棋盘信息的辅助方法，5-7 是用于估值函数的辅助方法，但由于时间关系，5 和 6 实际上没有被调用，估值时就只考虑了子力大小因素。

方法的具体实现参见 2.2.

2.2 函数说明

（1）output 函数

Player 类中的核心函数。这里主要考虑了分阶段决定搜索深度（由于时间关系还不完善），利用相应深度的的搜索树进行决策。代码如下：

```
1.  def output(self, currentRound, board: Chessboard, mode):
2.      boardcopy = board.copy()
3.      timeLeft = board.getTime(self.isFirst)
4.      if mode == 'position' or mode == 'direction':  # 按照剩余时间选择决策深度
5.          #将游戏分阶段（由于时间关系还不完善）
6.          stage = [20]
7.          #在构建决策树之前可以利用启发式规则先解决一些一本道的情况（由于时间关系没有考虑）
8.          currentState = TreeNode(board, self.isFirst, mode, currentRound)
9.          raw = board.getRaw()
10.         #根据当前回合来决定搜索深度
11.         def determineDepth(currentRound):
12.             if currentRound <= stage[0]:
13.                 return 2
14.             else:
```

```
                return 4
        DEPTH = determineDepth(currentRound)
        DTree = decisionTree(boardcopy, self.isFirst, self.array, DEPTH, mode
    , currentRound, stage, currentState)
        return DTree.makeDecision()
    return

```

（2）TreeNode 中的 createTreeNode 方法

实现了广度优先的树的建构。其中，每层深度的建构只把值得考虑的节点加入，排除了明显过分考虑在对手方加入棋子的情况（启发式算法）。代码如下：

```
def createTreeNode(self, DEPTH, currentRound, stage, searchQueue):
        # 建构所需深度决策树（即所有节点）；
        # 建构时根据根节点的 mode 状态（下一步所作操作）和所需深度，依次添加做所有可
    能操作的子节点
        # 建构是广度优先搜索，因为这样可以更方便的剔除一些垃圾子节点

        #先搜索比较好的走法,添加到列表 childrenToAdd 里面
        childrenToAdd = []
        newBoard = self.board.copy()

        if self.isFirst:
            if self.mode == 'position':#先手下棋
                Position = newBoard.getNext(True, self.currentRound)  # 通过
    随机序列确定己方下棋位置
                if Position != ():  # 确定是合法位置
                    newBoard.add(True, Position, 1)
                    newBoard.updateDecision(True, Position)
                    newNode = TreeNode(newBoard, False, self.mode, self.curre
    ntRound)  # 下一步是对方，mode 不变
                    newNode.depth = self.depth + 1
                    childrenToAdd.append(newNode)

                lst = self.board.getNone(False)  # 获取对方四列下棋位置
                if childrenToAdd == [] or len(lst) <= 2:#不能在自己空里下棋，目
    前就把所有在对方空里下棋的情况先全部加上吧:(
                    #另外对方空格不多于 2 时也把所有情况都加上
                    if lst:  # 确定是合法位置
                        for Position in lst:
                            newBoard = self.board.copy()
                            newBoard.add(True, Position, 1)
                            newBoard.updateDecision(True, Position)
                            newNode = TreeNode(newBoard, False, self.mode, se
    lf.currentRound)  # 下一步是对方
                            newNode.depth = self.depth + 1
```

```
                childrenToAdd.append(newNode)
            elif currentRound > stage[0]:#可以在自己空里下棋，选择性的加一些在对手空里下棋的情况
                pass     #因为时间关系来不及加了

        elif self.mode == 'direction':#先手合并
            goodMoveDetected = False  #是否找到能吃掉对手最大棋子的 move
            oppLargest = self.board.getScore(False)[-1] if self.board.getScore(False) != [] else 0 #对方最大的棋子等级
            for Direction in [0, 1, 2, 3]:  # 分别向三个方向移动
                if newBoard.move(True, Direction):  # 确定是合法移动
                    newBoard.updateDecision(True, (Direction,))
                    newNode = TreeNode(newBoard, False, self.mode, self.currentRound)  # 下一步是对方，mode 不变
                    newNode.depth = self.depth + 1
                    oppLargestAfter = newBoard.getScore(False)[-1] if newBoard.getScore(False) != [] else 0#获取移动后对方最大的棋子等级
                    if oppLargestAfter < oppLargest:#能吃掉对手最大的棋子
                        if not goodMoveDetected:
                            goodMoveDetected = True
                            childrenToAdd = []
                        childrenToAdd.append(newNode)
                    else:#不能吃掉
                        if not goodMoveDetected:
                            childrenToAdd.append(newNode)

                    newBoard = self.board.copy()

    else:
        if self.mode == 'position':#后手下棋
            Position = newBoard.getNext(False, self.currentRound)  # 在己方下棋
            if Position != ():  # 确定是合法位置
                newBoard.add(False, Position, 1)
                newBoard.updateDecision(False, Position)
                newNode = TreeNode(newBoard, True, 'direction', self.currentRound + 1)  # 下一步是对方，mode 改变
                newNode.depth = self.depth + 1
                childrenToAdd.append(newNode)
            lst = self.board.getNone(True)
            if childrenToAdd == [] or len(lst) <= 2:#不能在自己空里下棋，目前就把所有在对方空里下棋的情况先全部加上吧:(

```

```
                    if lst:
                        for Position in lst:
                            newBoard = self.board.copy()
                            newBoard.add(False, Position, 1)
                            newBoard.updateDecision(False, Position)
                            newNode = TreeNode(newBoard, True, 'direction', self.currentRound + 1)
                            newNode.depth = self.depth + 1
                            childrenToAdd.append(newNode)
                elif currentRound > stage[0]:#可以在自己空里下棋，选择性的加一些在对手空里下棋的情况
                    pass  #由于时间关系来不及加了


            elif self.mode == 'direction':#后手合并
                goodMoveDetected = False
                oppLargest = self.board.getScore(True)[-1] if self.board.getScore(True) != [] else 0 #对方最大的棋子等级
                for Direction in [0, 1, 2, 3]:  # 分别向三个方向移动
                    if newBoard.move(False, Direction):  # 确定是合法移动
                        newBoard.updateDecision(False, (Direction,))
                        newNode = TreeNode(newBoard, True, 'position', self.currentRound + 1)  # 下一步是对方，mode 不变
                        newNode.depth = self.depth + 1
                        oppLargestAfter = newBoard.getScore(True)[-1] if newBoard.getScore(True) != [] else 0#获取移动后对方最大的棋子等级
                        if oppLargestAfter < oppLargest:#能吃掉对手最大的棋子
                            if not goodMoveDetected:
                                goodMoveDetected = True
                                childrenToAdd = []
                            childrenToAdd.append(newNode)
                        else:#不能吃掉
                            if not goodMoveDetected:
                                childrenToAdd.append(newNode)

                        newBoard = self.board.copy()
        #搜索下一个节点
        for _ in childrenToAdd:
            self.addChild(_)
        if DEPTH > 1:
            searchQueue += childrenToAdd
        if searchQueue != []:
            searchQueue.pop(0)
```

```
104.             if searchQueue != []:
105.                 d = DEPTH if (searchQueue[0].mode == self.mode and searchQue
    ue[0].isFirst == self.isFirst) else DEPTH - 1
106.                 searchQueue[0].createTreeNode(d, searchQueue[0].currentRound
    , stage, searchQueue)
```

（3）TreeNode 中的 countValue 方法

实现了对确定棋面进行估值。由于调参不力，最后采用的版本没有考虑太复杂，只考虑了双方子力大小（其中包含了检查是否有吞并危险）。实际上我们原先写了针对平滑性、空间分布（单调性）的估值函数，即 sumOfFlatnessValue，sumOfPositionValue 等。

部分代码如下：

```
1.     def countValue(self, isFirst, mode):
2.         # 计算总估值
3.
4.         self.updateAllAdjacent([], [False for  _ in range(32)])
5.         for pos in range(32):
6.             self.updateDanger(pos)
7.         self.updateBorder()
8.         return self.sumOfChessValue(self.board, isFirst, self.currentRound) -
     self.sumOfChessValue(self.board, not isFirst, self.currentRound)
```

（4）TreeNode 中的 valueTreenode 方法

实现了剪枝和对树中需要节点的估值更新。代码如下：

```
1.  def valueTreenode(self, alpha, beta):
2.      # 对以此节点作为根的子树中所有节点估值
3.
4.      if self.children == []:
5.          self.value = self.countValue(self.isFirst, self.mode)
6.          return self.value
7.      if self.depth % 2 == 0:
8.          for i in self.children:
9.              alpha = max(alpha, i.valueTreenode(alpha, beta))
10.             if beta <= alpha:
11.                 break
12.         self.value = alpha
13.         return alpha
14.     else:
15.         for i in self.children:
16.             beta = min(beta, i.valueTreenode(alpha, beta))
17.             if beta <= alpha:
18.                 break
19.         self.value = beta
20.         return beta
```

（5）其他辅助方法的部分代码：

辅助方法中最重要的就是计算子力大小。计算子力大小时主要考虑以下两个因素：

① 由于一个 2048 比两个 1024 更好，所以（2**n）的子力大小被视为（2**n）**exp

（exp > 1）

② 当有棋子被孤立（这里连通分支大小小于一个阈值认为被孤立）或面临被吞并的危险时子力大小削弱。

代码如下：

```
def sumOfChessValue(self, board, belong, currentRound):  # 算出某玩家 belong 目前子力大小，如果某子落单或者要被吃掉就会子力削弱
    S = 0  # S 代表子力大小
    ROWS = 4  # 行总数
    COLUMNS = 8  # 列总数
    # 2 的 n 次方的子力应当被认为是(2 ** exp)的 n 次方
    exp = 1.5
    # 所在连通分支格子数小于 minGrid 判定为落单
    minGrid = 5
    # 落单的子的子力乘以 lonely
    lonely = 0.5
    # 要被吃的子的子力乘以 will_be_eaten 中的对应元素
    # True 是己方操作，False 是对方操作
    will_be_eaten = {(True, 'direction'): 0.5, (False, 'direction'): -2, (True, 'position'): 0.5,  (False, 'position'): 0.5}
    for _ in range(32):
        if self.grids[_]['belong'] == belong and self.grids[_]['number']:
            S += (self.grids[_]['number'] ** exp) * \
                 (will_be_eaten[(self.isFirst == belong, self.mode)] if self.grids[_]['in_danger'] != [] else 1) \
                 * (lonely if self.grids[_]['adjacent'] < minGrid else 1)
            # 加上该子的子力，考虑落单以及即将被吃两种情况
    return S
```

2.3 程序限制

在某些情况下（敌我方都没有过分紧逼，回数极多时）可能超时。

3 实验结果

3.1 测试数据

实验环境说明：

- 硬件配置：（CPU/内存）i5-1035G1
- 操作系统：（名称/版本）Windows 10
- Python 版本：（版本号）python 3.7
- 运行平台：天梯

下面是对于热身赛不同阶段的胜场胜率情况

阶段	总场数	胜率
全阶段	5270	27.42%
初阶段（取样）	200	<5%
中阶段（取样）	200	25.64%
终阶段（取样）	200	37.50%

天梯等级分：598.37

下面是各分段的主要出现的问题：

阶段	取样场数	战败场数	因报错战败	因超时战败	因结果结算战败	平均运行时间
初阶段（建树+傻子估值方法）	100	100	0	0	100	0.020
中阶段（初步引入估值与决策）	100	76	0	32	44	4.385（有个别局数出现 0.03 的平均耗时，是调试中的 bug）
终阶段	100	74	1	19	54	2.815

初阶段仅仅是建树阶段，并不具有参考价值，因此预估实际的胜率大概在 30%左右。

在中阶段的测试中发现，程序有以下两个主要问题：1.超时较频繁，2.送子给对方的倾向强。最终的版本在一定程度上优化了超时问题，但是在保护自家棋子方面依旧有所欠缺：在终阶段中，超过 50%的对局有己方最大或第二大的棋子被对方吞并。

由于这种代码无法用 timeit 具体测量运行时间，只能从比赛时间进行对比分析。我们发现，构建树的耗时平均在 0.01 秒量级上，而进行估值的平均耗时达到 1 秒的量级，说明主要时间耗费还是在估值函数上。

3.2 结果分析

我们初期采用的算法就是基本的 minmax 决策树深度搜索加 alphabeta 剪枝算法。但是在之后的试验和热身战中发现会因为超时的问题限制搜索的层数，严重影响算法的博弈能力。

我们也怀疑过可能是剪枝的问题，但通过测试发现 alphabeta 剪枝对于赋值速度的提升是显而易见的。如下表所示为搜索层数为 8 层时，调用 timer 使 n=100 时得到的对比数据。

使用 alphabeta 剪枝的时间（s）	11.994949	12.809693	14.040139	11.904696
不使用剪枝的时间（s）	43.018403	36.246153	38.074773	35.215884

接下来我们认为找到更好的提高效率的方法首先要明白在程序将大部分的时间花在哪一方面，我们将整个算法分为建立树和赋值两部分分析。用于上相同的方法进行测试

完整的算法所用的时间（s）	12.096859	11.928642	12.843106	14.338069
只完成建树所用的时间（s）	0.226653	0.221071	0.227402	0.232180
计算（两者相减）得到的赋值所用时间（s）	11.870206	11.7075571	12.615704	14.105889

由上表，可以看出我们绝大部分时间都用在了给节点赋值上，由于已经采用了有效的剪枝算法，我们只能认为是由于对子节点使用估值函数消耗的时间过长，这可能是由于我们在估值函数上使用大量的判据造成的，我们不仅仅考虑了单个棋子的价值判断，更加上了关于平滑性和单调性等带有启发式意味的判据，但我们相信经历适当的调参，我们关于这方面的努力是有意义的，更详细的估值函数应当会有效提高 AI 的博弈能力。

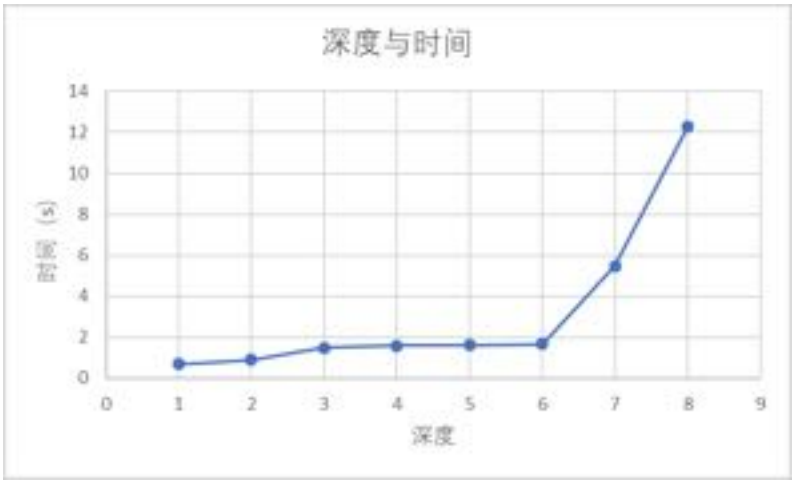
所以我们换角度考虑。决定进一步减少子节点的数量，从而达到提速的目的，经过讨论，最终决定在建树时使用启发式算法对节点进行筛选，使子节点减少。

但迫于 ddl，我们对于启发式算法的工作并没有完全完成，留下了不少的遗憾，但我们也已经总结出了思路：

在 mode='position'时破坏对方棋子的单调性，在对方格子还剩一格（后手方）或两格（先手方）且对方被填满之后无法移动，阻止对方大数合成等；

在 mode='direction'时先排除移动完成之后能被对方立即被吞并的情况如果全部方向移动后均会被吞并则选择损失最小的方向优先考虑剩下情况中自己能吞掉对方大数的情况等

在天梯赛落幕之后，对算法对于深度的时间复杂度做出一点验证：



深度小于 6 时增长不明显初步认为是完成的启发式算法起到了作用。

3.3 经典战局（可选）

本算法在对抗一些比较保守的小组中展现出了很强的攻击性。（中期算法）

但是收官能力较差，经常被“反杀”（平滑性等较差）。有一次比赛刚刚进行三分之一本方的格子就被填满，只能任凭对方摆布。

4 实习过程总结

4.1 分工与合作

分工是分阶段安排的，具体如下：

（1）前期：

主要是搜索资料（2048 好的算法、思路，需要的算法代码等），思考基本策略（思路层面）和实现方案（算法方面）。

分工没有细分。

（2）中期：

主要是集中攻克算法，试运行，debug，讨论，竞赛。

第一阶段具体分工如下：（对于 1.0 版本，即雏形版本）

1. 整体框架搭建与 debug：袁源（组长）；

2. 构建决策树部分：秦明洋，具体是 decisionTree 的 def createTree 方法；

3. 估值函数部分：郭维豪、支啸渤（郭维豪先写，再互相讨论，支啸渤可以负责一部分），具体是 TreeNode 的 def countValue 方法；

4. 搜索赋值剪枝部分：胡昊哲，具体是：decisionTree 的 def valueTree 方法。

第二阶段具体分工如下：（对于 2.0 版本，即加入了启发式算法）

调试组：郭维豪 支啸渤 袁源

通过运行程序发现可以改进的地方，反馈给代码组

代码组：胡昊哲 秦明洋

完成调试组提供的启发式方法代码，分阶段、特定场合决策；提前剪枝、优化时间

实际操作上，郭维豪完成了代码最后的 debug，并独自完成了较高效率的广度优先建构树的方法（从而没有采用代码组提供的代码）。

（3）后期：整理结果，完善报告

报告内容安排如下：

袁源：1.1 总体思路 1.2 算法流程图 1.3 算法运行时间复杂度分析 2.1-2.2 4.1 分工与合作 4.2/4.3 并负责编辑排版

郭维豪 & 支啸渤：2.1-2.3 只需最后给出整理好的完整代码，带有注释和程序限制说明 4.2-4.3

胡昊哲 & 秦明洋：3.1-3.3 尝试对代码效率进行测试，给出实验结果与分析 4.2-4.3

其中 4.2-4.3 是全组必做内容，主要是总结反思收获，发表感想。

交流方式主要通过微信以及北大网盘的共享文件夹，以及通过腾讯会议召开的组会。很遗憾，只有最后一次组会想起来留下了照片：

4.2 经验与教训

胡昊哲：

本次实习可能最大的问题还是出在交流不够方面吧，写完之后才交流导致可能其他人读起来费劲就会出现很多人的工作白费，吃力不讨好的情况，编程的越多，的确分担了一些任务，但总体上还是增大了工作量的。

秦明洋：

在本次实习过程中，得意之处在于每个组员或多或少都分到了编写代码的任务，即每个部分都有特定的人编写。这样的好处是每个方面都有认识非常深刻的人，可以让整体代码的质量提高；但缺点在于大家写的代码在向一起拼接时产生困难，需要繁复的 debug 和整合过程，在一定程度上降低了效率。这方面可以加以改进，加强组员间的联系以提高效率。

郭维豪：

在本次实习过程中，我认为得意之处在于组长在每个阶段都进行了具体的分工，这样的话每个人都有明确的任务要做。通过这种方式我们可以把大作业拆分成一个个明确的小目标，这样也可以在一定程度上提高效率。

至于缺点，我认为我们交流不充分应该是一个比较大的问题。我们当时的分工是每个人

写一部分代码，如果拼接成的代码出了问题，那么也不知道谁的代码出现了问题，这样 debug 需要很长的时间。我认为可以每个人写好代码后再创造一个 word 文档，来详细说明代码中用到的接口和变量，这样可读性也会大大提高。另外我认为天梯赛开始时就不需要每个人都写代码，一两个人写代码就行了，其他人可以进行一些时间测试等工作，来提出改进建议。

支啸渤：

在这次合作中产生了两个问题引起了我的思考。一是团队协作时产生的问题。编程人员不宜太多，至多两个足以。人员太多意味着所有人都要去熟悉代码，一方面消耗了本不必要的时间，另一方面每个人编程的思路不一样，设置的接口有所出入，实现的层次不同，如果所有人都编辑代码也会造成混乱的局面。而和别人合作编写代码时也要先商量好整体代码的架构，保证每人对于全局有很好的掌控。还有就是要加详细的注释，方便他人阅读。二是我自己在编写时发现的问题。在代码较多，类的方法较多时应该记录每个方法的输入及返回值，避免想当然引起 bug。还有 debug 的工作应该在编写过程中进行，每完成一个板块的编写，就去看看是不是有 bug，在最后 debug 错误比较错综复杂不易找到。

袁源（组长）：

之所以在安排上很大程度地均分了任务，是因为如果没有具体安排任务，基本等于没有任务。在不具体知道任务之前，组员基本不可能去完成任务，这是对于组长推动整个项目进行面临的挑战。为了克服这一困难，组长必须首先想清楚任务怎么执行，并且详细地拆分任务，设置 DDL，才有可能使得项目往前迈进。

这样做的好处也有不少，比如每个人都可以切身体验到编写代码的过程，并且将代码分包有利于后续责任落实到人、可以集思广益等等。坏处如上面各位同学提到的，整合困难，需要沟通的反复性大、效率并不高，可能出现做无用功的情况。

当然，或许还有更好的处理方式。老师推荐的少数人写代码主要适用于存在少数人时间时间充裕并且热爱书写代码工作的情况，这在考虑到有组员当时进行转院工作的情况下是不适用的。

4.3 建议与设想

胡昊哲：

寄语：妙脆卷，卷到妙极，但口感确实好

秦明洋：

寄语：这是一门综合性与实用性都很强的课程，任务量着实不少，但高任务量也意味着高回报。希望学弟学妹们能够以学到东西为觉悟，用心学好这门课。

郭维豪：

寄语：这门课的确是一门事情比较多的课，但是能够学到很多有用的知识。同时，会有一些趣味作业，当你认真花了几个整天去完成这些事情，最后是真的会有一种成就感的。

支啸渤：

寄语：这门课课程任务比较多，面授和 mooc 结合的上课方式也比较新颖，认真学习这门课能收获不少知识，并能提高编程、团队协作等能力。

袁源：

寄语：工业 4.0 的时代，所有人都应该对数据结构与算法有一定了解。

5 致谢

在 debug 的时候咨询过同在竞赛场上的两位同仁过往的经验，在此一并感谢@费诺福特@Berk 村民。

感谢活泼可爱的技术组和老师，使我们感到大作业实习无限的快乐。

6 参考文献

1. What is the optimal algorithm for the game 2048?, stackoverflow. https://stackoverflow.com/questions/22342854/what-is-the-optimal-algorithm-for-the-game-2048/22498940

2. 什么是游戏 2048 的最佳算法，CSDN. https://blog.csdn.net/kimylrong/article/details/21967869

数据结构与算法课程实习作业报告

F19X 组：曹顺顺* 陈风宇 孙宇辰 和冬华 陆帅甫

摘要：我们的贰零肆捌 AI 基于极大极小树和剪枝算法，通过计算空位数和棋子值的得分获取评估函数评估值。AI 搜索一定层数之后得到落子点。我们在学习了一系列知识并多次试验和改进后获得了发挥比较稳定的 AI，并在运行速度上得到了提升。

关键字：极大极小树、剪枝算法、2048、MCTS、Alpha Go、CNN

1 算法思想.....1
1.1 总体思路.....1
1.2 算法流程图.....3
1.3 算法运行时间复杂度分析.....4
2 程序代码说明.....4
2.1 数据结构说明.....4
2.2 函数说明.....5
2.3 程序限制.....6
3 实验结果.....6
3.1 测试数据.....6
3.2 结果分析.....7
3.3 经典战局.....8
4 实习过程总结.....10
4.1 分工与合作.....10
4.2 经验与教训.....13
4.3 建议与设想.....14
5 致谢.....14
6 参考文献.....14

1 算法思想

1.1 总体思路

游戏回顾：本次的 2048 对战游戏本质上是一个完全信息的零和博弈，即对局双方都知道关于棋局的所有信息并且可查询历史记录，博弈中只需要使对手收益最小化，自己的收益即可实现最大化。游戏的首要目的在于：在当前棋盘状态下，合并出更多更大的棋子。加入对抗性后，实现对对方棋子的吞并和堵截也成为玩家需要特别考量的重点，此外，游戏中以超时为首的意外结束情况，使胜负判定变得更复杂，对代码的性能也提出了更大的考验。

算法思路：在完全信息零和博弈的条件下，结合回合制的特点，我们将 minmax 决策树作为

整个算法的主体，利用 Alpha-Beta 剪枝节省搜索时间，完成了对 minmax 树的优化。我们将 minmax 树内置于模拟对战的函数中，辅以统计双方棋子数及价值的评估函数，完成整个博弈决策树的构建。考虑到算法的时间限制，我们将搜索的最大层数设为 6，尽量多地采用了位运算以及构建足够简单且高效的评估函数以缩短运算时间，最终我们选择了简单地统计棋子个数及价值的评估函数。在决策树中，评估函数最先应用于终局的叶子结点，随后自底向上计算每个结点的 minimax 值，最后我们从根结点选择 minimax 值最大的分支，作为行动策略。

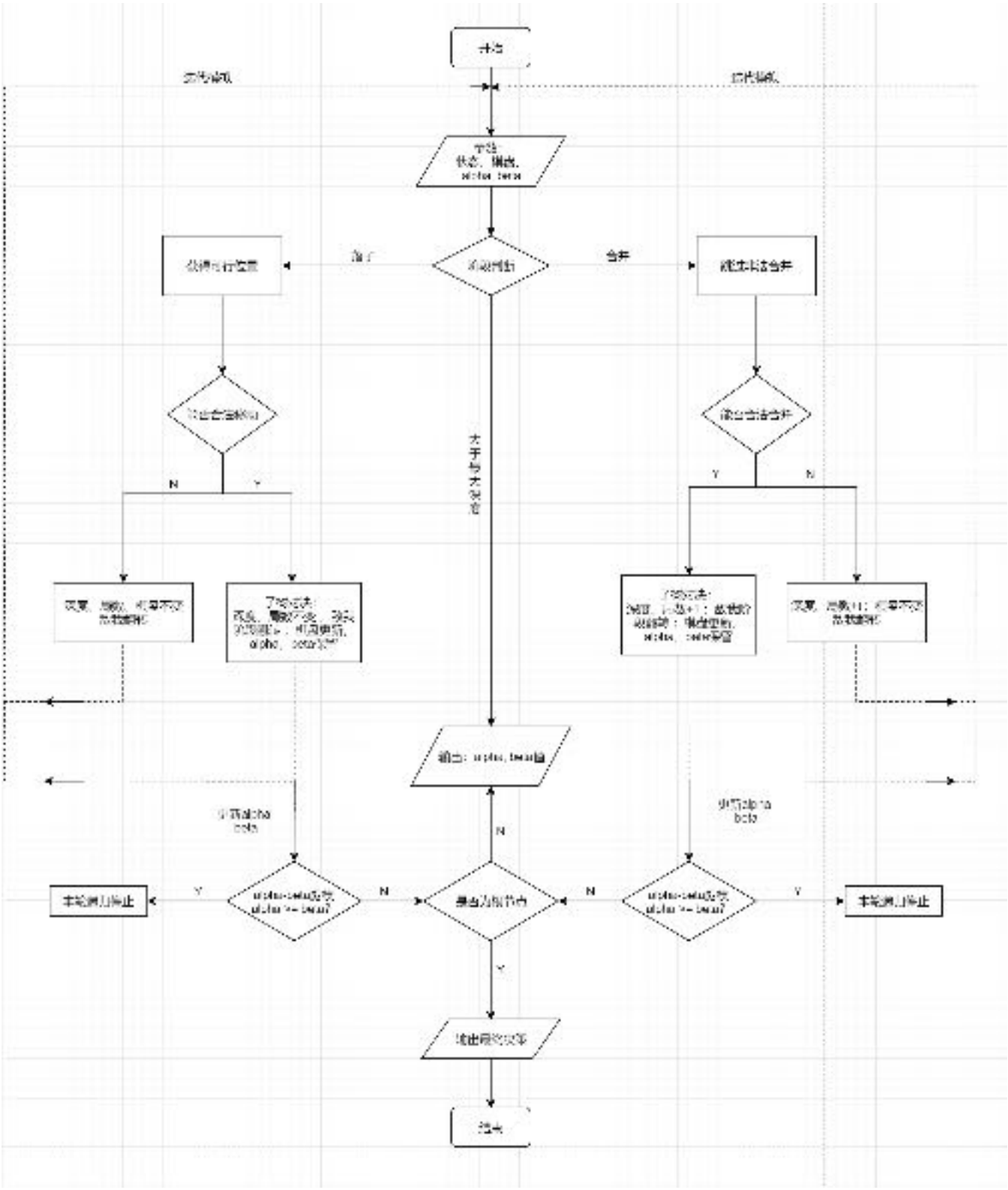
编写过程：

前期：在组长曹顺顺和组员孙宇辰的带领下，全组从 5 月 13 日开始一起了解学习极大极小算法、Alpha-Beta 剪枝算法、蒙特卡洛搜索算法等与作业相关的算法，最终组内确定以 Alpha-Beta 剪枝优化的 minmax 搜索树算法为基础算法参加比赛。由孙宇辰同学完成基础算法后，同学们开始学习 Alpha Go 算法并尝试通过蒙特卡洛树结构、遗传算法炼丹等方式实现更强的 AI。

中期：基础算法开始在天梯运行，取得了较为可人的成绩，相关工作以调参和评估函数的修改为主。与此同时，为了在激烈的竞争中胜出，我们开始向更强大的算法方向开拓。孙宇辰和陆帅甫同学各自完成了自己的 MCTS 结构，然而在后续测试中，我们发现在时间紧凑时，蒙特卡洛树并不能发挥比极大极小树剪枝更好的效果，该尝试作罢。在代码正式提交前的最后几天，孙宇辰同学开始尝试遗传算法炼丹以及卷积神经网络的应用。

后期：初次代码提交后，同学们的目光聚焦回基础算法的优化上，大家努力寻找更好的评估函数，利用更多的启发式规则，节省出更多的时间。在原有评估函数的基础上，大家开始大规模的调参实验，由于最终改观不大也再次作罢。之后，我们尝试从 MIT 的一个传统 2048 大师级 AI 上学习更快的运算方法和启发式规则，受限于 C++代码的阅读障碍及紧凑的时间，最终未能在本土代码上实现。终止提交前，组内再次进行了多次优化实验，我们最终提交了表现最稳定且组内天梯得分最高的基础算法代码。

1.2 算法流程图



1.3 算法运行时间复杂度分析

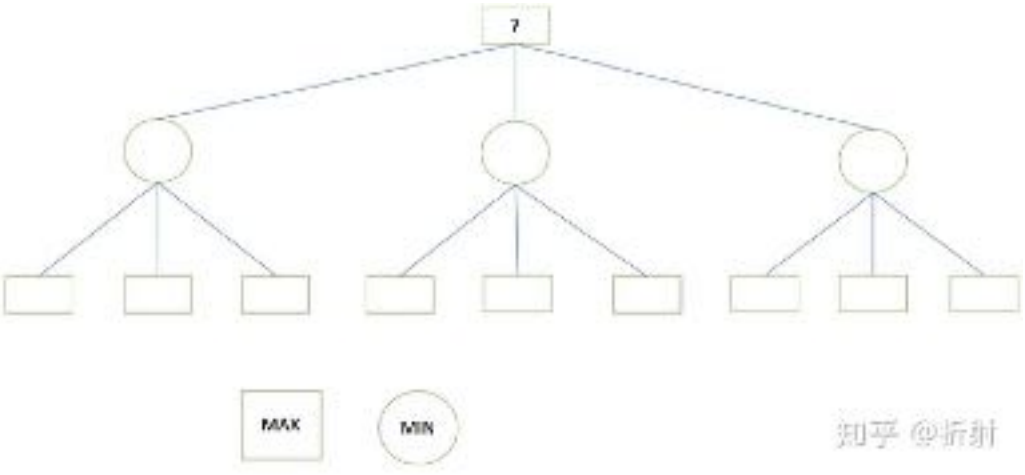
评估函数为两次单重的迭代运算，时间复杂度应为 O(n)。在极大极小搜索过程中，随着 AI 思考层数的上升，时间复杂度呈指数级增长。我们将最大搜索深度设为 6，则其时间复杂度约为 $O(p^6)$，p 为某一常数。理论上来说，经过排序的剪枝算法可以剪掉一半的枝，使得思考层数可以翻倍，但在随机情况下，我们无法得到具体的时间复杂度，但可以肯定其<= $O(p^6)$。

算法实现过程中的递归调用某种程度上也增加了多余的时间负担，最终我们的代码属于较为耗时的类型，在某些情况下仍比较容易超时。

2 程序代码说明

2.1 数据结构说明

1. 主要数据结构：极大极小树。(体现在代码的 simulation 函数中)　　极大极小树是策略类算法中常用的结构，是课程中树结构的一个变体。极大极小树中的每一层交替代表了我方落子收益和对方落子收益，要求最大化我方落子收益并最小化对方落子收益。收益由估值函数得出，通过极大极小树我们可以获得最大收益和一条收益最大的路径。在程序中为了节省搜索，极大极小树与 Alpha-Beta 剪枝算法被搭配使用。Alpha-Beta 剪枝算法相当于给收益值建立一个区间，随着搜索遍历更新区间的两端（即 Alpha 和 Beta 值），若出现区间左端大于右端的情况，则把出现该情况的节点排除出搜索范围，以此减小搜索量。（下图：极大极小树图示 from 参考文献 3）



2. 鉴于面向过程比面向对象的效能更高，我们的代码是用面向过程的思路编写的，数据结构体现在 simulation 函数中。本组在大作业最终代码中运用了列表、字典等基本数据结构，并尽量使用字典以节省时间。
3. 本组在代码中的自定义类有规定的 player 类，初始化如下

```python
class Player:
    def __init__(self, isFirst:bool, array:list):
        '''
        初始化
        参数: isFirst是否先手, 为bool变量, isFirst = True 表示先手
        参数: array随机序列, 为一个长度等于总回合数的list'''
        self.isFirst = isFirst
```

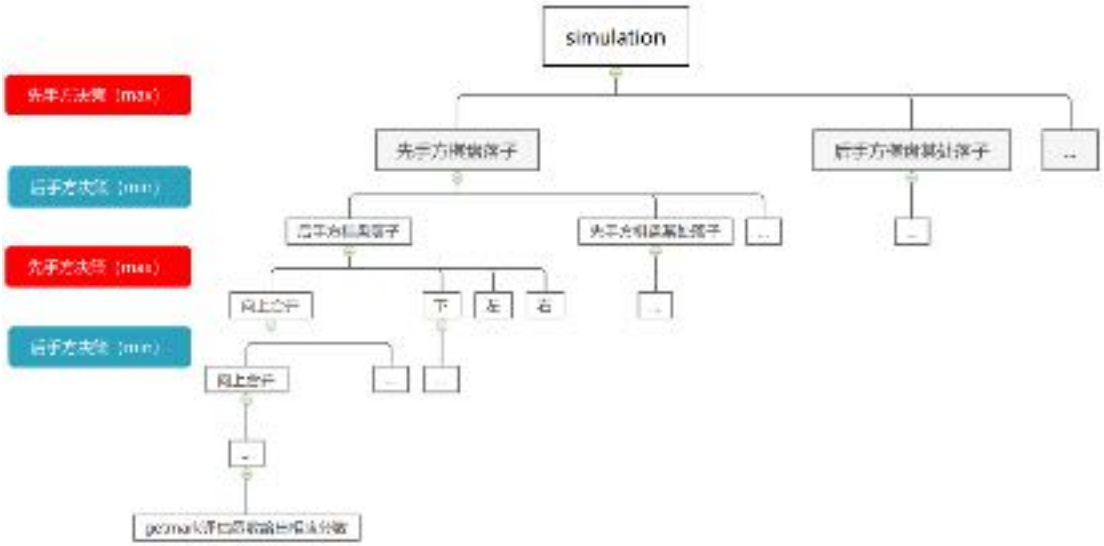
2.2 函数说明

（1）模拟函数

simulation(depth:int, Round:int, Gamemode:int, is_max:bool, board, alpha = Min_mark, beta = Max_Mark, is_root = False)：

simulation 函数接收极大极小树搜索深度、当前回合数、当前决策内容、当前棋盘、先后手、用于传递的 alpha 值、beta 值，是否为根节点等参数，通过递归建立极大极小树，同时用 alpha-beta 剪枝进行优化。当该节点为根节点时，返回最终作出的决策（落子位置或移动方向），当节点为叶节点时，用评估函数给出当前棋局的得分。同时在每个非根节点返回相应的 alpha，beta 的值。

建立的极大极小树如图：



(2)评估函数

getmark(board,isFirst):

评估函数 getmark 通过分析当前某方棋盘状况，返回对当前棋盘的评估值。评估方法为：对自己的棋盘遍历每个棋子，若为空位，则评估值加一，若为等级为 n 的数字，则评估值增加 2^n，同样对对手的棋盘，减去相应的值，得到最终的评估值。

```
Dict = {1: 2, 2: 4, 3: 8, 4: 16, 5: 32, 6: 64, 7: 128, 8: 256, 9: 512, 10: 1024, 11: 2048}
def getmark(board,isFirst):
    '评估函数'
    #part1:除了连接在一起的
    #必须两方所有棋子的从大到小的对比
    ours = board.getScore(isFirst)
    theirs = board.getScore(not isFirst)
    zeros = len(ours)-len(theirs)
    result = zeros
    for chess in ours:
        result += Dict[chess]
    for chess in theirs:
        result -= Dict[chess]
    return result
```

2.3 程序限制

1. 在天梯上，当比赛频次较高、天梯较为拥堵时，AI 超时的可能性会增加
2. 对于某些局面，Alpha-Beta 剪枝会不充分，并导致超时
3. 搜索深度 depth 目前最大到 6
4. 启发式规则较少，目前并未采用到较有效的启发式规则。

3 实验结果

3.1 测试数据

实验环境说明：

- 硬件配置：（i5-8300H/16G）NVIDIA Quadro P600
- 操作系统：Windows10
- Python 版本：3.7.4

我们组的实验分为多组，并且对不同的结构均有测试。

第一组测试了两种评估函数设计思路的好坏，设计思路有两种。第一种为由大到小比较各个数，并输出最大的数，代码为 Laser.py。第二种是将双方的棋子进行 2 的次方运算，然后自己的为正，对方为负，空格不算，代码为第二代 micro wave .py。通过与默认 player.py，发现后者获胜率接近 80%，前者获胜率先手为 75%，后手为 55%，故第一种效果好。

第二组实验测试了对空位的不同处理方式。本组实验将评估函数的空位部分设为双方空位数之和 red light.py 与双方之差 orange light.py。这代表了对全部空位一视同仁与分清敌我空位两种思路。在与别人和自己测试了 100 局以后，发现两者平均表现 orange light.py 较高。red light.py 发挥方差较大，经常为大比分胜负(即 10:0 或 9：1)，而 orange light.py 发挥较稳

定(除了个别较强代码没有大负，胜负较均衡)。故最后选择 orange light.py 为基础。

第三组实验了为更大的数设置更大优先级的结果。本组实验中，每个数字对应的值有 $(k*level+b)*2^{level}$，$2^{level}-1$，$\mathrm{int}\left(2.5^{level}\right)$、$\mathrm{int}\left(level^{2.5}\right)$相应代码有_ultraviolet ray.py、green light.py、yellow light.py、λ=400.py、Pulsar Pulse.py。通过本地与天梯左右互搏，发现当 k 较小时对胜率并无影响（胜率接近 5:5），k 较大时会使胜率降低（平均变为 3:7 或 4:6）。2.5、3 等数的 level 次方胜率均下降下降（平均为 3:7）。而$\mathrm{int}\left(level^{2.5}\right)$与原评估函数变现相似，但胜率略微下降（为 44:56）。故通过第三组实验后，我们组选择用原来 orangelight 的评估函数。

第四组实验测试了为不同位置设置不同的值的影响。思路为根据位置的重要级别为不同位置设置不同的权重，代码包括 green light_2、blue light、GAMMA RAY 2.0、GAMMA RAY。实验结果为因为计算量过大会导致超时，将时间设长后发现效果不如以前（平均为 7:13），故未被采用。

在我们用字典结构优化并提速之后，我们最后进行了一系列实验。我们先调整空位占的权重比，然后发现当权重在 1/3 到 4/3 之间时胜率不变，超过这个范围后胜率随着比例的变化急剧下降，故此采用了原来结果。

最后我们想到了一个优化方案：考虑到落子时基本上落在自己的区域，可以在落子环节把迭代层数下调，这样富裕的时间可以用来写更复杂的评估函数。我们重新考虑了不同位置的权重，发现效果并不好（平均 9:11），故最后上交的代码采用了较早的 orange light.py

除此以外，我们还进行了其他实验。我们先对孙宇辰与陆帅甫的 MCTS 进行了测试，发现严重超时。在不超时时，效果甚至不如默认的 player.py。当对局时间调整为每回合 10 秒时，效果才与 player.py 相近。最后，我们还通过天梯的数据进行了两组 CNN 的训练与测试，结果由于损失值 loss 下降过于缓慢而停止。将未完全训练的模型放入 purple light.py 在本地测试后，由于平台与 pytorch 不兼容（当时为了测试效果没用 numpy 而是直接用了 pytorch）导致测试失败。

3.2 结果分析

通过调参实验，我们发现了一些规律：对于棋子的权重，各个级别的数要有梯度而梯度不能太高，如两个 2 的权重不应该大于一个 4。如果给大数过大的权重，则 AI 会为了寻求更大的结果而放弃一些关键的较小的数。对于空位的权重，在一定范围内无影响是因为空位本身权重小于棋子，故一定范围内效果不变（即与原来评估函数单调性相同）。其余值过小会导致 AI 不关注空位以致于自己堵死自己让对手继续下赢，而空位权重过大则会导致之前所说的问题。对于各个位置的权重，由于本身敌我在同一位置会有矛盾，故很难准确地设计，且测试过程中出现了一味保守忍让失去进攻机会和过分进攻导致“白给”等现象。

在其他更高级的探索中，我们发现了如下规律：MCTS 的有效性只有搜索树较为庞大时才会体现出来，那时候它能有效地减少不必要探索的节点。而当搜索树规模较小时，由于计算的复杂程度较大，会使得性能反而不如 alpha-beta 剪枝。以及，许多 MCTS 树结构需要先验概率指导（如 alphaGo Zero 采用的 PUCT）,而探索出 PUCT 需要庞大的实验。

在关于卷积神经网络的测试中，我们发现：2048 由于情况过多，以致于较少的层数并不能使卷积核识别大部分情况。由于卷积核的每个输出对应着一种情况，故需要特别多的层数才能表示完整的棋局分析。而对于我们电脑的算力而言，过于复杂的模型几乎无法训练，

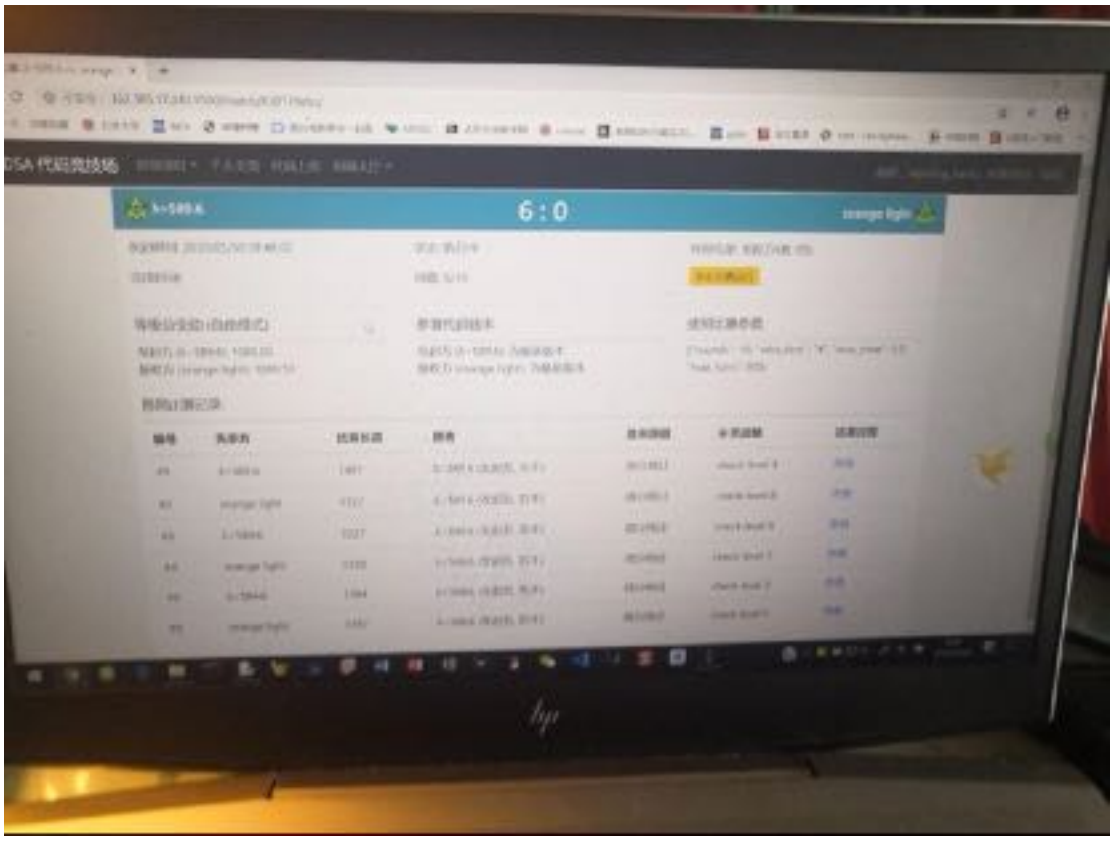
故 CNN 的结构对于普通学生而言并不现实。

3.3 经典战局

我们组经典战局如下：

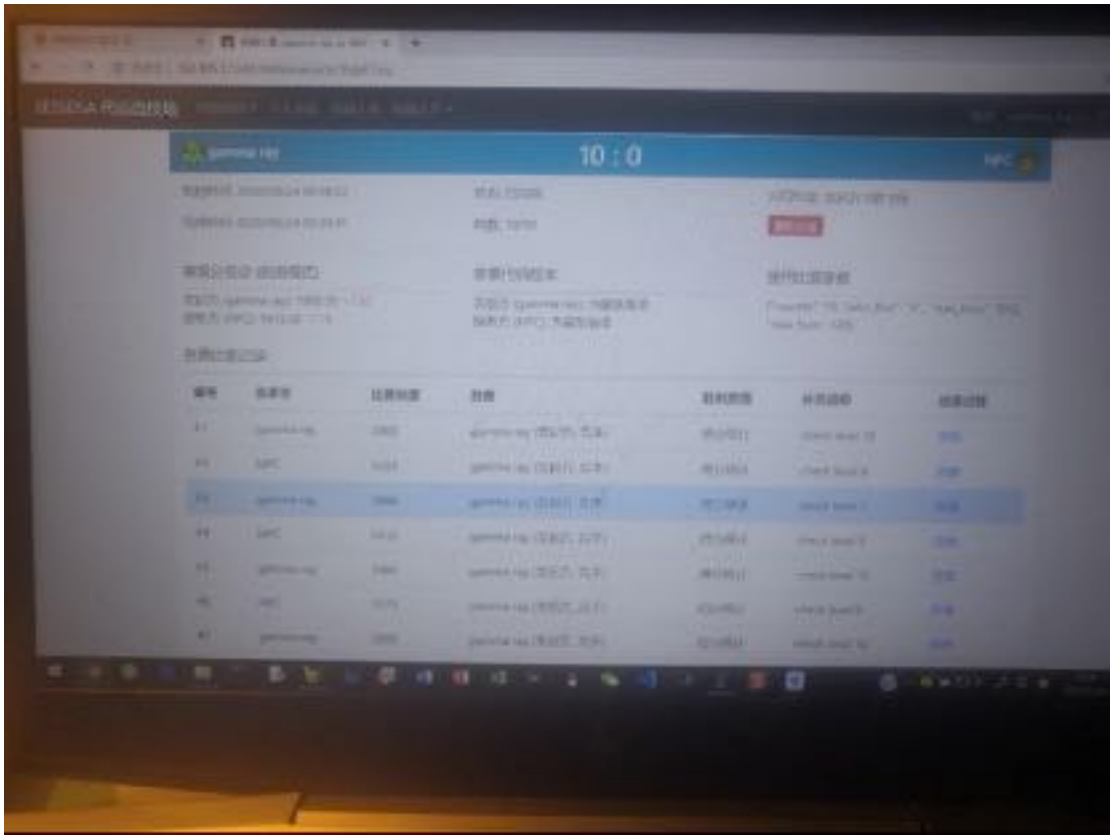
1.势如破竹（orange light 对战 red light）

2.偶然的幸运（一个胜率较低的实验代码 6 连胜）



3.拦腰折断（我们组代码为先手）

4.狂虐初代 NPC（gamma ray 为 red light 的前身，除了容易超时外与 red light 战力相同）



5.开门红（刚上天梯测试不超过一个小时双代码登榜首）

6.巅峰时刻（战胜 RBQ_4）

4 实习过程总结

4.1 分工与合作

五人共同参与了前期资料的查找、后期的代码测试和文档编写

曹顺顺：组织了一些分工工作，参与了评估函数编写和优化

陈风宇：参与了评估函数优化，提出了一些策略

孙宇辰：编写了树的基本框架，根据组员整理的资料尝试了 MCTS 和卷积神经网络，参与了评估函数编写优化

和冬华：参与了评估函数优化，整理并研究了 Alpha Go 和传统 2048 的资料

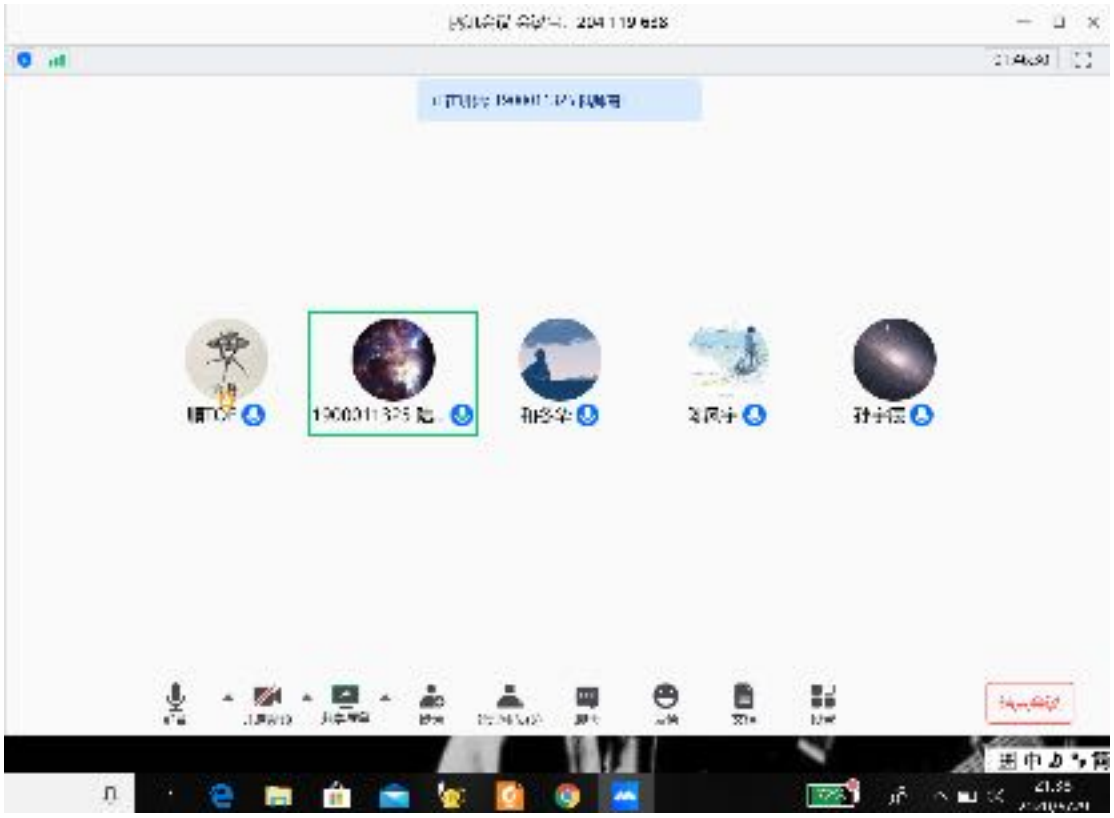
陆帅甫：根据 alphaGo Zero 的程序进行了初版的蒙特卡洛树编写和整理，参与了评估函数的优化

我们平日里采用微信群交流，并在腾讯会议上进行了三次讨论，主题分别是“算法的初讨论与分工（5 月 18 日）”、“AI 的提速和优化（5 月 29 日）”以及“文档编写的讨论和新优化思路（5 月 31 日）”

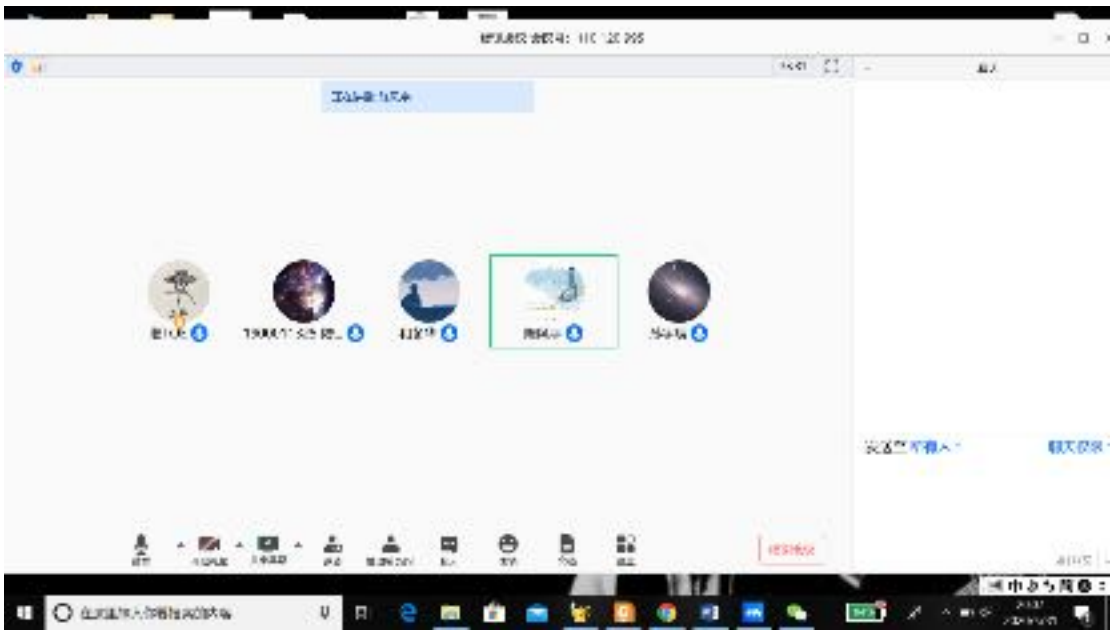
微信群截图：

第一次会议截图：

第二次会议截图：



第三次会议截图：



大作业合作过程代码、资料汇总：https://github.com/clbm/SESSDSA-2048-F19-X-ray-

4.2 经验与教训

1. 我们的代码从长时间看来发挥稳定
2. 学了很多东西（蒙特卡洛、神经网络等均有探索，甚至完整地读了 alpha Go 的两篇论文）
3. 明白了算法的效果不是绝对的，如在时间紧凑时，蒙特卡洛树并不能发挥比极大极小树剪枝更好的效果；矩阵维度较低时，numpy 的运算速度反而低于传统 for 循环。
4. 得到了 Python 对不同计算的速度比较：直接从字典取值 > 四则运算；移位运算“>>” > 一般运算；将数字看成二进制的运算“| & ^”大于普通的四则运算与取余数
5. 调参过程：在我们的计分体系下，我们对评估函数的设计（如战力，机动性能）等概念有了进一步了解。
6. 我们需要学习一下 os 库，这样可以自己建造对战平台并进行大批量自动对决。
7. 在目前的极大极小树体系中，很难往评估函数里加过多参量。
8. 要大胆尝试。在本地时，我们的代码一直超时。我甚至怀疑过它的战斗力。当我冒险将代码传上天梯时，我才知道我们的代码表现非常好而且不会超时。
9. 要学习多种语言，如 C++与 java。我们组很早以前找到了一个 C 写的传统 2048 的代码，但由于我们都对 C++语言不熟悉，所以直到最后才看，失去了对评估函数中平滑度与单调性的测试的机会。
10. 有时候不逼自己一下，是不知道自己的实力的。为了这个项目，我们甚至系统学习了 alpha Go 的算法。这是原来我们根本不敢想的。

通过阅读决赛的代码，我们有了如下经验与教训。（Golf 实现了我们组对启发式规则剪枝的思路，Uniform 组实现了我们组对平滑度设计的构想。）

1. 有时候解决不了问题，并不是我们把问题想得不够复杂，而是我们把问题想得不够简单。

我们苦于用 numpy 甚至 CNN 判断是否剪枝，却不知道别的组用枚举所有情况实现了。
2. 当不知道是否有效果时，最好的办法是大胆尝试而不是一味地考虑可行性。我们的评估函数没有平滑度，只是因为我们一直不知道敌方我方棋子在平滑度中如何区分。但我们应该大胆地把所有可能尝试，而不是一味地停留在纸上。
3. 不要过于迷信平台。当平台与代码实现接口相对不兼容时，可以大胆地尝试自己重写平台。如 Uniform 组的代码基本上全部为平台的重构。

4.3 建议与设想

1. 可以考虑固定一下天梯平台的对战频率，或许可以让平台运行更有效。
2. 可以更早地准备这个项目。最好在开学时就开始构想，期中时给出最后的规则，这样准备时间能更充分。因为期末很多同学其余科目任务较重，导致他们心有余而力不足，不能更好的研究与调试。

5 致谢

感谢我们小组五位成员的共同努力与付出，感谢陈斌老师对我们一些问题的解答。感谢王子铭同学的几次交流。特别感谢清华 18 级信科学长蔡严正提供了许多技术上的支持，包括 α - β 剪枝如何优化，MCTS 如何节省时间以及卷积神经网络的结构如何设计。最后感谢 B 站、知乎、CSDN、Github 上的各位不知名的 up 主、程序员、研究生，他们整理了国内外各种优秀的资源并进行了优秀的讲解，使我们能够学习。

6 参考文献

1. https://github.com/pkulab409/sessdsa.2048/blob/master/documents/sessdsa2048%E7%94%A8%E6%88%B7%E6%96%87%E6%A1%A3.md（制作方资料）
2. https://gitee.com/chbpku/sessdsa.2048/tree/master/（制作方资料）
3. https://zhuanlan.zhihu.com/p/55749861（极大极小树）
4. https://www.bilibili.com/video/BV1a7411K7g1（剪枝）
5. https://www.bilibili.com/video/BV1XJ411n7Ee（极大极小树）
6. https://www.bilibili.com/video/BV1p4411U7CK?from=search&seid=9789175941228542696（蒙特卡洛树）
7. https://blog.csdn.net/windowsyun/article/details/88770799（AlphaGo 相关，以及蒙特卡洛树）
8. https://www.bilibili.com/video/BV1YJ411R7Dc/?p=8（计算机博弈原理，深度学习，AlphaGo，蒙特卡洛树的解读）
9. http://www.nature.com/nature/journal/v529/n7587/full/nature16961.html（alphaGo 原论文）
10. https://blog.csdn.net/marble_xu/article/details/90450436（评估方法）
11. https://github.com/nneonneo/2048-ai（传统 2048 大师级 AI）
12. https://github.com/tensorflow/minigo（AlphaGo 主要部分的纯 Python 实现）
13. 《人工智能：一种现代的方法(第 3 版)》 Stuart J.Russell，Peter Norvig 著，清华大学出

版社
14. 《TensorFlow 深度学习》北京大学人工智能导论的参考资料

数据结构与算法课程实习作业报告

贺群超*、詹景昕、赵云飞、周雨琢、宾奕颉

摘要：我们的代码主要运用 MinMax 树配合α β剪枝的算法来确定移动方向，其中搜索树子节点的值由估值函数给出；落子时默认在己方落子，当我方棋盘已满时，通过对对手可用位置进行估值来确定在对手的落子位置。经过我们组团结攻关，代码能按要求在平台上平稳运行。但遗憾的是，我们组未能出线。在报告最后，我们根据胜出小组的代码详细分析了我组代码值得改进的地方。

关键字：MinMax 树 α β剪枝

1 算法思想.....1
1.1 总体思路.....1
1.2 算法流程图.....2
1.3 算法运行时间复杂度分析.....3
2 程序代码说明.....4
2.1 数据结构说明.....4
2.2 函数说明.....4
2.3 程序限制.....6
3 实验结果.....7
3.1 测试数据.....7
3.2 结果分析.....8
3.3 经典战局.....9
4 实习过程总结.....10
4.1 分工与合作.....10
4.2 经验与教训.....13
4.3 建议与设想.....13
5 致谢.....14
6 参考文献.....14

1 算法思想

1.1 总体思路

我们的代码主要采用了树等数据结构和 Minimax 算法、α β剪枝算法。

移动方向的给出利用了 MinMax 树配合α β剪枝的算法。MinMax 树有 Min 节点和 Max 节点，其中 Max 节点模拟己方的移动，Min 节点模拟对方的移动，且模拟时默认在各自的棋盘落子，只有当一方棋子满时才会在另一方落子。对 MinMax 树的搜索采用深度优先搜

索，直到深入到特定层数（一般设置为 4）时停止并利用估值函数（见下）对局面进行评估，当符合特定条件（见下）时进行 α β 剪枝。

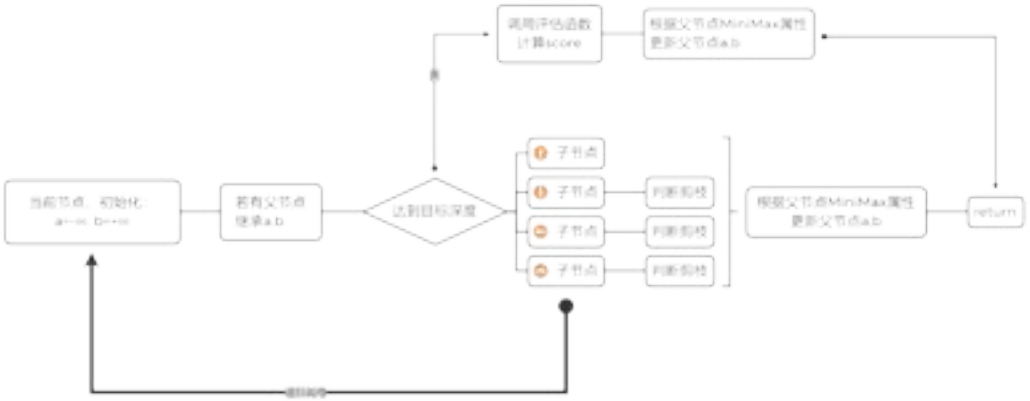
落子默认在己方落子，当己方已满时，将下在对方棋盘上上下左右最近的棋子等级和最大的空位。

算法的设计过程：

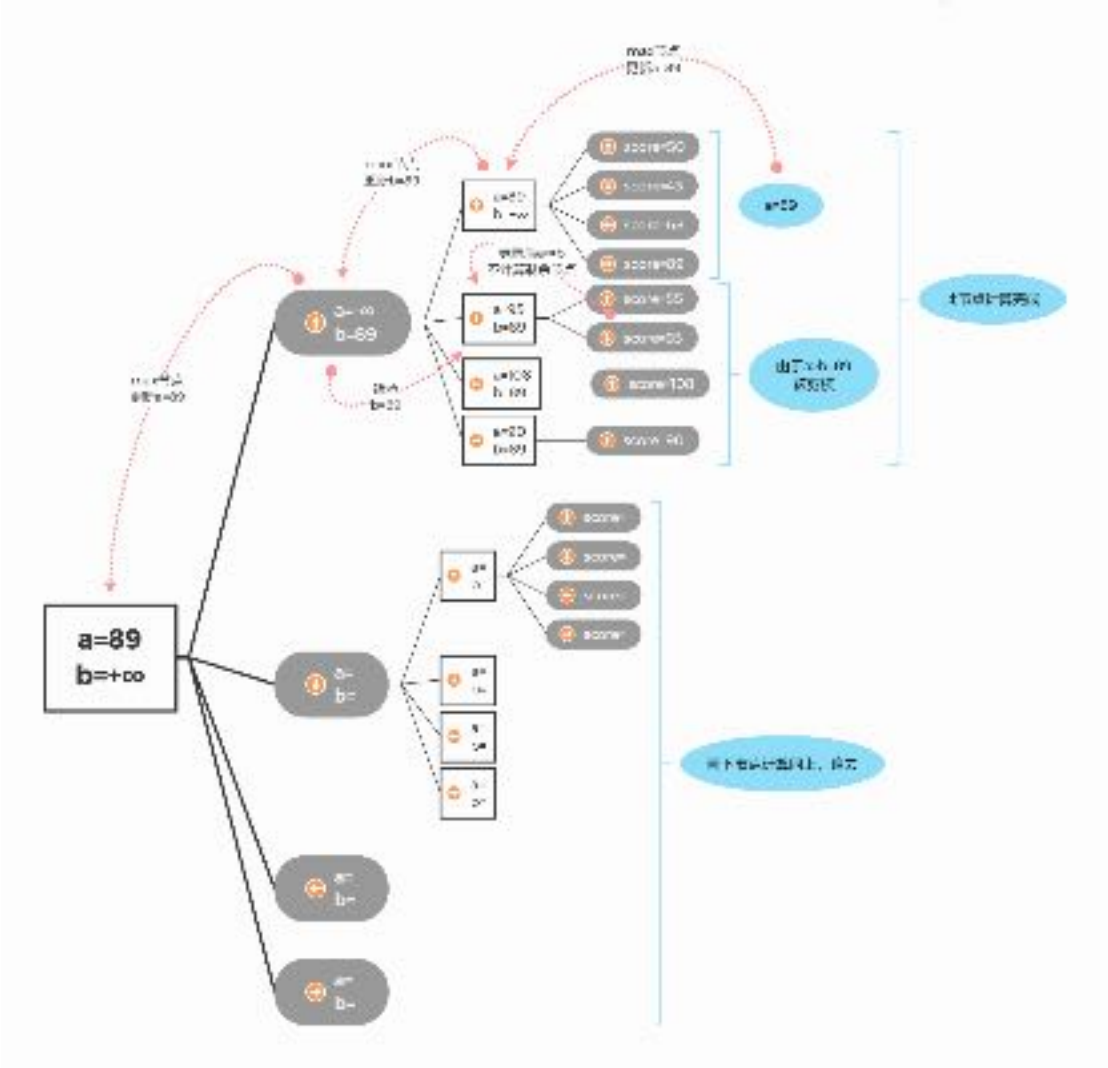
我们首先在网络上搜索了一些原始 2048 游戏的算法，注意到它们中有许多都利用了 MinMax 树配合 α β 剪枝的算法，据此我们编写了最初的代码。当只有一个玩家时，最初的代码表现良好，可以很快地合成 2048，但当它与随机代码进行对抗时，却经常出现它被吃掉很多子的局面。于是我们意识到我们所模仿的代码的评价函数和搜索树都没有考虑到对手对自己的影响（毕竟原来的代码解决的都是单人游戏），所以我们又添加了如占领对方总格数，被对方占领总格数等等的评价函数，并在决策树中考虑了对方对整体局面的影响，使算法的性能得到了较大的提升。此后我们进行的工作主要是测试新的评价函数和调整参数，对整体算法并无较大的改动。最终在清除一些较小的 bug 后，我们得到了最终的算法。

1.2 算法流程图

搜索和剪枝流程图



搜索过程示例图



1.3 算法运行时间复杂度分析

算法复杂度分析：

一：估值函数

设棋盘格数目为 n。

① emptyCells(chessboard)：由于 chessboard.getNone(self.isFirst)的复杂度是$O(n)$的，因此该估值函数的复杂度为$O(n)$。

② ScoreSum (chessboard)：复杂度为$O(n)$。

③ maxScore(chessboard)：由于 chessboard.getScore(self.isFirst)是$O(n)$的，因此该估值函数的复杂度为$O(n)$。

④ OccupySum(chessboard)：复杂度为$O(n)$

⑤ LoseSum(chessboard)：复杂度为$O(n)$

⑥ Monotonicity(chessboard)：复杂度为$O(n)$

⑦ Smoothness(chessboard)：复杂度为$O(n)$

⑧ isInvaded(chessboard)::：复杂度为$O(n)$

综上，整个估值函数 rating(chessboard)的复杂度为$O(n)$。

二:搜索树算法复杂度

设搜索深度为 m（即 depth=m）。虽运用了 α－β 剪枝，但在最坏的情况下，搜索树的节点个数为$O(4^m)$。搜索树最后一层节点数亦为$O(4^m)$，此外，最后一层节点需要调用 rating 估值函数（复杂度是$O(n)$），因此整个搜索树算法复杂度为$O(n4^m)$。

三:在对方落子的复杂度

addHelper(chessboard,isFirst,to_add)：复杂度为$O(n)$

上述分析绘制成表格如图所示：

注：默认期盼所提供接口的复杂度为$O(n)$。

函数名称	复杂度
emptyCells	$O(n)$
ScoreSum	$O(n)$
maxScore	$O(n)$
OccupySum	$O(n)$
LoseSum	$O(n)$
Monotonicity	$O(n)$
Smoothness	$O(n)$
isInvaded	$O(n)$
rating	$O(n)$
abSearch（搜索树）	$O(n4^m)$
addHelper	$O(n)$

综上，整个程序的复杂度是$O(n4^m)$

2 程序代码说明

2.1 数据结构说明

本组代码主要应用了树、列表等数据结构，Player 等自定义类。

以下为对于自定义类的功能说明

Player	规则要求
Node	Min 节点和 Max 节点的父类
MaxNode	MinMax 树的 Max 节点
MinNode	MinMax 树的 Min 节点
StateTree	模拟对局的搜索树

2.2 函数说明

① 评价和辅助函数

Ⅰ.列表说明如下

Check	判断合法的合并方向
closest	给出某一个格子上下左右四个方向最近的棋子的等级
emptyCells	计算己方剩余空位数
ScoreSum	计算己方棋子等级四次方之和，反映总分
OccupySum	计算占领对方的总格数
LoseSum	计算己方被占领的总格数
Monotonicity	计算己方棋盘的“单调性”
Smoothness	对己方每一个棋子计算其与上下左右最近棋子的高度差之和，再累计求和计算己方棋盘的“光滑度”
isInvaded	判断己方“内部”（中心边长为 2 的正方形）是否被对方侵入
rating	利用以上函数进行加权计算，给局面评分
MoveHelper	利用 abSearch 给出得分最高的合法移动方向
add_rate	给对方的空格评分
addHelper	利用 add_rate 找到对方得分最高的空位

Ⅱ.以 Monotonicity 为例进行说明：

代码如下

```
def Monotonicity(chessboard):
    totals=[0,0,0,0]

    for x in range(4,4):
        Current=myinit
        Next=myinit+1
        while Next<myinit+4:
            while Next<myinit+4 and not chessboard.getValue((x,Next)):
                Next+=1
            if Next==myinit+4:
                Next-=1
            deltaHeight=chessboard.getValue((x,Current))-chessboard.getValue((x,Next))
            if deltaHeight>0:
                totals[0]-=deltaHeight
            elif deltaHeight<0:
                totals[1]-=deltaHeight
            current=Next
            Next+=1

    for y in range(4,4):
        Current=0
        Next=1
        while Next<4:
            while Next<4 and not chessboard.getValue((Next,y)):
                Next+=1
            if Next==4:
                Next-=1
            deltaHeight=chessboard.getValue((Current,y))-chessboard.getValue((Next,y))
            if deltaHeight>0:
                totals[2]-=deltaHeight
            elif deltaHeight<0:
                totals[3]-=deltaHeight
            current=Next
            Next+=1

    return max(totals[0],totals[1])+max(totals[2],totals[3])
```

[1]如图所示，分别从左右和上下了两个方向进行讨论。
以水平方向为例，从左到右，用 totals[0]记录每行相邻棋子高度减小值（如果高度上升则不计）的总和，同样地，totals[1]记录高度增大值总和。竖直方向类似。

② 核心函数 abSearch

代码如下

```
def abSearch(self,node,depth,a,b):
    if depth==self.depth:
        node.score=rsting[node.state]
        return node.score
    if isinstance(node,MaxNode):
        v=-2**15
        to_move=check(node.state,self.isFirst)
        for i in to_move:
            new_board=node.state.copy()
            legal=new_board.move(self.isFirst,i)
            if not self.isFirst:
                my_to_add=new_board.getNext(self.isFirst,node.currentround+1)
                op_to_add=new_board.getNext(not self.isFirst,node.currentround+1)
                if my_to_add:
                    new_board.add(self.isFirst,my_to_add)
                if op_to_add:
                    new_board.add(not self.isFirst,op_to_add)
                nextnode=MinNode(new_board.copy(),node.currentround+1)
            else:
                nextnode=MinNode(new_board.copy(),node.currentround)
            nextnode.direction=i
            node.link(nextnode)
            v=max(v,self.abSearch(nextnode,depth-1,a,b))
            a=max(a,v)
            if b<=a:
                break
        node.score=v
        return node.score
    elif isinstance(node,MinNode):
        v=2**15
        to_move=check(node.state,not self.isFirst)
        for i in to_move:
            new_board=node.state.copy()
            legal=new_board.move(not self.isFirst,i)
            if legal==False:
                raise RuntimeError
            if self.isFirst:
                my_to_add=new_board.getNext(self.isFirst,node.currentround+1)
                op_to_add=new_board.getNext(not self.isFirst,node.currentround+1)
                if my_to_add:
                    new_board.add(self.isFirst,my_to_add)
                if op_to_add:
                    new_board.add(not self.isFirst,op_to_add)
                nextnode=MaxNode(new_board.copy(),node.currentround+1)
            else:
                nextnode=MaxNode(new_board.copy(),node.currentround)
            nextnode.direction=i
            node.link(nextnode)
            v=min(v,self.abSearch(nextnode,depth-1,a,b))
            b=min(b,v)
            if b<=a:
                break
        node.score=v
        return node.score
```

具体实现对决策树的深度优先搜索和 α β 剪枝

2.3 程序限制

本代码 MinMax 树是有缺陷的：当某一节点的子节点没有可以移动的方向时，该节点父节点的其他子节点都会被剪掉，这会导致给出错误的方向，但我们尝试修复这个 bug 后发现算法的效果变差了，最终只能换了另一种较为消极的方法避开这个 bug（即当检测出输出方向不合法时，输出按 0123 顺序排列的第一个合法方向）

[1] 计算方法参考 A simple AI for 2048. 2018-12-06(06)[2020-05-15].https://github.com/ovolve/2048-AI

3 实验结果

3.1 测试数据

实验环境说明：

- 硬件配置： Intel i5-8265U 1.8GHz/8GB
- 操作系统： Windows10 家庭中文版
- Python 版本：3.7.6

我们的算法主要经历了两轮有效优化过程

表一 各代码天梯对战结果

代码名称	胜	负	天梯积分
mp	118	145	908.66
mp4	742	847	986.86
mp4(2)	473	396	1022.59

在我们的初始算法 mp 中，估值函数只考虑了分数加权值（ScoreSum）、占领对方空地的数量（OccupySum）、被对方占领的空地数量（LoseSum）、己方空地数量（EmptyCell）四项。观察对战过程，发现存在以下问题：

（1） 合并出大子的意愿不够强。

（2） 棋盘混乱，导致后期很容易被塞满。

对于问题（1），我们决定进行两项改变：

a. 改变 ScoreSum 的计算权重，将大子的权重调大。将原来的 scores[i]*i*i*math.log(i+1)改变为 scores[i]*i*i*i*i.

b. 将 EmptyCell 在总分数中的权重由 0.2 调大至 1.0. 这样算法会更加在意空地的数量，从而产生更大的合并意愿。

改变之后，我们得到了 mp4 算法。实战证明其性能提升了。

mp4 完胜 mp

针对问题（2），即棋盘过于混乱，我们加入了单调性（Monotonicity）、光滑性（Smoothness），产生了代码 mp4(2)，并调整其 weight，分别将其与 mp4 进行比拼。结果如下：

表一 单调性 weight 设置为 0 时改变光滑性 weight 的结果

SmoothWeight	对战回合数	对战结果（mp4:mp4(2)）
1.0	60	26:34
10	60	29:31
100	60	35:25

表二 光滑性 weight 设置为 0 时改变单调性 weight 的结果

MonotonicityWeight	对战回合数	对战结果
1.0	60	31:29
10	60	41:19
100	60	44:16

由此可见，在 SmoothWeight 为 1 的时候，它能够对算法产生微弱的改善，而之后反而使得算法性能下降。而 Monotonicity 始终未对算法产生改进效果，且也会在 Weight 过大时产生反作用。最后我们决定选取 SmoothWeight=1.0，MonotonicityWeight=1.0

3.2 结果分析

对 ScoreSum 的调整：

在把 ScoreSum 的加权模式从 scores[i]*i*i*math.log(i+1)改变为 scores[i]*i*i*i*i 后，算法会更加关注是否有大子出现，故算法性能提高。但并不能将大子的权重无限制的增加，因为这样很可能忽视了相对更小的子的合成，而很多时候对战输了是在最大的子相同的情况下，因为相对更小的子数量比对方少而输。故我们最后选取了 scores[i]*i*i*i*i。

对 EmptyWeight 的调整：

适当调大 EmptyWeight 的绝对值，可以使算法有更大的动力合并棋子、增多空位数，从

而增加分数。但如果将这一值调得过大，可能导致一味追求小子的合并而忽视了大子的合成。故最终我们将 EmptyWeight 选取为 1.

对光滑性、单调性权重的调整：

这一部分是整个调参过程中最为玄学的部分。按理来说，增加光滑性、单调性应该能够提升算法性能，但测试中发现似乎并不明显，甚至有倒退趋势。经过观察对战记录，我们发现，将单调性光滑性函数调大以后，棋盘总体布局的确有所提升，但是由于这时大子更倾向于出现在边缘（包括与对方接壤的地方），所以有更大的被吃的可能性。但由于发现这一点的时候已经非常接近提交代码的 ddl 了，所以我们没能针对这一问题进行进一步的优化，而是将单调性、光滑性的权重调小，使其弱点不至于暴露。

时间开销：

本算法的时间开销主要在搜索树中。由于所采用的深度不大，运行过程中基本没有出现超时的情况。

3.3 经典战局

如图是我们用加入光滑性、单调性之前的 mp4 与加入之后的 mp4(2)对战的结果。Mp4 为后手方，mp4（2）为先手方。在上图所示的位置，先手方有一个 512 被先手方吃掉了。这样的情况在 mp4 与 mp4(2)的对战中十分多见。

4 实习过程总结

4.1 分工与合作

① 小组分工

贺群超（组长）	组织、编写估值函数、调参以及撰写报告
詹景昕	核心代码编写以及撰写报告
赵云飞	Debug 以及撰写报告
周雨琢	编写估值函数、调参以及撰写报告
宾奕颉	绘制头像以及撰写并整合报告

② 合作与交流的方式主要是使用微信以及腾讯会议进行小组讨论，确定分工，代码思路，以及后期根据实际情况协调有关工作。

③ 组会记录：

第一次：5 月 15 日

微信讨论头像事宜

第二次：5 月 17 日 下午三点

讨论了关于代码基础框架及算法思路的一些简单想法，确定了在第三次组会之前需要完成的学习内容。

会议时间

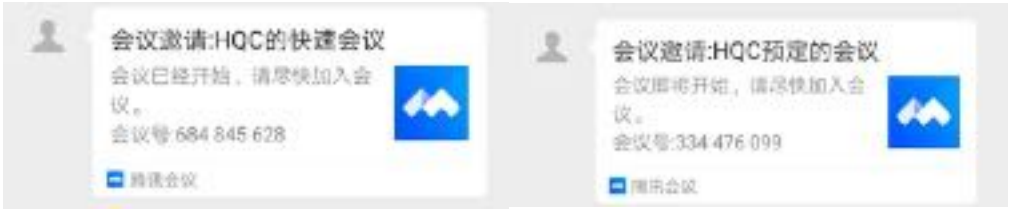
组员截屏

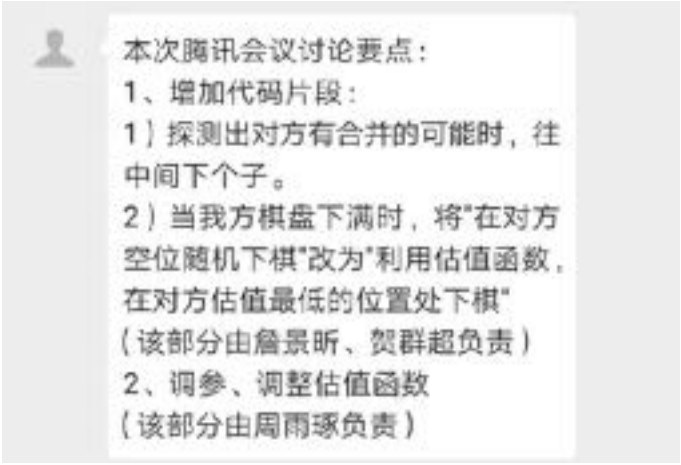
第三次 5 月 18 日晚上九点

微信讨论，在经过一天的知识补充后，对 2048 的代码进行深一步的思考和讨论

第四次 5 月 24 日下午四点 对已有程序进行修正与改进

第五次 5 月 28 日下午四点
第六次 5 月 29 日晚上八点三十
这两次会议内容相同，均是在天梯赛开启后对代码进行最后的讨论与修改





讨论要点总结

第七次 6 月 5 日晚上八点
比赛结束后对大报告撰写进行分工

4.2 经验与教训

① 得意之处：我们组在沟通交流方面有很多得意之处，现总结如下：
Ⅰ.每位组员均响应积极，尽了自己最大的努力和热情来完成工作
Ⅱ.大家能够互相理解难处，互相协调时间和任务内容
Ⅲ.从上述报告可以看出，我们组开了很多次会议，在没有花费太多时间的基础上，每一次会议都有所建树
Ⅳ.代码可读性很高，思路也很完整
② 不足之处：
Ⅰ.我们组的同学对 2048 都不太有灵感，大家都没有能提出非常好的编写代码的思路，所以虽然我们已经尽力完成工作，合作方面也无可指摘，但最终的成绩并不太理想。
Ⅱ.参数调整的效率不是很高。
Ⅲ.策略考虑还不够周全，没有分前中后期、先手方后手方的不同制定不同的策略。

4.3 建议与设想

① 建议：
Ⅰ.大作业时间临近期末，任务量繁重，加上这学期在线上教学的特殊性，导致在代码讨论和修改方面以及和同学的协作方面仍然不如线下方便，这在一定程度上增加了大作业的难度。建议在考虑时间和难度等综合因素的条件下，对完成大作业的时间进行更充分的考虑。

Ⅱ.增加一些更神奇的奖项评选，可以增加趣味性如“肝帝奖”（代码最长组），“稳如泰山奖”（比赛中排名几乎不变的组）等等。

② 对选修明年课程的学弟学妹的寄语：

数算虽然很累，但的确能学到很多，加油！

③ 实习作业后续工作的大胆设想：

Ⅰ.对于 2048 本身而言，可以把单纯的数字改成其他称呼，比如 2 是菜鸡，4 是咸鱼，2048 是大佬之类的，感觉更有代入感和趣味性。

Ⅱ.希望明天有更具有对抗性和策略性的游戏加入（有请下一位头秃同学）

5 致谢

感谢陈斌老师在这一学期对我们的谆谆教诲和用心良苦的各项策划！
感谢各位助教的帮助和敦促！
感谢技术组的各位助教和同学的付出！
感谢和我们一同学习了数算的其他同学！是你们给予了我们陪伴与动力！
感谢所有小组成员的共同努力！这次大作业因为有你们而分外精彩与充实！

6 参考文献

[1] A simple AI for 2048. 2018-12-06(06)[2020-05-15].https://github.com/ovolve/2048-AI

[2] 维基百科词条 Alpha-beta 剪枝.2020-05-24(24)[2020-05-15]. https://zh.wikipedia.org/wiki/Alpha-beta%E5%89%AA%E6%9E%9D

数据结构与算法课程实习作业报告

（陈海云 陈佳炜 冯杰超 孙磊 俞越*）

摘要：AI 核心是搜索树，主要使用的算法是 α-β 剪枝，并配以一些启发性的决策指导以及估值函数。通过天梯排位赛及出线小组对战，最终在本赛区获得季军。

关键字：（搜索树 α-β 剪枝）

1 算法思想.....1
1.1 总体思路.....1
1.2 算法流程图.....3
1.3 算法运行时间复杂度分析.....7
2 程序代码说明.....8
2.1 数据结构说明.....8
2.2 函数说明.....9
2.3 程序限制.....15
3 实验结果.....16
3.1 测试数据.....16
3.2 结果分析.....17
3.3 经典战局（可选）.....18
4 实习过程总结.....24
4.1 分工与合作.....24
4.2 经验与教训.....26
4.3 建议与设想.....26
5 致谢.....26
6 参考文献.....27

1 算法思想

1.1 总体思路

1.1.1 主要思路

Minimax 搜索树：

该算法应用于由两个玩家参与的零和游戏，并基于以下假定：**我方行动时会使自己收益最大，对方行动时会使我方收益最小**。为找到下一步的最优选择，以当前状态为根建立搜索树，节点代表局面，边代表操作，层内节点类型相同，而不同层由 Max 节点与 Min 节点交

替组成。然后从叶节点回溯，每个 Max 节点选择估值最大的子节点，Min 节点选择估值最小的子节点。回溯至根节点时，就能找到在当前展开层数下收益最大的操作。

α－β 剪枝：

在 Minimax 算法中，因为可以预判有些分支最终不可能被选取，若每次仍展开所有可行操作，会浪费大量时间。作为改进，将可能取值的最大下界记为 α，最小上界记为 β，每个节点存储 α、β 两个值。一开始，β 设为正无穷，α 设为负无穷。回溯时，经过的 Min 节点会更新 β 值，而 Max 节点会更新 α 值。再搜索其他分支时，子节点会继承父节点的 α、β 值。**若发现某节点的 α >= β，该节点的其他分支就不用探索了，因为不会找到更好的结果。**剪枝的缺点是算法效率依赖于搜索顺序，最差情况下，没有任何优化。**但平均来讲，会节约一半时间。**

带有先验知识的操作策略和评估函数

搜索树具有一定的短视性，而人肉观看对局可以总结出一些不用占用大量资源就可以实现的策略。用先验知识代替搜索树完成在某些情况下的操作决策，以及用先验知识在评估函数中对相应的操作进行奖惩，也是具有一定智能的 AI 的必经之路。

1.1.2 主要数据结构与算法

Minimax 算法所使用的**多叉树**，以及 **α－β 剪枝算法**

1.1.3 主要策略

一、启发式规则：

落子时不进行搜索，而是根据先验知识，按以下优先级给出落子位置：

a)是否落在对方某空位可以保护本方棋子；

b)是否可以在对方空位较少的情况下合理落在对方空位；

c)是否可以落在自己领地；

d)随机落在对方空位。

这样做可以在一般情况下获得较大收益，并为更重要的方向选择节省大量时间。

二、合理化搜索深度：

为防止超时，当剩余时间不多时，压缩搜索深度。

三、优化估值函数

估值函数平衡了以下因素：

a)对双方现有棋子的大小比较，棋子级别越大，对估值函数影响越大，且该影响为幂函数关系，以奖励合成更大棋子；

b)对占领对方领地的奖励；

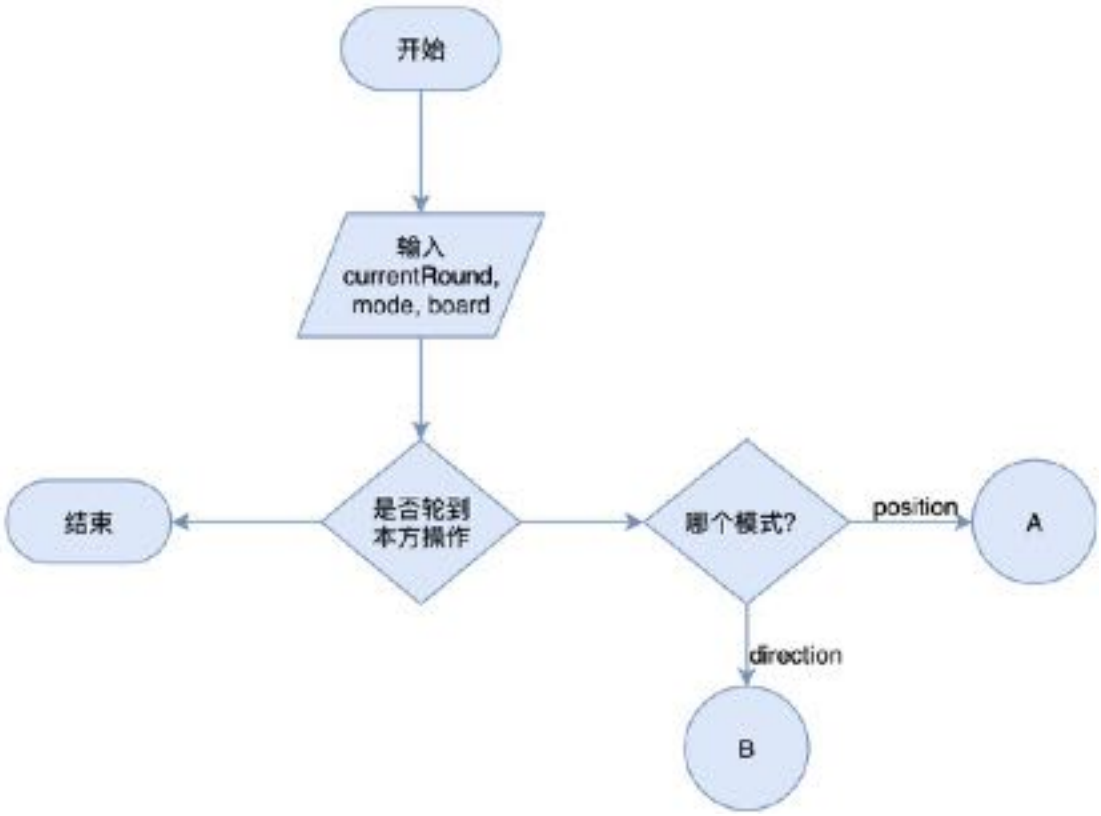
c)对风险的考虑，分先后手情况，对可能发生贴脸吃、远程吃的情况有相应奖惩。

1.1.4 算法设计过程

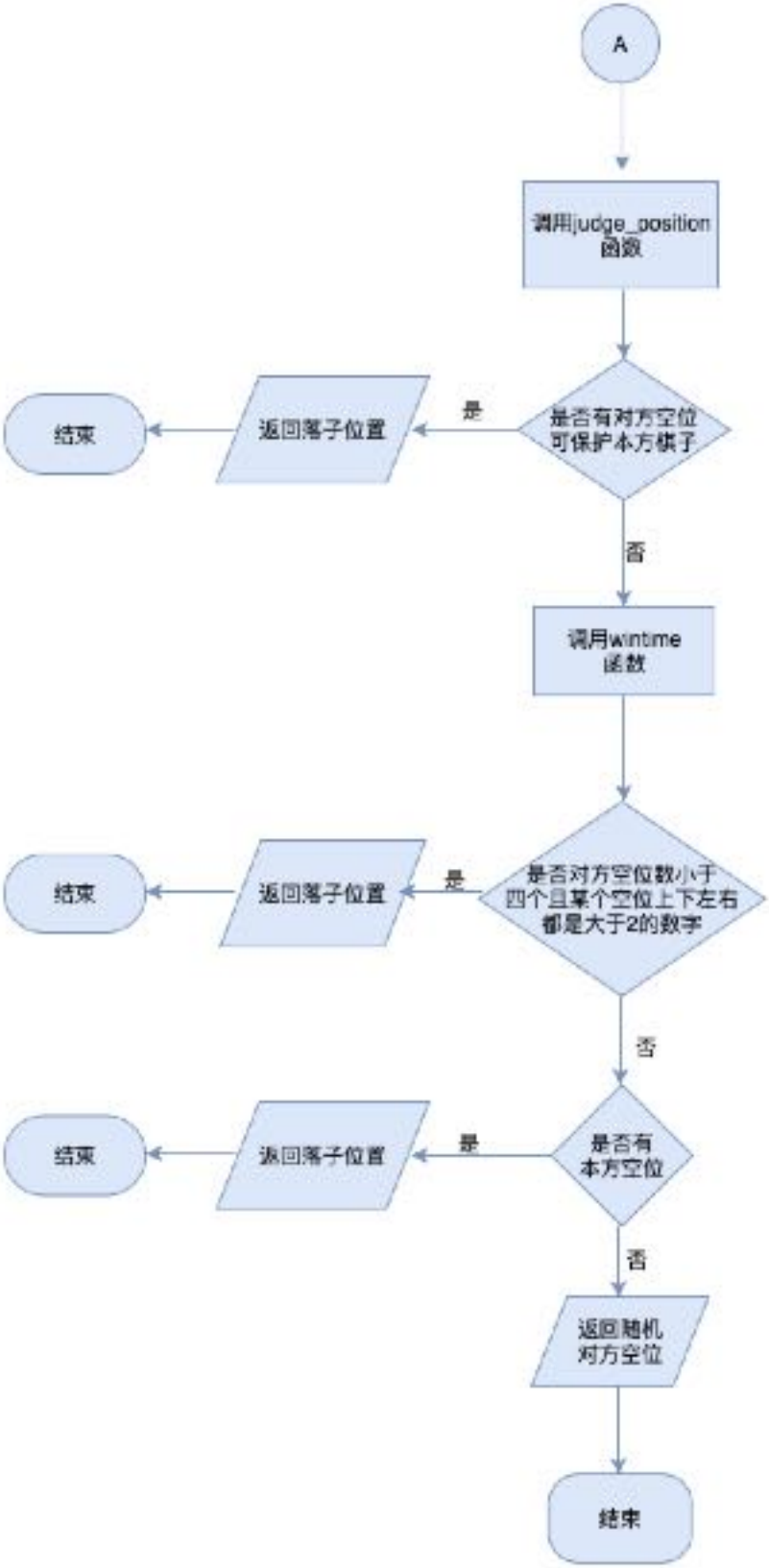
组内同学参加内测期间，形成了 Minimax 搜索树与 α－β 剪枝等基本框架。大作业正式开放后，重点放在优化时间、探索先验知识、采用启发式规则、优化估值函数以及改正 bug 等工作，最终形成参赛代码。

1.2 算法流程图

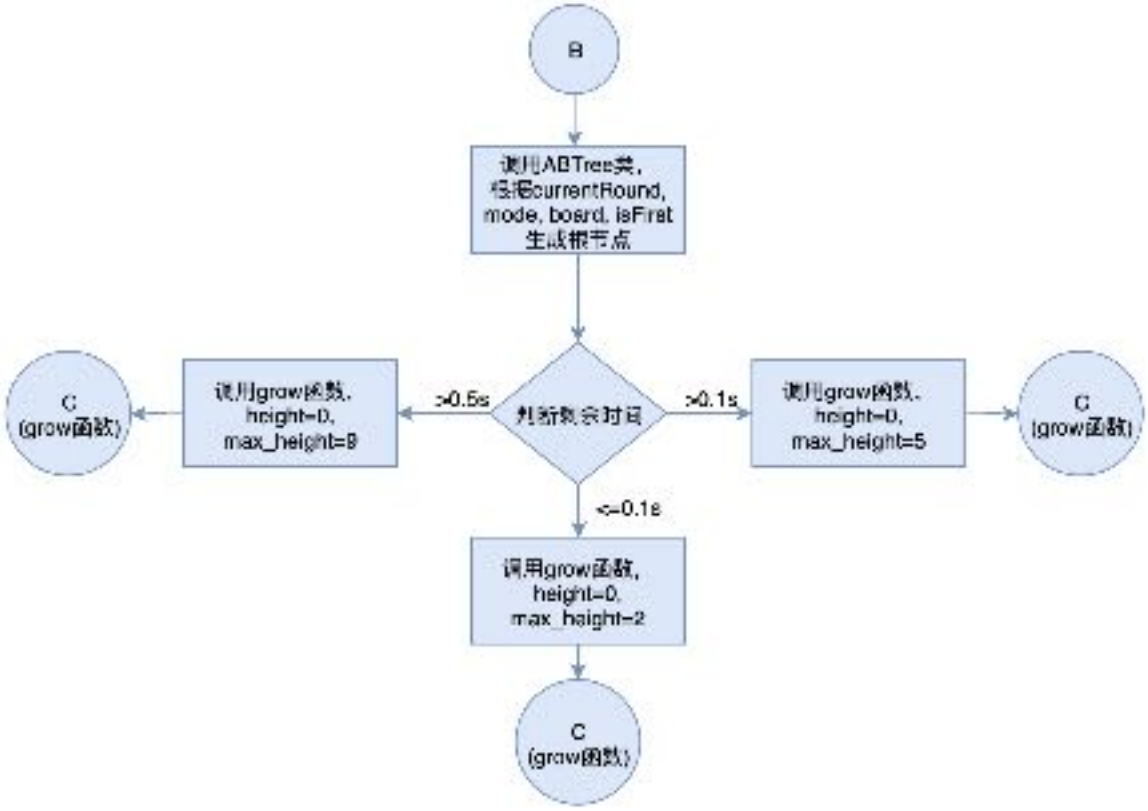
总图：



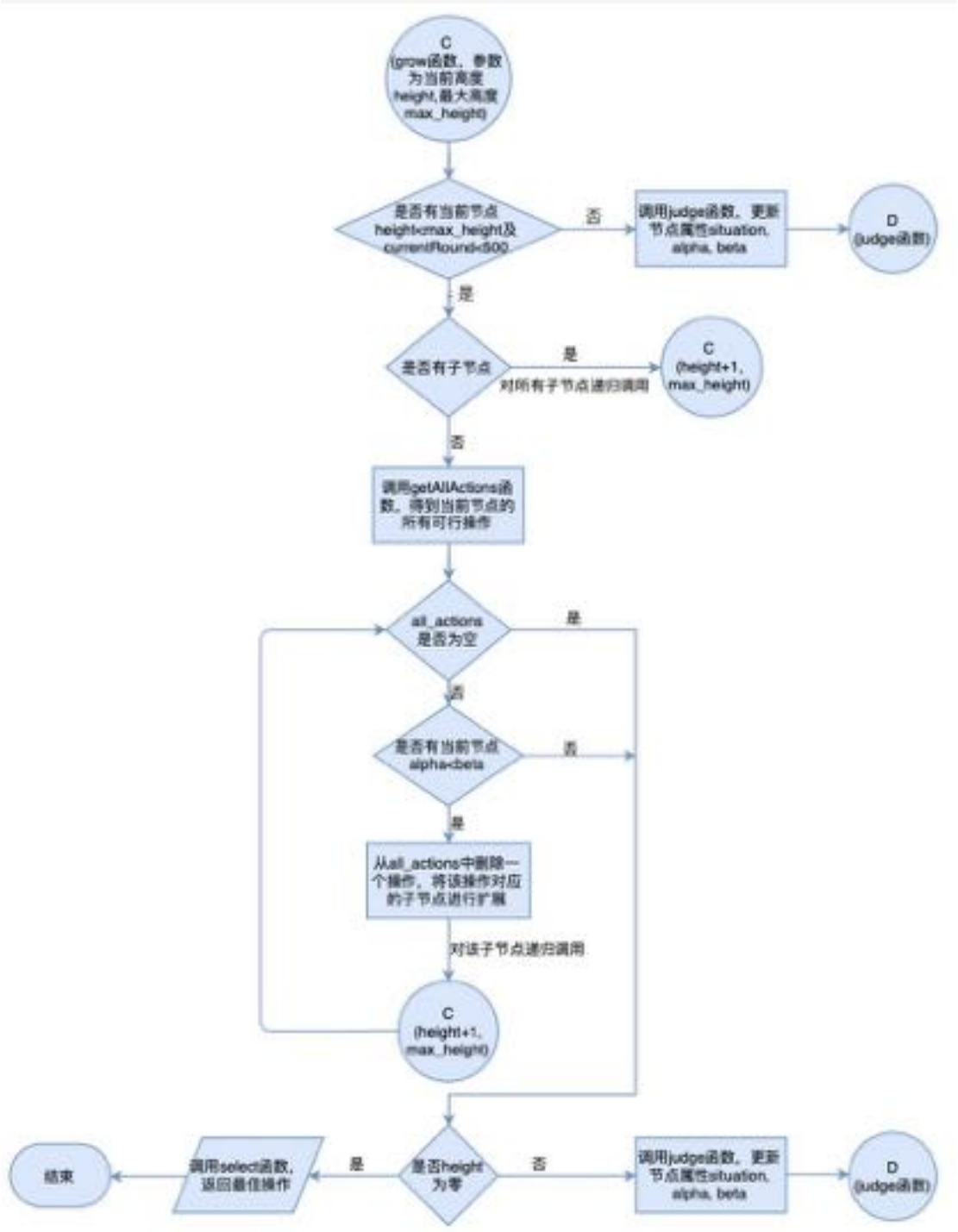
position 模式：


A
调用judge_position
函数
是否有对方空位
可保护本方棋子
是
返回落子位置
结束
否
调用wintime
函数
是否对方空位数小于
四个且某个空位上下左右
都是大于2的数字
是
返回落子位置
结束
否
是否有
本方空位
是
返回落子位置
结束
否
返回随机
对方空位
结束

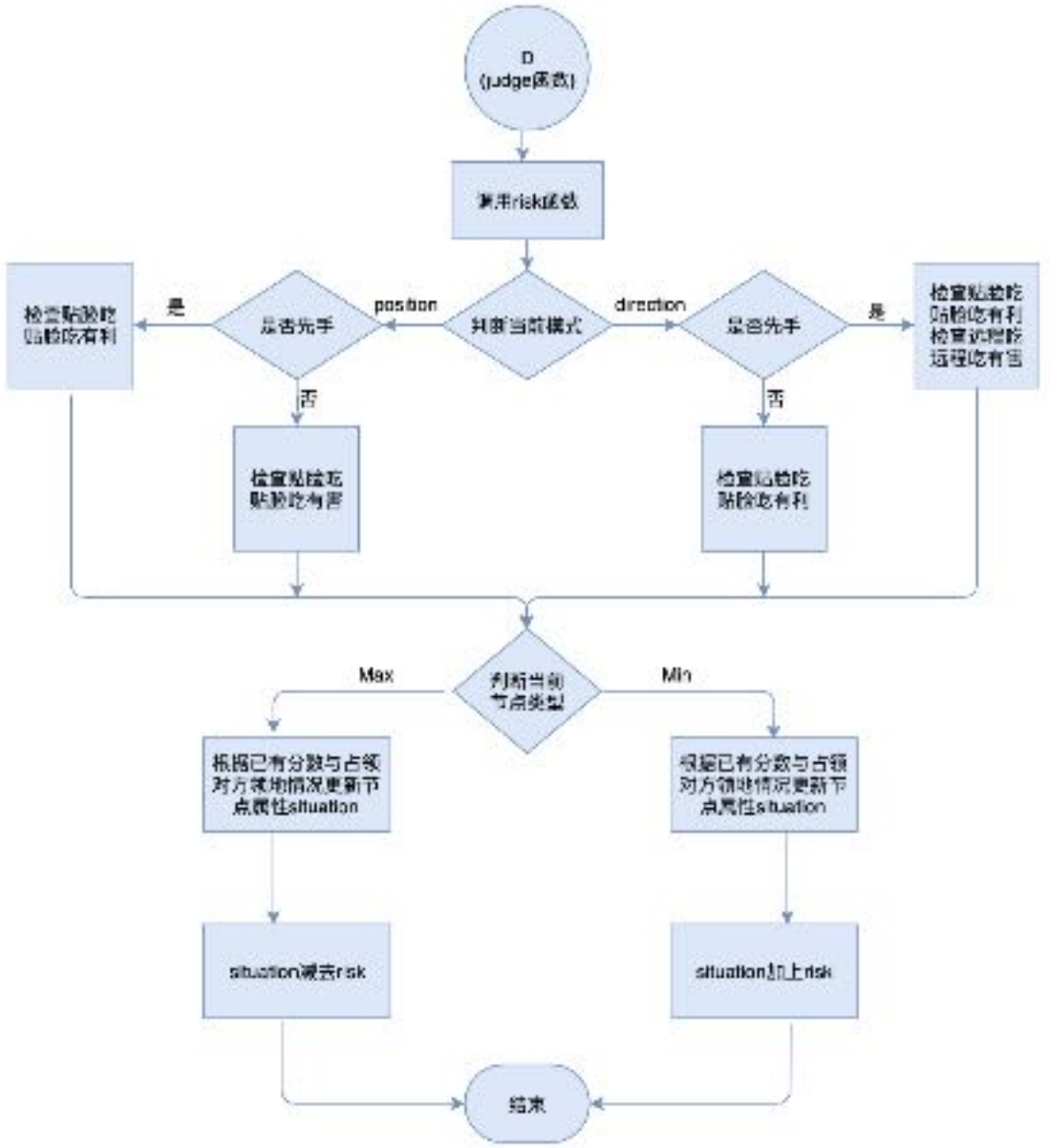
direction 模式：



grow 函数：


C
(grow函数，参数为当前高度height,最大高度max_height)
是否有当前节点height<max_height及currentRound<500
否
调用judge函数，更新节点属性situation, alpha, beta
D
(judge函数)
是
是否有子节点
是
对所有子节点递归调用
C
(height+1, max_height)
否
调用getAllActions函数，得到当前节点的所有可行操作
all_actions是否为空
是
否
是否有当前节点alpha<beta
否
是
从all_actions中删除一个操作，将该操作对应的子节点进行扩展
对该子节点递归调用
C
(height+1, max_height)
结束
调用select函数，返回最佳操作
是
是否height为零
否
调用judge函数，更新节点属性situation, alpha, beta
D
(judge函数)

judge 函数：



1.3 算法运行时间复杂度分析

算法的运行时间主要由生长搜索树和局面判断两个模块构成，其中生长耗时大约为判断耗时的 6 倍。

目前测得生长和局面判断耗时都为 O（2^n），n 为最大生长层数。

本地测试一局比赛一方平均耗时 17.45s。

position 模式下直接调用 position_judge 输出位置，理论上的时间复杂度为 O（1）。

direction 模式下生成 9 层的 ABTree，选择落子位置时生成一个子节点，选择移动方向时最多生成 4 个子节点，根据 α－β 剪枝的数学期望，平均生成两个子节点。每一个叶节点会

调用 judge 函数。预期的时间复杂度在 O（n）到 O（2^n）之间（n 为生成树的高度）。

2 程序代码说明

2.1 数据结构说明

整体上采用的是**树的结构**。

程序中使用了三个自定义类，ABNode 作为节点，ABTree 作为搜索树，Player 作为标准输入与输出。最主要的扩展是利用 α－β **剪枝**减少了无效的搜索，提高了搜索深度。

（1）Player 类：

在输出落子位置时，直接调用内部的评估函数进行输出；在输出方向时，会创建 ABTree 对可行的移动方向进行评估。

（2）ABTree 类：

抽象数据类型为树，只有__init__()与 output()两个函数，output()输出四个方向中判断最优的一个。

（3）ABNode 类：

节点类型，除了 children 和 parent，还有游戏局面（如 board 等）、all_actions、alpha、beta、type、situation 等参数，有递归调用自身的 grow 函数、评估局面的 judge 函数、创建子节点的 expand 函数等方法。

1、利用 **grow 函数**返回当前局面下最优的操作：grow 会**递归调用**自身，先获得可行的所有方向或者最佳的一个落子位置记录到 all_actions，为 all_actions 的每个元素 action 创建一个子节点，子节点的游戏局面随 action 更新，到达最大递归深度后进行评估，将评估的分数记录为 situation。

2、用节点的 type 属性对节点进行了分类，一类为对方操作的子节点 Min，一类为己方操作的子节点 Max，Min 节点会选择 situation 最低（即我方局面最差）的子节点的操作，Max 节点则与之相反（此处假设对手与己方都会选择最佳策略（在己方评估函数的判断下））。

3、α－β 剪枝：在 grow 进行递归调用的时候会传递 alpha，beta 值，前者为己方最大赢面，后者为己方最小赢面，要在子节点的最小赢面中找到最大的一个子节点作为最佳操作。对于一棵尚未完全展开的子树，若是某个叶节点的 situation 小于同层的已经完全展开子树的最小赢面，之后的搜索都不必继续，以此达成减少无效搜索的目的。

4、评估函数的优化：会调用 risk 函数，把可以直接吃掉对方以及被对方吃掉的情况加入评估的 situation 值中，**利用先验知识近似达到多展开一层的效果**。同时加上了对侵占领地的判定和相应的分数变动，但是目前看来这一步的作用并不明显，主要是因为难以把握分数变动的幅度。

2.2 函数说明

2.2.1 节点类 ABNode

属性

1. self.board 棋盘局面，Chessboard 类
2. self.currentRound 当前轮数，int 类
3. self.mode 游戏进行状态，str 类，‘position’为落子阶段，‘direction’为选定方向阶段
4. self.isFirst 此节点执子方是否为先手方，bool 类，True 为先手，False 为后手
5. self.type 此节点类型，bool 类，True 为 max 节点，False 为 min 节点
6. self.parent 父节点，ABNode 类
7. self.children 子节点的字典，dict 类，以操作为 key，子节点为 value
8. self.all_actions 当前局面下的可行操作，list 类
9. self.situation 当前节点尚未落子的时候的局势好坏，int 类
10. self.alpha = alpha 此节点的 alpha 值，即根节点对应玩家的赢面最小值，int 类
11. self.beta = beta 此节点的 beta 值，即根节点玩家对应的赢面最大值，int 类

方法

1. __init__(self, board, currentRound, mode, isFirst, parent=None, Type=True,
   alpha=-4 ** 11, beta=4 ** 11):

   初始化节点

   【参数】：

   board 棋盘局面，Chessboard 类

   currentRound 当前轮数，0~499 的整数

   mode 游戏进行状态，‘position’或‘direction’

   isFirst 此节点执子方是否为先手方，True 或 False

   parent 父节点，缺省值为 None

   type 此节点类型，缺省值为 True

   alpha 当前节点的赢面下限，缺省值为-4 ** 11

   beta 当前节点的赢面上限，缺省值为 4 ** 11

   【返回】：创建了一个初始化的 ABNode 对象，无 return 值

2. getAllActions(self):

   获取当前节点局面下的所有可行操作

   【参数】：无

   【返回】：可行操作列表，若为 position 模式，则列表中每个元素为一个表示可落子位置的元组，若为 direction 模式，则列表中每个元素为 0，1，2，3 中的一个整数，表示移动方向，若没有可行的操作，则列表中为唯一元素 None

   【原理】：

   在落子时，**首先判断是否有能保护当前节点一方棋子的位置**，如果存在则返回该落子位置坐标，**否则判断是否存在可以为对方添堵的落子坐标。**若有则该位置为唯一选择，**若没有，则遵循能落己方落己方，不能则随机落对面的原则**，即 position 模式下永远只有一种可行操作。

在选择方向时，**若对方已经卡死，则可行操作中不加入向己方移动，除非另外三个方向均无法移动。若对方没有卡死，则可行操作中添加所有可移动方向。**

3. judge_position(self, board):

根据当前局面判断是否可以通过在对方落子来避免对方吃掉自己的大子

【参数】：

board 当前棋盘

【返回】：True/False, Tuple[int, int]

【原理】：

首先根据该节点对应的先后手分为两种情况，两种情况下的原理相同，下以先手为例说明：

继续分五种情况：（行列均从一开始计数）

(1). 己方棋子在第五列，扫描竖向是否有对方相同大小棋子

对于不同行数分情况讨论扫描，若在第一行，则需要扫描第三行和第四行，如果有相同大小棋子，则返回 True 和己方棋子和对方棋子之间的位置。若在第二行，只需扫描第四行，如果有相同大小棋子，则返回 True 和己方棋子和对方棋子之间的位置。第三行和第四行的情况相同。

(2). 己方棋子在第五列，扫描横向是否有对方相同大小棋子

不需要对行数分情况讨论，只需要逐个扫描该行第七八列是否有和己方棋子相同的对方棋子，若有，返回 True 和己方棋子和对方棋子中间的位置。

(3). 己方棋子在第四列，扫描横向是否有对方相同大小棋子

同 2

(4). 己方棋子在第六列，扫描纵向是否有对方相同大小棋子

同 1

(5). 己方棋子在第六列，扫描横向是否有对方相同大小棋子

同 2

如果这五种情况都没有找到合适的落子位置，返回 False 和空元组

4. wintime(self, board):

根据对手剩余空位以及空位周围的情况判断是否可以通过在对方盘面内落子以达到添堵的目的

【参数】：

board 当前棋盘

【返回】：True/False, Tuple[int, int]

【原理】：

如果对方的空位个数小于等于 3，逐个扫描对方空位，如果某空位周围所有对方领地的值都大于等于 4，返回 True 和该空位的位置，如果扫描完全部空位发现没有符合条件的空位，返回 False 和空元组。

5. expand(self, action):

根据当前节点的局面及所选择的操作拓展子节点

【参数】：

action 选择的可行操作

【返回】：拓展出的子节点

【 注 】：**由于该游戏为对抗类游戏，故应当重视吃掉对方的棋子对盘面的影响**，而吃子是一个动态过程，不易在叶节点通过静态的局面来评估。故在每次拓展子节点时，子节点先继承父节点的局面值，若拓展的为选定方向操作，再**判断是否吃子并给予相应奖励或惩罚**。

6. select(self):

递归时选择当前节点执子方的最优操作，并相应修改局面值

【参数】：无

【返回】：当前节点的最优操作

7. grow(self, height=0, max_height=2):

搜索树的核心操作，αβ 剪枝生长。**通过构建搜索树来获取最佳决策，以 αβ 剪枝来简省不必要展开的节点**

【参数】：

height 当前节点生长到的高度，缺省值为 0

max_height 树的最大生长高度

【返回】：当整棵搜索树全部生长完成后，在根节点返回最佳决策，搜索树上的其他节点生长完后无返回值

8. judge(self):

对当前局面进行评估。

【参数】：无。

【返回】：situation 值。

【原理】：

此处评估函数由三部分组成。

第一部分评估总数值和合成奖励，首先使用己方与对方相等的棋子个数相减，然后乘 4 的相应阶数次方，之后求和（使得合并操作可以得到奖励），由于 2*4**1 < 4**2，即两个相等棋子合成后数值更高，因此奖励了合成。

第二部分评估侵占领地奖励，使用己方领地块数减去对方领地块数，然后乘 4 的最大阶数减二次方（~~表达能力有限~~），此处的数值为多次试验得到。

第三部分评估风险，通过调用函数 checkRisk 实现。

9. find(self, position, direction, n = 5):

输入对方棋盘内的空位，如果**在输入方向上**找到一个对于对方来说直接暴露在外（即可以被在该空位放置的一个相同大小的数通过 move 操作吃掉）的**不小于 n 的棋子**，返回 True ~~（是不是有点拗口）~~，**本质上是一个用来判断在输入位置落子是否可以保护大子被吃掉的函数**

【参数】：

position 起始搜索位置

direction 搜索方向

n 期待数字的最小值，缺省值为 5

【返回】：若可以找到则返回找到的数字，否则返回 False

【 注 】：position 必须为对方的盘内空位，否则无法达到预期效果

【原理】：

			+4	-0			
+7	+0	+0	**+0**	-6			
			+4	-0			
			+8	-0			

以上图中红色位置为 position 为例，执子方为后手

direction = 0 时返回 False（搜索到的 4 小于 5）

direction = 1 时返回 False（虽然 4 后面有大于 5 的 8，但是被 4 挡住了，搜索到 4 就会停止）

direction = 2 时返回 7（在 posiiton 防止 7 可以通过 move 操作吃掉（1，0）的 7）

direction = 3 时返回 6（在 posiiton 防止 6 可以通过 move 操作吃掉（1，4）的 6）

10. checkRemote(self, position):

以输入的位置为出发点，搜索是否存在过该位置的水平方向或竖直方向的**对于执子方的潜在的大于等于 5 的棋子被吃威胁（即不是贴脸被吃）**

【参数】：

position 搜索出发节点坐标

【返回】：若存在威胁则返回可能损失的分数，否则返回 0。可能损失的分数按照 judge 中的约定计算

【 注 】：position 必须为对方的盘内空位，否则无法达到预期效果

			-7	-0			
+6	+0	+0	**+0**	-0	-6		
			+0	-0			
			+7	-0			

以上图中红色位置为 position 为例，执子方为后手

则水平方向无风险，竖直方向上有风险，返回竖直方向上的可能损失分数

11. inboard(self, position):

判断输入的位置是否在棋盘中

【参数】：

position 待判断的位置坐标

【返回】：在棋盘内返回 True，否则返回 False

12. checkClose(self, position):

判断输入的位置是否存在被从上下左右紧贴的大于 3 的棋子吃掉的风险（即贴脸被吃）

【参数】：

position 待判断的棋子坐标

【返回】：存在风险则返回可能损失的分数，分数按 judge 约定计算；不存在则返回 0

13. getItsPosition(self):

根据当前节点的执子方先后手（isFirst）返回对方棋盘中紧贴己方的空位的列表

【参数】：无

【返回】：当前非执子方棋盘内紧贴执子方的空位坐标（元组）的列表

14. getMyBorder(self):

根据当前节点的执子方的先后手（isFirst）返回其处在前线的棋子坐标的列表

【参数】：无

【返回】：当前执子方的处在前线的棋子位置坐标（元组）的列表

15. checkRisk(self):

根据当前棋盘状况返回对于执子方的风险评估

【参数】：无

【返回】：当前执子方的风险，具体值为可能被吃而损失的分数。为正说明有害，为负则有利。

2.2.2 搜索树类 ABTree

属性

1. self.root 搜索树的根，ABNode 类

方法

1. __init__(self, currentRound, board, mode, isFirst):

初始化搜索树

【参数】：

currentRound 当前轮数，0~499 的整数

board 棋盘局面，chessBoard 类

mode 游戏进行状态：‘position’或‘direction’

isFirst 我方是否为先手，True 或 False

【返回】： 创建了一个初始化的只有根的 ABTree 对象，返回该树

【 注 】：传入的参数用于生成搜索树的根

2. output(self):

搜索树从根节点生长，获取最佳决策

【参数】：无

【返回】：树认为的最佳决策

【 注 】：当剩余时间大于 0.5 秒时，搜索树的搜索高度设置为 9，剩余时间大于 0.1 秒时，搜索树搜索高度设置为 5，剩余时间在 0.1 秒以内时，搜索树搜索高度设置为 2。

2.2.3 玩家类 Player

属性

1. self.isFirst　我方是否为先手，bool 类，True 为先手，False 为后手
2. self.array　开局给定的随机序列，tuple 类
3. self.MCTS　搜索树，ABTree 类，初始情况下为 None~~（为何该属性叫 MCTS，这是个历史遗留问题）~~

方法

16. __init__(self, isFirst, array):

初始化玩家

【参数】：

isFirst　我方是否为先手，True 或 False

array　给定的随机序列

【返回】：　创建了一个初始化的玩家，无 return 值

17. judge_position(self, board): # 用以判断是否有能够保护己方棋子的位置

根据当前局面判断是否可以通过在对方落子来避免对方吃掉自己的大子

【参数】：

board 当前棋盘

【返回】：True/False, Tuple[int, int]

【原理】：首先根据该节点对应的先后手分为两种情况，两种情况下的原理相同，下以先手为例说明：

继续分五种情况：（行列均从一开始计数）

(1). 己方棋子在第五列，扫描竖向是否有对方相同大小棋子

对于不同行数分情况讨论扫描，若在第一行，则需要扫描第三行和第四行，如果有相同大小棋子，则返回 True 和己方棋子和对方棋子之间的位置。若在第二行，只需扫描第四行，如果有相同大小棋子，则返回 True 和己方棋子和对方棋子之间的位置。第三行和第四行的情况相同。

(2). 己方棋子在第五列，扫描横向是否有对方相同大小棋子

不需要对行数分情况讨论，只需要逐个扫描该行第七八列是否有和己方棋子相同的对方棋子，若有，返回 True 和己方棋子和对方棋子中间的位置。

(3). 己方棋子在第四列，扫描横向是否有对方相同大小棋子

原理同 2

(4). 己方棋子在第六列，扫描纵向是否有对方相同大小棋子

原理同 1

(5). 己方棋子在第六列，扫描横向是否有对方相同大小棋子

原理同 2

如果这五种情况都没有找到合适的落子位置，返回 False 和空元组

18. inboard(self, position)：

用于判断某位置是否在棋盘内

【参数】：

position 需要判断的位置坐标

【返回】：True/False

19. wintime(self, board):

根据对手剩余空位以及空位周围的情况判断是否在对方落子

【参数】：

board 当前棋盘

【返回】：True/False, Tuple[int, int]

【原理】：如果对方的空位个数小于等于 3，逐个扫描对方空位，如果某空位周围所有对方领地的值都大于等于 4，返回 True 和该空位的位置，如果扫描完全部空位发现没有符合条件的空位，返回 False 和空元组。

20. output(self, currentRound, board, mode):

向对战平台最终输出我方做出的决策

【参数】：

currentRound 当前轮数，0~499 的整数

board 棋盘局面，chessBoard 类

mode 游戏进行状态：‘position’或‘direction’

【返回】：我方做出的决策，position 模式返回表示落子位置的元组，direction 模式返回代表移动方向的整数 0，1，2，3 中的一个，此外，若当前我方无可行操作，则返回 None

【 注 】：具体输出时，**若需输出落子位置**，则**先调用 judge_position 检查是否有能够保护己方棋子的位置**，若有，直接输出该位置，**若没有，则继续调用 wintime 检查是否可以塞住对方**，若可以则直接输出该位置。**若前两个函数都未返回合适位置，则己方有空落己方，否则随机落在对方**。若需输出方向，则创建 ABTree 并调用其 output 获得输出。

2.3 程序限制

1、用以判断落子位置的函数只扫描第四列和第五列的棋子（判断是否有被吃掉的风险以进行保护），对于**进攻更加深入的棋子无法进行保护。**

2、**选择落子位置完全利用先验知识**，在没有明显的最佳落子位置时优先落在己方，很多情况下可能并不是最优解，也可能出现自己把自己堵住的情况。

3、最大展开层数为 9 层，如果相邻空格存在四次以上的连吃情况无法进行有效的判断。

4、评估函数：对于对方在己方棋盘的棋子，不同大小的棋子改变的评估值相等，可能会导致己方一个大子被吃，对面两个小子被吃反而判断为优的不合理情况。

3 实验结果

3.1 测试数据

3.1.1 实验环境说明

硬件配置：（CPU/内存）

CPU：Intel(R) Core(TM) i7-9750H CPU @ 2.60GHz

基准速度： 2.59GHz

插槽： 1

内核： 6

逻辑处理器：12

虚拟化： 已启用

L1 缓存： 384Kb

L2 缓存： 1.5Mb

L3 缓存： 12.0Mb

内存：16.0 GB

操作系统：（名称/版本）

Windows 10 家庭中文版：

版本号 1909

操作系统版本 18363.836

Python 版本：（版本号）

python 3.8 (32 bit)

IDE：

thonny3.2.6

ai 版本：

最终版

3.1.2 测试方法

使用技术组开发的平台工具进行本地对战。

测试使用数据：

对局次数：20

搜索层数：9

3.1.3 测试结果

20 局消耗时间：348.93s

平均消耗时间：17.45s

20 局 grow 消耗时间：347.17s
grow 平均消耗时间：17.34s
20 局 judge 消耗时间：46.82s
judge 平均消耗时间：2.34s
PS: 编写的程序极为简单，无需展示。

3.2 结果分析

3.2.1 策略及起到的作用

本算法的**核心是搜索树和 α-β 剪枝**，由于所有的 AI 均采用此策略（官方推荐怎么可能不采用啊），此方法本身不具备超过其它 ai 的竞争力，报告开头已有介绍。

随着热身赛的不断进行，天梯代码的成熟度也越来越高，普通的 α-β 剪枝已经无法在天梯中胜出，于是我们第一个想到的方法是加深搜索层数。

观看的对局越来越多，我们发现绝大部分在对方棋盘的落子都没有很大的意义。同时由于在对方棋盘落子的选择很多，如果挨个进行搜索会极大的增加搜索难度，所以我们**在 AI 决策的时候舍弃了在对方领地落子的情况，不是通过树来判断落子，而是直接通过经验判断**，此方法极大地增加了 ai 的搜索深度，使 ai 获得了更强的战斗力。但这也带来了一个问题：落子操作的单一性，虽然后期也对落子操作进行了加强，比如说将棋子在适当的时候放在对面适当的位置，以防止对方吃掉己方棋子，或者在对方空间吃紧的时候为对方添堵达到塞爆的目的，但**不够灵活的落子操作的确是我们 AI 的一个缺陷**。

虽然搜索层次加深了，但由于纯粹的搜索终归是短视的，需要对评估函数加上一些带有先验知识的判断。虽然这也会带来调参的一些困难，但这毕竟是所有天梯前列的 AI 都绕不过去的一个坎

评估函数 judge 包括三方面的评估。

第一部分起到了促进己方合成更大棋子和吃掉对方棋子的作用，在前期对局中效果较明显，分数的比拼事实上也是判断输赢的唯一途径。

然而，由于 ai 搜索层数较低，不能准确预见之后的过程，这时就需要人工的经验进一步来评估当前的局面。

第二部分起到了促进 ai 侵占对方领地的作用，显然，领地大小也是当前局面的重要评估标准，己方的大领地意味着对方的小领地，既有利于增加己方的活动空间，也能使对方寸步难行。赋予侵占领地恰当的奖励之后，ai 达成了一种平衡，既不过于激进的侵占对方领地以至于己方的棋子被大量吞吃，又能适时侵占对方的领地造成对己方有利的局面。需要说明的是，由于度量侵占情况导致的分数变动的幅度较为困难，实际效果并没有想象中那么理想，侵占评估的效果应该还有进步的空间。

第三部分起到了规避风险的作用，通过粗略地判断当前局面是否存在被吃的风险，并赋予相应的奖励或惩罚，使 ai 能够评估当前局面的风险，从某种意义上说，这就**好像是为 ai 增加了搜索层数**，同时也更加符合人类玩家在下棋时的判断方式，增强了 ai 的战斗力。

此外，expand 中也包含了评估部分。

expand 中的评估部分主要目的在于奖励己方吃掉对方的棋子并且惩罚对方吃掉己方棋

子。在每次展开方向节点时，将移动前与移动后（即当前节点与子节点）的分数相比较，从而判断是己方棋子被吃还是己方吃掉对方棋子，并对当前节点的子节点的 situation 做出相应的改动，从而达到奖励和惩罚的作用。需要注意，由于调整数值比较困难，此处的数值并不一定为最优，还存在一定的进步空间。

另外一些小技巧虽然只是在特定情况下起作用，但还是能从整体上提高 ai 的战斗力。

比如使用 judge_position 和 wintime 函数来通过经验判断是否在对方落子。judge_position 的作用是在对方可以吞食己方的棋子的时候在对方领地内落子避免对方吞食己方，同时也可以干扰对方决策，wintime 的作用是在对方领地快要满了的时候在对方的大数中间落子，从而使对方更加周转困难。另外一个策略是在对方不能动弹的情况下只要不违规绝不后退，避免了到嘴的肥肉飞了的情况（痛打落水狗）。

3.2.2 算法的运行时间

（由于本地运行时间与天梯大为不同，未在 3.1 中所述环境下精确测量）

算法的运行时间较为合理，在天梯上既能在大部分情况下充分利用时间，又能通过最后减小搜索层数来保证不超时。在通常情况下，在己方活动回合数约为 360-400 回合的情况下消耗时间达到 4.5s，之后降低搜索层数，这样一般情况下不会超时间，如果剩余时间小于 0.1s，则再次降低搜索层数，几乎不会超时。

3.2.3 主要运行时间开销

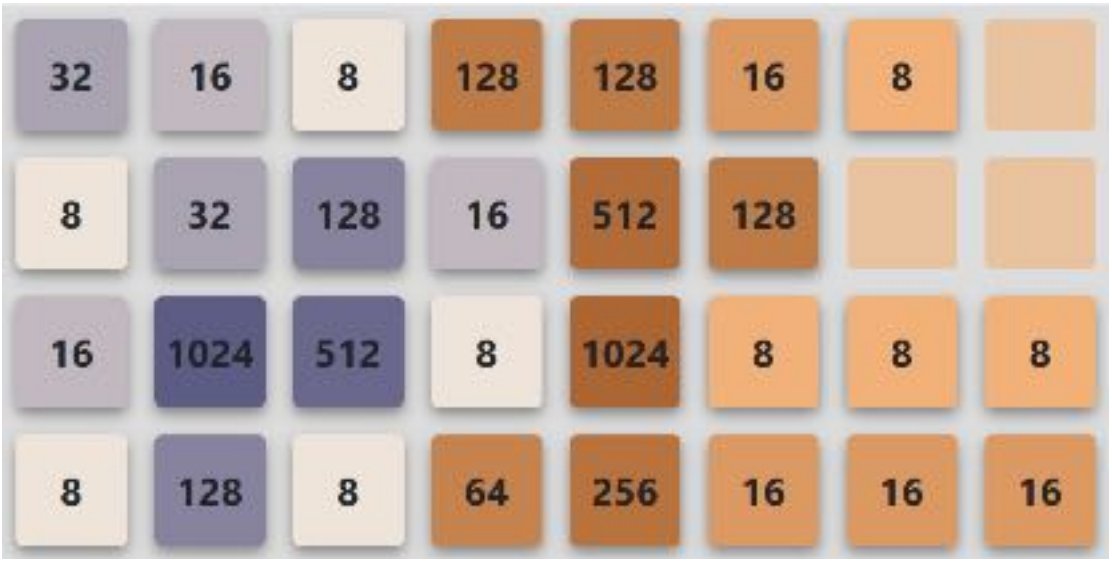
如 3.1 数据分析，算法的主要时间开销在于建树的过程。

3.3 经典战局（可选）

3.3.1 概况

我们组的 AI 的名字是 Thanos，取自“灭霸”，代表我们对它能像胖揍复仇者联盟一样威猛的期望与信心。

合理的侵占领地奖励使 Thanos 善于压爆，又能稳定发挥不过于鲁莽，深层的搜索使 Thanos 能更加准确的做出对自己有利的选择，在面对对手时有出色表现。



后手 thanos——成功实现压制

先手 thanos——即使不压制也能依靠紧凑的布局与远见获得优势

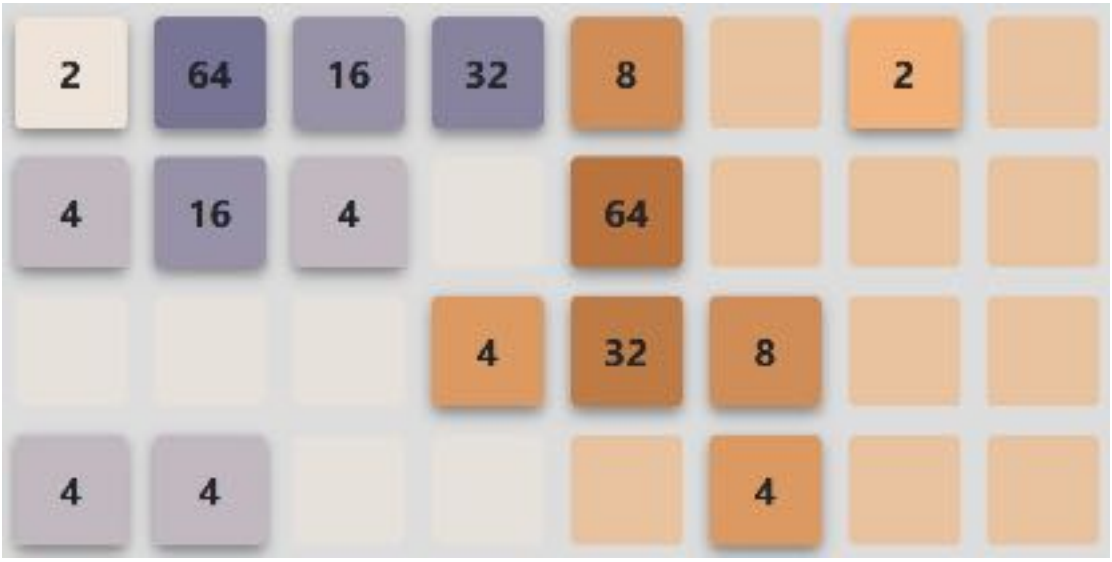
后手 thanos——没有侵占领地却实现了压制

3.3.2 几个例子

一、后面有大佬，勇敢往前冲（后手为 thanos）

最下面一列领地被对方侵占

thanos 左移合并



因为 4 上面有 8，对方不敢吃掉 4，只得把领地乖乖交还

二、及时补刀（后手为 thanos）

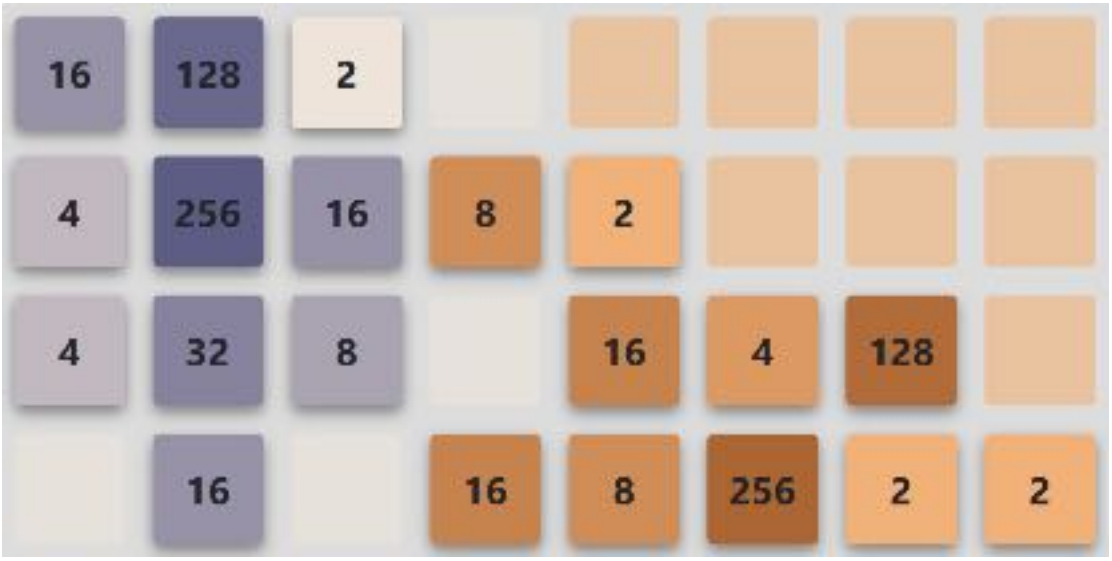
先手空位已经所剩无几

先手落子后只有一个空位，好机会！

现在你大概只能再移动一次了

三、乘虚而入，发展壮大（后手为 thanos）

转移阵地，干掉 4！


16 128 2
4 256 16 8 2
4 32 8 16 4 128
16 16 8 256 2 2

现在好了

4 128 4 16 2
8 32 8 8 8 2
4 256 2 2 128 256 2
2 64 16 32 4 4

好机会！

4 128 4 16 2
8 32 16 8 2
4 256 2 2 128 256 2
2 64 2 16 32 8

变得更加茁壮

4 实习过程总结

4.1 分工与合作

4.1.1 小组分工

陈海云：评估函数与落子位置判断函数的编写、代码的改进、内测、写实验报告
陈佳炜：代码的改进、后期先验知识的利用、写实验报告
冯杰超：主要的实验报告编写者
孙磊：代码的改进、写实验报告
俞越：α-β 剪枝树整体框架的搭建、代码的改进、测试、写实验报告

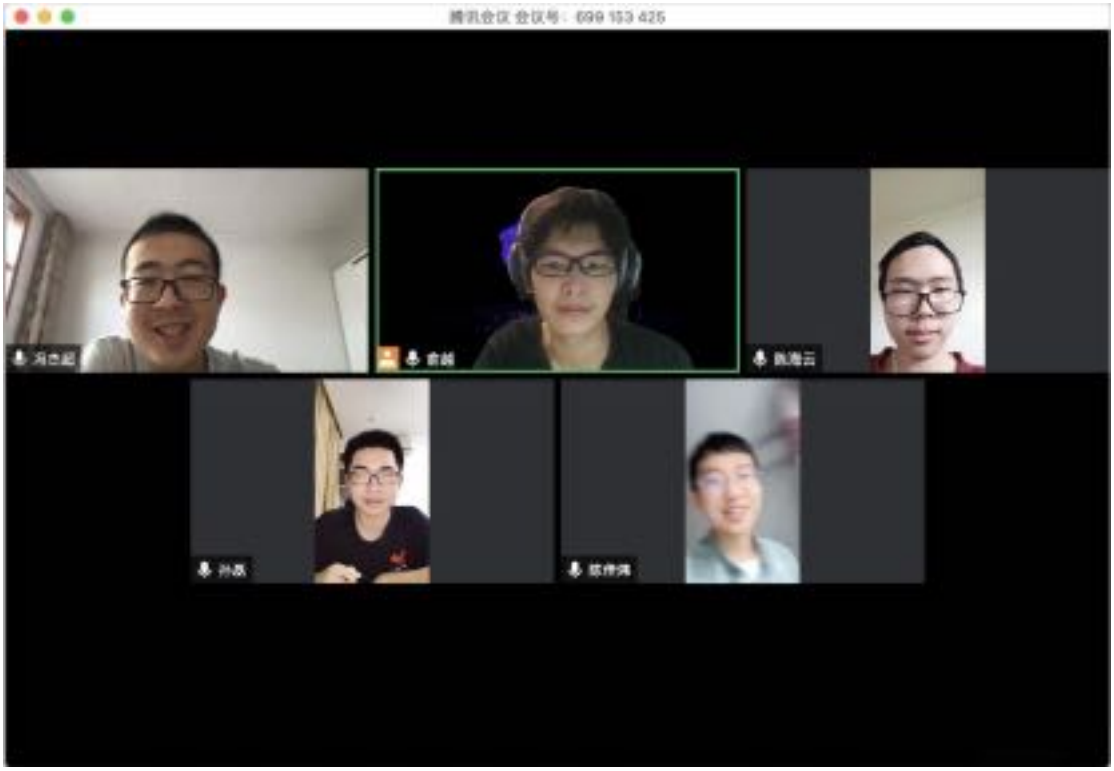
4.1.2 合作与交流的方式

每周一晚上利用微信群或者腾讯会议开组会
用 GitHub 上传每个人对代码的改进，记录实验日志
不定时在微信群里交流改进的想法和进度
正式实验报告由全组一起在金山文档共同编辑

组长试图 numpy 炼丹但奈何全连接网络不给力

冯杰超同学在看代码的过程中发现了一个大 bug

组内在 Github 上协同开发，共同撰写实验报告



陈佳炜同学快落到模糊

4.2 经验与教训

1、清晰的 ddl 和分工对于提高合作效率很有帮助
2、及时的交流能够减少重复的无用工作
3、清晰的代码和注释能加快后续工作速度

4.3 建议与设想

发现的一些小问题：
1、不知道出于什么原因，天梯赛的胜率相比本地比较不稳定，比赛随机性有点大。
2、热身赛的时候偶尔会出现大佬删码跑路的情况，导致无从测试（不过感觉也难以避免……）

5 致谢

感谢陈斌老师，在今年特殊的疫情期间坚持网上授课、答疑，尽管不能在现实中见面，但我们仍能感受到老师学识之渊博、见解之独到以及对待学生的热情与耐心。感谢助教与技术组、内测组的同学，为大作业的建设做出了主要贡献。最后还有我们组的组员，每个人都为本次大作业付出了许多，在此一并致谢。

6 参考文献

α-β 剪枝算法入门博文
https://blog.csdn.net/wenjianmuran/article/details/90633418