

SESSDSA2017
漂移乒乓大作业
实习报告集

SESSDSA2017

漂移乒乓大作业
实习报告集



2017 年 7 月

指导老师：陈斌
编辑：郭浩
特约撰稿人：郭晴宇
特约封面设计：罗哲楷

谨以此书献给

SESSDSA2017 全体成员

目录

Part I 作业描述

游戏设计与基础设施简介.....1
平台代码修改历史.....3
附录：关于旋转球影响游戏平衡性的讨论.....5

Part II 竞赛纪实

从热身赛到总决赛.....9

Part III 报告汇编

冠亚季军

Oscar.....19
Mike.....47
Golf.....73

八强队伍

Alpha.....86
Charlie.....100
Hotel.....119
India.....131
Victor.....156

东区 East

Foxtrot.....182
Echo.....194
Romeo.....221
Yankee.....249
Delta.....262

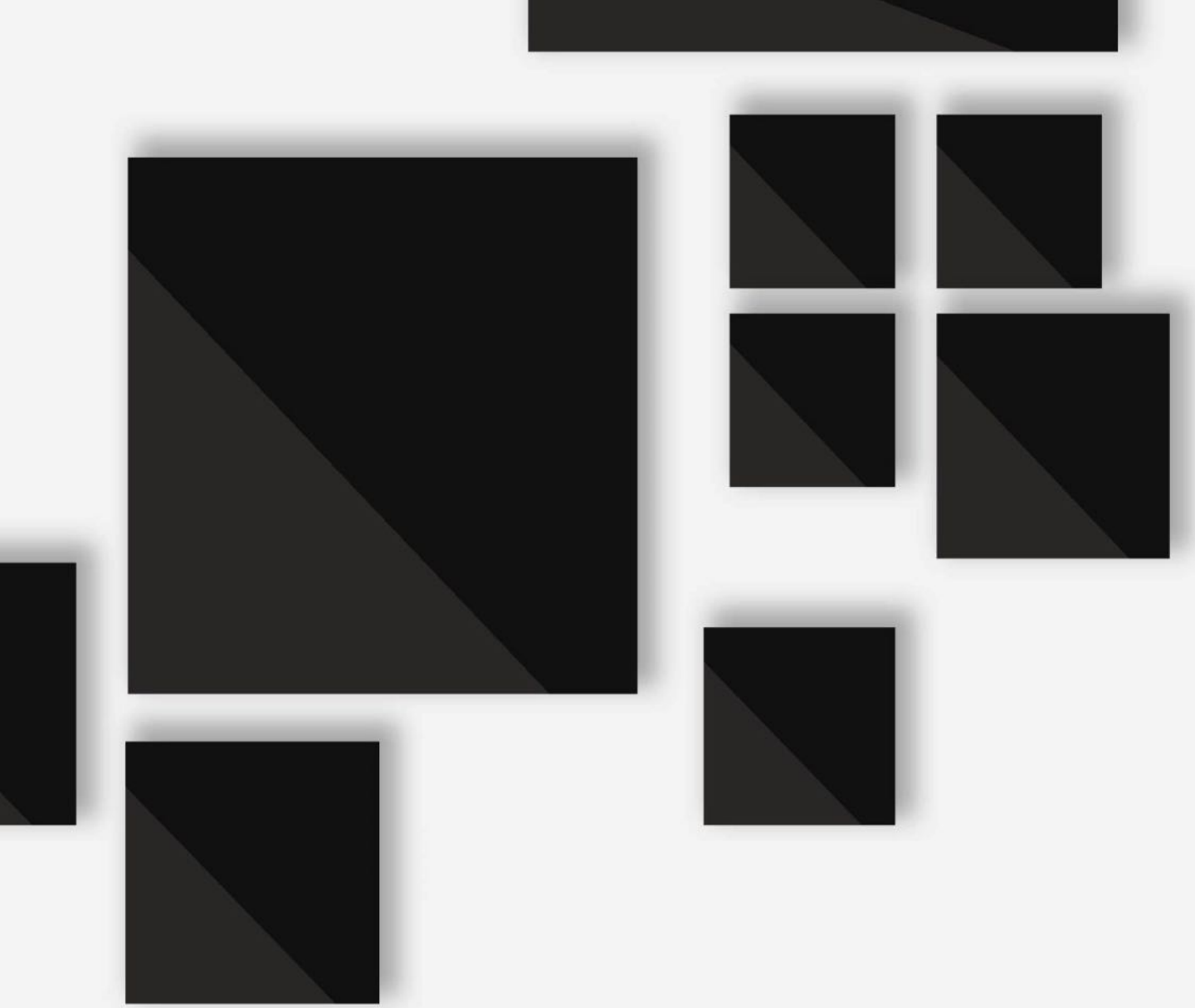
西区 West

Papa.....284
Lima.....296
Zulu.....315
Whiskey.....323

南区 South
Sierra.....351
Bravo.....377
Quebec.....391
November.....434

北区 North
Kilo.....440
Juliet.....465
Uniform.....477
X-Ray.....488
Tango.....498

编后记.....504



Part I

作业描述

游戏设计与基础设施简介

1 简介

漂移乒乓，是由 2017 春季数据结构与算法课程的期末大作业，以游戏的方式，促进了大家对数据结构与算法以及编程的理解和应用。漂移乒乓的主要规则由陈斌老师决定，其平台的核心代码也主要由陈斌老师编写。在这个过程中也有不少同学对规则提出了意见，或是单独编写了一些代码文件，使得这个项目逐渐得以完善，游戏的平衡性也逐渐增加。

2 游戏规则

漂移乒乓的主要规则如下：

游戏由双方在一个 1800000*1000000（长*高）的场上进行游戏。双方在游戏开始时均有 100000 体力值，而迎球、跑位、给球加速等动作会扣减体力值。双方轮流用球拍击球，直到有一方由于体力不支或者其他原因导致击球失败，则游戏结束。

首先由一方发球，发球的位置和纵向速度可以由发球方决定，但横向速度是 1000。接着对方进行迎球动作，也就是说将球拍移到球的落点位置，然后进行击球动作，这会使得球的横向速度反向，并且可以由玩家决定是否改变球的纵向速度，也就是是否加速。击完球后可以移动一段距离，即跑位。击出的球撞在上下边界上会发生反弹，横向速度不变而纵向速度反向。而击出球的合法条件是球碰上下边界 1-2 次，多或少都会被判为犯规。

在迎球，跑位，对球进行加速的过程中，迎球的距离，跑位的距离，对球改变速度的多少，都会按照二次函数的关系以一定的比例在玩家的体力之中扣减。而每次球能移动的最大距离是和体力值成正比的，如果体力不支，则会出现丢球的情况。

为了使得玩家能够做出更加合理的决策，双方都拥有一个字典形式的数据储存，用来保存历史数据。综合历史数据进行决策有可能会带来更好的效果。

为了增加游戏的趣味性和不确定性，游戏中引入了“道具”的机制。道具是由平台代码决定，会在一定的时间间隔内在场上随机出现的物品。一方在击出球后，如果球穿过某个道具，则击球方收集到该道具。在一方击球之前，玩家可以决定是否使用某个道具。场上的道具个数是有限制的，多于 10 个的道具将会被移除。玩家最多能够拥有 3 个道具，如果此时又收集到新道具，则会移除最旧的道具。道具分为六类，分别是“旋转球”、“隐身术”、“补血包”、“掉血包”、“变压器”和“瞬移术”。旋转球的作用是抵消下一回合对方对于球的加速的一半。隐身术的作用是隐藏被用道具放的位置和跑位方向一次。补血包和掉血包分别是给被用道具方加上和减掉 2000 体力值。变压器是将下一次对方迎球、加速和跑位时的体力消耗放大为两倍。瞬移术使被用道具方可以移动一段距离而不消耗体力值。道具的出现，使得玩家所需要考虑的策略变得更加复杂，这也增加了游戏的变化。

3 基础设施

为了实现这一平台，陈斌老师和几位同学编写了了几个基础文件 。

Table.py：这个文件中实现了几个关键的类，用面向对象的思路完成了大体的设计。这几个类主要是 Card（道具），CardBox（道具箱），Ball（球）,Racket(球拍)，RacketAction（球拍动作），Table(桌面)，TableData（桌面信息）。其中 Racket、Table 和 Ball 是最为关键的部分。Table 作为对战进行的媒介，有两个 Racket 和一个 Ball 作为成员变量。这两个 Racket

即是相应的两个玩家，在生成的时候回被指定与某个文件绑定，之后再进行动作的时候，就会调用玩家编写的相应函数。CardBox 是作为 Racket 的成员变量而存的。在游戏中，Table 会反复交换击球方和迎球方，每次交换之后调用 time_rum()函数。这个函数会调用玩家的决策函数和 Ball 的 fly()函数。 每次到了改到玩家做决策的时候，会将目前场上的所有信息打包成一个 TableData 并传递给 Racket 的决策函数。Racket 在决定策略之后，将相关的行动数据生成一个 RacketAction 再传回 Table。然后 Table 再根据 RacketAction 对 Ball 的数据进行修改。在球从一方飞向另一方的时候会调用 Ball 的 fly()函数，这也是游戏正常进行的关键。这个函数会根据 Ball 被改变后的状态，使其位置改变为飞过场后的正确位置，并且计算出碰撞了那些道具，撞了几次壁，并且将这些数据返回 Table，然后 Table 会将这些道具从场上去掉并加入 Racket 的 CardBox，并且根据碰壁次数来决定游戏是否分出胜负。

Pingpong.py：这个文件中定义了 race 函数，用来制定两个程序进行对战。整个比赛执行的代码也在这个文件中。它会将当前目录下所有字样为”T_***.py”的文件打开，将其迭代，使得每对玩家都能够有一场对决。而”T_***.py”文件中应当有三个函数，分别是 serve(op_side,ds)，play(tb,ds)和 summarize(……)，这三个分别是发球函数，击球函数和总结函数。发球函数接收对方的名字和自己的数据储存，做出发球决策。在一局比赛中，只会调用其中一方（先手方）的发球函数。Play 函数接收当前场上状况（类型为 TableData）和数据储存 ds 并且返回动作（类型为 RacketAction）。Summarize 函数是每局对决结束之后，双方都会被调用一次的函数，用于总结这次的比赛，主要是讲这次历史数据整理到自己的数据储存中。

Show.py：这个文件是由张颢丹同学利用 pygame 模块编写的可视化程序。在对战的过程中，场上的所有信息会被记录下来，最终被存储进文件并被放入当前文件夹中。Show.py 的作用就是读取这些文件中的信息，并在屏幕上演示出来。

Reader.py: 由郭浩同学编写。平台对局后数据会以 shelve 的形式储存起来，reader 的作用就是将对局的所有信息提取出来，生成一个便于阅读的 txt 文件，可用于调试时检查数据。

Mat_plot_battle: 由杨帆同学开发。这一模块可以用图表的形式呈现出对一局对局的战术分析，包括生命值变化，球的落点等。

data_analysis.py：由助教团为现场比赛开发，可以实现对各算法胜场数及排名的实时显示。随着平台的运行，胜场和排名的变化为比赛增添了紧张气氛。

data_analysis_cx.py：由助教团为现场比赛开发，可以用图表实现对竞赛对局的战术分析。

4 项目资料网址

大作业专题网页：http://gis4g.pku.edu.cn/course/pythonds/sessdsa17-pingpong/

Github：https://github.com/chbpku/pingpong.sessdsa/（平台代码下载）

注意：各组参赛算法作为本书配套电子资料，也会在 Github 上公布，欢迎下载！！

【编者注】：本文由特约撰稿人郭晴宇撰写，编者在收入本书时对个别地方进行了修改。

平台代码修改历史

“漂移乒乓”游戏是为本次大作业原创的新游戏，刚发布时在规则、代码、配套功能上难免有不足之处。项目发布后，同学们在开始编写代码的同时，积极参与到项目的完善之中。截至比赛结束，Github 平台上该项目开启了 33 个 issue，pull request 总量达到 40 次，旋转球 bug 等问题得到修正，辅助模块也一一补齐。最终完成的 pingpong 游戏是老师、助教、同学集体智慧的结晶。因此，编者特将代码修改历史收录如下，作为纪念。

20170515：

- 陈斌创建代码文件和 README.TXT 文件。

20170517：

- 陈斌根据同学建议完善了 PPT 规则，并在代码增加了发球和永久保存历史数据。

20170519

- 李逸飞重写了 fly 函数的主要算法，更加简洁；
- 陈斌更新了 Vector 判断相等方法__eq__，考虑误差+/-1；
- 陈斌修改了 play 函数中传递对方跑位方向；
- 陈斌增加了球拍移动的最大速度限制，与体力值相关，满体力时速度是球的横向速度 2 倍，按比例线性降低；
- 陈斌增加了最后一个回合的数据。

20170520

- 增加了常见问题 FAQ.md。

- 张颢丹开发 GUI，使用 pygame 模块，实现对战记录可视化回放。
- 陈斌增加了道具系统的代码。
- 李逸飞开发了道具获取的算法函数。
- 陈斌处理了迎球方返回速度和距离值，规整为整数。
- 郭浩开发 reader.py，将复盘数据转换为文本文件输出。

20170522

- 陈斌增加了算法第三个函数 summarize，可以用来保存终局的分析数据；
- 陈斌修改了旋转球的 param 为 0.5，被用道具一方对球的加速会减少 50%；

20170523

- 李逸飞修正了 fly 和 get_card 的 bug

20170527

- 陈斌在 serve 函数中增加一个参数，对手的名称，以便根据以往记录来决定如何发球

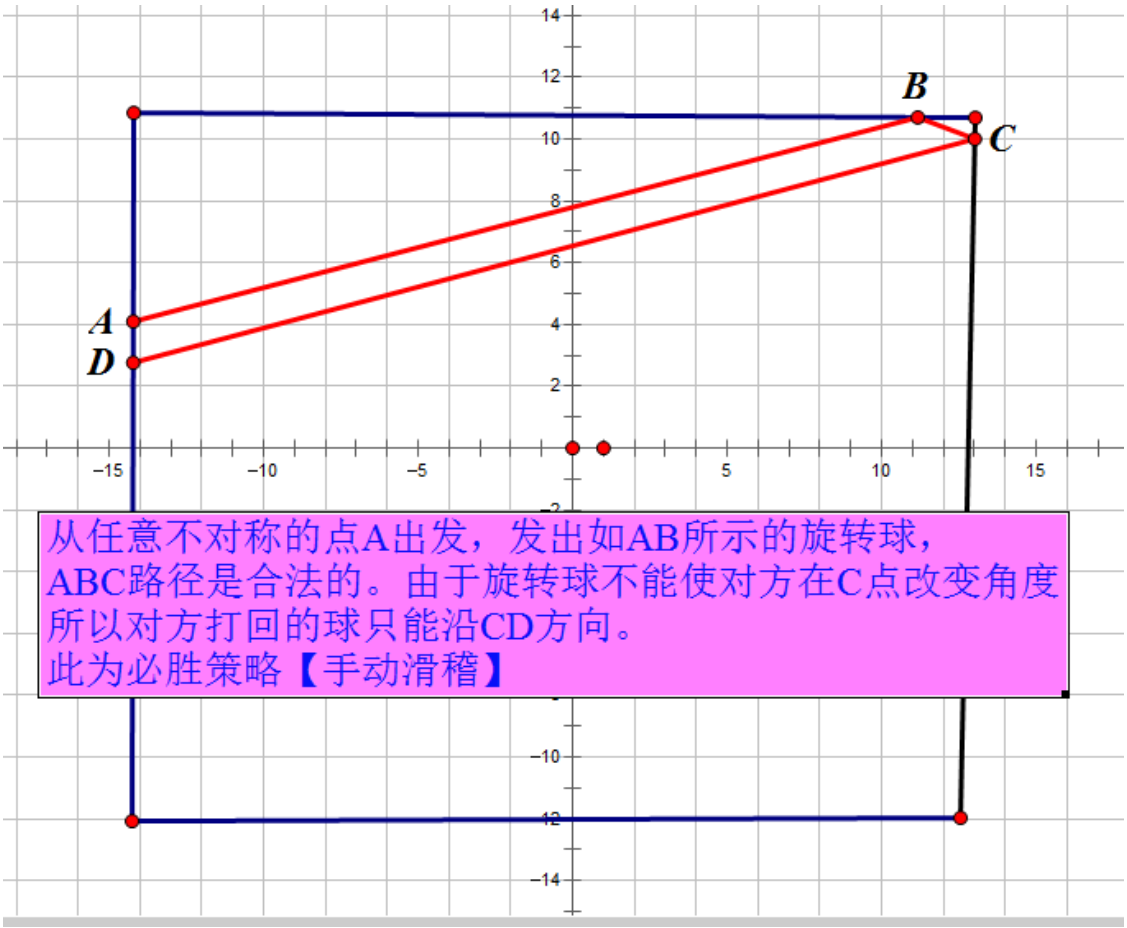
20170604

- 陈斌增加了物理时间限制，并给 play 传入各方消耗的物理时间 clock；
- 李逸飞发布了 data_analysis.py，竞赛过程成绩的实时可视化进度显示；
- 陈旭发布了 data_analysis_cx.py，竞赛复盘数据的战术数据统计。

附录：关于旋转球影响游戏平衡性的讨论

【编者按】：旋转球道具最初的效果是将对方的加速度全部抵消。针对这一设置，有同学发现可据此制定一击必杀策略，于是展开讨论。此段内容节选自 Github 上的 issue，可以作为一个典型例子反映平台开发中师生互动的情况。

Ribombalt：May 20



老师我之前汇报过一次的.....这个过于影响游戏平衡。

个人感觉这个属于设计漏洞，拿到旋转球的玩家拥有必胜策略，而旋转球的出现是随机的，独立于代码水平之外的，私以为用这个来决定胜负和掷骰子决定胜负没有什么本质区别。因此我建议，删除这个道具，或者削弱这个道具的功能，或者限定这个道具只能在残局（比如双方生命值均小于 30%，有大量范围无法运动到达）时出现。

Chbpku：May 20

那么，如果把旋转球的作用改成：将对方施加在球拍上的加速度减半。

这个如何？@RibomBalt

LuoZheKai：May 21

这招是有死角的吧。把球打到对方的角落处，让对方在角落处接球，球将会沿对角线运动，这样根据碰撞次数的计算方法，即使对方有旋转球也无法做到一击必胜了。

Ribombalt：May 21

@LuoZhekai 角落处还可以构造出那种弹两次的必杀情况。

@chbpku 感觉可以，不过 1/2 的系数还需要调整。这个会对策略有一定制约。

另外能否知道对方用了道具？如果可以知道，我有个建议，系数用 1/3 到 2/3 间的一个随机数，既保证了随机性，又可以进行一定的控制。

Chbpku：May 21

@RibomBalt 能知道对方用了道具。随机数的意思是用道具的一方不能预知效果？但是被用道具的一方能知道这个数？如果双方都不知道这个随机数，可能不太合理。应该说，旋转球的设置，不是为了一招制胜，但它是消耗对方体力的一个有效手段，让对方想达到预定效果击球的话，需要付出更多的体力。这个道具跟变压器类似，但也有区别。

Ribombalt：May 21

@chbpku 如果双随机不合理，也可以让发球方自主选择参数（调用道具那里加一个参数，可以用*args，不影响不需要参数的部分）。这样，发球方可以知道参数，参数必须限定在一个合法的范围内，接球方只能知道对方用了道具。

同时为了保证不会一招制胜，我建议，可选择的范围应该保证：接球方可以找到一个发球角度（与参数无关），使得他 按照这个角度算出的加速值对球加速，无论选择了什么参数，发出的球都是合法的（符合弹 1-2 次）。（下文把 这个加速值的邻域 称为安全区）。这样，接球方可以选择这个很小的安全区，但会导致发球方向不一定符合自己的利益，并且 安全区算法对方应该也能算出来，这一回合可以被道具方 预判；也可以不选择安全区，冒被秒杀的风险来制造随机性。

另外参数选择范围也可以和使用道具的一方位置有关，降低调参难度。这样也可以实现（比如给道具一个小函数），没有很大的技术困难。如果用一种隐晦的方式告诉大家安全区的存在，那我觉得就彻底解决了秒杀的问题，并且也能达到迷惑战局、消耗对方的目的。

Chbpku：May 21

@RibomBalt 参数合法范围如果是一个相对值的话，在规则复杂性和调试复杂性方面都会加大很多，但由于也还是要考虑到对方有不被秒杀的可能，其实际效果可能跟确定一个固定的加速倍数基本相同。而如果参数值固定为，比如将对方的加速削弱一半的话，也同样达到了加大对方体力值消耗，干扰其正常策略的目的。

Ribombalt：May 21

@chbpku 如果是固定值的话，假如对方不知道你用了道具，那么就比较有可能被秒杀【比如他想擦边打，结果对方用了道具，导致调整的角度不够弹 1-2 次】。假如知道，那就只剩下多消耗对方体力的目的了，不能增加战场的随机性。两害相权取其轻，我觉得还是让

对方知道你用了道具比较好。或者可以这样，让对方知道你用了道具，但是不知道你用了哪个道具。这样调试难度没有，然后也保证了一定的随机性。

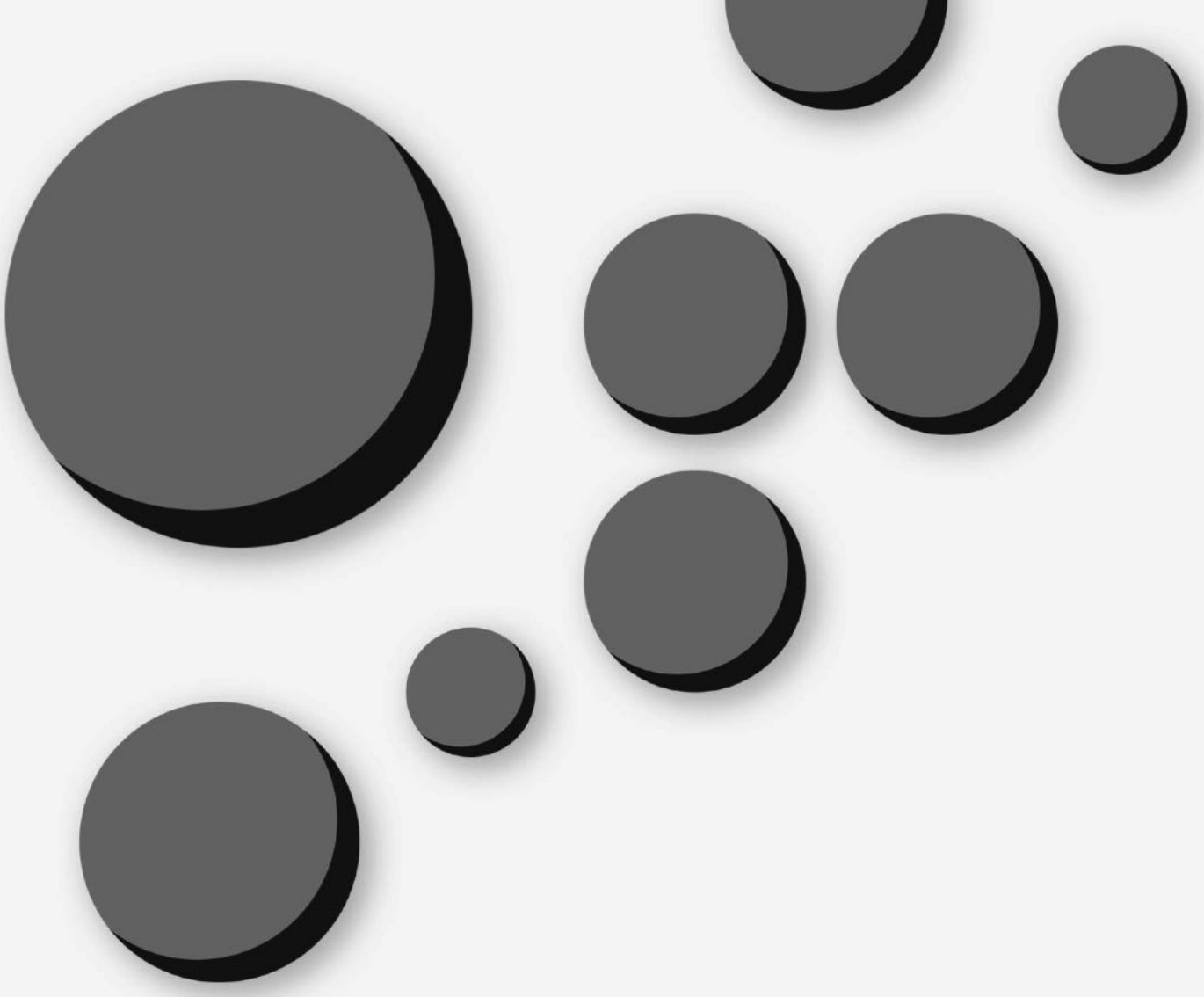
Chbpku：May 22

对方知道我使用的道具，当然对方也可以在没有危害的前提下选择不对球加速，就没有任何体力损失，相当于逼迫对方更换应对策略。

至于随机性，我希望平台所产生的随机性尽量少，可预测。把随机性的发挥留给双方算法去耍。

Ribombalt：May 22

可以。（Close issue）



Part II

竞赛纪实

从热身赛到总决赛

漂移乒乓大作业全过程历时数周。作业发布一周后，首先举办两次热身赛，各小组自愿匿名参加，以了解自己的实力。正式比赛分为分区循环赛、八强淘汰赛、四强半决赛和决赛四各阶段。下面对漂移乒乓大作业的竞赛过程进行回顾。

1.热身赛

5 月 29 日下午，地空数算现场举行热身赛。

热身赛的规则是：各组以匿名方式报送代码，可提交多个算法；最终所有算法分为上下两个半区，每个半区中所有算法进行循环赛，每两个算法对战 10 场。赛后助教将公布比赛结果和复盘数据，以便各小组调试。

由于设备电量不足，结果没能在现场公布。在两节课的时间里，各组组长一一发言，介绍本组的算法进展、分工、组会等情况。各位组长的发言，或诙谐幽默，或清晰严谨，给在场观众带来欢笑和思考。

十分幸运的是，校图书馆对本次热身赛进行了录像，大家还可登录图书馆网站观看，重温当时的欢笑，再览各位组长的风采。网址是：

http://media.lib.pku.edu.cn/index.php?m=content&c=index&a=show&catid=33&id=6318



视频截图（图为陶天阳（L）、王梦瑶（M）、闵靖涛（I）、李思彤（F））

2.分区赛&八强赛

6 月 6 日下午，地空数算 2017 期末大作业“漂移乒乓”现场竞赛，在理教 107 开打！26 个小组将在 CPU 和内存构成的虚拟空间里展开一场玄幻的球赛。这里有推来挡去，上下翻滚，还有满桌暗器，一触即发。

开赛前的躁动。助教已备好赛场——i7 7700K+16GB DDR4+250GBSSD。



从 A 到 Z，26 支球队已经准备就绪！



本次竞赛借鉴了世界杯足球赛的规则，26 支球队分属东南西北 4 个区，首轮的分区循环赛将从每个区出线 2 支球队，决出八强。

循环赛的规则：



北区是耗时最长的分区，原因是本区拥有两支高耗时球队：每局 6-8 秒的 OSCAR 和每局 2 秒左右的 JULIET，相比起他们来说，其他球队的耗时基本都低 2-3 个数量级！

最终大家见证了 OSCAR 不慌不忙绝杀全场，全胜出线，而 HOTEL 则以 5 胜 1 负出线。从整体看，北区实力的座次排得相当整齐。



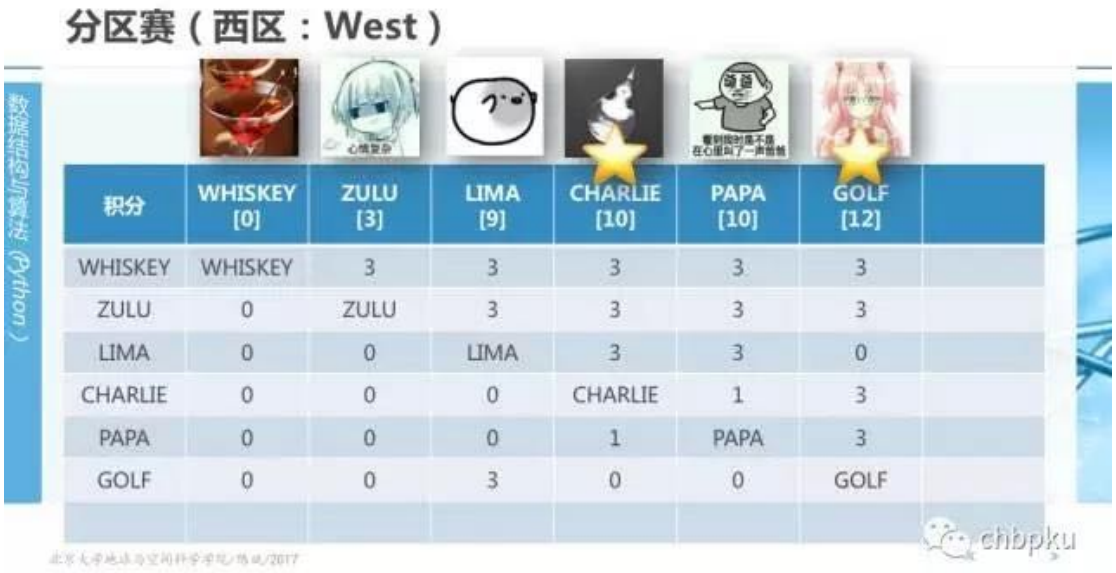
积分	HOTEL [15]	KILO [12]	TANGO [0]	UNIFORM [6]	OSCAR [18]	JULIET [9]	X-RAY [3]
HOTEL	HOTEL	0	0	0	3	0	0
KILO	3	KILO	0	0	3	0	0
TANGO	3	3	TANGO	3	3	3	3
UNIFORM	3	3	0	UNIFORM	3	3	0
OSCAR	0	0	0	0	OSCAR	0	0
JULIET	3	3	0	0	3	JULIET	0
X-RAY	3	3	0	3	3	3	X-RAY

东区各球队实力梯次也是相当整齐，MIKE 以全胜战绩轻松出线，但 FOXTROT 和 VICTOR 则各积 13 分紧随其后，相等的积分触发了体力值排序规则，结果 VICTOR 以胜局中保留的更多体力值，以小组第二身份出线，FOXTROT 则遗憾出局。



积分	YANKEE [3]	FOXTROT [13]	VICTOR [13]	ROMEO [6]	DELTA [0]	MIKE [18]	ECHO [9]
YANKEE	YANKEE	3	3	3	0	3	3
FOXTROT	0	FOXTROT	1	0	0	3	0
VICTOR	0	1	VICTOR	0	0	3	0
ROMEO	0	3	3	ROMEO	0	3	3
DELTA	3	3	3	3	DELTA	3	3
MIKE	0	0	0	0	0	MIKE	0
ECHO	0	3	3	0	0	3	ECHO

西区局势有点诡异，小组第一的 GOLF 并没有占据绝对优势，对 LIMA 的攻势受挫，但 LIMA 又败给了并列第二的 CHARLIE 和 PAPA 队，形成一个相克循环。CHARLIE 和 PAPA 则因为相持不下，战成平局，以 2 分之差屈居 GOLF 之下。同时，又触发了残酷的体力值排序规则，这样，PAPA 体力落后，CHARLIE 获得出线权！



积分	WHISKEY [0]	ZULU [3]	LIMA [9]	CHARLIE [10]	PAPA [10]	GOLF [12]
WHISKEY	WHISKEY	3	3	3	3	3
ZULU	0	ZULU	3	3	3	3
LIMA	0	0	LIMA	3	3	0
CHARLIE	0	0	0	CHARLIE	1	3
PAPA	0	0	0	1	PAPA	3
GOLF	0	0	3	0	0	GOLF

南区也是波澜不惊，**ALPHA** 以全胜的绝对优势出线，**INDIA** 则以 4 胜 1 负战绩分区第二出线。

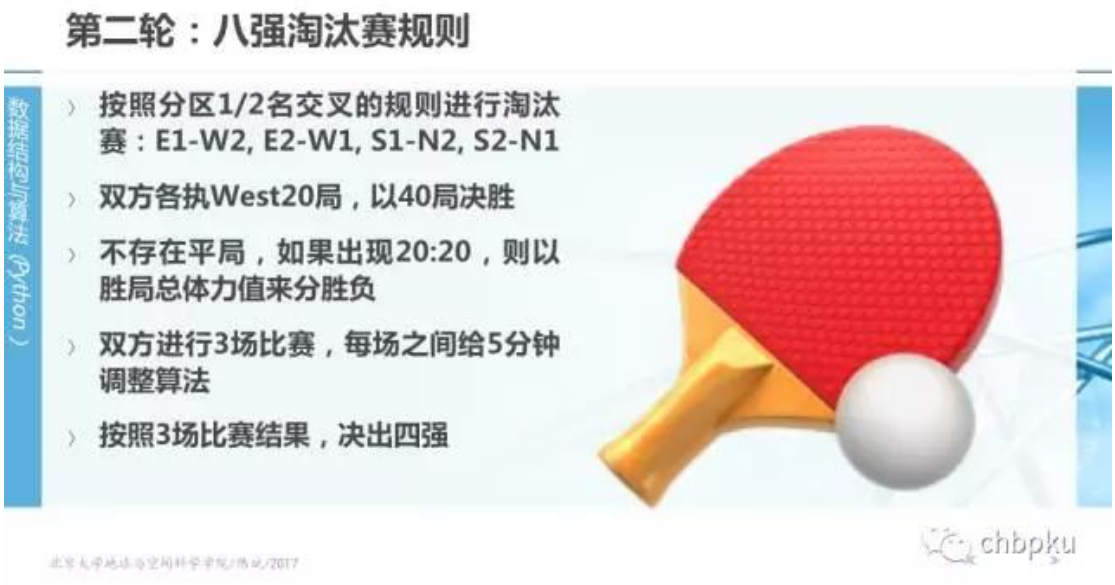


积分	INDIA [12]	BRAVO [6]	QUEBEC [4]	SIERRA [7]	NOVEMBER [0]	ALPHA [15]
INDIA	INDIA	0	0	0	0	3
BRAVO	3	BRAVO	0	3	0	3
QUEBEC	3	3	QUEBEC	1	0	3
SIERRA	3	0	1	SIERRA	0	3
NOVEMBER	3	3	3	3	NOVEMBER	3
ALPHA	0	0	0	0	0	ALPHA

这样，四区八强决出：北区 OSCAR、HOTEL；东区 MIKE、VICTOR；西区 GOLF、CHARLIE；南区 ALPHA、INDIA。

经过紧张的数据整理，部分八强球队调整了算法，八强淘汰赛鸣锣开赛。

为了最大限度排除偶然性，八强淘汰赛大幅增加了对局数，双方须对战 40 局，并经过 3 场比赛才能分出胜负。每场之间，球队还可以依据上一场的战绩和数据，调整算法再战。



八强赛要求无平局地决出胜负，更加残酷激烈，虽然已有未获出线的球队离席，分散在教室各个角落的八强队更显紧张。

HOTEL 组紧张分析数据修改算法。ALPHA 针锋相对，与 HOTEL 展开拉锯战！





经过两场竞赛，东区第一的 MIKE 和北区第一的 OSCAR 已稳入四强，但另外两对则在第二场提交了修改后的代码后，出现了逆转，GOLF 和 ALPHA 恢复了两区第一的风采，并在第三场成功翻盘！

至此，各区的第一名成为“漂移乒乓”四强，第三轮半决赛将在 6 月 8 日（周四）上午 10 点，仍然在理教 107，展开激烈角逐。

3.半决赛&决赛

6 月 8 日，漂移乒乓前三名的角逐如期展开。Oscar 早已成为江湖中众人称赞的第一强队，Mike 小组赛 120 胜 0 负的战绩也着实令人敬服。面对强手，Alpha 和 Golf 也未放弃：Alpha 组员认真研究了八强赛 Mike 与 Charlie 的对战数据，Golf 更是将同组 Charlie、Lima 两个实力不俗的算法借来研究、测试，使自己算法的战力再上一个台阶。



比赛前播放了 Golf 组友情提供的真人版乒乓暖场视频。

四强的批处理文件刚刚运行，Mike 与 Alpha 的结果很快出来，Mike33:7 赢下第一盘。Alpha 调整算法后第二盘取得 18:22，有所提升，但终究没有挡住 Mike 前进的步伐。

而 Golf 与 Oscar 的比赛因 Oscar 树搜索算法的长耗时迟迟没有结果。等待结果期间，陈斌老师对本学期的课程活动如递归树、树莓派编程活动等进行了回顾。最终 Oscar 果然取胜，而 Golf 单盘打败 Oscar10 局以上的表现也值得肯定。



Golf 组的瞬间



严阵以待的 Oscar 组

三四名决赛中，Golf 以 37:3 和 27:13 连下两盘，夺取季军。Alpha 屈居第四，获得了 M&M 花生牛奶巧克力豆的特别奖励。

最后的悬念是 Oscar 与 Mike 的巅峰对决。开始几盘互有胜负，Oscar 胜场多些。陈斌老师借此机会采访两位组长。Mike 组组长王梦瑶表示，Mike 先手的几局都打成这样，可见没有什么希望了。而 Oscar 组组长吴宜谦平静地表示，Mike 组也具有很强的实力，现在还不能下结论，请大家继续关注比赛。第一盘 Oscar 以 26:14 获胜。随后的第二盘 Oscar 再度获胜，赢得了总冠军。Mike 赢得亚军。冠亚季军的奖品是 SESSDSA'17 限量纪念版超大鼠标垫。

吴宜谦代表 Oscar 组发表了感言。他简要介绍了算法的原理，与大多数组直接翻译人工制定的战略不同，他们组充分发挥机器的搜索能力，使用 AlphaBeta 剪枝搜索完成决策。（的确，Oscar 是唯一一个需要注意 30 秒时间限制的小组）此外他还放出豪言称这个游戏已被 Oscar 组开发得差不多了。然而据编者在编辑此书时对各队提交的最终算法运行的结果，即使给 Oscar60 秒的充足时间，前十名中还是有 6 个算法在 20 盘中战胜过 Oscar，3 个算法战胜 5 盘以上。可见这个游戏最佳策略的探寻还远远没有走到终点，编写出比 Oscar 更强的算法还是很有希望的。

【编者注】：本篇文字，第二节摘自 chbpku 公众号的文章，其余部分由编者结合自己的回忆和记录编写。第一节图片为视频截图，第二节图片摘自 chbpku 公众号，第三节图片为编者当时现场拍摄。

Part III

报告汇编



OSCAR

组长：吴宜谦

组员：祝奇文，曾羿东，纪晴，傅昊博，石晓霏，姚照原

小组赛战绩：N1

决赛战绩：冠军







OSCAR的照片墙

Oscar 小组
数据结构与算法课程
漂移乒乓实习作业报告

（吴宜谦*、祝奇文、纪晴、傅昊博、曾羿东、石晓霏、姚照原）

摘要：

我们组的算法使用机器计算模拟并结合了人工启发，主要运用深度递归搜索思想，在每一局中对己方和对方的跑位、迎球、击球和使用道具情况进行模拟，并根据生命值之差对模拟结果进行比较，并执行其中最优的策略。我们使用了搜索树进行搜索，并且使用了 Alpha-Beta 剪枝算法裁剪搜索树中没有意义的不需要搜索的树枝，以提高运算效率。同时我们也开发了 ds 历史数据，通过对历史记录统计分析，推测对手的行为，有针对的进行决策。在代码运行方面，我们组的算法取得了很好的成绩，在对战的过程中虽然耗时较长，但在所有组中成绩最优。

关键词：

深度递归搜索　搜索树　Alpha-Beta 剪枝算法　历史数据分析　tbcopy

1 算法思想

1.1总体思路

1. 算法的总体思路：模拟计算结合人工启发

首先根据己方和对方的球拍的位置为双方划分出合理的加速和跑位区间，然后对每一个回合中己方和对方的跑位、迎球、击球、收集道具、使用道具的情况进行模拟，通过生命值剩余的多少进行估值，找出本回合中己方的最优方案以及对方的最优方案。其中速度值区间的选取为人工启发，即在大量的对战中人工找到的最优解，即打角或者打道具。

2. 采用的主要数据结构与算法&采用的算法策略：

搜索树、Alpha-Beta 剪枝算法、ds 数据开发、人工启发

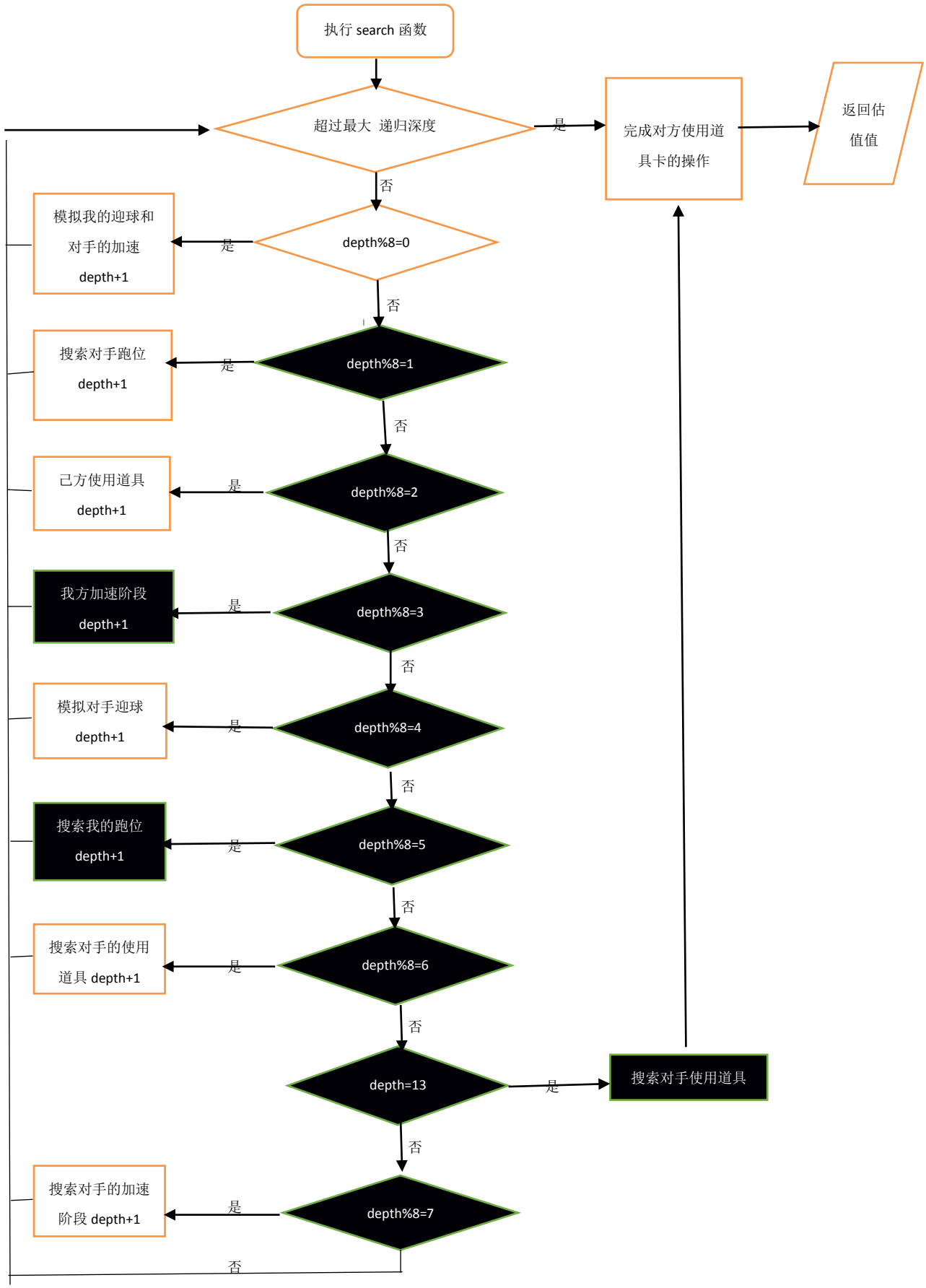
进行最大递归深度为 14 的搜索（一个半回合、8 个阶段）：第一阶段（depth%8==0）：模拟己方的迎球和对手的加速，不分枝；第二阶段（depth%8==1）：搜索对手跑位阶段；第三阶段（depth%8==2）：搜索己方的使用道具阶段，这里由于人工启发，选取了有道具就使用的策略，不存道具；第四阶段（depth%8==3）：搜索我的加速阶段；第五阶段（depth%8==4）：模拟对手的迎球不分枝；第六阶段（depth%8==5）：搜索我的跑位阶段；第七阶段（depth%8==6）：搜索对手使用道具阶段，同样采取有道具就用的策略；第八阶段（depth%8==7）：搜索对手的加速阶段。为了节约运算时间，对搜索树进行 Alpha-Beta 剪枝，即设定参数 alpha、beta，并设 alpha、beta 分别为己方的最大值估值和对方的最小值估值，在每一层递归操作中都会通过比较获利情况来相应的改变 alpha、beta 的数值，如果出现了最大值小于最小值的情况，则将这一枝剪去，不再对这一分枝进行更深层的递归；同时在每一层搜索中根据情况相应的返回估值值和操作值的列表。

在模拟回合和预测对手时，我们加入了对 ds 数据的利用，计算存储对手的跑位、加速度值，并对这些数据进行统计分析，推测对方的加速和跑位策略，然后根据预测做出相应的调整。

人工启发在算法中也发挥了重要的作用。通过观察大量的对战，人工确定最优的参数和区间，更高效的解决决策中遇到的问题。

1.2算法流程图

下面是主函数（search 函数）的流程图


执行 search 函数
超过最大 递归深度
是
完成对方使用道具卡的操作
返回估值值
否
depth%8=0
是
模拟我的迎球和对手的加速 depth+1
否
depth%8=1
是
搜索对手跑位 depth+1
否
depth%8=2
是
己方使用道具 depth+1
否
depth%8=3
是
我方加速阶段 depth+1
否
depth%8=4
是
模拟对手迎球 depth+1
否
depth%8=5
是
搜索我的跑位 depth+1
否
depth%8=6
是
搜索对手的使用道具 depth+1
否
depth=13
是
搜索对手使用道具
否
depth%8=7
是
搜索对手的加速阶段 depth+1
否

1.3算法运行时间复杂度分析

算法运行时间的主要构成为两个部分：深度搜索和 tbcopy 函数

1. 深度搜索：

   Alpha-Beta 剪枝算法用于减少搜索的节点数目，其效率很大依赖于节点的排列，记枝条分叉数为 b， 递归深度为 d。在理想的排序下，算法复杂度为 O(b^(d/2)),相较于未经剪枝的搜索树，它可以使搜索的节点的数量减少一半，从而使在相同时间下的搜索深度增加一倍。在随机排序下，算法复杂度平均为 O（b^(d*3/4)）。 搜索的深度越深，这个算法也就越智能，但搜索深度每增加一层，算法的时间复杂度会以指数进行增长，由于有时间的限制，所以我们的搜索深度有限。

2. tbcopy 函数：

   tbcopy 函数的主要作用：在每一次搜索时进行一次 tbcopy，将传入的桌面参数拷贝，使在搜索中不改变桌面的参数，防止上层被影响。我们最开始使用的是 deep copy 函数，但该函数的耗时太长，所以我们有自己编写了时间复杂度为 O（n）的 tbcopy 函数来执行相同的功能。

2 程序代码说明

2.1数据结构说明

1. 算法中采用的基本数据结构：
   ① 线性数据结构
   ② 树型数据结构
2. 对基本数据结构的改进：
   对树型结构的改进：用 Alpha-Beta 算法对搜索树剪枝。
3. 所用到的类有：
   ① math
   ② Python 内置基本类（列表、元组等）
   ③ random
   ④ table

2.2函数说明

1. serve（op_side: str, ds: dict）->tuple 发球函数：

   只有在作为 west 方时才会发球，其中 ds 为函数可以利用的储存字典，函数需要返回的是球的 y 坐标，和 y 方向的速度。首先从 0,1 中随机选取一个数，如果是 0 就从下角发球碰撞两次打对面上角如果是 1 就从上角发球碰壁两次打对面下角。

```
def serve(op_side: str, ds: dict) -> tuple:
    if random.randint(-1, 1) == 0:
        return 0, 5000/3 - 1
    else:
        return 1000000,-5000/3 + 1
```

2. play(tb: TableData, ds: dict) 打球函数：

函数中 tb 为 TableData 类型的对象，ds 是函数可以利用的储存字典，该函数返回的是一个 RacketAction 对象。首先调用记录数据到 ds，然后筛除场上的隐身和瞬移道具，调用 search 函数搜索最佳策略，返回一个列表[估值、己方加速、己方跑位、己方要使用的道具]。

```
def play(tb:TableData, ds:dict) -> RacketAction:
    #print(tb.tick)
    counter(tb, ds)
    tb.side['active_card'] = None
    tb.cards['cards'] = list(filter(lambda x: not x in ['DS', 'TP'], tb.cards['cards']))
    searchvalue = search(tb, None, 0, -1000000000, 1000000000, ds)
    if not searchvalue[3]:
        a = None
    else:
        a = searchvalue[3].code
    ds['side_acceleration%d'%(ds['count'])] = int(searchvalue[1])
    return RacketAction(tb.tick, tb.ball['position'].y - tb.side['position'].y, searchvalue[1], searchvalue[2], None, a)
```

ds 记录自己每次的加速的值，用于计算对方的加速度。函数最后返回对自己动作的声明。

3. summarize 为对局后保存历史数据的函数：

本函数在对局结束后调用，用于双方分析和保存历史数据

```
def summarize(tick:int, winner:str, reason:str, west:RacketData, east:RacketData, ball:BallData, ds:dict):
    return
```

4. interval 族函数：

这些函数全部都是搜索枝条的选取函数，参量一般是一个 tb，即桌面的情况，但对于对手的 interval 函数，会带上 ds，即历史信息来预测对方的行为。interval 函数的返回值全部是一个列表，即搜索区间，里面包含了在树搜索中的搜索值。

① interval_myacc(tb: TableData) :

我方的加速区间选取函数，参数是 tb 即桌面信息，返回的是一个列表，里面所有的元素是一个元组，其中第一个元素为加速之后球的目标速度值。第二个元素为该目标速度上附带的道具信息，其中速度值的选取为人工启发，即在大量对战中人工找到的最优解，即打角或者打道具。

v1 到 v8 分别为 8 个符合两次碰撞的打角速度值

```
def interval_myacc(tb:TableData):
    interval=[]
    v1 =((DIM[3] - tb.ball['position'].y) / (-DIM[0] + DIM[1]) * 1000)+1
    v2 = ((2 * DIM[3] - tb.ball['position'].y) / (-DIM[0] + DIM[1]) * 1000)
    v3 = ((2 * DIM[3] - tb.ball['position'].y) / (-DIM[0] + DIM[1]) * 1000)+1
    v4 = ((3 * DIM[3] - tb.ball['position'].y) / (-DIM[0] + DIM[1]) * 1000)-1
    v5 = ((- tb.ball['position'].y) / (-DIM[0] + DIM[1]) * 1000)-1
    v6 = ((-DIM[3]- tb.ball['position'].y) / (-DIM[0] + DIM[1]) * 1000)
    v7 = ((-DIM[3] - tb.ball['position'].y) / (-DIM[0] + DIM[1]) * 1000)-1
    v8 = ((-2*DIM[3] - tb.ball['position'].y) / (-DIM[0] + DIM[1]) * 1000)+1
    interval.append((v1, None))
    interval.append((v5, None))
    interval.append((v2,None))
    interval.append((v6, None))
    interval.append((v3,None))
    interval.append((v4, None))
    interval.append((v7,None))
    interval.append((v8, None))
```

接下来是打道具的速度值的计算

```
if len(tb.cards['cards']) != 0:
    for card in tb.cards['cards']:
        if  card==CARD_SPIN or card==CARD_INCL or card==CARD_DECL  or card==CARD_AMPL:
            cardv = [None,None,None,None,None]
            cardv[0] = (card.pos.y - tb.ball['position'].y) / abs((card.pos.x - tb.ball['position'].x)) * 1000
            cardv[1] = (2 * DIM[3] - card.pos.y - tb.ball['position'].y) / abs((card.pos.x - tb.ball['position'].x)) * 1000
            cardv[2] = (2 * DIM[3] + card.pos.y - tb.ball['position'].y) / abs((card.pos.x - tb.ball['position'].x)) * 1000
            cardv[3] = (- card.pos.y - tb.ball['position'].y) / abs((card.pos.x - tb.ball['position'].x)) * 1000
            cardv[4] = (- 2 * DIM[3] + card.pos.y - tb.ball['position'].y) / abs((card.pos.x - tb.ball['position'].x)) * 1000
            for i in range(5):
                if (cardv[i] >= 0 and cardv[i] > v1 and cardv[i] < v4) \
                    or (cardv[i] < 0 and cardv[i] < v5 and cardv[i] > v8):#若这些打到道具的速度值能够满足一次或者两次碰撞，则加入interval
                    interval.append((cardv[i],card))
return interval
```

② interval_itsacc(tb:TableData,ds)：

对方加速区间的选取函数，参量有两个，一个是 tb，一个是 ds，即桌面信息和历史数据，ds 的作用是试图推测对方的会如何加速，返回值同上面的 interval_myacc。速度值的选取依然遵从人工启发原则，即打角速度和打道具速度，此处返回的速度也都是目标速度，而非速度的变化量。

interval 用来存速度值目标，以元组为元素（速度值，目标（卡片名称或者 None））v1 到 v8 分别为 8 个符合两次碰撞的打角速度值

```
def interval_itsacc(tb:TableData,ds):
    interval = []
    v1 =((DIM[3] - tb.ball['position'].y) / (-DIM[0] + DIM[1]) * 1000)+1
    v2 = ((2 * DIM[3] - tb.ball['position'].y) / (-DIM[0] + DIM[1]) * 1000)
    v3 = ((2 * DIM[3] - tb.ball['position'].y) / (-DIM[0] + DIM[1]) * 1000)+1
    v4 = ((3 * DIM[3] - tb.ball['position'].y) / (-DIM[0] + DIM[1]) * 1000)-1
    v5 = ((- tb.ball['position'].y) / (-DIM[0] + DIM[1]) * 1000)-1
    v6 = ((-DIM[3]- tb.ball['position'].y) / (-DIM[0] + DIM[1]) * 1000)
    v7 = ((-DIM[3] - tb.ball['position'].y) / (-DIM[0] + DIM[1]) * 1000)-1
    v8 = ((-2*DIM[3] - tb.ball['position'].y) / (-DIM[0] + DIM[1]) * 1000)+1
```

若认为猜中了对方的加速而且对方是打对角原则，即远角原则，就只考虑那四个速度：本方在跑位前的位置在球桌上方，就添加对方往下打的四个加速度（由于模拟反角时，若不准确，造成的损失较大，所以删除了反角的猜测功能）；若本方跑位前在球桌下方，就添加对方往上打的四个加速度：

```
if guess_itsacc(tb, ds)[0] and guess_itsacc(tb, ds)[1] == 'dui':
    if ds['ball_position%d' % (ds['count'])] > DIM[3] / 2:
        interval.append((v2, None))
        interval.append((v3, None))
        interval.append((v5, None))
        interval.append((v8, None))

    else:
        interval.append((v1, None))
        interval.append((v4, None))
        interval.append((v6, None))
        interval.append((v7, None))
```

若无法判定对方如何加速，就添加对方的 8 个打角速度

```
else:
    interval.append((v1, None))
    interval.append((v5, None))
    interval.append((v2, None))
    interval.append((v6, None))
    interval.append((v3, None))
    interval.append((v4, None))
    interval.append((v7, None))
    interval.append((v8, None))
```

对方打道具的速度计算，与 interval_myacc 相同

```
if len(tb.cards['cards']) != 0:
    for card in tb.cards['cards']:
        if  card==CARD_SPIN or card==CARD_INCL or card==CARD_DECL  or card==CARD_AMPL:
            cardv = [None,None,None,None,None]
            cardv[0] = (card.pos.y - tb.ball['position'].y) / abs((card.pos.x - tb.ball['position'].x)) * 1000
            cardv[1] = (2 * DIM[3] - card.pos.y - tb.ball['position'].y) / abs((card.pos.x - tb.ball['position'].x)) * 1000
            cardv[2] = (2 * DIM[3] + card.pos.y - tb.ball['position'].y) / abs((card.pos.x - tb.ball['position'].x)) * 1000
            cardv[3] = (- card.pos.y - tb.ball['position'].y) / abs((card.pos.x - tb.ball['position'].x)) * 1000
            cardv[4] = (- 2 * DIM[3] + card.pos.y - tb.ball['position'].y) / abs((card.pos.x - tb.ball['position'].x)) * 1000
            for i in range(5):
                if (cardv[i] >= 0 and cardv[i] > v1 and cardv[i] < v4) \
                    or (cardv[i] < 0 and cardv[i] < v5 and cardv[i] > v8):
                    interval.append((cardv[i],card))
return interval
```

③ interval_myrun(tb:TableData):

己方的跑位值的选取函数，参量是 tb，返回值是一个列表，其中的元素全部是我的位置的变化量，即跑位值，跑位点的选取有人工启发和跑位范围内的平均取值。

自己在上面的情况，跑位遵从不跑过中点的原则。y2 是跑位值的最小值，此处选择跑到中点时的跑位值。第二个 if 是防止在之后的迎球中最大半径无法覆盖全局的死亡修正，若处在了一个对手通过打远角就可以一击必杀的位置，那么就要不能采用这种位置，即跑位区间要修改。若之后可以覆盖到下角，则 y1 就是跑到上顶端的跑位值，y1 为跑位值的最大值；若之后无法覆盖到下角，则 y1 就选择刚好能覆盖到下角的值。此处的 651300 为人工启发点，根据大量对战推测出的一个比较好的跑位点，若到这个点能够覆盖到最下角的话，就添加到 interval 中。自己在下方的情况，与上面的是对称的:

```
def interval_myrun(tb:TableData):
    interval = []
    if tb.side['position'].y > DIM[3] / 2:
        y2 = DIM[3] / 2 - tb.side['position'].y
        if int((tb.side['life'] / RACKET_LIFE) * (-DIM[0] + DIM[1]))>=DIM[3]:
            y1=DIM[3]-tb.side['position'].y
        else:
            y1=18*(tb.side['life']-700)-tb.side['position'].y
        if (651300-tb.side['position'].y)<y1:
            interval.append(651300-tb.side['position'].y)
    else:
        y1 = DIM[3] / 2 - tb.side['position'].y
        if int((tb.side['life'] / RACKET_LIFE) * (-DIM[0] + DIM[1]))>=DIM[3]:
            y2=-tb.side['position'].y
        else:
            y2=DIM[3]-tb.side['position'].y-18*(tb.side['life']-700)
        if (348711-tb.side['position'].y)>y2:
            interval.append(348711-tb.side['position'].y)
```

最后再从 y2 到 y1 取等四分，加入到 interval 中：

```
    Delta_y=(y1-y2)/4
    for i in range(5):
        interval.append(y2+i*Delta_y)
    return interval
```

④ interval_itsrun(tb:TableData):

对手非首次跑位的取值函数，非首次跑位意味着是在模拟中的跑位，没有跑位方向这个确定量，而且模拟中偏差会较大，就没有加入 ds 的预测。

与前一个函数的区别是：对手的跑位粗糙化了，只取了三等分。

```
def interval_itsrun(tb:TableData):
    if tb.op_side['position'].y>DIM[3]/2:
        if int((tb.op_side['life'] / RACKET_LIFE) * (-DIM[0] + DIM[1]))>=DIM[3]-tb.op_side['position'].y:
            y1=DIM[3]-tb.op_side['position'].y
        else:
            y1 = int(1.8 * (tb.op_side['life'] - 650) - tb.op_side['position'].y)
        y2=DIM[3]/2-tb.op_side['position'].y
    else:
        y1=DIM[3]/2-tb.op_side['position'].y
        if int((tb.op_side['life'] / RACKET_LIFE) * (-DIM[0] + DIM[1])) >=DIM[3]:
            y2 = -tb.op_side['position'].y
        else:
            y2 = int(DIM[3] - tb.op_side['position'].y - 1.8 * (tb.op_side['life'] - 650))
    interval = []
    Delta_y = (y1 - y2) /3
    for i in range(4):
        interval.append(y2 + i * Delta_y)
    return interval
```

⑤ interval_firstitsrun(tb:TableData,ds):

对手的首次跑位模拟，即有确定的跑位方向的第一次跑位，此时要调用 ds 来预测对手的跑位值，同样遵从不跑过中点的原则。

其中 guess 为猜测出来的结果，guess[0]表示对猜测有信心：

```
def interval_firstitsrun(tb:TableData,ds):
    interval = []
    guess=guess_itsrun(tb,ds)
    if guess[0]:
        if guess[1]=='r':
            if tb.op_side['position'].y>DIM[3]/2:
                interval.append(DIM[3]/2+guess[2]-tb.op_side['position'].y)
            else:
                interval.append(DIM[3]/2-guess[2]-tb.op_side['position'].y)
```

若认为没有猜中，则进行同样的大范围预测，y1 与 y2 的选择与之前相同。并针对跑位方向，更精确的确定跑位范围，依然是取三等分；最后也考虑了对手使用隐身术的情况：

```
else:
    if tb.op_side['position'].y>DIM[3]/2:
        if int((tb.op_side['life'] / RACKET_LIFE) * (-DIM[0] + DIM[1]))>=DIM[3]-tb.op_side['position'].y:
            y1=DIM[3]-tb.op_side['position'].y
        else:
            y1=int((tb.op_side['life'] / RACKET_LIFE) * (-DIM[0] + DIM[1]))
        y2=DIM[3]/2-tb.op_side['position'].y
    else:
        y1=DIM[3]/2-tb.op_side['position'].y
        if int((tb.op_side['life'] / RACKET_LIFE) * (-DIM[0] + DIM[1])) >= tb.op_side['position'].y:
            y2 = -tb.op_side['position'].y
        else:
            y2 = -int((tb.op_side['life'] / RACKET_LIFE) * (-DIM[0] + DIM[1]))
    if tb.op_side['run_vector']==1:
        Delta_y = (y1) / 3
        for i in range(4):
            interval.append((3-i)*Delta_y)
    elif tb.op_side['run_vector']==-1:
        Delta_y = (y2) / 3
        for i in range(4):
            interval.append((3-i) * Delta_y)
    else:
        Delta_y = (y1-y2) / 3
        for i in range(4):
            interval.append(y2+i * Delta_y)
return interval
```

5. image 族函数：

模拟桌面情况变化的函数，一般来说参量有两个，第一个是 tb，即桌面情况，第二个是 move，即递归枝条中的行动值，比如加速加多少，跑位跑多少。函数返回值全部是为改变之后的 tb。

① image_mybat(tb:TableData):

模拟我的球拍迎球阶段的桌面情况变化，参量为 tb，没有 move 值，或者说行动值唯一确定，即球的落点减去当前球拍位置的值。

最后的判断为接不到球的死亡修正，若在迎球阶段，接不到球，则一定不能采取这种策略，将生命值调味负无穷，在接下来的 evaluate 中得到一个负无穷的值：

```
def image_mybat(tb:TableData):
    life = tb.side['life']
    dy = abs(tb.ball['position'].y - tb.side['position'].y)
    param=2 if tb.op_side['active_card'][1] == CARD_AMPL else 1
    tb.side['life'] -= (dy ** 2 // FACTOR_DISTANCE ** 2)*param
    tb.side['position'].y = tb.ball['position'].y
    if tb.op_side['life'] > 0 and int((life / RACKET_LIFE) * (-DIM[0] + DIM[1])) < dy:
        tb.side['life'] = -100000000
    return tb
```

② image_itsbat(tb:TableData):

模拟对手的迎球阶段的函数，与 image_mybat 类似。

```
def image_itsbat(tb:TableData):
    life = tb.op_side['life']
    dy = abs(tb.ball['position'].y - tb.op_side['position'].y)
    tb.op_side['life'] -= (dy ** 2 // FACTOR_DISTANCE ** 2) * (CARD_AMPL_PARAM if tb.side['active_card'] == CARD_AMPL else 1)
    tb.op_side['position'].y = tb.ball['position'].y
    if tb.side['life'] > 0 and int((life / RACKET_LIFE) * (-DIM[0] + DIM[1])) < dy:
        tb.op_side['life'] = -100000000
    return  tb
```

③ image_myacc(tb:TableData,aim_v):

模拟我方的加速过程的函数，其中也包含了加速之后球的飞行的过程中桌面情况的变化，move 值为 aim_v，即想要加速到的球的速度。

进行了旋转球和变压器修正。其中的的 Delta_v 为 play 函数声明的加速度值，aim_v 不随着旋转球改变，但是 Delta_v 会随着对手使用旋转球而*2。之后改变桌面道具信息和加速后球飞完了之后的 y 坐标：

```
def image_myacc(tb:TableData,aim_v):
    param=1
    if tb.op_side['active_card'][1]==CARD_SPIN:
        param=2
    Delta_v=(aim_v[0]-tb.ball['velocity'].y)*param
    param_=2 if tb.op_side['active_card'][1] == CARD_AMPL else 1
    tb.side['life'] -= (Delta_v ** 2 // FACTOR_SPEED ** 2) *param_
    tb.ball['velocity'].y +=int(Delta_v*(CARD_SPIN_PARAM if tb.op_side['active_card'][1] == CARD_SPIN else 1))
    if aim_v[1] != None:
        tb.cards['cards'].pop(tb.cards['cards'].index(aim_v[1]))
        tb.side['cards'].append(aim_v[1])
    Y = tb.ball['velocity'].y / 1000 * (-DIM[0] + DIM[1]) + tb.ball['position'].y
    count = Y // DIM[3]
    tb.ball['position'].y = (Y - count * DIM[3]) * (1 - 2 * (count % 2)) + DIM[3] * (count % 2)
    tb.ball['velocity'].y = tb.ball['velocity'].y * ((count + 1) % 2 * 2 - 1)
    return tb
```

④ image_itsacc(tb:TableData,aim_v):

对手的加速过程模拟,与 image_myacc 类似：

```
def image_itsacc(tb:TableData,aim_v):
    param=1
    if tb.side['active_card']==CARD_SPIN:
        param=2
    Delta_v = (aim_v[0]-tb.ball['velocity'].y)*param
    tb.op_side['life'] -= (Delta_v**2//FACTOR_SPEED ** 2) *(CARD_AMPL_PARAM if tb.op_side['active_card'] == CARD_AMPL else 1)
    tb.ball['velocity'].y +=int(Delta_v*(CARD_SPIN_PARAM if tb.side['active_card'] == CARD_SPIN else 1))
    if aim_v[1] != None:
        tb.cards['cards'].pop(tb.cards['cards'].index(aim_v[1]))
        tb.op_side['cards'].append(aim_v[1])
    Y = tb.ball['velocity'].y / 1000 * (-DIM[0] + DIM[1]) + tb.ball['position'].y
    count = Y // DIM[3]
    tb.ball['position'].y = (Y - count * DIM[3]) * (1 - 2 * (count % 2)) + DIM[3] * (count % 2)
    tb.ball['velocity'].y = tb.ball['velocity'].y * ((count + 1) % 2 * 2 - 1)
    return tb
```

⑤ image_myrun(tb:TableData,Delta_y)& image_itsrun(tb:TableData,Delta_y):

跑位过程的模拟，move 值为真实的跑位值 Delta_y，如果碰到瞬移卡，则从距离减去 CARD_TLPT_PARAM 再计算体力值减少：

```
def image_myrun(tb:TableData,Delta_y):
    if tb.side['active_card'] == CARD_TLPT:
        if abs(Delta_y)-CARD_TLPT_PARAM> 0:
            tb.side['life'] -=(Delta_y-CARD_TLPT_PARAM) ** 2 // FACTOR_DISTANCE **2
    else:
        tb.side['life']-=Delta_y ** 2 // FACTOR_DISTANCE ** 2
    tb.side['position'].y+=Delta_y
    return tb
def image_itsrun(tb:TableData,Delta_y):
    if tb.op_side['active_card'][1] == CARD_TLPT:
        if abs(Delta_y)-CARD_TLPT_PARAM> 0:
            tb.op_side['life'] -=(Delta_y-CARD_TLPT_PARAM) ** 2 // FACTOR_DISTANCE **2
    else:
        tb.op_side['life']-=Delta_y ** 2 // FACTOR_DISTANCE ** 2
    tb.op_side['position'].y+=Delta_y
    return tb
```

⑥ image_mycard(tb:TableData,used_card):

我的首次使用道具阶段的模拟，该过程声明道具卡后，所有道具卡的效果可以完全真实模拟出来，move 值为使用的卡 used_card。若获取加血减血卡立即生效：

```
def image_mycard(tb:TableData,used_card):
    if used_card !=None:
        tb.side['active_card'] = used_card.code
        tb.side['cards'].pop(tb.side['cards'].index(used_card))
    else:
        tb.side['active_card']=None
    if tb.side['active_card'] == CARD_INCL:
        tb.side['life'] += CARD_INCL_PARAM
    elif tb.side['active_card'] == CARD_DECL:
        tb.op_side['life'] -= CARD_DECL_PARAM
    return tb
```

⑦ image_nextmycard(tb:TableData,used_card,ds):

我方第二次使用道具过程的模拟，由于此二次使用道具通常是第一次加速之后接到了一个道具再使用，因此由于搜索深度不够，在模拟中，像旋转球这样的道具来不及发挥作用。于是就有人为的估值模拟，此处其他的与 image_mycard 类似，但是多添加了旋转球修正：减扣对手一定的体力值，通过 guess_spin_to_itsacc 来实现，不过由于实际效果，对手加速值默认为 250。

```
def image_nextmycard(tb:TableData,used_card,ds):
    if used_card !=None:
        tb.side['active_card'] = used_card.code
        tb.side['cards'].pop(tb.side['cards'].index(used_card))
    else:
        tb.side['active_card']=None
    if tb.side['active_card'] == CARD_INCL:
        tb.side['life'] += CARD_INCL_PARAM
    elif tb.side['active_card'] == CARD_DECL:
        tb.op_side['life'] -= CARD_DECL_PARAM
    elif tb.side['active_card'] == CARD_SPIN:
        tb.op_side['life']-=((2*guess_spin_to_itsacc(tb,ds)[1])** 2 // FACTOR_SPEED ** 2)
    return tb
```

对方使用道具的模拟过程与此类似，这里不再陈述。

6. guess 函数：

猜测对手的行为的函数，参量有两个，第一个为 tb，桌面信息，第二个为 ds，即历史数据，返回值通常是一个猜测的值

① guess_spin_to_itsacc(tb:TableData,ds):

猜测我方使用旋转球之后对手的反应的函数，由于效果不是很好，模拟之后的风险比较大，所以并没有采用，而是直接默认为无法猜测；Param_guess=250，为通过对方最少在相持中损耗的体力值推测出来的加速度，为虚假值，只是一个为了模拟对方的体力值减少而计算出来的值：

```
def guess_spin_to_itsacc(tb:TableData,ds):
    itsacc = (False, Param_guess)
    return itsacc
```

② guess_itsacc(tb:TableData, ds):

猜测对手的加速的值的函数，一般来说，为通过历史数据来判断对手是打对角还是反角（远角和近角）；返回一个对手的策略，(True，'dui')，True 代表对猜测有信息，'dui'代表对手会选择打到对角：

```
def guess_itsacc(tb:TableData, ds):
    result = (False, None)
    if ds['count'] >= 2:
        pos1 = ds['ball_position%d'%(ds['count']-1)]
        pos2 = ds['ball_position%d'%(ds['count'])]
        if abs(pos1 - pos2) >= 970000:
            result = (True, 'dui')
        elif abs(pos1 - pos2) <= 30000:
            result = (True, 'fan')

    return result
```

③ guess_itsrun(tb:TableData,ds):

猜测对手的跑位值的函数，第一种为判断对手是否会对称跑距离中点的半径，第二种为对手是否跑到同一个地方，第三种为对手是否假跑位，第二第三种效果不好，就删除了：

```
def guess_itsrun(tb:TableData,ds):
    result = (False, None)
    if ds['count'] >= 4:
        runposition1 = ds['opside_runposition%d'%(ds['count']-2)]
        runposition2 = ds['opside_runposition%d'%(ds['count']-1)]
        if runposition1 and runposition2:
            if abs(abs(runposition1 - DIM[3]/2) - abs(runposition2 - DIM[3])) <= 30000:
                result = (True, 'r', 0.5*(abs(runposition1 - DIM[3]/2)+abs(runposition2 - DIM[3])))
    return result
```

7. evaluate(tb:TableData):

到达搜索深度之后的估值函数，此处选取的是我方体力值和对方体力值之差，越大对我越有利，越小对对方越有利

```
def evaluate(tb:TableData):
    return tb.side['life']-tb.op_side['life']
```

8. search(parametertb:TableData,move,depth,a,b,ds):

主体函数 search，为一个递归函数，参量有 6 个 parametertb 为桌面信息，为一个 TableData 类，move 为上一层告诉下一层应该执行的操作，depth 为递归深度。a,b 为阿尔法贝塔剪枝中的两个参量，分别表示我的操作带来的最大估值值和对手的操作带来的最小估值值，ds 为历史数据。

若超过了最大深度,此处 dept 为 14，最后还是有一个对方使用道具卡的操作，则完成对方使用道具卡的操作、返回估值值：

```
def search(parametertb:TableData,move,depth,a,b,ds):
    tb=tbcopy(parametertb)
    if(depth>maxdepth)
        tb = image_itscard(tb, move)
        return evaluate(tb)
```

若为到达最大深度：

第一阶段，模拟我的迎球和对手的加速，不分枝

```
    else:
        if depth%8==0:
            if depth!=0:
                tb = image_itsacc(tb, move)
            tb = image_mybat(tb)
            a=search(tb,None,depth+1,a,b,ds)
            return a
```

第二阶段，搜索对手跑位阶段。depth=1 时：要一同返回搜索中得到的我方的最佳操作，返回带有估值值和操作值的列表；若 depth!=1，则只需返回估值值，其他操作与上面相同：

```
        if depth%8==1:
            if depth==1:
                myacc = 0
                myrun = 0
                mycard=None
                for i in interval_firstitsrun(tb,ds):
                        searchvalue=search(tb,i, depth + 1,a,b,ds)
                        if b > searchvalue[0]:

                            b = searchvalue[0]
                            myacc = searchvalue[1]
                            myrun = searchvalue[2]
                            mycard = searchvalue[3]
                return [b, myacc, myrun, mycard]
            else:
                for i in interval_itsrun(tb):
                    if a < b:
                        searchvalue = search(tb,i, depth + 1,a,b,ds)
                        if b > searchvalue:
                            b = searchvalue
                return b
```

第三阶段，搜索我使用道具阶段，由于人工启发，此处为有道具就用的策略，不存道具，depth=2 时：返回我方的操作值，返回带有估值值和操作值的列表；depth 不为 2 时，无需保存操作值：

```
elif depth%8==2:
    tb = image_itsrun(tb, move)
    if depth==2:
        i=None
        if tb.side['cards']:
            i=tb.side['cards'][0]
        searchvalue=search(tb,i,depth+1,a,b,ds)
        a=searchvalue[0]
        myacc=searchvalue[1]
        myrun=searchvalue[2]
        mycard=i
        return [a, myacc, myrun,mycard]
    else:
        i = None
        if tb.side['cards']:
            i = tb.side['cards'][0]
        searchvalue = search(tb, i, depth + 1,a,b,ds)
        a=searchvalue
        return a
```

第四阶段，搜索我的加速阶段，depth=3 时：模拟我使用道具之后的桌面情况变化，并对于 interval_myacc 中的所有的速度值一一搜索，若 a<b 则搜索进行下去，最后返回带有估值值和操作值的列表；若发生在更深层，则无需保存操作值，只需返回估值值：

```
elif depth%8==3:
    if depth == 3:
        tb = image_mycard(tb, move)
        param = 1
        if tb.op_side['active_card'][1] == CARD_SPIN:
            param = 2
        myacc = 0
        myrun = 0
        for i in interval_myacc(tb):
            if a<b:
                searchvalue=search(tb, i, depth + 1,a,b,ds)
                if a < searchvalue[0]:
                    a = searchvalue[0]
                    myacc = (i[0] -tb.ball['velocity'].y)*param
                    myrun =searchvalue[1]
        return [a, myacc, myrun]
    else:
        tb = image_nextmycard(tb, move,ds)
        for i in interval_myacc(tb):
            if a<b:
                searchvalue = search(tb, i, depth + 1,a,b,ds)
                if a <searchvalue :
                    a =searchvalue
        return a
```

第五阶段，模拟对手的迎球，不分枝搜索

```
elif depth%8==4:
    tb = image_myacc(tb, move)
    tb = image_itsbat(tb)
    b=search(tb,None, depth + 1,a,b,ds)
    return b
```

第六阶段，搜索我的跑位阶段，首先判断本步是否要返回操作，若 depth=5,则为第一个回合中的加速，要返回，并且进行分枝，最后返回带有估值值和操作值的列表；若 depth!=5,无需返回操作值：

```
elif depth==5:
    if depth == 5:
        myrun=0
        for i in interval_myrun(tb):
            if a<b:
                searchvalue = search(tb, i, depth + 1,a,b,ds)
                if a < searchvalue:
                    a = searchvalue
                    myrun = i
        return [a,myrun]
    else:
        for i in interval_myrun(tb):
            if a < b:
                searchvalue = search(tb,i, depth + 1,a,b,ds)
                if a < searchvalue:
                    a =searchvalue
        return a
```

第七阶段，搜索对手使用道具阶段，处理方法也是猜测对手有道具就用，操作与 depth%8==1 相同，且此处无需返回操作值：

```
elif depth==6:
    tb = image_myrun(tb, move)
    i=None
    if tb.op_side['cards']:
        i=tb.op_side['cards'][0]
    searchvalue = search(tb,i, depth + 1,a,b,ds)
    b=searchvalue
    return b
```

搜索对手使用道具阶段，这里调换了一下顺序，为了多搜索一下对手在第二回合中的道具使用情况，避开第二回合中我的跑位阶段搜索（那个阶段的搜索无意义，因为我的跑位的价值要体现在下一回合的迎球中，所以这种深度下，并无意义）：

```
elif depth==13:
    i = None
    if tb.op_side['cards']:
        i = tb.op_side['cards'][0]
    searchvalue = search(tb, i, depth + 1, a, b,ds)
    b = searchvalue
    return b
```

第八阶段，搜索对手的加速阶段，模拟对手用道具带来的 tb 的变化，并且对对手的每一个加速值进行搜索，最后返回估值参量。

```
else:
    tb=image_itscard(tb,move)
    for i in interval_itsacc(tb,ds):
        if a < b:
            searchvalue = search(tb, i, depth + 1,a,b,ds
            if b > searchvalue:
                b =searchvalue
    return b
```

9. tbcopy 函数：

我们为了避免使用 deepcopy，自己实现一个针对 TableData 的 copy 函数。

```
def tbcopy(tb):
    newtick = tb.tick
    newstep = tb.step
    newside = {
        'name': tb.side['name'],
        'position': Position(tb.side['position'].x, tb.side['position'].y),
        'life': tb.side['life'],
        'cards': [i for i in tb.side['cards']],
        'active_card' : tb.side['active_card']
    }
    newopside = {
        'name': tb.op_side['name'],
        'position': Position(tb.op_side['position'].x, tb.op_side['position'].y),
        'life': tb.op_side['life'],
        'cards': [i for i in tb.op_side['cards']],
        'active_card': tb.op_side['active_card'],
        'accelerate': tb.op_side['accelerate'],
        'run_vector': tb.op_side['run_vector']
    }
    newball = {
        'position': Position(tb.ball['position'].x, tb.ball['position'].y),
        'velocity': Vector(tb.ball['velocity'].x, tb.ball['velocity'].y)
    }
    newcard = {
        'card_tick': tb.cards['card_tick'],
        'cards': [i for i in tb.cards['cards']]
    }
    return TableData(newtick, newstep, newside, newopside, newball, newcard)
```

10. countor 函数：

    记录 tb 的数据，并作分析，最终在 ds 上记录。在开局时：要清空记忆、计数、记录球落点列表和对手跑位列表；每次计数时 count 加 1，并记录双方的信息，当计数大于等于 3 时，开始计算并记录对方的跑位：

```
def counter(tb, ds):

    if tb.tick in (0, 1800):
        ds.clear()
        ds['count'] = 0
        ds['ball_position_list'] = [0, ]*intervalnum
        ds['opside_runposition_list'] = [0, ]*intervalnum

    ds['count'] += 1
    ds['ball_position%d'%(ds['count'])] = tb.ball['position'].y
    ds['opside_position%d'%(ds['count'])] = tb.op_side['position'].y
    ds['ball_velocity%d'%(ds['count'])] = tb.ball['velocity'].y
    ds['opside_rundirection%d'%(ds['count'])] = tb.op_side['run_vector']
    ds['opside_life%d'%(ds['count'])] = tb.op_side['life']
    ds['opside_active_card%d'%(ds['count'])] = tb.op_side['active_card'][1]
    ds['side_active_card%d'%(ds['count'])] = tb.side['cards'][0].code if tb.side['cards'] else None
    record_ball_position(tb, ds)
    if ds['count'] >= 3:
        record_opside_runposition(tb, ds)
```

11. record_ball_position(tb, ds)& record_opside_runposition(tb, ds):

    分别用于记录对手的击球落点和对手上一次的跑位位置。

12. caculate_opside_acc(tb, ds)& caculate_opside_runposition(tb, ds):

    分别用于计算对手真实加速度和真实跑位位置。

2.3程序限制

Oscar 代码设计思路缜密，对漂移乒乓也有了很深入的开发，总体上没有受限，但在一

些方面也存在一定的局限：

因为对战时是不完全决策，所以我们对对手行为的预测会出现偏差。我们采用的方式是想象对手会做出最优的决策，并据此找到我们的策略。但事实上，对手不一定会按照预想选择最优解，在这种情况下，绝大多数时候 0scar 会处于优势，但如果碰到了针对 0scar 的代码，就会出现问题： 0scar 认为对手不会刻意留下死角，并在这个前提之下做出决策，如果对手刻意放弃一个死角，0scar 难以抓住这个机会消耗对手，并且可能会被对手拖住、损耗生命值。这个问题归根结底还是信息不完全所造成的，0scar 无法做到准确的猜测跑位。虽然理论上根据现有的信息不可能完全算出对手的跑位，但 0scar 在一方面还可以使算法更优化。

另外一个比较细节的问题是关于旋转球的使用，0scar 在模拟先加速接旋转球再用的时候可能会出问题。

3 实验结果

3.1测试数据

实验环境说明：

- 硬件配置：cpu 2.20ghz，8g
- 操作系统：window10、ios
- Python 版本：Python 3.5

代码测试：

我们组最开始开发出来了 3 套主算法和 3 个版本的陪练代码，主要算法分别是：T_ex (之后改造成 T_exer，是最终 Oscar 代码的原型)、T_Hades (优化的打角并跑位算法)、T_Oscar (也是基于打角跑位的思想)；陪练代码有 T_idiot1、T_idiot2、T_idiot3，我们在老师提供的陪练算法的基础上，将其进行了更智能的升级，让它可以更好的起到陪练的效果。

在开发了一段时间之后，我们在三套算法之间进行了相互测试，并在对局过程中相互发现问题、改正 bug。最后选取了更完善，更智能的 T_ex, 并对其进行了修正，在热身赛前补充了对 ds 的应用，作为最终的 Oscar 代码。但是 Oscar 代码耗时太长，容易超时，所以最后将耗时很多的 deepcopy 函数改写为了 tbcopy 函数，大大缩减了一个回合所需的时间。

在热身赛中，我们组的提交了 4 份代码，都取得了不错的成绩，其中一个在所有组中积分最高。在热身赛之后，我们有和其他一些组的算法进行了比赛，对自己的代码也做出了调整。在小组赛中，我们的代码虽然耗时最长，但几乎以全胜的比分晋级了四强，并且在最后夺冠，达到了预期的效果。

3.2结果分析

Oscar 算法和其他组算法一大区别在于，Oscar 选取的策略不是人工想出的优势策略，它的迎球、跑位、击球、接道具和使用道具是通过大量的机器计算来决定的，而人工主要起到的是启发的作用，并且 Oscar 也具有对历史数据进行学习的能力。正因为这样， Oscar 的决策几乎没有漏洞，在面对一些很聪明的代码时，也不会受到影响。我们在测试、热身赛和实

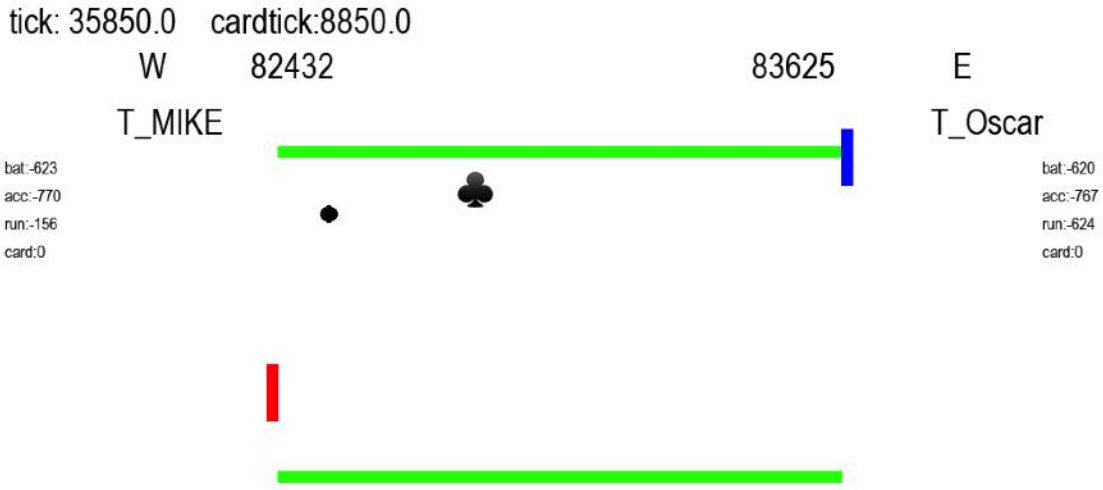
战对弈过程中都取得了最好的成绩，并且在最后取得了冠军。代码在最开始运行时，时间很长，我们对其进行了剪枝，并对 deepcopy 函数进行了替换，在降低了时间复杂度后，我们又适当的细化了区间、加深了递归深度，最后的运行时间还是比较理想的，时间主要消耗在 search 函数进行搜索时。

3.3经典战局

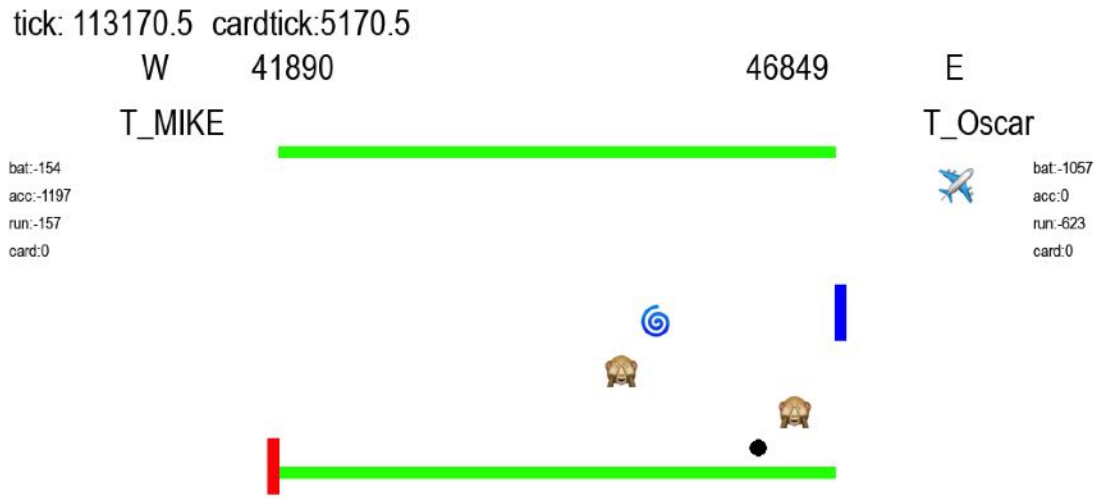
Oscar 算法在实习过程中的表现基本与预期一致，算法的功能也很强大，所以在比赛现场我们并未对结果感到过多的惊奇。不过在与 Golf 组和 Mike 组进行对战时，有很精彩的对局。下面是 Oscar 与 Mike 对战时的一些精彩对战图：

（1）[W.M]T_MIKE-VS-T_Oscar-010：

在这一局中 Mike 为发球方，占有一定的优势，但 Oscar 在开局后不久就通过使用掉血包和利用对方跑位的漏洞领先对手。

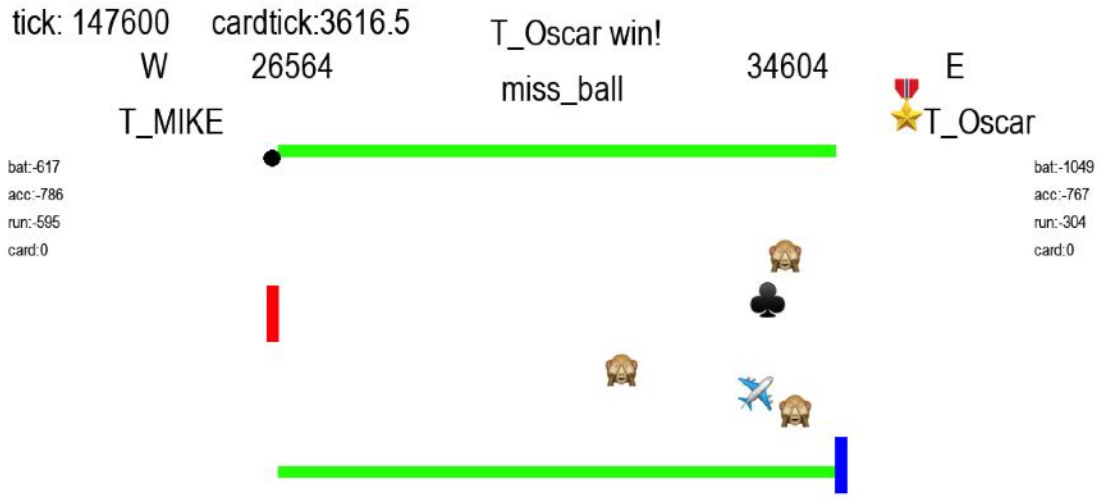


在对战的过程中 Oscar 和 Mike 都多次使用道具，但 Oscar 始终保持领先，并逐渐拉大了和对手之间的差距。



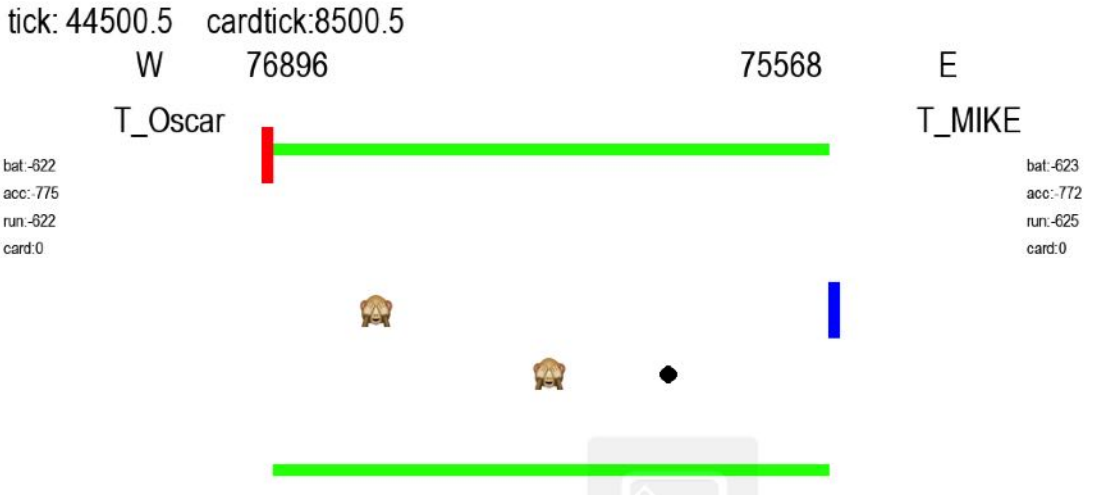
在对战的最后阶段，Oscar 在下边和 Mike 进行拉锯，消耗了对方的体力，并在最后击打

反角，这时 Mike 由于生命值所限没有接到球，Oscar 获胜。

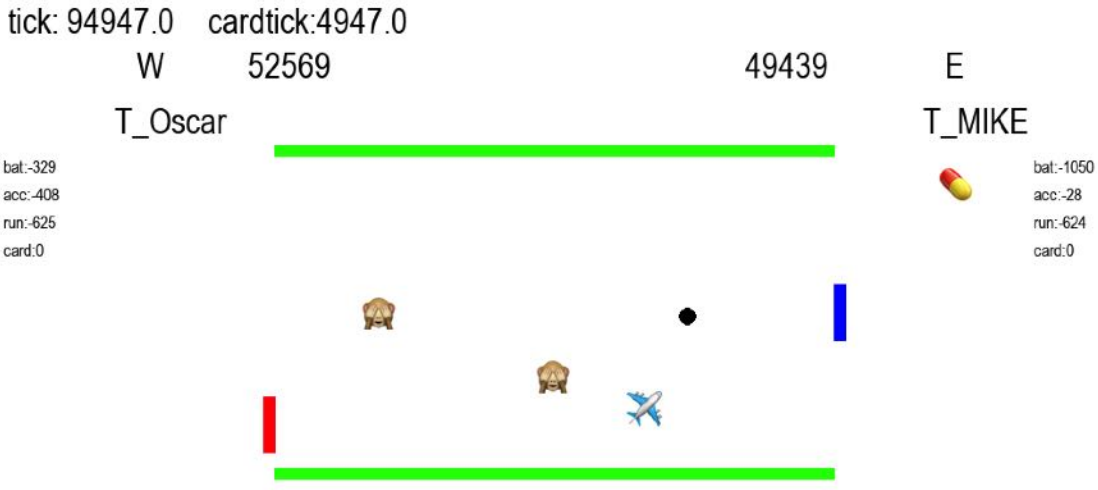


（2）[E.M]T_MIKE-VS-T_Oscar：

这一局对战中，双方的生命值始终相差不多。在对局的前半部分中，双方的接球和跑位都比较谨慎，没有留出破绽，道具的使用也比较相当。



对局的后半部分十分紧张，双方胶着不下，Oscar 和 Mike 对道具的处理方式相似，都是只要得到就直接使用，Oscar 先使用瞬移术拉开了差距，但随后 Mike 又利用补血包赶了上来。

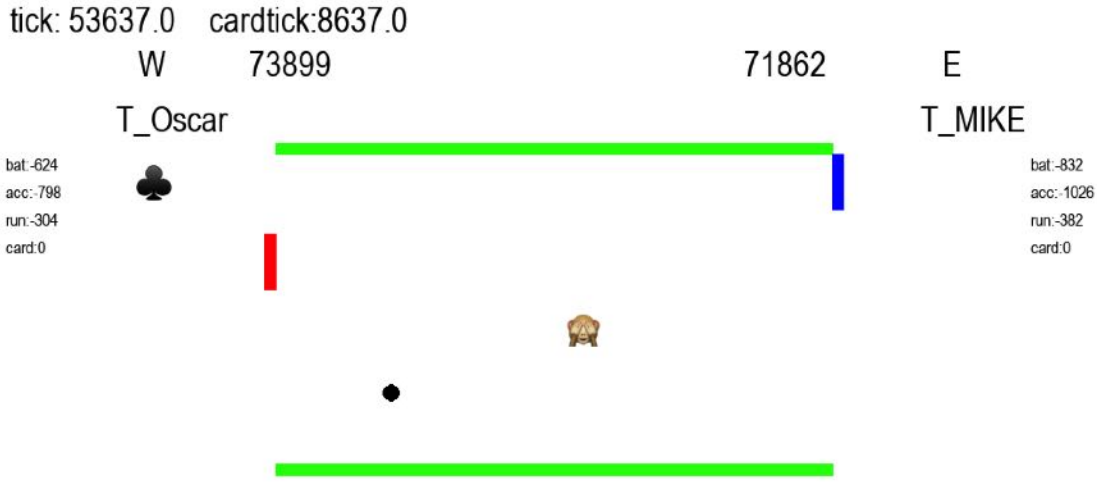


在对局的最后，Mike 希望绝杀 Oscar，但 Oscar 成功的跑位，接到了对方的球，并利用打反角，在处于微弱劣势的情况下战胜了 Mike。

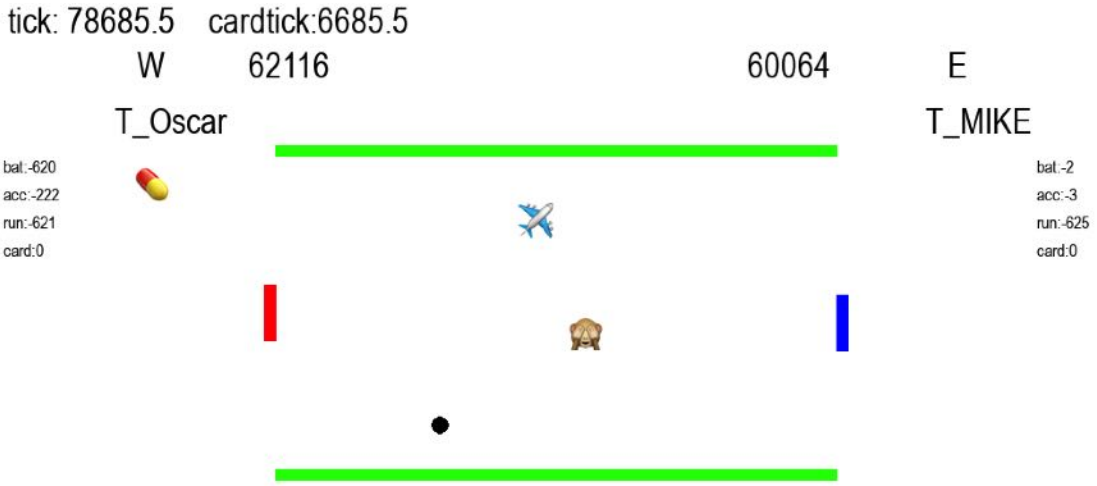


（3）[E.M]T_Oscar-VS-T_MIKE

在这一局中，Oscar 很好的使用了球桌上的道具，并通过道具拉开与对手的差距。。



期间 Oscar 比较集中的使用了变压器、隐身术、补血包和瞬移术等道具，并最终赢得了比赛



4 实习过程总结

4.1分工与合作

本组成员：吴宜谦*，祝奇文，纪晴，傅昊博，曾羿东，石晓霏，姚照原
合作交流方式：

1. 就近在 28 楼 454 寝室进行；
2. 随时在微信群中进行沟通讨论，上传程序进行协同操作；
3. 小组会议集中讨论分析。

分工：

1. 开发算法和编程任务：祝奇文、纪晴、傅昊博、曾羿东和吴宜谦同学负责主要基础代码的撰写，石晓霏和姚照原主要负责写报告并对于算法思想提出一些建议。同时吴宜谦同学参与全局、控制整个实习作业的进度。

2. 小组报告：石晓霏同学算法思想阐述大体结构框架构建和的撰写，吴宜谦和祝奇文完善最终代码的注释，姚照原同学负责总结回顾展望部分。

4.1.1 小组初建

五月十一日，数算课程上说到期末大作业——“漂移乒乓”可以开始准备组队。当天中午，小组快速成立，并且采用传统方法，同时结合自愿原则选出了组长（如图）。最终愉快地决定脸最白的吴宜谦做组长。一周后，根据大部分组员的在线讨论，选出了最终的组名和头像:滑稽照耀下的 Oscar，小组就此正式成立。



组长选举



小组头像

4.1.2 第一次组会（无图）

时间：2017/05/19,pm 3:30　地点：四教 509　与会人员：全体组员　记录人员：姚照原

- 第一次组会上确立了以 454 寝室全体（祝奇文、纪晴、傅昊博、曾羿东）为中坚力量强攻编写代码、石晓霏和姚照原拟写报告外加辅助、组长吴宜谦总揽全局，同时每一环节尽量互帮互助的大致方针；
- 组长吴宜谦在组会上对整个大作业的基本情况做了简要说明；
- 祝奇文为算法进程划定了一个思路：先开发基础算法，然后进行算法升级——利用 ds 文件设计优化的算法，实现一定的智能化。

4.1.3 第二次组会

时间：2017/05/25,pm 6:00　地点：四教 505　与会人员：全体组员　记录人员：姚照原



● 第二次组会上，大家带来了第一期编写的代码，分别完成了 Oscar 的基础算法和“T_ex”算法，升级了作战平台中附带的 idiot 文件；

● 吴宜谦讲解了他主持开发的“T_ex”一代算法；454 完成了 Oscar 的基础算法，并开始 ds 文件利用的的构思，同时升级了作战平台中附带的“T_idiot” 文件；

● 当天组会结束后，代码编写组集合在一起整合两部分算法，最终决定继续升级陪练算法“T_Hades”系列，同时勠力同心优化“T_ex”算法。

4.1.4 第三次组会

时间：2017/06/03,pm 4:00　地点：四教 503　与会人员：全体组员　记录人员：石晓霏



● 利用热身赛的复盘数据，思考改进方法；

● 使用了 Alpha-Beta 剪枝算法简化算法**[1]**，完成算法的重要部分 tbcopy 函数**[2]**，最大搜索深度设为 13 层；

● “T_ex”与陪练算法“T_Hades4”进行现场较量，发现陪练算法胜率超出预期；

● 对“T_ex”开始现场 debug，完善正式代码。

4.1.5 比赛前夜

时间：2017/06/08,am 0:10

地点：28 楼 454

与会人员：代码编写组

记录人员：姚照原

● 调试代码，努力排除可能出现的 bug。

● 提交最终代码。



☝决赛前夜还在紧张地检查代码的祝奇文同学（主力代码编写者之一）

☛绝望但决不放弃（吴宜谦同学未出镜）



4.2经验与教训

4.2.1 组长感想：

很荣幸能够担任 Oscar 小组的组长，感谢每位成员的努力，他们都很优秀。我觉得，作为组长，最重要的任务应该就是根据每个成员的特点分配工作任务，并组织讨论。在我看来，我们组有几个地方做的很好，比如说积极参加两次热身赛，分别派出四个代码和最终代码出战，也认真分析复盘数据。再比如说大家在负担各自任务的同时也能对于整个过程提出自己的想法，集齐七个人的智慧才战胜了全部的对手杀向冠军。从刚开始的初级版白痴代码，到最终的代码，从刚开始的运行几十分钟到后来的 6-8 秒，甚至在比赛前夜，我们都不停地进行代码的优化。这中间的努力，可能只有我们自己知道了。Oscar 慢吞吞但是极其出色的表现赢得了全场赞叹，但是我们没有骄傲和停止前进，比赛前的晚上也不曾放弃、尽最大的努力让 Oscar 更优秀。所以，最终我们能拿到冠军，也不是一件惊奇的事，因为我们付出的努力是我们能做到的最好。另一方面，报告也比较顺利，代码组写完代码以后详细地讲解要点，在组员们的共同努力下呈现出来。

当然，我们组也有做的不完美的地方。比如说，组会前后没有成功的约过饭。再比如说因为对战时是不完全决策，所以我们对对手行为的预测会出现偏差。我们采用的方式是想象对手会做出最优的决策，并据此找到我们的策略。在这种情况下，绝大多数时候 0scar 会处于优势。但事实上，对手不一定会按照我们的预想选择最优解，如果碰到了针对 0scar 的代码，就会出现问题： 0scar 认为对手不会刻意留下死角，并在这个前提之下做出决策，如果对手刻意放弃一个死角，0scar 难以抓住这个机会消耗对手，并且可能会被对手拖住、损耗生命值。这个问题归根结底还是信息不完全所造成的，0scar 无法做到准确的猜测跑位。虽然理论上根据现有的信息不可能完全算出对手的跑位，但 0scar 在一方面仍旧可以达到更优。另外一个比较细节的问题是关于旋转球的使用，0scar 在模拟先加速接旋转球再用的时候可能会出问题。一次期末大作业，让我们学到了很多课本之外的知识，我们自己的能力也得到了极大地锻炼，而大作业的成功离不开我们每个人的努力。再次感谢我的组员们和所有为了大作业而尽心尽力的老师、助教和同学，谢谢你们。

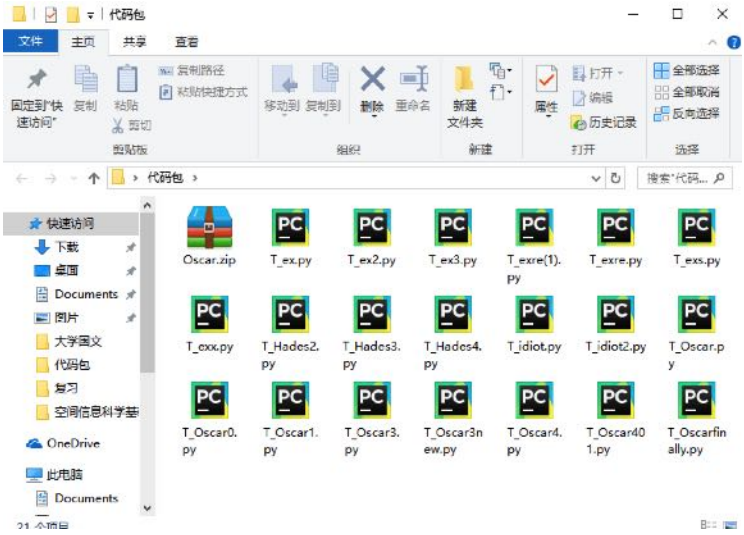
4.2.2 组员感想：

很高兴能够作为 Oscar 组的一员和组内其他同学一起合作完成这次大作业。对于大作业我最大的印象就是团结：编写代码的时候大家一起挑错、代码中比较复杂的部分互相讲解，报告组支持代码组工作、代码组也帮助报告组修改报告，并没有“事不关己高高挂起”的心态，积极参加热身赛和私下约战，比赛前修改代码到深夜、面对层出不穷的 bug 不曾放弃，我想正是这样的团结和争先精神让 Oscar 成为最终的影帝。能有优秀的队友，见证并全程参与这次大作业，我感到很荣幸。

——石晓霏

这次的大作业的编写代码可以说是面临挑战不断。虽然我们常常抱着滑稽抱枕说 Oscar 有来自东方的神秘力量，但是我们知道这力量是我们合起来的力量，Oscar 的每一个成员都是不可或缺的。编写代码的时候常常会遇到各种各样的 bug，也会遇到 Oscar 不听话乱打球的情况，这些都需要我们一一纠正。对手实力强劲，我们必须要不断的改进算法，遇到了问题会一起讨论很久，实在解决不了的困难就会求助于老师和网络，还要利用热身赛数据让 Oscar 变聪明一点。我们做了很多努力，但最后耗时的问题还是解决的不够理想。最终慢吞吞的 Oscar 夺下冠军，也算是长舒一口气。

——祝奇文



图为 Oscar 研发过程中的各个版本

4.3建议与设想

4.3.1 一点建议

期末大作业的时间说长不长、说短不短，但是我们在自己能力范围之内已经可以说是做到很好了。这次大作业对很多同学都是一种新的小组合作体验，能够亲手做出来一个完整的代码的过程是幸福的。不过 pingpong 是新提出的项目，虽然完善了很多，但还是存在一些问题，比如说版本更新非常多，能否提前开始对战平台的编写？还有能否加入更多的对战回

放？这样就不会出现像这一次大家都在等待 north 区而不知道干什么的情况了。代码也有很大的提升空间，可以在后台开启擂台赛互相切磋，以娱乐和在 GitHub 上互相完善为主。

4.3.2 寄语：

数算课为我们打开了一扇新世界的大门，除过各种精巧的数据结构类型，还有从零做起的大作业，可以学到很多知识，包括课本外的合作、自信。对于这门课程的下一届学生，我想说团队合作是最重要的，不管能力如何，总是要贡献自己的力量，没有人是一无是处的，也没有谁是可以免除工作的。小组分工也不是规定了工作内容，小组内互帮互助、拧成一股绳是很重要的。另外一定要有自信，相信自己和队友能够搞出不逊色于其他人的算法，不要太在意对战输和赢——要做最好的自己，记得收获到学习和进步的乐趣。祝你们也能够在这门课上有所收获。

5 致谢

本次作业在组长的带领下，全员参与，利用老师和助教讲解的知识和网络查找的资料完成。这次大作业让我们学到了很多，主要是对本学期学习的数据结构知识进行了综合利用，并用 python 语言编写了更加适用于本算法的 tbcopy 函数，对本课知识的理解更进一步。

首先要感谢陈斌老师和助教的漂移乒乓创意、对战平台编写成员的工作，让我们有机会自己从零起步实现一个完整算法编写过程。同时他们的随时答疑和全程关注对于本组最终算法的产生与提高有着至关重要的作用，在此深表感谢。除此之外还要感谢 Oscar 全体成员的共同努力，祝奇文、纪晴、傅昊博、曾羿东和吴宜谦同学负责主要基础代码的撰写，石晓霏和姚照原主要负责写报告并对于算法思想提出一些建议，吴宜谦同学参与全局、控制整个实习作业的进度，每个人都从这次作业实习中学到了很多东西。也要感谢和 Oscar 私下约架的其他小组成员，感谢和你们的每一场切磋给 Oscar 带来的进步。希望以后还会有更多的这样的活动。

再次感谢陈斌老师和助教的指导和全体组员的戮力同心。

6 参考文献

[1]《一看就懂的 Alpha-Beta 剪枝算法详解》

http://blog.csdn.net/baixiaozhe/article/details/51872495

[2]《Python 函数中，参数是传值，还是传引用？》

http://mp.weixin.qq.com/s/DOBG0yF0y8CwodCky_LALA



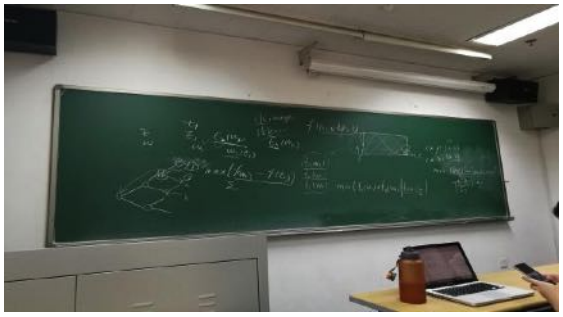
MIKE

组长：王梦瑶

组员：任庆杰，朱宇森，方睿，王冠力，金山木，唐金潼

小组赛战绩：E1

决赛战绩：亚军

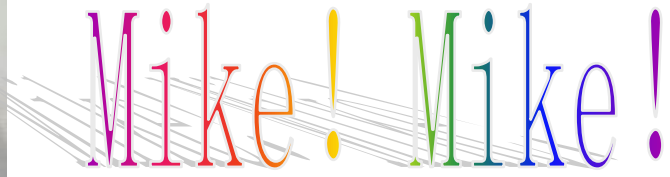











Mike! Mike!


OMG

数据结构与算法课程实习作业报告

（王梦瑶* 任庆杰 朱宇森 方睿 王冠力 金山木 唐金潼）

摘要：

参赛算法将回球分为两种模式：打角模式和捡道具模式。在这两种模式下，都可能产生多种路线方案；对每条路线，我们都计算相应的跑位和道具使用策略，并用估值函数计算我方与对方的消耗差值，最终选择差值最小的方案。同时，我们还加入了一些学习功能，根据对手的历史数据调整跑位和修正估值。参赛算法的时间复杂度控制在O(1)，并在实际运用中取得了很好的效果。另简要介绍备战过程中研究开发的遗传算法和神经网络，待后续完善。

关键字：

乒乓对战 固定算法 评估函数 统计预判 遗传算法 神经网络

1 算法思想

1.1 总体思路

一、算法总体思路

算法将回球分为两种模式：打角模式和捡道具模式。顾名思义，打角模式中球会控制打角，尽力消耗对手体力，捡道具模式则控制球捡道具。在这两种模式下，都可能产生多种路线方案。每条路线下，我们都计算对应的跑位和道具使用策略，并用估值函数计算它们的消耗，选择消耗最小的方案。估值函数和两种回球模式都是根据复盘数据归纳总结出来的。

二、采用的主要数据结构与算法

本次实习作业，我们没有采用一些较为复杂的数据结构和算法，仅使用了列表、字典等常规数据类型。

三、算法策略

1、安全速度策略：

由于规则明确指出：发球和每次回球时，在墙壁上碰撞反弹的次数为 1~2 次，且如果恰好打到角上，则算作碰壁，故对于每一个点，都有一个确保不违规的“安全速度”范围。

如下图所示，设场地长为 L，宽为 H，红色的球距 x 轴高度为 h，水平运动时间为定值 t。则在纵向上，向上打的最小速度为 $V_{upmin}=\frac{H-h}{t}$，最大速度为 $V_{upmax}=\frac{3H-h}{t}$；向下打的最小速度为 $V_{downmin}=\frac{h}{t}$，最大速度为 $V_{downmin}=\frac{2H+h}{t}$。同时，对于整张桌面，可以取到一个安全速度范围 $(\frac{H}{t},\frac{2H}{t})$，在这个速度范围内，则可确保在任何点上不发生 0 次或 2 次以上反弹。在预测对方行为时，也是采用安全速度来估计：对方接到球后，必须采取相应地加速度，来确保触球后球的速度在安全速度范围之内（否则对方击球后将以 invalid bounce 告负）。

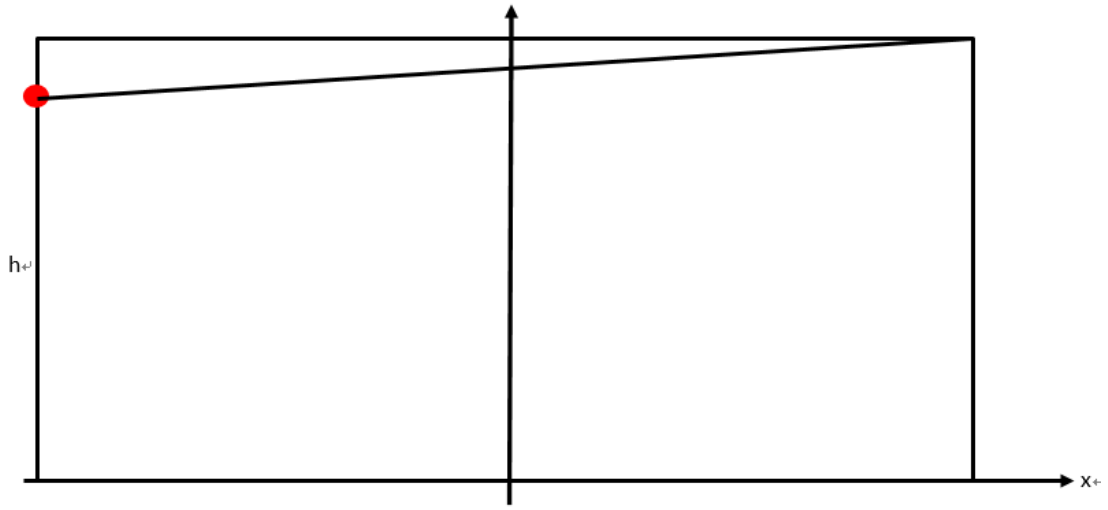


图 1.1 桌面态势图

我方初始策略是尽量将球速控制在我方安全速度之内和对方安全速度之外，企图以此逼迫对方每次接球都必须给球加速使其回到安全速度范围内，而加速可以迅速的消耗对方的体力，属于进攻型策略。

2、min（我方消耗－对方消耗）策略：

设 cost=我方消耗-对方消耗+道具消耗，其中我方消耗=我方加速花费+我方跑位花费+我方下次击球花费，对方消耗=对方击球花费+对方加速花费，道具消耗=捡道具花费+使用道具花费（可正可负）。同时注意到我们在计算道具消耗的时候引入了机会成本的概念，即我们取道具意味着要放弃原来的最优路线，而那也应该算作取道具的成本之一。我方采用搜索算法，对处于安全速度范围内的加速度进行 cost 的计算，选择最小的 cost 作为最优加速度并进行击球。cost 估值函数是本算法的核心部分，此后将不断根据策略和参数的调整而进行修改，最终达到捡最有用的道具和打对方远角的效果。

3、跑位策略

由于跑位和迎球的消耗是二次函数，因此在已知乒乓球落点的情况下，跑位到总路程的二分之一是最优策略。但由于初始设计的时候并不能确定对方球的落点，因此采取了跑中间的跑位策略。

在第一次热身赛后，经过测试和复盘数据的观察，发现各组基本上都采用的打角模式，因此球的落点基本上锁定在两角。这就区分为两种情况：如果球的下一次落点和此次落点是不同的角，那么跑位到中间是最优策略；如果球的下一次落点和此次落点是同样的角，那么有微小的跑位甚至不跑位是最优策略。因此我们转换了盲中策略，而转向两角策略，即如果对方打另一个角，则跑位到中间，如果对方依然打这个角，那么移动很小的距离，给一个跑位方向以迷惑对手。两角策略相比盲中策略在实战上有了很大进步，但是同时又产生了另外的问题：如果对方捡道具，则落点就不再是两角。

通过观察优秀算法的复盘数据，我们猜测并实现了四分之一跑位策略，即设

置一个分段函数：如果上一次击球点在两角，则跑位到两角最近的四分之一点；如果上一次击球点在中间，那么跑位到距离击球点最近的四分位点。四分之一跑位策略的应用使得算法的性能有了很大的改进。

但随着我们开始应用历史数据预判对方行为，我们的跑位策略也逐渐向智能方向改进，开始依据预判得到的对方落点来进行跑位。至于如何预判对方的落点，详情请参考统计预判策略。

我们的跑位策略最后添加了两个细节：当我方的体力值已经不足以覆盖四分之三桌面宽度的时候，一律往中间跑位，避免被对方打到死角；当我方的体力值已经不足以覆盖二分之一桌面宽度的时候，一律往预测角的四分位点跑位，因为跑到中间会导致任何球都接不到。

4、打角策略

我方初始采用的打角策略来自于评估函数，即根据各种参数和最小化（我方消耗－对方消耗）来计算出“最优”的落点。但是实际上，这样造成的效果是我们并没有贯彻暴力打角。但是在实战中，我们发现，暴力打角的算法有其独有的优势。因此，我们改进了 evaluation 函数，使其能够尽量的打离对方的跑位更远的角，即所谓的打远角策略。至于如何预判对方的跑位，详情请参考统计预判策略。

5、道具取舍策略

由于不同道具达成的效果不同，每种道具在实际应用中也必然有取舍。每个道具设置两个参数——取用价值和留存价值。

道具的取用价值视为取得道具的收益，每次出现道具时，比较道具的取用收益和为取道具改变路线所付出的成本，只有收益大于成本，才做出改变路线去道具的决定，否则不取道具。道具的留存价值指道具放在包内的价值，可以视作本轮使用道具的成本，每次使用道具时，比较道具用于本轮的收益和成本，只有道具的使用给对方造成的消耗在本轮的收益大于包内价值，才做出使用道具的决定，否则将道具留在包内。

无论是取用道具还是使用道具，都针对不同的道具比较成本和收益，最终选择净收益最大的道具。

6、统计预判策略

本组对于历史数据的应用，主要体现在通过统计来预判对方的行为方面。考虑对于跑位和打角而言，如何才能够做到我方消耗最小和对方消耗最大？打角的目的是通过预判对方的跑位来使得对方消耗最大，即总是打对方的远角；跑位的目的是通过预判对方的落点来使得我方消耗最小，即总是跑位到我方触球点与落点的中间位置。

首先思考如何预判对方的跑位。我们先换位思考一下，对方是如何决定自己的跑位？是根据我方的落点，即对方的击球点。我们将我方的落点分为三类：远角、近角和中间，然后统计对方历史上在此分类下跑位的数据，由于考虑到代码的复杂性，我们仅仅依赖于最近一次在这个分类下对方的跑位，即每打一回合，

出现新的跑位数据后，就直接更新这个分类下的数据，然后在根据上一回合我方的落点，调用我们的历史存储信息，判断对方这一回合的跑位。

然后思考如何预判对方的落点。对方的击球决定落点，对方的击球给予什么给出？我方的跑位。因此，我们假定对方根据我方跑位进行落点的选择。由于实战数据显示，除了取道具外，其他落点基本集中来两角。根据跑位的位置，落点有远角和近角之分，因此我们根据我方的三种跑位情况（三个四分位点）进行分类，统计对方远角和近角的频率。采用与预判跑位相同的分类统计思路，根据这一回合的跑位调用历史数据，判断对方下一回合的落点。

虽然统计预判策略只是更新最新一轮的数据，并没有对全部的历史数据进行追踪和分析，但是在实战结果上已经有了显著的优势。

7、斩杀策略

在双方每回合触球后球速均在安全速度范围内的情况下，我方设置了斩杀函数。一旦对方生命低于一个阀值，跑位距离受到限制，出现死角，我方即采取斩杀策略：直接进攻死角，使对方下个回合因迎球失败（miss ball）而告负。

1.2 算法流程图

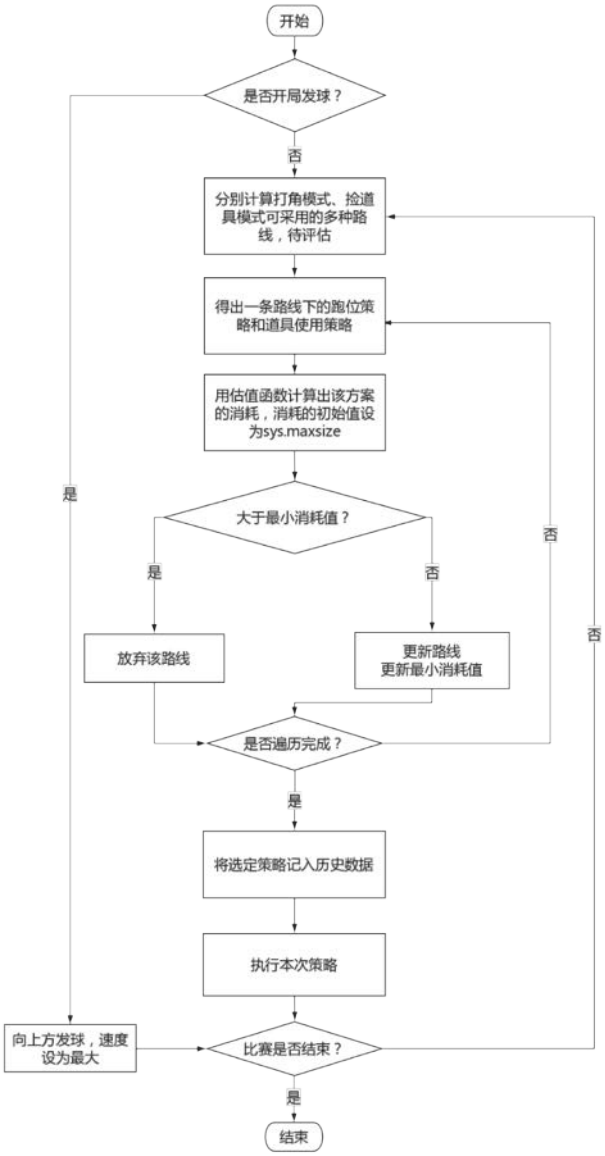


图 1.2 算法流程总图

1.3 算法运行时间复杂度分析

该算法运行时间复杂度为 O(1)，整个算法以判断为主，循环结构很少，客观条件决定路线数量有限（如在打角时只考虑打上角或打下角，桌面上道具数量不超过 10 个等），且未使用较复杂的数据结构，故最终整个算法复杂度可控制在 O(1)。

2 程序代码说明

2.1 数据结构说明

一、主要数据结构

本算法使用的数据结构以线性数据结构为主，未在数据结构上作出较大扩展和改进，本算法的优势主要是在策略上。

二、自定义类

1、Route（路线）类：包括起始位置、球的起始纵向速度、落点、对应的action、自定义的回球模式 mode、路线的反弹次数和经过的道具，并判断路线是否合法。

2、RunMode（跑位模式）类：包括模式参数 traits，以及参数的添加和判断相等的方法。

3、Rally（回合）类：包括该回合中我方采取的路线，以及双方该回合的基本信息。

以上自定义类都写在 play 函数下。

2.2 函数说明

1、vel_critical(pos)：临界速度计算函数，计算出各个位置（pos）向上击打、向下击打两种情况下恰好反弹 1 次、2 次、3 次的速度值，取最邻近的合法速度，并以嵌套列表的形式返回这些速度值。

2、isLegal(pos, vel)：根据 vel_critical 函数算出的临界速度值判断在位置 pos 处以速度 vel 击出的球是否合法，若合法，则返回 True，反之返回 False。

3、is_atcorner(pos)：判断一个位置 pos 是否在角落里，若距上下边界的长度小于球桌宽度的 1/4 即视为在角落里。

4、get_op_pos(pos, vel)：返回在位置 pos 处以速度 vel 击出的球到达对方的落点。

5、get_op_vel(pos, vel)：返回在位置 pos 处以速度 vel 击出的球到达对方的速度。

6、serve(op_side: str, ds: dict) -> tuple:：发球函数，按向上速度最大模式发球，并初始化历史数据，返回元组，其中第一项没有实际用处，第二项 vel_critical(0)[0][0]不符合原有设想，但由于修改代码的重点在 play 函数上，故遗漏了这里的错误。

7、play(tb: TableData, ds: dict) -> RacketAction：打球函数，最重要的函数之一。此处

（1）initialize_run_menu()：根据击球方式，对跑位模式进行初始化，分为向上击球、向下击球和捡道具，无返回值；

（2）match(traits, test_route)：跑位模式匹配函数，根据击球模式 traits 和待评估的回球路线 test_route，经过匹配，返回对应经初始化的跑位模式。

（3）test(action)：测试运行结果，返回 bounces, cards_hit，在自定义类 Route 中作为反弹和道具的初始值。

（4）hit_corners()：获取正常打角模式下的路线。

（5）hit_cards()：获取捡道具的路线。

（6）check(route)：防止我方生命值刚刚小于 LIFE_LIMIT 时被斩杀。该函数在 get_run_strategy(route)中被调用，以生成更好的跑位策略。

（7）get_run_strategy(route)：获取跑位策略。

在分区赛、八强赛中，当我方生命值低于设定阈值时，采取中间跑位；在此后的比赛中，增设了第二生命阈值（第一、第二阈值分别对应迎球时最大移动距离为桌面宽度的 3/4 和 1/2 情况下的生命值）。初步得到跑位策略后，需要调用 check 函数，模拟击球和执行跑位后我方剩余的生命值；若该生命值位于两阈值之间，则修改跑位到恰好不会被斩杀的位置。

初步的跑位策略如下：

若选择捡道具，则优先按模式匹配函数进行匹配（向远角方向以最小消耗跑位）；

若我方击球位置太靠近中间，采取中间跑位；

若选择打角，则还要引用对对方加速消耗的统计数据。若统计结果指示对方倾向于以加速消耗为 0 的模式回球，则在起始位置位于球桌上半部分时，跑位至球桌总宽度 3/4 处；在起始位置位于球桌下半部分时，跑位至球桌总宽度 1/4 处。若统计结果指示对方倾向于以加速消耗大于 700 的模式回球，则在起始位置位于球桌上半部分时，跑位至球的当前位置与球桌下端的中点；在起始位置位于球桌下半部分时，跑位至球的当前位置与球桌上端的中点。

其他情况：采取中间跑位的保守策略。

（8）get_card_strategy(route)：获取道具使用策略。考察我方拥有的道具；若为补血包、掉血包、隐身术，则直接使用；若为旋转球、瞬移术、变压器，则在道具使用可以保证一定收益时暂定使用，若道具箱已满，则直接使用。

（9）evaluate(route)：估值函数，本算法的核心函数。将花费（cost）的初始值设为 0，按照以下步骤计算，最后返回 cost 值：

I.加我方加速花费；

II.加我方跑位花费；

III.根据对方上轮跑位减去对方击球和跑位花费；

IV.在只考虑安全速度的情况下减去对方加速花费；

V.按近似量化的方式减去获取道具的等效花费，我方生命值较低时，提高补

血包的获取价值，即减少花费；获取道具的路线的落点靠近中间时，降低道具的获取价值（容易得不偿失），即增加花费；

VI.减去使用道具花费。在此步中，使用不同的道具都有初始的花费，而根据实际情况，花费还会有所调整。其中不使用道具的策略也减少一定花费；用历史记录中对手的动态加速花费来评估旋转球的使用效果；我方生命值较低时，提高补血包的使用价值；对方生命值较低时，提高掉血包的使用价值；瞬移术的使用价值按当前路线的跑位策略评估；对方生命值较低时，提高变压器的使用价值。

VII.结合对手的最大移动距离，若能够斩杀对方，则将该路线的评估结果（即cost）设为 - sys.maxsize。

（9）update_memory()：单局开局时更新历史数据。

（10）memorize(route)：单局结束时更新历史数据。

2.3 程序限制

由于我们的策略通过人观测和计算得出，没有采用机器学习，故虽然面对固定战术时能显示出较强威力，但在面对更为灵活的机器学习算法时还是会显露出不完善的地方。

3 实验结果

3.1 测试数据

一、实验环境说明

本组测试工作分散在各个组员电脑上进行，方睿使用的是自己所在实验室的电脑。

组员 / 项目	王梦瑶	朱宇森	方睿（实验室）	金山木	王冠力	唐金潼
CPU	2.7 GHz Intel Core i5	Intel(R) Core(TM) i7-4710MQ CPU @ 2.50 GHz	Intel®Core™ i7-4930K CPU@ 3.40GHz	2.7 GHz Intel Core i7	Intel(R) Core(TM) i5-4200M CPU @ 2.50 GHz	Intel®Core™i5-4200H CPU@ 2.80 GHz 2.79 GHz

内存	8GB 1867 MHz DDR3	4.00 GB	32 GB	4GB 1333MHz DDR3	4.00 GB	4 GB
操作系统	OS X El Capitan 10.11.5	Windows 7 旗舰版	Windows 10 专业版	MacOS Sierra 10.12.5	Windows 7 旗舰版	Windows 10 家庭中文版
PYTHON 版本	3.5.2	3.6.2	3.6.1	3.6	3.6.0	3.4.4

二、测试方法

1、参与热身赛进行实战测试；

2、编写简单陪练算法 T_RAW，将其作为对手；

3、将改进后算法与改进前算法对战。

测试结果是，我们将算法不断改进，使算法实力大大提高。

T_RAW 的策略是根据 1.1 中 min（我方消耗－对方消耗）策略，分类讨论速度范围，暴力搜索所有合法的解，并选取最优解，且不考虑道具，可以应对对方使用旋转球，该算法未经后期细化处理，较为粗糙。

三、热身赛结果

1、第一次热身赛胜负数据（化名 FANCY）：

总共对战 101 局，胜 73 场。

胜局中对方失败原因：{'invalid_bounce': 27, 'miss_ball': 46}

负局中我方失败原因：{'invalid_bounce': 4, 'miss_ball': 24}

2、第二次热身赛胜负数据（化名 SWIFT）：

总共对战 170 局，胜 98 场。

胜局中对方失败原因：{'invalid_bounce': 42, 'miss_ball': 56}

负局中我方失败原因：{'miss_ball': 72}

3.2 结果分析

由于我们根据可视化复盘分析了对手的跑位方式、击球方式、各项消耗，在热身赛、实战中，我们发现某些算法之间可能存在相克的情况，故针对不同的战术风格，我们都编写了代码，在代码中最大程度上体现了兼容性。这样的思想让我们能够较自如地应对不同风格的对手。

3.3 经典战局

一、T_FANCY vs T_AUGST

第一次热身赛经典战局。我们从与 T_AUGST 的对局中观察出其算法的跑位策略：若有打角的趋势，则只向另一侧跑位一小段距离。其余情况向中间跑位。

二、T_FANCY vs T_FMQED

第一次热身赛经典战局。我们从与 T_FMQED 的对局中观察出其算法的跑位策略：按四分之一分段进行跑位，将所在边划为四段，根据某种方法计算出对方击球趋势，选择最近的四分点进行跑位。

我们在后续的算法中将上述两种跑位策略进行了测试，并采用了四分之一分段跑位法。

三、T_Mike vs T_Kilo

四组私下约战的经典战局。我们发现，尽管优化了跑位方案，但仍然负局居多。经分析发现，我方在体力下降至一定值后，面对对方打远角的回球，极易 miss_ball，因此我们修改了跑位策略，即上文中提到的阈值，达到该阈值后跑向对应位置（详见算法策略 3）。我们也从对方打远角的策略中吸取经验，在我们的算法中增加了回球打远角的策略。另外，在与 Oscar 组的对局中，由于我们对其跑位策略无法做出判断，故猜想对方在跑位上可能使用了学习算法（当然，最后经证实，我们的想法是错误的）。

四、T_SWIFT vs T_excited

第二次热身赛经典战局。四组私下约战是在第二次热身赛前一天，故由于时间关系，我们发现自己弱点后未及时作修改，所以仍然在打角算法前占明显下风。但本次热身赛的结果进一步确定了我们对跑位策略的优化方向。

4 实习过程总结

4.1 分工与合作

1、小组分工

算法组：全组集体讨论算法，提出打角策略和道具取舍策略，朱宇森提出安全速度策略、评估函数和斩杀策略，王梦瑶和金山木提出跑位策略，王梦瑶提出统计预判策略，方睿提出遗传算法和神经网络算法。

代码组：代码主要由朱宇森和王冠力负责完成，任庆杰负责辅助工作。

测试组：测试主要由王梦瑶、金山木和唐金潼完成。

报告组：报告由唐金潼主要负责撰写，同时全组进行修改。

2、合作与交流方式

本小组采取微信群交流与线下组会交流结合的方式，在微信群中交流实时进度，在组会中共同完善策略，工作效率较高。

3、历次组会记录

（1）第一次组会（5 月 15 日）（如图 4.1.1）

梳理游戏规则，根据“最大化对方击球消耗”原则确定打角策略。

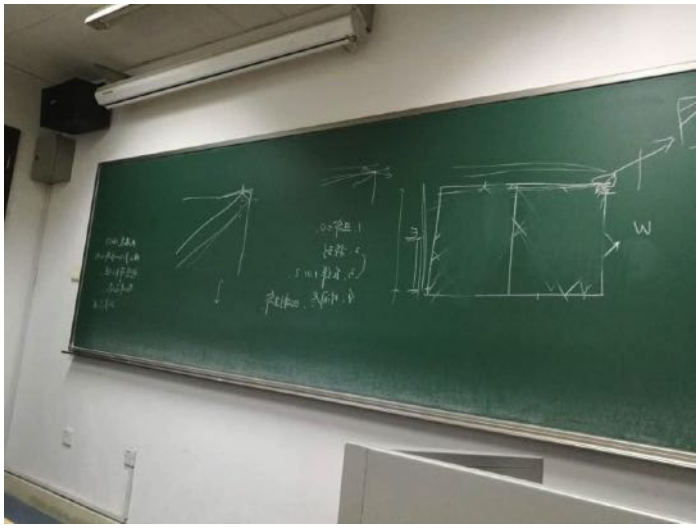


图 4.1.1 第一次组会

（2）第二次组会（5 月 20 日）（如图 4.1.2）

1.任庆杰提出迭代方法，但由于未考虑到历史数据而被否决。

2.朱宇森提出安全速度的计算范围，即在保证有效反弹次数的前提下，每个点都有自己的最大安全速度范围。

3.王梦瑶提出基于贝叶斯不完全信息博弈理论下的决策树，但由于技术问题未能采用。

4.王梦瑶提出每次发球控制在我方安全速度之内且对方安全速度之外，倒逼对方每次击球都要改变速度消耗体力，属于强攻型策略。但最终全组讨论得出采用最大化对方消耗和最小化我方消耗的评估函数，函数的具体细节由朱宇森和王冠力完成，同时王冠力提出可以通过假设对方跑位到路程中间的最坏情况来做保守估计。

5.提出机器算法学习对方策略

任务：代码组完成代码初步编写工作，其余组员进行机器学习的了解。



图 4.1.2 第二次组会

（3）第三次组会（5 月 23 日）（如图 4.1.3）

1.初步程序修改和测试

2.代码讲解

任务：寻找预测对方落点和跑位的算法；继续思考道具系统；王冠力单独写一个暴力打角算法作为陪练算法。



图 4.1.3 第三次组会

（4）第四次组会（5 月 24 日）（如图 4.1.4）

1.金山木提出根据落点进行统计，作为预判策略。

2.方睿提出遗传算法。

3.小组统一意见：机器学习难以实现，放弃该方案；接球按照固定算法，道具看情况选用别的算法。

4.现有问题：生命值降低到某一个值后，就出现死角，接不到球，对方打角则容易失败；如果对方先出现死角，我们就果断打角。如果对方用了旋转球，我们就使用安全速度之内的最低消耗。或者不消耗。

5.确定本组头像，选用张怡宁“冷漠脸”动漫头像。

任务：方睿完成遗传算法，测试组开始调参数

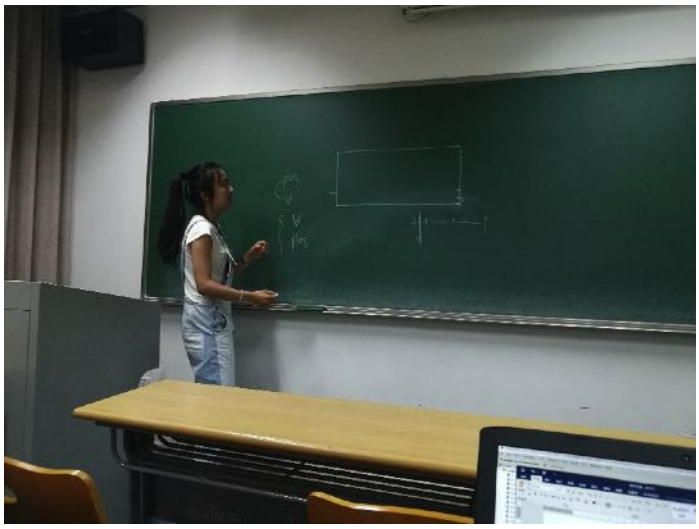


图 4.1.4 第四次组会

（5）第五次组会（5 月 26 日）（如图 4.1.5）

1.朱宇森、王冠力、方睿测试并修改各自程序，准备热身赛。在修改过程中发现，遗传算法暂时无法解决道具问题，和王冠力的算法对打都因为无效击球而输。朱宇森和王冠力调试代码，捡道具情况稍有好转。

2.王梦瑶联系到其他小组，约定另找时间私下对战，以便测试出较好的策略。

3.代码组经反复测试，终于修复了导致不能捡道具的 bug，并根据老师的几次新修改又进行了对应调整。

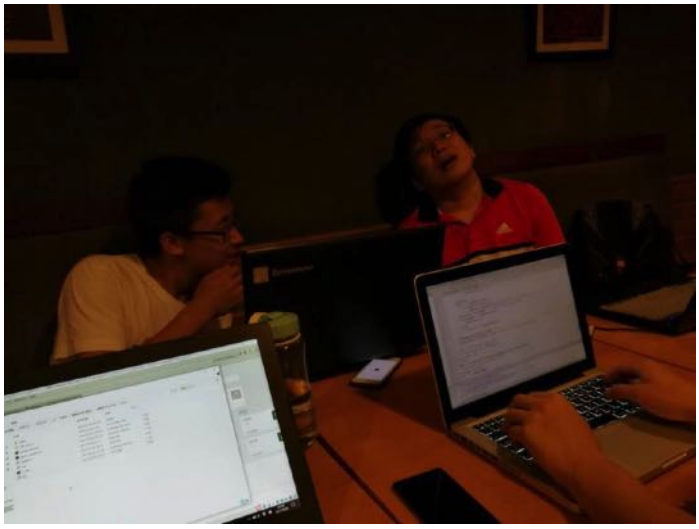


图 4.1.5 第五次组会

（6）第六次组会（5 月 30 日）（如图 4.1.6）

根据第一次热身赛比赛数据，我们的算法胜率在 70%以上，属于胜率较高的小组。

1.朱宇森在现场解决了优化搜索问题，使程序运行速度大大加快；

2.方睿继续研究遗传算法。

3.测试组观看复盘数据，总结出四分之一跑位策略和捡道具策略（重点是血包和变压器）

4.分析输球的主要原因：道具积压太多，还未使用就被挤出了道具包；打角效率在生命值降低到 30000 左右后降低，导致无效反弹；跑位仅仅跑到中间，在与部分算法对战的过程中，每次跑位消耗体力值平均比对方多大约 400。

任务：王冠力、朱宇森、任庆杰继续写优化代码，方睿继续研究遗传算法，唐金潼、金山木、王梦瑶对数据进行测试，调整参数和分段方式。



图 4.1.6 第六次组会

（7）第七次组会（6 月 7 日）（如图 4.1.7）

1.王梦瑶提出统计预判策略。

2.朱宇森和王冠力连夜按照统计预判策略修改代码，代码性能大大改进。

3.测试组观察其他三强的跑位数据，分析对手特点。



图 4.1.7 第七次组会



图 4.1.8 全组合影

4.2 经验与教训

1、关于程序本身

（1）算法没有特别的针对性，但是对每一种固定算法都有良好的克制。在算法中，通过对上一步的记录，来判断对方本次触球后的行为。特别是对于行为模式单一算法（比如策略保守的防御型策略、策略激进的攻击型算法等），我们算法的克制作业特别明显。而对于一般的算法，我们的算法都能利用其行为模式产生比较准确的预测，进而对对手造成压制。

（2）最终版本没有使用的智能算法，是一个遗憾。我们在初期考虑过使用遗传算法、神经网络等，并生成了一些具有一定能力的版本（例如某个版本的遗传算法进化后击败了早期的某个算法）。可惜最终智能算法并没有形成更加有效的策略，所以只能在实战中暂时搁置。

（3）算法改进过程中，参照单一。由于在算法编写的时期，只有自己的算法对抗自己的算法，通过旧版本来检验新版本的实际优劣，所以算法改进对实际能力的影响，很难观测到。首先，对过去的策略的一个小错误的修正，和对算法结构与策略的大变动，结果都是对抗实战的胜率更高，但具体修正后实际能力提

高了多少，在对抗其他对手时是否具有显著优势，却很难观测到。其次，算法结构的改变。最后，囿于已有算法的结构和思路，新的改动往往是对既有算法的细节的修订，对热身赛中其它算法的分析也是建立在已有算法的结构上进行。

（4）数据结构比较简单。纵观整个算法，以线性结构为主，并没有使用到树结构和图结构。而在实际的回合类对抗游戏中，其实符合一个二叉树结构；而图结构在搜索中（包括广度优先搜索与深度优先搜索）具有良好的应用。如果能够考虑和借鉴树结构与图结构的优点，并在算法中加以应用，可能会对我们的策略产生较好的改进。

2、关于小组合作

（1）团队合作实现 1+1>2。本次小组作业中，大家群策群力，分工明确。朱宇森和王冠力具有优良的代码实现能力，程序主要由两位同学完成；王梦瑶作为组长，一方面积极协调每个成员的任务进度，金山木和唐金潼同学，虽然没有直接参与到代码的工作中，但是在策略的构思中也做了许多积极的贡献。特别是通过不断分析热身赛结果中其他对手的策略，来寻找我方算法的改进点。可以说，小组所有成员都积极参与、各司其职、各尽其能，才有了算法一次次的优化与提高，才有了最终的成绩。

（2）合作平台的使用，是我们组要加强的地方。本次期末实习作业使用了 github 作为项目合作平台，但是我们小组并没有使用 github 作为小组项目的共享工具，而是通过传统的文件发送的方式来进行。一定程度上，这降低了团队合作开发的效率。

（3）代码注释的必要性。在团队合作中，不同人的代码风格不同，使用丰富的注释，来增强代码的可读性，可以大大提高合作开发的效率。

（4）不断改进程序。我们的算法，从初期开始，到实战的前一天晚上，我们一直在对算法不断地分析、优化。最后投入实战的算法共有 4 版。每一次，或者是针对已有错误的修正，或者是将借鉴的其它算法优点的引入。每一次修正后，通过既有旧版本进行检验，判断新算法的改进效果。

4.3 建议与设想

1、建议

希望在发布大作业后降低修改源代码和参数的频率，在发布大作业前提早开启源代码的编写和完善工作，将一个已经相对成熟的大作业代码发布出来，便于同学们一次性完成，而不是一遍一遍地适应代码的变动，耽误时间。

2、后期设想

由于方睿编写的遗传算法[1][2]与神经网络算法[3]在与 T_RAW 的实战中胜率过低，暂时未显示出较强的能力，我们最后没有采用。

（1）遗传算法的思路

I.基因和个体的编码

利用“查表法”来决定每一步的策略。具体来说，就是利用对方击过来的球的速度和落点位置，来唯一确定我方击球的速度和跑位（图 4.3.1）。

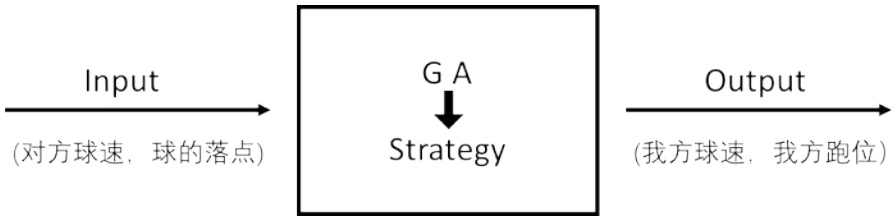


图 4.3.1 根据对方球速、球的落点确定我方球速、跑位

为了减少参数空间，我们将输入的对方球速、球的落点划定区间，落入相同区间的输入值将对应同一个输出。每一个区间分别对应一个确定的输出情况（最开始是随机生成的），称为一个基因；包含了所有区间的对应情况的集合称为一个个体（或染色体）。我们产生大量这样的个体，就能进行遗传算法的进化过程。

II.遗传算子

我们采用重组、突变和复制等遗传算子来使得遗传算法能够找到最优解。重组即选中的两个个体交换一部分基因；突变即部分个体的部分基因发生突变；复制即保留个体，或者引申为两个个体按照一定的重组规则产生下一代个体。在每一轮进化筛选过程中，都要经历上述过程。

III.筛选机制

这是算法能否演化出一个最优解的核心，也是我们遇到问题的所在。筛选机制决定了每一轮进化中，哪些个体将被淘汰，哪些个体将被保留，进入下一轮筛选。实际采用的筛选机制有两种：

①个体之间捉对厮杀，淘汰输者，保留胜利者；

②和一个测试算法（如 T_idiot）进行厮杀，淘汰输者，保留胜利者。

（2）神经网络算法的思路

采用三层神经网络，分别为输入（对方打来球的落点 pos，球的速度 vel，对方位置 oppos，我方血值 life）、隐藏层（神经元个数可调）、输出（我方跑位 run，我方击球球速 vel）。

各个信号通路的权重 weight 为（-1，1）浮点数，输入信号为（0，1）浮点数（需要提前归一化），每个节点接收到的信号为 s=∑weight*signal

响应函数为 logistic 形：1/(1+exp(-s/p)), p 值可调， 默认取 1。

通过遗传算法优化各条通路上的 weight。

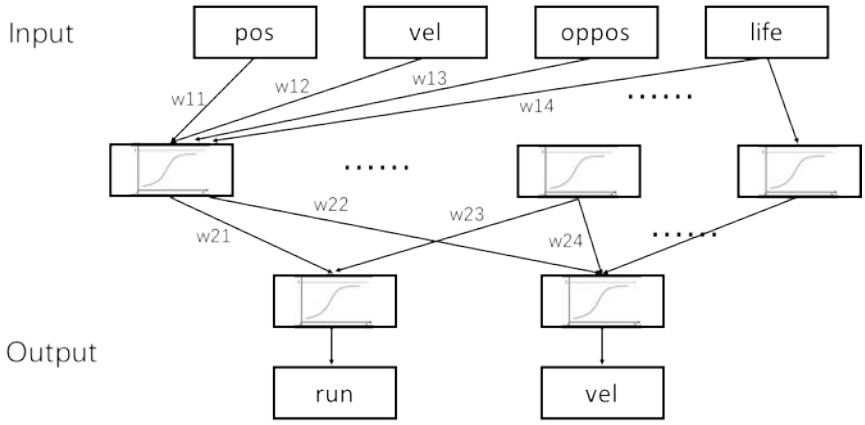


图 4.3.2 神经网络算法示意图

具体实现过程和结果分析详见附加文档：《基于遗传算法与神经网络的尝试》。

若有机会，希望将这两种算法付诸实践，发挥应有威力。

3、对学弟学妹的寄语

这门课程最酷的一点是——你遇到的许多问题都可以被抽象成各种数据结构与算法——换句话说，就是用数据的眼光看待世界。尽管你不一定会喜欢上编程，但这门课所带来的全新的思维方式会有助于你以后解决五花八门的问题，实现千奇百怪的想法，甚至，它会更新你原本对事物的认知，带来前所未有的思想与情感体验。所以，用探索的心态享受这门课程吧！

5 致谢

首先感谢陈老师和助教们一学期的热切指导和辛勤付出！

感谢 Charlie、Kilo、Oscar 组和我们进行热身赛，帮助我们提升算法兼容度！

感谢 Alpha 组在决赛环节对我们的帮助！

最终特别感谢冀锐助教帮我们测试代码、回传数据！

6 参考文献

1. 梅拉尼·米歇尔. 复杂[M]. 长沙:湖南科学技术出版社, 2011. 164-178

2. zzwu. 神经网络入门(连载之五)[EB/OL]. http://blog.csdn.net/zzwu/article/details/576531, 2006-01-11

3. 赖勇浩. 用遗传算法加强足球游戏的人工智能[J]. 程序员：游戏创造, 2006, (10): 67-71

附录：基于遗传算法与神经网络的尝试

方睿

【背景】

组里在分工时，被派到研究智能算法，开始的想法是利用机器学习，但考虑到正式比赛时局数过少，无法提供足够的数据用于学习，所以放弃了这个想法。选择遗传算法是因为很久以前在《复杂》这本书中看到过一个扫地机器人的例子[1]。其中书中的一段话打动了我：“GA 经常能演化出有用的答案，但是很难看出为什么会有用。这是因为 GA 找到的好答案与人类想出的相当不同。”当所有的小组都势必会关注打角、绝杀等技巧时，通过遗传算法是否能得到一种完全不同的策略，使得我们的程序能够获得胜利？

而使用神经网络是因为遗传算法遇到了“不进化”的问题，尝试通过连续的函数变换代替遗传算法的“查表法”，这在后续会细说。

由于组里的大神们在开发固定算法，因此我尝试 GA 和 NN 的出发点也仅仅是尝试。希望还是要有的，万一见鬼了呢？

【遗传算法部分】

【设计思路】

1.基因和个体的编码

前面已经提到了，决定利用“查表法”来决定每一步的策略。具体来说，就是利用对方击过来的球的速度和落点位置，来唯一确定我方击球的速度和跑位（图 1）。

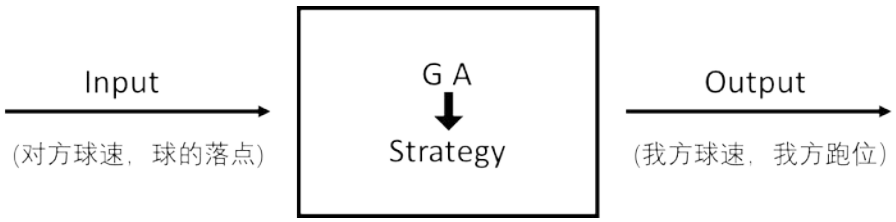


图 1

为了减少参数空间，我们将输入的对方球速、球的落点划定区间，落入相同区间的输入值将对应同一个输出。每一个区间分别对应一个确定的输出情况（最开始是随机生成的），称为一个基因；包含了所有区间的对应情况的集合称为一个个体（或染色体）。我们产生大量这样的个体，就能进行遗传算法的进化过程。

2.遗传算子

[1] 《复杂》，梅拉尼.米歇尔，2009，中文版 2011，湖南科技出版社，P164-178。

我们采用重组、突变和复制等遗传算子来使得遗传算法能够找到最优解。重组即选中的两个个体交换一部分基因；突变即部分个体的部分基因发生突变；复制即保留个体，或者引申为两个个体按照一定的重组规则产生下一代个体。在每一轮进化筛选过程中，都要经历上述过程。

3.筛选机制

这是算法能否演化出一个最优解的核心，也是我们遇到问题的所在。筛选机制决定了每一轮进化中，哪些个体将被淘汰，哪些个体将被保留，进入下一轮筛选。实际采用的筛选机制有两种：

①个体之间捉对厮杀，淘汰输者，保留胜利者；

②和一个测试算法（如 T_idiot）进行厮杀，淘汰输者，保留胜利者。

实际上，这两种机制都存在一定的问题，后续将进一步探讨。

4.一些琐碎

由于最终得到的其实是一张索引表，所以只需要将最终的进化结果导出为 txt 文件，要进行比赛时再读入到 T_GA 中，T_GA 就是一个通过遗传算法演化得到的固定算法，每次在得到球的速度和位置信息以后，查表得到我方需要作出的回应，输出相应操作即可。

另外，很重要的一点是由于开始的较早，道具系统还未完善，并且考虑到道具会大大增加参数空间复杂性，因此全程都没有涉及到道具的使用。

【具体实现】[2]

在两周中，我先后尝试了五个版本的遗传算法，并将所得到的结果进行测试。这五个版本分别为 gene_screen_1-5，其中主要的函数和类有：

类：

Gene：一个基因，定义见【设计思路】。含有方法“mut”，进行基因突变。

competitor：是为了将全部基因信息在比赛时转化为具体的操作。

函数：

GeneGenerate：产生一个个体。

GroupGenerate：产生一个待进化的群体。

race：基本来自 pingpong.py，实现具体的对抗。

evolution：这是算法的核心，包含了进化的全过程，尤其是筛选的机制，五个版本也基本

[2] 详见代码 gene_screen_1-5.py。

上只有这一部分进行了改动。

下面逐一介绍各个版本的筛选机制，为了方便明白相互之间的联系，将他们各自的问题也写在下方。

gene_screen_1

两两厮杀，淘汰失败者，每次筛选掉一半，最后剩下一个即为演化出的策略。

问题：确定了群体的初始数量后，演化的代数就确定了。在尝试的一些取值的情况下，所得到的最后结果都很弱。

gene_screen_2

和组员最先写出的一个固定算法 T_RAW 比赛，淘汰掉失败者，如果随着演化的进行，仅能剩下一个胜利者,则选择这一胜利者为最优策略;如果最后有固定个数的胜利者一直能活下来，就任意选择一个作为最优策略。

问题：在最开始筛选的时候全部个体就都输给了 T_RAW 然后被淘汰了,群体数量瞬间变为 0。

gene_screen_3

以 T_idiot 作为最初的选择压，根据赢 T_idiot 的局数将群体 rank，选择排名首位的作为新的选择压，淘汰部分个体、补充新个体保证群体数量不变，演化固定代数以后将排名首位的输出作为最优策略。

问题：T_idiot 很容易被打败，所以很可能大家的水平都一般般，收敛到了一个很弱的算法上。最终的结果也不理想。

gene_screen_4

以 T_RAW 作为最初的选择压，根据赢的局数和剩余生命值双重标准递减排序，将群体 rank，淘汰掉排名靠后的部分个体，由排名靠前的个体繁殖补充个体，保证群体数量不变，演化固定代数后输出排名首位的作为最优策略。

问题：赢的局数一直都是 0，所以这一指标的 rank 没有任何意义，在演化了很多代之后大家依旧都是 0；多加入一个生命值进行 rank 也没有明显的改进。

gene_screen_5

回到两两厮杀的策略，但每次淘汰掉一半以后，通过胜利者的繁殖补充新个体使得群体数量不变，演化固定代数以后回到 gene_screen_1（在 gene_screen_5 中对应的为 gene_screen_final），厮杀到只剩下一个个体输出为最优策略。

问题：演化了好几天，也没有看到明显的进步。

【结果分析】

遗憾的是，尽管尝试了两个礼拜，但是最终并没有演化出能打败 T_RAW 的个体，尽管他们基本都能够打败 T_idiot。回想起来，整个过程其实存在很多的问题，而这些问题都没有得到解决，因此失败似乎也是很正常的。

1.可以看到，这五个版本基本绕不开捉对厮杀或和一个固定算法进行厮杀这两种筛选机制。捉对厮杀似乎容易陷入局域最小值，形成一种“在矮子里面挑长子”的局面；而引入一个选择压则要么太弱，所有个体都能活下来，没有发挥选择作用。要么太强，所有个体都被淘汰。在整个过程中，我一直试图找到一个合适的筛选机制，譬如改变 rank 的方法，然而由于理论知识的局限性，加上对这个系统的了解过于浅薄了，最终都没有什么好点子。实际上想想也是，遗传算法不会如此简单。如果这样几百行代码简单的进化几天就可以得到一个天下无敌的算法，那简直不可思议。

2.能够采用遗传算法来演化得到一个最优策略的前提条件是，这个游戏确实存在一个全局的最优解，这样只要功夫深，或许总有一天能找到它。但是我们连这个前提条件都无法证明。因为这是一个交互式的游戏，场上的情况不是完全确定的，它比能用遗传算法解决的背包问题等复杂很多。

3.我们将速度和位置分到了有限的区间里，进行策略的决定。而实际上，速度和位置的可能取值范围非常大，相比之下区间的个数就显得略少了。采用简单粗暴的划分可能会对结果产生很大影响，这也是导致我接着尝试神经网络的主要原因。

4.前面已经提到了，这个方法没有考虑任何道具的使用，而加入了道具会使得整个系统变得复杂得多，这都是没有仔细考虑的，也注定了它不会被选为参赛选手。

【神经网络部分】

虽然称其为神经网络，但是这其实依旧是建立在遗传算法的基础上的，唯一改变的是由“查表法”变为了“函数法”，简单来说，之前得到的最终策略是一张表，而现在得到的是一个将输入映射为输出的函数关系。后一种方法的优点是，这时候不需要再划分区间，输入是什么值，进行计算的时候就用什么值，可以唯一确定一个输出。但是这个函数关系还是通过遗传算法来演化。演化过程和前面提到的类似，就不重复说了，下面仅简单介绍一下设计思路和最终的结果，具体的实现可以参考代码包 NeuralNetwork。

【设计思路】

采用三层神经网络，分别为输入（对方打来球的落点 pos，球的速度 vel，对方位置 oppos，我方血值 life）、隐藏层（神经元个数可调）、输出（我方跑位 run，我方击球球速 vel）。

各个信号通路的权重 weight 为（-1，1）浮点数，输入信号为（0，1）浮点数（需要提

前归一化），每个节点接收到的信号为 s=Σweight*signal

响应函数为 logistic 形：1/(1+exp(-s/p))，p 值可调， 默认取 1。

通过遗传算法优化各条通路上的 weight。

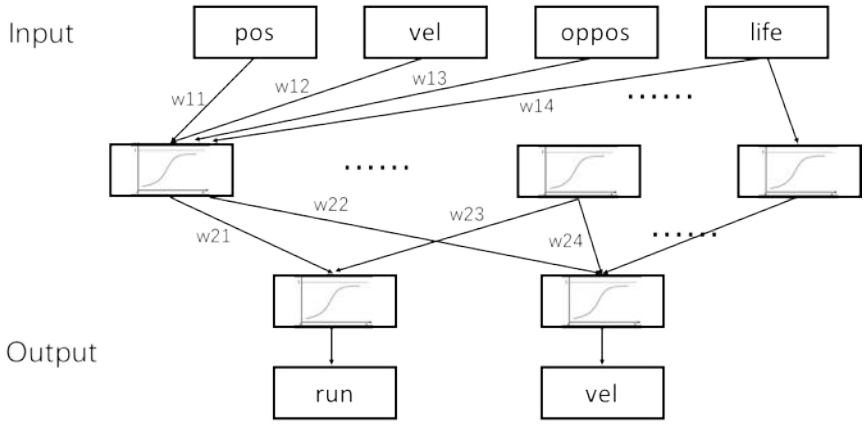


图 2

【结果分析】

效果比遗传算法似乎要好，在进化的开始阶段，一段时间以后的策略总能完胜早一些生成的策略，甚至在某一天进化出了能打败 T_RAW 的策略。但是随着进化时间延长，策略开始退步了。怀疑陷入了局域极小值，但是为什么会退步不得而知。

不仅如此，改变参数重新开始进化，之后就没有得到过能打败 T_RAW 的策略了。神经网络加上遗传算法得到的结果如此不稳定，让人感到非常费解。细想应该还是因为对这个系统了解过少，没有抓住核心问题吧。

但是唯一值得肯定的是，通过开始的的确确观察到的进步阶段，可以看出将离散的查表法改成函数法，还是一定程度上优化了我们的算法。

【总结】

这短短两周的时间，充分体验了一把“现学现用”的感觉，遗憾的是仅仅只是学了皮毛中的皮毛，无法很好地解决问题，甚至都不知道问题的所在。所以最后我的这一套方法都没有什么实际的效用。所幸组员们非常厉害，我们最终的算法非常厉害，当然这都是后话了。

无论是遗传算法还是神经网络，应该都是高深的学问，大概需要系统的学习才能够对它们有深刻的了解。尤其是神经网络结合机器学习，是这次大家都很想实现，但最终因为时间和精力的有限而放弃了的。

虽然最后有一种“哈，果然失败了”的宿命感，但是这的确激起了我对遗传算法的兴趣。通篇所述仅仅是一次尝试，但愿在未来能有机会真正实现它们吧。



GOLF

组长：林荣

组员：曾挺，胡宇城，隋清宇，张卓，李宇轩，王悦翔

小组赛战绩：W1

决赛战绩：季军





GOLF——一流程序员



数据结构与算法课程实习作业报告

组名：GOLF

组员[3]：胡宇城，林荣*，李宇轩，隋清宇，王悦翔，曾挺，张卓；均为地球与空间科学学院 16 级本科生。

摘要：

本报告反映了我组在本次数算大作业中的主要工作和经历，从算法出发，细致地阐述了我组算法的原理，即找出一步最优的解，又从数据结构、函数等方面展示了实现该算法的程序，并附上了前期测试中的结果。此外，本报告还记录了我组在完成作业中经历的各个阶段及我组成员的心路历程，并对本次作业进行了总结与反思。

关键字：

单步最优 打角策略 道具价值 均衡跑位

[3]按拼音顺序排列，带*的为组长

1 算法思想

1.1 总体思路

在漂移乒乓游戏中，我们要返回的最重要的两个参数是加速度和跑位，这也是各组代码的差异所在。

就加速度这一问题而言，打角策略是各组公认的最优策略。因此我组算法也以打角作为最基本的思路。但是即使是打角，打哪个角，用什么方式打角仍然是值得考虑的问题。针对这些问题，我们开发了两套算法，一套利用了存储的历史数据来预测对手下一次的行为，从而有针对性地打角（在小组赛及八进四中使用）；另一套（这也是我组报告中描述的算法）则是基于我们八强赛输给 Lima 之后以及观看与各组复盘后讨论的结果。这一算法放弃了去“赌”对方会趋向于采取什么方式回球，直接采取单步最优策略，这更具有普适性。在选定了打角的具体方案后，我们再根据预先设定（并不断调试）的道具价值计算捡道具的收益，通过与打角方案的收益进行比较决定是否用能捡到道具的加速度来替代原先加速度。

就跑位这一问题而言，我们在第一套算法中是凭着感觉在中间，四等分点及其附近进行不断调试确定具体位置，这个纯粹靠主观感觉确定位置的方案虽然在之前的赛事中取得了一定的成果，但是我们对此仍然不满意。在最终的算法中，我们放弃了纯粹主观的跑位方法，而是计算出了对手的两种主要回球方式对我方影响均等的位置，向这个位置进行跑位。具体的计算方法是假定对手和我们采取了同样的打角策略，进行分析。以打上角为例：假设我们以接近 0 的水平方向速度从上角打到对手的上角，然后进行跑位。对手可以以接近 0 的加速度使球原路返回，击打我方上角，这样我方少跑位可以极大省体力，但是对方给球加速也几乎没有耗费体力；对手也可以击打我方下角，使得我方跑位加迎球消耗大量体力，但对手在对球加速的过程中也会消耗相当多的体力。我们计算出对手这两种主要回球方式对我方影响均等的位置，向这个位置进行跑位，不论对手采取怎样的策略，我方都能不处于劣势，也减小了偶然性。之后与一些智能化程度比较高的算法的对战也印证了这种跑位方式在血量充足的情况下是高效率的。

对于数据结构的应用，我们算出了所有打角方案的加速度，并把他们按照打上角，打下角分别储存到了两个列表中；对于道具的价值，我们则是算出了获得道具合理加速度中绝对值最小的，然后生成了字典，场上的每个道具作为键值对应获取这些道具需要的加速度；同时，设定各个道具对应的价值也是通过字典的方式实现的。

1.2 算法流程及复杂度分析

第一部分：初始发球

在这一步中，始终是稳定的站到中间发上角策略。打到上角的速度有 278（跑 0.5*DIM[3]）和 1390（跑 2.5*DIM[3]）两种，我们选择 278，此处没有过多的考虑。

第二部分：主算法（打球函数 play）部分

第一步：利用副算法中的加速度列表生成函数得到所有打角的加速度，并把它们归类为打上

角和打下角。因为碰墙次数的限制，无论站到哪个位置，可选的打角加速度都是有限的，因此过程的复杂度为 O(1)。

第二步：把每一个加速度都拿出来进行分析，假设对手采取了能够打角的绝对值最小的加速度；我们分析对手迎球、加速的消耗和我们打出球，跑位，以及去迎球的消耗，然后选取我方消耗减对手消耗最小的加速度进行回应。但是考虑到我们可能无意中连续打同一个角从而触发对手的偷懒算法，我们增加了在一定情况下来回打角的方案；又考虑到残局时加速度的消耗过大可能让我们无法承受，因此我们也加入了在一定条件始终以绝对值最小的打角加速度进行回应的方案。因为每个人打角的加速度都是有限个确定的值，因此无论怎么假定和考虑，这一步总的时间复杂度保持在 O(1)。

第三步：利用副算法部分中的捡道具函数对第二步中得到的加速度值进行修正。此过程的复杂度将在副算法内进行分析。

第四步：考虑对手使用旋转球的情况，对加速度值再次修正，如果对手使用旋转球，我方选择最小加速度回球。此步优先于前面所有的加速度选择，一是为了防止碰撞次数不满足条件，二是为了减少加速造成的额外消耗（这是主要的，因为如果选择最小加速度回球一般 acc 都是 0，不然很有可能消耗到 3000 以上，很伤）。此过程复杂度为 O(1)。

第五步：根据前文所说方案进行跑位。我们考虑到了残局[4]时可能会因为跑到比较靠边的位置而在三四万血量的时候接不到球，因此我们选择在自己体力占有或持有瞬移的时候稳定跑到中间位置。同时，因为时间截止的时候球没有到达我们这里，但是跑位消耗已经产生，为了防止这一因素对我们有不利影响，我们在比赛即将结束的时候不跑位。因为此过程只有简单的判断和赋值，且步数一定，因此复杂度为 O(1)。

第六步：利用副算法部分中的使用道具函数返回使用道具的相关内容。因为道具箱内的道具数量最多为 3，且一趟只能使用一个道具，因此本步的算法复杂度为 O(1)。

总的来说，我们用的都是 if、elif 和 else，所以算法的复杂度基本上都是 O(1)（很惭愧）。

第三部分：副算法（辅助函数）部分

函数一 absmin：给定正整数为元素的列表，返回其中绝对值最小的一个。在此过程中，我们选择了类似冒泡排序的方法，但是因为我们只需要绝对值最小的元素，因此进行一遍冒泡过程就可以实现目的。此过程的复杂度为 O(n)，但是考虑到在漂移乒乓的环境下，列表中的元素值始终是恒定的，此过程复杂度退化为 O(1)。

函数二 activatelist：给定当前拍的位置和球的速度，生成所有打角加速度的函数。主要思想是通过镜面对称来计算结果。因为加速度的个数恒定，因此复杂度为 O(1)。

函数三 howfast：根据球在拍上反弹之前的速度计算对手的出球速度。也可以用于计算我方的发球到对方拍时是否反向。复杂度 O(1)。

函数四 where：根据击完球的速度和当前位置计算球到对面的位置，这一方法从 table.py 中原样搬来。用于在后续函数中计算道具的收益。复杂度 O(1)。

函数五 accelerate_range：计算在当前形势下所有合法加速度的列表。本来是为一个老版的组内算法定做的，但是因为这个函数可以用于判断捡道具的加速度是否合法，因此得以保留。复杂度 O(1)。

函数六 eatcard：副算法中最长，也是最复杂的函数。通过对称得到能捡到道具的各个加速度，然后生成字典，分别储存每个道具的收益和能捡到它们最划算的加速度。通过与传入的不考虑道具的最优加速度比较，确定是否用捡道具的加速度替换原先的加速度。如果不替换，则返回输入的加速度。此过程中大量应用了字典这一 Python 自带的数据结构，在使用的过程中，因为 card 类无法作为字典的键值，因此我们只能用坐标构成的数组来作为键值。因

[4]我们把残局定义为我方血量已经下降到了有接球盲区之后的情况

为场上道具数有最大限制，对于每个道具，对称的方案又是有限的，因此此函数时间复杂度为 O(1)。
函数七 TPexist：判断箱内有没有瞬移。因为主函数中有“残局下我方优势或持有瞬移道具，则跑位中间求稳”的设定，因此我们通过返回一个布尔值来实现这一判断。
函数八 usecard：使用道具函数，除了瞬移留在手中，待残局需要长途跑位时再使用以外，其余道具都是能用则立刻用出。如果箱内没有道具，则返回(None, None)。

2 代码分析

对于我方算法的及程序代码的解析，上文（以及程序代码中的注释）已有说明，在这里说说我方组内对自身代码进行分析的主要方式。

我方的代码经过了前几次组会的讨论已经有了基本的发展方向，在四强赛之前也只进行了各个参数的小改。但是我们发现，与其分析对手过去的选择，然后猜测他下一次的行为，不如我们假设对手采取了一个较优解，然后我们去针对这个较优解。基于这个想法，我们在四强赛的前一天晚上对算法的基本方向进行了一次大改。但是因为我们对于对手下一次可能采取的策略仍然是基于我们的主观猜测，因此如果对手的算法比我们的假定智能，我们的算法就可能被单方面克制得很惨。

其次，我们的一些参数是观看与强组（如 Oscar）之间的复盘得到的，我们对于这些参数的得出过程，以及真正的触发机制仍然不甚了解，因此遇到原版的智能打法仍然不好应对。

因为我们在计算打角时，没有真正的“打角”，而是给出了一个允许的误差范围，因此我们不会因为地板除的机制而出现意外死亡（invalid bounce）的情况。

2.1 数据结构说明

实在是非常的惭愧.....因为我们用到的数据结构可能仅限于列表和平台代码里定义的类。主要数据结构就是列表。

2.2 函数说明

其实在 1.2 里已经写出了我们的所有函数及其复杂度分析，就不再赘述了.....

2.3 程序限制

程序出错？不存在的。因为情况考虑的还算是比较周全的，应该是不会出 bug 的（反正比赛的时候没出）。值得一提的是 pingpong.py 更新到最新版之后，之前出 bug 的程序不会报错了，跑的时候会显示 run error.....然后使 debug 变得异常的困难。不过后来发现一般出现 run error 都是函数传参的时候出了偏差。

3 实验结果

3.1 测试数据

（附一张图）

这是测试主力林荣电脑的运行环境：

OS: Microsoft Windows 7

版本 6.1.7601 Service Pack 1 内部版本 7601

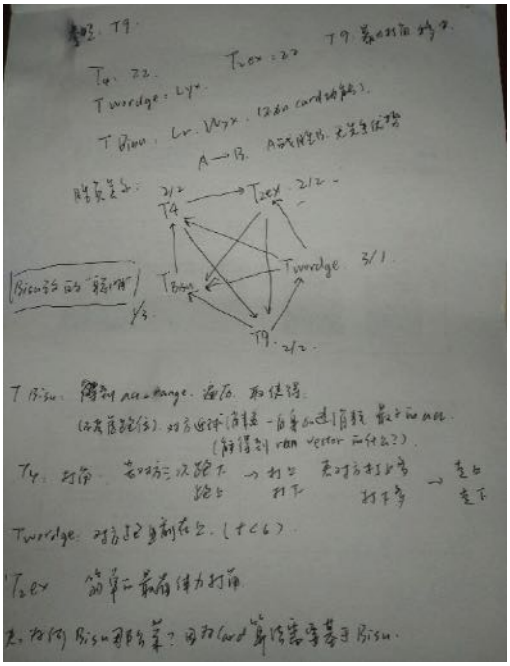
PyCharm 版本：

PyCharm Community Edition 2016.3.3

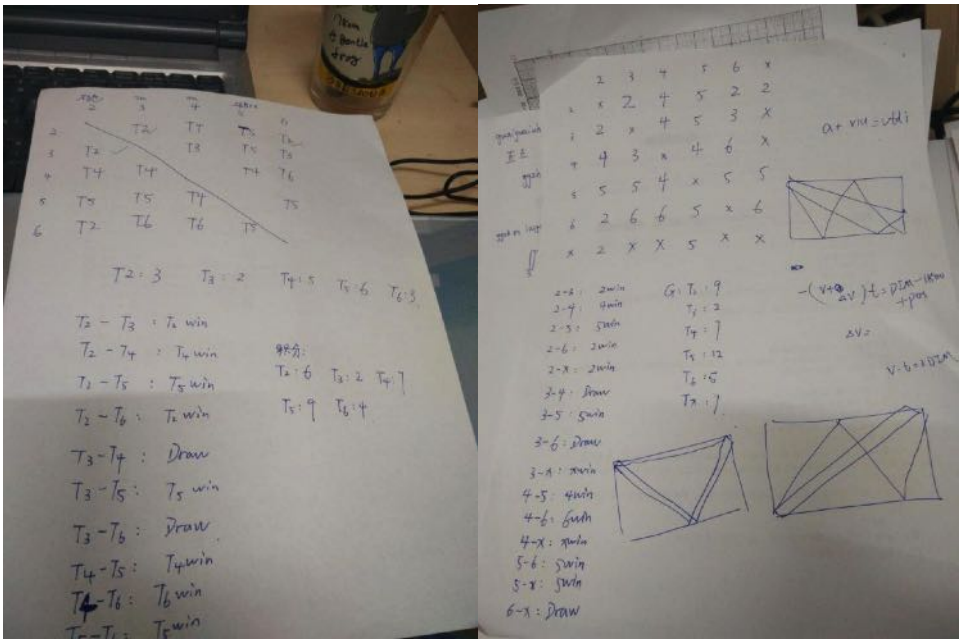
built on March 11, 2017

JRE: 1.8.0_112-release-408-b6 x86

JVM: OpenJDK Server VM by JetBrainss.r.o



（右图为林荣所做的多算法胜负关系分析）



我们会把有价值的算法存起来，测试新算法的方法就是把所有的老算法拿出来对打……也没有编写测试程序，统计全靠人工。（上图是张卓的手稿）

在第一次热身赛中，我们的成绩差强人意，虽然提交上去的几个程序忘记了加入道具系统，但是胜率也比较可观。而在第二次热身赛中，我们的程序取得了相当优异的成绩。只有当时的 sabac 和 lkjhg 真正打败了我们。但是因为我们最终在四强赛和三四名争夺中采取的算法只参加了那两场正式的比赛，所以相关的数据也不是很多。

3.2 结果分析

在热身赛的第一版算法（利用历史数据）中，我们对于对手打角跑位规律的总结确实为我们节省了不少体力，而针对对手的偷懒跑位，我们的不连续多次打同一个角也让对手猝不及防。但是这些设定在之后的四强赛算法中都被抛弃了，因为四强赛中大家的算法都进行了一定的考虑，算法的机制也使得它很难再出现连续打一个角的情况，跑位也不再那么偷懒，因此我们从看历史的行程变成了去尽量知道不可预料的未来。这一改动的最初版写好后，就立刻大胜之前利用历史的算法，因此我们也就在四强赛的前一晚上对这一算法进行了彻夜的完善。

然而，由于我们采取的是单步最优策略，可以说只是冠军算法 Oscar 的一个单步近似算法，因此虽然我们的程序运行较快，但始终无法战胜采用深度搜索的更加智能的对手，不过这样一个简单的算法没有以 40-0 被击败，已经满足了我们的愿望。

3.3 经典战局

在四强赛当天的早上两点，我们通过了一次调试后再次开始了测试，这一次测试的结局是相当令人激动的，我们最终以 21-19 的微弱优势战胜了 Oscar 组发给我们的老版代码（即他们组在四强赛中也采用了改动很大的全新代码）。图为我们的一部分测试结果！当我们得到这个结果后，心情十分激动，这也又给了我们继续工作的动力！

```
000) West win! for miss_ball, West:T_GOLFf(32816), East:T_Oscar(8685),总时间: 144000 ticks
player_pos:<900000,305600> ball_pos:<900000,999200>
001) West win! for miss_ball, West:T_GOLFf(30253), East:T_Oscar(8416),总时间: 158400 ticks
player_pos:<900000,293600> ball_pos:<900000,998800>
002) West win! for miss_ball, West:T_GOLFf(26520), East:T_Oscar(8055),总时间: 162000 ticks
player_pos:<-900000,971862> ball_pos:<-900000,996600>
003) East win! for miss_ball, West:T_GOLFf(30483), East:T_Oscar(30858),总时间: 185400 ticks
player_pos:<900000,307000> ball_pos:<900000,999600>
004) West win! for miss_ball, West:T_GOLFf(26462), East:T_Oscar(8453),总时间: 255600 ticks
player_pos:<900000,256400> ball_pos:<900000,1200>
005) West win! for miss_ball, West:T_GOLFf(26954), East:T_Oscar(20175),总时间: 162000 ticks
player_pos:<900000,925600> ball_pos:<900000,999600>
006) West win! for miss_ball, West:T_GOLFf(28342), East:T_Oscar(24939),总时间: 165600 ticks
player_pos:<900000,680200> ball_pos:<900000,200>
007) West win! for miss_ball, West:T_GOLFf(28691), East:T_Oscar(8794),总时间: 165600 ticks
player_pos:<900000,682000> ball_pos:<900000,111800>
008) West win! for miss_ball, West:T_GOLFf(27019), East:T_Oscar(8688),总时间: 165600 ticks
player_pos:<-900000,165527> ball_pos:<-900000,1400>
009) East win! for miss_ball, West:T_GOLFf(26522), East:T_Oscar(26634),总时间: 189000 ticks
player_pos:<900000,254800> ball_pos:<900000,1400>
010) West win! for miss_ball, West:T_GOLFf(34472), East:T_Oscar(20186),总时间: 147600 ticks
player_pos:<-900000,884680> ball_pos:<-900000,910400>
011) East win! for miss_ball, West:T_GOLFf(29252), East:T_Oscar(31630),总时间: 163800 ticks
player_pos:<-900000,922307> ball_pos:<-900000,999000>
012) East win! for miss_ball, West:T_GOLFf(27967), East:T_Oscar(29872),总时间: 203400 ticks
player_pos:<900000,705800> ball_pos:<900000,600>
013) West win! for miss_ball, West:T_GOLFf(28757), East:T_Oscar(8056),总时间: 165600 ticks
player_pos:<900000,873400> ball_pos:<900000,887400>
014) West win! for miss_ball, West:T_GOLFf(28990), East:T_Oscar(23620),总时间: 144000 ticks
player_pos:<-900000,913424> ball_pos:<-900000,918600>
015) East win! for miss_ball, West:T_GOLFf(30776), East:T_Oscar(34499),总时间: 199800 ticks
player_pos:<900000,275400> ball_pos:<900000,998600>
016) West win! for miss_ball, West:T_GOLFf(29572), East:T_Oscar(7594),总时间: 144000 ticks
player_pos:<900000,12720> ball_pos:<900000,2000>
017) West win! for miss_ball, West:T_GOLFf(27357), East:T_Oscar(25320),总时间: 162000 ticks
player_pos:<-900000,856003> ball_pos:<-900000,965400>
018) East win! for miss_ball, West:T_GOLFf(27671), East:T_Oscar(9070),总时间: 156600 ticks
player_pos:<900000,90600> ball_pos:<900000,1000>
019) West win! for miss_ball, West:T_GOLFf(27970), East:T_Oscar(23547),总时间: 230400 ticks
```

4 实习过程中历史的行程

4.1 第一阶段——在迷茫中探索

刚开始的时候，因为平台本身的功能也不完善，因此我们暂时处于思考，讨论阶段；此时的代码进程十分缓慢。在研讨中，我们出现了两种主要的想法：一种是暴力打角（我组的“T_数字”系列算法），另一种是求出所有合法的加速度区间（我组的“T_Bisu”算法），遍历每一个加速度，并计算以该加速度打出后的收益，选择收益最大的加速度回应。此时对于跑位的理解也只局限于跑位中间。

因为在实战中，第二种算法（遍历所有加速度）得到的最优解仍然是打角，所以我们重点开发了多种打角的方法：来回打角；最省体力打角；根据对手来球位置打远角。此时的代码相当简陋，甚至存在着致命的错误。在 T_Bisu 算法中，因为混淆了 tb.ball[‘position’].y 和 tb.side[‘position’]曾经出现过第一次迎球就死于碰墙三次的情况，好在经过修复后，我们满足了算法无 bug，也不会死于非法碰撞的要求。因为此时的旋转球说明是“使对手无法加速”，因此我们的工作也涉及到了研究旋转球完杀对手的打法。

虽然两种算法最终得到了同样的结果，但是因为考虑到体力消耗参数可能仍然有后续的改动，遍历算法并没有被完全废弃。

4.2 第二阶段——智能的道具时代

随着道具系统的修改和完善，我们开始把道具作为另一个重头进行考虑。起初，我们为遍历算法量身打造了捡道具系统，针对每一个加速度，抄来了 table.py 中判断是否获得道具的代码，但是这个版本的道具代码很快被废弃了，原因有二：一点在于遍历算法本身性能已经比不过经过各种优化，考虑更多的打角算法；其次在于有的时候我们会因为不明原因无法获得道具。但是在这一版本的道具代码中，为每一个道具设置一个心理价值，然后比较捡与不捡的收益这一点已经确立。

于是，在第二版的道具算法中，我们回归了镜面对称确定所需加速度的方法。我们通过对称获得了道具的五个位置，然后判断五个加速度是否合法，并考虑到了我方加速消耗和到达对手区域时对手的消耗。这一代代码可以在各种主算法之间通用，且捡道具的效率高于之前的版本，因此即使到了决赛，我们捡道具的方法也是基于这个版本进行了调整。

与此同时，对于 ds 的使用也在进行。我们在组内对战的时候发现有的时候会出现连着多次打同一个角的情况发生，于是我们启用了 ds，增加了如果对手连着几次打同一个角就减少跑位距离的功能。同时，我们为了防止对手也有这种功能，也加入了不能连续多次打同一个角的策略。之后，我们多次调试，修改了“减少跑位距离”的具体数值，达到了一个全新的代码强度！之后我们只对这一代代码进行了微小的修饰，修改内容也无非是跑位跑多少，道具的权重设定为多少等问题。就这样，我们迎来了第二次热身赛。

4.3 第三阶段——傲视群雄的成就感

第二轮热身赛中，我们取得了相当好的成绩：从大比分上看，只有两个组胜过了我们。因此，我们研究决定最终提交热身赛中的算法。但是提交前，我们再次对各种参数进行了调整。

虽然我们取得了相当好的战绩，但是我们没有被冲昏头脑，我们发现了强敌 sabac（就是后来的冠军算法 Oscar），在热身赛的五局对抗中，我们以全负战绩输给了对手。当时观看复盘时，只觉得对手跑位超出了我们的认知；对手也仿佛知道我们要干什么一般，打得十分针对。但是因为我们一开始的目标只是力争区内出线，因此我们暂时没有过多考虑这一组的算法应该如何应对。

在正式的小组赛中，我们组的算法战胜了两个“智能”的强敌 Charlie 和 papa，但是因为我们的参数设定恰好又被暴力打角所克制，因此我们以负 lima 一场的成绩成为了小组第一并出线。在八进四的比赛中，我们也是在第二局修改了两个参数就实现了 17-23 到 40-0 的逆转；第三局卓老哥主动背锅，说要提交他之前没有舍弃的一个算法，大家也给予了充分的信任，果然这个陪练算法也没有让大家失望！当时我们就念了两句诗，叫做“苟利 GOLF 生死以，岂因祸福趋避之”，就这样，我们开始着手四强赛。

4.4 第四阶段——再次被 Oscar 支配的恐惧

随着小组赛结束，我们都互相了解了大家在匿名热身赛中的名称，所以当我们发现之前完胜我们的劲敌 sabac 就是我们将要面对的 Oscar 时，我们感受到了一种深深的无力感。好在我们尚有时间再次进行修改！

在我们两个组的线下交流中，他们透露了自己采用了深搜的算法；从这，我们组获得了启发，开发了新的打法。在这一版打法中，我们完全推翻了之前利用 ds 的算法，因为我们与其去研究对手过去的打法，不如向后考虑一步，研究一下局面未来的走势。同时，我们针对跑位也进行了修改，我们考虑了对手打上角打下角对我们影响一样大的一个微妙的位置。就这样，我们在短短一段时间里开发了完全不同的一套算法，组内对战显示这套代码已经能够战胜之前的所有版本。

与此同时，Oscar 队将它们在小组赛中的旧版代码无私地赠与了我们，但是我们恪守着不能抄袭的原则，只用这个代码进行对战并且观看结果。一开始的战绩不尽人意，但是我们通过观察对手在残局时的处理方式，归纳出了一些规律，并把这些规律写到了自己的代码中，并不断调试自己的参数。同时，令人激动的是，我们的开局跑位和 Oscar 变成了一样的“神奇跑位”。终于，在四强赛当天的早上四点，我们实现了零的突破，我们实现了从负到胜的突破！我们最终版的打法在与老版 Oscar 的对战中，实现了 21-19 的险胜。也正因我们在此过程中付出了很多的心血，取得了巨大的成果，所以我们在四强赛中，没有输的太惨，成为了第一个胜 Oscar 十局以上的队伍！也正因如此，我们在三四名的争夺中以巨大的优势战胜了 Alpha 队！

5 实习过程总结

5.1 分工与合作

林荣：（一流程序员）查看组内/组间对战结果及复盘，分析胜负原因，参与部分 DEBUG。
王悦翔：（顶级程序员）代码编写的主力，编写了一些陪练算法，最终完善了与道具有关的算法。
张卓：（一流程序员）代码编写的主力，对打角策略进行了彻底的分析，开发了启用 ds 的算法。
李宇轩：代码编写的主力，开发了 “均衡跑位”算法，应用数学工具做了一些分析和计算
曾挺：参与编写代码，主要工作是对道具系统的分析，以及与道具有关算法的编写。
胡宇城：（一流程序员）作会议记录、撰写报告。
隋清宇：（一流程序员）参与算法的思考，提出对游戏本身及我组各阶段算法的见解。
（注：本实习报告由组员共同完成，存在肉眼可见的文风和表达方式差异，但大体上不影响阅读。具体情况以程序代码及其注释为准。）

5.2 经验与教训

整体的纪律性不太好，经常会出现我这里的和你那里的同名算法文件内容不一样导致信息误导的情况。代码规范也是需要改进的，比如注释和空格等，组里的某位李姓同学（手动滑稽）虽然编写的算法都十分的强力，然而其他人并不能读懂.....which is 坠痛苦的。

5.3 建议与设想

实际上，根据我们的了解，很多组在考虑道具的时候，隐身这一个道具都给了很低的价值，大多数是给了 0，而我们组更是把它的权重改成了-1000，这完全是为了避免捡到这个没用的道具。而旋转球一开始的设定也注定了它必被修改的命运。如果有可能，希望今后的道具能够多考虑一下。

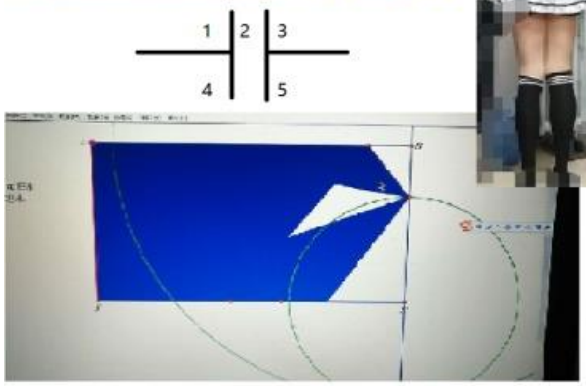
竞赛的赛制可以修改，因为我所看的电竞比赛多为 16 进 8 采取分 4 组，组内双败 BO1 赛制出线两人，8 强之后两两之间进行 BO5 的比赛。所以第一次看到这个规则还是略有不适应。

6 组会照片以及不可描述

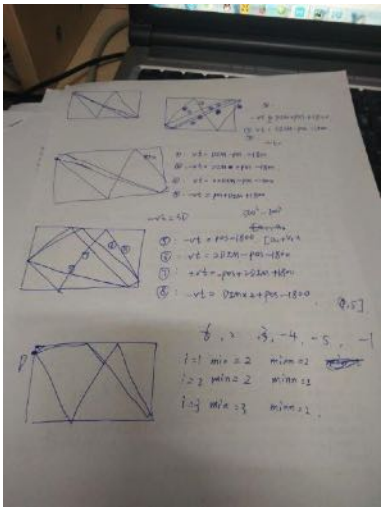












6



7



8



图一：四强赛当天三点的两位肝帝
图二：我组顶级程序员
图三：我组视频会议照片
图四：李宇轩的分析
图五：组会照片
图六：张卓的加速度分析
图七：可爱的康娜
图八：可爱的康娜
图九：可爱的康娜（左图）

7 致谢

感谢陈斌老师，感谢助教团，感谢黄岭贝，感谢陶天阳组和郭浩（地空）组（在四强赛前提供他们的算法给我们陪练），感谢吴宜谦组，感谢裴召文组（一直和我们陪练），感谢游卡桌游的三国杀，感谢麻辣香锅，感谢学一食堂铁板烧，感谢发丝缘假发的女装。

8 参考文献（滑稽，不存在的）

《他改变了中国：江泽民传》——（美）罗伯特·劳伦斯·库恩
《怒斥港记》
《视察二院》
《谈笑风生》
《演奏 Hawaii Guitar》
《鸡年新春》



ALPHA

组长：常啸寅

组员：戴琪，黄岭贝，滕沅建，陶韵竹，王瑞敏，张颢丹

小组赛战绩：S1

决赛战绩：四强









Alpha—“Go”

Alpha 组实习作业报告

常啸寅* 戴琪 黄岭贝 滕沅建 陶韵竹 王瑞敏 张颢丹

摘要：

本组主要采用动态改变打角跑位模式，利用 tree 类简单预测最大价值的原理编写算法，利用列表、字典、自定义 tree 类等数据结构实现，希望根据对方的策略临时改变道具、跑位等参数实现动态调整算法。在这样的思想指导下，本组在热身赛中胜 157 场负 28 场，胜率约为 85%。在正式比赛中以南区小组第一名成绩出线，最终成为第四名。

关键字：

动态循环克制 极大极小值算法 列表 字典 自定义 tree 类

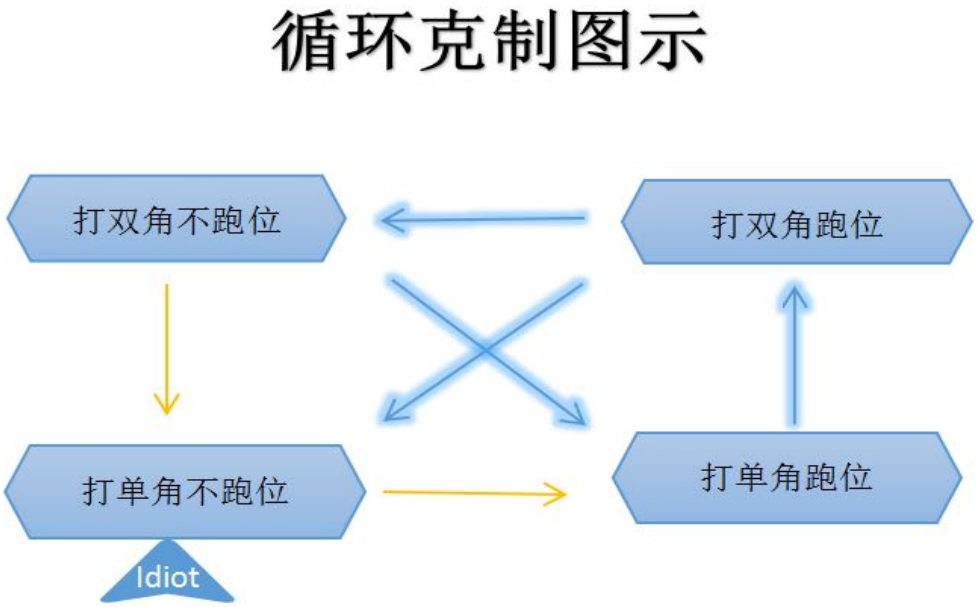
1 算法思想

1.1 总体思路

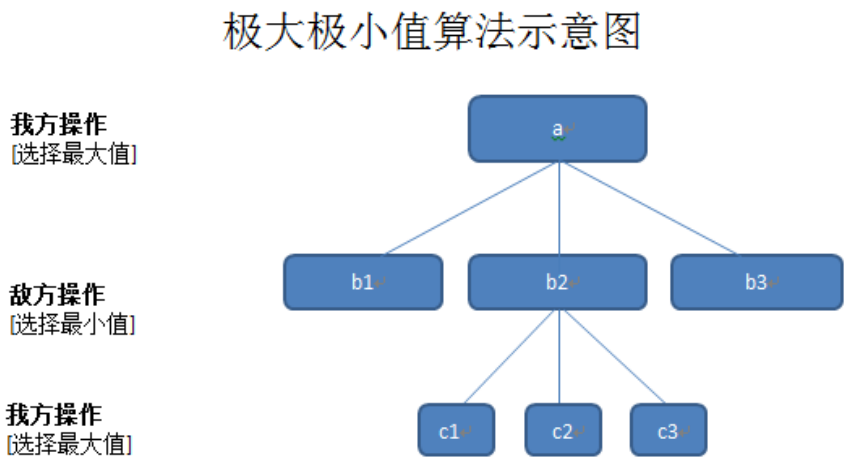
1、总体思路：

动态改变打角跑位模式，简单预测最大价值的操作。

一、把打角模式总结为打双角（交替打对方的两个角）和打单角（一直打对方的某一个角）两种，把跑位模式总结为不跑位和居中跑位两种，并两两结合形成四个基本的打角跑位模式，总结出其中的相互克制关系（见下图）。在比赛过程中动态探测对方的打角跑位模式，并依情况改变自己的模式来克制对方的模式。诸如探测对方的模式时的一些判断、居中跑位的具体跑位等细节需要根据测试结果来优化。



二、利用树的数据结构实现极大极小值算法，树的节点存储不同操作，我方选择价值最大的操作。道具的价值设为定值，在进行极大极小值算法时参与计算，用以判断是否吃道具。（见下图）



2、主要数据结构与算法：列表、字典、自定义的树 tree 类。

3、算法进化过程：

初期总结出打边角打法是使对方消耗最大的打法，预测交替打两个边角是最佳打法，采用了打双角打法。根据总结出的打双角打法，采用了居中跑位的打法来应付对方的打双角打法。未涉及有关道具的打法。

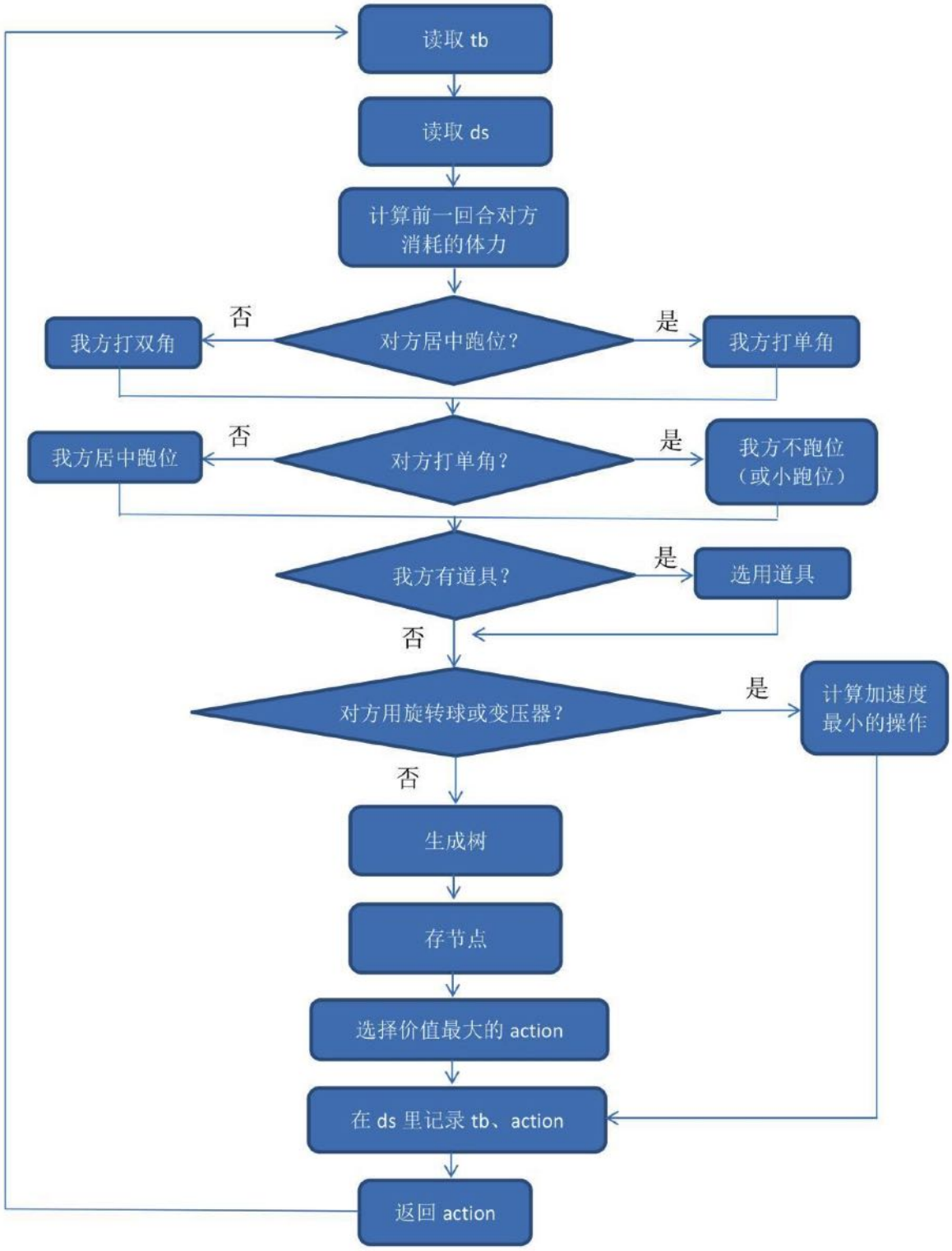
随后产生预测的想法，尝试并确定使用树的数据结构实现短期预测即极大极小算法。

热身赛后，总结出四个基本的打角模式和相互克制关系，实现了动态探测对方模式及改变我方的模式，改变了应对变压器、旋转球的具体方法，改变个别道具的价值的参数。

初赛进行后，认识到高估了道具的价值，于是降低道具的参数，同时发现对手常使用打单角模式，于是将居中跑位的具体跑位降低。

决赛前，给探测对方模式的判断中增加灵活性，思考了判断对方模式时的参数范围，优化了探测对方算法的算法。此时的算法已经是最终算法，并可以根据对方的策略临时改变道具的参数、居中跑位的参数、对对方模式判断的参数等等来调整算法。

1.2 算法流程图


读取 tb
读取 ds
计算前一回合对方
消耗的体力
对方居中跑位？
否
我方打双角
是
我方打单角
对方打单角？
否
我方居中跑位
是
我方不跑位
（或小跑位）
我方有道具？
是
选用道具
否
对方用旋转球或变压器？
是
计算加速度
最小的操作
否
生成树
存节点
选择价值最大的 action
在 ds 里记录 tb、action
返回 action

1.3 算法运行时间复杂度分析

我们的算法依靠人脑总结的规律，没有大量的搜索和计算，但是包含大量的判断，所以复杂度不高，总体算法复杂度为 O(1) – O(n). 生成和利用树时，需要遍历场上的每一个道具的每一个可行操作。树的节点（叶节点）由打两个边角和每个道具的所有可行操作组成，节点数记为 n，每个点对应 2-3 个可行操作，所以算法复杂度为 2n – 3n. 用大 O 数量级估计，算法复杂度就是 O(n). 考虑到场上道具数量有限而且偏小，所以算法复杂度接近 O(1). 算法运行时间的主要构成就是生成和利用树的过程。

实际运行中，一轮自己对打的比赛耗时 0.1 – 0.2 秒。

2 程序代码说明

2.1 数据结构说明

算法中主要使用了列表与字典，并使用了自定义的 tree 类。

列表用于储存如目标点、树节点、可选的操作等信息；字典则用于储存道具与其对应的价值，以及 ds 中的信息；自定义的 tree 类用于极大极小值算法的实现。

2.2 函数说明

一. 主要函数：

delife 函数：根据操作计算消耗的生命值；

fly 函数：根据操作计算对应的落点；

eatcard 函数：根据道具位置计算打道具需要的加速度；

target 函数：根据落点计算需要的加速度；

count_col 函数：根据球的位置与速度、加速度计算球撞墙的次数

opplay 函数：用于对对方操作的预测，根据 ds 判断出的对方打角与跑位类型确定对方的打法，参数为球拍与球的状态、使用的道具等信息，返回在对方策略下的最优操作

play 函数（参数：桌面状态与 ds 数据）：具体实现见基本思路

二、算法策略：

1、基本思路：首先读入各种数据。如我方的状态（位置、拥有的道具、生命值）、对方状态（位置、拥有的道具、生命值、使用的道具等）、球的状态（位置、速度）、场上道具状态（计时）。然后读 ds，对对方的策略进行判断。ds 中记录了每回合的球桌信息及我方的动作，我方根据前一回合中对方消耗的体力，和对方如果居中跑位会消耗多少体力的差值判断对方跑位情况，二者的差小于一定值时则认为对方是居中跑位，否则是不跑位；此外，

根据两回合球的落点也能判断对方打角的方式是打单角还是打双角。根据对方打角方式确定跑位，打双角则居中跑位，打单角则不跑位（或四分之一、五分之一跑位）

2、道具使用：

a.我方道具的使用：如果有道具就使用最顶端的那个道具，跑位距离过小则不使用瞬移术。（注：我们设定的道具价值在不断降低，因为发现捡道具经常造成对方接球很容易，并且如果对方有应对道具的方法，它能给对方造成的伤害也不大。所以最后我们就是加血包与减血包价值设为 3000，变压器 400，瞬移术 300，旋转球只有 100。）

b.对方使用道具的应对：如果对方使用了变压器或旋转球，则返回加速度最小的操作，防止过多的体力损失

3、选择打球的目标点：只有两个角以及能打道具的点。对每个点求出对应的加速度以及操作，存入列表。创建存储树节点的列表，将每个有效的操作转为树节点，最后根据树节点得出价值最大的操作并返回。（返回之前都要记录 ds 数据）

4、一些细节的判断：如体力值过小时只顺路吃道具；体力小到一定程度后跑位至少要保证能在对角接到球；考虑到对方可能跑位三分之一、四分之一等中间数值，我们在判定对方的跑位方式时会结合不同的打角方式来确定更有利的判断。

2.3 程序限制

1、搜索深度和广度都不大。

2、算法不完备，存在弱点

只根据前一次对方的动作来判断策略并不可靠，常会出现判断错误，造成不跑位而迎球时跑到对角，或者对方不跑位或少跑位而我们打回原角的情况。并且没有对有道具的回合进行单独判断。

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) i5-3210M CPU@2.50GHz
  （内存：4.00 GB）
- 操作系统：Windows 7
- Python 版本：3.5.3

具体测试办法：

我们的算法是一个不断完善的过程，我们通过人工的演推不断寻找需要完善和补充的地方，然后将进行了改进的算法与原有的算法进行比赛来进行测试，基本每一次的改进都能保证战胜之前的算法。

我们的算法大致经历了以下几个过程：仅有最基本的跑位和吃道具——选择

最佳点进行击球——完善对道具的获取与使用，加入道具的价值——加入几种相互克制的跑位策略，利用 ds 对策略进行选择使用。所有之前的历史算法都曾经充当过我们新改进算法的陪练并且被击败。

关于热身赛的结果，我们胜 157 场，负 28 场，总体胜率大约为 85%。下面为部分热身赛结果数据：

对手	T_sabac	T_excited	T_lkjhg	T_Apollo	T_track	T_AVCNA	T_test	T_eighth	T_RABBIT	T_harry
对方胜场	9	7	5	2	2	1	1	1	0	0
我方胜场	1	3	5	8	8	9	9	9	10	10
我方胜率	10%	30%	50%	80%	80%	90%	90%	90%	100%	100%
对手	T_rsbgd	T_Hellofine	T_King	T_learn	T_zxcvb	T_unique	T_SWIFT	T_yzzxx	T_simple	T_David
对方胜场	0	0	0	0	0	0	0	0	0	0
我方胜场	10	10	10	10	10	10	10	10	10	10
我方胜率	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%

3.2 结果分析

一、算法在对面使用旋转球和变压器时采取尽量原速打回的策略，这样让己方的能量消耗尽可能降到最小。

二、在面对对面打单角和打双角时采取不同的跑位策略，对面打单角采取四分之一跑位，对面打双角采取居中跑位，这样从一定程度上减少了跑位的消耗。

三、通过对面体力消耗推断对面跑位方式，面对对面小跑位采用打双角，面对对面大跑位采用打单角，这样从一定程度上减少了加速的消耗。

四、所有对于战术策略上的变化都运用 ds 进行分析和操作，这样使得战术变得更具灵活性，可以应对更多不同的战术策略。

五、在进行算法的分析面对体力消耗的比较时采用了博弈树，更加方便地寻找到最少的消耗。

点评：我们的算法效率很高，因为时间复杂度并不高，也没有用到深度搜索等需要耗费大量时间的功能，算法在运行时间上完全在我们预料之中，十分的快速，其中大部分的时间都用在对情况进行判断上，然后就采取我们预先设计好的相应对策，不需要多余的步骤。

3.3 经典战局

在小组赛八进四的比赛中，我们被对手先胜一局。在对复盘数据的分析过程中，我们发现我们的加速消耗体力远远高于对手，因此我们将算法中针对对方四分之一跑位的策略由打双角转变为打单角，从而大大减少了我们自身的消耗，并且在接下来的比赛中连胜两局，从而反败为胜挺进四强。

我们能够在绝境之中反败为胜，很重要的一点就是我们的算法具有足够大的灵活性，可以通过改变判断条件来对自己在不同情况下采取的策略进行更改，从而能够更有针对性的应对对手。

4 实习过程总结

4.1 分工与合作

一、小组分工

姓名	常啸寅*	戴琪	黄岭贝	滕沅建	陶韵竹	王瑞敏	张颢丹
分工	1 思考算法 2 编写部分代码	1 思考算法 2 分析复盘数据 3 敌情侦查员	1 思考算法 2 编写部分代码 3 数算报告	1 思考算法 2 为代码作注释 3 数算报告	1 思考算法 2 数算报告	1 思考算法 2 数算报告	1 思考算法 2 核心代码人员

二、小组大事件时间轴（历次组会记录及照片）

5.11

一个“你们羡慕不来的男女比例”队横空出世！

5.20

第一次组会顺利召开！

熟悉游戏规则，讨论大致思路，确定实现基本功能函数，小组分工



5.20 —— 5.25

1、实现了居中跑位、选点打、最简单的吃道具（有道具就吃）等功能

2、以双方消耗的体力差作为判断标准，使用极大极小值算法，构造了博弈树

3、选头像

4、思考讨论了 ds 的可能利用方法



5.26

第二次组会在静园草坪顺利召开！





1、研究道具使用策略
2、讨论跑位情况
3、观察复盘数据，调整算法
4、讨论 ds 的使用
5、讨论如何预测对方打法

5.27 —— 5.29
1、完善吃道具策略：给每个道具一定价值，加入体力值的计算与比较中
2、利用 ds 记录了每一回合的桌面信息及我方操作

参加热身赛！

5.29

第三次组会顺利召开！





对四种打角与跑位方式进行了分析（打双角跑位、打双角不跑位、打单角跑位、打单角不跑位），得出相互克制的关系，决定根据 ds 判断对方策略，再使用相应的能克制对方的策略
（图为组员分析复盘数据和张颢丹的帅照）

组会结束后，“肯教”欢乐聚餐！

赛前细节优化，参数调整，私下约友谊赛

6.5

比赛开始！
分区赛小组第一成绩出线

分区赛（南区：South）



积分	INDIA [12]	BRAVO [6]	QUEBEC [4]	SIERRA [7]	NOVEMBER [0]	ALPHA [15]
INDIA	INDIA	0	0	0	0	3
BRAVO	3	BRAVO	0	3	0	3
QUEBEC	3	3	QUEBEC	1	0	3
SIERRA	3	0	1	SIERRA	0	3
NOVEMBER	3	3	3	3	NOVEMBER	3
ALPHA	0	0	0	0	0	ALPHA

四强争夺赛！
在第一局失利的情况下，小组成员紧急修改代码，终连赢两局成功翻盘！



6.8

决赛止步四强

4.2 经验与教训

在实习过程中，大家相处的很融洽，都在为大作业作出自己的努力，组会上讨论十分热烈，但仍存在组会效率太低，分工有些不均，核心算法的实现主要靠一位同学完成等问题。后期如果还有机会，希望能发挥出每个组员的能动性，使得实习配合更加高效完美。

4.3 建议与设想

一点点小建议：
1、希望能实现更具观赏性的可视化战局
2、在比赛中更多的增加对时间的考量，即对算法复杂度进行控制
3、可将基础实施代码设置为加分项，由学生完成
4、希望赛制变成循环赛制，即 N*N 对打积分模式

5 致谢

1、感谢组内张颢丹同学承担了核心代码编写工作，并创建可视化战局
2、感谢冀锐助教提供热身赛平台
3、感谢王旭斌、杨子珍、吴宜谦同学所在小组与本组进行友谊赛

6 参考文献

【1】数据结构与算法：C 语言描述 （美）维斯 北京：机械工业出版社 2004.1



CHARLIE

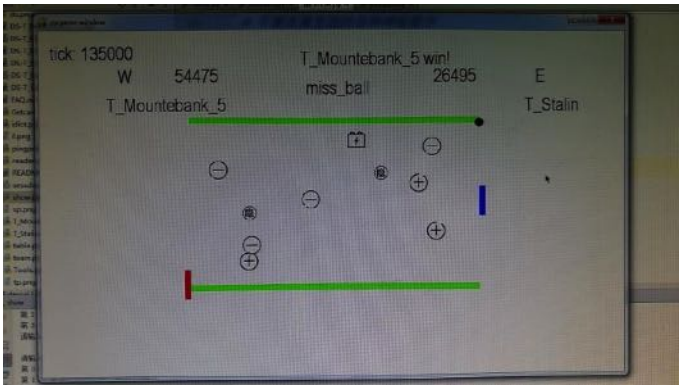
组长：郭浩（地空）

组员：郭晴宇，哈杰，罗哲楷，孙景南，王泽鑫，杨德鼎

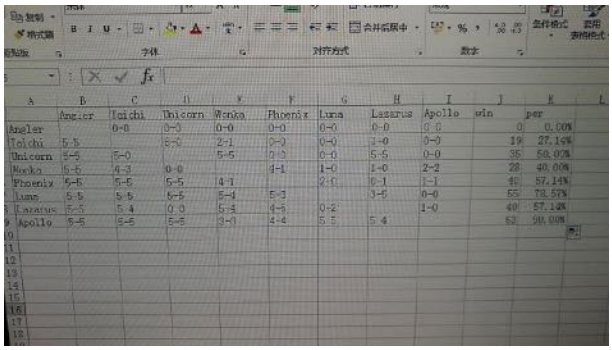
小组赛战绩：W2

决赛战绩：八强




tick 135000
T_Mountebank_5 win!
W
54475
miss_ball
26495
E
T_Mountebank_5
T_Stalin





Charlie



数据结构与算法课程实习作业报告

Charlie 组

（郭浩*，郭晴宇，罗哲楷，孙景南，杨德鼎，王泽鑫，哈杰）

摘要：

Charlie 组的漂移乒乓算法，以估值搜索为主体，并加入了对对方算法模式的判别、多种状态的切换，基于历史数据的对方落点估计等元素。Python 的列表和字典数据结构在程序中发挥了关键的作用。最终，该算法在 26 个组中取得八强的战绩。

关键字：

漂移乒乓，python dict，估值函数，模式判别

1 算法思想

1.1 总体思路

Charlie 组算法是一个以估值搜索策略为主体，根据对方打法和当前局势在各模式间切换的决策算法。整个程序按照结构化程序设计理念分六个模块完成，这六个模块是最优化估值搜索算法、模式分析、状态转换、逆推型跑位算法、道具系统、函数库。其中模式分析模块与函数库原本作为独立的 py 文件被调用；状态切换、估值搜索算法、学习型跑位算法构成主程序；道具系统则分散在各处。

本算法在数据结构方面的特色是设计了历史数据存储结构 ds 的框架，实现对历史数据的全面存储（因此 DS-Charlie 可能是各组 DS 中最大的一个），其中的部分数据的确起到了辅助决策的作用。相关代码体现了对 python list 和 dict 两种数据结构的灵活运用。

与一些具有鲜明的单一思路的算法不同，本算法更像是多种思考、多个招数的综合。最终程序的结构是集思广益、逐步完善的结果。估值搜索算法、函数库、道具系统是最初建立的基础设施；其余则是在之前的基础上添加的。

对战游戏 AI 的决策问题，可以化归为一个反映游戏特征的估值函数的最优解问题。因此，在最为关键的加速度确定上，估值搜索便成为我们的第一个正式思路。由于加速度要经过取整处理，加上碰壁次数的限制，解空间只有不到 4000 个加速度值；对每个加速度，对方跑位加迎球的距离是可以确定的，双方的加速代价也是可以确定的，可以以生命损耗为标准确定朴素估值函数；所以这一想法是充分可行的。并且假如参数调整，单纯的打角算法可能得不偿失，估值算法则仍有较好的表现。但我组的估值仅停留在一层贪心，原因在于我们经讨论后认为这一游戏不同于棋类游戏，对对方策略和道具因素的依赖性较大，增加深度可能偏离实际。

估值搜索算法完成后，我们发现道具系统可按照改变参数的方式与估值算法兼容，即遍历各加速度时，如果拿到道具，就在估值函数中加上相应的价值；如果使用变压器或旋转球，可以在估计对方伤害时加倍计算；检测到对方使用旋转球后以单独模式应对，可以避免 invalid bounce. 于是道具系统加入主算法。与此同时，函数库的开发也在进行。当两者均完成并对接后，因体力不支 miss ball 成为我组算法失利的唯一原因。

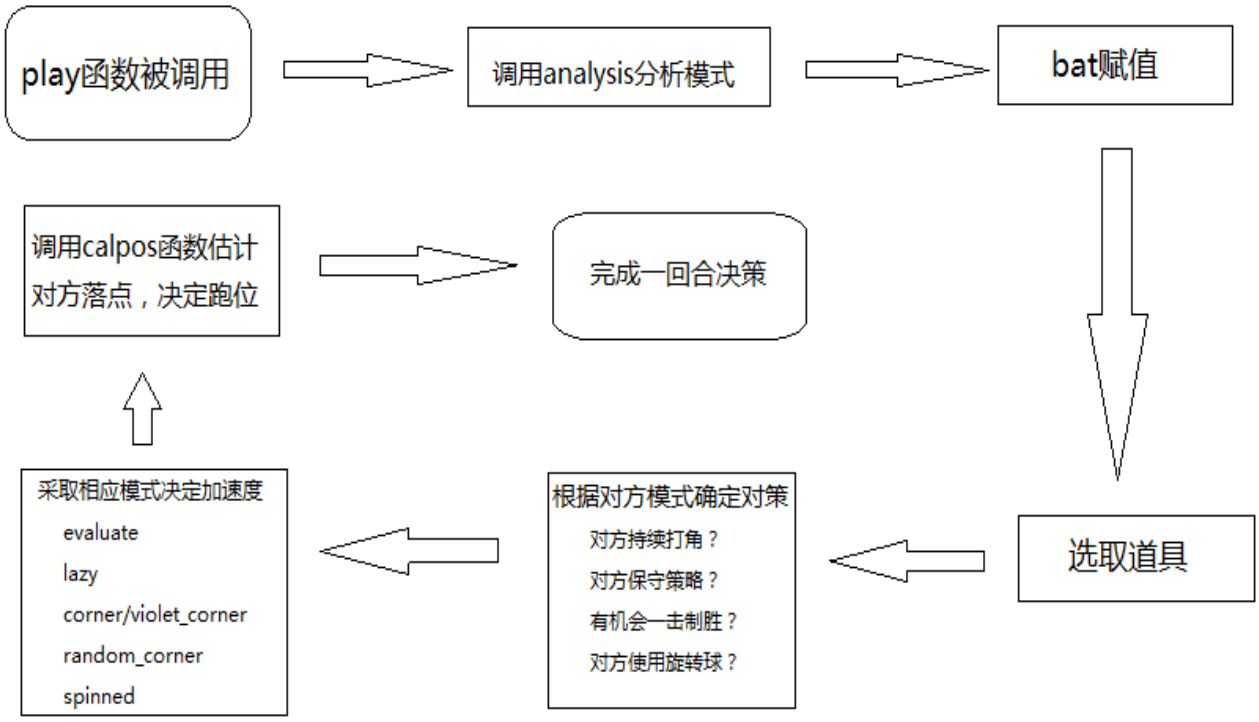
测试中我们发现对不同的模式有着不同的有利对策，即存在斗兽棋中“老鼠吃大象”的循环现象。如果能识别对方并采取有利的对策，就可以面对不同对手总保持在对方上游。"基于历史数据的模式分析"这一想法由郭晴宇提出并编写代码实现。为实现这一想法，我们建立了 ds 的存储结构。紧接着应对不同模式采用不同状态的状态转换模块也就自然地加入了主程序。

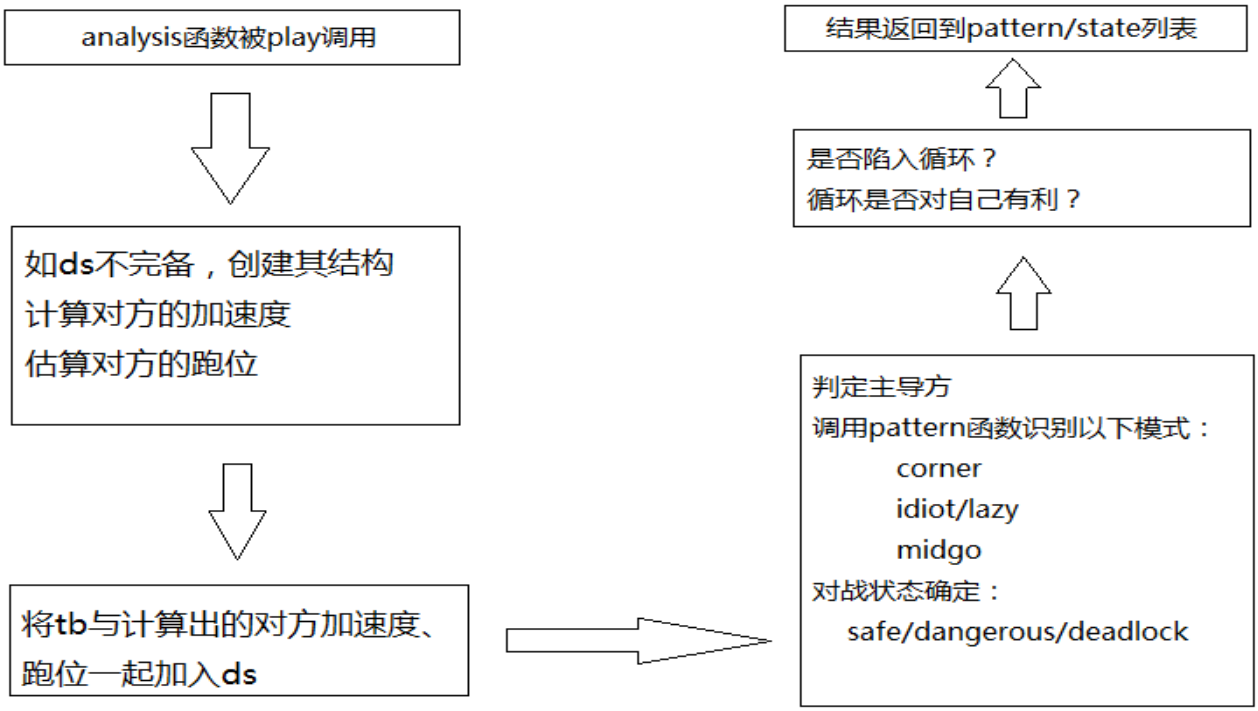
热身赛中我们又观察到不少算法以跑位少取胜，相比之下我们的中点跑位或远端角中值跑位（跑到当前位置与距自己较远的角的中点）并不占优势。杨德鼎提出用统计方法估计对方落点，这一思路得到大家的认可。于是这一部分最终加入了主程序，Charlie 算法成型。

比赛前，有人曾这样评论漂移乒乓大作业：“参加的算法可归为三类：正常算法克随机算法（随机因素较大，非完全随机）；学习算法克正常算法；随机算法克学习算法。因此获胜的关键在于正确估计各种算法的数目。”从这个角度上看，Charlie 的主体是正常的估值算法；跑位则能够根据对方打法调整，有一定的学习性；状态转换又使得它具有一定的随机性，不易被其它学习算法把握。可以说，Charlie 是这三种思路的有机融合。

1.2 算法流程图

T_Charlie.py 算法总流程图





analysis 函数算法流程图

1.3 算法运行时间复杂度分析

在 play 函数中，可能花费大量时间的有估值搜索、analysis 函数和估计跑位的 Calpos 函数。

估值搜索对于解空间的加速度总数 n，时间复杂度是 O(n)的。这一部分对近 4000 个加速度值中的每一个都进行至少 4 次函数调用（如计算落点，计算消耗等），每个函数进行约 10 次四则运算，共计约几十万次四则运算，是耗时的主要来源。

analysis 函数的时间消耗体现在对已进行的 k 个回合的遍历，时间复杂度是 O(k)的。k 一般只有百次量级，故对总时间影响不大。Calpos 函数的时间消耗体现在对列表 T 中的 m 个回合的遍历，时间复杂度是 O(m)的。T 包括当前对局数据和之前与同一对手对战的部分数据，但总数据量限定在 1000 以内。因此此部分的时间开销也不大。

综合来看，估值搜索部分是运行时间的主要部分。实际运行中 Charlie 一般在 1-2 秒内完成一局。

2 程序代码说明

2.1 数据结构说明

算法除使用 python 提供的列表、字典外没有使用其它特殊数据结构和自定义类。

列表和字典的使用则是无处不在的。例如：参数部分 DEST_LIST 用列表存储打角落点，便于 corner_least_acc 等函数遍历；CARD_VALUE 用字典存储各道具的估价，在估值部分调用；analysis 函数的返回值是 res 字典，其中的 value 值为两个列表。更重要的是 ds 被维护为一个多级字典，包括 current、history 两个键值，其内部又有 pattern，data，action 等条目，每条目又是一个列表。这种列表的多级字典形式与主程序接近，项目清晰、有条理，有利于调用和分析。

2.2 函数说明

根据 Charlie 的结构化编程特征，分模块介绍各函数的接口、功能和采用的算法策略。

2.2.1 函数库

在 Charlie 的代码中主要用到函数库中以下函数：

1. knock(posy,vy)

此函数功能为计算从 posy 出发，以 vy 速度的撞墙次数。在此函数中先考虑球在没有墙壁的情况下的落点，再根据镜面对称的原理计算碰壁次数。在计算时要考虑球落在墙角与落在中间两种情况，落在墙角时需要考虑是否遗漏最后一次碰墙。

2. dest_y(posy,vy)

此函数功能为计算从 posy 出发，以 vy 速度击出后的落点纵坐标。原理与前一个函数相同，分落在墙角与落在中间两种情况计算实际落点位置。

3. res_v(posy,vy)

此函数功能为计算从 posy 出发，以 vy 速度击出后在落点处速度。原理与第一个函数类似，根据球在运动过程中碰墙次数判断球的纵向速度的改变次数，计算球到达对方边界时的速度。

4. get_card(posy ,vy, card, direc ,eps=2000)

此函数功能为判断在 direc 位置（“W"或”E“）从 posy 出发，以 vy 速度发球能否得到道具 card。在此函数中会计算出道具“card”与球的运动路线的距离，若这个距离在误差“eps”以内，则可以捡到此道具，否则不能。

5. loss_acc(acc)

此函数功能为计算加速度 acc 扣血值，用于估值。计算方法为规则中的扣血方法。

6. loss_run(run)

此函数功能为计算跑位 run 扣血值，用于估值。计算方法为规则中的扣血方法。

7. legal_least_acc(posy,vy)

此函数功能为计算合法击出的最小加速度，用于己方策略的判断与对对方的估值。posy 为己方位置，vy 为当前球速。计算时分两种情况，若不改变加速度直接让球反弹时球的运动就是合法的，那么最小加速度为 0；若直接反弹时球的运动违法，则需改变球的速度，不难看出，采用与反弹速度最接近的一个打角速度时加速度最小，返回此加速度即可。

8. corner_least_acc(posy,vy)

此函数功能为计算打角的最小加速度，用于己方策略的判断和对对方的估值。posy 为己方位置，vy 为当前球速。此函数会遍历出从此位置打角的所有落点，从中选择加速度绝对值最小的一个作为返回值。

9. corner_violet_acc(posy,vy,side)

此函数功能为计算将球打到远离对方上次接球位置的角所需的加速度，即暴力打角。posy 为己方位置，vy 为当前球速，side 为“up”或“down”。此函数可用于根据对方的位置进行打角，如果对方在上一局中接球的位置位于中点上方，则打下方的角落；若接球的位置位于中点下方，则打上方的角落。

10. damage(desty,op_vy,op_posy,mode=None,acc_param=1,bat_param=1)

此函数主要功能为计算落点为 desty，落到对方时球速 op_vy，对方（跑位前）位置为 posy 的对方生命值损耗；如果对方为 corner 模式，mode 传入“corner”，则按照对方会打到角上的假设计算加速损耗；根据道具的功能来计算使用道具时的生命损耗，如果己方使用变压器，bat_param 为 2；如果己方使用旋转球，acc_param 为 2。用于估值。

2.2.2 模式分析

这一模块由三部分组成，第一部分是对 ds 数据的维护，包括其中数据的反推，第二部分是对于对方的行为模式的判别，第三部分是对场上局势的分析。代码中体现为主函数 **analysis(tb,ds)**，以及它所调用的 pattern，similar 等函数。

analysis 函数首先是对 ds 的维护和数据的反推，这不得不提到我们的 ds 的结构是怎样的。为了有条理地保存大量数据，我们将字典层层嵌套，最后一层作为列表，来存放数据。首先 ds 有两个 key，一个是'current'，另一个是'history'，分别用来保存当前局数据和历史局数据。'current'再映射到一个字典，key 分别为'data'，'action'和'pattern'。‘data’映射到一个列表，按照序号索引一个 TableData 类。这样做是非常方便的，因为我们只需要在每回合开头将 tb 添加到这个列表末尾就可以保存数据。同理，‘action’是一个 RacketAction 的列表，是每局己方的行动。而‘pattern’也是一个列表，表示对方的行为模式，存放的是一些

字符串。这个列表每次分析模式时进行修改。ds['history']也是一个字典，里面通过对手的名字来映射到一个和 ds['current']类似的字典。这个字典会在每次 summarize 的时候进行维护。如果数据量过于庞大，则会删除掉最旧的一些数据，防止数据越来越多导致对局越来越慢。然后是对于数据的反推。我们尝试了对对方的加速和跑位的反推。加速是可以精确地计算出来的，因为对方的接到球的速度是可以用上一局的 action 推算的，而对方击出球的速度也是可以推算的。但是跑位是没有办法精确计算的，主要是因为生命值的计算中使用了地板除法。因此只能进行对计算的结果进行猜测，给出一个最有可能的结果。对于对方的加速和跑位的反推对于己方的策略是有一定影响的。

然后是对于对方行为模式的判别。这一部分通过 analysis 函数调用 **pattern(ds)**函数完成。通过我们的初步讨论和自己想出的一般策略，我们将模式分为'idiot','lazy','corner','midgo'。这样分其实是非常粗糙的，但是也没有想出能够更加精确判定的模式类型。'idiot'表示对方不执行跑位操作。这个判别并没有什么用处，当时只是为了验证模式判别的可行性而进行的尝试。最终结果表明这个判别是有效的。'lazy'表示在加速时通过消耗最小的体力值来进行击出。'corner'表示对方是打角的模式。'midgo'表示对方每次击球后会跑位到中点。对于对方的行为模式的判别的主要作用在于，如果对上了一些行为固定的对手，就可以采取针对性的策略来进行反击。'corner'的判别是简单地判断对方最近十个回合击球位置是否都在角落。'lazy'也是类似的，判断对方最近十个回合是否都以最小合法加速度击出。而'idiot'和'midgo'则稍微复杂一点，需要用到生命值的计算。假设对方采取了'idiot'或'midgo'的策略，然后计算对方的生命值损耗，如果和实际相符，那么可以判断对方采取了对应的策略。

最后是对于场上形势的分析。这一部分主要是判断是否己方占优，是否陷入了不好的局面或者进入了坏的循环。一旦进入到坏的局面或者坏循环，就需要采取不同的策略来打破僵局。这一部分主要将场上局势分为'domin', 'deadlock', 'safe', 'dangerous', 'good', 'bad', 'repeat', 'good_repeat', 'bad_repeat' 几个部分。'domin'表示当前场上占优的一方。'deadlock'表示双方处于僵持状态。'safe'和'dangerous'分别表示己方在生命值上领先很多或落后很多。'good'和'bad'表示如果持续这种打法是好的还是坏的。'repeat'表示场上进入循环状态。'good_repeat'表示对己方有利的循环。'bad_repeat'表示对己方不利的循环。这一部分的实现是通过对 ds 中数据的迭代，将历史数据与当前数据进行比较来进行判断。如果有一局的场上的所有数据与这一局都非常相似，那么可以判定为进入循环。对于场上形势的分析其实是比模式判别更加重要的一个环节，其中最核心的部分就是对于循环的判定。如果进入了不利循环，那么己方一定是要去打破的，这可以通过改变击球策略来实现。如果不去打破不利循环，那么己方输球可以说是一定的。而其他的一些分析，实际上并没有得到应用，这是因为这些分析数据没有什么针对性，难以根据这些数据来确定一个比较好的策略，这可以说是比较遗憾的。

2.2.3 状态转换、估值、其它策略

这一部分是 play 函数中的加速度决策部分。在此之前，play 函数还决定了道具的使用，方法是只要有道具一定会用，加血包和减血包优先。

对于状态转换部分，我们自己想出了一些击球策略，并在此基础上判断对方的击球策略来选择我们自己的策略，以求得到针对性的效果。

针对打角算法（corner），即“最大伤害对方”的策略，我们起初采用了“lazy”，即消耗最小的回球，但在热身赛中表现不佳，回放中发现因为 lazy 多次出现了非常坏的击球——几乎回到对方的板上，造成对方跑位消耗很少，而我方被打得满场跑。在之后的修改与调试中，我们认为游戏的参数对攻击型算法更有利，一味防守只会导致被动局面，因此决定采

用更具有攻击性的方法，多次修改后，确定为最终的估值算法与反打角相结合的策略。

针对对方采用最小消耗回球策略（lazy），经过测试发现，打角对其极为有效，这也促使我们放弃以 lazy 应对打角的策略。

而面对具有一定智能性的选手，无法简单的判断对方是什么策略，因此以我方主要的策略——估值算法作为主要的应对手段。通过热身赛我们发现，打角仍然具有非常好的效果，在后续的修改与测试中发现随机打角也是一个能对对方造成很大杀伤的策略，因此我方加入了这两个方式，用以辅助估值算法，以达到出其不意的效果。

在前期的测试中我们发现，为了寻求最优解，双方总是会陷入某种循环，而道具也不足以打破这种循环，而在循环中不利的一方虽然有些回合占优，但整体吃亏，但若只采用估值算法，因为每回合都寻求最优解，难以跳出此不利循环，在循环中消耗而死。因此我们加入了针对不利循环的策略，若判断我方陷入了不利循环，则采用之前未采用的击球策略以跳出循环，lazy 在此处发挥了很大的作用。

前期我们并未考虑打死角的问题，我们的策略一直相对保守，但通过热身赛发现由于死角的存在，某些我们始终占优的局面意外输掉；而且如果对方不采用中点跑位，就会在生命值仍有 30000 甚至 40000 时暴露出死角，这时打死角也可以作为一种逆境下的翻盘策略，因此我们先对对方的跑位策略作出判断，若对方并不采用中值跑位，且存在可能的死角，则我方暴力打远角，尽量打到对方死角，造成一击制胜。

若对方使用了旋转球，则我方以消耗最小的合法加速度击回去，主要是要给加速度加倍，避免出现 invalid bounce。

在加速度的确定部分，我们主要有估值算法，随机打角，最小消耗打角，暴力打远角，最小消耗，应对旋转球等几种模式。

估值算法（evaluate）是我方的核心策略，思路为计算每一次击球自己的总消耗（跑位消耗与击球消耗之和）与造成对方的跑位消耗之差作为当次击球的收益，如击球路径上有道具，则总收益还应加上道具的收益，遍历所有可能的击球加速度，找到收益最大的击球加速度。对道具的价值的估计很大程度上影响着估值算法的准确性，曾经有机器自动调参来调整道具价值估计的想法，但最后并未实现，因此道具的价值作为一个固定的字典给出，其中的具体数值是我们认为的效果最好的价值估计，此处还有改进空间。

随机打角模式（random_corner）调用 corner_violet_acc 函数，随机打上角或打下角，此种模式主要针对相对保守偏重防守尤其是具有一定学习能力的算法，加入随机性使对方难以判断下次的击球点，使对方跑位得不到好效果。

最小消耗的打角模式（corner）调用 corner_least_acc 函数，打角以对对方造成较大的伤害，且优先保证自己的消耗最小。但这一策略偏保守，也相对容易被针对，当对方不跑位到中点时，如果回球回到对方板附近，对方消耗会很少。

暴力打角模式（violet_corner）直接调用 corner_violet_acc 函数，主要针对对方存在死角的情况，判断对方的位置并打远端角，希望造成对方 miss_ball。

最小消耗模式（lazy）直接调用 legal_least_acc 函数，是最保守的策略，仅在需要降低损失时使用。

应对旋转球模式（spinned）调用 legal_least_acc 函数，是对旋转球进行防御的算法.若对方使用了旋转球，我方这一击只求合法击出且消耗最小。采用最小的合法击球加速度，在输出时乘 2。

2.2.4 跑位算法

跑位算法是 play 函数在决定加速度后的最后一部分。

我们的最初的想法就是每一次跑到中点，或者跑位到当前位置与距自己较远的角的中点。

然而，经历了第一次热身赛，我们发现，中值跑位算法看似稳妥，实际会带来巨大的不必要损耗。因为我们在每一次击球之后都会固定的跑到中点，因此每一次击球后都会有固定的不小的损耗；而且，很可能出现的一种情况是，我们的两次接球都在中点同侧，显然跑到中点这个选择是很恶劣的。跑位到当前位置与距自己较远的角的中点，也有类似的问题。

因此我们基于逆推的思路编写了新的跑位算法。其原理是：每一次 table 函数在执行时会将我方击出球的纵坐标和速度传给对方函数，对方会通过这两个参数推算出接下来球的落点并进行跑位击球。如果对方的算法不具有学习性，那么我们提出一个概念：这两个参数是完全的。具体的进行解释，意思是，我方击球的纵坐标和竖直方向速度和对方接球点的纵坐标以及球的竖直速度是等价的，因此在不考虑学习性的前提下，对方的所有程序相当于是这一纵坐标和竖直速度二元函数。因此这两个参数可以完全决定对方在这一回合的击球方式，对球的加速度以及之后的跑位位置。形象的讲，对方的程序就是一个黑箱，而现在这一黑箱成为了一个泛函 f(v,y)。v 是竖直速度，y 是纵坐标，那么我们可以通过之前存储的数据对于对方的程序内容进行反推。

算法的核心思想就是统计平均方法。我们基本可以确定对方的二元函数是一个连续的函数，因此我们可以通过介值定理进行估计。我们的程序会将每一次我们击出球的位置和速度记录，同时存储对方返回球的落点（本质上是这个函数的函数值）。在我们某一次击完球后，程序会在历史数据里搜索速度与纵坐标这一整数对在某一范围内的历史数据，记录下对方下一次击球的落点，作为我们对于对方击球落点的预估位置。这样，我们可以跑到当前位置和这一位置的中点。

代码中函数 **Calpos(T,A,y0,v0)**就实现了预估位置的功能。A,T 是存储的历史数据（action 和 tabledata），v0，y0 是本回合球击出的速度和纵坐标。

之后的测试中，我们发现，这一算法使得我们原有的程序得到了一定的提升，在对方的程序不具有学习性和随机性的时候，我们的跑位算法在后期甚至可以非常精准的完成估计，从而将跑位损失降到了很小。但是，我们在后来的热身赛中再一次发现了跑位算法的缺陷。有些时候，我们可能为了跑到某一位置而过于偏向某一个角落，这样一旦对方调整策略击打另外一个角落，我们会在迎球时消耗巨大的体力值。而随机性的算法，比如含有随机打角成分的程序本身就不是由自变量确定的函数，预测也无从谈起。总的来说就是我们激进的跑位会带来很多风险，而且这一思路并不具有完全的合理性。

最终，我们对跑位程序进行了改进，具体的方法是进行分类讨论，如果预计落点位于距离中点的六分之一附近，那么直接跑位到中点，因为这极有可能是对方打上角、下角平均的结果，这个预估的均值没有参考意义；否则，让球拍跑到当前位置与预计落点的中间（维持原策略）。

2.3 程序限制

未发现程序有极端条件导致出错的情况（包括出现 invalid bounce）。

3 实验结果

3.1 测试数据

3.1.1 与组内测试算法的测试

实验环境说明：

- 硬件配置：
  CPU: Intel Core i5-4590 CPU 3.30GHz
  RAM: 4.00GB
- 操作系统：Windows 7
- Python 版本：3.5.3

测试方法：

将 Charlie 与五个测试算法在 pingpong 平台上运行，两两循环制对打 20 局。

参加测试的算法：

Idiot：平台提供的测试程序，只会被动接球
Taichi：最小加速击出算法，即主程序 lazy 模式
Gambler：随机打上下角的程序
Angler：暴力打远端角程序
Wonka：含有捡道具功能的纯估值搜索程序

Charlie 算法测试结果：

对手	Charlie 先手胜场	Charlie 后手胜场	总胜场	胜率
Idiot	10	10	20	100%
Taichi	10	10	20	100%
Gambler	9	10	19	95%
Angler	10	10	20	100%
Wonka	4	6	10	50%

3.1.2 热身赛与组间比赛结果

日期	比赛名称	派出的主算法	结果
5 月 29 日	第一次热身赛	Luna	128 胜 68 负，上半区第 3
6 月 1 日	MOCK 四组约战	Charlie	57 胜 13 负，第 2
6 月 2 日	第二次热身赛	Apollo	132 胜 53 负，总胜率第 6
6 月 4 日	比赛前夜挑战 Oscar	Charlie	2 胜 8 负
6 月 5 日	小组赛（West 区）	Charlie	71 胜 29 负，小组第 2 出线
6 月 5 日	八强战（Charlie vs Mike）	Charlie	31 胜 49 负，被淘汰

3.2 结果分析

总体上，Charlie 面对模式相对固定的简单算法优势明显，这得益于 Charlie 的模式分析模块判定了对方的模式，可以以相应对策还击，而且跑位算法也能比较好地估计出对方的跑位。而面对有一定随机因素的比较复杂的算法，优势并不明显，这是因为模式分析和跑位的统计平均都是在对方策略相对固定的情况下才有明显效果，这时 Charlie 的表现与单纯的估值相当。

另外，打破坏循环和一击制胜两种策略起到一定作用，前者有时使程序在陷入坏循环时扭转局面，后者有时使自己在生命值落后的情况下一击翻盘。这是我们观察到的惊艳表现。

算法运行时间一般在 1-2 秒，至多 3 秒，符合预期。

4 实习过程总结

4.1 分工与合作

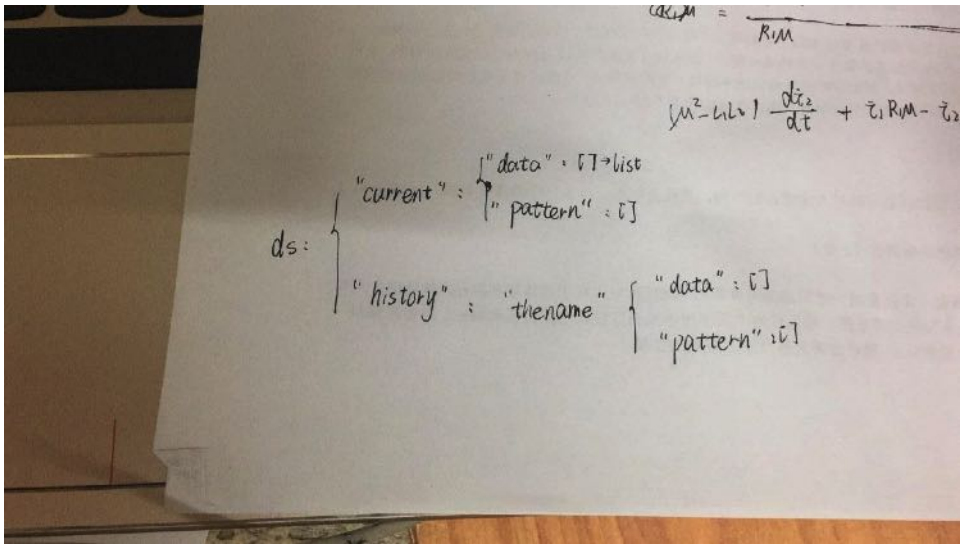
4.1.1 分工与各组员的贡献

最初，组长郭浩出于增加每个组员参与度，最大限度地集中智慧的考虑，采取分模块编程的方式，即赛前每个人都承担自己部分代码的编写和测试任务；赛后每个人都承担自己所完成部分的代码编写任务。但此种方式实际上并没有很好地执行。一方面，各组员在提出算法、编程、测试的能力各不相同，让每个人完成一个模块的所有流程的确不太切合实际；另一方面，两三个程序员协作开发的对接比较容易，七份代码的合并则带来很多麻烦；最终结果是有的组员承担了额外的任务，有的组员编写的代码则没有被采用。

在实际的工作中，编码任务主要由郭晴宇、郭浩、罗哲楷完成；孙景南、王泽鑫的主要贡献在测试方面；杨德鼎、哈杰则提出了一些策略和思路。

作为组长，郭浩负责召集组会、选派算法、为每位组员分配任务，并从全局上把握开发的进程，负责各位组员成果的整合与协调。他在初次看到程序时，发现 shelve 中复盘数据不易读取的问题，于是编写了 reader.py 用于检查中间结果，并贡献到平台上。这一程序在 invalid bounce 错误的排除中起到了一定的作用。他完善了估值搜索模块并加入了道具系统，这一部分一直沿用到最终版本。此外，他还修复了一度困扰全组的 east 不捡道具的 bug，尽管这 bug 是他自己造成的。

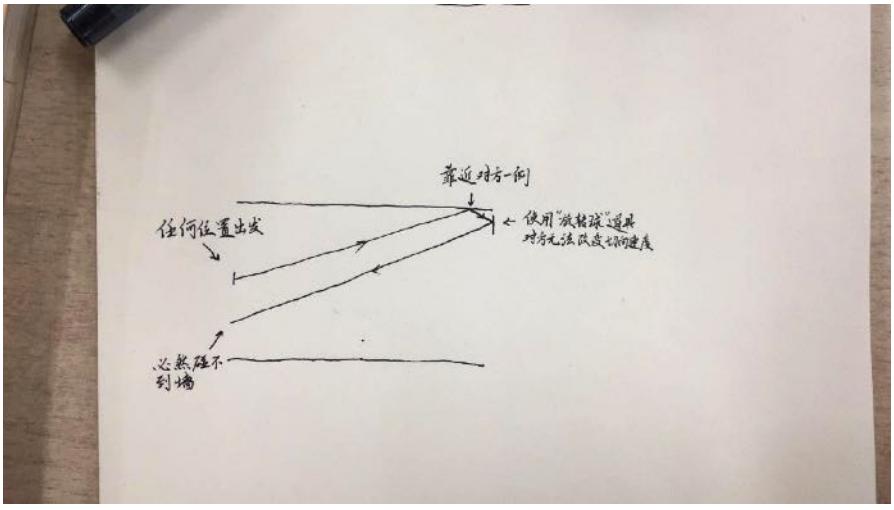
郭晴宇在熟悉平台代码的阶段为全组同学编写了一份 1500 字的帮助文档解释平台的运行机制，方便组员阅读代码。在第一次组会上，他提出模式分析的思路。接受编写这一模块的任务后，他首先参与制定了 ds 的使用方式，随即编写了 analysis.py 实现对当前状态和对手模式的判别，并加入了对对方加速度的计算和跑位的估计。在这一过程中，他还完善了函数库，并对部分函数进行了改写。在最后阶段，他还把杨德鼎提出的新跑位算法成功加入主程序。他为漂移乒乓大作业所做的大量工作得到了全体组员的好评。民主评议时，他全票通过获得加分。



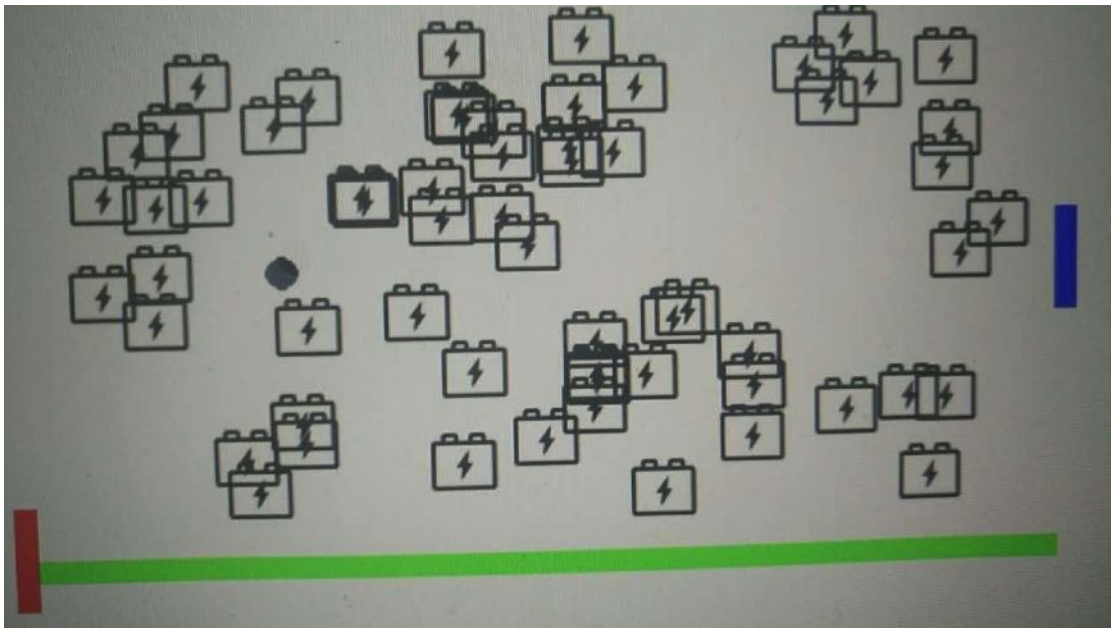
郭晴宇、哈杰讨论设计的 ds 结构

罗哲楷最初的关注点在游戏规则的研究，探究了旋转球的 bug 等问题。开发开始后，他在自己尝试的程序中编写了五种不同算法，作为陪练在之后的编程中起到了重要的参考比较作用，其中 lazy、corner 是最终模式的原型，evaluate 模式经郭浩改写后成为核心算法。在这一版本中，他还初步建立了函数库。最后阶段，他还为小组贡献了两个新的陪练程序。

罗哲楷同学手稿（旋转球 bug）



孙景南同学由于积极完成对组内各算法的测试逐渐成为测试专员。为更好地完成测试工作，他还参与了 github 主程序的完善，实现了 show.py 中可选择播放对战、颁发奖杯等功能。在测试中，他还通过修改平台代码的方式来达到突出的测试效果。此外，他会在每次热身赛、约战后第一时间观看复盘，向大家介绍发现的问题。



孙景南暴力测试变压器时的截图

杨德鼎作为即将成为数学科学学院一员的数学奇才，在发现程序的跑位问题后，从统计学的角度思考，提出了根据对方历史落点数据估计落点的方法，推动了主程序战斗力的又一次提升。除此之外，他还提出预测对方跑位来指导己方加速，但由于对方跑位扣血的整除处理导致的误差会引起 bug 以及时间原因，这一想法没有实现。

王泽鑫同学前期参与了道具使用与防守策略的制定，后期与孙景南共同测试，为最终算法的模式参数提出了建议。

哈杰同学前期参与了 ds 的设计。她后期展开对其它跑位算法的探索，但这部分代码由于对接未完成没有很好地测试和运用。

4.1.2 组会与其它合作方式

我们组组员间的合作交流包括组会、微信群讨论、共同工作三种形式。

组会的主要流程是：首先分析热身赛对局记录和数据（如果有），之后各位组员谈自己的想法并进行讨论，最后由组长分配下一阶段的任务。组会共进行了四次。

第一次组会于 5 月 22 日数算课后在理教 107 举行。组会确定了第一周的目标是在第一次热身赛前完成无低级失误、可运用道具、具有一定实力的程序。各组员分工开展对估值搜索部分、道具系统的编程。

第二次组会于 5 月 30 日上午 10:00 在二教树下学习空间举行。组会根据热身赛发现的问题，按照完善原有算法与尝试新想法两部分展开工作。完善部分包括对模式决策的修正、east 拿道具的 bug、统一算法的结构等，新尝试包括新型跑位策略、对方跑位的计算等。

第三次组会于 6 月 3 日上午 10:00 在二教树下学习空间举行，讨论赛前的最后工作。最后的工作主要是根据第二次热身赛的结果调整跑位策略和模式对策。

第四次组会于 6 月 5 日数算课后在理教 107 举行。八强赛中被 Mike 击败，我组提前进入整理算法、编写报告的阶段。这次组会根据每个人的工作对报告进行了分工。

微信群“漂移乒乓 Charlie”用于文件的共享和随时发布信息，是组员平时交流的主要途径。

共同工作即 2-3 人一起完成代码的编写、测试或修改。后期我组采用了此种方式，如杨德鼎与郭晴宇合作完成新跑位算法。

4.2 实习过程回顾

4.2.1 Charlie 组大事记

5 月 11 日，七人小组建立，郭浩自荐担任组长。

5 月 17 日，项目初稿正式发布。

5 月 17 日-5 月 21 日，小组阅读平台代码，郭晴宇、孙景南分别就游戏代码的运行和平台的使用撰写了帮助文档。郭浩、郭晴宇、罗哲楷分别尝试编写了对战程序，实现了打败 idiot 的最低目标。

5 月 22 日，第一次组会。

5 月 24 日，经过投票，哈杰推荐的“三体-水滴”成为 Charlie 组头像。

5 月 24 日，ds 设计完成。

5 月 25 日，第一个含估值搜索的正式算法 Chaplin 完成。

5 月 26 日，模式识别部分初步完成。

5 月 27 日，道具部分加入，基础开发完成。

5 月 28 日，选派一支战队参加第一次热身赛，包括从暴力打角算法和最新若干版本的程序。

5 月 29 日，课上 26 各组交流本组进展，郭浩特别提及了打角的优势难以压制以及先发方的优势。后来发现，先发方的优势是我们自己的 bug 造成的；我组因发现估值对于打角在当前参数下优势不明显，一直建议修改参数，但未被采纳。

5 月 29 日，第一次热身赛。

5 月 30 日，第二次组会。

5 月 31 日，加入统计平均跑位算法。

6 月 1 日，郭浩发起 MOCK 四组约战，派出当时组内最强的 Apollo 算法，改名为 Charlie，交由冀锐助教运行。郭浩是如何察觉到 M、O 这两个未来的冠亚军组的实力而邀请他们的？这是一个可与决赛 ppt 末尾的“OMG”媲美的神奇预测。然而戏剧性的是，约战的成绩与四个组最终的名次完全相反。（据 O、M 两组组长透露，当时 M 组的主算法战力的确如此，后来提升很大；O 组的算法已取得第一次热身赛第一，新版本用时很长，于是提交了组员纪晴开发的一个新版本，但其中有 bug。）

6 月 2 日，第二次热身赛，发现 lazy 模式对付打角的明显问题。

6 月 3 日，第三次组会。

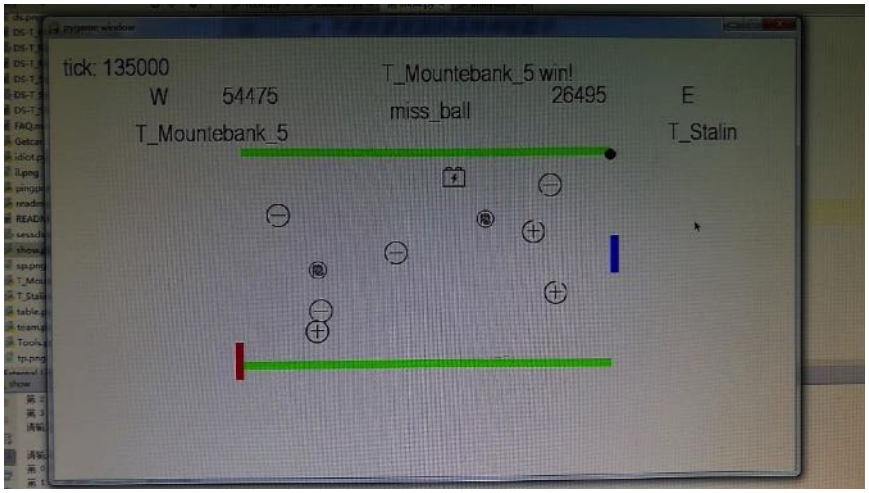
6 月 4 日，最终算法提交。

6 月 5 日，现场竞赛中，Charlie 以生命值的优势挫败 Papa 惊险出线（当时算法中还有 lazy 的选项），八强赛首盘告负，修改代码后仍不敌 Mike。课后第四次组会。

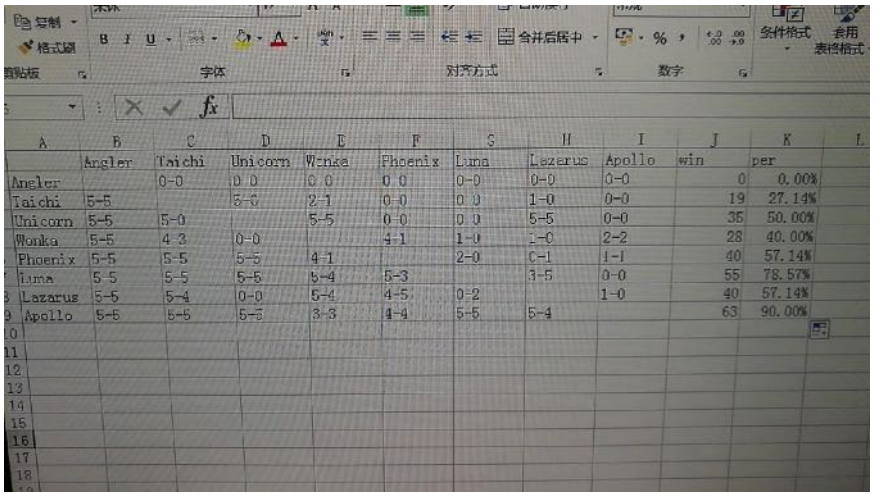
6 月 10 日-6 月 13 日，组员共同完成了实习报告的编写和代码注释的完善。（罗哲楷：2.2.1 节；郭晴宇 2.2.2 节；王泽鑫 2.2.3 节；杨德鼎 2.2.4 节；郭浩完成其它部分并统稿）

6 月 13 日，添加好注释的 Charlie 算法由郭浩在 github 上开源。网址为：https://github.com/Nithouson/Charlie-and-its-training-partners

4.2.2 影像记录



组内持续的程序对打



	Angler	Taichi	Unicorn	Wonka	Phoenix	Luna	Lazarus	Apollo	win	per
Angler		0-0	0-0	0-0	0-0	0-0	0-0	0-0	0	0.00%
Taichi	5-5		5-0	2-1	0-0	0-0	1-0	0-0	19	27.14%
Unicorn	5-5	5-0		5-5	0-0	0-0	5-5	0-0	35	50.00%
Wonka	5-5	4-3	0-0		4-1	1-0	1-0	2-2	28	40.00%
Phoenix	5-5	5-5	5-5	4-1		2-0	0-1	1-1	40	57.14%
luna	5-5	5-5	5-5	5-4	5-3		3-5	0-0	55	78.57%
Lazarus	5-5	5-4	0-0	5-4	4-5	0-2		1-0	40	57.14%
Apollo	5-5	5-5	5-5	3-3	4-4	5-5	5-4		63	90.00%

热身赛程序的组内选拔



正式比赛中的 Charlie 组



止步八强



Charlie 组合影

（6 月 5 日第一场正式比赛、第四次组会后拍摄于理教 107 摄影：孙宁远）

（第一排左起：郭浩、罗哲楷、杨德鼎、郭晴宇）

（第二排左起：孙景南、哈杰、王泽鑫）

4.2.3 花絮：组内各版本程序的命名

1. 以作者命名：Azazel（郭晴宇的微信名，意为魔鬼）；Mountebank（罗哲楷的微信名，意为江湖骗子）。
2. 以算法思路命名：Unicorn（独角兽，意为打角程序）；Midgoer（跑中位）；Unimidgo（前两者的杂交品种）；Taichi（lazy 打法，取太极以柔克刚之意）；此两种前期常用。
3. 与组名相关：Chaplin（出自喜剧大师 Charlie Chaplin）；Wonka（出自 *Charlie and the Chocolate Factory* 中巧克力工厂的主人 Wiley Wonka）；Violet（同一作品中另一人物 Violet Beauregarde）。
4. 与圣经故事有关：由郭晴宇首先推出 Issac；其姐妹篇为 Ismael；第一个融合道具与模式判别的程序定名为 Lazarus。
5. 与神话有关：后期沿用，包括 Phoenix，Luna，Apollo，Hera，Zeus 等，原则是要一个比一个厉害。

4.3 经验与教训

算法方面，我们的程序综合了多种想法，具有一定的复合性。其中对出现的循环进行判断应该可算得上程序的一个亮点。但我们并没有真正把握游戏的要领；程序中的模式对策和程序的参数，都是我们通过人工测试和观看对局调整的，因此不能排除测试对手带来的片面性。

总结来看，面对这样一个游戏，有两种写出高水平程序的思路：一是人为制定策略，然后翻译成代码交给计算机去执行，这就需要反复测试、分析，调整战术；二是发挥机器的搜索能力，依靠增加计算量尽可能去优化自己的策略。本次竞赛的亚军组 Mike 和冠军组 Oscar 就分别是这两种策略的典型。反观我们组，开始的估值函数遇到打不过打角算法的情形后，我们没有对估值函数加以改进，也没有试图调整跑位策略，而是用模式的切换掩盖了估值函数的问题。最终结果是两条思路骑墙走，哪一条也没有走好。此外，有深度的估值是被我们开始就否定了的，但从 Oscar 树搜索成功的结果来看，在限定搜索的关键点后可以有效控制时间，因此并非不可行，放弃这条道路也是轻率的。

组织方面，组长郭浩认为本组一起投入、用于共同工作的时间较少，因此没有能够发挥组员的全部潜力。

4.4 建议与设想

第一，仍然建议在规则上作出调整，以限制打角算法，因为这样做在不加速则不能合法击出的前提下不仅自己加速消耗最小，对方迎球消耗也最大，违反游戏中不同选择的平衡性。实战发现 Oscar 除拿道具外几乎所有回球都是打角的，这很能说明问题。本组组员杨德鼎建议把场地变为一个圆柱的侧面，双方球拍在圆柱两底面的边缘移动，这样所有的位置等价，不存在打角的问题，游戏更能反映程序的决策能力。

第二，建议下次作业中助教参与到前期平台的调整中来，即他们也编写保密或公开的对战程序，以及时发现平台的问题。这些也可以参与到热身赛中，正式赛后冠军队有挑战助教

的“荣誉战”资格。

第三，建议效仿 botzone，为各组随时约战其它程序建立一个平台，还可以维护一个天梯排名。

第四，一个比赛项目不应因学期的结束而终止，可定期举办漂移乒乓算法比赛，面向全校乃至全网络开放；甚至可以设想举办包括坦克大战、漂移乒乓以及数算课今后几年积累的项目的算法运动会。

第五，可在将 26 个组算法开源的基础上，组织感兴趣的同学开展关于漂移乒乓算法的相关研究，完成一篇关于其策略选取的学术论文。

5 致谢

感谢陈斌老师在 github 上及时回应我组同学的提问，及时解决有关平台的问题。

感谢冀锐助教对我组参加热身赛与约战的支持。

6 参考文献

漂移乒乓对战平台：https://github.com/chbpku/pingpong.sessdsa



HOTEL

组长：刘立洋

组员：汪其霖，贺麒霖，张峻伟，刘璇，马嘉成，李嘉冕

小组赛战绩：N2

决赛战绩：八强









组T恤Photo

HOTEL 小组-数据结构与算法课程实习作业报告

（刘立洋*、张峻伟、李嘉冕、汪其霖、刘璇、贺麒霖、马嘉成）

摘要：

本组的总体思路，建立在一个收益值函数上，即每一回合我方消耗与对方消耗的体力值的差值，差值最小的时候为最优解，通过分析对方击球、跑位、使用道具的情况、历史击球数据，并对对方的策略进行一定程度的预测后，调整己方策略（包括击球、跑位和使用道具），最终达到保存自己，消耗对方的目的。算法中使用了列表和字典的数据结构。

关键字：

收益函数 列表 字典

1 算法思想

1.1 总体思路

本组的总体思路，建立在一个收益值函数上，后期经过热身赛以及多次讨论，在此基础上进行完善与改进。

漂移乒乓获胜要求是将对方打到体力不支判负，己方能获知的信息是球的速度位置和对方击球时的位置跑位方向。故而最开始，在不考虑道具的情况下，我们的收益值函数是每一回合我方消耗与对方消耗的体力值的差值，差值最小的时候为最优解。在不知道对方跑位策略的前提下，将对方跑位视为零。我方的消耗=跑位消耗+加速消耗+迎球消耗，其中跑位和迎球建立在对对方的预估加速上，即视对方采取同样找最优解策略，对方消耗=迎球消耗+加速消耗，最后得出一个关于加速的函数。遍历所有加速求出加速的最优解，同时将加速限定在反弹速度范围内并且设定为不改变球的 y 方向速度。我方的跑位策略是在安全区外就跑到安全区，安全区内随机跑一段较短距离迷惑对方。

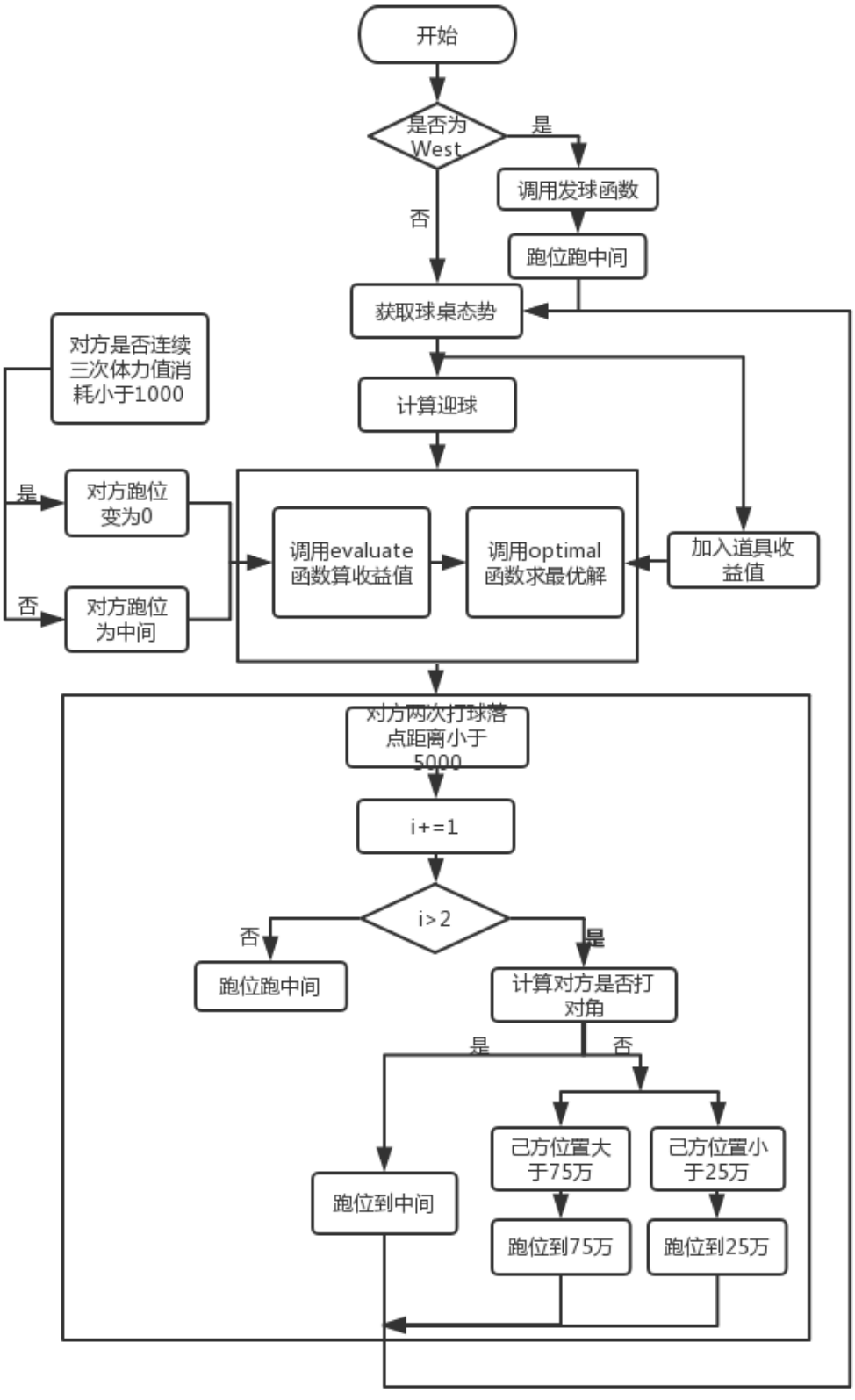
但是在测试中发现，可能由于最初设定加速消耗太小，收益值函数考虑五个因素与只考虑我方加速与对方迎球的策略结果相同，输出结果都是暴力打对角。因而不再预估对方加速。

热身赛后，在对各组比赛情况的复盘分析中，发现多数小组的跑位策略都是跑到中点。于是，在收益值函数中修改了这一参数。在对方跑中间的情况下，打两个角的加速消耗不一样，优先选取加速消耗更小的。在对方将我们的跑位也视为中间时，考虑对方是否一直打对角。若只打对角，一直跑中间，若可能打对角，考虑对方打哪个角消耗更少，如果是对角，跑到中间，如果是同一个角，只跑四分之一。考虑到如果对方跑位策略和我方一样，我方不只打对角会利于对方，又加进了条件语句，判断对方是否连续几次跑位消耗都极少，如果是，加速策略变回打对角。

对于道具，将捡道具的收益加进收益值算法中，道具的收益值如何得出具体见第二部分。将被用道具的情况用条件语句加进收益值函数。

在信息获取不完全的情况下，我方的策略建立在对对方的预估上，纯粹地计算最优解，不强求战术的复杂，只是需要计算对方是否符合预估，以便改变加速和跑位战略。本组的思路是见招拆招，经过多次修改最终完成。

1.2 算法流程图


开始
是否为West
是
否
调用发球函数
跑位跑中间
获取球桌态势
对方是否连续三次体力值消耗小于1000
计算迎球
是
否
对方跑位变为0
对方跑位为中间
调用evaluate函数算收益值
调用optimal函数求最优解
加入道具收益值
对方两次打球落点距离小于5000
i+=1
i>2
否
是
跑位跑中间
计算对方是否打对角
是
否
跑位到中间
己方位置大于75万
己方位置小于25万
跑位到75万
跑位到25万

1.3 算法运行时间复杂度分析

我们组的算法比较注重时间的控制，并没有采用太过复杂的算法，在整个算法中只有optimal 函数对所有可能的加速度做了一次遍历，因而时间复杂度为 O（n）。

2 程序代码说明

2.1 数据结构说明

数据结构使用了字典，列表。

2.2 函数说明

（1）函数名称：play
引用函数：本函数为整个算法究极函数大 boss，因此之后的所有函数都被引用了。
函数功能：综合各种数据来决定球拍移动
所用参数：tb.tick: 时间 bat_vector：球拍运动矢量, acc_vector：球拍加速矢量，foresee_ball：预测球的位置 run_vector：跑位矢量，op_pos：对方位置
算法策略：此为究极主函数，算法策略同 1.1 总体思路和之后的函数。

（2）函数名称：optimal
引用函数：availrange, op_card, evaluate
函数功能：决定球的加速度
所用参数：self_y：己方位置纵坐标，self_speed:己方 y 方向速度，oppo_y:对方位置纵坐标，tbcard:桌面道具列表，op_card:对方使用道具，self_card:己方道具箱，self_x:己方位置横坐标，velocity_x:横向速度, ds:函数可以利用的存储字典
算法策略：

1. 先用 availrange 函数判断有利区间范围
2. 如果对方使用旋转球，则将有利区间最大最小值都乘以二
3. 应用枚举的办法，在有利区间的范围内得到一个最有利的加速度值。

（3）函数名称：optimal_run
引用函数：无
函数功能：跑位策略
所用参数：tb: 桌面相关参数 ds: 存储字典 foresee_ball：球预测落点
算法策略：

1. 这个函数用来判断对方的打法，进而用来决定跑位策略。
2. 利用 ds 来存储接球的位置，如果对方连续三次的球落点都是在同一位置附近，判断对手是打一角的打法。则我们相应的每次跑位都离相应的角落不远
3. 如果不是，则跑位至中间。

4. 特殊情况：若体力不足，则跑中间。

（4）函数名称：evaluate
引用函数：calc_ball,get_card,cardvalue
函数功能：预估我方收益值
所用参数：self_y:己方位置纵坐标, self_speed:己方 y 方向速度, self_accelerate:己方加速值, oppo_y:对方位置纵坐标,tbcard:桌面道具列表, op_card:对方使用道具,self_card:己方道具箱,self_x:己方位置横坐标, velocity_x:横向速度, ds:函数可以利用的存储字典
算法策略：
1.计算我方加速体力消耗，若对方使用变压器消耗加倍。
2.计算加速后的速度，若对方使用旋转球加速减半，引用 calc_ball 计算球到达对方的位置和速度。起初假定对方跑位策略为台子中央，若累计 3 局对方体力减少小于 1000，则假设对方不跑位，依据球的位置计算对方迎球体力消耗，要考虑对方使用瞬移术的情况。
3.For 循环遍历道具箱中道具，引用 get_card 判断若能捡到道具，若道具箱已满，则有道具将消失，道具收益为二者之差；若未满，道具收益即为 cardvalue 中的估计值。我方最终消耗值应减去道具收益值。
4.返回双方体力消耗及二者之差，即我方收益值。

（5）函数名称：calc_ball
引用函数：无
函数功能：计算击球后求到达对方时的位置和速度
所用参数：current_y:己方击球位置纵坐标,current_speed:己方击球后球在 y 方向的速度, period:一方击球至另一方迎球的时间间隔
算法策略：如图 1，我方为 West（若为 East 结果相同），假设两侧墙壁不存在，求出球到达对方时的纵坐标 y。使用 if 条件语句按坐标范围分为向上击球反弹 2 次、向上击球反弹 1 次、向下击球反弹 2 次、向上击球反弹 1 次 4 种情况，分别计算并返回球到达对方时的纵坐标和 y 方向速度（反弹次数为奇数，速度方向改变，偶数次方向不变）。

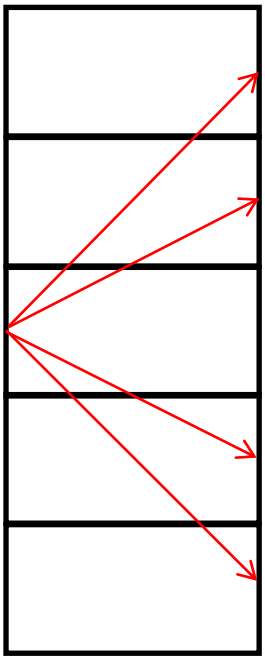


图 1 calc_ball 函数示意图

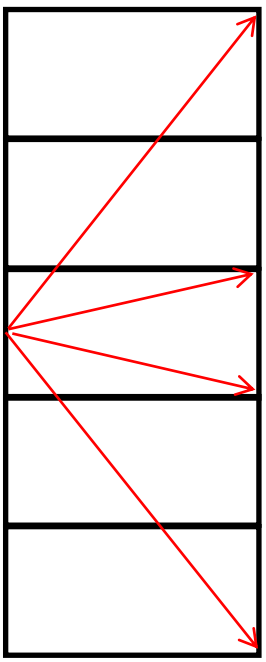


图 2 availrange 函数示意图

（6）函数名称：availrange
引用函数：math.floor,math.ceil
函数功能：计算使得球能满足碰壁 1 或 2 次的加速度的有效范围
所用参数：y:迎球方位置纵坐标,speed:击球前球在 y 方向的速度
算法策略：如图 2，迎球方为 West（若为 East 结果相同），使用 if 条件语句按 speed 的正负分为球从下方来和球从上方来 2 种情况。球碰壁两次后恰好到达边界和球恰好不碰壁为加速度取最值的两个临界条件，求出速度的最值取整减去现有速度即得到加速度的最大、最小整数值 Max 和 Min。最终返回 Min+1，Max—1，对加速度的范围进行了缩小以防止球在边界处出现无效反弹的情况。

（7）函数名称：op_optimal
引用函数：availrange,calc_ball,op_evaluate
函数功能：预测对方的击球策略，以便制定我方跑位策略。
所用参数：self_y:对方击球位置纵坐标,self_speed:对方击球前球的纵向速度, oppo_y:对方预测的我方跑位位置,accelerate:对方加速值,optiaml:当前最大收益值,temp:某加速值对应的收益值,op_pos:某加速对应的我方落点,result:当前最大收益对应的我方落点
算法策略：引用 availrange 计算对方加速的有效范围。利用 for 循环使加速度遍历，引用 calc_ball 计算每一对方加速度对应的我方迎球位置，为减少计算量，利用 if 条件语句使得只有当球落到我方两角附近 10000 以内的位置时，才引用 op_evaluate 计算此加速度下对方收益值 temp，并与先前的最大收益 optimal 进行比较。循环结束后会得到一个最大收益 optimal，对应的 accelerate 即为我方预测的对方最佳加速策略。函数最终返回对方收益最大时对应的球在我方的落点 op_pos。

（8）函数名称：op_evaluate
引用函数：calc_ball
函数功能：预估对方的收益
所用参数：self_y:对方击球位置纵坐标,self_speed:对方击球前球的纵向速度, self_accelerate:对方加速度, oppo_y:对方预测的我方跑位位置
算法策略：
1.计算对方加速体力消耗。
2.引用 calc_ball 计算对方击球后我方的迎球位置，可求出我方由跑位位置迎球的体力消耗。
3.返回双方体力消耗之差，即对方收益值。实际使用时此函数时假定对方预测的我方跑位策略是跑到台子中央，即 oppo_y=500000。

（9）函数名称：cardvalue（道具使用函数）
引用函数：无
函数功能：记录所有道具的收益值，返回对应道具的收益值
所用参数：card－道具种类
算法策略：旋转球收益值为经验值，隐身术收益值设为 0，补血包和掉血包收益值由规定参数决定为 2000，瞬移术收益值由跑 500000 和 250000 的体力消耗差值决定，经过计算得为 450。

（10）函数名称：get_card
引用函数：无
函数功能：返回是否捡到道具
所用参数：速度的 x 方向分量，y 方向分量，道具位置坐标 x，y，道具种类，误差值 2000
算法策略：
直线方程为 l:-v*x+u*y+v*x0-u*y0=0，card 经过多次对称后，位置为(x1,±y1+2*k*l)，点到直线距离公式 dist=|-v*x1+u*(±y1+2*k*l)+v*x0-u*y0|/|self.velocity|，记-v*x1+u*(±y1)+v*x0-u*y0=A1,A2,u*2*l 为 delta。则求最短距离，即是求 A1%delta,A2%delta,-A1%delta,-A2%delta 中最小值，再除以 self.velocity 的模长。

（11）函数名称：use_card
引用函数：无
函数功能：判断是否适用道具，返回使用道具的类型及作用对象
所用参数：自身在 y 方向位置，道具箱中已有道具，对手位置
算法策略：对道具箱中的道具是否使用进行判断，补血包、掉血包、隐身术和旋转球都是捡到之后马上被用，瞬移术在自身位置比较偏离安全区域时才使用（y<250000/ y>750000），变压器因为试图放大对方迎球消耗的体力值，所以在对手非常靠边的时候，体力值可能消耗非常大的时候才用（y<8000/ y>992000）

2.3 程序限制

似乎没有

3 实验结果

3.1 测试数据

实验环境说明：

- 硬件配置：Intel®Core™i7-4710MQ CPU @ 2.50GHz，内存 8.00 GB
- 操作系统：Windows 7 旗舰版
- Python 版本：Python 3.5.3

本小组一方面通过安装 pygame，利用张灏丹开发的 show.py 进行对战战局的直观展现，在复盘视频中摸索本组 AI 是否实现了设计的功能，或者出现的 bug；另一方面通过输出 history 数据到 txt 文件中保存，history 数据包括：“每一次击球过程的：自身击球位置、球初速、给球加速大小、加速可行范围、对方击球位置、每局计算得到的净赚体力值”。

基础功能的测试：在开发前期，为了测试迎球、跑位、捡取道具、使用道具等功能是否实现，故仅通过 show 程序显示视频，直观测试是否有相应动作。在实现基础功能后，本小组加入击球策略：每一次计算能使得己方净赚体力最多的步骤，这种贪心法的使用的直观体现是每次击球到对方的角落，使之跑位距离最大。前期，我们没有开发新的测试程序，仅对

战 idiot.py，测试是否能实现我们的击球算法，通过观察视频即可。

道具系统的测试：在实现基本算法保证每次击打处于合理范围，不会出现不击墙壁或者击墙三次的情况之后，本小组探索了在何种情况捡取道具，何种情况使用道具。在此处，我们使用的测试对手是实现基本功能（但未加入道具使用系统）的前期版本和对道具赋予不同值的版本，继续利用 show 程序观察对战视频，判断是否在不捡道具的时候打角，判断是否按照设计立即使用道具，这些功能都能通过视频直观显示出来，如果出现特殊情况，我们会录屏截取特殊情况的画面（如：尝试捡道具但未捡到等），再分析其 history 数据，观察在那个情况下加速范围是否处于我们设计的合理范围内，或者合理范围是否不符合实际情况，再进行修改。

跑位系统的测试：在前期，本小组的 AI 不进行跑位，仅一次移动到位，消耗了大量体力，之后我们加入跑位的评价函数，仍旧通过视频观察跑位方向是否正确，跑位是否超出界面，是否根据我们设置的范围进行区分跑位等。如果出现 bug 则根据视频获得相应步骤的 history 数据，再进行分析是否参数设置错误。在这一阶段测试的对象是 AI 复制版本，但跑位策略进行调整，我们根据对于对手的猜测，设计了几种对手，如每次跑位到球台中间、仅位于边缘 1/3 时进行跑位、只有体力不支时进行跑位等几种跑位策略，进行测试，确保本小组的跑位策略能处于大部分的优势地位。

3.2 结果分析

在对战过程中，本小组较好的策略在于：每一步以最小的己方消耗获取对方最大的消耗（最优解），跑位上按体力分成两种跑位策略：跑向安全范围（250000-750000）和跑向正中间，以体力为划分条件。这种跑位策略相对于一些策略复杂的队伍（如 Oscar 组）可能不能处于优势，但是对 North 区的其他对手，相对于一些没有指定跑位策略或者简单跑位策略的小组，我们的跑位策略能够获得较好的效果。

我们小组的时间复杂度较低，尤其是对比 Oscar 和 Juliet，我们的处理时间都是瞬间级别，但是没有复杂、智能、全面的分析体系，也让我们在面对这种穷尽各种应对情况的队伍时处于劣势。

在出线后应对 Alpha 时，本来难分伯仲，甚至我们占据优势，但是 Alpha 队伍修改了策略后，我方落后，哪怕后来微调了跑位策略，但仍未能逃脱出局命运，止步四强外。

通过算法的优化后，输的战局都是因为体力不支，无法达到迎球的位置而 Miss Ball 的原因，正式对战时没有出现过击墙次数错误的低级错误。

晋级赛：只落后于本小组 Oscar 队，其他队伍由于策略复杂性差距与本组较大，故能轻松取胜，对于 Oscar 队，从时间复杂度上可以看出本组 AI 考虑的情况没有 Oscar 多，之后也未能再战 Oscar，可能与其精准的跑位策略有关系。

八强赛：第一轮（对战 Alpha），以胜率 29/40 稍微领先 Alpha，通过分析获胜情况，可能和本小组对于道具的赋权较合理有关，尤其是球台上的“变压器”道具，很多时候由我们捡到道具，靠道具获得体力的反超，在跑位上，两边拉开的差距不大。通过 show 程序分析我们落后的战局，发现由于我们小组跑位只是在前期和后期采取跑中间（分别是试探对方打角策略和节约体力）的策略，但 Alpha 总是采取打安全范围另一边的角落，造成我们小组的迎球消耗较大，而且没有开发相应的评价函数来应对这种打角策略。

在之后的修改过程中，由于时间较紧，本小组在前期也未能开发出节约时间的分析出对方打角、跑位策略的程序，只能通过效率较低的观察视频来推测对方策略，故未能根据这种情况修改战略，只是将安全范围作了一个微调，未做大的改动。

第二轮，胜率为 10/40，经过中间的调整，可以从复盘数据清楚看到 Alpha 调整了跑位

策略，改为了会通过预测我们打角的方向，决定是否跑位，如果预测我们打原来的方向，则不跑位，这样很可能就能完全省下一次跑位+迎球的体力消耗，如果预测我们打对面角，则跑位到中间，虽然 Alpha 组的预测不是每次都预测得准，但是有很大的概率都预测准了，所以我们在第二轮的对战中被扼制得很精准，因为我们打角的策略是两边都可能打（假设对方都跑位到中间），而对方都是打对面的角，所以每次只跑位到 250000 的安全范围边缘是不科学的。

第三轮，胜率为 6/40，在本轮，Alpha 队选择没有修改代码，仍旧采用第二轮的跑位策略，而当时我们小组求胜心切，但由于修改时间短，未能分析出第一轮获胜但第二轮落后的原因，最后还是未能扭转局势。所以我们最后还是败给了 Alpha，止步四强外。

4 实习过程总结

4.1 分工与合作

小组分工：
代码编写：张峻伟（技术大腿）、贺麒霖、汪其霖、刘璇
平台解读：李嘉冕、马嘉成、刘立洋
报告撰写：刘立洋、马嘉成、汪其霖、刘璇
代码测试：李嘉冕
额外加分：张峻伟 2 分，刘立洋 1 分

全组同学精诚团结，克服大三狗科研、实习繁忙的困难，每次组会都在深夜召开，以 10 次组会一举夺得开组会最多小组。游戏代码的策略基本上由大家共同讨论得出，随后大家分为两个大组：平台规则解读和代码实现。在大家的共同努力下，最终拿出了一份还不错的代码。

（摄影：李嘉冕）



4.2 经验与教训

组会数量可以适当减少，有一些工作可以靠个人完成，不必开会解决。

4.3 建议与设想

希望以后的游戏可以尽早把规则敲定，以免中途需要不停的调整代码适应新规则。

5 致谢

1）谢谢陈斌老师一学期细致的教诲和耐心的答疑解惑，是您为我们打开了 python 语言下的神奇世界，谢谢助教老师的尽心尽力，是你们为我们批改代码，为我们跑程序
2）谢谢每一位参加小组赛的对手，是你们让我们有了友谊赛之后的成长
3）谢谢组长刘立洋的组织监督，谢谢每一位小组成员的尽心付出，让我们共同有了这次愉快的旅程

6 参考文献

没有参考，全靠大脑。



INDIA

组长：闵靖涛

组员：代国峰，韩潇霖，鲁洪亮，衣可心，孙唯一，常海宁

小组赛战绩：S2

决赛战绩：八强







INDIA



数据结构与算法课程实习作业报告

（闵靖涛*、代国峰、鲁洪亮、衣可心、韩潇霖、孙唯一、常海宁）

摘要：

小组算法以估值函数为核心，将其应用在三个主要动作跑位、击球、获取道具上，并根据基础的博弈常识，选择对自己消耗最小而使得对方消耗最大的战术进行对战。通过字典灵活调试各种权重参数，可以灵活应对各种类型的对手。

关键字：

估值函数 贪心 博弈 学习

1 算法思想

1.1 总体思路

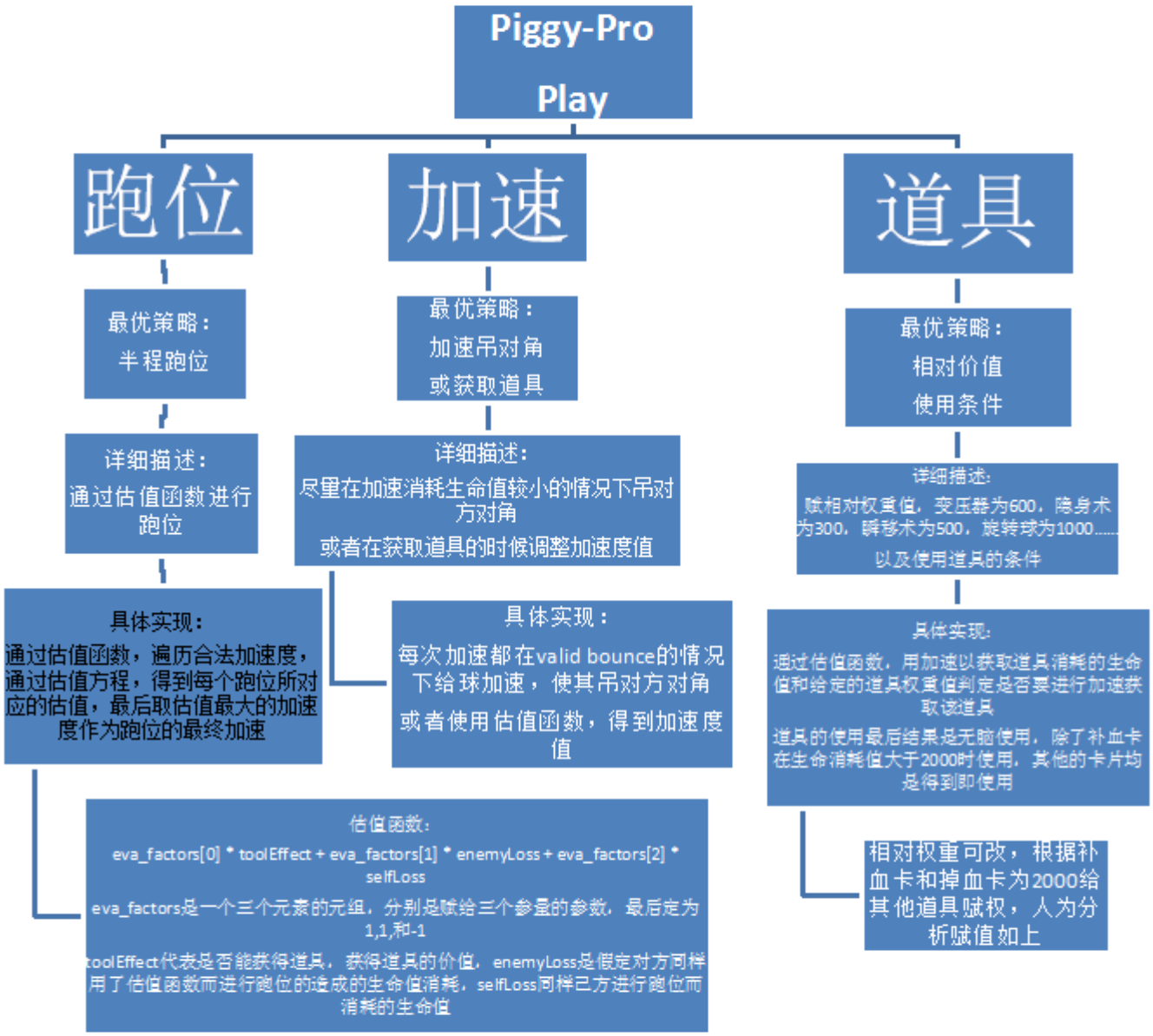
第一次组会，在全体小组成员头脑风暴后，代码组成员即开始进行更加细致的算法策略分析与整理。算法大致包括三个部分：跑位策略、迎球时加速策略、道具的获取与使用策略。

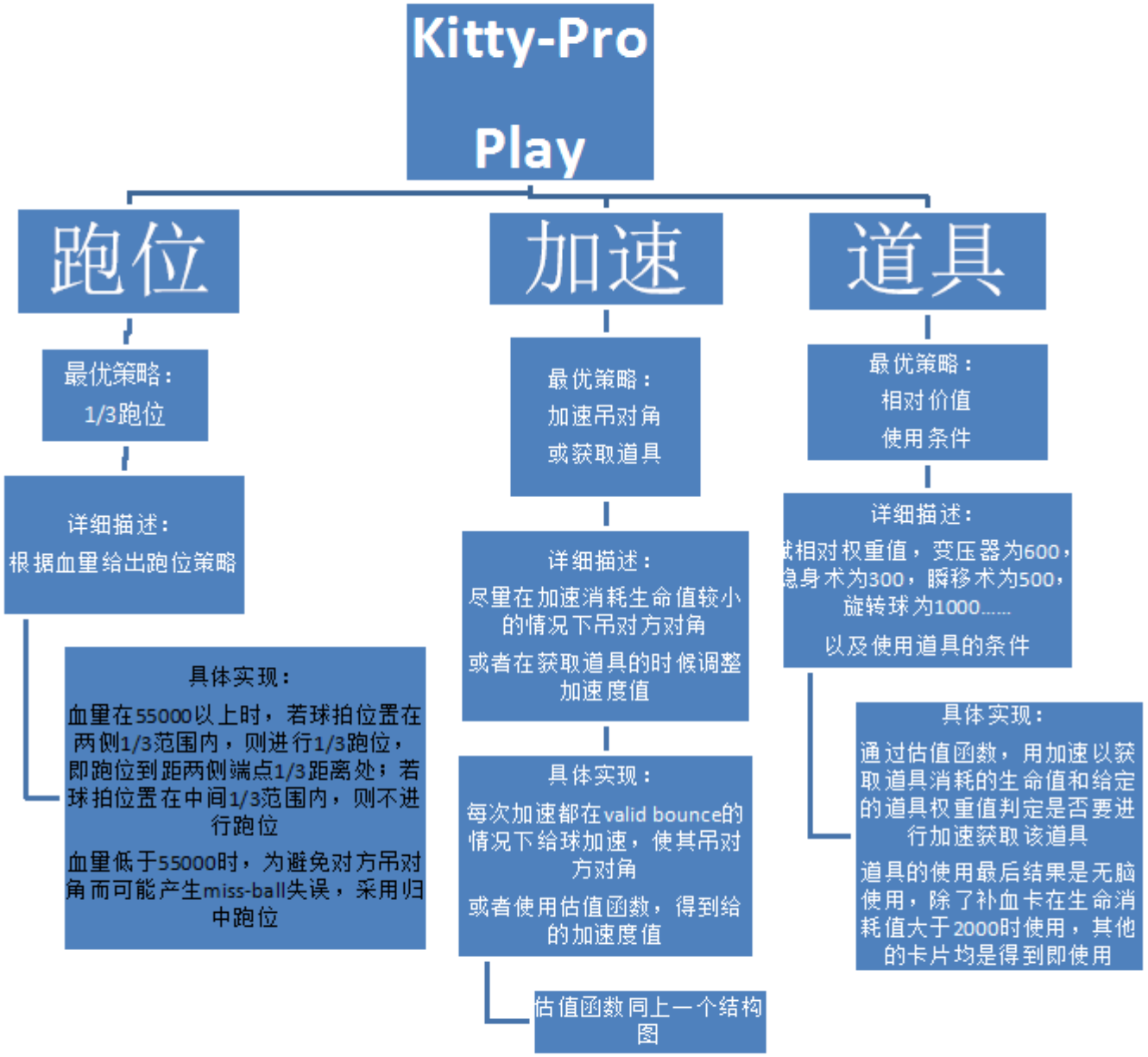
关于跑位的策略，我们最先写出来的陪练代码 intelligent 的跑位是半程跑位，即假设对方没有对球进行加速而直接击球的落点为终点，进行半程跑位。这种跑位过于简陋。此后写出的最初对战版本 INDIA 的跑位是归中跑位，即不论如何，在击球后都跑到 Y 轴的中间位置。随后的 plus_SPIN_attention 和 betaPingPong 的跑位同样是归中跑位。再之后的 last_update 的跑位则是利用估值函数计算得到对方最可能的击球落点，量化获取道具的成本，再综合击球加速的成本，得到的对自己最大优势的跑位和击球加速值。Aquinas 则增加了利用 ds 数据进行学习的功能，其跑位是通过学习对方击球落点数据进行的跑位。Piggy、Piggy-Pro、Kitty、Kitty-Pro 分别是以 last_update 和 plus_SPIN_attention 为基底进行更加细化的策略改进得到的程序。

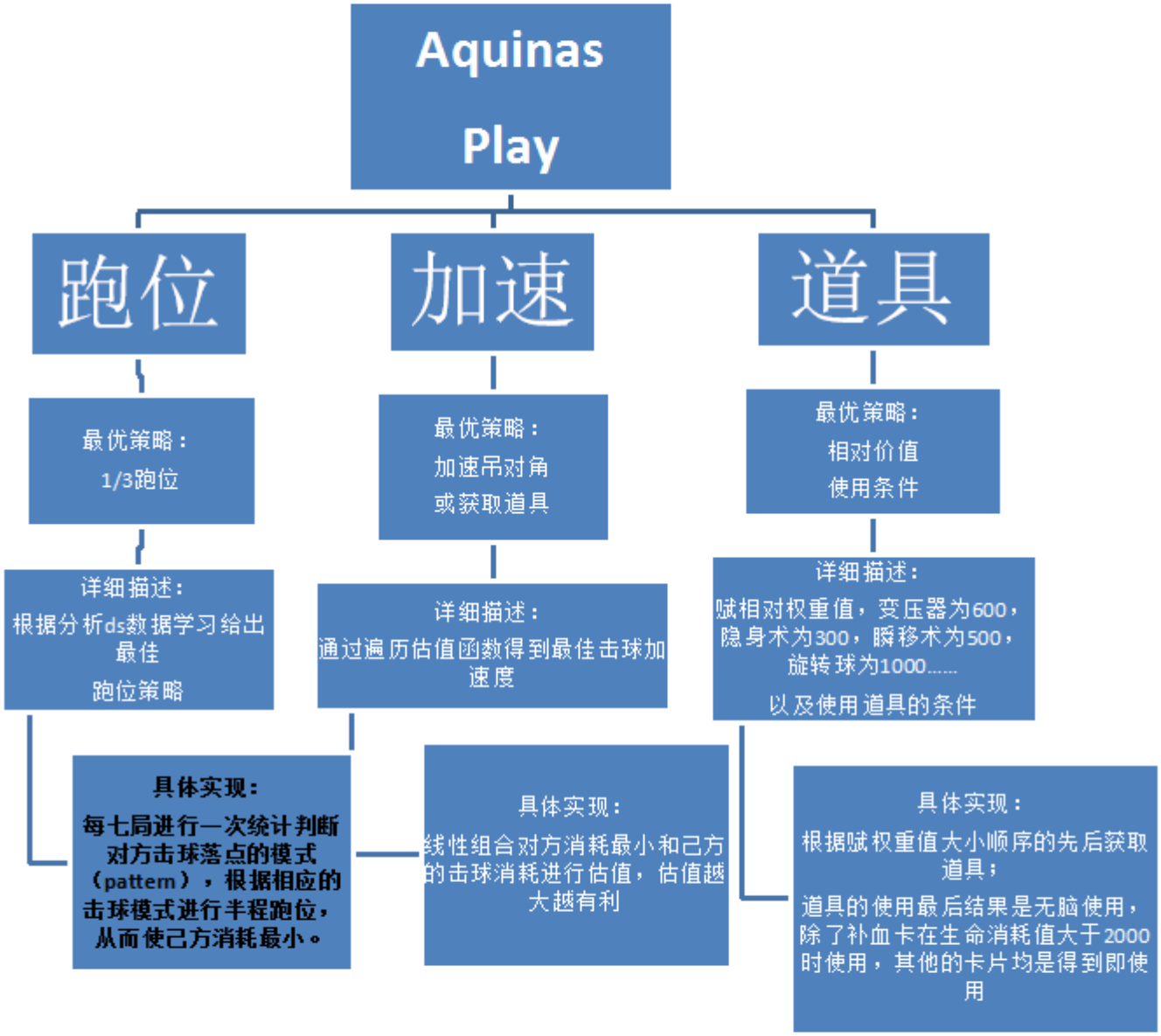
以下即对三个成型的算法，Piggy-Pro、Kitty-Pro、Aquinas 进行详细的算法流程图解释。

1.2 算法流程图




Piggy-Pro Play
跑位
加速
道具
最优策略：
半程跑位
详细描述：
通过估值函数进行跑位
具体实现：
通过估值函数，遍历合法加速度，通过估值方程，得到每个跑位所对应的估值，最后取估值最大的加速度作为跑位的最终加速
估值函数：
eva_factors[0] * toolEffect + eva_factors[1] * enemyLoss + eva_factors[2] * selfLoss
eva_factors是一个三个元素的元组，分别是赋给三个参量的参数，最后定为1,1,和-1
toolEffect代表是否能获得道具，获得道具的价值，enemyLoss是假定对方同样用了估值函数而进行跑位的造成的生命值消耗，selfLoss同样己方进行跑位而消耗的生命值
最优策略：
加速吊对角
或获取道具
详细描述：
尽量在加速消耗生命值较小的情况下吊对方对角
或者在获取道具的时候调整加速度值
具体实现：
每次加速都在valid bounce的情况下给球加速，使其吊对方对角
或者使用估值函数，得到加速度值
最优策略：
相对价值
使用条件
详细描述：
赋相对权重值，变压器为600，隐身术为300，瞬移术为500，旋转球为1000……
以及使用道具的条件
具体实现：
通过估值函数，用加速以获取道具消耗的生命值和给定的道具权重值判定是否要进行加速获取该道具
道具的使用最后结果是无脑使用，除了补血卡在生命消耗值大于2000时使用，其他的卡片均是得到即使用
相对权重可改，根据补血卡和掉血卡为2000给其他道具赋权，人为分析赋值如上


Kitty-Pro
Play
跑位
最优策略：
1/3跑位
详细描述：
根据血量给出跑位策略
具体实现：
血量在55000以上时，若球拍位置在两侧1/3范围内，则进行1/3跑位，即跑位到距两侧端点1/3距离处；若球拍位置在中间1/3范围内，则不进行跑位
血量低于55000时，为避免对方吊对角而可能产生miss-ball失误，采用归中跑位
加速
最优策略：
加速吊对角
或获取道具
详细描述：
尽量在加速消耗生命值较小的情况下吊对方对角
或者在获取道具的时候调整加速度值
具体实现：
每次加速都在valid bounce的情况下给球加速，使其吊对方对角
或者使用估值函数，得到给的加速度值
估值函数同上一个结构图
道具
最优策略：
相对价值
使用条件
详细描述：
相对权重值，变压器为600，隐身术为300，瞬移术为500，旋转球为1000……
以及使用道具的条件
具体实现：
通过估值函数，用加速以获取道具消耗的生命值和给定的道具权重值判定是否要进行加速获取该道具
道具的使用最后结果是无脑使用，除了补血卡在生命消耗值大于2000时使用，其他的卡片均是得到即使用



对于 Aquinas 学习算法更细致的一些阐述：

- 跑位的具体实现：当发现对手的击球落点存在较明显规律时进行对手模式跑位，否则始终进行稳定跑位

  ① **对手模式跑位：**每七局进行统计一次对方击球的落点，从而分析出对方击球的模式（pattern）。如果对方的模式是打对角，则每次的跑位为半程跑位，即进

行归中跑位；如果对方击球模式是打任意角，则进行 1/4 跑位，即每次向中点方向跑到 1/4 距离处；如果对方的击球模式是打回角，则进行 1/10 跑位，即向中点方向跑到 1/10 距离处，其实此种情况下本不应进行跑位，但出于谨慎，进行 1/10 跑位。

判断对方击球模式的函数的判断参数包括计算出的落点的方差和均值。如果落点的方差较大，则考虑对方每次打角都是上一次打角的对角，称为模式一，打对角；如果方差不够大，则是简单的打角，称为模式二，打任意角；在方差较小的情况下，看均值靠近哪一个角，则判定为对方习惯打的角，称为模式三，打回角。

② **稳定跑位**：稳定点指的是，在血量低于 55000 的时候稳定区域的边界。稳定区域指的是这样一个区域，跑位到这个区域以后，在该血量限制下，下一次击球依然可以达到任意一点。稳定跑位即指，如果在稳定区域外，就跑到稳定区域；如果在稳定区域内，就跑到稳定区域的中点。

- **道具的获取**：综合考虑获得道具的成本和道具的相对价值，决定是否去主动获取道具；获取道具的顺序为补血卡>旋转球>变压器>掉血卡>瞬移术>隐身术。
- **道具的使用**：同样有一个应用的先后，一般来说依然是有道具即使用，补血卡有耗血量是否小于 2000 的判断，其它卡片有使用的先后顺序问题，先后顺序由人为分析给出；道具的应用顺序为旋转球>补血卡>变压器>掉血卡>瞬移术>隐身术。
- **击球策略**：遍历 valid bounce 里所有可能的加速度值，选取估值函数最大的加速度值作为击球的加速度。
- **估值函数**：此处的估值函数仅由两部分组成，一部分是己方损耗，一部分是对方最小损耗。己方损耗即己方迎球时的加速度耗费生命值，同时考虑了对方是否使用变压器；对方最小的损耗即默认对方进行的是最佳的半程跑位。系数依然是 1 和-1，二者权重相等。此种估值函数较为保守。

1.3 算法运行时间复杂度分析

Piggy-Pro 算法运行时间的主要构成在于估值函数，估值函数以 20 为梯度遍历了所有的加速度值，并且假定对方同样有这样一个估值函数，并同样用来遍历估值，于是在跑位函数中遍历了我方不同跑位时对方最大价值击球的价值，所以算法复杂度是 $n\times n=n^2$，即 $O(n^2)$。

Kitty-Pro 同样使用了估值函数，但是只进行一轮，用于己方的发球加速和道具的获取，算法复杂度应为 O（n）。

最后的 Aquinas 则同样有一个估值函数，用于分析对比，得出己方的击球时的加速度值和是否获取道具，算法复杂度为 O(n)；另外一个主要构成的部分是判断对方模式而进行的计算，这里的计算包括方差均值，这个复杂度同样也是 O（n）。因此 Aquinas 的算法复杂度为 n+n=2n，所以其大 O 数量级依然为 O(n)。

2 程序代码说明

2.1 数据结构说明

算法中极大程度地开发了字典和元组两种基本数据类型的应用，使得参数的修改更加容易。

2.2 函数说明

三个程序均有的函数：

① **发球函数：**

```
def serve( op_side, ds: dict) -> tuple:
    tick_step = (DIM[1] - DIM[0]) // BALL_V[0]
    Vy = 3 * (DIM[3] - DIM[2]) // tick_step // 2
    return ((DIM[2] + DIM[3]) // 2, Vy)
```

因为发球不消耗生命值，所以每次发球都采用打对角的策略。

② **Play 函数：**

```
def play(tb: TableData, ds: dict) -> RacketAction:
    retCard = getCard(tb, ds)
    ball_Vy = tb.ball['velocity'].y
    ball_posX, ball_posY = tb.ball['position'].x, tb.ball['position'].y
    tools = tb.cards['cards']
    op_side_X = tb.op_side['position'].x
    op_side_Y = tb.op_side['position'].y
    op_active_card = tb.op_side['active_card'][1]
    ENEMY_POSITION_Y = get_opside_pos(tb, ball_Vy, ball_posX, ball_posY, op_side_Y)
    retAcc = getAcc(tb, ball_Vy, ball_posX, ball_posY, tools, ENEMY_POSITION_Y, op_active_card)[0]
    posVector = positioning(tb, ds)
    return RacketAction(tb.tick, tb.ball['position'].y - tb.side['position'].y, retAcc, posVector, retCard[0], retCard[1])
```

此函数为算法的主函数，会调用其他的子函数，完成跑位、迎球、加速、获取和使用道具等动作。

③ **获取道具函数：**

```
def hit_cards(tb, ds):
    retVys = []
    cards = [card.code for card in tb.cards['cards']]  # cards is list of string
    velocity = tb.ball['velocity'].y
    for card_name in PICK_PRIORITY:
        if card_name in cards:
            for card in tb.cards['cards']:
                if card.code == card_name:
                    pos = card.pos  # Vector
                    step = abs(pos.x) / BALL_V[0]
                    retVys.append((card_name, (pos.y - tb.ball['position'].y) / step - velocity))
                    retVys.append((card_name, (DIM[3] - tb.ball['position'].y + DIM[3] - pos.y) / step - velocity))
                    retVys.append((card_name, (DIM[3] - tb.ball['position'].y + DIM[3] - DIM[2] + pos.y - DIM[2]) / step - velocity))
                    retVys.append((card_name, (-1) * (tb.ball['position'].y - DIM[2] + pos.y - DIM[2]) / step - velocity))
                    retVys.append((card_name, (-1) * (tb.ball['position'].y - DIM[2] + DIM[3] - DIM[2] + DIM[3] - pos.y) / step - velocity))
    return retVys
```

此函数通过桌面上道具的位置参数，计算出此时改变多少的加速度可以获取道具，以及选择获取什么道具。获取道具的优先级顺序之前已经提到。

④ **使用道具函数：**

```
def getCard(tb: TableData, ds: dict) -> tuple:
    for card in APP_PRRORITY:
        if card in tb.side['cards']:
            if card == CARD_INCL and tb.side['life'] < RACKET_LIFE - CARD_INCL_PARAM:
                return tb.side, card
            elif card == CARD_DECL or card == CARD_AMPL or card == CARD_SPIN:
                return tb.op_side, card
            elif card == CARD_TLPT or card == CARD_DSPR:
                return tb.side, card
    return None, None
```

此函数用来使用道具，当消耗生命值小于 2000 时即使用补血卡，其他条件下则一律无脑使用其他获得的道具。在有大于一件道具存在的情况下，则按照使用道具的优先级顺序使用道具，顺序在之前也已经提到。

⑤ **决定加速度大小函数：**

此函数在 Piggy-Pro 和 Kitty-Pro 中一样，但在 Aquinas 中不同，差别在于有无道具元素参与。

首先看 **Piggy-Pro** 和 **Kitty-Pro** 的函数：

```
def getAcc(tb,ball_Vy:int, ball_posX:int,ball_posY:int,tools:Card, enemy_posY, op_active_card=None) -> int:
    tick_step = (DIN[1] - DIN[0]) // BALL_V[0]
    Pos_Y = ball_posY
    Vy = ball_Vy
    upMin = int((DIN[3] - Pos_Y) // tick_step - Vy)
    upMax = int((2 * (DIN[3] - DIN[2]) + DIN[3] - Pos_Y) // tick_step - Vy)
    downMax = int((-1) * (Pos_Y - DIN[2]) // tick_step - Vy)
    downMin = int((-1) * (2 * (DIN[3] - DIN[2]) + Pos_Y - DIN[2]) // tick_step - Vy)

    newUpmin, newUPmax, newDownmin, newDownmax = upMin + 1, upMax - 1, downMin + 1, downMax - 1
    if op_active_card == CARD_SPIN:  # 如果对方使用旋转球，则调整合法加速值范围
        if (upMin + 1) < 0 and (upMax - 1) < 0:
            newUpmin, newUPmax = upMin + 1, (upMax - 1) * 2
        elif (upMin + 1) > 0 and (upMax - 1) > 0:
            newUpmin, newUPmax = (upMin + 1) * 2, upMax - 1
        if (downMin + 1) < 0 and (downMax - 1) < 0:
            newDownmin, newDownmax = downMin + 1, (downMax - 1) * 2
        elif (downMin + 1) > 0 and (downMax - 1) > 0:
            newDownmin, newDownmax = (downMin + 1) * 2, downMax - 1
```

```
    # 捡道具
    cards_deltaV = hitCards_delta_Vy(ball_Vy, ball_posX, ball_posY, tools)
    MINVALUE = -100000
    FINALCHOICE = None
    for var in cards_deltaV:
        if int(var[1]) in range(newUpmin, newUPmax) or int(var[1]) in range(newDownmin, newDownmax):
            value = evalueate(tb, ball_Vy, ball_posY, var[0], var[1], enemy_posY)
            if value > MINVALUE:
                MINVALUE = value
                FINALCHOICE = var
    # 不捡道具
    MAX_VALUE = -90000
    for dt in range(newUpmin, newUPmax, 10):
        value = evalueate(tb, ball_Vy, ball_posY, 'NOCARD', dt, enemy_posY)
        if value > MAX_VALUE:
            MAX_VALUE = value
            retV = dt
    for dt in range(newDownmin, newDownmax, 10):
        value = evalueate(tb, ball_Vy, ball_posY, 'NOCARD', dt, enemy_posY)
        if value > MAX_VALUE:
            MAX_VALUE = value
            retV = dt
    # 综合考虑选择估值较大的加速值
    if MINVALUE > MAX_VALUE:
        return int(FINALCHOICE[1]), MINVALUE
    else:
        return retV, MAX_VALUE
```

此函数为以上两个成熟算法中决定加速度大小的函数，在 valid bounce 的加速度区间中时，以 20 位一个单位遍历整个区间，使用此加速度下的参数进行估值，选取估值最大的加速度即为我们选择的最优加速度。

以下为 **Aquinas** 的获取最优加速度的函数：

```
# 求加速度函数，输入tb，ds，输出加速度
def get_acc(tb, ds):
    min_up, max_up = valid_bounce(tb.ball['position'].y, 1)
    min_down, max_down = valid_bounce(tb.ball['position'].y, -1)
    acc_min_up, acc_max_up = min_up - tb.ball['velocity'].y, max_up - tb.ball['velocity'].y
    acc_min_down, acc_max_down = min_down - tb.ball['velocity'].y, max_down - tb.ball['velocity'].y
    if tb.op_side['active_card'][1] == CARD_SPIN:
        acc_min_up = acc_min_up * 2 + 1
        acc_max_up = acc_max_up * 2 - 1
        acc_min_down = acc_min_down * 2 + 1
        acc_max_down = acc_max_down * 2 - 1
        choice_up = None
        choice_up_value = float('-inf')
        for acceleration in range(acc_min_up, acc_max_up, 20):
            value = evaluate(tb, ds, acceleration)
            if value > choice_up_value:
                choice_up_value, choice_up = value, acceleration
        choice_down = None
        choice_down_value = float('-inf')
        for acceleration in range(acc_min_down, acc_max_down, 20):
            value = evaluate(tb, ds, acceleration)
            if value > choice_down_value:
                choice_down_value, choice_down = value, acceleration
        choice, val = (choice_up, choice_up_value) if choice_up_value > choice_down_value else \
            (choice_down, choice_down_value)
        return choice
```

```
else:
    choice_up = None
    choice_up_value = float('-inf')
    for acceleration in range(acc_min_up, acc_max_up, 20):
        value = evaluate(tb, ds, acceleration)
        if value > choice_up_value:
            choice_up_value, choice_up = value, acceleration
    choice_down = None
    choice_down_value = float('-inf')
    for acceleration in range(acc_min_down, acc_max_down, 20):
        value = evaluate(tb, ds, acceleration)
        if value > choice_down_value:
            choice_down_value, choice_down = value, acceleration
    choice, val = (choice_up, choice_up_value) if choice_up_value > choice_down_value else (choice_down, choice_down_value)
    return choice
```

此函数大体上和 Piggy-Pro 以及 Kitty-Pro 的思路一致，唯一的区别在于估值函数中，道具因素的参与考虑与否。接下来进行估值函数的介绍。

⑥ **估值函数：**

Piggy-Pro 和 **Kitty-Pro**：

```
def evalueate(tb, ball_Vy:int, ball_posY:int, card, deltaV, enemy_posY) -> int:
    eva_factors = [1, 1, -1]
    if tb.op_side['life'] < 55000 and tb.op_side['life']>26000 and tb.side['life'] > tb.op_side['life']:
        eva_factors[1] = 1 * 55000 / tb.op_side['life']
    elif tb.side['life'] < 55000 and tb.side['life'] < tb.op_side['life']:
        eva_factors[2] = -1 * 55000 / tb.side['life']
    elif tb.op_side['life'] < 26000:
        eva_factors[1] = 100
    toolEffect = TOOL_VALUE[card]
    selfLoss = (abs(deltaV) ** 2) / (FACTOR_SPEED ** 2)
    opponentY = get_opp_posY(ball_Vy, ball_posY, deltaV)
    enemyLoss = (abs(enemy_posY - opponentY) ** 2) / (FACTOR_DISTANCE ** 2)
    return int(eva_factors[0] * toolEffect + eva_factors[1] * enemyLoss + eva_factors[2] * selfLoss)
```

此函数主要用于对各个加速度可能产生的结果进行估值，估值越大，对己方优势越大，因此选取使得估值函数取最大值时的加速度值作为最优加速度值。用于估值的方程为 eva_factors[0] * toolEffect + eva_factors[1] * enemyLoss + eva_factors[2] * selfLoss。

eva_factors 是一个三个元素的元组，分别是赋给三个参量的参数，最后定为 1,1,和 -1。toolEffect 代表是否能获得道具，获得道具的价值，enemyLoss 是假定对方同样用了估值函数而进行跑位的造成的生命值消耗，selfLoss 为己方进行跑位而消耗的生命值。

在 **Aquinas** 中：

```
def evaluate(tb, ds, acc):
    mid_point = breadth // 2
    time = tb.tick // (2 * tick_step) + 1
    lost = (abs(acc) ** 2 // FACTOR_SPEED ** 2) * \
           (CARD_AMPL_PARAM if tb.op_side['active_card'][1] == CARD_AMPL else 1)
    acceleration = acc * (CARD_SPIN_PARAM if tb.op_side['active_card'][1] == CARD_SPIN else 1)
    oppo_position = opposite_calculate(tb.ball['position'].y, tb.ball['velocity'].y + acceleration)
    if time > 1:
        distance = oppo_position - ds[time - 1].op_side['position'].y
    else:
        distance = oppo_position - mid_point
    op_lost_min = 2 * (abs(distance // 2) ** 2 // FACTOR_DISTANCE ** 2)
    sum_value = fac_1 * op_lost_min - fac_2 * lost
    return sum_value
```

根据己方加速消耗和认为对方进行最优跑位而消耗的最小生命值的线性组合，系数同样绝对值都为 1。

Piggy-Pro's feature functions:

① 判断对方跑位函数：

```
def get_opside_pos(tb, ball_Vy, ball_posX, ball_posY, opside_posY:int) :
    pos0 = opside_posY
    pos1 = (DIM[3] - DIM[2]) // 2
    notools_Acc = getAcc(tb, ball_Vy, ball_posX, ball_posY, [], opside_posY)[0]
    pos2 = get_opp_posY(ball_Vy, ball_posY, notools_Acc)
    pos = pos0*enemy_pos_factors[0] + pos1*enemy_pos_factors[1] + pos2*enemy_pos_factors[2]
    return pos
```

假定对方的跑位是由我们设想出来的三种可能跑位的线性组合，我们根据推断出的跑位位置击球，给予对方较大打击。三种可能的跑位方式是不跑位、归中跑位、对方同样用了己方的估值函数而进行的跑位，最后得出的结果为三者的线性组合。

此函数中三者相对参数可调，我们最后定的参数为 0.6447368421052632、0.046052631578947366、0.3092105263157895，得出这三个参数的方法为：假定每中跑位有一个权重，我们在（0,1）中随机赋值，两两对打，最后获胜的即为最佳参数。

② 跑位函数：

```
def most_possible_return_y(tb: TableData, ds: dict) -> int:
    temp_tb = copy.deepcopy(tb)
    for var in hitCards_delta_Vy(temp_tb.ball['velocity'].y, temp_tb.ball['position'].x, temp_tb.ball['position'].y,\
                                 temp_tb.side['cards']):
        for temp_card in temp_tb.cards['cards']:
            if temp_card.code == var[0]:
                temp_tb.cards['cards'].remove(temp_card)
    for card in temp_tb.cards['cards']:
        card.pos.x = -card.pos.x
    tick_step = (DIM[1] - DIM[0]) // BALL_V[0]
    temp_ball_velosity_y = temp_tb.ball['velocity'].y + getAcc(tb, temp_tb.ball['velocity'].y, temp_tb.ball['position'].x,\
        temp_tb.ball['position'].y, temp_tb.side['cards'], temp_tb.op_side['position'].y, temp_tb.op_side['active_card'])[0]
    if temp_ball_velosity_y > 0 and temp_ball_velosity_y * tick_step < 2 * DIM[3] - temp_tb.ball['position'].y or\
                    temp_ball_velosity_y < 0 and temp_ball_velosity_y * tick_step > -1 * DIM[3] - temp_tb.ball['position'].y:
        temp_ball_velosity_y = -temp_ball_velosity_y
    else:
        pass
    temp_tb.ball['velocity'].y = int(temp_ball_velosity_y)
    temp_tb.ball['position'].y = int(reflect_calculate(temp_tb.ball['position'].y, temp_ball_velosity_y)[1])
    if getCard(tb, ds) == CARD_SPIN:
        temp_tb.op_side['active_card'] = CARD_SPIN
    MIN_MAX_VALUE = 100000
    MIN_MAX_VALUE_ACC =10000

    for y in (DIM[2], DIM[3]):
        temp_tb.op_side['position'].y = y
        if getAcc(tb, temp_tb.ball['velocity'].y, temp_tb.ball['position'].x, temp_tb.ball['position'].y, temp_tb.side['cards'],\
                  temp_tb.op_side['position'].y, temp_tb.op_side['active_card'])[1] < MIN_MAX_VALUE:
            MIN_MAX_VALUE = getAcc(tb, temp_tb.ball['velocity'].y, temp_tb.ball['position'].x, temp_tb.ball['position'].y,\
                                   temp_tb.side['cards'], temp_tb.op_side['position'].y, temp_tb.op_side['active_card'])[1]
            MIN_MAX_VALUE_ACC = getAcc(tb, temp_tb.ball['velocity'].y, temp_tb.ball['position'].x, temp_tb.ball['position'].y,\
                                       temp_tb.side['cards'], temp_tb.op_side['position'].y, temp_tb.op_side['active_card'])[0]
    return reflect_calculate(temp_tb.ball['position'].y, temp_tb.ball['velocity'].y + MIN_MAX_VALUE_ACC)[1]
```

跑位函数即通过判断对方最有可能将击球到己方的位置，及对方会将球击到对我方最不利的地方，或者是我方不跑位，或者是我方进行归中跑位，三种情况加权算的我方我方可以进行的最优的半程跑位，以此结果进行跑位。在第一种情况下，可能会出现对方为了获取道具而改变加速度值，但此种情况随机，且出现概率占比较小，因此可认为主要都是对方将球击倒对我方最不利的位置上来。

Kitty-Pro's feature functions:

① 跑位函数：

```
def positioning(tb: TableData, ds: dict) -> int:
    if tb.side['life'] > 55000:
        if tb.ball['position'].y < ((DIM[3]-DIM[2])//3):
            return (DIM[3]-DIM[2])/3-tb.ball['position'].y
        elif tb.ball['position'].y > ((DIM[3]-DIM[2])*2//3) :
            return (DIM[3]-DIM[2])*2/3-tb.ball['position'].y
        else:
            return 0
    else:
        return int((DIM[3] - DIM[2]) // 2 - tb.ball['position'].y)
```

此函数控制 Kitty-Pro 的跑位，血量在 55000 以上时，若球拍位置在两侧 1/3 范围内，则进行 1/3 跑位，即跑位到距两侧端点 1/3 距离处；若球拍位置在中间 1/3 范围内，则不进行跑位；血量低于 55000 时，为避免对方吊对角而可能产生 miss-ball 失误，采用归中跑位。

Aquinas' feature functions：

① 稳定跑位函数：

```
def stable_positioning(tb, ds, delta_life):
    mid_point = (DIM[2] + DIM[3]) // 2
    tri_1 = (DIM[2] + DIM[3]) // 3
    tri_2 = DIM[3] - tri_1
    run_life = -(abs(tri_1)) ** 2 // FACTOR_DISTANCE ** 2
    run_life2 = -(abs(500000)) ** 2 // FACTOR_DISTANCE ** 2
    if int(((tb.side['life'] + delta_life + run_life) / RACKET_LIFE) * BALL_V[0]) * tick_step > tri_2:
        if tb.ball['position'].y < tri_1:
            destine = tri_1
        elif tb.ball['position'].y > tri_2:
            destine = tri_2
        else:
            destine = mid_point
    elif int(((tb.side['life'] + delta_life + run_life2) / RACKET_LIFE) * BALL_V[0]) * tick_step > mid_point:
        div_2 = int(((tb.side['life'] + delta_life + run_life2) / RACKET_LIFE) * BALL_V[0]) * tick_step
        div_1 = DIM[3] - div_2
        if tb.ball['position'].y > div_2:
            destine = div_2
        elif tb.ball['position'].y < div_1:
            destine = div_1
        else:
            destine = mid_point
    else:
        destine = mid_point
    movement = destine - tb.ball['position'].y
    return movement
```

这是在没有根据 ds 数据分析出对方有什么比较明显的套路时采用的跑位。稳定点、稳定区间以及稳定跑位的概念均已经在上文中有解释，此处不再赘述。

② 分析对手击球模式函数：

```
def pattern_judgement(tick, ds):
    case_code = None
    current = tick // (2 * tick_step) + 1
    pos_list = [ds[time].ball['position'].y for time in range(current - 8, current)]
    if mean(pos_list) >= (breadth / 2) ** 2 * 0.80:
        if adjacent_judge_pos(pos_list) >= breadth ** 2 * 0.88:
            case_code = 'Opposite Corner'
        elif breadth ** 2 * 0.45 <= adjacent_judge_pos(pos_list) <= breadth ** 2 * 0.65:
            case_code = 'Random Corner'
    elif mean(pos_list) <= (breadth / 2) ** 2 * 0.20:
        if average(pos_list) >= breadth * 17 // 20 or average(pos_list) <= breadth * 3 // 20:
            case_code = 'Original Corner'
    pos_list2 = [reflect_calculate(ds[time-1].ball['position'].y, ds[time-1].ball['velocity'].y + ds[time-1].side['acc'])[1]
                 for time in range(current - 8, current)]
    sub_list = [pos_list[i] - pos_list2[i] for i in range(len(pos_list))]
    if mean(sub_list) <= 90000 ** 2:
        case_code = 'Small Acceleration'
    return case_code
```

通过计算对手击球落点的方差、均值来判断对方的击球套路如何，具体情况在之前的算法分析中已有阐述。

③ **对手击球函数：**

```
if pattern is not None:
    max_move = judge_validity(tb, ds, delta_life)
    target = delta_dist + tb.ball['position'].y
    max_needed = max(target, breadth - target)
    if max_needed > max_move:
        delta_dist = stable_positioning(tb, ds, delta_life)
    else:
        pass
else:
    delta_dist = stable_positioning(tb, ds, delta_life)
return delta_dist
```

对手击球模式指根据分析对手击球模式函数的结果，判断对方击球是否有明显套路，从而进行半程跑位，使得跑位和击球消耗生命值最小。具体实现情况在上线的算法分析中同样已有阐述。

④ **汇总跑位函数：**

```
def positioning(tb, ds, time, acc, bat_move):
    if time == 1:
        ds[time].side['pattern_code'] = None
    elif (time - 3) % 7 == 0 and time > 7:
        ds[time].side['pattern_code'] = pattern_judgement(tb.tick, ds)
    else:
        ds[time].side['pattern_code'] = ds[time - 1].side['pattern_code']
    pattern_code = ds[time].side['pattern_code']
    delta_dist = pattern_positioning(pattern_code, tb, ds, acc, bat_move)
    return delta_dist
```

此函数用来每隔七局进行一次数据分析，得到对方实时跑位以及击球策略。并将结果汇总，以便直接在 play 函数中调用。

2.3 程序限制

小组程序的考虑较为全面，暂时未发现会使得程序出错的极端条件。因为算法的复杂度

最高为 $O(n^2)$，计算不算繁杂，也不会出现程序超时的情况。

综合来说，我们小组的算法考虑较为完善，程序的编写目前未出现 bug。

3 实验结果

3.1 测试数据

实验环境说明：

- 硬件配置：CPU: Intel® Core™ i7-6700HQ @ 2.60GHz; RAM: 8.00 GB
- 操作系统：Windows 10 版本 15063
- Python 版本：3.5.3

在不断开发代码的过程中，主力代码版本完成了从 1 代到 6 代的更迭，在 3 代以后从不同角度优化开发出了 Piggy、Kitty、Doggy 三个各具优点的小动物版本及之后的 pro 版本。陪练代码由最初暴力打角的 intelligent 进化为哲学家 Parmenides、Aquinas 以及最终未完成的 Descartes。主力代码一代代不断优化，通过代间的测试赛可以明显看出进步；陪练代码一般针对各代主力代码开发，在帮助主力算法进步方面发挥了重要的作用。此外 Aquinas 作为替补参赛选手参加了 8 强赛，同样值得肯定。

下面展示的测试赛数据包括了各代主力代码间的测试赛、主力代码与陪练间的测试赛以及参加的热身赛数据，其中组内测试赛数据均基于 100 局对打得出。

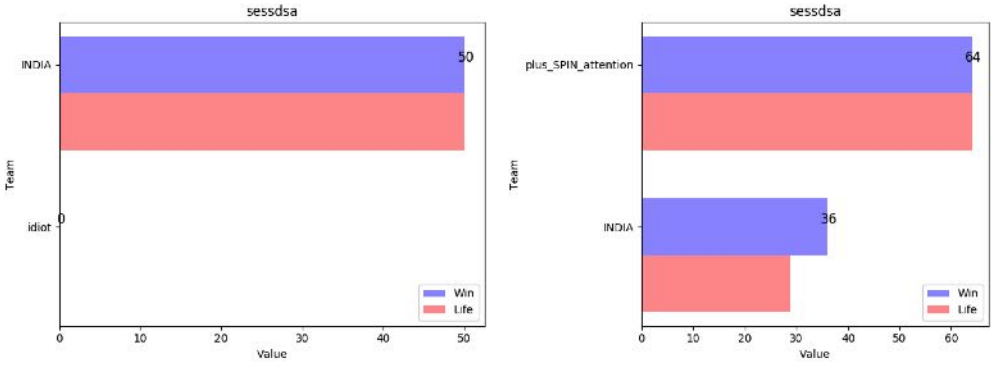
a.各代主要代码间的测试赛数据

1)1 代版本 INDIA vs idiot

INDIA 在作为 East 方时完成 50:0 的完胜，但作为 West 方时出现 bug 无法完赛，说明 1.0 版本具备一定的实力但仍需完善。

2)2 代版本 plus_SPIN_attention vs 1 代版本 INDIA

plus_SPIN_attention 以 64:36 完胜 INDIA，且在体力值上占据较大优势。

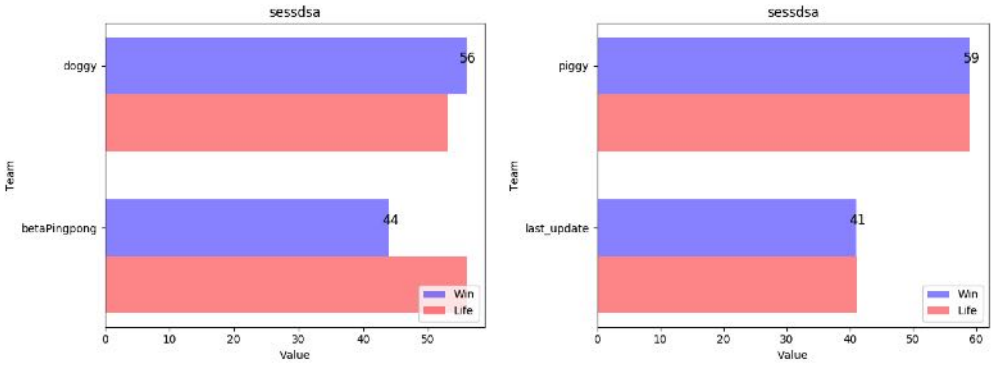


3)针对 betaPingpong 开发的 Doggy vs 2.5 代版本 betaPingpong

针对 betaPingpong 开发的 Doggy 实现了胜场上的优势，但在体力值等其他方面尚不如 betaPingpong。

4)3 代版本 last_update vs 基于 2 代版本开发的 4 代版本 piggy

last_update 以 41:59 不敌 piggy，但两算法各有优点。

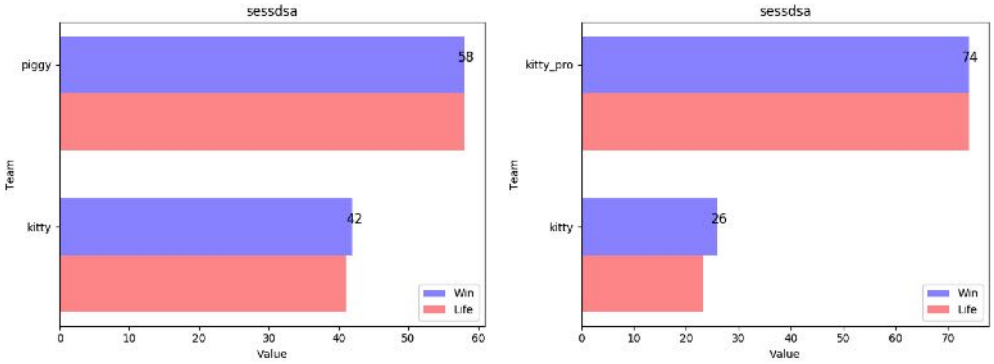


5)4 代版本 piggy vs kitty

综合战绩 piggy 以 58%胜率赢过 kitty，其中作为 West/East 均胜 29 局，胜率相同。

6)5 代版本 kitty_pro vs 四代版本 kitty

5 代版本 kitty_pro 以 74%的胜率碾压 4 代版本 kitty，融合了 3 代版本 last_update 后进步明显，在跑位方面有较大优势。

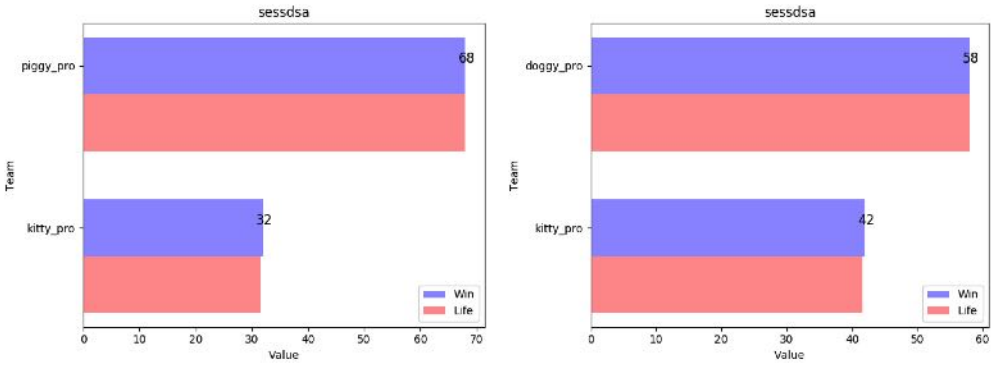


7)5 代版本两大主力算法 piggy_pro vs kitty_pro

综合战绩 piggy_pro 以 68%胜率赢过 kitty_pro，其中作为 West/East 均胜 34 局，胜率相同。

8)在比赛结束后完成的 6 代版本 doggy_pro vs 参赛版本 kitty_pro

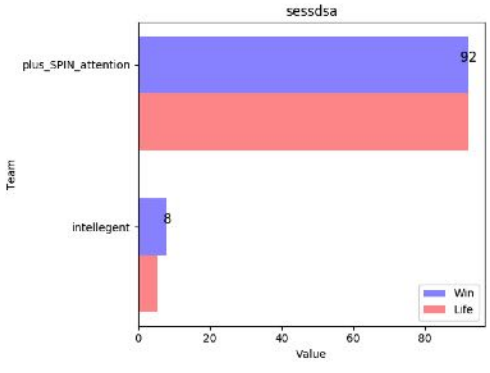
doggy_pro 以 58:42 压过 kitty_pro，且在各方面都表现出不俗的实力。



b.陪练代码与主力代码的测试赛数据

1)2 代主力 plus_SPIN_attention vs 暴力打角陪练 intelligent

intelligent 在应对 1 代主力 INDIA 时表现不俗，但 2 代主力代码以 92:8 完虐 intelligent，进步



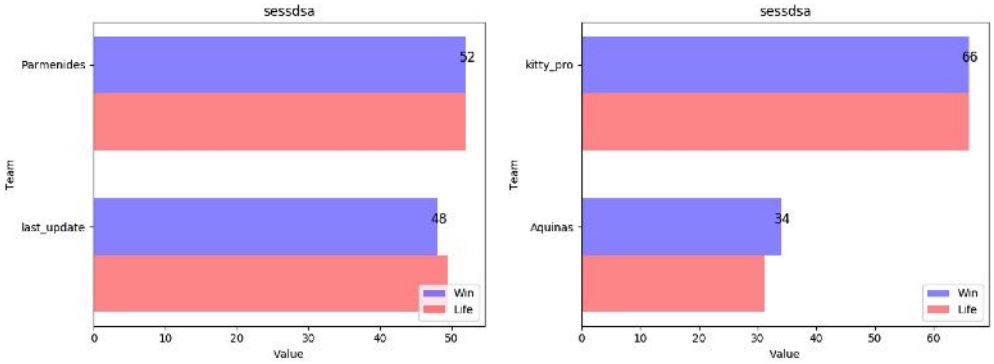
明显。

2)3 代主力 last_update vs 二代陪练 Parmenides

3 代主力 last_update 以微弱劣势败给 Parmenides，表现出跑位弱点，仍需改进加强。

3)参赛版本 kitty_pro vs 替补参赛陪练 Aquinas

虽然陪练 Aquinas 依旧被主力参赛选手 kitty_pro 大比分击败，但也为检验代码综合实力提供了大量数据。



c.热身赛数据

我们派出了 2 代主力 plus_SPIN_attention、3 代主力 last_update 和增加了低血量中位跑位策略的 3.1 版本 Augustinus 参加第一次热身赛，并且幸运地同时参加了上下半区的热身赛。第二次热身赛则派出了发挥不是很稳定的 5 代主力 piggy_pro，以检验其在实际比赛中的水平。

1)第一次热身赛上半区

3.1 版本主力 Augustinus 和 3.0 版本主力 last_update 分别以 179 和 131 的胜场数排名上半区前两名，胜率分别为 86.9%和 66.5%，表现出色。而 2 代版本 plus_SPIN_attention 也以 109 胜和 55.9%的胜率跻身中上游。所有输球的原因均为体力耗尽而出现 miss_ball，表明代码未出现低级击球失误且运算时间控制良好。

2)第一次热身赛下半区

下半区的竞争明显比上半区激烈，Augustinus 胜场数 126，胜率 70.4%，成绩明显下滑；last_update 取得了 84 胜，56%的胜率；plus_SPIN_attention 仅获得了 53 胜，35.3%的胜率。强劲的对手 Silly 以 141 胜，96.6%的完美发挥摘得下半区的热身赛桂冠；在胜率上超过我们的还有 FANCY、Hades、FMQED、shlcy 和 KILO 这 5 个选手。

3)第二次热身赛

总体而言排名依旧不容乐观，excited、sabac、lkjhg 和 ALPHA 这些对手在胜率上都超过了 piggy_pro，但 piggy_pro 依靠其实力依旧挤进了第一梯队。

3.2 结果分析

1 代主力版本 INDIA 横扫 idiot 是我们取得的第一个阶段性成果，此后针对暴力打角的陪练算法 intelligent 开发成功，并且完虐 INDIA。这帮助我们分析了 INDIA 在各方面的不足，尤其是采用的单一归中跑位，对 INDIA 的跑位等策略进行调整后，我们在 2 代版本 plus_SPIN_attention 实现了对暴力打角的优势。

此后主力版本逐步加入吃道具等功能，并针对对手回球预判增加了估值函数。通过估值函数遍历合法的加速度得到每个跑位对应的估值，取估值最大的点作为跑位。这帮助我们在

对打中尽可能减少跑位迎球的体力消耗，同时也能判断吃道具是否合适。第一次热身赛我们能在半区取得好成绩，估值函数奠定的基础功不可没。

第一次热身赛中我们仔细研究了在下半区表现出色的 Silly、FANCY 等对手，发现了 1/3 跑位具有一定优势。在此后的 piggy_pro、kitty_pro 等版本中都相应的吸收了这方面的策略，最终采用在两侧则跑到 1/3 处，在中间 1/3 段则不跑位的跑位策略。此后的测试结果表明这确实是一种比较优秀的跑位策略，但在一些细节处理上仍需加强。

第三代陪练 Aquinas 加入了从 ds 中读取存档数据，每 7 回合分析一次对手的击球策略的历史数据分析函数，帮助进行跑位的安排。但在内部测试时发现对手的击球策略很难直接归入打对角、打原角、较小加速等类型中，历史数据在分析中能得到策略的回合数较少，故不能显著提升实力。另一方面由于时间紧张，所以这部分学习的代码没有进入到主力算法中。

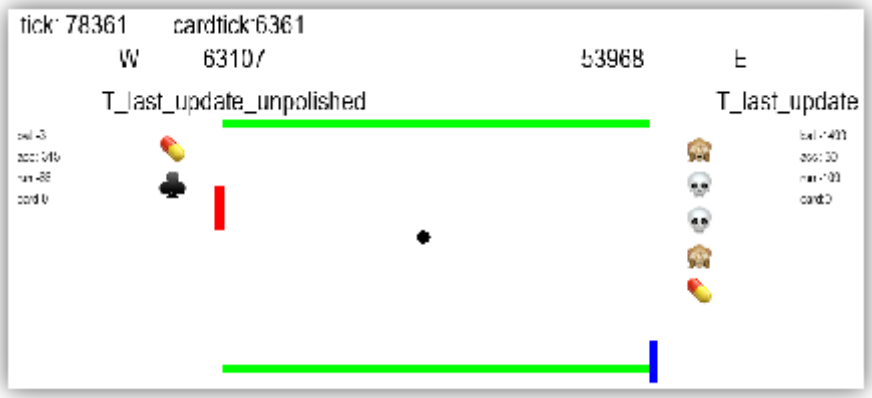
在运行时间方面，last_update 和 kitty_pro、piggy_pro、doggy_pro 三大小动物主力以及陪练 Aquinas 的每局总耗时均控制在 0.5s 以内，都在较短时间内运算完毕。其中 last_update 平均耗时在 0.1s 左右；kitty_pro 运行时间在 0.05-0.15s 之间波动，平均也在 0.1s 左右。这两者在主要估值算法上区别不大，故运行时间也差不多。相对而言，piggy_pro 由于使用估值函数对我方和对方跑位的击球价值进行了两层遍历，运行时间较长，达到了 0.2-0.35s；Doggy_pro 由于采用了不同的加权估值算法，运行时间也较长，平均运行时长为 0.4s。陪练 Aquinas 由于只进行一些方差均值的计算，估值函数没有涉及道具等更复杂的计算，故耗时较少，一般控制在 0.05s 以内。

各个算法的运行时间基本与其计算最大击球价值的估值函数复杂程度呈正相关，也说明了这部分环节是代码运行主要的时间开销部分。但这几个算法间对战的成绩与运行时间的关系并不明确，doggy_pro 最长久的计算确实为它在对决 kitty_pro 和 piggy_pro 时取得优势，但快速给出结果的 Aquinas 也能在对阵 piggy_pro 时取得完胜。也许与最终冠军 Oscar 队长达数秒的运行时间相比，我们的算法的计算进行得不够充分，还不能更有效地指导击球、跑位等，从而未能显示出计算量产生的优势。

3.3 经典战局

这里展示了我们在开发过程中和 8 强赛对阵冠军 Oscar 的两场经典战局。

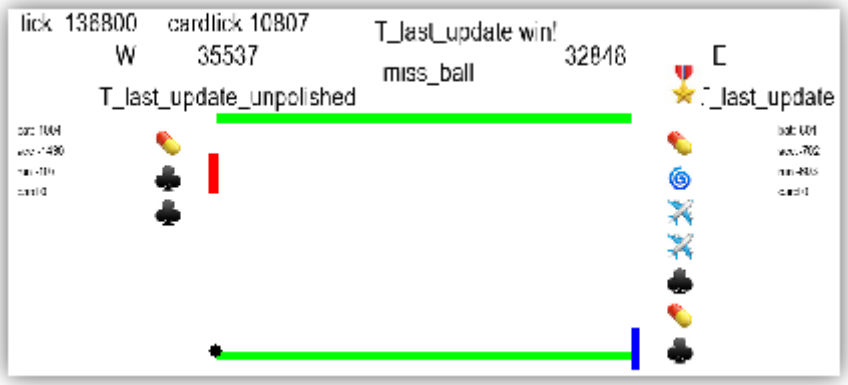
在第一次热身赛前我们对 3.0 版本的主力算法 last_update 进行了一次重要的改动，加入了体力 50000 以下采用归中跑位的设定，开发出了 3.1 版本的 Augustinus。Augustinus 在热身赛中相对于 last_update 的胜率有了显著的提升。这里展示了未更新的 3.0 版本也即 last_update_unpolished 与加入新策略的 last_update 也即 3.1 版本的 Augustinus 的经典对局。



两方除了残局跑位外没有其他区别，故在开局阶段除了 West 方占据了一些发球优势外，两方势均力敌。在道具使用方面的差距可能也与 West/East 方有关。

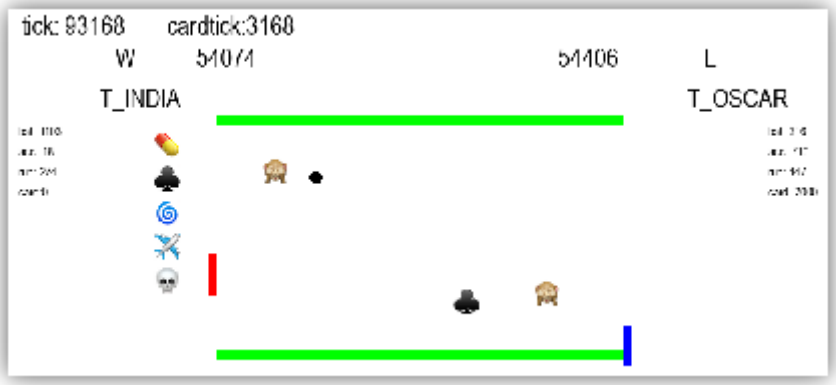
但进入体力 50000 以下的残局阶段后，采用归中跑位的 Augustinus 在体力消耗上明显拉近了与 last_update_unpolished 的差距，并且可以维持更多回合不出现击球盲区。而 last_update_unpolished 由于跑位靠边，在被 Augustinus 吊对角时极容易出现击球盲区而失利。

同时在跑位方面我们也注意到 last_update 的跑位距离过小，在热身对抗中处于劣势，此后的主力算法转而吸纳了热身赛中某些组采用的 1/3 跑位的策略。

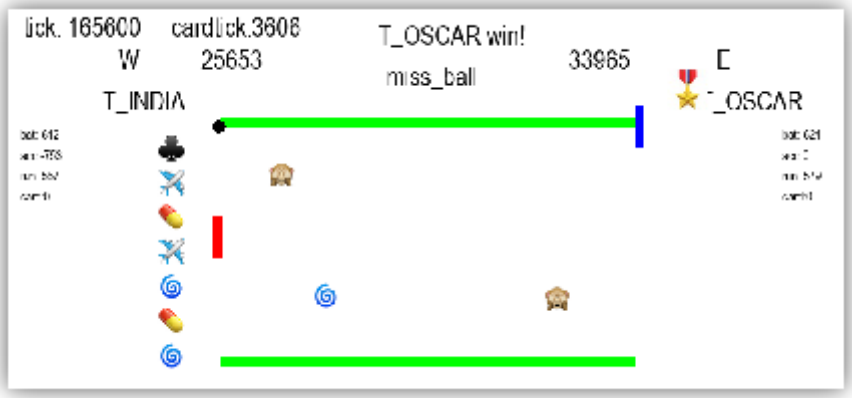


此外，在 8 强赛中，我们遭遇了实力雄厚的冠军队伍 Oscar，被以 80:0 的成绩零封。这里展示了与 Oscar 的一个经典对局。

Oscar 采用了包括 1/3、归中、不跑位等多种跑位策略，大多数情况下采用 1/3 跑位，视情况而定会采取归中跑位或不跑位，而且选择这两种跑位的时机都比较恰当。我方以 1/3 跑位与不跑位结合的方式应对，在前半局没有被拉开体力差距。



但是我们注意到，Oscar 除了补血包和掉血包以外几乎不吃道具，于是我方在使用道具的数量上远远超过了 Oscar。同时为了吃道具，我方在某些回合放弃了打角的击球策略，而 Oscar 则会根据道具位置计算我方回球进行最佳跑位，因而我方在大量使用道具后反而占了下风。



于是在进入体力 35000 以下的残局阶段时，Oscar 往往会有数千点的体力优势，而我方此时采取归中跑位，双方体力拉锯式下降，最终对方打角会将我方体力耗尽。所以在 8 强赛第一轮结束后调整时，我们便试图大幅调整我们算法里赋给道具的价值，尽量不去吃普通的道具。但仓促中修改的代码在内部测试中反映出了明显的削弱，于是我们第二轮不得不派出陪练 Aquinas 参赛，但可惜未能力挽狂澜。

4 实习过程总结

4.1 分工与合作

5.11 日我们建立小群，正式组队，由组长闵靖涛，组员代国峰、鲁洪亮、常海宁、韩潇霖、孙唯一、衣可心组成。

5.20 日晚上八点在四教 419 召开了第一次组会。主题是组员分工和一些可能的算法探讨。我们初步的分工是：

代码组：代国峰、闵靖涛；

测试组：韩潇霖、鲁洪亮、孙唯一；

报告组：常海宁、衣可心

不过在最终的实际操作中，我们的主程序由鲁洪亮在代国峰的代码的基础上完成。闵靖涛独立完成了多代代码，很好地完成了陪练功能，并在 1/4 决赛的第二轮中得以上场。

此外，第一次组会中，国峰给同学们简要的介绍了遗传算法、广度优先算法、深度优先算法等（虽然并没有在正式代码中得以体现）。同时，我们讨论了基本的游戏策略，包括道具使用、迎球、跑位等方面。





5.23 的上机时间，小组七位同学在机房集合，一起分析了基本的 table 等基本代码，并且再次进行了初步讨论。





5.25 日晚九点，五位男生在 434 寝室召开了组会，确定了我们组的头像，代国峰主笔提出了算法的初稿。

5.26 日，代国峰完成了第一代代码：T_INDIA，闵靖涛完成了第一代陪练代码：T_intelligent。然而，“暴力打角”的 intelligent 对 INDIA 的“碾压”让我们对 INDIA 单纯的归中跑位产生了质疑。

5.27 日，代国峰完成了第二代代码：T_plus_SPIN_attention。通过对参数的进一步调整，克服了 INDIA 存在的问题，足以在不使用 CARD_SPIN 的情况下赢过 intelligent。

5.28 日，鲁洪亮在 SPIN 的基础上改动了跑位诞生了 T_Piggy_attention 代码和 doggy 代码。同时，代国峰完成了第 2.5 代代码：betaPingpong。在微调之后，第三代代码 T_last_update 诞生。last_update 已经可以吊打 piggy。

5.29 日，在热身赛即将到来之时，发现 last_update 的一个缺陷：50000 生命以下的时候如果迎球位置太过于靠近边，容易被对方吊对角而 miss ball。因此，鲁洪亮在 50000 生命的时候做出修改，改为中点跑位，得到 AGUST。AGUST 在热身赛中取得 176/206 的好成绩，我们感到开心的同时发现了一些新的问题。在当晚，六位男生在 434 寝室召开组会，共同分析了复盘记录，对我们的跑位提出了更高的要求。



5.30 日，闵靖涛修正了 intelligent，得到了第二代陪练算法 Parmenides，已经可以战胜 betaPingpong，但还是不敌 AGUST。

6.1 日，鲁洪亮在 SPIN 的基础上，修改为 1/3 跑位，得到了 kitty。kitty 可以战胜其他很多代码，但是始终不敌 piggy。晚上在 last_update 的基础上，又修改跑位得到了相应的 piggy_pro 和 kitty_pro。

6.2 日的第二次热身赛中，上交的 piggy_pro 并没有取得非常理想的成绩，比第一次热身赛有较大的差距，一度让我们非常沮丧。当天晚上小组几名成员一起分析了复盘记录，通过观察和 Sabac、Apolo 等强队的对战，分析了我们的不足和对方的战略。



6.4 日，闵靖涛改进到了第三代陪练算法 Aquinas，依然是打角的战略，依然不会吃道具。在作为主算法的 piggy_pro 始终不敌 kitty_pro 的情况下，为了求稳，我们最终提交了 kitty_pro 作为最终代码。在提交备用的“猫猫”的时候，组长和其他组员的内心都是崩溃的。正如我们在热身赛上所言，“对出线不抱希望”。

6.5 日，我们迎来了最终的课堂竞赛。我们所在的 N 区有 Alpha、Quebec 等实力强劲的组。令人惊喜的是，在小组赛开始之初，N 区率先完成了所有比赛。Alpha 全胜 15 分小组第一出线，我们也以 12 分组二出线。这成绩让本已“绝望”的我们感到振奋，即使是不苟言笑的代国峰也露出了微笑。



无奈，出线后我们首先遇上了实力强劲的冠军种子 Oscar，并在第一轮比赛中以 0:40 的成绩惨败。然而我们并没有因为实力悬殊而轻易言败，根据复盘记录更改得到 kitty_pro_pro。由于时间所限，kitty_pro_pro 并未成型，因此第二轮中提交了陪练算法 Aquinas。遗憾的是，Aquinas 也 0:40 遭 Oscar 碾压，无奈出局。



6.9 日，在一切皆已尘埃落定的情况下，鲁洪亮根据 doggy 的跑位改进得到了第四个成熟代码 doggy_pro。虽然 doggy_pro 没有上场一展身手的机会，但它的诞生也使“小动物家族”得以完全，也展现了我组精益求精的精神。

4.2 经验与教训

在算法思想方面，我们强调启发式算法和侧重于道具使用，而忽略了传统搜索算法的使用，在一开始便认为搜索或者剪枝算法没有太大的优势，因而把一步贪心作为算法主体。不过根据 Oscar 组的经验，搜索来做最优解还是可行并行之有效的。这的确是一处小遗憾。

在小组合作过程中，我们组内交流还是不够紧密，降低了工作效率也浪费了不少时间。最终的代码没有充分体现我们的思路思想，也与此有关。希望以后的小组作业中，组员“相敬如宾”之外，沟通也要更加紧密。

4.3 建议与设想

漂移乒乓作为自己研发的游戏，具有很强的创新性。不过在实习过程中规则不断的更新，一方面规则的确得以完善，但另一方面也为我们设计算法、编写程序等造成了小小的麻烦。如果可能的话，希望以后的大作业规则和平台在正式发布之前更加完善。

由于时间等各方面的限制，我们最终的代码并不足够令人满意。虽然 Oscar 组已经将这个游戏“开发得差不多”了，但我们仍希望能尽力将我们的想法完善改进，推陈出新。

5 致谢

感谢陈斌老师给我们提供的实习的机会，感谢助教老师和同学们的付出和帮助。

感谢团结全队、各方沟通并亲力亲为屡败屡战锲而不舍乐观开朗的组长闵靖涛，感谢完成代码主体为最终代码打下坚实基础默默耕耘任劳任怨的代国峰，感谢自称划水实则挑灯夜战不眠不休不离不弃的鲁洪亮，感谢提供后勤保障服务并撰写报告的常海宁、韩潇霖、孙唯一、衣可心。

感谢 INDIA 组在实习过程中展现出的“相互协作、各尽所长、互不抱怨”的团队精神。感谢大家“不抛弃，不放弃”的奋斗拼搏坚强信念。

感谢大家“知其不可而为之”的执着和勇气。

感谢大家“有福同享，有难同当”的温暖。

感谢缘分让我们相逢相聚于 INDIA。

感谢三十天时光温柔以待。

临表涕零，不知所言。

6 参考文献

1. 王小春 编著. PC 游戏编程（人机博弈）[M]. 重庆:重庆大学出版社, 2002.
2. 吉根林. 遗传算法研究综述[J]. 计算机应用与软件, 2004, 21(2): 69-73



VICTOR

组长：端韵成

组员：谢冠旖，王璐璐，吴放，赵芯露，李邦卓，王云蛟

小组赛战绩：E2

决赛战绩：八强

数据结构与算法课程实习作业报告

谢冠旖 端韵成* 王璐璐 赵芯露 李邦卓 王云蛟 吴放

摘要：

本小组（VICTOR 组）采用了较为简单明了的策略进行漂移乒乓球的比赛：在发球函数中使用了简易的学习方法，使我方程序能够尽快找到可以持续胜利的发球位置和速度；跑位和接球函数在不同阶段使用了不同的方法，体力足够时不跑位，体力不足时适当跑位以避免失球，这一函数在后来的比赛中还有更改；回球是采用对方消耗减去我方消耗差值最大的方法；在使用道具上，我们尽量去捡对我方来说帮助较大的道具，例如旋转球，补血包。整个程序中使用了字典，链表，栈等数据结构。在经过内部测试和两次热身赛后，我们进步明显，从第一热身赛 27/105 的低胜率到第二次热身赛的 71/170，到最后进入八强，离不开小组成员的努力。

关键词：

字典 链表 栈 最大能耗差 学习

1.算法思想

1.1&1.2.总体思路和算法流程图

发球函数

发球函数需要确定的的发球的位置和速度。我们组使用了“伪学习”的方式对发球位置和速度进行选择。因为每一个对战的对手策略不同难以预测，放到一起统计可能会造成问题，所以我们针对每个对战的队伍记录和该队伍竞赛的历史数据，获得最终胜利的场次的发球位置将会有更大的机会被使用。由于数据和位置的组合情况非常多，而测试的比赛场数很少（和每个队只有 20 局比赛，我方发球只有 10 次机会），不能在过程中遍历所有的发球位置，所以我们采用将发球位置分区间的方法。

我们首先按照某些方法（详见后文叙述）选出可行的发球区域和对应的速度，这是我们发球位置的选择范围。

在对战某一队时：

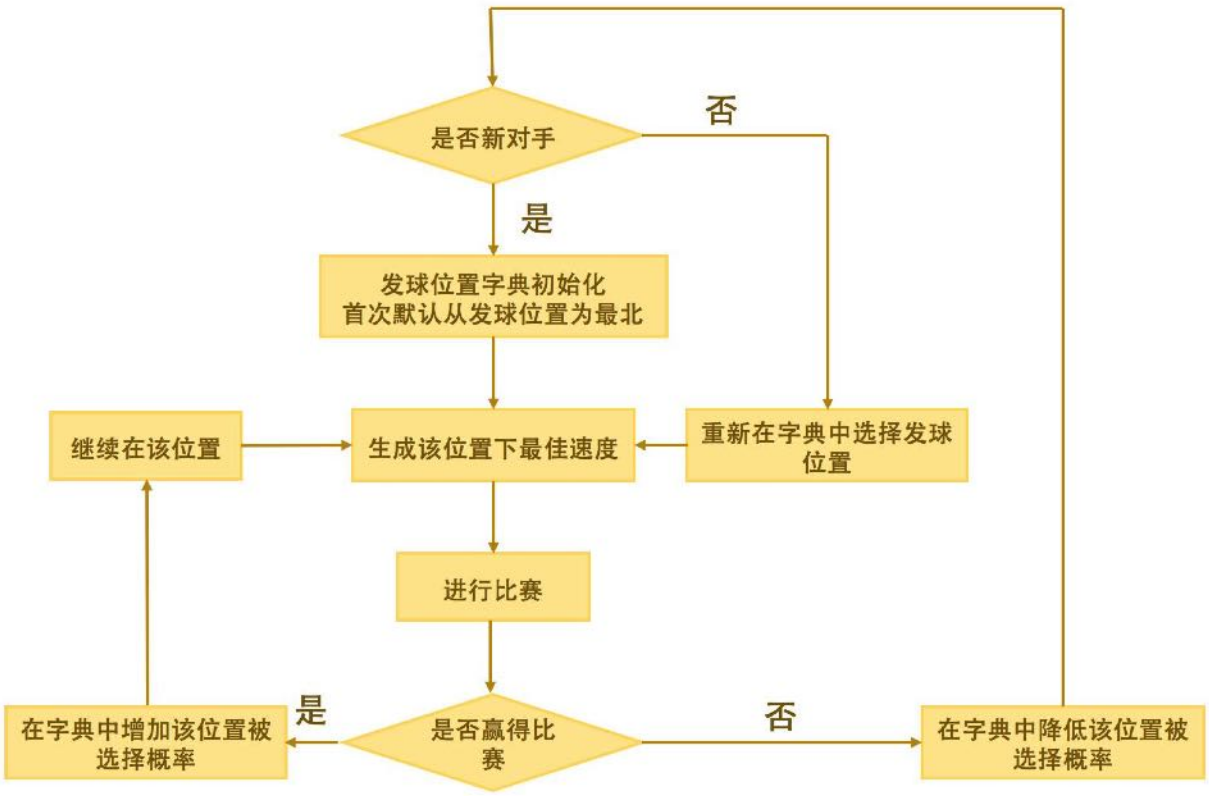
1. 每当更换了新对手时，我们先将发球位置选定在之前计算出的最好发球位置，即球桌的最北方；

2. 如果赢得比赛，则该区局加权分数增加，并继续在赢局的发球位置发球；

3. 如果这个发球点发球输掉，则该发球点就重新在发球位置的集合中选择新的发球位置。

注意：更换对手则历史数据重置。程序框图如图-1 所示。

关于最适合发球速度的选择，确定发球位置后，可以根据必须碰撞 1~2 次的原则，得出速度的合理区间，然后便历一遍，得出让（对方跑位消耗-我方发球改变速度消耗）最大的最优解，返回该发球位置下的速度。



接球和跑位函数

这是在打球函数中的一部分，接球函数很直观，就是从上次跑位停下的地点开始跑到球将会落到的位置，如果以现在的速度无法达到球的位置则会被判为 miss_ball，所以跑位和接球的平衡在这里十分重要。由于跑位事实上是为了下一次接球服务，所以在此时，我们先计算下次接球可能的速度。我们通过计算下次接球之前可能的体能来计算下次接球的速度：

1. 在不跑位的情况下，下次接球时的体能=此时体能-此时接球消耗的体能（us_HP_consume_pos）- 此时改变速度消耗的体能（us_HP_consume_vector）

2. 在确定了要跑位的情况下，之后还要减去跑位的最大消耗。但我们开始都会先计算#1，得到速度后判断是否需要跑位，然后再选择是否进行#2。

关于如何判断是否需要跑位：得到#1 中的体能时，通过 table 中的 bat_velocity = int(当前剩余体力值 / RACKET_LIFE) * BALL_V[0])公式，得到不进行跑位时，下一轮球拍的全速。之后，通过 runrange_max = abs(TICK_PER_TURN * bat_velocity)得到球拍在下一轮以全速的可以跑动的最远距离 runrange_max。之后与球拍可能需要跑动的最远距离进行比较，若是一次无法跑到，则判断为需要跑位。

根据是否跑位进行后续的步骤：

1. 不需要跑位，则假装跑位（因为对手只知道跑位的方向），例如我们的位置靠北，则向南走 1 步，认为这样既损耗我方很少的体力，还有可能误导对方的回球（但事实证明没有队伍上当）。这一策略在我们八强赛第二轮输给 GOLF 之后进行了更改，匆忙更改成了-如果在偏向一侧的位置，则向中间移动和中间位置差的 0.2 倍（详见代码），以减小跑位和接球的总能耗。

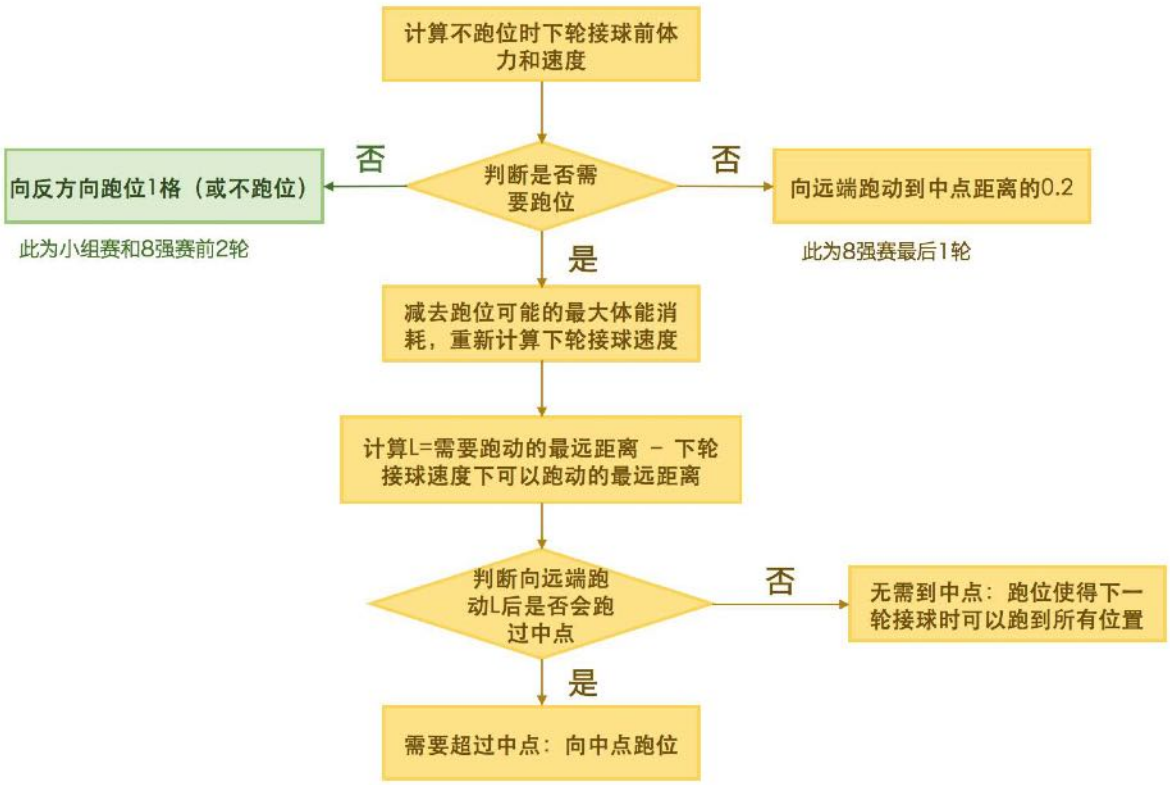
2. 需要跑位，即一次无法达到可能需要跑动分最远距离，则先减去跑位所需的最大能耗（即跑到中间的能耗），之后计算跑位距离和方向。我们的跑位策略是：保证能跑到任何位置情况下尽量少跑位。

因此我们跑位的方式也有两种：

2.1. 不需要跑到中点也可以在接下来可以跑到全场，即：可能需要跑动的最远距离 - 可以跑动的最远距离< 可能需要跑动的最远距离 - 中点位置。

2.2. 如果上一种移动方法会使得球超过中点，则适得其反，移动使得球在中点。这两种情况下，如果连跑位都跑不到，就不跑动了，听天由命了。

程序框图如下图所示：



回球函数-概况

因为回球的时候位置是固定的，所以回球策略的重点是加速度的选择，本组所采用的的回球策略如下：

1. 首先，获得对方回球之后球到达本方时，球在 y 方向的位置和速度；

2. 然后，在满足球 1-2 次碰壁的要求的合法速度范围内，分两种情况讨论：分别是回球的 y 速度为正或者负；

3. 当 y 速度为正时，通过在合法速度范围内的逐个取值，找出使对方最小消耗和我方改变速度消耗之差最大的速度；

4. 当 y 速度为负时，采用与 #3 一样的策略来获得此时的最优速度；

5. 最后，如果发现没有道具或者捡道具不如原计划划算，就放弃道具，但是如果捡道具的好处更大，就捡道具，具体捡道具时的速度计算见道具策略；

6. 另外，还需要考虑对方道具的影响，在判断对方使用了旋转球后，我们把 y 加速度变为 2 倍。

回球函数-捡道具部分

本组捡道具的基本策略是：

1. 首先获得并遍历场上出现的道具的名称和位置；

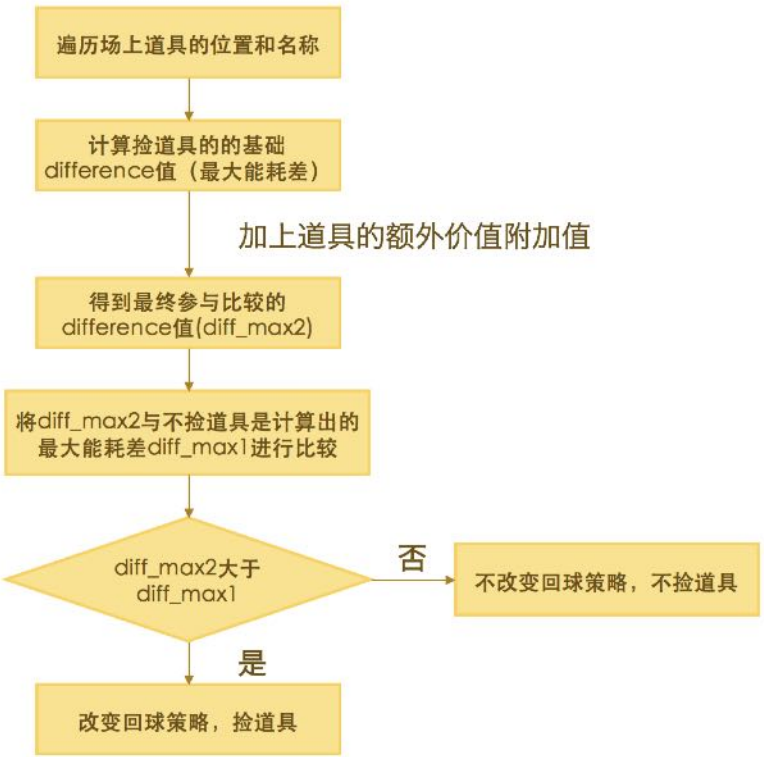
2. 计算捡每一个道具时，对方的最小消耗（接球时的跑位消耗）减去我方给球加速的消耗的差值，记为基础 difference；

3. 捡道具时，有三种情况：球尚未反弹时捡道具，球反弹一次后捡道具，球反弹两次后捡道具，针对三种可能情况，计算出分别的基础 difference，取最大值返回；

4. 针对不同道具，我们根据自己的需求排位，给一个额外的 difference 附加值，其中，旋转球 SP 为第一，掉血包和补血包为第二，变压器为第三，隐身术瞬移术为第四，计算基础 difference 值加上道具价值的附加值得到最后的 difference；

5. 用最后的 difference 与不捡道具时的 diff_max 进行比较，取其中最大的值并按照得到该值时的加速策略改变回球速度。

程序框图如下图所示：



回球函数-使用道具部分

本组使用道具的基本策略是：

1.在回球时判断此时我方是否有道具能够使用；

2.判断道具种类，如果不是旋转球，则直接使用，并按照原有回球策略回球；如果是旋转球，则需要进一步判断。

3.决定是否使用旋转球时，判断我方是否能够通过改变特定速度使得对方在旋转球作用下如果没有防守很大可能会死亡。如果存在这种方法，就使用，并且回球策略为该特定速度回球；如果不存在，就不使用，并按照原有回球策略回球。

1.3.算法运行时间复杂度分析

1.3.1.发球函数

每一局中算法复杂度为 O(n),n 为合法发球速度的区间范围。

基本步骤：由于测试局数较少，数据集有一定的偶然性，无法使用机器学习等方法。我们采用了一个比较取巧但是实际效果很好的方法：和某一个对手测试对阵时，如果胜利就继续相同地点发球，如果失败就更换发球地点。

发球函数要求返回发球的位置和发球的速度，我们采取的方法是给发球位置提供一个选择（在字典中）。然后根据发球地点计算出合理的发球速度区间，便历一遍，计算接发球的体力消耗差，得到最优发球速度。最后返回发球位置和速度。

1.3.2.接球和跑位函数

每一次 play 接球和跑位函数的算法复杂度为 O(1)，即常数。

基本步骤：因为接球和跑位函数只涉及了有限次（个位数）的计算，不存在循环的结构，也没有使用类似于 append 等本身就有一定算法复杂度的语句。因此可以认为接球和跑位函数的算法复杂度为常数。

1.3.3.回球函数-普通版（不捡道具和用道具）

velocity_range 函数是计算在考虑道具后的使得双方能耗差最大的加速度。每一局算法复杂度为 O(n)，n 为满足 1-2 次碰壁的合法速度范围。

基本步骤：在前面已经计算的合法速度范围内，分为 y 正方向和 y 负方向两类，然后分别采用同样的处理方法。如果是正方向，就在正方向的合理范围内遍历，找到能使对方接球的最小体力消耗和我方加速的体力消耗之差最大的速度。如果是负方向亦然。另外，还要根据道具是否可见和捡道具是否优于不捡，来决定是否改变速度。

1.3.4.捡道具

caldiff 函数为判断每一次捡一个道具的最优方法，分为不反弹时捡道具，反弹一次时捡道具及反弹两次捡道具三种情况，调用该函数一次，时间复杂度为 O(1)。

Cardsornot 函数将每一次返回的捡道具的最优路径的双方体力消耗值加上价值附加值得到最终的 difference 与不贱道具的 diff_max 进行比较，返回最优回球策略，时间复杂度为 O(n)。

1.3.5.道具的使用

calc_velocity 函数判断是否存在回球策略使得使用旋转球道具后能够赢得比赛的函数，在每一局的算法复杂度为 O(n)，其中 n 为己方合法回球的速度数目。usingcards 函数检查是否存在可用道具，并判断是否使用，在每一局的算法复杂度为 O(1)。

基本步骤：道具的使用首先判断道具箱最上方的道具种类，如果不是旋转球就直接使用，因为其他道具在各个时刻使用无太大差异;如果是旋转球则需要进一步判断是否存在能够大几率赢得比赛的策略，如果存在就按照该策略回球，如果不存在就正常回球。

对于旋转球的策略我们首先取得正常回球的速度范围并进行遍历,每一次都判断若对方不改变速度方向该球是否是合法的，即是否能够保证碰壁一到两次。如果使用某种速度回去后，对方不改变速度方向的回球是不合法的，则这种策略就是我们所寻求的回球方式。此外，在寻找到每一次策略后都与上一个策略所消耗的体力所比较，并保留消耗体力最小的那一种。

2.程序代码说明

2.1.数据结构说明

发球函数

使用了字典和链表这两种数据结构。其中字典 pos_dict 用于储存供选择的发球位置，通过 pop 和 setup 来增添和减去字典中的键值对，达到更改各发球位置权重的目的。而链表则属于暂时的，发球时若判断得出这轮发球需要从发球位置字典中抽取一个位置发球，则将字典 pos_dict 的 key 提取出来生成一个链表，之后使用 random 包中的工具 random.choice 从这个 list 当中随机抽取一个 key，相当于在 dict 中随机抽取了一组键值对，得到了新的发球位置。

接球和跑位函数

利用 TableData 中的字典读取数据，例如 ball_pos = tb.ball.get(‘position’).y 等等，都是获取 table 中某一 key 对用的 value 的形式。

回球函数-普通版（不捡道具和用道具）

同样，在不同的部分也利用了 TableData 中的字典读取数据，例如：

```
ball_op_pos = tb.op_side.get('position').y  # 对方在 t0 时刻的位置
ball_pos = tb.ball.get('position').y  # 对方球 y 方向到达本方的位置
ball_velocity = tb.ball.get('velocity').y  # 对方球 y 方向到达本方的速度
```

回球函数-捡道具

同样，在不同的部分利用了 TableData 中的字典读取数据。另外，也利用了表示桌面上道具的链表，遍历链表中的各项，即遍历了桌面上的所有道具。举例：cards = tb.cards.get('cards')，其中得到的这个 cards 就是桌面上道具的链表。

回球函数-道具的使用

同上，在不同的部分利用了 TableData 中的字典读取数据。此外，在读取和使用道具时，进行了读取“栈”（这个类似于“栈”的类型其实是 table 中的 CardBox 类型，是 list 的

一个子类型）中的数据，和类似于将数据出“栈”的工作。

2.2.函数说明

2.2.1.发球函数

我们的发球部分包括了 2 个部分：发球函数—serve(op_side, ds)本身，和记录储存函—summarize(tick, winner, reason, west, east, ball, ds)。同时我们也使用了在整个程序中的辅助函数—合法速度函数—get_legal_velocity(pos)。

首先，说明发球函数：

```
41  def serve(op_side, ds):
42      global pos_dict, name_opside, last_opside, roundround_op, serve_pos, islastwin, last_pos  # 为了和summarize中同步，全部都要global!
43      print(pos_dict)
44
45      try:
46          if last_opside != op_side:  # 如果换了对手（last_opside上一盘的对手，op_side这一盘的对手）
47              name_opside = op_side  # 本局的对手
48              pos_dict = {1: 1000000, 2: 990000, 3: 980000, 4: 800000, 5: 700000, 6: 500000, 7: 300000,
49                          8: 100000}  # 发球位置字典回到初始状态
50              serve_pos = 1000000  # 开始默认的发球位置是计算出的最好位置
51              roundround_op = 0  # 总共赢了的局数
52          else:
53              name_opside = op_side  # 如果没换对手，上一次赢了就用上一次的发球位置
54              if islastwin:
55                  serve_pos = last_pos
56              else:
57                  # print('round number before choose',roundround_op)
58                  # 如果上一次没有赢，就从剩下的位置里面随机选择
59                  choose_pos = random.choice(list(pos_dict.keys()))
60                  serve_pos = pos_dict[choose_pos]
61      except:  # 如果出错了的话，比如这是所有比赛的第一局，或者因为一直输字典已经删空了
62          name_opside = op_side
63          serve_pos = 1000000
```

在这部分中，我们首先通过判断这局对手和上局对手是否相同，选择是否初始化发球位置字典（44—53）。之后通过上一局输赢的状况选择是否要从字典中重新选择发球位置（54—60）。serve_pos 为在该部分选择出的发球位置。

```
65          pos = serve_pos  # 得到发球位置之后和原来一样计算发球速度
66          v_ranges = get_legal_velocity(pos)
67          vmin, vmax = v_ranges[0]
68          for v_postive in range(vmin, vmax):  # 直接取正向速度，否则发球速度损耗过大
69              velocity_HP_consume = (v_postive - 1000) ** 2 / FACTOR_SPEED ** 2  # 我方改变速度消耗
70              presume_op_pos, presume_op_v = Presume_ball_Status(pos, v_postive)
71              op_HP_consume = (presume_op_pos - 500000) ** 2 / FACTOR_DISTANCE ** 2  # 对方跑位消耗
72              difference = op_HP_consume - velocity_HP_consume #对方跑位消耗减去我方发球改变速度消耗
73              diff_max = 0
74              if difference > diff_max: #选择消耗差最大的速度
75                  diff_max = difference
76                  v_max = v_postive  # v_max keeps record of v that gives maximum difference
77          return pos, v_max
78
```

在这部分中，我们计算该发球位置下的最适合的发球速度，在这里我们引入 get_legal_velocity(pos)函数，该函数的作用是判断在某一位置下，发球或者回球的合法速度区间。之后，可以根据必须碰撞 1~2 次的原则，在合法速度区间内遍历一遍，得出让（对方跑位消耗-我方发球改变速度消耗）最大的最优解，返回该发球位置下的速度。

接下来解释历史数据储存函数：

```
576 def summarize(tick, winner, reason, west, east, ball, ds):
577     global pos_dict, name_opside, last_opside, roundround_op, serve_pos, islastwin, last_pos
578     # 为了和serve中同步，全部都要global！！
579     try:
580         last_opside = name_opside #对局结束后，该局对手就变成了"上一局的对手"
581         last_pos = serve_pos #该局的位置也变成上一局的位置
582         if winner == 'West': #我方胜利
583             pos_dict[roundround_op + 9] = serve_pos #把该位置作为value，为避免重复，赢局数+9作为该value的key
584             islastwin = True
585             roundround_op += 1
586         else: #如果我方输了这局，则在字典里删去对应的key-value
587             try:
588                 thenumber = list(pos_dict.keys())[list(pos_dict.values()).index(serve_pos)]
589                 pos_dict.pop(thenumber)
590                 islastwin = False
591
592
593             except: #避免可能的错误发生，例如字典已经被删空
594                 islastwin = False
595     except:
596         return
597
598     return
```

在该函数中，记录必要的历史数据，并且进行输赢的基本判断。如果数据显示赢家（winner）为发球方（即我方），则在发球位置字典中加入此位置，也意味着加权的增加（582—585），并且记录 inlastwin 为 True，以便在下一局发球函数 serve 中用于判断。但如果数据显示赢家为对方，则在发球位置中删去此位置，并且记录 inlastwin 为 False（588—594）。

接下来解释我们组在这一步使用的辅助函数 get_legal_velocity(pos)，这个辅助函数在整个程序中使用多次。

在该函数中，我们分正向（y 方向速度为正）和反向（y 方向速度为负）两种情况考虑，分别形成各自情况下的最大和最小速度，函数最终返回的是（正向速度范围，负向速度范围）的元组。

2.2.2.接球和跑位函数

在接球和跑位函数中，一个很重要的部分就是预测下次接球时球拍的速度和可能到达的位置。下面的代码求出在不跑位的情况下，下一轮可能的接球速度。

```
515     v_vector = acc_vector.y  # 我们需要改变球的速度的y方向分量
516     us_HP_consume_vector = v_vector ** 2 / FACTOR_SPEED ** 2  # 我们用于改变速度的体能消耗
517     us_HP_consume_pos = (ball_pos - racket_pos) ** 2 / FACTOR_DISTANCE ** 2#我们用于接球的移动消耗
518
519     bat_velocity = int(
520         ((tb.side.get('life') - us_HP_consume_pos - us_HP_consume_vector) / RACKET_LIFE) * BALL_V[0])  # 球拍的全速，是整数形式
521

27  max_d2 = 2 * DIM[3] + pos  # 球的速度为反向时
28  min_d2 = pos
29  max_v2 = math.floor(- max_d2 / TICK_PER_TURN) + 1
30  min_v2 = math.floor(- min_d2 / TICK_PER_TURN)
31  vrange2 = (max_v2, min_v2)  # 速度直接取范围
32  vrange = (vrange1, vrange2)
33  return vrange
```

在该部分中，我们得到改变球速的 y 分量，依照 table 中的公式计算改变速度的体能消耗，同时也根据公式计算我们用于接球的体能消耗（分别为 516 行和 517 行）。而后根据速度和剩余体能的关系公式计算出在不跑位的情况下下一轮的接球全速，为接下来预测是否需要跑位和如何跑位使用。

```
526     runrange_max = abs(TICK_PER_TURN * bat_velocity)  # 球拍全速移动在t1到t2或从t2到t3可以走的最远距离
527
528     south_max = ball_pos - DIM[2]  # 离南边的最远距离
529     north_max = DIM[3] - ball_pos  # 离北边的最远距离
530     direction_max = 'south'  # 判断离哪边最远
531     if north_max > south_max:
532         needrun_max = north_max
533         direction_max = 'north'
534     else:
535         needrun_max = south_max
536
```

在上图的部分中，我们首先计算了球拍全速移动可以走的最远距离（526 行），并确定了球拍需要跑动去接球的最远距离。之后与球拍可能需要跑动的最远距离进行比较，若是一次无法跑到，则判断为需要跑位，若是可以达到则判断为不需要跑位。

在小组赛和 8 强赛对抗 GOLF 的前两轮，我们在不需要跑位的时候采用了如下的“跑位方式”-假装跑位（因为对手只知道跑位的方向），例如我们的位置靠北，则向南走 1 步，

```
532     if needrun_max < runrange_max:  # 如果体力值足够，球拍可以按时到达
533
534         if ball_pos > (DIM[2] + DIM[3]) / 2:  # 如果我们在北半边，假装往南移动一格
535             run_vector = Vector(0, 0)
536         else:  # 在南边，就不用动了，对方会以为我们向北移动了
537             run_vector = Vector(0, 0)
```

认为这样既损耗我方很少的体力，还有可能误导对方的回球。

当第二轮以 0：40 输给 GOLF 后我们改变了回球策略：如果在偏向一侧的位置，则向中间移动和中间位置差的 0.2 倍（详见代码），以减小跑位和接球的总能耗。详见下图的代码：

```
537     if needrun_max < runrange_max:  # 如果体力值足够，球拍可以按时到达
538
539         if ball_pos > (DIM[2] + DIM[3]) / 2:  # 如果我们在北半边，则向南移动，更靠近中间
540             run_vector = Vector(0, -0.2*(ball_pos - (DIM[2] + DIM[3]) / 2))
541         else:  # 在南边，则向北移动，更靠近中间
542             run_vector = Vector(0, -0.2*(ball_pos - (DIM[2] + DIM[3]) / 2))
```

然而，在体力不足时，我们需要跑位。但在决定如何跑位之前，要先减去跑位所需的最大能耗（即跑到中间的能耗），之后计算跑位距离和方向（见下图代码 547 行到 552 行）。我们的跑位的大策略是：保证能跑到任何位置情况下尽量少跑位。具体的跑位方式有两种，具体如下图的代码所示：

跑位的方式有两种：

```
544     else:  # 不足以到达，需要补位的时候进行提前移动
545         if needrun_max == runrange_max:  # 刚好够，不移动
546             run_vector = Vector(0, 0)
547         else:  # 不够，需要移动
548             us_HP_consume_runmax = ((DIM[2] + DIM[3]) / 2 - ball_pos) ** 2 / FACTOR_DISTANCE ** 2  # 我们最大的用于跑位的消耗
549             bat_velocity = int(
550                 ((tb.side.get('life') - us_HP_consume_pos - us_HP_consume_vector-us_HP_consume_runmax) / RACKET_LIFE) * BALL_V[
551                     0])  # 球拍的全速，是整数形式int
552             runrange_max = abs(TICK_PER_TURN * bat_velocity)  # 球拍全速移动在t1到t2或从t2到t3可以走的最远距离
553 
554             if needrun_max - runrange_max < needrun_max - (DIM[3] - DIM[2]) / 2:  # 移动使得等对方回球后，我方在t1到t2时间内可以够到全场所有
555                 if runrange_max > needrun_max - runrange_max:
556                     if direction_max == 'north':
557                         run_vector = Vector(0, needrun_max - runrange_max)
558                     else:
559                         run_vector = Vector(0, runrange_max - needrun_max)
560                 else:
561                     run_vector = Vector(0, 0)  # 如果这也跑不到就听天由命不动了
562             else:  # 如果上一种移动方法会使得球超过中点，则适得其反，移动使得球在中点
563                 if runrange_max > needrun_max - (DIM[3] - DIM[2]) / 2:
564                     run_vector = Vector(0, (DIM[2] + DIM[3]) / 2 - ball_pos)
565                 else:
566                     run_vector = Vector(0, 0) # 如果这也跑不到就听天由命不动了
567 
```

1. 不需要跑到中点也可以在接下来可以跑到全场，即：可能需要跑动的最远距离 - 可以跑动的最远距离< 可能需要跑动的最远距离 - 中点位置。（见上图代码 554 到 561）
2. 如果上一种移动方法会使得球超过中点，则适得其反，移动使得球在中点。这两种情况下，如果连跑位都跑不到，就不跑动了，听天由命了。（见代码 562 行到 566 行）

2.2.3.回球函数-概况

在这部分非常重要的函数是 velocity_range 函数，在打球函数 play 中，由 usingcards 函数引入到这个 velocity_range 函数，目的是计算考虑了道具的情况下，回球时加速度选择的最好方式。首先，在这里我们先解释不考虑捡道具时的想法，后续会详细解释捡道具的过程。

```
106 def velocity_range(tb: TableData, ds: dict):
107     ball_op_pos = tb.op_side.get('position').y  # 对方在t0时刻的位置
108     ball_pos = tb.ball.get('position').y  # 对方球y方向到达本方的位置
109     ball_velocity = tb.ball.get('velocity').y  # 对方球y方向到达本方的速度
110 
111     Vranges = get_legal_velocity(ball_pos)
112     vranges_postitve, vranges_negative = Vranges  # 获得击球时的返回球合法速度，分为正向，反向
113     v_pos_min, v_pos_max = vranges_postitve
114     v_neg_min, v_neg_max = vranges_negative
115     diff_max = 0  # maximum difference of HP
116     Found_one = False
```

以下开始解释 velocity_range 这一关键函数（106 行-156 行）：

在该部分，我们首先获取对方在 t0 时刻的 y 位置、对方球到达本方的 y 位置和 y 速度。调用前面 get_legal_velocity 函数获取了此时位置的合法速度 Vranges，分别找出正速度和负速度的最大值最小值，并初始化双方体力消耗之差的最大值 diff_max=0，Found_one = False。之后在合法速度内取值，找到消耗差的最大值。

```
117     for v in range(v_pos_min, v_pos_max):  # 在所有合法速度范围内一个一个取值
118         prsum_pos, prsum_velocity = Presume_ball_Status(ball_pos, v)
119         v_vector = v - ball_velocity  # 速度矢量改变
120         op_minHP_consume = (prsum_pos - ball_op_pos) ** 2 / FACTOR_DISTANCE ** 2  # 此时对手最小消耗
121         us_HP_consume = v_vector ** 2 / FACTOR_SPEED ** 2  # 我方消耗
122         difference = op_minHP_consume - us_HP_consume
123         if Found_one == False:
124             diff_max = difference
```

在上图的代码中，在正速度的最小值 v_pos_min 和最大值 v_pos_max 范围内取值，调用 Presume_ball_Status 函数来计算到达对方的位置还有速度，op_minHP_consume 表示对方接球时的最小的移动距离体力消耗值（假设对方没有跑位），us_HP_consume 表示我方改变速度的体力消耗。通过一个个比较，找出最优速度 v_max。

在负速度的最小值 v_neg_min 和最大值 v_neg_max 范围内取值，具体算法和在正速度里取值完全一样。

```
133     for v in range(v_neg_min, v_neg_max):  # 在所有合法速度范围内一个一个取值
134         prsum_pos, prsum_velocity = Presume_ball_Status(ball_pos, v)
135         v_vector = v - ball_velocity  # 速度矢量改变
136         op_minHP_consume = (prsum_pos - ball_op_pos) ** 2 / FACTOR_DISTANCE ** 2  # 此时对手最小消耗
137         us_HP_consume = v_vector ** 2 / FACTOR_SPEED ** 2  # 我方消耗
138         difference = op_minHP_consume - us_HP_consume
139         if Found_one == False:
140             diff_max = difference
141             v_max = v
142             Found_one = True
143         else:
144             if difference > diff_max:
145                 Found_one = True
146                 diff_max = difference
147                 v_max = v  # v_max keeps record of v that gives maximum difference
```

调用 cardsornot 函数，判断是否捡道具，如果 k==False 说明没道具可见或者捡道具不如原计划，保持 v_max 不变，否则用捡道具的 v_fin 代替。最终返回 x 方向的加速度 0 和 y 方向加速度 v_fin - ball_velocity。

最后介绍如果碰到对方使用旋转球时的策略代码：

```
511     op_card_active = tb.op_side.get('active_card')[1]
512     if op_card_active == CARD_SPIN:
513         acc_vector = Vector(0, 2 * acc_vector.y)
514
```

op_card_active 是通过 tb.op_side.get 获得的对方道具情况，acc_vector 是在 usingcards 函数里面通过调用 velocity_range 函数来获取的加速度。如果判断对方使用了旋转球，那么我们就采用 2 倍的加速度来应对。

2.2.4.回球函数-捡道具函数

1.首先是 caldiff 函数：计算捡道具的基础 difference 值，返回的是符合合理速度范围的能耗差最大的那一种方法，若在符合速度范围的条件里没有能够捡到该道具的方式，返回 None。这个函数在判断是否使用道具的函数-cardsornot 函数中被调用。
下面按照代码顺序解析代码：
1.1.针对正向速度和反向速度，一共有五种情况，计算五种情况下的等效 y 坐标（248 行-252 行）。

```
242 def caldiff(tb, eachcode):  # 这里面要返回的是符合合理速度范围的能耗差最大的那一种方法，###特别特别要注意的是，如果在符合速度范围的条件
243     op_pos = tb.op_side.get('position').y  # 对方在t0时刻的位置#################修改了名字
244     current_pos = tb.ball.get('position').y  # 对方球y方向到达本方的位置
245     current_pos_x = tb.ball.get('position').x
246     current_v = tb.ball.get('velocity').y  # 对方球y方向到达本方的速度
247     pos_x = eachcode.pos.x  # 道具的x坐标
248     pos_y_1 = eachcode.pos.y  # 反弹之前就捡到道具的y坐标
249     pos_y_2 = 2 * DIM[3] - pos_y_1  # 反弹一次等效的道具y坐标
```

1.2.获得在我方发球位置时的合法速度范围

```
253         v_ranges = get_legal_velocity(current_pos)  # 该位置的合法速度范围
254         Vrange1, Vrange2 = v_ranges
255         vmin, vmax = Vrange1  # 正向回球的速度范围
256         vmin2, vmax2 = Vrange2  # 反向回球的速度范围
```

1.3.通过两个坐标值，速度之比等于位移之比计算速度，计算五种情况下捡道具时需要的 y 方向的速度，如下图的代码所示：

```
260         expec_v_1 = (pos_y_1 - current_pos) / abs(pos_x - current_pos_x) * 1000
261         expec_v_2 = (pos_y_2 - current_pos) / abs(pos_x - current_pos_x) * 1000  # 通过两个坐标值，速度之比等于位移之比计算速度
262         expec_v_3 = (pos_y_3 - current_pos) / abs(pos_x - current_pos_x) * 1000
263         expec_v_4 = (pos_y_4 - current_pos) / abs(pos_x - current_pos_x) * 1000
264         expec_v_5 = (pos_y_5 - current_pos) / abs(pos_x - current_pos_x) * 1000
```

1.4.以第一种情况为例：首先判断 y 方向的速度是否合法，并获取到达对方的位置，计算此时的两方消耗的差值，将该值与已得到的最大的基础 difference 值比较，并赋值 v_max，diff_max

```
266         if expec_v_1 > vmin and expec_v_1 < vmax:  # 判断该速度合法
267             prsum_pos, prsum_v = Presume_ball_Status(current_pos, expec_v_1)  # 获取到达对方的球的位置
268             v_vector = expec_v_1 - current_v  # 我方速度矢量改变
269             op_minHP_consume = (prsum_pos - op_pos) ** 2 / FACTOR_DISTANCE ** 2  # 此时对手最小消耗
270             us_HP_consume = v_vector ** 2 / FACTOR_SPEED ** 2  # 我方消耗
358         if found_one:  # 如果有可以返回的合法速度
359             return (v_max, diff_max)
360         else:
361             return None  # 速度不合法时，返回NONE
276                 if difference > diff_max:
277                     diff_max = difference
278                     v_max = expec_v_1  # v_max keeps record of v that gives maximum difference
279             found_one = True
```

1.5.若有合法的速度值，返回该值，否则，返回 NONE

2.接下来，介绍最终判断是否捡道具的 cardsornot 函数：
cardsornot 函数：判断此时要不要捡道具，给道具评分，分数由基础 difference 值+有用程度组成，在 velocity_range 函数中被调用 。
下面按照顺序解析代码：
2.1.遍历每一种道具，注意这里的可以根据是否需要捡某种道具进行修改。具体代码如下图

```
364     # 判断此时要不要捡道具，给道具评分，分数由实际的能耗差+有用程度组成，在velocity_range函数中被调用
365     def cardsornot(op_position, ball_pos, ball_velocity, tb: TableData, diff_max1):
366         cards = tb.cards.get('cards')  # 得到目前桌面上的cards的列表，这是一个列表！！！
367         found_one = False  # 开始假设没找到任何可能捡的道具
368         thiscodename = str()  # 遍历时当前道具的名称
369         fin_codename = str()  # 遍历结束后，选定额道具的名称
370         diff_max2 = int()  # 遍历结束后，选定道具的最后得分（实际能耗差+有用程度）
371         for eachcard in cards:  # 开始遍历
372             if eachcard.code == 'IL' or eachcard.code == 'DL' or eachcard.code == 'SP'or eachcard.code=='AM' or eachcard.c
373                 calculate_card = caldiff(tb, eachcard)  # 捡这个道具的实际最大能耗差（能耗差为对方能耗减我方能耗）和对应的速度-形式:
374
375                 if calculate_card != None:  # 如果速度合法范围内可以捡到某道具,返回捡该道具的能耗差最大值
376                     realdiff = calculate_card[1]
```

所示：

后面有些很长的部分与图中显示的大同小异，因为格式所限没有截取进去。

2.2.以补血包和掉血包为例介绍“有用程度的表示”：

```
                if eachcard.code == 'IL' or eachcard.code == 'DL':  # 如果是掉血或者补血包，然后就是作比较
                    diff_in_mind = realdiff + 1000  # 得分（也就是在心目中的分量哈哈哈哈）
                    thiscodename = eachcard.code
                    if not found_one:
                        diff_max2 = diff_in_mind
                        fin_codename = thiscodename
                        v_card = calculate_card[0]  # 对应的速度
                    else:
                        if diff_in_mind > diff_max2:
                            diff_max2 = diff_in_mind
                            fin_codename = thiscodename
                            v_card = calculate_card[0]
```

在上述的代码里，我们给补血包和掉血包加上 1000 分的附加得分，参与到最优打法的比较中。

2.3. 同理，如果是旋转球，变压器还有瞬移术，得分的计算不同，如下图中的代码所示：

```
                else:
                    if eachcard.code == 'SP':  # 如果是旋转球，反正还是作比较，就是得分的计算不同
                        diff_in_mind = realdiff + 1800
                        thiscodename = eachcard.code
                        if not found_one:
                            diff_max2 = diff_in_mind
                            fin_codename = thiscodename
                            v_card = calculate_card[0]
                        else:
                            if diff_in_mind > diff_max2:
                                diff_max2 = diff_in_mind
                                fin_codename = thiscodename
                                v_card = calculate_card[0]
                    else:
                        if eachcard.code=='AM':#如果是变压器
                            diff_in_mind = realdiff + 500
                            thiscodename = eachcard.code
                            if not found_one:
                                diff_max2 = diff_in_mind
                                fin_codename = thiscodename
                                v_card = calculate_card[0]
                            else:
                                if diff_in_mind > diff_max2:
                                    diff_max2 = diff_in_mind
                                    fin_codename = thiscodename
                                    v_card = calculate_card[0]
                        else:
                            if eachcard.code=='DS'or'TP':
                                diff_in_mind = realdiff + 100
                                thiscodename = eachcard.code
                                if not found_one:
                                    diff_max2 = diff_in_mind
                                    fin_codename = thiscodename
                                    v_card = calculate_card[0]
                                else:
                                    if diff_in_mind > diff_max2:
                                        diff_max2 = diff_in_mind
                                        fin_codename = thiscodename
                                        v_card = calculate_card[0]
```

2.4.如果找到了捡道具的速度并且得分比我们按照原来策略打对手的得分高的话,就返回那个速度了，没找到或者得分不如按照原计划就返回 False。在 velocity_range 函数中，若 cardsornot 返回了 False 回去，则采用原有方案。

```
    if found_one:  # 如果找到了捡道具的速度并且得分比我们按照原来策略打对手的得分高的话,就返回那个速度了,
                   # 没找到或者得分不如按照原计划就返回False
        if diff_max2 >= diff_max1:
            print('codename:', fin_codename)
            return v_card
        else:
            return False

    else:
        return False
```

2.2.5.回球函数-使用道具部分

1.首先介绍 calc_velocity 函数，这一函数在 usingcards 函数中被调用，当道具箱中存有旋转球时用该函数“杀死”对手：
该函数假设对方不改变速度回球，判断此时对方回球是否合规。
1.1.判断己方正常回球速度范围，并且默认此时没有找到特定速度使得对方在不改变速度回球的条件下回球不合规。
1.2.此时存在两个速度范围，分别是正向回球和反向回球，以正向发球为例，反向发球与之相同：
1.2.1.首先，遍历每一个我方合法速度回球，判断此时敌方不改变速度回球是否合规。利用 table 中判断是否 invalid_bounce 的方法（174 行-184 行）：即判断穿墙数目是否合法—是否是一次或两次，否则不合法，即找到合适速度。

```
for v in range(vmin, vmax):  # 正向发球时，在所有发球速度范围内取值
    presume_pos, presume_v = Presume_ball_Status(ball_pos, v)  # 获得打到对方的位置和速度（都是y方向）
    if presume_pos < DIM[3]-10 and presume_pos>DIM[2]+10:
        Y = presume_v * TICK_PER_TURN + presume_pos  # Y是没有墙壁时到达的位置
        if Y % DIM[3] != 0:  # case1：未在边界
            count = abs(Y // DIM[3])  # 穿过了多少次墙（可以是负的，最后取绝对值）
        else:  # case2：  恰好在边界

            # 两种情形：a）  向上穿墙，穿了1 - Y // self.extent[3] 次（代入Y = 0验证）
            #           b）  向下穿墙，穿了 Y // self.extent[3] 次（代入Y = self.extent[3] 验证）
            # 无论怎样，实际位置要么在0，要么在self.extent[3]。直接模( 2 * self.extent[3] )即可。
            # 速度只和count奇偶有关，同上。
            #count = 1
            count = (Y // DIM[3]) if (Y > 0) else (1 - Y // DIM[3])

        if count not in (1,2):
            v_vector = v - ball_velocity  # 速度矢量改变
```

1.2.2 计算此时我方消耗，在找到的各种合适速度中选择我方消耗最小的速度为最终回球速

```
        if count not in (1,2):
            v_vector = v - ball_velocity  # 速度矢量改变
            us_HP_consume = v_vector ** 2 / FACTOR_SPEED ** 2  # 我方消耗
            if found_one == False:
                v_fin = v
                min_usHP = us_HP_consume

            else:
                if us_HP_consume < min_usHP:
                    v_fin = v
                    min_usHP = us_HP_consume
            found_one=True
```

度。
如果没有找到合适方法，按照原有回球策略回球；如果找到合适方法，按照该方法回球。

```
    if not found_one:  # 如果没找到合适的就算了，返回None
        return None
    else:  # 找到了让对方必须换速度的速度，就返回这个速度把对方打死

        #print(v_fin)
        presume_pos, presume_v = Presume_ball_Status(ball_pos, v_fin)  # 获得打到对方的位置和速度（都是y方向）
        #print('到对方',presume_v,'到对方位置',presume_pos)
        v_prsum_ranges = get_legal_velocity(presume_pos)  # 对方回击时返回球速的范围
        #print(v_prsum_ranges)
        return (0, v_fin - ball_velocity)
```

2.之后介绍 usingcards 函数：检查是否存在可用道具，并判断是否使用道具以及此时回球策略。

2.1.通过 table 读取此时的状态：

```
448 def usingcards(tb:TableData,ds:dict):
449     ball_pos = tb.ball.get('position').y  # 对方球y方向到达本方的位置
450     ball_velocity = tb.ball.get('velocity').y  # 对方球y方向到达本方的速度
451     acc_vector = velocity_range(tb, ds)  # 我们需要改变球的速度，矢量
452     cardlist = tb.side['cards']#道具箱
```

2.2.检查此时道具表中是否存在道具

```
454     if len(cardlist) != 0:#如果有卡片的话
455         print('cardlist0', cardlist[0].code)#就检查的时候看一眼用了啥道具
```

2.3.判断种类以及决定是否使用

如果是旋转球，判断是否存在能让对方很可能死亡的回球方法，如果有，就使用；如果没有

```
456         if cardlist[0].code == 'SP':#如果最上面的一层道具是旋转球
457             let_them_die = calc_velocity(ball_pos, ball_velocity)#得到让对方死掉的速度方法
458             print(str(let_them_die))
459             if let_them_die != None:#如果确实找到了这种方法
460                 print('letthemdie')
461                 acc_vector = Vector(let_them_die[0],let_them_die[1])#改变acc_vector到这种方法
462                 card_to_use=cardlist[0]
463                 side_to_use=tb.op_side['name']
464             else:#如果没有找到这种方法
465                 card_to_use=None
466                 side_to_use=None
```

就不使用，并保持默认回球策略。此时调用了上述 calc_velocity 函数。

如果是其他种类道具，立即使用，并用原有回球策略回球：

```
467         else:
468             if cardlist[0].code =='DS':
469                 card_to_use=cardlist[0]
470                 side_to_use=tb.side['name']
471             if cardlist[0].code=='IL':
472                 card_to_use=cardlist[0]
473                 side_to_use=tb.side['name']
474             if cardlist[0].code=='DL':
475                 card_to_use = cardlist[0]
476                 side_to_use = tb.op_side['name']
477             if cardlist[0].code=='TP':
478                 card_to_use = cardlist[0]
479                 side_to_use=tb.side['name']
480             if cardlist[0].code=='AM':
481                 card_to_use = cardlist[0]
482                 side_to_use = tb.op_side['name']
```

最后，将相应的选择返回给 play 打球函数：

```
483     else:
484         return (acc_vector,None,None)
485     #print('card_to_use',card_to_use.code,'side to use',side_to_use)
486     return (acc_vector,side_to_use,card_to_use)
```

2.3.程序限制

目前没有发现程序会出错的情况，我们在多次测试中已经尽可能避免了 bug 的产生。

3.实验结果

3.1.测试数据

实验环境说明：
硬件配置：CPU—2.9 GHz Intel Core i5
内存—8 GB 2133 MHz LPDDR3
操作系统：macOS Sierra 版本 10.12.1
Python 版本：python 3.5.3

我们的测试分为 6 个阶段：

阶段	我方程序	对手	目的	结果
1	T_test22_1	idiot	保障发球和打球过程基本没有 bug，方便后续添加别的策略	全胜且无 bug
2	T_new 增加了对发球速度的最优选择	热身赛 1	探究最基本的策略是否可以在对战中不出 bug	27/105 win for {'invalid_bounce': 4, 'life_out': 1, 'miss_ball': 22} lose for {'invalid_bounce': 6, 'life_out': 5, 'miss_ball': 67}
3	T_new	对角陪练	探究针对打角函数，我方策略是否合适	各有输赢，输赢相当
	T_learntoserve 在发球中加入了“学习”的部分	与之前的自己比较 与陪练比较	探究修改后的代码是否有 bug，进一步探究这一发球策略是否是好的策略	使用了“学习”的发球后，赢局数目显著升高了，但仍是各有输赢
	T_learntoserve 修改了回球函数	与之前的各个对战程序	探究修改纰漏之后是否有显著提升	几乎全胜，偶有输局
4	T_learntoserve 发球、回球已修改	热身赛 2	探究新的发球方法和修改后的回球方式是否有效，评估对手们的水平	71/170 win for {'miss_ball': 52, 'invalid_bounce': 19} lose for {'miss_ball': 93,'invalid_bounce': 6}
5	T_learntoserve	“预测两次”的陪练	探究预测一次和预测两次哪种更好	预测一次结果略好于预测两次
	T_tryeverycards	与之前的程序对战 即 T_learn	探究道具的使用策略	可以通过旋转球杀死缺少相关防守手段的对手

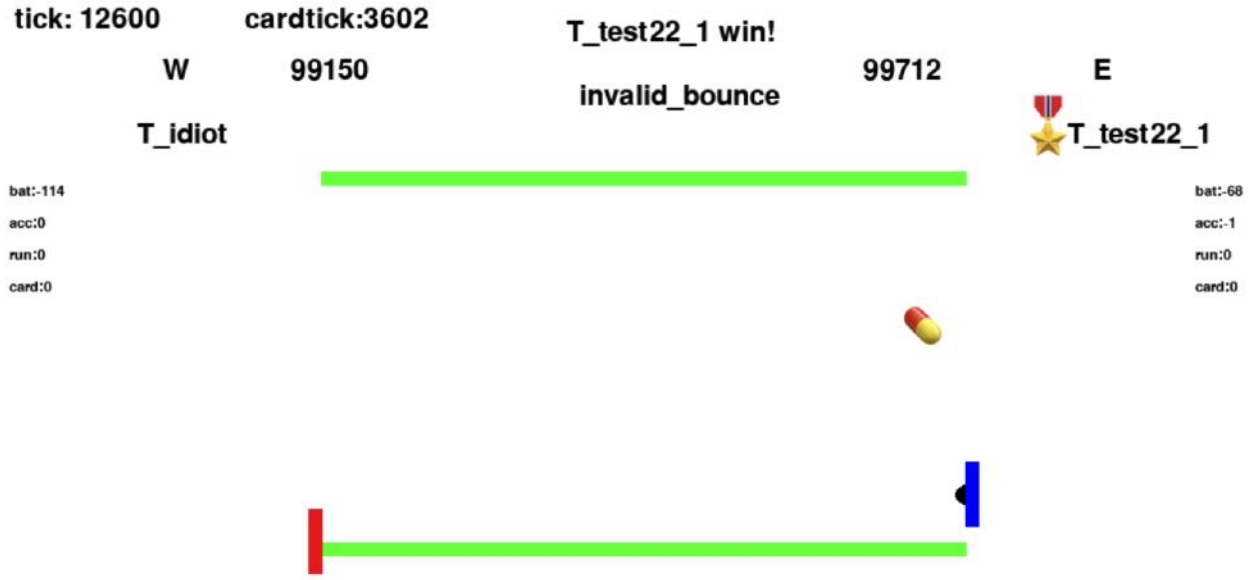
	T_ori 也就是参加比赛的 T_VICTOR	修改跑位问题后与之前我方比较	探究修改纰漏之后是否有显著提升	有显著提升，不遇对方出旋转球时几乎全胜
6	T_VICTOR*	和 GOLF		依然输掉了，赢局数 3/40

一些代码和测试收获会在下一节 3.2.结果分析中体现，打角陪练的代码会以附件的形式上传，正文中不在贴上了。

3.2.结果分析

3.2.1.测试阶段一—和 idiot 对战

我方程序状态：
1.发球为随机发球（地点随机，速度合法）；
2.回球加速度采用对方减我方能耗差最大的方式，和最终类似（但在第一次热身赛之后发现程序虽然没有 bug 但是有失误，之后会详细说明）；
3.跑位函数与最终类似（在第二次热身赛发现程序虽然没有 bug 但是有失误）；
4.没有道具使用。
目的：保障发球和打球过程基本没有 bug，方便后续添加别的策略。
测试结果：和 idiot 交战全胜，因为 idiot 只会被动接球，因而经常因为 invalid_bounce 和 miss_ball 出局。我方程序在反复修改后确认没有运行 bug。
截图示例：图中 T_test22_1 为我方

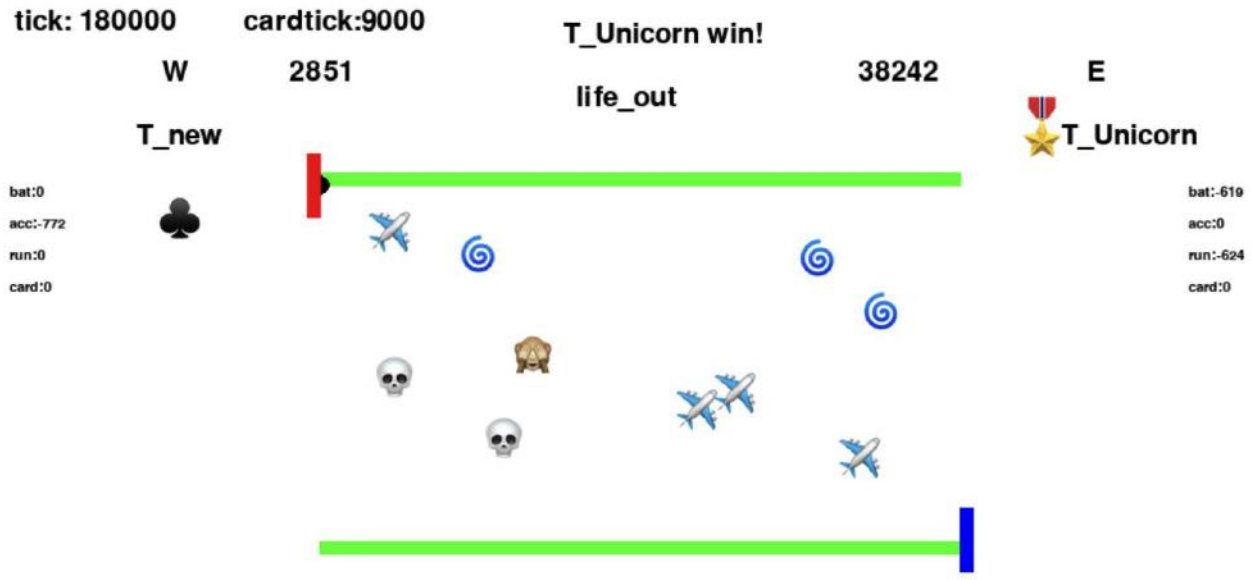


3.2.2.测试阶段二—第一次热身赛

我方程序状态：
1.发球：位置随机，但速度为在函数中求得的“最佳速度”：即使得对方接球的消耗与我方发球消耗的差值最大；

```
def serve(op_side,ds):
    pos = random.randint(DIM[2], DIM[3])
    v_ranges = get_legal_velocity(pos)
    vmin, vmax = v_ranges[0]
    for v_postive in range(vmin, vmax):     #直接取正向速度，否则发球速度损耗过大
        velocity_HP_consume = (v_postive - 1000) ** 2 / FACTOR_SPEED ** 2       #我方改变速度消耗
        presume_op_pos, presume_op_v = Presume_ball_Status(pos, v_postive)
        op_HP_consume = (presume_op_pos - 500000) ** 2 / FACTOR_DISTANCE ** 2   #对方跑位消耗
        difference = op_HP_consume - velocity_HP_consume
        diff_max = 0
        if difference > diff_max:
            diff_max = difference
            v_max = v_postive  # v_max keeps record of v that gives maximum difference
    return pos, v_max
```

2.回球加速度采用对方减我方能耗差最大的方式，和最终类似（但在第一次热身赛之后发现程序虽然没有 bug 但是有失误，之后会详细说明）；
3.跑位函数与小组赛类似，即体力足够时先不跑位（在第二次热身赛发现程序虽然没有 bug 但是有失误）；
4.没有道具使用。
目的：探究最基本的策略是否可以在对战中不出 bug。
测试结果：获胜 27/105，且在观察很多场的对战中，我方因为 miss_ball（67）和 life_out（5）出局。而且对手中极少有使用道具。
截图示例：图中 T_new 为我方



由此，我们分析出可能是回球或者跑位函数不适应大多数对手的策略。注意到很多的对局中，对手都会向对角打，因而编写了一系列的陪练函数用于优化算法，排除策略错误。

3.2.3.测试阶段三—自行测试，与陪练打，修改完善后与之前我方的程序对战

3.2.3.1.和对角陪练对战

打角函数 dajiao2.py 是写来作为陪练的函数，但打角的范围其实很广，有时候只打最“角”的地方会损失很多的体能因此该函数在近似于打角的范围内选择消耗最少的。并且打角消耗不大则打角，打角消耗大则原计划执行。

程序中主要的部分如下两图所示。

```
73      def velocity_range(tb: TableData, ds: dict):
74          ball_op_pos = tb.op_side.get('position').y  # 对方在t0时刻的位置
75          ball_pos = tb.ball.get('position').y  # 对方球y方向到达本方的位置
76          ball_velocity = tb.ball.get('velocity').y  # 对方球y方向到达本方的速度
77
78          Vranges = get_legal_velocity(ball_pos)
79          vranges_postitve, vranges_negative = Vranges            #获得击球时的返回球合法速度，分为正向，反向
80          v_pos_min, v_pos_max = vranges_postitve
81          v_neg_min, v_neg_max = vranges_negative
82          diff_max = 0 #maximum difference of HP
83          lifecost_v=dict()
84          lifecost=list()
85
86
87          for v in range(v_pos_min, v_pos_max):       #在所有合法速度范围内一个一个取值
```

```
104     lifecost.sort()#找出最小消耗生命值
105     print(lifecost)#找到对应的速度
106     if lifecost[0]<=200:#消耗小于200就执行这个
107         return Vector(0, lifecost_v[lifecost[0]] - ball_velocity)
108     else:
109         diff_max = 0  # maximum difference of HP
110         for v in range(v_pos_min, v_pos_max):  # 在所有合法速度范围内一个一个取值
111             prsum_pos, prsum_velocity = Presume_ball_Status(ball_pos, v)
112             v_vector = v - ball_velocity  # 速度矢量改变
113             op_minHP_consume = (prsum_pos - ball_op_pos) ** 2 / FACTOR_DISTANCE ** 2  # 此时对手最小消耗
114             us_HP_consume = v_vector ** 2 / FACTOR_SPEED ** 2  # 我方消耗
115             difference = op_minHP_consume - us_HP_consume
116             if difference > diff_max:
117                 diff_max = difference
118                 v_max = v  # v_max keeps record of v that gives maximum difference
119
120         for v in range(v_neg_min, v_neg_max):  # 在所有合法速度范围内一个一个取值
121             prsum_pos, prsum_velocity = Presume_ball_Status(ball_pos, v)
122             v_vector = v- ball_velocity  # 速度矢量改变
123             op_minHP_consume = (prsum_pos - ball_op_pos) ** 2 / FACTOR_DISTANCE ** 2  # 此时对手最小消耗
124             us_HP_consume = v_vector ** 2 / FACTOR_SPEED ** 2  # 我方消耗
125             difference = op_minHP_consume - us_HP_consume
126             if difference > diff_max:
127                 diff_max = difference
128                 v_max = v  # v_max keeps record of v that gives maximum difference
129         return Vector(0, v_max - ball_velocity)
```

我方程序状态：和热身赛 1 的时候一致。

目的：查看对战对角陪练时更细致的数据，希望找出程序的纰漏。

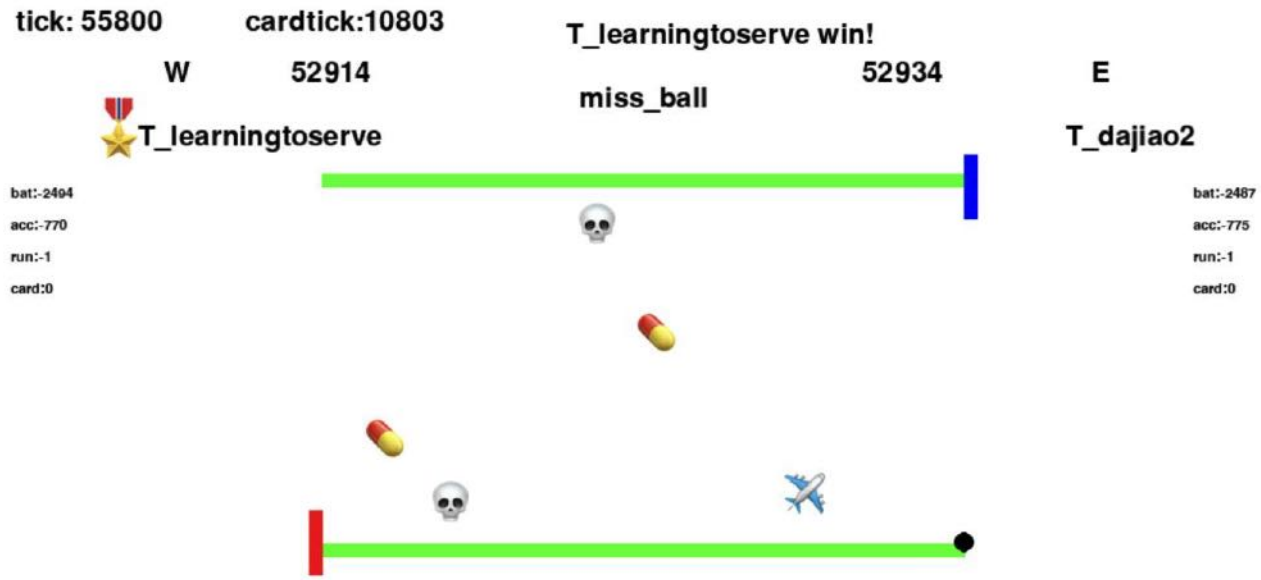
测试结果：和对角陪练各有输赢，赢局数大致相等，所以推测可能还有别的策略问题，而当时没有发现。

3.2.3.2.与之前的自己比较&与对角陪练比较

我方程序状态：在热身赛 1 程序的基础上将发球函数修改成带有“学习”功能的函数，也是最终使用的对战程序中的发球函数。

目的：探究修改后的代码是否有 bug，进一步探究这一发球策略是否是好的策略

测试结果：使用了“学习”的发球后，赢局数目显著升高了，但仍是各有输赢。可见学习的程序是有成效的。
截图示例：其中 T_learningtoserve 是修改成了带有学习的发球函数，而 T_dajiao2 是之前使用过的打角陪练。



3.2.3.3.修改回球函数后，与之前使用过的程序对战
而后在仔细检查回球函数（改变球的速度）时，发现将一个变量带错了，所以导致计算不准确。由于在可视化和 pingpong 的对战结果中很难看出不对劲的地方（因为球还是会大致的向对方的远点打去），但仔细看起来还是找到了致命的错误。
我方状态：修改了回球函数的纰缪
目的：探究修改纰漏之后是否有显著提升
测试结果：几乎全胜，偶有输局
截图示例：图中 T_learningtoserve 是我方这次使用的程序，而 T_test22_1 是之前再热身赛 1 中使用的程序



3.2.4.测试阶段四—热身赛 2

我方程序状态：使用了上述修改过的 T_learntoserve。
目的：探究新的发球方法和修改后的回球方式是否有效，评估对手们的水平。
测试结果：170 局中只赢了 71 局。

3.2.5.测试阶段五—自行测试，与自己的程序对战

3.2.5.1.与回球中“预测两次”的陪练对战
我方程序状态：同热身赛 2 相同。
目的：探究预测一次好还是预测两次好。
测试结果：预测两次没有显著作用，甚至赢局数小于一次预测。而且由于预测两次时，由于回球函数变为 O(n^2)，所以整个程序跑出来很慢。在这次测试后，决定沿用之前的回球函数。

3.2.5.2.加入道具后和以前的程序对战
根据统计，在使用了旋转球后，有 3/4 的概率可以杀死没有防御措施的对手。因而可以得出旋转球是有较大杀伤力的。之后我们又根据不同道具的特点、效用，给道具进行附加分的评选。得出了最终的使用道具方法。
我方程序状态：在热身赛程序的基础上加入了道具的使用
目的：探究道具的使用策略
测试结果：可以通过旋转球杀死缺少相关防守手段的对手。其他道具的使用也增加了胜率。

3.2.5.3.修改跑位函数后和之前的程序对战
在查看比赛结果时发现，我们的程序经常在 50000 左右因为 miss_ball 输掉比赛，观看了可视化后，发现在输掉时，我们只是留在原地不跑位，并没有像计划中的那样，如果生命值有限球拍无法单次移动到最远点就跑位。【这个发现对后来我们程序的改观非常重要！】
而后，发现我们在计算下轮接球速度时，用到的下轮预测生命值计算有误—使用了这一轮的生命值，没有减去这轮中回球和跑位的生命消耗。
我方程序状态：在上一个测试的基础上修改了跑位函数。
目的：探究修改纰漏之后是否有显著提升。
测试结果：有显著提升，不遇对方出旋转球时几乎全胜。

3.2.6.八强对抗—与 GOLF

在八强对抗赛中，第一局以 23：17 的微弱优势赢了 GOLF 之后，GOLF 改变战略以 40：0 赢了第二局，由于我方提交程序比较仓促，有很多策略上想实现但代码上未来得及实现的过程，在我们修改代码的 5min 中进行了实现：
1.修改了跑位函数：如果在偏向一侧的位置，则向中间移动和中间位置差的 0.2 倍（详见代码），以减小跑位和接球的总能耗：

```
538     if needrun_max < runrange_max:  # 如果体力值足够，球拍可以按时到达
539
540         if ball_pos > (DIM[2] + DIM[3]) / 2:  # 如果我们在北半边，则向南移动，更靠近中间
541             run_vector = Vector(0, -0.2*(ball_pos - (DIM[2] + DIM[3]) / 2))
542         else:  # 在南边，则向北移动，更靠近中间
543             run_vector = Vector(0, -0.2*(ball_pos - (DIM[2] + DIM[3]) / 2))
```

2.修改了回球函数，使得对方使用旋转球道具时，我们不会因为 invalid_bounce 出局：

```
512     op_card_active = tb.op_side.get('active_card')[1]
513     if op_card_active == CARD_SPIN:
514         acc_vector = Vector(0, 2 * acc_vector.y)
```

4.实习过程总结

4.1.分工与合作

大作业实习部分分工

项目	策略与代码分工	策略与计算贡献	代码贡献	参赛代码实现
发球	李邦卓/王云蛟/吴放	李邦卓/王云蛟/吴放	李邦卓/王云蛟/吴放/端韵成/谢冠旖	端韵成
回球	谢冠旖/王璐璐	谢冠旖/王璐璐	谢冠旖	谢冠旖
接球与跑位	端韵成	谢冠旖/王璐璐/端韵成	端韵成	端韵成
捡道具	谢冠旖/王璐璐/赵芯露	谢冠旖/王璐璐/赵芯露	谢冠旖/端韵成	谢冠旖/端韵成
使用道具	谢冠旖/王璐璐/赵芯露	谢冠旖/王璐璐/赵芯露/端韵成	谢冠旖/赵芯露/端韵成	赵芯露/端韵成
模拟对战和测试	端韵成/谢冠旖			

大作业报告分工

项目	人员
发球算法思想&程序代码说明	李邦卓/王云蛟/吴放
回球算法思想&程序代码说明	王璐璐
接球与跑位算法思想&程序代码说明	端韵成
捡道具算法思想&程序代码说明	谢冠旖
使用道具算法思想&程序代码说明	赵芯露
实验结果&实习过程总结	端韵成

合作与交流方式：

1.主要方式：微信线上沟通，包括策略、进度、存在的问题，在微信群中，组长设立每一轮更新和代码的要求，组员们在群中上交策略，代码。
2.组会：共两次

第一次组会：
时间：5 月 20 日
内容：分工、要求、目标
参与人员：端韵成，谢冠旖，王璐璐，赵芯露，王云蛟，吴放（少 1 人）

第二次组会：
时间：5 月 29 日
内容：讨论接下来的安排—每部分新的策略的讨论，代码安排

参与人员：端韵成、谢冠旖、王璐璐、王云蛟（由于通知时间为当天所以人没有来齐，但每部分都有成员参加）





左图为第一次组会参与图：
成员从左到右为：吴放、王云蛟、谢冠旖、赵芯露、端韵成、王璐璐
右图为第二次组会组织图：
忘记拍组会合影了

4.2.经验与教训

小组分工明确，全体成员都要参与到策略与代码的编写工作当中，针对一个打球中的步骤进

行较为深入的思考，并从想法开始就对自己的代码负责，而不是一部分负责策略一部分负责代码。同时，负责策略和代码的同学也会同时负责撰写报告中的相应部分。
不足之处主要在于，见面沟通较少，很多问题难以讨论清楚。线上更新到最后才开始严格执行，编写代码和测试的时间仓促。

4.3.建议与设想

建议： 如果能同步的看到可视化也许会增加课堂上竞赛的趣味性和精彩程度；
建立完善的组员自评互评体系，分阶段进行，以督促组员更多贡献力量；
在比赛中发现跑位的作用大于其他因素，希望能让获胜的队伍分享一下代码（例如如何运用深度搜索），以供学习。

5.致谢

首先感谢老师和助教搭建的平台，让这个小游戏竞技性和趣味性十足，也使得我们在实习中学习和运用了很多数算知识。
在比赛中 VICTOR 组成功出线，这与 V 组组员们的日夜操劳、认真研究策略和码代码是分不开的。
在实习的过程中，十分感谢陈斌老师、冀锐助教、李逸飞特约助教热心的解答问题，这对我们这个经常莫名获得 bug 的小组帮助极大！
最后还要感谢外援彭湃同学，在思考策略和检查代码中都提供了很大的帮助。

6.参考文献

1.数据结构与算法（北京大学地空学院）课程网页和 ppt
2.博主 littlethunder 的专栏 http://blog.csdn.net/littlethunder/article
3.www.python.org
4.https://www.youtube.com/watch?v=NE_KKM0e38s 利用 deeplearning 对打乒乓球的视频资料
5.https://github.com/asrivat1/DeepLearningVideoGames 利用 deeplearning 对打乒乓球的代码资料



FOXTROT

组长：李思彤

组员：刘锣居，缪舜，张鑫，王远，周展，王玉琦

小组赛战绩：E3

数据结构与算法课程实习作业报告

（李思彤*，缪舜，刘锷居，王远，王玉琦，张鑫，周展）

摘要：

本文主要介绍 FOXTROT 小组在数据结构与算法课程学习中的整体实习过程。参与漂移乒乓球赛小组所采用的算法主要思想即两个关键词：搜索和学习。整体来看对战过程，算法通过记录分析敌我双方参赛回合中的各项数据，包括跑位、速度、生命值、道具使用等，以此为基础来模拟对方的对战策略与己方的迎战策略，再结合搜索算法，通过对不同的情况（处于不同位置、采取不同速度）进行取样分析，搜索得到最佳对战方案。算法依靠的数据结构是线性表 list，通过创建 list，添加数据，调用数据，数据排序等实现对赛况的实时记录、对敌我双方的具体情况的分析、对敌方策略的预测和分析以及对己方迎战策略的设计与选择。算法与 idiot 的对战均能取得极高的胜率，尤其是当自己掌握发球权时，能够一球击杀对方；在参与热身赛的过程中，通过对战也发现了己方算法的不足并及时对其进行改进；最后的最终竞赛，第一轮对战，胜局数量位居本区第二，但是由于生命值消耗过大而遗憾无缘下一轮对战。

关键字：

list、学习算法、搜索算法

1 算法思想

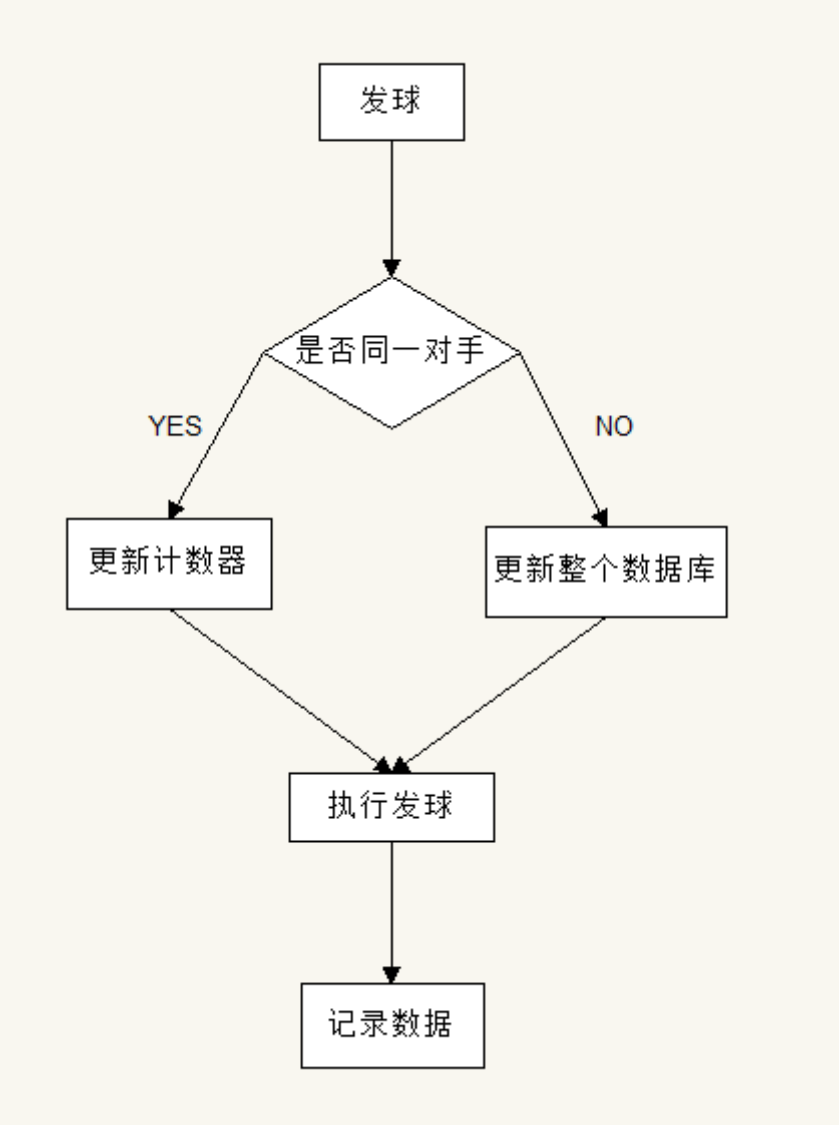
1.1 总体思路

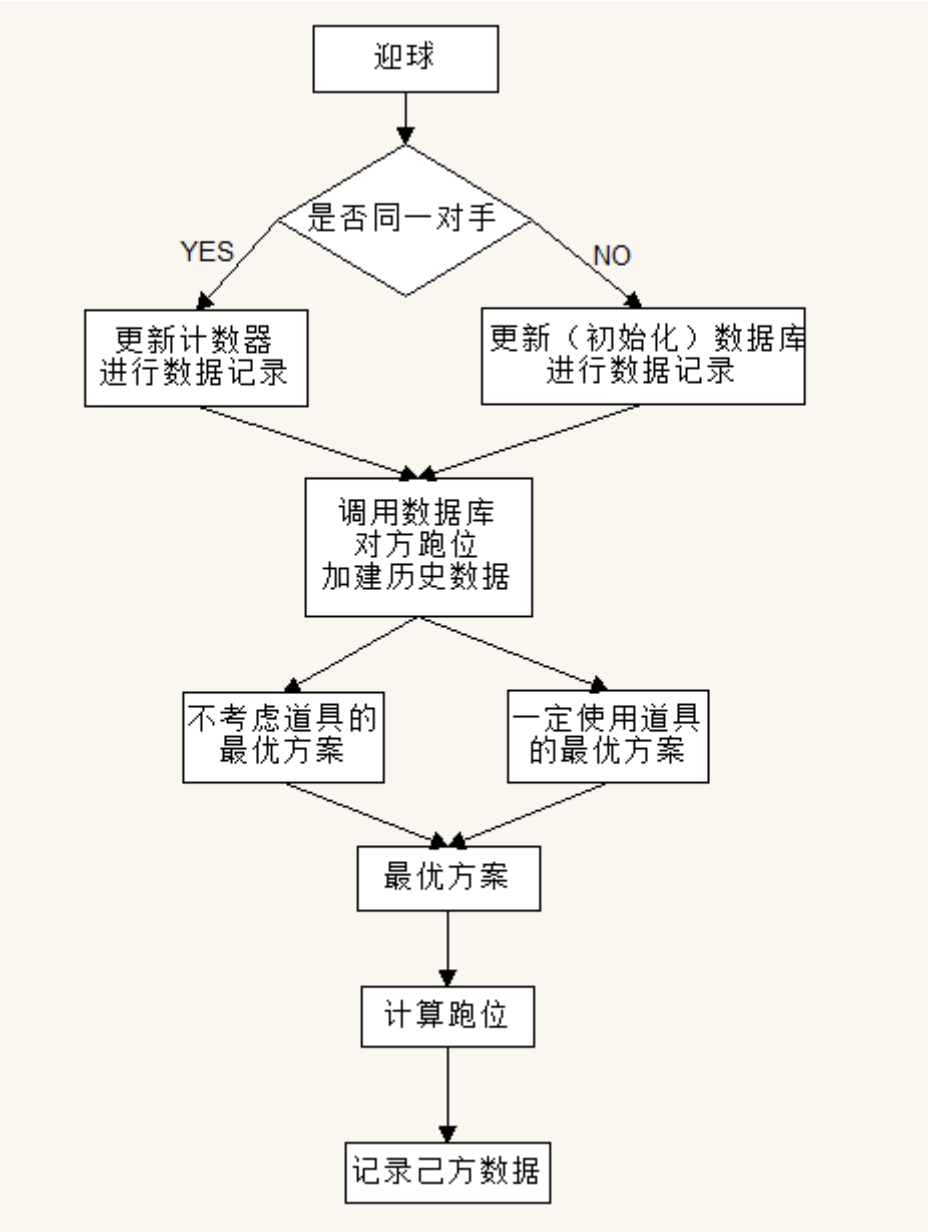
针对本次漂移乒乓球赛，小组设计的算法思想主要包含两个关键词，其一是学习，其二是搜索。算法最主要的对战策略是学习，即，通过与对手的对战，收集并记录下对手的迎战策略（对手的位置、一次击球回合中对手的跑位、对手击球的速度等数据。），将多轮对战收集到的数据汇总在一起进行比对分析，计算己方迎战时所采取的每一种迎战策略的收益损失比，选择能够让己方获利最大的策略与对手对战，力图在赢得尽可能多的对局的情况下，还能最大化地保存生命值。因此，对于我们的算法来说，回合数进行的越多越有利于我们数据的丰富，从而使我们的分析结果更加精确全面，进而让己方对战策略的选择更加完善。

另外，整个程序中必不可少的一种算法是搜索算法，即在选择迎球方案的过程中，通过选择不同的取样点，计算取样点的位置，计算该取样点的最佳加速度以及对手回球的加速度、回球距离等数据，分析对手的跑位策略，计算获得己方的迎球策略方案，通过比对不同方案下最终的收益来决定己方的最终策略。

我们小组算法主要使用的数据结构是线性表 list，创建不同的 list 表对不同的数据进行记录，通过 list 的添加和修改功能来实现对数据记录的更新，对于不同应击球方案的选择都依赖于遍历 list 表，对不同方案实施情况下造成结果的最终收益进行简单的排序。总的来说，算法的最终实现都有赖于 list 的强大与便捷。

1.2 算法流程图





1.3 算法运行时间复杂度分析

本小组算法运行时间复杂度具体分析如下：

首先，发球函数的整体复杂度是 O（1），发球函数主要是进行数据重置或者记录的添加，通过 list 列表的创建与添加来实现。

随着回合或者球桌规模的扩张（以此为 n）,shorten 函数的时间复杂度为 O（1），shorten 函数主要是对四种情况下速度、加速度进行计算，并通过简单的排序以提供选择方案。与 shorten 函数类似，shortenandvalid 函数的是用于选择合法前提下的最小加速度，时间运行复杂度为 O（1）。

随着回合或者球桌规模的扩张（以此为 n），choosewithoutcards 函数的时间复杂度应该是 O（n），其占用 O（n）级运行时间的算法是搜索算法，算法运行时间主要消耗在搜索的过程，即选择己方行动时，所进行的取样分析。我们将球桌分成了 n 点，会通过计算球拍跑位至不同的取样点时的各项数值的消耗或者收益，程序每运行一次，相当于都经历了一次

对于一个长度为 n 的 list 表的遍历与相关计算。函数中调用对手历史数据的算法时间复杂度为 O（1）。

再接着，随着回合或者球桌规模的扩张（以此为 n），随着回合或者球桌上道具规模的扩张（以此为 n），choosetaronlywithcards 函数的时间复杂度为 O（n），其中搜索算法、建立列表进行大小比对大致占用了 O(n)，而调用对手历史数据的算法时间复杂度为 O(1)，但是考虑到真实比赛中，球桌上的道具实际上是有限的一个常数，因而随着球桌规模或者回合规模的扩张（以此为 n），该函数大致只是一个 O（1）级别的函数。
最后，数据存储部分虽然空间复杂度较高，但是时间复杂度整体也是 O（1）级别的。

2 程序代码说明

2.1 数据结构说明

算法主要采用的数据结构是线性表 list。算法的主要工作包括数据的记录、计算与分析比对。因此，我们主要是通过 list 表格或者说 list 表格的嵌套来完成相应的任务。我们运用嵌套 list 建立起算法运行所必要的数据库，通过 list 的 append 来实现数据的添加（数据库的更新），以保证对手每一回合具体数据（跑位、速度、生命值、加速损失、道具使用等内容）的记录以及己方迎战方案的记录都能够顺利完成。整体程序所采用的搜索算法也是建立在 list 这个基础的数据结构之上的。另外，比较重要的己方迎击球策略的选择，也是通过确定不同的方案，计算不同方案下的最终收益，将代表不同方案的收益收入 list 表中，通过排序来为算法提供决策支持。

2.2 函数说明

算法中的各个主要函数：

发球函数（serve），专门控制发球。

打球函数（play），是程序的核心函数。在正式执行 play 函数前，需要建立数据库。与发球函数类似，击球开始前需要检验是否存在对局历史记录，如果有对手的历史记录，直接调用历史记录进行分析。如果没有，则需要重建记录。核心函数里，需要收集对方的初始速度、回球速度，以及对方最终击球速度，计算对方在我方不同使用道具情况下的跑位、赢球的收益（损失）。根据收集所得数据对己方的对战方案进行选择，确定己方跑位预计球策略，己方对于道具采取只要有道具就使用的策略。需要注意的是，每回合结束都要进行充分的数据记录。

其中的 shorten 函数主要是用于选择击球方式：计算在某个点不同击球方式所需要的加速度，选择并返回最小加速度。

Choosetarwithoutcards 函数用于获取不选择道具的情况下的最佳战术。函数主要分为量大模块，其一，在可以选择暴力打角的情况下一定要选则往各个边角击打球。其二，在无法暴力打角的情况下，根据不同行动下收益与损失之差来选择相应的行动。这里我们小组设计了对大量取样点进行计算，计算每个取样点的位置和最佳加速度，由此对对手理想状态下的移动（对手回球加速度、对手跑位距离等）进行猜测，再加上对以往数据的分析，模拟出对手下次的跑位和加速度，判断对手在有效跑位和无效跑位的情况下，己方所选方案的最终收

益，通过简单的排序得出最大收益下的策略。将函数结果留作之后对比分析所用。

Choosetaronlywithcards 函数用于获取一定选择道具情况下击球的不同策略，此函数中需要记录道具的情况，计算的出使用道具情况下的击球速度。同样，也需要根据以往的数据来模拟出对手下次的跑位和加速，判断对手在跑位有效的前提下，适用不同道具带来的收益与损失，并将最终能够获取最大收益的策略记录下来。

Validjustice 函数主要用于判断速度是否符合要求，避免出现 invalid bounce。

Shortenandvalid 函数主要是通过计算不同的速度与加速度，进行简单的排序，来用于选择合法前提下的最小加速度。

finalchoice 函数: 判断不同方案（综合所有情况下的全部策略）下的收益，获取最优选择。

caculate_shortest_length(tb)函主要是用于计算能够迎球覆盖全场的最短跑位距离（在血量 28000～57000 的情况下）。

2.3 程序限制

程序的不足主要集中在两个方面。其一，本算法对道具的判定不够成熟，以至于没能充分利用道具为自己创造利益。其二，基于小组算法的总体思想，作战回合数的数量，对算法的成效影响显著。

对于道具的判定，由于时间限制，小组成员在编写有关判定是否使用道具的代码时，默认了每使用任何一种道具都能固定给己方带来一定的收益（收益值由小组成员提前设定。），而与此同时，默认对方选择道具会给己方带来一定的损失。然而我们知道，是否要使用道具，在何种情况下使用道具，如何使用不同的道具，都是需要更精进更庞大的计算来获取数据进行分析的，使用道具或者不使用道具，使用何种道具，带来的收益与损失都可以通过统计己方或者对方的跑位、击球、体力值等各种数据信息进行对比分析，这样整理数据获得的最终结果支持下的最终决策，才应该是更科学、更严谨也更合理的决策。但是由于时间紧迫，小组成员未能研究出合适的代码来实现上述想法，没能对道具的使用进行更加深入的探索，因此也使得本小组的算法留下了遗憾。但是竞赛过后，小组成员也仍然在对算法进行再加工，希望能够改善这一部分的代码，对算法进行进一步的优化改进。

由于本算法在对战过程中的对战策略，依赖于对对手各项信息的分析判断，来猜测对手的行为，从而预测出己方可以采取的不同策略，然后在不同方案中选取最优方案。也即是说，与对手对战回合数越多，算法所收集到的数据越多、越全面也更加详细，那么算法计算得到的结果也就更有价值，可信度更高。因此我们小组的算法需要更多的时间、更多的回合数进行不断的对战训练，不断地学习成长，需要一定的记录累积才能凸显自己的强大，算法在学习中不断地变强，这既是算法的优点所在，也是限制算法成长的一个重要因素。

3 实验结果

3.1 测试数据

实验环境说明：

硬件配置：Inter(R) Core(TM) i7-5500U CPU @ 2.40GHz

操作系统：Windows 10
Python 版本：pycharm 2016.3.2.1

3.2 结果分析

在算法参与的各项对弈过程中，本算法的学习和搜索策略都起到了显而易见的效果。通过对对手各项数据的收集和分析，来预判对手的迎战方式，从而预测己方可选择的方案并进行优选，能够在一定时间内迅速掌握一定程度上对手的对战策略，从而根据不同对手的对战策略来调整己方的迎战策略与出击策略。唯一遗憾的是，回合数越多本小组算法的优势越明显，因此，可能在对战回合数量不够充足的情况下，对于对手的分析略有差异，己方的迎战策略的选择也略有不足。程序中对于各项数据的充分记录，对于历史数据的留存和调用，也为算法增添了不少优势。

在此算法中，正式执行 play 函数前，我们建立了数据库，记录每一次对战的数据。在执行 play 函数时，可以调用数据库中的数据，检验是否存在对局历史记录，若有则直接调用该策略，若无则需重建记录。回合结束，数据库会收集己方和对手的数据，以供下一回合进行调用。这样的策略意味着本算法会随着对战局数的增多而体现出优势。对战回合数越多，收集到的对手数据就越多、越详细，算法的可信度也就越高。

在发球和击球策略选择时，主要考虑在使用数据和不使用数据的情况下选择最优方案。在发球时，利用 shorten 函数选择击球方式，在发球前计算某个点不同击球方式所需的加速度，并返回最小的加速度。算法主要分为两个部分，使用道具和不使用道具下的损益计算和发球策略选择。不选择道具的战术下，采取暴力打角的策略，若无法暴力打角，则通过击球位置搜索，在 1000 个取样点中计算该点的最佳加速度。同时，根据以往数据猜测和模拟对手的跑位，计算对手不同跑位下己方的收益与损失，选择最大收益的发球策略。在击球算法中，shortenandvalid 函数起了选择合法前提下的最小加速度的作用。在一定选择道具的策略下，choosetaronlywithcard 函数记录道具情况，计算击球速度。并根据以往数据模拟对手下次跑位，进行迎球位置预测。然后计算对手使用不同道具情况下的损益，判断不同方案下的收益并选取最优选择。

该算法的时间运行基本在我们的预期之内，主要的运行时间开销就是我们在“搜索”最合适的跑位的时候，进行了大量的取点计算（1000 个点）。

3.3 经典战局

实习过程中，本算法最出彩的表现，除了在最终的竞赛中以微弱差距输于对手没能获得出线资格意外，值得一提的是本算法与 idiot 的对战中，只要我们掌握发球权，基本可以做到“一球秒杀对手”，这足以说明本算法的罚球函数相当优秀，在对战过程中占据了极大的优势，是能够给对手造成极强的威胁的。

4 实习过程总结

4.1 分工与合作

具体组会记录：

由于我们小组成员分布院系不同并且是由 15 级、16 级两个年级的同学组成，所以平时的组会时间协调相对较为困难。根据小组成员不同的能力划分，我们将小组七个人划分为代码组和报告组。代码组以缪舜为首，主要攻克代码难题，报告组以李思彤为首，主要负责代码攻坚过程中的思路提供以及最终报告的完成。我们的组会以线上、线下相结合的形式进行。



图为小组第一次线下会议

第一次线下组会是在二教讨论区，小组成员到齐，首先，组会对各位成员的分工进行了明确，大概交流了各自对于本次作业的看法。明确了目前课程网站所提供给同学们的各项资源道具，确定了编写漂移乒乓代码之前已经固定的条件，也指出了目前存在的未确定因素。其次，组会探讨了对于击败对手的策略，确定下来小组算法迎战的核心思想，是尽可能做到每一次击球都让对手体力值消耗比己方体力值消耗更大，并且明确地提出，要尽可能地记录下对手的各项数据以供分析学习使用。最后，组会也对道具的使用策略进行了简要的交流。分工明确后，确定了具体的时间轴，明确了代码完成时间、测试时间以及参与竞赛的时间、报告撰写时间等。

第二次线下组会是竞赛结束后，由代码租的代表缪舜同学与报告组进行交流。主要是缪舜对报告组再次大体阐释了算法的整体思路、策略以及不足。将报告所需的数据资料提供给报告小组的成员。报告组在撰写报告之前罗列出了需要代码组协助解决的问题，也一并在此次组会中提出，通过两个小组之间的交流使问题的到解决以保证报告的顺利撰写。

线上组会由代码组缪舜发起，主要针对代码组的几位成员。是在代码撰写的过程中，对代码组几位成员进行更加具体详细的分工，并且对成员的工作进程进行实时的播报。

报告组的线上组会主要是确定报告各个模块内容的分工，所需资料的搜集与整理任务。小组分工：

代码部分：

这一部分的负责人是以缪舜为首的刘锷居、王远和王玉琦代码四人组。他们四个人的具体分工如下：

（1）初期：四个人进行了算法整体结构的设计，第一次组会就讨论了大致的算法框架，并且对于算法需要的模块进行了大致的构想，以及提出了算法在运行过程中可能会面临的问题，并且对一些问题进行了计算和探索，也对于道具的运用进行了讨论。

（2）中期，四个人分工合作，创造出了几个初版的代码进行探讨，几个人分别就自己的任务对不同的算法模块进行改进，在这个过程中，注意尝试为算法添加新的功能。

（3）后期，缪舜、刘锷居两个人主要负责最终代码的完成，包括函数的编写、实现与改进，整个算法功能的添加、修改和完善。四个人共同分担了算法的测试任务，以缪舜为主，对算法进行不断的调试和修改，结合对战复盘数据对算法进行再加工和再完善。最后，由缪舜进行算法结构的适当调整和最终的优化。

报告部分：

报告部分的负责人是以李思彤为首的张鑫、周展报告三人组。在第一次组会期间参与全体对算法结构框架的讨论，提出对算法架构存在的问题和看法，并给出关于比赛的相关建议。三人主要负责后期的报告整理工作。其中，李思彤负责对最终代码注释部分的相关完善，在报告中负责摘要的阐述、总体思路的简要叙述、程序代码说明模块对程序数据结构的说明、函数的说明、程序限制的说明以及实验结果的简要概括；周展负责算法流程图的设计，对本次实习过程中的经验教训的概括与总结，以及对数据结构与算法课程实习作业的建设与改进；张鑫负责实习过程中所需资料的收集与整理，对实习过程中经验与教训的总结，以及后期的建议与设想模块，对不同部分内容整合以及最终对报告的整体进行加工和完善。特别提出，缪舜同学提供了报告中算法时间运行复杂度的分析。

4.2 经验与教训

在实习过程中，我们在分工和代码实现这两块上，既有做的好的方面，也有不足。

在分工上，小组充分考虑了每位组员的优势和缺陷，选择编程能力高的组员进行代码的编写、测试等。编程能力弱的组员则负责其他技术性较弱的工作，如撰写报告、提问题和看法。这样的分工很科学，既充分发挥了组员的才能，又不会导致任务量的不平均。在这样的分工下，我们的实习最终也获得了不错的成果。但这样的分工同样也会导致非编程人员对于程序的了解不够深入，最后撰写报告和总结时无法准确叙述编程过程，因此，我们在编写实验报告时针对此问题进行了一次线下交流，通过细致的交流也很好的解决了这一问题。在实现代码方面，我们组员采取的是以一两人主导，其他组员协助的方式进行，可能会导致在算法实现的过程中忽略许多有独创性的、个性的想法，如果在一开始提出想法时每位组员针对不同的算法进行实现，或许可以获得更多竞赛方式，同时互相进行优化。但是我们以一两位组员进行统筹，在资源搜集和利用效率得到了很大的提高，齐头并进，在优化算法效率方面得到了很大的弥补。

在代码实现方面，基本上完成了功能设计，并且在对道具的使用方面做出了优化，同时在热身赛的过程中也取得了不错的竞赛成绩，但代码仍存在一些问题，需要改进，如对道具

判定的不成熟和作战回合数的限制等。因此，在出线赛中从获胜局数来看，获得了小组第二名的好成绩，但是由于赛制的限制，体力值损耗略大于平局的另一小组，遗憾于小组赛。由此，我们在时间对战的过程中更应该实时优化自己的代码武器，可以有针对性地进行策略调整，在不断优化的过程中，相信我们也能收获更大。

此外，小组开组会的次数较少，这对于小组成员进行交流有一些限制。

由于我们全组线下的组会只开了一次，其余基本都是线上交流，或者负责编程的几个组员进行线下交流，全部组员的交流较少。这可能也导致了组员想法的想法没能充分地采用，对算法的讨论深度不够，导致我们在整体的算法架构和程序设计上存在漏洞和不足。

在第一次的小组会议中，我们对整体漂移乒乓的算法策略进行了讨论，组员更是饶有兴趣地拿着笔和纸开始了思想交锋，在这一过程中，我们进行了队每位小组成员进行了了解，进行了小组的大致分工，和初步想法的设计。在这一过程中，从对实习主题的简单探讨，再到使用策略的深入分析，都对我们后来算法的设计和优化打下了良好的基础。在第二次的小组会议中，主要是编程小组的人员进行了交流，他们对初步算法进行了讨论，并针对其他小组可能出现的算法进行了研究和比对，加上了使用道具的模块，使得整个算法体系更加完善和科学。在整个实习进行过程中，我们小组时刻保持线上的交流，有了新的想法直接在微信群中与组员进行有效的探讨，在有需要的情况下，约定线下交流的时间。对于这一过程中产生的问题和需求，组长进行有组织的协调，与助教和老师积极进行沟通，例如关于对战平台的使用问题等，这对于我们小组进行下一步的测试实验有很大的助益。因此在多次的线上线下的交流过程中，我们小组一直和谐有序地进行着实验任务，这是我们小组宝贵的财富，也启发我们在今后的合作项目中，一个合理分工、各司其职、协同一致的小组，将会是顺利完成项目的最大助力。

4.3 建议与设想

建议：

我们认为在时间允许的情况下，可以增加对战的局数。如此一来，在对战时能够出现更多的随机情况，也能够实现对代码的充分利用和开发，更加有利于每组根据自己的局限性进行有针对性的修改和优化。当然，在课后也可以让更多的小组进行对战，而不再是同区小组之间的交流，通过与更多的算法进行混战，以及各小组之间算法的交流，或许可以获得更多有趣的想法和严谨的思路；在每个小组进行备战的过程中，也可以在群里进行充分的交流，互相优化，取长补短。

改进：

在代码上，可以参考坦克大战的算法设计，尝试使用路径搜索算法和碰撞检测等算法，如 dijkstra 算法、SPFA 算法等。代码应当在运行中，通过统计己方或者对方的跑位、击球、体力值等各种数据信息进行对比分析，整理数据获得的最终结果，做出最终决策，而不是通过预定好的收益值进行决策。总之，算法与程序是在不断的对战竞赛中进行优化，在对战中收集更多数据，为最优决策的选择提供更多更精准的参考。

而对于本次的实习作业，组队上可以进一步细化，以及小组讨论交流时间应加长。我们小组认为 6（+-1）人组成一个小组十分合理，因为组队的随机性，使得小组可能会有十分擅长编程的队员，也有不善于编写代码的队员，一方面可以保证实习项目的完成性，另一方面也可以激发各小组成员的创造性和协作性。一个项目不能由一个人单独完成，因此十分考验小组的协作性，问题和改进双管齐下，才能在较短的时间内取得进步，而我们小组在这一方面的欠缺就在于线下交流有些过少。因此我们也在意识到这个问题后，加强了小分队的线交流，以及全员的微信群交流。

除此之外，我们小组在总结的过程中也认为在代码实践的过程中，应当进行更多次的代码测试，在测试中寻找问题，通过不断优化完善，我们的程序才会更加具有抵抗性。在最终的测试中，小组的程序很遗憾没能进入总决赛，虽有遗憾，但也没虚度。

给学弟学妹们的寄语：

数据结构与算法这门课是非常有趣的，学好这门课不仅能让自己的专业知识水平提高，课程中的算法和思想还能应用在生活中的方方面面。在学习这门课程中，希望学弟学妹能够认真听课，认真理解老师的授课内容。在学习中，要在概念的基础上构建知识框架。在学习算法时，掌握它的思想和用法，并在平时要勤练习，多使用这些算法进行编程，在实践中更加深刻地理解它，这样才能真正学好这门课程。

后续工作的大胆设想：

我们小组认为可以将大家的算法整合，开发一个完善的漂移乒乓的游戏，根据大家代码的攻击性，可以分为人机对战模式、双机对战模式等，以及可以划分难易程度。让大家的算法成果得到更多的交流。另外，实习作业的算法不仅仅可以用在漂移乒乓游戏的设计中，可以利用里面的算法和程序来设计更高级、更复杂的游戏或应用，如坦克大战等二维游戏。

5 致谢

首先，在本次课程作业的实习中要感谢课程教师和助教老师对于我们小组所提供的各项支持与帮助，谢谢老师为小组成员进行答疑解惑，为小组代码提供测试平台，依照各小组成员的建议与要求改进本次实习作业的各项内容，并应各小组建议提供热身赛，将热身赛数据进行收集导出并返回给各小组成员，为小组改进和完善代码提供了极大的帮助。

其次，要感谢本小组各位成员的努力付出。不论是代码小组辛苦地编写、调试、修改代码还是报告小组成员收集资料，整理报告，大家都付出了自己的努力，充分保证了本次实习作业的顺利完成。在这里尤其要感谢缪舜同学，不仅协助组织进行了实习作业的核心——代码的编写和完善，也为报告小组撰写报告提供了极大的帮助。

另外也要感谢其他小组在本次实习作业中的努力付出，为本小组的整体进步提供了经验教训与优点借鉴，其他小组的优秀之处一定程度上也促进了我们小组对自我的完善和发展。

6 参考文献

[1]http://www.cnblogs.com/LanTianYou/p/4544614.html

[2]https://baijiahao.baidu.com/po/feed/share?wfr=spider&for=pc&context=%7B%22sourceFrom%22%3A%22bjh%22%2C%22nid%22%3A%22news_3326764760834642637%22%7D

[3] https://www.codeforge.cn/article/242258



ECHO

组长：蒋天骥

组员：李婧宇，王宇萱，杨家鑫，黄骁，李浩，周广财

小组赛战绩：E4

数据结构与算法课程作业

“漂移乒乓”实验报告

ECHO 组

1 算法思想阐述

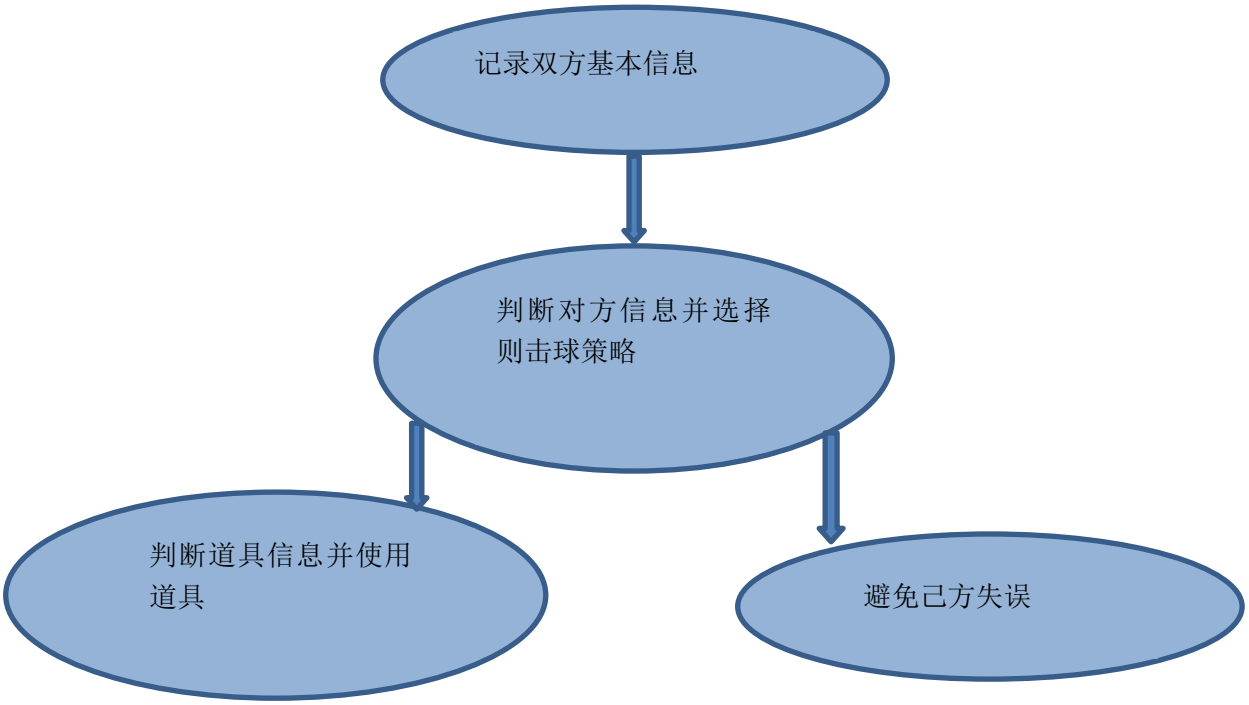
1.1 总体思路

本次课程作业最终提交代码的主要算法思想是最大杀伤结合基本的学习策略（累计记录对手上一回合击球速度和击球点，在己方策略中选择下一回合能使得对方体力消耗与己方体力消耗之差最大的策略），

在开局没有对方记录的情况下，己方默认对方采用最大保全的打角策略，依此选择相应策略击球。从对方第一次击球开始记录己方和对方每一次的击球速度和击球点，每一回合根据己方击球速度，击球位置和上一回合对手击球加速度，推测对手击球加速度，从而选择击球策略和跑位方向。以对方体力消耗和己方体力消耗之差为准遍历所有策略，从中选择差值最大的策略。

在道具选用方面，我们只选择变压器和旋转球两种道具。在这两种道具之一出现在桌面上的时候，选择能吃到道具的路径，吃到后马上使用。在使用变压器的情况下，己方下一次击球采用最大杀伤策略，选择较大的加速度。在使用旋转球的情况下，己方下一次击球采用最大保全策略，选择较小的加速度。

1.2 算法流程图



1.3 算法运行时间复杂度分析

本算法运用的非线性结构较少，主要运用的是列表、字典的线性结构。这就是我们的代码具有较为“亲民”的风格，即通俗易懂，不高深莫测。但同时带了的就是时间复杂度的代价，本算法中包含的赋值语句、循环结构、判断结构比较多，因此时间复杂度较大。

2 程序代码说明

2.1 数据结构说明

本算法运用的数据结构是列表、字典，充分利用线性结构，记录乒乓球的位置、加速度等信息。同时定义了多个函数，通过函数调用加快算法运行效率。同时满足击球、判断对方位置、吃道具等多项功能。

2.2 函数说明

2.2.1 记录双方击球信息

在击球函数中定义字典 ds 来记录双方击球信息。包括己方击球点、己方给球的加速度、对方击球点、对方放接球时的球速、对方给球的加速度，其中每一类信息都已列表类型保存在字典中，每一回合将击球信息采用 append 方法添加进对应列表中。每出现一个新对手，己方都会重新设置这些用于记录的变量。具体代码块截图如下：

```
op_speed = list() #op_spped存储的是本回合结束回合对手的回球速度

op_getspeed = list()#op_getspeed是本回合击球后下一回合对手所面临的球速
op_hitpoint = list()#op_hitpiont是本回合击球后下一回合对手的击球点
my_addspeed = list()
my_hitpoint = list()
```

```
ds = {"my_hitpoint": my_hitpoint, "op_speed": op_speed, "op_hitpoint": op_hitpoint, "my_addspeed": my_addspeed,
      "op_getspeed": op_getspeed}
ds["my_hitpoint"].append(tb.ball['position'].y)
'''
    op_getspeed[i]是第i回合对手接到球的速度
    op_op_speed[i]是第i-1回合对手给球的加速
'''
if len(ds["op_getspeed"]) == 0: #如果是对方发球的首回合，那么记上一回合结束对手的的回球加速为0
    ds["op_speed"].append(0)

else:
    enermy_getspeed = ds["op_getspeed"][len(ds["op_getspeed"]) - 1]
    enermy_add_speed = find_opp_addspeed(tb.ball['velocity'].y, enermy_getspeed, tb.ball['position'].y)
    ds["op_speed"].append(enermy_add_speed)
```

```
def is_new_oponent(name):
    global setpoint
    global namecard

    global op_speed
    global op_hitpoint
    global op_getspeed
    global my_addspeed
    global my_hitpoint
    if setpoint == 1:
        namecard = name
        setpoint = 0
    else:
        if namecard != name:
            namecard = name
            setpoint = 1
            op_speed = list()
            op_getspeed = list()
            op_hitpoint = list()
            my_addspeed = list()
            my_hitpoint = list()
```

2.2.2 推测对方击球信息

己方对对方击球信息的推测主要是对方击球时给球的加速度。根据己方上一回合击球点、击球加速度和落球点，函数返回对方击球加速度，从而选择跑位和击球加速度等策略。另外，函数 findlocation 用于倒推对手击球的位置，这里主要考虑对手使用隐身道具的情况。具体实现是通过球触碰球桌边缘的次数来进行判断。程序相关部分截图如下：

```
def findlocation(position, vy):#本函数用于倒推对手击球的位置
    sy = -vy*crosstime + position
    location = 0 #初始化location,location为对方击球时位置的y坐标
    if sy > 0:
        if sy > DIM[3] and sy < 2*DIM[3]:#当球触碰上边一次时
            location = DIM[3] - abs(sy - DIM[3])
        elif sy >= 2*DIM[3]:
            location = sy - 2*DIM[3] #当球先触碰下边再触碰上边时
    else:
        if abs(sy) < DIM[3]: #当球触碰下边一次时
            location = -(sy)
        elif abs(sy) >= DIM[3] and abs(sy) <= 2*DIM[3]: #当球先触碰上边再触碰下边时
            location = sy +2* DIM[3]
    return location
```

```
def find_opp_addspeed(my_getspeed,opp_last_getspeed,my_hitpoint): #find_opp_addspeed是根据本回合我方接球位置，接球速度，和上一回合对手接球的速度
    # 来测算对手给球的加速
    the_speed = my_getspeed * counttouchpoint(- my_getspeed,my_hitpoint)
    return the_speed - opp_last_getspeed
```

```
def get_opp_getspeed(my_add_velocity,orgin_velocity,my_hitpoint): #获取对手接到球时球的速度
    the_velocity = my_add_velocity + orgin_velocity
    the_velocity = counttouchpoint(the_velocity,my_hitpoint)*the_velocity
    return the_velocity
```

得知对手击球加速度和击球位置后，就可以计算对手本次击球的体力消耗，用于跟己方策略集中的所有策略体力消耗作对比从而选择策略。

```
def enermycost(hitplace,initialv,ds:dict,tb):
    enermy_move = hitplace - tb.op_side['position'].y

    if judge_opp_addspeed(hitplace,initialv,ds) != -1:
        enermy_add_speed = judge_opp_addspeed(hitplace,initialv,ds)

    else:
        if tb.side['position'].y <= DIM[3]/2:
            enermy_add_speed = enermy_chooseroute(hitplace,DIM[3],initialv)
        else:
            enermy_add_speed = enermy_chooseroute(hitplace, 0, initialv)

    return (enermy_move / FACTOR_DISTANCE) ** 2 + (enermy_add_speed / FACTOR_SPEED) ** 2
```

```
def costcounting(mymove, mycost, enermycost):
    MY_Cost = (mymove / FACTOR_DISTANCE) ** 2 + (mycost / FACTOR_SPEED) ** 2
    return enermycost - MY_Cost
```

2.2.3 选择击球策略

基于上述收集到的双方击球信息，己方选择每一回合的击球策略。在 chooseroute 函数中，己方在击球最大触边数的前提下，分别讨论每一种触边数和触边类型的所有路径，遍历所有策略返回对手体力消耗与己方体力消耗差值最大的策略。enemy_chooseroute 函数主要考虑了对手采用己方未记录的策略的情况，在这种情况下，函数推测对手采用最大保全策略，具体做法是遍历对手策略集计算每一个策略的体力消耗，选定对手体力消耗最小的那一种作为默认的对手策略，并依此确定己方策略。

```
def chooseroute(touchpoint,target,initialv,move,tb,ds):#touchpoint:击球点，#target:球击中对侧的目标位置，initialv：击球前球在y方向上速度

    MAX_touch = 2#MAX_touch是球行进过程中允许的最多触边次数

    touchpointupper = DIM[3] + DIM[3] - touchpoint #touchpointupper1:击上边1次的情况
    touchpointunder = -touchpoint #touchpointunder1:击下边1次的情况

    targetupper = DIM[3] + DIM[3] - target
    targetunder = -target

    velocity = (target - touchpoint)/crosstime
    velocityup1 = (targetupper -touchpoint)/crosstime #这里可以画一个草图来研究一下
    velocitydown1 = (targetunder - touchpoint)/crosstime
    velocityup2 = -(targetunder -touchpointupper)/crosstime
    velocitydown2 = -(targetupper - touchpointunder)/crosstime
    if target == 0 or target == DIM[3]:#球的目标是对方的球桌顶点，相当于球在被对方击回前还有一次触边，球在行进过程中可触边的次数-1
        MAX_touch = MAX_touch - 1

    if len(tb.side['cards']) != 0:
        for j in tb.side['cards']:
            if j.__eq__('AM'):
                if MAX_touch == 1:
                    opp_cost1 = judge_opp_addspeed(getlocation(touchpoint,velocity + tb.ball['velocity'].y),
                                                   get_opp_getspeed(velocity,tb.ball['velocity'].y,touchpoint),ds)
                    opp_cost2 = judge_opp_addspeed(getlocation(touchpoint,velocityup1 + tb.ball['velocity'].y),
                                                   get_opp_getspeed(velocityup1,tb.ball['velocity'].y,touchpoint),ds)
                    opp_cost3 = judge_opp_addspeed(getlocation(touchpoint, velocitydown1 + tb.ball['velocity'].y),
                                                   get_opp_getspeed(velocitydown1, tb.ball['velocity'].y, touchpoint), ds)
```



```python
def enermy_chooseroute(touchpoint, target, initialv):#touchpoint:击球点，#target:球击中对侧的目标位置，initialv：击球前球在y方向上速度

    MAX_touch = 2#MAX_touch是球行进过程中允许的最多触边次数

    touchpointupper = DIM[3] + DIM[3] - touchpoint #touchpointupper1:击上边1次的情况
    touchpointunder = -touchpoint #touchpointunder1:击下边1次的情况

    targetupper = DIM[3] + DIM[3] - target
    targetunder = -target

    velocityup1 = (targetupper -touchpoint)/crosstime #这里可以画一个草图来研究一下
    velocitydown1 = (targetunder - touchpoint)/crosstime
    velocityup2 = -(targetunder -touchpointupper)/crosstime
    velocitydown2 = -(targetupper - touchpointunder)/crosstime
    mincost = min(abs(velocitydown1 - initialv),abs(velocityup1 - initialv),abs(velocitydown2 - initialv),abs(velocityup2 - initialv)
    if abs(velocitydown1 - initialv) == mincost:
        return velocitydown1 - initialv
    elif abs(velocityup1 - initialv) == mincost:
        return velocityup1 - initialv
    elif abs(velocitydown2 - initialv) == mincost:
        return velocitydown2 - initialv
    elif abs(velocityup2 - initialv) == mincost:
        return velocityup2 - initialv

def find_out_route(touchpoint, velocity, move, tb, ds):
    enermy_hitpoint = getlocation(touchpoint, velocity + tb.ball['velocity'].y)
    enermy_getspeed = get_opp_getspeed(velocity, tb.ball['velocity'].y, touchpoint)
```

该算法的学习理念的主要体现在于 judge_opp_addspeed 函数。函数判断当前落点如果与落点记录中的某一个落点距离差小于 2000，且球到达速度与速度记录中同一个回合的到达速度差值小于 2000，则认为对手采用了击球记录中那一回合所用的同一策略，己方就按照所记录策略迎战。后来反思中，我们发现用 2000 来限定速度差并不科学，我们推测在合理计算的基础上适当减小这一标准能够一定程度上提高胜率。

```python
def judge_opp_addspeed(hitpoint, op_get_speed, ds:dict):
    global op_speed
    global op_hitpoint
    global op_getspeed

    for i in range(len(ds['op_hitpoint'])):
        if abs(hitpoint - (ds['op_hitpoint'][i]))< 2000:
            if abs(op_get_speed - ds['op_getspeed'][i]) < 2000:
                if i != len(ds['op_hitpoint']) - 1:
                    return ds['op_speed'][i+1]

    return -1
```

2.2.4 获取和使用道具

这部分内容包括如何选择获取道具的优先级，如何使用。初始化道具箱和使用道具的函数如下：

```
def usecard(tb):
    card = None
    if len(tb.side['cards'])==0:#道具箱为空
        card = None
    else:
        for j in tb.side['cards']:
            if j.__eq__('AM'):
                return j
        card = tb.side['cards'][len(tb.side['cards'])-1]
    return card

# 使用道具对象的函数
def cardside(card):
    if card == None:
        return None
    elif card.__eq__('SP') or card.__eq__('DL') or card.__eq__('AM'):
        return 'OPNT'
    else:
     return 'SELF'
```

下图所示条件语句用于判断是否能在不违反规则的条件下获得道具

```
if Catchcard_Cost != -10:
    ds["my_addspeed"].append(Catchcard_Cost)
    velocity = get_opp_getspeed(Catchcard_Cost, tb.ball['velocity'].y, tb.ball['position'].y)
    ds["op_getspeed"].append(velocity)
    ds["op_hitpoint"].append(getlocation(hitplace,Catchcard_Cost))
    return RacketAction(tb.tick, move,Catchcard_Cost,
                        running(tb.side['life'], tb.ball['position'].y),cardside(usecard(tb)),usecard(tb))
else:
    ds["my_addspeed"].append(MY_COST)
    velocity = get_opp_getspeed(MY_COST, tb.ball['velocity'].y, tb.ball['position'].y)
    ds["op_getspeed"].append(velocity)
    ds["op_hitpoint"].append(getlocation(hitplace, MY_COST))
    Y = (MY_COST+tb.ball['velocity'].y)*1800 + tb.ball['position'].y
    count = (Y // DIM[3]) if (Y > 0) else (1 - Y // DIM[3])
   # print(Y,isinvaild(tb,MY_COST + tb.ball['velocity'].y),count)
    return RacketAction(tb.tick, move, MY_COST,
                        running(tb.side['life'], tb.ball['position'].y), cardside(usecard(tb)),usecard(tb))
```

下面函数为了获取离道具圆心 2000 的切点位置，规划获取道具的路径。

```
def cardpoint(tb:TableData,card:Card):#求切点

    class point:
        def __init__(self, p1, p2):
            self.p1 = p1
            self.p2 = p2

    #落点到圆心距离
    d = math.sqrt((card.pos.x - tb.ball['position'].x)*(card.pos.x - tb.ball['position'].x)+(card.pos.y - tb.ball['position'].y)*(card.pos.y -
    #落点到切点的距离
    l = math.sqrt(d*d-r*r)
    #点到圆心的单位向量
    x0 = (card.pos.x - tb.ball['position'].x)/d
    y0 = (card.pos.y - tb.ball['position'].y) / d
    #切线与点心连线的夹角
    f=math.asin(r/d)
    #正反两个方向旋转单位向量
    x1 = x0 * math.cos(f) - y0 * math.sin(f)
    y1 = x0 * math.sin(f) + y0 * math.cos(f)
    x2 = x0 * math.cos(f) - y0 * math.sin(f)
    y2 = x0 * math.sin(f) + y0 * math.cos(f)
    #新坐标
    x1 = (x1 + card.pos.x) * 1
    y1 = (y1 + card.pos.y) * 1
    x2 = (x2 + card.pos.x) * 1
    y2 = (y2 + card.pos.y) * 1
    point.p1=(x1,y1)
    point.p2=(x2,y2)
    return point
```

下面 getwhichcard 函数具体规定了算法中只获取变压器和旋转球两种的具体判断。

```
def getwhichcard(tb:TableData):#只获取变压器和旋转球
    list1=list()
#分别用于存四种优先级的道具的加速度
    if len(tb.cards['cards']) == 0:#场上没有道具
        return False
    else:
        for card in tb.cards['cards']:
            if togetcard(tb, card) != False:
                if card.__eq__('AM')or card.__eq__('SP'):
                    list1.append(togetcard(tb,card))
        if len(list1)!=0:
            return min(list1)
        else:
            return False
```

下面 togetcard 函数具体规划为了获取道具的击球路径。

```
def togetcard(tb:TableData,card):
    t1 = (card.pos.y+1410 - tb.ball['position'].y)/abs(card.pos.x-1410- tb.ball['position'].x)
    t2 = (card.pos.y-1410 - tb.ball['position'].y)/abs(card.pos.x +1410-tb.ball['position'].x)
    vy1 = t1 * BALL_V[0]
    vy2 = t2 * BALL_V[0]
    acc1 = vy1 - tb.ball['velocity'].y
    acc2 = vy2 - tb.ball['velocity'].y
    acc_list= list()
    a1=max(acc1,acc2)
    a2=min(acc1,acc2)
    for acc in range(int(a2),int(a1)):
        if isvaild(tb,acc+tb.ball['velocity'].y)== True:
            acc_list.append(acc)
    if len(acc_list)==0:
        return False
    else:
        min_acc = acc_list[0]
        for acc in acc_list:
            if abs(acc)<abs(min_acc):
                min_acc = acc
        return min_acc
```

2.2.5 避免己方失误

这一部分主要内容包括防止贴边飞行和判断是否合法碰壁，以及跑位策略。

贴边飞行情况指己方击球点在桌角时，向同侧桌角击球的策略，这会导致己方由于 invalid_bounce 输球。这一漏洞多次出现在组内前几版的代码中，因此在最终代码中我们特别进行了判断。如果己方击球点为桌角顶点，选择策略时按照所有碰边的情况只选择最终落点在另一边桌角附近的策略，防止己方失误导致失败。

```
if hitplace <= 2000 or hitplace >= DIM[3] - 2000:  # 这里讨论击球点是否为顶点，是为了避免贴边飞行的情况出现
    i = 0
    if hitplace <= 2000:
        if tb.op_side['position'].y <DIM[3]/2:
            MY_COST = int(chooseroute(hitplace,DIM[3] - 2001,tb.ball['velocity'].y,move,tb,ds)) + 1
            while not isinvaild(tb,MY_COST+tb.ball['velocity'].y):
                if i < 1000:
                    MY_COST = MY_COST + 1
                    i = i + 1
                elif i >= 1000:
                    MY_COST = MY_COST - 1
        else:
            MY_COST = int(chooseroute(hitplace,2001, tb.ball['velocity'].y, move, tb, ds))-1
            while not isinvaild(tb,MY_COST+tb.ball['velocity'].y):
                if i < 1000:
                    MY_COST = MY_COST + 1
                    i = i + 1
                elif i >= 1000:
                    MY_COST = MY_COST - 1
    elif hitplace >= DIM[3] - 2000:
        if tb.op_side['position'].y <DIM[3]/2:
            MY_COST = int(chooseroute(hitplace,DIM[3] - 2001,tb.ball['velocity'].y,move,tb,ds))+1
            while not isinvaild(tb,MY_COST+tb.ball['velocity'].y):
                if i < 1000:
                    MY_COST = MY_COST + 1
                    i = i + 1
                elif i >= 1000:
                    MY_COST = MY_COST - 1
        else:
            MY_COST = int(chooseroute(hitplace,2001, tb.ball['velocity'].y, move, tb, ds))-1
            while not isinvaild(tb,MY_COST+tb.ball['velocity'].y):
```

另一项内容是判断是否合法碰壁。合法碰壁的要求是碰壁次数为 1 或 2，定义函数 isinvalid 通过 y 方向球速来判断函数是否碰壁，bounce_count 记录碰壁总数，函数返回布尔值。

```
def isinvaild(tb:TableData,vy):#判断是否合法碰壁
    Y = vy * crosstime + tb.ball['position'].y
    if Y % DIM[3]>2000:
        count = Y //DIM[3]
        bounce_count=abs(count)
    else:
        count = (Y // DIM[3]) if (Y > 0) else (1 - Y // DIM[3])
        bounce_count = count
    if bounce_count in (1,2):
        return True
    else:
        return False
```

跑位策略主要包括开始跑位的时间和跑位方向等。跑位的好处在于可以较快到达击球

点，缺点在于比较耗费体力。我们认为己方体力小于一定值时开始跑位能够平衡体力消耗和保证击到球的关系，通过实践，最终将 60000 定为开始跑位的临界体力值。

```
def running(myenergy, myposition):
    if myenergy < 60000:
        return DIM[3]/2 - myposition
    return random.randint(-1, 1)
```

2.3 程序限制

我们的算法仍然存在很多问题。比如我们在总结和分析中发现最终算法判断速度比较时 2000 的取值并不科学，如果能够适当优化（减小该值），应该能提高胜率。这次课程作业对于我们所有组员来说都是一次挑战和有趣的体验，我们小组成员各司其职、发挥优势，一起把最初的想法一点点实现并不断调整。但这次作业的趣味性使得我们更积极地投入在其中，这次作业的经历对于我们所有成员来说都是一次巨大的成长。

3 实验结果

3.1 测试数据与测试过程

测试过程主要分三个部分：

1. 避免 invalid bounce，击败 idiot 和其他简单策略
2. 评估不同的一步最优策略选择最优
3. 评估跑位和回球策略选择最优

第一部分，初步的算法在测试中对于 idiot 并没有很高的胜率，且败因基本都是贴边飞行带来的 invalid bounce。我们的解决方法是判断当己方击球点在桌角时，采用击打对角线的策略。这样优化的结果是大大降低了 invalid bounce 的概率，因此胜率也大大提高。

第二部分，我们算法的基本思路是一步最优，通过分析几种不同的一步最优策略互相之间对峙的数据，测试几种策略的胜率。最终选择的策略是假想对手采用最大保全和最大杀伤策略的概率各位 50%，推算对手体力消耗与己方体力消耗的差值，从中选择最大的那个策略

第三部分，评估跑位和回球策略选择最优。具体的测试过程是更换开始跑位的不同体力值，考察胜率的变化，最终选定体力值小于 60000 开始跑位。回球策略也是更换回球不同的落球点胜率变化，最终选定打角策略。

3.2 具体实验过程与结果分析

3.2.1 具体实验过程

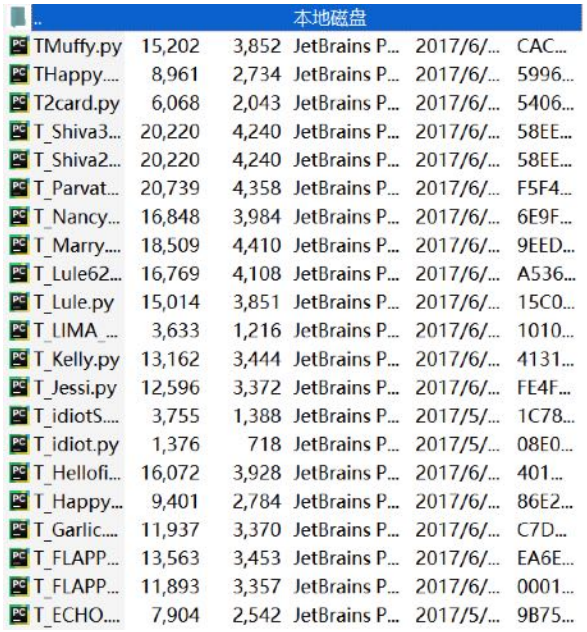
实验过程主要分为三步：

第一步，实现功能，保证在不犯规的情况下把球打过去 战胜 idiot 和其他基本策略。

第二步，选择一步最优的决策策略并具体选定。

第三步，机器学习纳入到算法中。

小组的代码开发是在一个首先提出的比较简单的算法思路基础上不断调试和改进，并添加新想法的过程。下图是小组所有开发出的代码。小组的第一个代码是 T_idiotS，基本是在 idiot 基础上稍加改进的一个“傻瓜”算法，仅添加了基于打角策略的击球路径选择。虽然这一算法并没有很强大的攻击力，但其中 chooseroute 函数基本成为我们以后所有算法编写路径选择函数的基础，对每次击球碰壁次数的考察也是我们在所有算法中都保留的代码部分。



```
def chooseroute(touchpoint, target, initialv):#touchpoint:击球点，#target:球击中对侧的目标位置，initialv：击球前球在y方向上速度

    MAX_touch = 2#MAX_touch是球行进过程中允许的最多触边次数

    touchpointupper = DIM[3] + DIM[3] - touchpoint #touchpointupper1:击上边1次的情况
    touchpointunder = -touchpoint #touchpointunder1:击下边1次的情况

    targetupper = DIM[3] + DIM[3] - target
    targetunder = -target

    velocityup1 = (targetupper -touchpoint)/crosstime
    velocitydown1 = (targetunder - touchpoint)/crosstime
    velocityup2 = -(targetunder -touchpointupper)/crosstime
    velocitydown2 = -(targetupper - touchpointunder)/crosstime
    if touchpoint == 0 or touchpoint == DIM[3] : #从球桌顶点击球，相当于球已经触边一次后再飞出，球在行进过程中可触边的次数-1
        MAX_touch = MAX_touch - 1

    if target == 0 or target == DIM[3]:#球的目标是对方的球桌顶点，相当于球在被对方击回前还有一次触边，球在行进过程中可触边的次数-1
        MAX_touch = MAX_touch - 1

    if MAX_touch == 0:
        return ((target - touchpoint)/crosstime - initialv) #球直接在球桌上做对角线飞行
    elif MAX_touch == 1:
        mincost = min(abs(velocitydown1 - initialv),abs(velocityup1 - initialv))
        if abs(velocitydown1 - initialv) == mincost:
            return velocitydown1 - initialv
        elif abs(velocityup1 - initialv) == mincost:
            return velocityup1 - initialv
    elif MAX_touch == 2:
        mincost = min(abs(velocitydown1 - initialv),abs(velocityup1 - initialv),abs(velocitydown2 - initialv),abs(velocityup2 - i
        if abs(velocitydown1 - initialv) == mincost:
```

小组第二版代码主要添加了使用道具的代码部分。在这一版中获取道具的算法思路是只获取非隐身功能的道具，在其他所有在桌面上的道具中选择加速度最小能获得的道具作为目标获取的道具，另一个函数用于确定取道具路径是否合法。这里的路径选择使用了求切点方法规划取道具路径，这一方法也在以后的代码中保留。同时这一版本代码中还包括检查碰壁合法性，通过来球路径倒推对手击球位置等，这些部分也在以后的算法中得以保留。

```
if getwhichcard(tb)!= False:#如果要去取道具，所需的最小加速
    MIN_Cost = getwhichcard(tb)




if len(CardBox())==0:#道具箱为空
    usecard = None
else:
    usecard = CardBox().pop()

return RacketAction(tb.tick, tb.ball['position'].y - tb.side['position'].y, MIN_Cost, 0, cardside(usecard),usecard)

#本函数用于倒推对手击球的位置
def findlocation(position,vy):
    sy = vy*crosstime + position
    location = 0 #初始化location,location为对方击球时位置的y坐标
    if sy > 0:
        if sy > DIM[3] and sy < 2*DIM[3]:#当球触碰上边一次时
            location = DIM[3] - abs(sy - DIM[3])
        elif sy >= 2*DIM[3]:
            location = -(DIM[3] - abs(sy - DIM[3])) #当球先触碰下边再触碰上边时
    else:
        if abs(sy) < DIM[3]: #当球触碰下边一次时
            location = -(sy)
        elif abs(sy) >= DIM[3] and abs(sy) <= 2*DIM[3]: #当球先触碰上边再触碰下边时
            location = -sy - DIM[3]
    return location
```

```
#本函数用于倒推对手击球的位置
def findlocation(position,vy):
    sy = vy*crosstime + position
    location = 0 #初始化location,location为对方击球时位置的y坐标
    if sy > 0:
        if sy > DIM[3] and sy < 2*DIM[3]:#当球触碰上边一次时
            location = DIM[3] - abs(sy - DIM[3])
        elif sy >= 2*DIM[3]:
            location = -(DIM[3] - abs(sy - DIM[3])) #当球先触碰下边再触碰上边时
    else:
        if abs(sy) < DIM[3]: #当球触碰下边一次时
            location = -(sy)
        elif abs(sy) >= DIM[3] and abs(sy) <= 2*DIM[3]: #当球先触碰上边再触碰下边时
            location = -sy - DIM[3]
    return location

def getwhichcard(tb:TableData):#如果没有隐身以外的道具，就不去get道具;有其他道具，返回加速度最小能获得的
    acc_list1 = list()#获得每个道具需要的最小加速,对隐身返回FALSE
    for card in tb.cards['cards']:
        if not card.__eq__('DS'):
            acc_list1.append(togetcard(tb,card))
        else:
            acc_list1.append(False)
    acc_list2=list()
    for acc in acc_list1:
        if acc != False:
            acc_list2.append(acc)
    if len(acc_list2)== 0:#没有符合条件的道具
        return False
    else:
        return min(acc_list2)

def togetcard(tb:TableData,card:Card):#返回获得一个道具的最小加速度，如果无法在合法条件内获得道具，返回FALSE

    t1 = (cardpoint(tb,card).p1[1]- tb.ball['position'].y)/(cardpoint(tb,card).p1[0]- tb.ball['position'].x)#tan
    vy1 = t1*BALL_V[0]
    acc1 = vy1 - (-tb.ball['velocity'].y)
    t2 = (cardpoint(tb,card).p2[1] - tb.ball['position'].y) / (cardpoint(tb,card).p2[0] - tb.ball['position'].x)
    vy2 = t2 * BALL_V[0]
    acc2 = vy2 - tb.ball['velocity'].y
    if acc1*acc2<0:
        acc = 0
    else:
        acc = min(abs(acc1),abs(acc2))
    if isinvaild(tb,acc+tb.ball['velocity'].y):
        return acc
    else:
        return False
```

```
def isinvaild(tb:TableData,vy):#已知y方向速度，判断是否合法碰壁
    Y = vy * crosstime + tb.ball['position'].y
    if Y % DIM[3] != 0:
        count = Y //DIM[3]
        bounce_count=abs(count)
    else:
        count = (Y // DIM[3]) if (Y > 0) else (1 - Y // DIM[3])
        bounce_count = count
    if bounce_count in (1,2):
        return True
    else:
        return False

def cardpoint(tb:TableData,card:Card):#求切点
    class point:
        def __init__(self, p1, p2):
            self.p1 = p1
            self.p2 = p2
    #落点到圆心距离
    d = math.sqrt((card.pos.x - tb.ball['position'].x)*(card.pos.x - tb.ball['position'].x)+(card.pos.y - tb.ball['position'].
    #落点到切点的距离
    l = math.sqrt(d*d-r*r)
    #点到圆心的单位向量
    x0 = (card.pos.x - tb.ball['position'].x)/d
    y0 = (card.pos.y - tb.ball['position'].y) / d
    #切线与点心连线的夹角
    f=math.asin(r/d)
    #正反两个方向旋转单位向量
    x1 = x0 * math.cos(f) - y0 * math.sin(f)
    y1 = x0 * math.sin(f) + y0 * math.cos(f)
    x2 = x0 * math.cos(f) - y0 * math.sin(f)
```

小组真正意义上第一版代码是 T_ECHO，基本是 idiotS 和 1card 的结合。新增内容为讨论击球点是否为顶点，避免贴边飞行的情况出现。在这里添加这部分的原因是我们在测试对战 idiot 中发现己方由于贴边飞行而输球的情况非常多。

```
hitplace = tb.ball['position'].y #球的落点
enermy_location = findlocation(hitplace, tb.ball['velocity'].y)
if hitplace == 0 or hitplace == DIM[3]:  # 这里讨论击球点是否为顶点，是为了避免贴边飞行的情况出现
    if hitplace == 0:
        MY_COST = chooseroute(hitplace, DIM[3] / 2, tb.ball['velocity'].y)
        return RacketAction(tb.tick, tb.ball['position'].y - tb.side['position'].y, MY_COST, 0, None, None)
    elif hitplace == DIM[3]:
        MY_COST = chooseroute(hitplace, DIM[3] / 2, tb.ball['velocity'].y)
        return RacketAction(tb.tick, tb.ball['position'].y - tb.side['position'].y, MY_COST, 0, None, None)
```

后面几版代码基本在分析双方击球数据和进行策略选择上取得了进步。

T_FLAPPY 主要添加了字典对双方击球数据的记录，并计算了双方每一回合的体力消耗，规定了体力小于 6000 开始跑位（每次击球后返回终点的策略）。添加了 find_opp_addspeed 来根据本回合己方接球位置、接球速度和上一回合对手接球速度来推测对手给球的加速度。

```
ds = {"my_hitpoint": my_hitpoint, "op_speed": op_speed, "op_hitpoint": op_hitpoint, "my_addspeed": my_addspeed,
      "op_getspeed": op_getspeed}
```

```
def costcounting(mymove,enermymove,mycost,enermycost):
    MY_Cost = (mymove/FACTOR_DISTANCE)**2 + (mycost/FACTOR_SPEED)**2
    ENERMY_Cost = (enermymove / FACTOR_DISTANCE) ** 2 + (enermycost / FACTOR_SPEED) ** 2
    return ENERMY_Cost - MY_Cost
def running(myenergy,myposition):
    if myenergy<60000:
        if myposition != DIM[3]:
            return DIM[3]/2 - myposition
    return random.randint(-1,1)
```

```
def find_opp_addspeed(my_getspeed, opp_last_getspeed, my_hitpoint): #find_opp_addspeed是根据本回合我方接球位置，接球速度，和上一回合对手接球的速度
    the_speed = my_getspeed * counttouchpoint(- my_getspeed, my_hitpoint)
    return the_speed - opp_last_getspeed
```

Garlic 算法是在 FLAPPY 的基础上调整了几个参数，并没有明显的算法改进。

FLAPPY2 算法是在 FLAPPY 的基础上添加获取对方击球位置和对方接球时球速的信息。使得记录双方击球信息更加完整。

```
def getlocation(position, vy):
    sy = round(vy*crosstime + position)
    location = 0 #初始化location, location为对方击球时位置的y坐标
    if sy > 0:
        if sy >= DIM[3] and sy < 2*DIM[3]:#当球触碰上边一次时
            location = DIM[3] - abs(sy - DIM[3])
        elif sy >= 2*DIM[3]:
            location = sy - 2*DIM[3] #当球先触碰下边再触碰上边时
    else:
        if abs(sy) < DIM[3]: #当球触碰下边一次时
            location = -(sy)
        elif abs(sy) >= DIM[3] and abs(sy) <= 2*DIM[3]: #当球先触碰上边再触碰下边时
            location = sy +2* DIM[3]
    return location

def get_opp_getspeed(my_add_velocity, orgin_velocity, my_hitpoint):
    the_velocity = my_add_velocity + orgin_velocity
    the_velocity = counttouchpoint(the_velocity, my_hitpoint)*the_velocity
    return the_velocity
```

为了验证该算法的合理性和测试实力，我们还专门编写了以打角为主要策略的算法 HAPPY 作为“陪练”算法。但在测试中发现 HAPPY 对 FLAPPY 有非常高的胜率，这一结果也对我们未来的算法开发提供的借鉴。

Hellofine 算法是在吸取 FLAPPY 与 HAPPY 对战 FLAPPY 败因基本是体力消耗过多的经验开发的。主要改进是获取道具的优先级。新算法优先获取补血/掉血包，然后是变压器和瞬移，然后旋转球，然后隐身术。这种优先级的变化提高了算法对战 HAPPY 的胜率。

```
lef getwhichcard(tb:TableData):#优先获取补血/掉血包，然后变压器和瞬移，然后旋转球，然后隐身术
    list1=list()
    list2=list()
    list3=list()
    list4=list()#分别用于存四种优先级的道具的加速度
    if len(tb.cards['cards']) == 0:#场上没有道具
        return False
    else:
        for card in tb.cards['cards']:
            if togetcard(tb, card) != False:
                if card.__eq__('DL')or card.__eq__('IL'):
                    list1.append(togetcard(tb,card))
                elif card.__eq__('TP')or card.__eq__('AM'):
                    list2.append(togetcard(tb,card))
                elif card.__eq__('SP'):
                    list3.append(togetcard(tb,card))
                else:
                    list4.append(togetcard(tb, card))
        if len(list1)!=0:
            return min(list1)
        elif len(list2)!=0:
            return min(list2)
        elif len(list3)!=0:
            return min(list3)
        elif len(list4)!=0:
            return min(list4)
        else:
            return False
```

Jessi 算法在之前一类算法基础上有一个较大的改变。在遍历路径策略中，上一代算法条件语句只区分了球的目标是对方的球桌顶点的情况，Jessi 区分了己方从球桌顶点击球和击球目标是对方球桌顶点的情况进行策略遍历。

```
if touchpoint <= 2000 or touchpoint >= DIM[3] - 2000 : #从球桌顶点击球，相当于球已经触边一次后再飞出，球在行进过程中可触边的次数-1
    MAX_touch = MAX_touch - 1

if target <= 2000 or target >= DIM[3] - 2000:#球的目标是对方的球桌顶点，相当于球在被对方击回前还有一次触边，球在行进过程中可触边的次数-1
    MAX_touch = MAX_touch - 1

if MAX_touch == 0:
    return ((target - touchpoint)/crosstime - initialv) #球直接在球桌上做对角线飞行
elif MAX_touch == 1:
    mincost = min(abs(velocitydown1 - initialv),abs(velocityup1 - initialv))
    if abs(velocitydown1 - initialv) == mincost:
        return velocitydown1 - initialv
    elif abs(velocityup1 - initialv) == mincost:
        return velocityup1 - initialv
elif MAX_touch == 2:
    mincost = min(abs(velocitydown1 - initialv),abs(velocityup1 - initialv),abs(velocitydown2 - initialv),abs(velocityup2 - initialv))
    if abs(velocitydown1 - initialv) == mincost:
        return velocitydown1 - initialv
    elif abs(velocityup1 - initialv) == mincost:
        return velocityup1 - initialv
    elif abs(velocitydown2 - initialv) == mincost:
        return velocitydown2 - initialv
    elif abs(velocityup2 - initialv) == mincost:
        return velocityup2 - initialv
```



后面几版代码（Kelly、Lule、Lule62）都是在 Jessi 算法基础上做了一些细节和参数的调整。

下一版 Marry 算法的主要改进在于添加了 enemy_chooseroute 函数用于遍历所有对手可能采用的策略，并假定对方采用最大保全策略。这一内容的增加使得己方选择击球策略更加有主动性，提高了胜率。Find_out_route 函数选择对方体力消耗与己方体力消耗差值最大的击球策略应对本回合。

```
def enermy_chooseroute(touchpoint,target,initialv):#touchpoint:击球点，#target:球击中对侧的目标位置，initialv：击球前球在y方向上速度

    MAX_touch = 2#MAX_touch是球行进过程中允许的最多触边次数

    touchpointupper = DIM[3] + DIM[3] - touchpoint #touchpointupper1:击上边1次的情况
    touchpointunder = -touchpoint #touchpointunder1:击下边1次的情况

    targetupper = DIM[3] + DIM[3] - target
    targetunder = -target

    velocityup1 = (targetupper -touchpoint)/crosstime #这里可以画一个草图来研究一下
    velocitydown1 = (targetunder - touchpoint)/crosstime
    velocityup2 = -(targetunder -touchpointupper)/crosstime
    velocitydown2 = -(targetupper - touchpointunder)/crosstime
    maxcost = max(abs(velocitydown1 - initialv),abs(velocityup1 - initialv),abs(velocitydown2 - initialv),abs(velocityup2 - initialv))
    if abs(velocitydown1 - initialv) == maxcost:
        return velocitydown1 - initialv
    elif abs(velocityup1 - initialv) == maxcost:
        return velocityup1 - initialv
    elif abs(velocitydown2 - initialv) == maxcost:
        return velocitydown2 - initialv
    elif abs(velocityup2 - initialv) == maxcost:
        return velocityup2 - initialv
```

```
def find_out_route(touchpoint, velocity, move, tb, ds):
    enermy_hitpoint = getlocation(touchpoint, velocity + tb.ball['velocity'].y)
    enermy_getspeed = get_opp_getspeed(velocity, tb.ball['velocity'].y, touchpoint)

    return costcounting(move, velocity, enermycost(enermy_hitpoint, enermy_getspeed, ds, tb))
```

Nancy 算法主要在取道具方面进行了优化。首先增加了 Catchcard_Cost 变量记录取道具的体力消耗，更改了取道具有限级，只获取变压器和旋转球两种道具，尽量减少由于取道具带来的过多体力消耗。

```
Catchcard_Cost = -10
if getwhichcard(tb)!= False:#如果要去取道具，所需的最小加速
    Catchcard_Cost = getwhichcard(tb)
if Catchcard_Cost != -10:
    ds["my_addspeed"].append(Catchcard_Cost)
    velocity = get_opp_getspeed(Catchcard_Cost, tb.ball['velocity'].y, tb.ball['position'].y)
    ds["op_getspeed"].append(velocity)
    ds["op_hitpoint"].append(getlocation(hitplace,Catchcard_Cost))
    return RacketAction(tb.tick, move,Catchcard_Cost,
                        running(tb.side['life'], tb.ball['position'].y),cardside(usecard(tb)),usecard(tb))
else:
    ds["my_addspeed"].append(MY_COST)
    velocity = get_opp_getspeed(MY_COST, tb.ball['velocity'].y, tb.ball['position'].y)
    ds["op_getspeed"].append(velocity)
    ds["op_hitpoint"].append(getlocation(hitplace, MY_COST))
    Y = (MY_COST+tb.ball['velocity'].y)*1800 + tb.ball['position'].y
    count = (Y // DIM[3]) if (Y > 0) else (1 - Y // DIM[3])
   # print(Y,isinvaild(tb,MY_COST + tb.ball['velocity'].y),count)
    return RacketAction(tb.tick, move, MY_COST,
                        running(tb.side['life'], tb.ball['position'].y), cardside(usecard(tb)),usecard(tb))
```

```
def getwhichcard(tb:TableData):#只获取变压器和旋转球
    list1=list()
#分别用于存四种优先级的道具的加速度
    if len(tb.cards['cards']) == 0:#场上没有道具
        return False
    else:
        for card in tb.cards['cards']:
            if togetcard(tb, card) != False:
                if card.__eq__('AM')or card.__eq__('SP'):
                    list1.append(togetcard(tb,card))
        if len(list1)!=0:
            return min(list1)
        else:
            return False
```

Shiva 算法主要改进在于，在确定道具选用原则的基础上，将获取道具加入 chooseroute 函数中作为策略集进行遍历考察是否选用。这样更加理性的考虑获取道具的情况，使得最终选择的策略更适合当前回合情景。Shiva3 算法是在 Shiva2 算法基础上的小范围修改。

```
def chooseroute(touchpoint,target,initialv,move,tb,ds):#touchpoint:击球点，#target:球击中对侧的目标位置，initialv：击球前球在y方向上速度

    MAX_touch = 2#MAX_touch是球行进过程中允许的最多触边次数

    touchpointupper = DIM[3] + DIM[3] - touchpoint #touchpointupper1:击上边1次的情况
    touchpointunder = -touchpoint #touchpointunder1:击下边1次的情况

    targetupper = DIM[3] + DIM[3] - target
    targetunder = -target

    velocity = (target - touchpoint)/crosstime
    velocityup1 = (targetupper -touchpoint)/crosstime #这里可以画一个草图来研究一下
    velocitydown1 = (targetunder - touchpoint)/crosstime
    velocityup2 = -(targetunder -touchpointupper)/crosstime
    velocitydown2 = -(targetupper - touchpointunder)/crosstime
    if target == 0 or target == DIM[3]:#球的目标是对方的球桌顶点，相当于球在被对方击回前还有一次触边，球在行进过程中可触边的次数-1
        MAX_touch = MAX_touch - 1

    if len(tb.side['cards']) != 0:
        for j in tb.side['cards']:
            if j.__eq__('AM'):
                if MAX_touch == 1:
                    opp_cost1 = judge_opp_addspeed(getlocation(touchpoint,velocity + tb.ball['velocity'].y),
                                                   get_opp_getspeed(velocity,tb.ball['velocity'].y,touchpoint),ds)
                    opp_cost2 = judge_opp_addspeed(getlocation(touchpoint,velocityup1 + tb.ball['velocity'].y),
                                                   get_opp_getspeed(velocityup1,tb.ball['velocity'].y,touchpoint),ds)
                    opp_cost3 = judge_opp_addspeed(getlocation(touchpoint, velocitydown1 + tb.ball['velocity'].y),
                                                   get_opp_getspeed(velocitydown1, tb.ball['velocity'].y, touchpoint), ds)
                    max_opp_cost = max(abs(opp_cost1),abs(opp_cost2),abs(opp_cost3))

                    if max_opp_cost > 700:
```

```
        elif MAX_touch == 2:
            opp_cost2 = judge_opp_addspeed(getlocation(touchpoint, velocityup1 + tb.ball['velocity'].y),
                                           get_opp_getspeed(velocityup1, tb.ball['velocity'].y, touchpoint), ds)
            opp_cost3 = judge_opp_addspeed(getlocation(touchpoint, velocitydown1 + tb.ball['velocity'].y),
                                           get_opp_getspeed(velocitydown1, tb.ball['velocity'].y, touchpoint), ds)
            opp_cost4 = judge_opp_addspeed(getlocation(touchpoint, velocityup2 + tb.ball['velocity'].y),
                                           get_opp_getspeed(velocityup2, tb.ball['velocity'].y, touchpoint), ds)
            opp_cost5 = judge_opp_addspeed(getlocation(touchpoint, velocitydown2 + tb.ball['velocity'].y),
                                           get_opp_getspeed(velocitydown2, tb.ball['velocity'].y, touchpoint), ds)
            max_opp_cost = max(abs(opp_cost2), abs(opp_cost3),abs(opp_cost4),abs(opp_cost5))

            if max_opp_cost > 700:
                if max_opp_cost == abs(opp_cost2):
                    return opp_cost2 - initialv
                elif max_opp_cost == abs(opp_cost3):
                    return opp_cost3 - initialv
                elif max_opp_cost == abs(opp_cost4):
                    return opp_cost4 - initialv
                elif max_opp_cost == abs(opp_cost5):
                    return opp_cost5 - initialv
else:
    return enermy_chooseroute(touchpoint,target,initialv)

if MAX_touch == 1:
    gain1 = find_out_route(move, touchpoint, velocity - initialv, tb, ds)
    gain2 = find_out_route(move, touchpoint, velocityup1 - initialv, tb, ds)
    gain3 = find_out_route(move,touchpoint,velocitydown1 - initialv,tb,ds)

    gain = max(gain1,gain2,gain3)
    if gain == gain1 :
        return velocity - initialv
```

在以上所有算法的基础上，最终算法为 Parvati。这一算法与之前所有算法相比最大的变化在于 judge_opp_addspeed 函数，这一函数判断击球落点如果与记录中某一回合落点距离小于 2000，且到达速度与那一次记录的到达速度差小于 2000，就认为对方会采用上一次相同策略，以及选择应对策略。

```
def judge_opp_addspeed(hitpoint,op_get_speed,ds:dict):
    global op_speed
    global op_hitpoint
    global op_getspeed

    for i in range(len(ds['op_hitpoint'])):
        if abs(hitpoint - (ds['op_hitpoint'][i]))< 2000:
            if abs(op_get_speed - ds['op_getspeed'][i]) < 2000:
                if i != len(ds['op_hitpoint']) - 1:
                    return ds['op_speed'][i+1]

    return -1
```

综上所述，小组的算法开发是一个在已有算法的基础上，根据算法之间相互对战数据不断改进已有思路和提出新想法的过程。在实际开发和实验的过程中，我们发现许多情况与我们自己想象的情景并不相同，也经常发生用于测试和“陪练”的算法在实际对战中多次战胜主要算法的情况。因此我们在开发算法的过程中也经常出现迂回和更换策略（比如多次更

换获取道具的优先级等）。我们在开发初期就一直想采用的学习思想也是在不断优化算法的过程中逐渐添加进代码中的，具体学习方法也有一个不断深化和优化的过程（从最初单纯记录双方击球信息，推测对方击球加速度到遍历对方所有可能策略，最后将回合的落点和速度信息与记录集信息作比较来推测对方策略）。

3.2.2 实验结果分析

在第一场热身赛中，我们的代码有许多不足，在 200 场对战中全负。

但通过一个阶段的调整与修改，我们在策略的选取和应对方案方面有了本质的提升，同时添加了对对方信息的推测与学习的算法，获取和使用道具方面也有了较为完善的判断机制。这些修改使得我们的胜率大大提升。

4 小组分工和实验过程

4.1 分工与合作

4.1.1 小组分工

蒋天骥：主要算法和基本代码实现
黄骁：算法补充和部分代码实现
李婧宇：算法补充和道具代码实现
杨家鑫：部分道具代码实现和实验报告修改补充
王宇萱：代码 debug 和实验报告
周广财：代码 debug
李浩：部分算法补充

4.1.2 小组会议



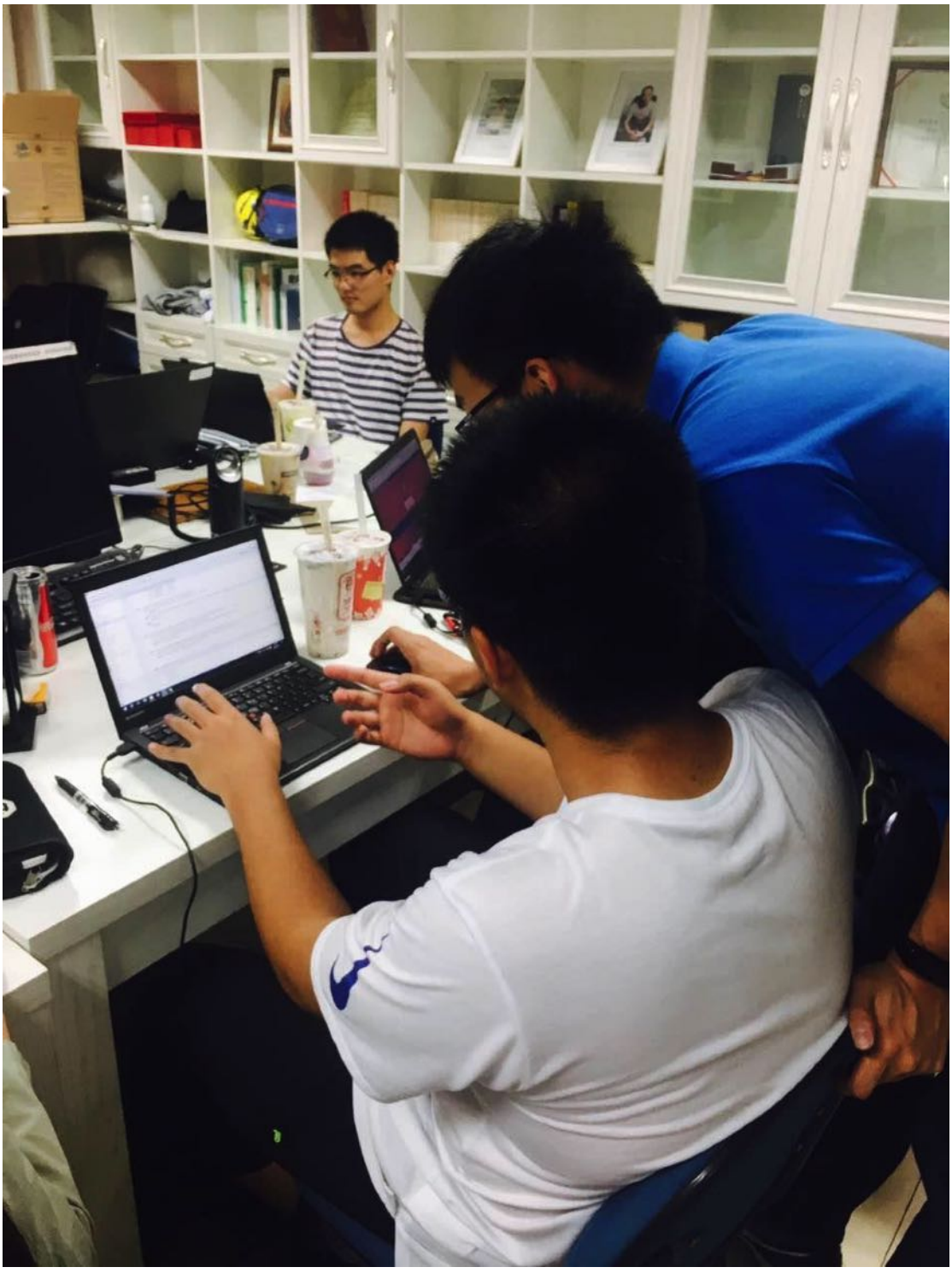
5 月 24 日，小组第一次会议，研究规则和具体要求，讨论算法的基本思路。

5 月 25、26 日，小组第二次、第三次会议，讨论算法思路，编写基本代码。



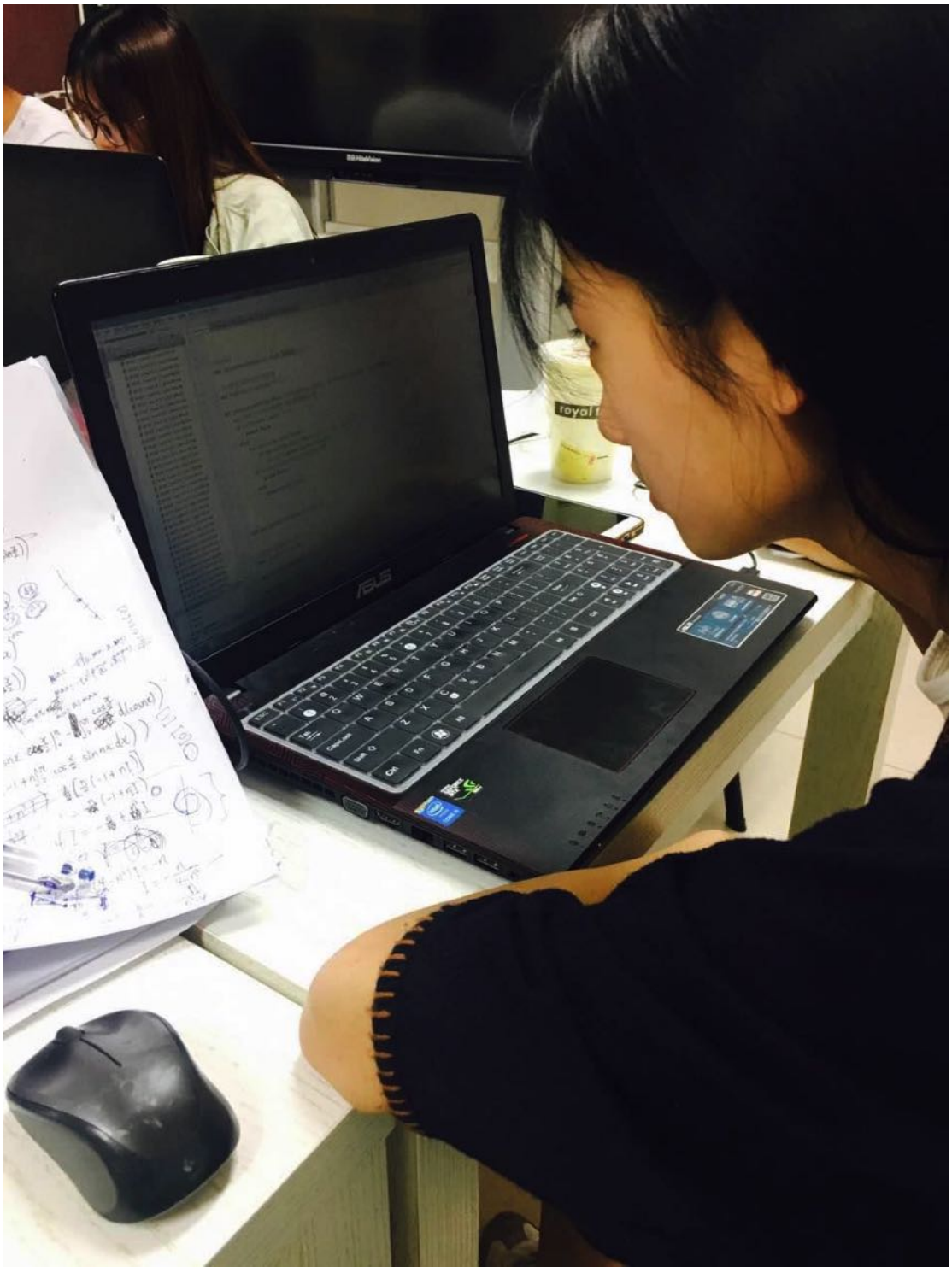


5 月 28、29 日，整合代码，讨论算法改进思路和道具策略。





6 月 2 日，研究热身赛复盘数据，提出改进思路。



6 月 4 日，提交前检查代码。

4.2 经验与教训

在本次算法设计中，我们小组主要采用的是线性结构。通过简单易懂的循环结构、赋值语句等构造起算法的整个框架。操作与理解非常简单，而且在功能实现方面也十分出色。

但由于对非线性结构如图、树等不熟悉，又由于时间精力有限，因此未对不熟悉的领域开展深入探索。但如果我们能够充分学习并利用到这些非线性结构，我们的算法肯定会更

加精炼、效率更高。

4.3 建议与设想

组队方面，老师助教可以根据平时作业情况划分小组，达到实力、院系、年级等均匀分配的目的。

由于基础代码的改进确实有同学们的帮助后会更完善，因此可以保持同学参与修改完善的过程，但建议可以稍微提早一点提出期末大作业设计的时间，方便同学们建立初步的想法，也会后续实现提供更多时间。

5 致谢

在各学科事务繁多、社团学生会等多方面繁忙的期末季，十分感谢每一位小组成员能够用心的、团结的完成本次“乒乓球大赛”。

尤其需要感谢组长蒋天骥同学，用心的组织每一次组会，带领大家一起分析当前代码情况并合理分工。此外，积极设计与思考算法的实现与改进方法，不辞辛苦地一遍又一遍的修改提升完善小组的代码。

同时也十分感谢黄骁与李婧宇同学，在重要代码段：推测对方信息、选择击球策略、道具获取与使用实现过程中的贡献。

我们还要感谢信息管理系的林殷年同学，他在我们的算法开发中提供了许多宝贵的想法和意见，为我们的开发提供了许多启发。

6 参考文献：

1. 郭亚宁,冯莎莎. 机器学习理论研究[J]. 中国科技信息,2010,(14):208-209+214.

2. 王珏,石纯一. 机器学习研究[J]. 广西师范大学学报(自然科学版),2003,(02):1-15.



ROMEO

组长：杨子珍

组员：李佳益，王旭斌，冯禄，江世豪，王今朝，刘小辉

小组赛战绩：E5







ROMEO



Romeo 组 数据结构与算法课程实习作业报告

杨子珍* 王旭斌 刘小辉 李佳益 江世豪 王今朝 冯 禄

（北京大学 地球与空间科学学院）

摘要 我组代码分为三个阶段，热身赛阶段（miaou）、友谊赛阶段（haha、hiahia）、最终阶段（romeo）；主要是去根据游戏规则去寻找策略以及针对敌方策略的应对方法，可以说是一个防守型战术。在多次热身赛和友谊赛后又不断舍弃和添加一些处理方法，最终形成一个以打角为攻击手段，对道具拾取和使用有优先级判断的击球方式，跑位方式主要根据该局中对方的攻击策略进行选择的算法。

关键词 list；dict；stack；priority queue

1.算法思路

1.1 总体思路

通过简单地拾取算法（在不会 ib 的情况下完全随机给一个加速度）与 idiot 算法的比赛结果，本组认为该游戏的设计思路主要是增大加速带来的增益，并提高跑位的准确性。

加速的增益包括避免简单反弹造成的撞击次数违规，牵引对方大位移迎球增大其损耗，拾取有用道具并正确使用从而拉大差距，尽管各有策略，但这一部分在我们的实验中与敌对算法表现出的差距不大（对 miaou 修正后）。对于跑位准确性的提高来讲，虽然各家普遍使用“打角”策略，但具体打的是上角还是下角，有规律可循还是存在一定随机性，为跑位带来了难度。

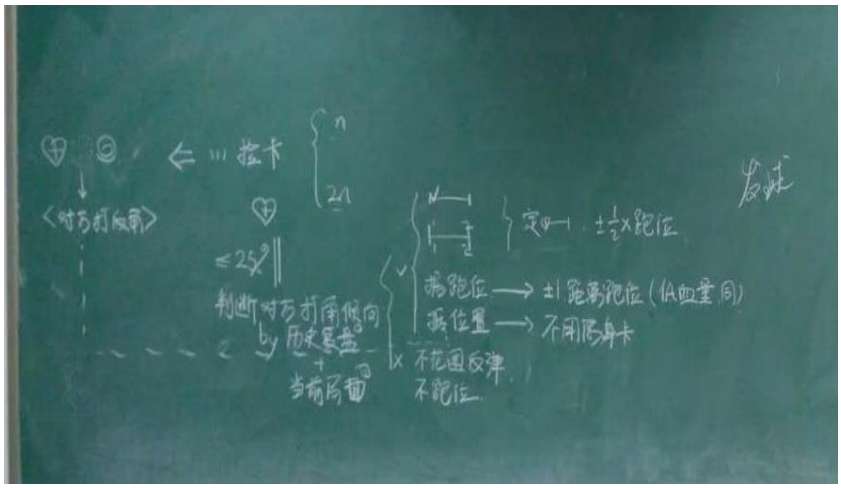


图 1-热身赛代码 miaou 设计思路

在参加热身赛时的 miaou 中我们基本实现了我们的全局战略（见图 1），甚至有一部分在最终提交的 remeo 中被隐蔽掉而没有展现。之后在修正了边界迎球改变加速的体力消耗，对“真边界”和“目视近似边界”相关参数进行调整后，通过友谊赛的锤炼，我们最终做了一个艰难的决定：摒弃我们依托于当前局及过往若干局历史记录所进行的对方打角预测，而改用仅依据当前局的最近前两个回合我方赢球位置，预测对方动向。最终我们得到了如今的 remeo。

在击球动作中，我们首先考虑道具的拾取，并对道具进行了优先级的划分，即“血包类>瞬移术>变压器>其他”，并各自设置了应有的盈亏对比以判定是否捡起。这里特别提出，对于血包的拾取判定，我组认为应以二倍血包增益与拾取所需消耗进行对比，假设血包量以 A 表示，我方拾取消耗为 B，对方拾取消耗为 C，只要我方保证 A-B>-(A-C)即可，即 2A>B-C>B。在对道具采取不拾取操作后，考虑进行低消耗打角策略。在道具的使用选择中我们采用了与道具拾取相同的优先级，并注意道具使用的合理性及收益大化。跑位策略上，我们通过对己方之前连续两次迎球位置的读取，判断对方这一阶段具有交替打角或专一打角的特征，从而进行 1/10 或 1/4 或 1/2 跑位。至此，完成函数的输出参数，并记录存储。

1.2 算法流程图(见文后附件 1)

1.3 算法运行时间复杂度分析

这里列出 romeo 最终代码的大 O 分析：

Serve：0——O（1）

Choosethepath1：O(1)

Choosethepath2：5——O（1）

Choosethepath3:5——O（1）

Speedcheck：2——O（1）

Tryforcards:9n+21——O(n)

Kicktocorner:13——O(1)

Usecard:5n+21——O(n)

Positioning:15——O(1)

Play:9+9n+21+13+3+2n+1+2n+5n+21+1+15+3=16n+87——O(n)

2.程序代码说明

2.1 数据结构说明

使用的数据结构类型：

主要为线性数据结构，其中包括：

(1)列表：Python 列表是任意对象的有序集合，通过索引访问指定元素。本代码中列表主要用途包括记录我方所拥有的可选择路径，并寻找最佳方案；记录我们所拥有的道具等

(2)字典：Python 字典是另一种可变容器模型，且可存储任意类型对象，保存大量键值对。本代码中字典主要用途包括记录历史数据，用来判断对方是否有打角倾向等

(3)栈：事实上，在判断打角算法时用到的对临近几句态势分析，对不久前的历史数据的 pop 操作隐式地使用了栈

(4)优先队列：对于所拥有的道具的使用，以及如何吃道具的方法，我们赋予了其优先级构成了优先队列

(5)是否有自定义的类：无

2.2 函数说明

这里选择我组在对战中表现较好并有较大不同的算法分别进行说明。

正式比赛代码（romeo）

具体代码说明：

```
20  def serve(op_side, ds):
21      return 400000, -BALL_V[1] // 4
22
```

[20] serve 发球函数，总是作为 west 才发球

```
24  def ChooseThePath1(current, target, velocity, limitations):
25      time = abs((current.x - target.x) / velocity.x)
26      distance_y = [2 * DIM[3] + target.y - current.y,
27                    2 * DIM[3] - target.y - current.y,
28                    target.y - current.y,
29                    -(0 * DIM[3] + target.y + current.y),
30                    -(2 * DIM[3] - target.y + current.y)]
31      pre_y = velocity.y * time
32      delta_vy = [round((dist - pre_y) / time) for dist in distance_y]
33      abs_delta_vy = [abs(v) for v in delta_vy]
34      min_delta_vy = None
35      while delta_vy != []:
36          index = abs_delta_vy.index(min(abs_delta_vy))
37          min_delta_vy = delta_vy[index]
38          if (limitations[0] >= min_delta_vy >= limitations[1]) or (limitations[2] >= min_delta_vy >= limitations[3]):
39              break
40          else:
41              min_delta_vy = None
42              delta_vy.pop(index)
43              abs_delta_vy.pop(index)
44      return min_delta_vy
45
46
47  NearCorner = 2000
```

[42] ChooseThePath1 （后面并没有用到）在通常情况下，由多路径中选择一条路径的的方法

```
48  def ChooseThePath2(current, target, velocity):
49      time = 1800
50      if target.y == DIM[2]:
51          distance_y = [2 * DIM[3] - current.y + NearCorner,
52                        2 * DIM[3] - current.y,
53                        2 * DIM[3] - current.y - NearCorner,
54                        -(0 * DIM[3] + current.y + NearCorner),
55                        -(2 * DIM[3] + current.y - NearCorner)]
56      else:
57          distance_y = [3 * DIM[3] - current.y - NearCorner,
58                        1 * DIM[3] - current.y + NearCorner,
59                        -(1 * DIM[3] + current.y + NearCorner),
60                        -(1 * DIM[3] + current.y),
61                        -(1 * DIM[3] + current.y - NearCorner)]
62      pre_y = velocity.y * time
63      delta_vy = [round((dist - pre_y) / time) for dist in distance_y]
64      abs_delta_vy = [abs(v) for v in delta_vy]
65      return delta_vy[abs_delta_vy.index(min(abs_delta_vy))]
```

[47] ChooseThePath2 如果我们的位置在距离两侧不超过 1/4 的地方的打角路径选择：根据现在和目标的位置来选择的路径，来说会有 5 条可行路到达对面某个角落，而如果我们球在墙壁，则只会有三条路，我们选择一条最短的，并改变速度

```
68  def ChooseThePath3(current, velocity):
69      time = 1800
70      distance_y = [2 * DIM[3] - current.y + NearCorner,
71                    2 * DIM[3] - current.y,
72                    2 * DIM[3] - current.y - NearCorner,
73                    -(0 * DIM[3] + current.y + NearCorner),
74                     -(2 * DIM[3] + current.y - NearCorner),
75                   3 * DIM[3] - current.y - NearCorner,
76                    1 * DIM[3] - current.y + NearCorner,
77                    -(1 * DIM[3] + current.y + NearCorner),
78                    -(1 * DIM[3] + current.y),
79                    -(1 * DIM[3] + current.y - NearCorner)]
80      pre_y = velocity.y * time
81      delta_vy = [round((dist - pre_y) / time) for dist in distance_y]
82      abs_delta_vy = [abs(v) for v in delta_vy]
83      return delta_vy[abs_delta_vy.index(min(abs_delta_vy))]
84
```

[68] ChooseThePath3 如果在距离中间 1/4 以内的地方，有 8 条路径供选择，我们选一个最合适的打角算法

```
85  def SpeedCheck(v, active_card):
86      if v == None: return None
87      ret = v
88      if active_card[1] == CARD_SPIN: # If our rotation has been halfed
89          ret /= CARD_SPIN_PARAM
90      return ret
91
92
```

[85] SpeedCheck 检查这个速度是否满足要求

[93] TryForCards 先将道具进行分类，1 是补血包，2 是瞬移术，3 是掉血包，4 是旋转球和隐身术。

如果要吃血包的条件：吃血包补充的体力小于两倍的为了吃血包消耗的体力，因为只要是这样为了吃血包都是在平均程度上有益的：如果是少于一倍，则是相当于没消耗体力值的瞬移，而如果是两倍，则相当于使得对方平均消耗的体力值小于我方消耗的。对于吃掉血包同理

如果要吃瞬移的条件：瞬移消耗的体力小于为了吃瞬移消耗的体力。

在判断完以上两个之后，如果此时自己和对方的总生命值小于球拍的总生命值，就不考虑吃了

如果要吃变压器的条件：吃这个消耗的体力小于对方跑过全场 1/4 长度消耗的体力

如果我们以上什么都没有吃：

如果我方生命值小于对方为了保存体力，就不吃了，如果恰好碰到那就是运气问题了

如果我方有优势，我们也不能放弃这些优势，所以在计算吃完道具之后我方

生命值仍领先对方生命值的 1/8 以上就吃。

```
def TryForCards(TICK, TICK_STEP, SELF, OPNT, BALL, CARD, limit_v):

    # Data initialization
    self_position, self_life, self_cards = SELF['position'], SELF['life'], SELF['cards']
    opnt_position, opnt_life, opnt_cards = OPNT['position'], OPNT['life'], OPNT['cards']
    opnt_active_card, opnt_run_vector = OPNT['active_card'], OPNT['run_vector']
    ball_position, ball_velocity = BALL['position'], BALL['velocity']
    ball_velocity.x = -ball_velocity.x
    card_cards = CARD['cards']

    # Sort the cards
    # 1. Cards for life
    # 2. Cards for flash
    # 3. Cards for double enemy's consumption
    # 4. Cards for others(rotation and disappearation)
    card_life = list()
    card_tp = list()
    card_am = list()
    card_others = list()
    for card in card_cards:
        if card == CARD_INCL or card == CARD_DECL:
            card_life.append(card)
        elif card == CARD_TLPT:
            card_tp.append(card)
        elif card == CARD_AMPL:
            card_am.append(card)
        else:
            card_others.append(card)

    # If there is a card that we want.
    # The spot which is nearest is the one to choose.
    ret_V = None

```

```
    # 1. Card for life
    for card in card_life:
        min_delta_vy = ChooseThePath1(ball_position, card.pos, ball_velocity, limit_v)
        if min_delta_vy != None and (ret_V == None or abs(ret_V) > abs(min_delta_vy)):
            ret_V = min_delta_vy
    ret_V = SpeedCheck(ret_V, opnt_active_card)
    if ret_V != None and (ret_V / FACTOR_SPEED) ** 2 <= CARD_INCL_PARAM:
        return ret_V

    # 2. Cards for flash:
    for card in card_tp:
        min_delta_vy = ChooseThePath1(ball_position, card.pos, ball_velocity, limit_v)
        if min_delta_vy != None and (ret_V == None or abs(ret_V) > abs(min_delta_vy)):
            ret_V = min_delta_vy
    ret_V = SpeedCheck(ret_V, opnt_active_card)
    if ret_V != None and (ret_V / FACTOR_SPEED) ** 2 <= (CARD_TLPT_PARAM / FACTOR_DISTANCE) ** 2:
        return ret_V
    else:
        ret_V = None

    # When it's time for a result, do something more useful.
    if self_life + opnt_life <= RACKET_LIFE:
        return None

    # 3. Cards for double enemy's consumption
    for card in card_am:
        min_delta_vy = ChooseThePath1(ball_position, card.pos, ball_velocity, limit_v)
        if min_delta_vy != None and (ret_V == None or abs(ret_V) > abs(min_delta_vy)):
            ret_V = min_delta_vy
    if ret_V != None: ret_V = SpeedCheck(ret_V, opnt_active_card)
    if ret_V != None and (ret_V / FACTOR_SPEED) ** 2 <= ((DIM[3] // 2) / FACTOR_DISTANCE) ** 2:
        return ret_V
    else:
        ret_V = None

    # 4. If we still get nothing, but we have lots of life to WASTE, we can try to get something.
    if self_life <= opnt_life:
        pass  # We are in a bad condition, so we should save our life.
    else:
        for card in card_others:
            min_delta_vy = ChooseThePath1(ball_position, card.pos, ball_velocity, limit_v)
            if min_delta_vy != None and (ret_V == None or abs(ret_V) > abs(min_delta_vy)):
                ret_V = min_delta_vy
        if ret_V != None: ret_V = SpeedCheck(ret_V, opnt_active_card)
        if ret_V != None and (ret_V / FACTOR_SPEED) ** 2 <= (self_life - opnt_life) / 20:
            return ret_V
        else:
            ret_V = None
    return ret_V
```

```python
def KickToCorner(TICK, TICK_STEP, SELF, OPNT, BALL, CARD, ds):

    # Data initialization
    self_position, self_life, self_cards = SELF['position'], SELF['life'], SELF['cards']
    opnt_position, opnt_life, opnt_cards = OPNT['position'], OPNT['life'], OPNT['cards']
    opnt_active_card, opnt_run_vector = OPNT['active_card'], OPNT['run_vector']
    ball_position, ball_velocity = BALL['position'], BALL['velocity']
    ball_velocity.x = -ball_velocity.x
    card_cards = CARD['cards']

    if 'my_pre_pos' in ds.keys() and ds['my_pre_pos'] <= DIM[3] // 4 * 1 and ball_position.y <= DIM[3] // 4 * 1:
        ret_V = SpeedCheck(ChooseThePath2(ball_position, Position(opnt_position.x, DIM[2]), ball_velocity), opnt_active_
    elif 'my_pre_pos' in ds.keys() and ds['my_pre_pos'] >= DIM[3] // 4 * 3 and ball_position.y >= DIM[3] // 4 * 3:
        ret_V = SpeedCheck(ChooseThePath2(ball_position, Position(opnt_position.x, DIM[3]), ball_velocity), opnt_active_
    else:
        ret_V = SpeedCheck(ChooseThePath3(ball_position, ball_velocity), opnt_active_card)
    if not ret_V: ret_V = 0
    return ret_V
```

[177] KickToCorner 打角算法：如果我们的位置在距离两侧不超过 1/4 的地方，有 3、5 种选择能走的最短的路径打角，如果在距离中间 1/4 以内的地方，有 8 种打角方法。

```python
def UseCard(TICK, TICK_STEP, SELF, OPNT, BALL, CARD, vx, vy, enemy_y):

    # Data initialization
    self_position, self_life, self_cards = SELF['position'], SELF['life'], SELF['cards']
    opnt_position, opnt_life, opnt_cards = OPNT['position'], OPNT['life'], OPNT['cards']
    opnt_active_card, opnt_run_vector = OPNT['active_card'], OPNT['run_vector']
    ball_position, ball_velocity = BALL['position'], BALL['velocity']
    ball_velocity.x = -ball_velocity.x
    card_cards = CARD['cards']

    # If we have cards for life.
    for card in self_cards:
        if card == CARD_INCL:
            return ('SELF', card)
        elif card == CARD_DECL:
            return ('OPNT', card)
```

[197] UseCard 用卡算法：有补血包掉血包直接用掉

如果球拍要到达球的距离大于卡能提供的距离，那么就用瞬移术

如果对方需要跑的距离大于球桌的 1/2，那么就用变压器

如果对方不加速球不会撞墙 1/2 次，那么就用旋转球

如果我们的卡片满了或者我们的生命值小于对方，而以上的卡片又都没有用，那么就随便用一张。如果对方喜欢打角，那么就不用隐身术

```
    # If we have no card for life, we can find a card for flash instead.
    if abs(ball_position.y - self_position.y) >= CARD_TLPT_PARAM:
        for card in self_cards:
            if card == CARD_TLPT:
                return ('SELF', card)

    # If we have no card for life or flash, we can find a card for double enemy's consumption at least.
    if abs(enemy_y - opnt_position.y) >= DIM[3] // 2:
        for card in self_cards:
            if card == CARD_AMPL:
                return ('OPNT', card)

    # If enemy doesn't give the ball an acceleration,
    # if the ball will hit the wall 0, 3, 4, 5, 6... times,
    # we use the card for rotation.
    my_y = int(enemy_y + vy * TICK_STEP)
    opnt_collision = 0
    while my_y < DIM[2] or my_y > DIM[3]:
        if my_y < DIM[2]:
            my_y = 2 * DIM[2] - my_y
        else:
            my_y = 2 * DIM[3] - my_y
        opnt_collision += 1
    if not opnt_collision in (1, 2):
        for card in self_cards:
            if card == CARD_SPIN:
                return ('OPNT', card)

    # If we use nothing, or we are in a bad condition, use something.
    # Also, if enemy prefer kicking the ball over the corner, don't use card for disappearation.
    if len(self_cards) == MAX_CARDS or self_life <= opnt_life:
        if self_cards:
            card = random.choice(self_cards)
            for i in range(6):
                if card == CARD_DSPR:
                    card = random.choice(self_cards)
            if card in (CARD_DSPR, CARD_INCL, CARD_TLPT):
                return ('SELF', card)
            else:
                return ('OPNT', card)
    return None
```

[257]跑位算法(对方如果是 golf，跑位是球桌长度与自己位置平均值的 1/2 处—根据友谊赛做出的针对性调整，虽然并没有碰上)

如果我上一次和上上一次的位置是在距离两侧 1/4 以内的话，那么就认为对方轮流打角，跑到距离这一次距离这次对方可能打角的 1/10 处

我上一次和上上一次的位置是在距离同一侧的 1/2 以内进行在所记录的那一侧的 1/4 跑位

否则回中跑位

[292] 打球函数的开始

[308] 设置界限，必须碰撞 1-2 次才可以

[318] 尝试所有场上的卡片并决定是否吃掉、怎样吃掉，如果没有我们能吃的那就打角

[347] 用卡片，保存历史数据

```
257 def Positioning(TICK, TICK_STEP, SELF, OPNT, BALL, CARD, ds, nexty):
258
259     # Data initialization
260     self_position, self_life, self_cards = SELF['position'], SELF['life'], SELF['cards']
261     opnt_position, opnt_life, opnt_cards = OPNT['position'], OPNT['life'], OPNT['cards']
262     opnt_active_card, opnt_run_vector = OPNT['active_card'], OPNT['run_vector']
263     ball_position, ball_velocity = BALL['position'], BALL['velocity']
264     ball_velocity.x = -ball_velocity.x
265     card_cards = CARD['cards']
266
267     if EnemyName == 'T_GOLF' and 'my_pre_pos' in ds.keys():
268         if ds['my_pre_pos'] <= DIM[3] // 4 * 1 and ball_position.y <= DIM[3] // 4 * 1:
269             return (DIM[3] - ball_position.y) // 2
270         if ds['my_pre_pos'] >= DIM[3] // 4 * 3 and ball_position.y >= DIM[3] // 4 * 3:
271             return (DIM[2] - ball_position.y) // 2
272         return 0
273
274     if 'my_pre_pre_pos' in ds and 'my_pre_pos' in ds:
275         if ds['my_pre_pre_pos'] <= DIM[3] // 4 * 1 and ds['my_pre_pos'] <= DIM[3] // 4 * 1 and ball_position.y <= I
276             return DIM[3] // 10 * 1 - ball_position.y
277         if ds['my_pre_pre_pos'] >= DIM[3] // 4 * 3 and ds['my_pre_pos'] >= DIM[3] // 4 * 3 and ball_position.y >= I
278             return DIM[3] // 10 * 9 - ball_position.y
279
280     if nexty < DIM[3] // 2 and ball_position.y < DIM[3] // 2:
281         return DIM[3] // 4 * 1 - ball_position.y
282     elif nexty > DIM[3] // 2 and ball_position.y > DIM[3] // 2:
283         return DIM[3] // 4 * 3 - ball_position.y
284     else:
285         return DIM[3] // 2 - ball_position.y
286
287
```

友谊赛阶段（hiahia）

首先是对敌方策略的一个判断，包含对一下几种情况的判定条件：1、打一个角 2、打离我方位置远的角 3、打与我方跑位方向相反的角 4、轮流打角 5、打敌方最低消耗角 6、捡血包

根据历史数据与本盘数据判断敌方策略，选择我方策略,Pprefer=0.3 为判断敌方策略倾向的临界值，大于 0.3 即认为敌方选择该策略。敌方打一个角或交替打角我方用二分之一跑位，根据我方位置不用隐身卡，根据我方跑位我方则跑位 1 个单位。

具体代码说明：

[37-57 行]：历史数据函数

[70-83 行]：发球函数

无对方历史数据，发球坐标为 400000，y 方向速度为初速度的负四分之一

有对方历史数据，对方无打角历史数据，发球坐标为 400000，y 方向速度为初速度的负四分之一

有对方历史数据，对方有打角历史数据，判断对方打角倾向，对方倾向于上角，则使球在回来时在下角附近反弹，纵坐标 200000，初速度二分之一，反之亦然，纵坐标 800000，初速度负二分之一。

[86-157 行]：判断敌方打角策略函数（不包括吃血包）

记录敌方打的位置的次数，判断敌方的可能策略，选择系数最高的一种策略与 Pprefer 比较，高于即认为敌方选择该策略。如果敌方根据我方跑位方向打角，泽敌方上当，球位于中点返回零，否则返回一；若敌方不是根据我方跑位，返回敌方目标位置与球位置差的四分之一。

160-230 行：判断敌方吃血包与针对策略函数，若敌方吃血包，则在对方接球点中选择对方要消耗血量与己方消耗血量差最大的位置。

[234-258 行]：选择我方策略函数

根据敌方策略选择我方策略，每种策略有 5 种具体路径，若球位置在角附近则只有 3 种路径，选择消耗最少路径，同时避免不合理的反弹次数

[261-270 行]：检查我方速度函数

判断我方是否被使用旋转球与变压器，并避免自杀球

[273-369 行]：使用道具函数

优先级为血包-闪现卡-变压器-其余道具。判断桌面上存在道具我方是否想要，如有想要道具则选择离道具位置最近的合理路径。判断是否想要的条件：血包：加速损失血量小于己方获得血量或敌方损失血量的 2 倍；闪现卡：加速损失血量小于闪现节省的血量；变压器：加速消耗血量小于敌方跑四分之一个桌面消耗的血量。如果此时没有符合条件的可得道具，则判断我方血量与敌方血量关系，若少于敌方则放弃吃道具而减少体力消耗；若多于敌方，若捡包消耗血量小于我方血量优势的八分之一，泽捡剩余血包。

[372-454 行]：我方可能策略函数

若敌方生命小于四分之一，直接打到对方无法接到的位置；敌方有吃血包倾向，我方使用前面的针对策略；对方在中点，随机选择两个角打角，不在中点，选择距离更远的角打角；检查是否有不合理碰撞次数。

[457-523 行]：使用道具函数

优先级为血包-闪现卡-变压器。没有这三种道具且我方寿命很短，用隐身卡，

在对方并不总是打角的时候有几率续命，；如果对方没有给球加速，使用旋转球会造成对方碰撞次数不合理，使用旋转球；其他情况或我方条件不利时使用能使用的道具，但在对方一直打角的情况下不能使用隐身卡。

[526-586 行]：跑位函数

根据敌方策略确定跑位策略，如果我方血量在百分之三十到百分之五十五之间，则跑到接近中点的剩余血量能到所有位置的位置；生成 0-1 随机数，若产生的数小于 0.2（用于迷惑？），或我方血量小于百分之三十，则向中点跑二分之一距离。

[603 行]：主函数

载入历史数据；判断敌方策略；选择我方策略；设置速度限制防止不合理碰撞次数；判断我方想得到的道具，判断是否有路线可捡到，如何捡到，如果没有，调用我方可能策略函数；使用道具；跑位；保存数据

2.3 程序限制

由于我们选择的算法是人为寻找策略并调整参数，采用的数据结构较为简易，所以并没有发现太多的出错情况。

经过讨论，我们认为我们的算法在对敌一些算法时有被克制的情况，尤其是学习算法。

我们曾被带有学习功能的算法打败，原因主要是我们的数值判断、跑位策略有一定部分是按照给定常数来计算的，而未应用变量，学习算法对于常数的判断计算比较容易；而同时我们仍然采用了大量的打角算法，在对对方进行打角判断的同时，自身也可能被对方的策略围困，造成一定可能被克制。

3.实验结果

3.1 测试数据

实验环境说明：

硬件配置：CPU core i7，8G

操作系统：Windows 10

Python 版本：Python 3.6

具体算法及测试数据：

①random：初期写的随机加速、随机跑位、不获取道具的算法，由于随机消耗过大，甚至难以战胜 idiot。

②miaou：第三次组会具体商量策略后用于参加第一次热身赛的算法，其跑位、获取道具策略与上面具体分析的 remeo 有不少相似处，但有所差别：

1）通过判断对方是否有喜欢补血包的倾向，来获得我们的策略。通过多次的储存历史数据判断对方的获取补血包，如果次数大于我们的设定值（有多种多次的调试比例，例如 10 次有血包中大于 2 次吃到等），则认定有这种倾向，我们便打球到对方吃血包要花的代价加上补血量比我们打球平均代价更高的地方（因为判定对方一定会吃血包）。

2）当对方的生命值很小的时候，他将有一定不能跑到的位置，我们将尽量打到这个地方。我们选择对方生命值小于球拍生命值 25%的时候实行这个策略，因为此时对方能跑的最长距离平均也只有球桌长度的一半，我们便可以花费比最短路径等策略更高的代价，直接打到他不能触及的地方。

3）如果对方在中间 20%以内的地方，我们直接打角。

4）如果我们不能做以上的事情，就检查我们是否可以发一些对方不加速就 invalid bounce 的球打过去。

5）在发球时尽量选择的路径是对方不加速就一定 invalid bounce 的做法，且我们发球的代价小于他们接球的最小代价。（已通过算法找到）

6）跑位时，如果我们生命在 30%到 55%之间，则不论对方怎样都跑到一个位置，使得我们下一次一定能够到达任何地方。如果不是上述情况，则随机生成一个 0-1 之间的数，若小于 0.2，或我方生命值小于 30%则向中点跑二分之一的距离。

但是这些判断很多是通过和 idiot 或者之前写出的算法形成的经验性判断，因此造成对历史数据的使用不科学，判断也不精确，甚至还不如不判断，以至于在 150 局热身赛中只获得了 62 局的胜利，面对较强的算法毫无反抗之力。在后期我们调试中不断进行了优化和删除。

③haha：在正式提交代码前用于与别组约架的算法，改良了打角的加速策略，但在一些细节的参数上面没有较好的调试，以至于被 Oscar 组和 Alpha 组的算法完虐。

④hiahia：在第一轮约架失利后由技术人才调试后的产物，在第二轮的约架中表现良好，甚至击败了 Golf 组的 zzzz 算法，但在算法中还存在一些不必要的判断和不太合理的道具使用条件。

⑤remeo：最终提交代码，去除了 hiahia 中的不必要的判断，明确了道具使用条件，在跑位上有了一些优化，在最终竞赛中战胜东区两个对手，获得 6 分，但最终遗憾未能出线。

P.S.每一次更新的代码都能够完虐以前版本的算法，这也是我们最基本的测试方式。

热身赛统计见文后附件 2。

3.3 结果分析

本组代码未用到学习、深度搜索之类的复杂算法，主要是技术人才手动调整参数来让代码升级。我组算法的策略主要是通过历史数据判断对方的策略，然后做出相应的判断，这里对对方的策略尽自己的想象做了枚举，可能就是枚举不足，导致在应对一些较强的算法时我们没能有很好的方法压低对方的生命，导致了最终的失败。但是我组算法对待道具、跑位之类的策略是比较完善的，对各种道具都有优先级的判定、对不同生命范围有着不同的跑位策略，在各种比赛中，鲜有因为跑位或者道具而失利的。虽然最终未能出线，但也不负我组的努力。

4.实验过程总结

4.1 分工与合作

小组成员：王旭斌、李佳益、杨子珍*、冯禄、王今朝、刘小辉、江世豪

任务分配：

1、开发算法和编程任务：王旭斌同学负责代码的撰写，刘小辉同学负责测试、王今朝及江世豪同学负责提供算法思想、冯禄和李佳益同学负责会议总结和

一些细节性问题、杨子珍同学负责照相，大家共同查找资料提供思路。

2、小组报告：刘小辉同学负责实验结果部分，冯禄同学负责算法思想阐述和大体结构框架的撰写，杨子珍同学负责实习过程总结，李佳益同学负责报告各部分的整合和最终的排版，江世豪和王今朝同学负责程序代码说明。

合作讨论方式：

线上：在微信群中进行讨论，通过文字、图片、语音进行交流。

线下：小组会议集中讨论。

小组会议实录：

1、第一次小组会议

参与人员：王旭斌、李佳益、杨子珍*、冯禄、刘小辉、江世豪

时间：2017 年 5 月 15 日 17：00 到 17：40

地点：理教 107

主要内容：当天老师刚刚发布比赛规则，临时决定召开小组第一次会议，因此有一位同学没有来。组内成员互相见面，熟悉彼此。发布任务，希望每位组员仔细剖析比赛规则并查阅相关资料。



第一次会议记录照片

2、第二次小组会议

参与人员：全体人员

时间：2017 年 5 月 22 日 17：00 到 17：50

地点：理教 107

主要内容：针对比赛规则进行商讨，根据每一种道具的实用性确定比赛策略。



第二次会议记录照片

3、第三次小组会议

参与人员：全体人员

时间：2017 年 5 月 25 日 20：40 到 22：00

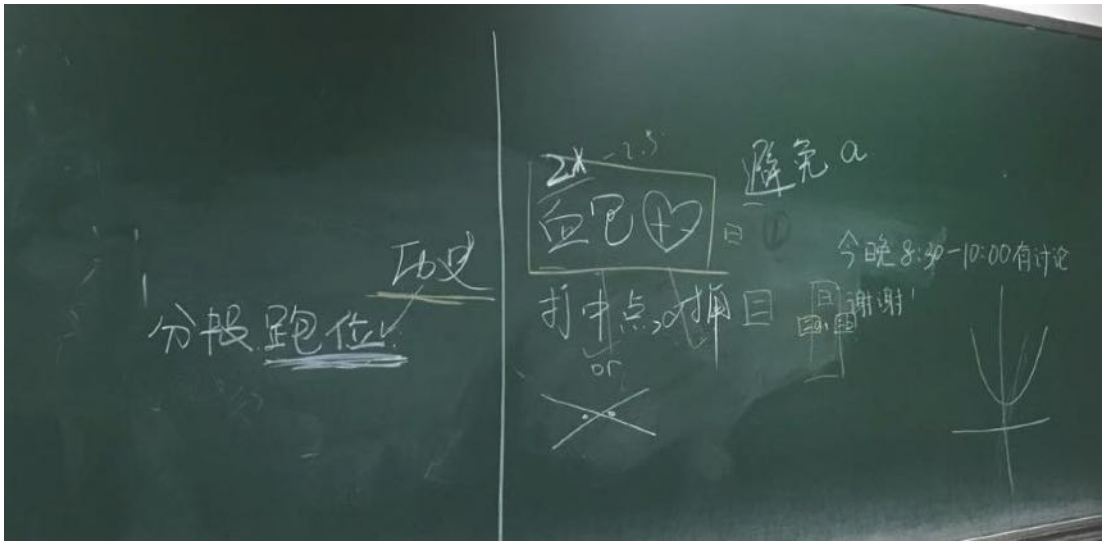
地点：四教 507

主要内容：搭建了算法的大体框架，初步确定了几种策略：根据历史数据分

段跑位、避免加速、吃血包、交叉打角。



第三次会议记录照片



第三次会议记录思维导图草稿

4、第四次小组会议

参与人员：全体人员

时间：2017 年 5 月 28 日 20：30 到 22：10

地点：四教 403

主要内容：在之前算法的框架上进一步完善算法。



第四次会议记录照片

5、第五次小组会议

参与人员：全体人员

时间：2017 年 6 月 3 日 18：00 到 21：30

地点：四教 403

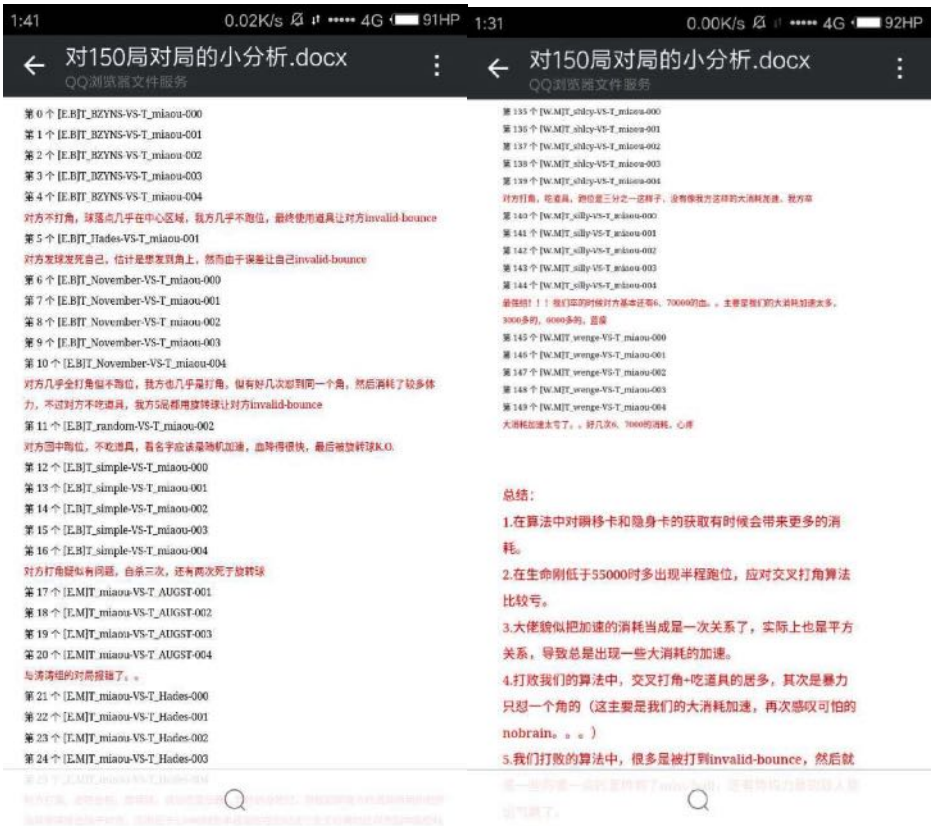
主要内容：根据第一次热身赛的结果，分析我们的不足，调整算法。



第五次会议记录照片-1



第五次会议记录照片-2



第五次会议进展

6、第六次小组会议

参与人员：全体人员

时间：2017 年 6 月 4 日 14：00 到 16：00

地点：二教二楼与三楼楼梯拐角处

主要内容：根据当天和各组友谊赛的结果，分析我们的不足，进一步调整算法。

会议照片：无

4.2 经验与教训

Romeo 组一共有七位组员，每一位组员都很优秀，也很认真，为这次大作业付出了很多。虽然我们小组最后并没有出线，但这并不能抹杀我们的努力。在整个过程中，我们收获了很多，这些是上机考试所不能带给我们的。很高兴能够看到大家从一开始组会时的面面相觑，到最后聚餐的谈天说地。

在刚刚开始我们大作业的时候，我们并没有进行明确的分工，这是希望大家都能参与到算法思路的构建中，不要被分工局限了。但这样大家不知道怎么行动，效率很低，所以在第一、二次小组会议中并没有取得突破性进展。

同时，除了编程人员，其他组员大都对比赛规则了解的不够透彻。因为很多具体的细节是在代码里面体现出来的，所以 India 组的做法可能就会比较好，他们组在初期就组织了大家一起看代码。借鉴过来，如果我们可以让组内优秀的程序员带领组员一起过一遍代码，那么我们对比赛规则的了解就会更透彻。

为了进一步分析我们组的不足，特地采访了一下冠军队伍，他们组是在第一次小组会议时就确定了算法的整体思路——通过深度搜索寻求最优解，在之后进行算法的实现和改进。我们组在初期也提出过类似的想法，但因为计算量等各种原因，很快就舍弃了。在后期我们主要进行的就是制定策略。如此比较来看，在这方面的确不能够勉强，毕竟各个组的策略不同是在所难免的。不过我们组的策略确实存在一定的漏洞，因此在应对一些较强的算法时我们没能有很好的方法压低对方的生命，导致了最终的失败。

在整个过程中，我们面临了很多困难，情绪也几经起伏。从刚开始的毫无头绪，到取得突破性进展的兴奋激动，到第一次热身赛后的失落，到改进算法后的重振精神，再到邀约友谊赛时的跃跃欲试……

现在期末大作业的一切都随着前三名 OMG 的出现而渐渐落幕了，很荣幸能欣赏到这样一场精彩的比赛。最后，谢谢我的组员们，也谢谢所有为了大作业而付出努力的老师和同学们！

4.3 建议与设想

在这样短的时间内，老师和技术组以及每个小组的成员能够把漂移乒乓做成这样，已经是非常不容易的了。但是，也正是因为时间有些仓促，所以的确有些做的不足的地方。首先比赛规则的细节一直在变动，基础设施代码不断在更新，虽然这对我们影响不是很大，但也造成了一定的干扰。其次，比赛输赢的判定希望以后能够尽早明确。这次直到正式比赛的时候才知道比赛采取积分制，在积分相同的条件下看剩余血量。积分怎么给，给多少，这些在比赛之前并没有公布。观看比赛的时候，为那些虽然赢得局数多，但最后却没有出线的小组感到可惜。另外，希望能够开发一个友谊赛平台，方便大家互相约架。

数据结构与算法真的是一门很好的课程，它提供了期末大作业这样一个考核形式，我们不仅能够学到知识，而且还能锻炼自己的能力，培养团队合作精神。小组合作的过程中，大家是一个整体，每个人都要为了整个小组而努力。另外，

比赛肯定会有胜负之分，也会有分数的差别。有人得意，有人失落，我们每个人都需要理性地看待期末大作业的结果，胜也好，输也罢，重要的是在这个过程中所收获的。

5.致谢

感谢陈斌老师一学期以来的指导。

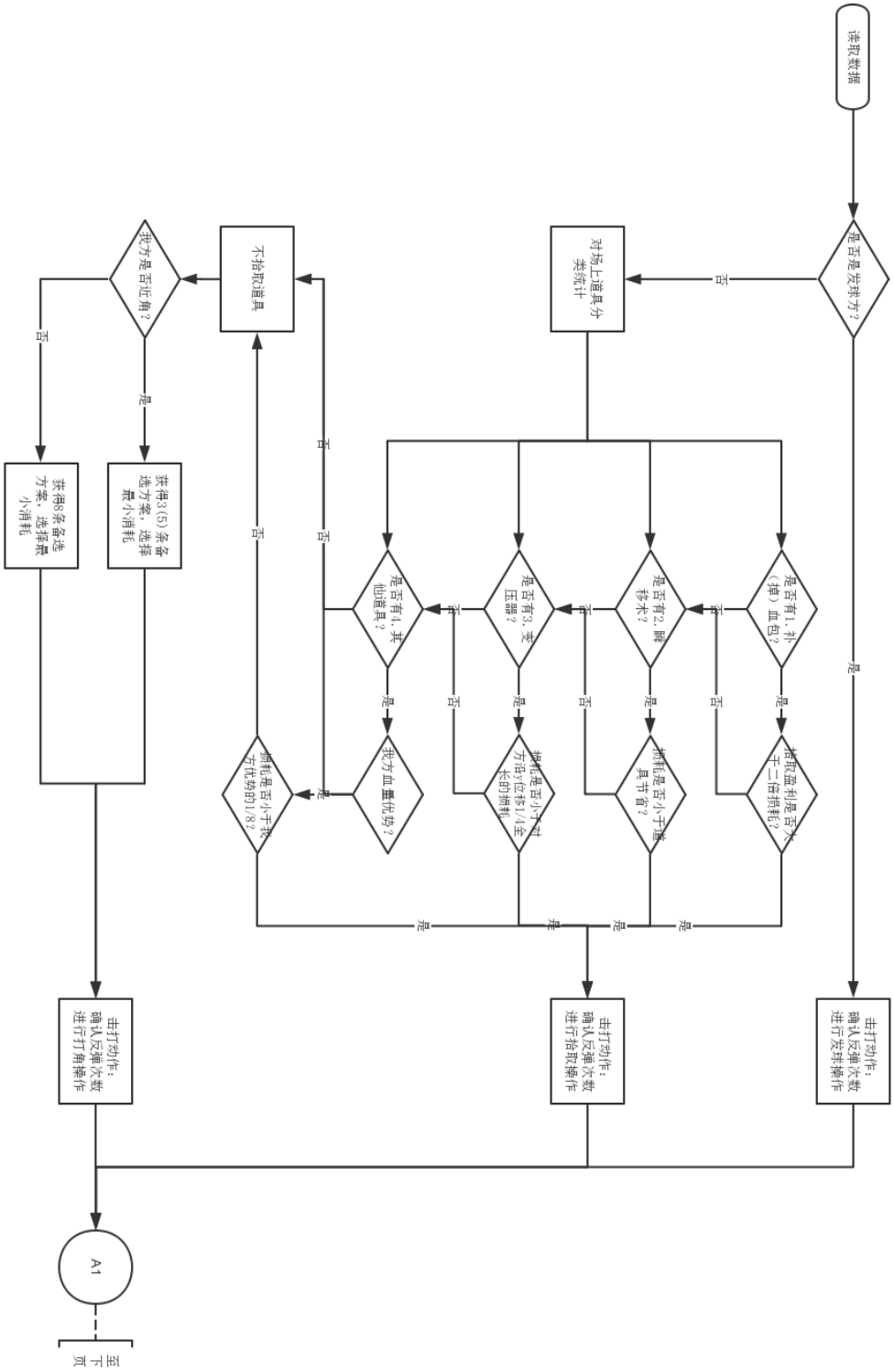
感谢冀锐助教在实习进行以及友谊赛进行过程中提供的帮助。

感谢组员家属在生活中的关照。

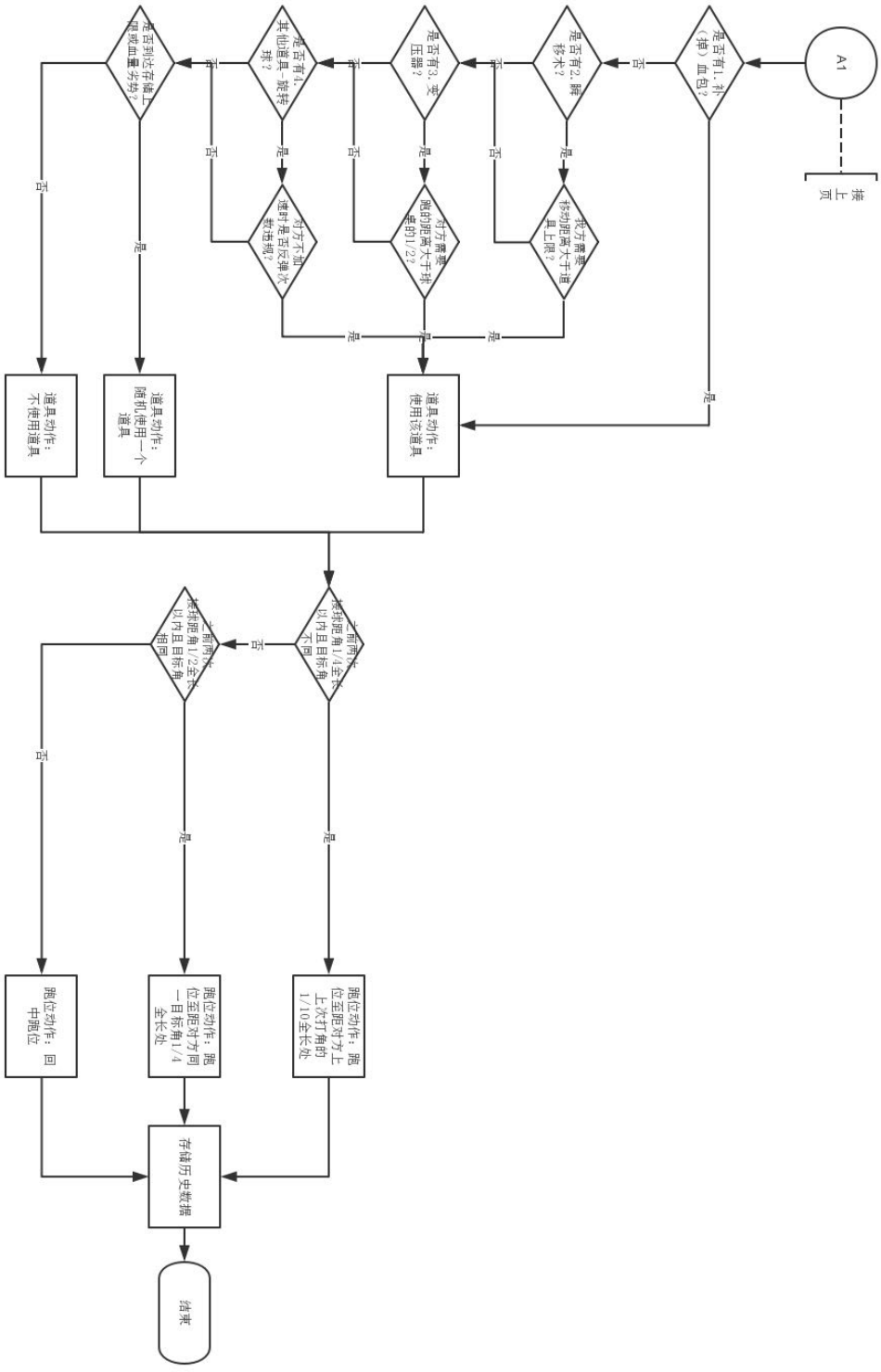
感谢与我们进行友谊赛 alpha、oscar、golf 组。

感谢比赛现场的各位到场同学。

附件 1-算法流程图-1/2


读取数据
是否是发球方?
是
否
击打动作;
确认反弹次数
进行发球操作
对场上道具分
类统计
是否有1.补
(掉)血包?
是否有2.瞬
移术?
是否有3.零
压器?
是否有4.其
他道具?
拾取盈利是否大
于三倍捕耗?
捕耗是否小于对
方具节省?
操作是否小于对
方位移1/4全
长的捕耗?
我方血量优势?
捕耗是否小于我
方优势的1/8?
否
是
不拾取道具
我方是否近角?
否
是
获得3(5)条备
选方案,选择最
小消耗
获得8条备选
方案,选择最
小消耗
击打动作;
确认反弹次数
进行拾取操作
击打动作;
确认反弹次数
进行打角操作
A1
至下页

附件 1-算法流程图-2/2


A1
接上页
是否有1.补(掉)血包?
是否有2.降移木?
是否有3.变压器?
是否有4.旋转其他道具?
是否刮边存储上限或血量劣势?
我方角色移动距离大于道具上限?
对方角色离的距离大于球的1/2?
对方不动时是否反弹次数达成?
道具动作:使用该道具
道具动作:随机使用一个道具
道具动作:不使用道具
上面两个球以内且目标角1/4全长不同
上面两个球以内且目标角1/2全长相同
跑位动作:跑位至距对方上上次打角的1/10全长处
跑位动作:跑位至距对方同一目标角1/4全长处
跑位动作:回中跑位
存储历史数据
结束
是
否

对手	发球方	胜局数	胜因	败局数	败因
BZYNS	BZYNS	5	对方 invalid-bounce	0	——
	miaou	5	对方 invalid-bounce	0	——
November	November	5	对方 invalid-bounce	0	——
	miaou	5	对方 invalid-bounce	0	——
random	random	5	对方 miss-ball	0	——
	miaou	5	对方 miss-ball	0	——
simple	simple	5	对方 invalid-bounce	0	——
	miaou	5	对方 invalid-bounce	0	——
AUGST	AUGST	0	——	5	我方 miss-ball
	miaou	0	——	5	我方 miss-ball
Hades	Hades	0	——	5	我方 miss-ball
	miaou	0	——	5	我方 miss-ball
KILO	KILO	0	——	5	我方 miss-ball
	miaou	0	——	3	我方 miss-ball
nobrain	nobrain	0	——	5	我方 miss-ball
	miaou	0	——	5	我方 miss-ball
PMNDS	PMNDS	0	——	5	我方 miss-ball
	miaou	0	——	5	我方 miss-ball
silly	silly	0	——	5	我方 miss-ball
	miaou	0	——	5	我方 miss-ball
SCRTS	SCRTS	5	对方 miss-ball	0	——
	miaou	5	对方 miss-ball	0	——
Sheikah	Sheikah	2	对方 miss-ball	3	我方 miss-ball
	miaou	0	——	5	我方 miss-ball
shlcy	shlcy	0	——	5	我方 miss-ball
	miaou	0	——	5	我方 miss-ball

wenge	wenge	0	——	5	我方 miss-ball
	miaou	0	——	5	我方 miss-ball
vtest	vtest	5	对方 invalid-bounce	0	——
	miaou	5	对方 invalid-bounce	0	——

附件 2-热身赛统计结果



YANKEE

组长：向汗青

组员：石浩正，许豪伟，周嵘，路鹏飞，张昊勋，侯相至

小组赛战绩：E6



Yankee



数据结构与算法课程实习作业报告

(YANKEE 组)

数学科学学院 向汗青* 1300010707

数学科学学院 石浩正 1300010730

生命科学学院 许豪伟 1300012207

地球与空间科学学院 周嵘 1600012617

地球与空间科学学院 张昊勋 1600012605

地球与空间科学学院 侯相至 1600012643

地球与空间科学学院 路鹏飞 1600012623

摘要：本组的算法，是以数学计算与程序设计相结合的。我们依据“消耗最小、损敌最多”的指导思想，从跑位、加速、道具等各方面加以考虑，对许多条件加以判断，最终一步一步的写出了一个体量小但有成效的乒乓对战算法。在此过程中，组内成员互帮互助、团结努力，针对我们算法的条件与问题进行改进。虽然存在这样或那样的不足，我们小组最终没有实现出线的目标，但是总体而言达到了一个较为满意的结果。

关键字：临界条件，判断语句，道具使用，消极策略，学习算法。

1 算法思想

1.1 总体思路

我们小组算法的总体思路，是尽量减小己方的体力消耗，同时对对方产生最大的影响。因为我们的“乒乓球对战”与实际的乒乓球规则有着较大的区别，无论最后希望以什么样的方式取胜（如使对方接不到球，能接到球但最后体力值优于对方等），其过程都需要以生命值的形式体现。因此可以说，本场比赛与其称之为“乒乓大赛”，称之为“生命力大赛”似乎更为合适一些。为了让自己有最少的体力消耗，则要从给球加速，以及跑位策略两方面考虑；为了给对方最大的影响，则球就要尽量打到对方球场的角上，增加对方的跑位与击球代价。而观察各组的策略，也都是从这两点出发的。各组之间的差异，则主要体现在细节的实现思路上。

在本次作业中，我们需要实现的是发球与击球两个函数。对发球而言，因为位移到发球位置不消耗任何生命值，因此总体思路较为简单：用最大的速度打到对方的角上，使得对方不仅要有最远的跑位，还迫使其施加最大的加速度才能把球成功击回来。这样给对方的消耗无疑是最大的。同时，为了防止对方学习到我们的发球策略，站在原地等着我们的来球，我们还在发球函数中加入一个随机数，当该随机数为偶数时，就从最下方发球打对手的上角；当随机数为奇数时，就从最上方发球打对手的下角。

而对于击球函数来说，我们采取的是较为消极的迎球策略：每次跑位跑到正中间，回球时采用能打到对方角上的最小的加速度。回球的时候，因为有在两侧墙壁碰撞次数的限制，因此根据我们击球时所处的位置，我们击出球的速度有一定的限制。临界速度总共有六种，向上击球与向下击球各三次，且如果按照临界速度来将球回出，则必然来到对方的角上。我们在判断中，认为给球加速的消耗相比跑位而言更大，因此我们采取最节省体力的方法——判断来球的速度改变成哪一个临界速度时，所需的加速度最小，我们就将球加速到这一速度。这样的策略，既满足了打角的要求，也使得自己的消耗最小。在实际操作中，为了避免精确打到角上所可能产生的系统判定问题，我们将速度的范围设定的略小，留出了一定的余量。而关于跑位策略，如果对方来球的位置是随机均匀分布的，或者关于中点对称，则显然中点是方差最小的地方，跑位回到中间是最容易想到、在随机情况下最节省体力的策略。

而在击球函数中，除了我们最为基本的策略之外，还要考虑到一些细节问题。

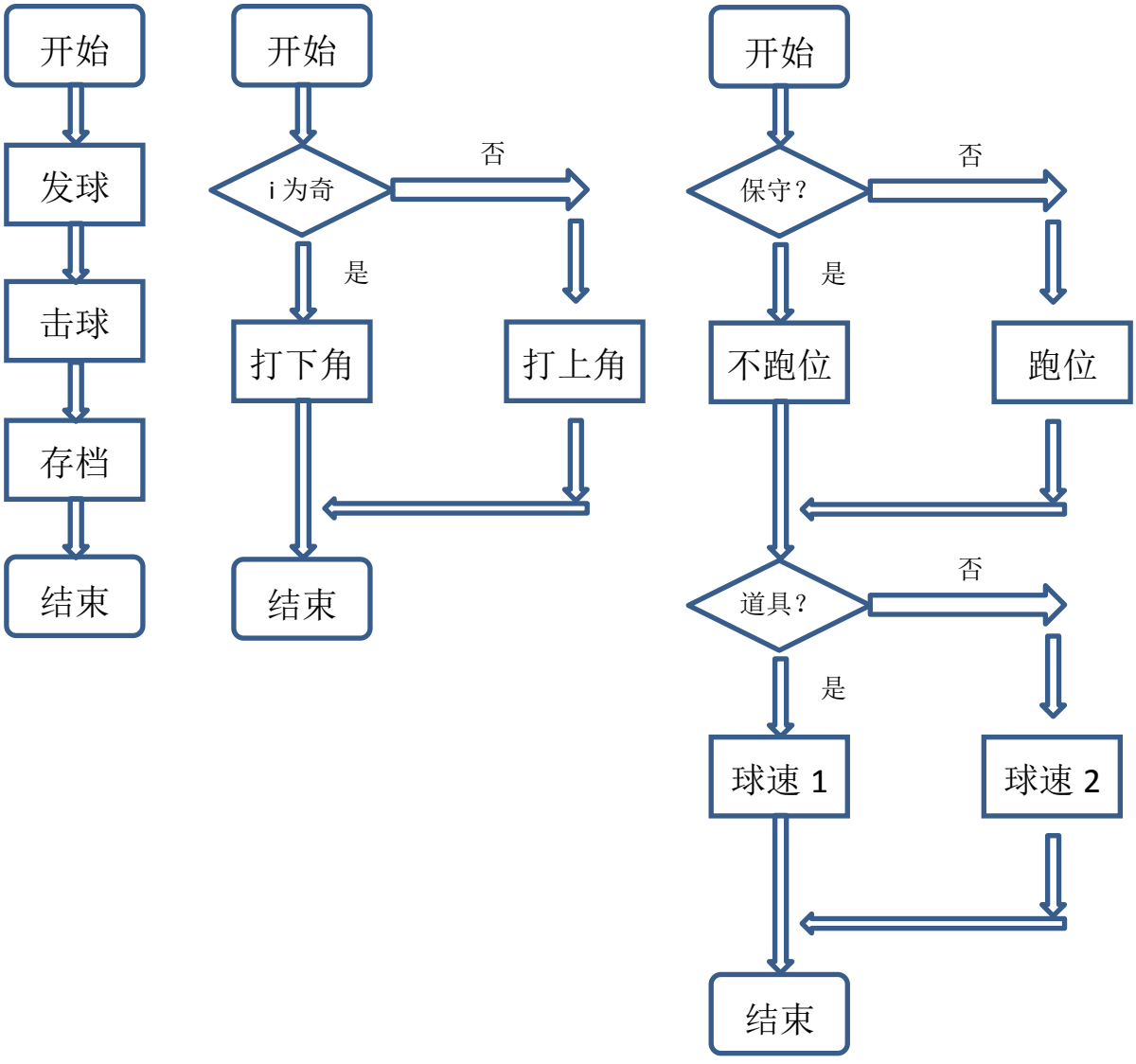
如关于道具的使用。本次比赛，各种条件的设定并没有给我们太多的自由发挥空间，各组的算法都不会有本质上的不同。因此，当两支策略非常类似的队伍相遇的时候，道具的使用就会对比赛结果产生很大的影响。我组的策略是，遇到能打到、值得打的道具就会去打，并且会立刻使用出来。所谓“能打到”，指的是可以击打到道具并且同时满足碰撞次数要求，我们会算出打到道具所需的速度范围，并与之前解出的碰撞临界速度进行比较，判定是否可以打到这一道具。而所谓“值得打”，则主要指补血包与掉血包两种道具，看它能给我方增加或给对方减少的体力值，是否大于我方为了打到这一道具所损失的体力值。如果大于，那么我们就值得改变速度去打这一道具；如果小于，则放弃这一道具。将得到的道具在第一时间用出去的想法，我们的想法是，这样可以保证每一个捡到的道具都能发挥作用，不会产生过期被丢掉的问题，进而对对方产生最大的伤害。

而关于学习，我们小组也编写了一个较为简单的学习与反学习算法。发球函数中的反学习算法如上文所述，在这一段中我们将简要介绍一下我们击球函数中的学习算法。我们认为，大家的策略不外乎两种：消极策略与积极策略。我们自己的迎球策略就是典型的消极策略，

以自己消耗的体力最小化为目标，尽量使得给球的加速度最小；而积极策略则相反，更为灵活多变，会积极改变球速，随机打对手的两个角落，调动对方多多跑位。面对消极策略，我们分析发现，发球之后，随着球局的进行，对手的回球会逐渐集中于某一个角落。这样，采取跑位返回中间的方式就显得不合理了。而面对积极策略，因为其算法的多样性，我们没有想出能很好针对积极算法的学习策略。因此，我们学习算法的核心，就是要判断对手是否采取了和我们一样的消极策略。首先，我们判断对手是否每次都往角上打；如果是，我们再来判断，是否有连续多次对手的回球都来到了同一个角上。如果是，那么我们可以相信，对手采取了消极打法，我们不再跑位。

以上就是我们算法的主要思路。相比起其他一些小组而言，显得比较简单，容易想到，但是每一种设想都有其显而易见的合理性。

1.2 算法流程图



1.3 算法运行时间复杂度分析

我们小组的算法较为简单，主要的构成部分有两方面：判断与运算。其中运算部分，例如对打道具与总体速度的限制，我们都是通过人工计算得出，并直接输入的，因此我们算法中的运算部分，均只有一部，不涉及复杂的算法。因此，根据我们的估计，我们算法的复杂度应为 O(1)。而在比赛与测试的实际运行时，结果也基本是瞬间出来，占用的时间与资源极小。

2 程序代码说明

2.1 数据结构说明

在上文中我们也提到，我们的实际算法其实是比较简单的，算法中设计的主要数据结构是判断语句，个别的地方运用了循环结构。体现我们总体思路以及核心的计算部分，实际是在程序之外完成的。

从我们的算法流程图可以看出，我们的算法的主要作用是，判断出面对每一种情况，我们应该采取什么样的策略，而不是进行复杂的计算。在总体思路部分，我们提及了一些对我们可能采用的操作的限制，如碰撞次数限制、打道具的速度范围限制、道具作用限制等，这些最基本的限制由数学计算产生，而为什么采取这些限制，则是根据我们对可能出现的比赛状况进行的分析，由小组讨论确定大体的方向。我们认为，我们程序需要做到的，是最基本的实现，而实现的具体方法，需要体现我们小组的共同智慧，这是不能被机器所取代的。因此我们可以看到，在实际运行中，我们小组的用时很少，算法执行快并且占据的资源较少。

当然，这种思路是对是错，我们比赛中的具体情况和这一思想有着怎样的关系，这是值得在赛后进行思考的。

2.2 函数说明

本组的算法中，并没有设计程序接口，我们的程序功能是通过三个主函数来实现的。其中第一个函数，即发球函数，其实现在上文中已经提到；而第二个函数，即 summarize 函数，是将我们每一局能获取到的数据记录到一个 list 中。下面，我们来简单分析一下我们主函数中计算的原理。

关于速度临界值：我们画出了当球的落点在纵坐标为 y 的位置时，向上与向下击球到达对方角落的六种回球路径，进而计算出了需要的临界速度。同时，为避免打到角上出现判定问题，我们的程序留出了一定的余量。

关于打道具的方法：我们注意到，以道具位置为中心，做一个菱形，那么可以打到道具的路径，恰好会和两条对角线中的一条相交。利用这一思想，我们做出了可能的路径，并与之前解出的临界速度进行比较。

2.3 程序限制

我们小组的程序总体而言运行较为平稳，从第一次测试环节开始，就没有出过 bug，满足了最为基本的要求。这一点，我认为我们小组的表现是比较出色的。

但是，在实际运行对战中，我们的程序还是出现了一些其他的问题。失球的来源主要有两个：miss ball 与 invalid bounce。其中，涉及到 invalid bounce 的部分，我们小组也开会进行了研究与探讨，并且成功排除了在总体速度限制中，因为打到角上而产生的系统判定问题。但是，在加入使用道具的模块之后，这一问题再次出现，且问题出现的病因我们并没有找到。考虑到热身赛与正赛中，我们因为 invalid bounce 这一问题而产生的数量不小的失球，这一点可以说是我们程序的最大的问题。

此外，我们小组加入的学习算法较为简单，如果是对方有针对性的策略，例如随机出现连续打某一个角，之后再打另一个角的策略，那么我们的学习算法带来的后果将是灾难性的。因为如果出现这种情况，达到一定的次数，我们的程序就会将其判定为消极策略，进而采用不跑位的应对方法。但是如果对方突然换成打另一个角，就会极大的增加我们的体力消耗。

3 实验结果

3.1 测试数据

实验环境说明：

- 硬件配置：（Intel i5-6300/4GB）
- 操作系统：（Windows 7 64 位操作系统）
- Python 版本：（Python 3.5）

关于赛前测试的情况，我们小组并没有编写专门的测试函数。我们平时竞赛的对象，主要是 idiot 与自己的算法。与 idiot 对战，主要是监测我们算法中有没有奇怪的、容易轻易被对手抓住的 bug；而与自己对战，主要是测试我们对战消极策略的算法时，胜率究竟如何，藉以测试更为高级的思路部分。

第一次热身赛的结果，我们的程序运行顺利，但是表现较差，甚至对 idiot 都不能保证全胜。之后，经过后续组会的讨论与改正，我们总体保持了较高的胜率。

我们的热身赛结果如下：

Win	excited	Hellofine	SWIFT	harry	RABBIT	APOLLO	sabac	eighth
invalid bounce	43	30	42	30	18	30	32	27
miss ball	110	23	56	53	13	102	126	74
Lose								
invalid bounce	0	30	0	28	107	0	0	0
miss ball	22	90	72	64	32	53	12	74

Win	simple	David	AVCNA	zxcvb	lkjhg	unique	rsbgd	test
invalid bounce	2	3	37	3	44	24	0	0
miss ball	7	2	91	7	111	14	0	4
Lose								
invalid bounce	139	45	0	15	0	0	1	0
miss ball	22	125	49	13	15	132	9	6

Win	King	Alpha	learn	yzzxx	track	Aldiot_2	exxxx	Luna
invalid bounce	22	33	19	10	40	10	11	13
miss ball	47	124	52	24	61	75	107	115
Lose								
invalid bounce	0	6	6	85	27	0	39	0
miss ball	101	22	93	51	44	120	55	68

Win	AZ_2	wordge	AUGST	Ismael	fool4	PMNDS	wesds	Lazarus
invalid bounce	36	10	29	10	8	17	5	14
miss ball	43	78	14	84	12	90	55	75
time out						2		
life out			1					
Lose								
invalid bounce	0	15	0	0	3	0	12	0
miss ball	127	87	27	101	21	86	16	106
time out							2	

Win	Issac	LNRXZ	Stoke	DHZ_2	ECHO	Wonka	Unicorn	SCRTS
invalid bounce	10	12	10	31	0	13	10	28
miss ball	59	86	55	88	0	90	116	103
Lose								
invalid bounce	0	0	20	0	200	0	0	0
miss ball	126	95	120	83	0	88	69	66
life out		2						

Win	Taichi	Stork	Phoenix	Sheikah	AUGST	vtest	simple	silly
invalid bounce	28	10	20	25	49	10	15	34
miss ball	103	88	73	43	77	15	4	107
time out							2	
life out			1					
Lose								
invalid bounce	0	31	0	10	0	4	126	0
miss ball	97	72	102	72	53	31	6	5
time out							2	

Win	PMNDS	FANCY	Tango	Hades	Noveber	BZYNS	miaou	SCRTS
invalid bounce	32	27	0	42	5	10	35	42
miss ball	19	46	0	76	0	0	27	42
time out	2							
Lose								
invalid bounce	0	4	1	11	145	145	0	0
miss ball	95	24	2	28	0	5	88	66
time out	2							

Win	FMQED	wedge	shlcy	KILO	random	nobrain		
invalid bounce	10	44	43	34	25	30		
miss ball	9	65	68	95	1	34		
Lose								
invalid bounce	0	9	0	0	20	37		
miss ball	2	32	40	21	106	52		

3.2 结果分析

在测试数据中，我们发现，只要是我们发球，对付 idiot 程序也好，自己与自己左右互搏也好，都有着很高的胜率。这说明，我们的发球策略总体还是非常成功的，后续就没有再对发球函数进行调整。

而在热身赛过程中，我们可以看到总体结果还是胜多负少，己方的问题主要体现在 invalid bounce 与 miss ball 两种。invalid bounce 问题我们对程序进行了调整，加大了对速度的限制，在主程序中避免了这一问题，但打道具所引起的这一问题，没有得到有效的解决。而 miss ball 问题则是因为正常的体力值消耗，不存在打不到球的意外情况，且我们的体力值总体占优。而关于运行时间，我们小组的算法本身并不困难，复杂度低，因此耗时很短，瞬

间完成，在测试、热身赛与正赛中都符合我们的预期。而在正赛中，我们的表现并不突出，小组赛遗憾出局。关于这一结果，我们小组之后的反思与总结，将在 4.2 节中进行详细的说明。

4 实习过程总结

4.1 分工与合作

在我们小组，每次组会，算法及其实现的讨论由全体同学参与，集思广益；而在之后的工作中，则各有分工。石浩正同学全权负责撰写小组报告，而其他同学则编入代码组，根据具体要求编入不同的小组，如道具组、学习组等，分别承担代码不同部分的写作任务。

在完成实习作业期间，我们建立了微信群，有什么问题、想法或者通知，都会发在群里，大家及时讨论；而在线下，我们总共召开了四次组会，地点都在二教三楼树下空间。这四次组会大致的内容如下：

第一次组会：5 月 20 日上午 9 点。本次组会，大家都是初次相识，首先进行自我介绍，互相熟悉。然后，我们开始从无到有，讨论算法最基本的思路。首先我们对最初的题目要求进行了深入的研究，讨论我们知道的信息以及不知道的信息，如何对其进行利用。然后，我们初步确定了向中间跑位、根据碰撞次数限制速度大小的策略。最后，我们大致确定了各自的分工情况。因为此时还有许多地方没有定下来，所以我们记录下了感觉不清楚或者有问题的地方，准备之后向老师提问。当天晚上，组长向汗青同学写出了最初版本的代码。



第二次组会：5 月 26 日下午 3 点。本次组会，因为和上次组会之间相隔 5 天，在此期间老师对游戏规则进行了较大幅度的改动与补充。所以一开始，我们对于 Github 上给出的

各种问题进行了研究探讨，对上次组会的一些思路与计算错误进行了改动。之后，我们对道具进行了更为细致的研究，确定了主要思路，即积极打道具，迅速用道具，并对打到道具的数学条件进行了一定的研究，并提出要针对不同道具的特点，对击打道具的代价进行判断。此外，我们还讨论了如何利用历史数据进行学习，改良我们自身的算法。但是这一点，因为并不清楚如何接入历史数据，也不清楚历史数据中包含的信息，因此并没有取得太多的成果。

第三次组会：5 月 28 日下午 2 点 30 分。本次组会，因为是临时组织，有几位同学都有各种各样的事情要忙，未能参会，到场的基本是代码组的核心成员。因为前一天刚刚进行了第一次热身赛，因此这次组会主要是针对复盘结果，进行研究与改进，完成了道具部分与应对道具部分。



第四次组会：6 月 1 日晚上 7 点。本次组合，是根据之前的热身赛的数据进行的针对性讨论。因为临近代码提交的期限，所以这次组会是持续时间最长的一次。在这次组会上，我们的议题主要有两个：第一个是如何避免 invalid bounce 问题。这一点让我们大家都感到极度苦恼，因为此处失球实在过多。我们对代码进行了现场的调试，最终对一开始的速度限制条件进行了小幅度的修改，避开了可能引起系统判定错误的正角上，相对有所收敛。这一修正，在隐藏道具模块的时候，取得了完全的成功，invalid bounce 问题不再出现。但是，道具部分仍然存在着这一问题，我们在现场并未调试成功。第二个是是否、如何加入学习算法。经过长时间的商议，我们改进了之前的思路，确定了如何判断消极策略，之后如何应对。期间我们还花了一定的时间，找到接入历史数据的接口。此次组会的后续是，大家分工，分别进行 invalid bounce 原因的查找，以及学习算法的编写。但是，很遗憾，因为时间过于仓促，原因查找并没有成功，这一问题在最后也没有得到解决。而学习算法得到了成功的写作，但是我们还是有所疏忽，没有能够判断该学习算法的有效性。

总体而言，我认为我们的组会是有意义而且高效的。每次我们的讨论，都是针对最新出现的情况，循序渐进，发现、改正问题，并且加强算法。且平日的交流，我们也会在群里进

行，不会耽误任何时间，极大的促进了各个部分之间的合作。

4.2 经验与教训

本组最大的亮点，在于我们充分的讨论与分工合作。组员中有 16 级地空的小学弟，能够及时获取其他小组的一些想法，并且学习热情充沛，有着非常多非常有用的想法；也有 13 级的大四狗，其中两位来自数院，能够对算法中的数学部分进行计算。我们小组密切合作，集思广益，最终创造出了一个简单、体量小但并不容易被对手击败的算法。

但是，我们小组的成绩最终没有达到我们出线的目标，这是值得我们反思的。我认为我们小组可能存在的问题有两个：

第一，我们没有解决 invalid bounce 的问题。观察我们热身赛的数据，在很多场中，如果不出现这一问题，是可以反败为胜的。因为我们给出了关于总碰撞次数的速度限制，在后续修正中，我们对速度范围做了少许修正，略微缩小了边界，使得球不会恰好打到角上，这就解决了主程序中的 invalid bounce 的问题。但是，在添加道具模块之后，这一问题再次出现，我们尽了最大的努力也没有将病根排查出来。我想，这样的失球应该就是我们在正赛中发挥不佳的最大问题。

第二，关于我们的学习算法。前边也提到，我们的学习算法较为简单，核心就是根据几个球的落点判断对手是否采取了消极策略。这一判断条件，是很容易被对手利用，并且借此给我们致命打击的。因为时间紧促，学习模块在算法提交的前一天才被加入我们的成绩，我们没有来得及认真分析热身赛数据，看看加与不加学习算法，我们的结果有无区别，这一算法对我们的比赛是否起到了正面的作用。如果之后还有机会遇到这样的问题，我认为这是值得认真探究的。

而其他的方面来看，我们的对手确实非常强大。如 Oscar 组的高级搜索算法，我们并没有能力实现好。与高手过招，我们的确存在技不如人的地方，没有什么可遗憾的。虽然最后我们的希望没有实现，但是总体而言，我们还是交出了一份满意的答卷。

4.3 建议与设想

我们认为，本次小组实习大作业的工作还是非常有趣，而且非常有意义的。通过这次经历，我们意识到，原来经过一学期的课程，我们对于数据结构与算法的掌握可以让我们完成如此复杂而精美的程序，在做完并且得知没有 bug 的时候，是非常有成就感的。所以非常希望这个传统能在地空的数算课程上延续下去，让同学们亲自动手，利用所学的专业知识，创造出一些东西来。

但是关于本次竞赛，我觉得乒乓对战这个选题有点过于宽泛，里面需要设定的地方太多。比如按照一开始的规则，利用旋转球道具可以直接一招制敌，这就是设定考虑不周全的地方。这次比赛中，我认为正是因为这样的问题太多，所以需要一直修改参数改变游戏设计，这也直接导致了我们后来没有充足的时间来进行准备与讨论。在这一点上，我觉得乒乓对战的实习任务不如之前的黑白棋。这一点，也希望老师在之后的教学中能够有所考虑，尽量减少对规则的后续修正。

5 致谢

本次实习作业的完成，首先要感谢陈斌老师与冀锐助教，以及所有参与到这一平台的开发中的志愿者。我们有幸能够在这一课堂中，参与如此缤纷多彩的活动，在实践中增强自己分析问题、解决问题的能力，这与老师与助教提供的机会是密不可分的。

其次，我们还要感谢为本次作业进言献策、指出不足的同学们。Github 上的 issue 有很多，而且每一个都是与我们的程序作业息息相关的。这次作业一开始的设定有很多不足，正是班上每一个同学的火眼金睛，弥补了不足，让本次大赛变得严谨而精彩。

最后，我们还要感谢我们小组内的每一个人。在紧张的期末季与毕业季，每个人都为我们小组的成功付出了很大的时间、精力与心血。大家能够积极参与到这次作业的讨论中，集思广益，努力付出，才有了我们这个过程顺畅、颇具实力的 Yankee 组。

6 参考文献

http://gis4g.pku.edu.cn/course/pythonds/



DELTA

组长：贾昊凝

组员：李能韬，史红毅，杨佳琦，覃程，彭路赢，顾书纶

小组赛战绩：E7

Delta 小组数据结构与算法课程实习作业报告

（贾昊凝*、李能韬、杨佳琦、覃程、史红毅、彭路赢、顾书纶）

摘要：

漂移乒乓比赛中，需要各组编写发球、打球、跑位、道具使用以及分析对方行为的算法，通过两两对抗确定输赢。本组采用 Python 语言，综合运用了本课程所学的数据结构与算法思想，编写出本组算法。本报告就算法思想、程序代码、测试过程及实验过程总结等方面展开论述。

关键字：

一步最优，两步最优，估计对手位置

1 算法思想

1.1总体思路

1. 一步最优解算法：

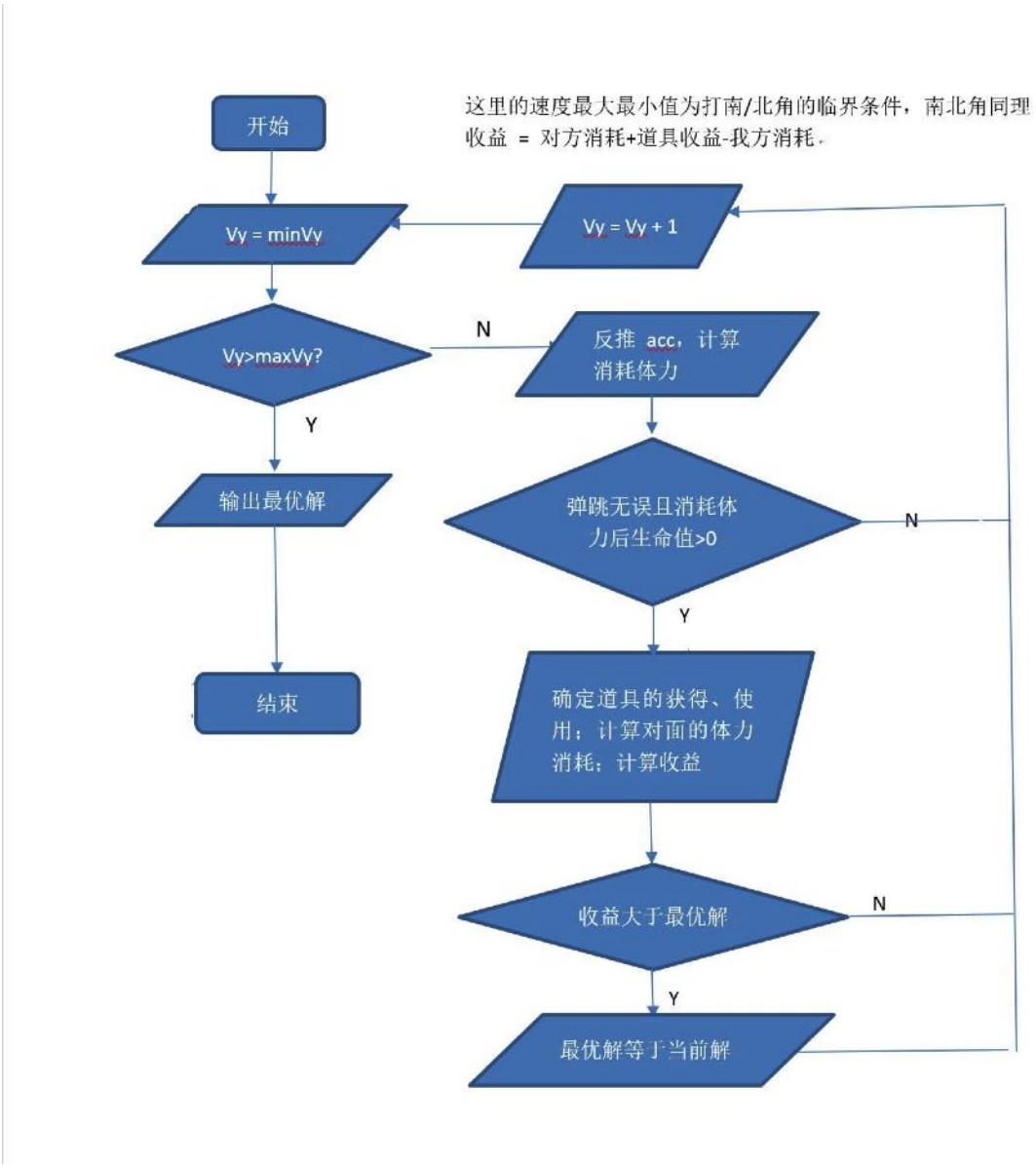
根据对方的击球位置及跑位方向，估计对方跑位之后的位置。经计算，为保证碰壁不超过两次，球 y 方向的速度最大范围是-1666 到 1666。然后以我方击球后球 y 方向的速度 Vy 为参量，计算 Vy 在[-1666,1666]中的情况（不一定要逐个计算），从可行结果（符合碰壁 1-2 次等条件）中找到最优解。这里的最优解指（对方接此球时消耗的 LP-我方加速此球时消耗的 LP）取最大值。不考虑对方的加速、跑位等环节。我方的跑位另行考虑。

道具的使用，简单起见，优先级为加/减血包、瞬移术>变压器>旋转球>隐身术。其中，先使用加/减血包还是瞬移术，看哪个节省的体力值更多。隐身术只在自己的位置靠近中间时使用。

2. 两步最优解算法：

以上述的一步最优解算法作为假想敌。同样先估计对方跑位后的位置。同样以我方击球后球的 Vy 为参量，在可行结果中找最优解。但考虑对方的反应（假设对方就是以一步最优解算法做出反应），使[-（我方加速消耗的 LP+我方跑位消耗的 LP）+（对方迎球消耗的 LP+对方加速消耗的 LP+对方跑位消耗的 LP）-我方下一步迎球消耗的 LP+道具收益等效 LP]取最大值。由于考虑了对方下一步的行动，可以对道具的使用收益做量化。道具收益包括使用道具的收益，和获得桌面上道具的价值（加/减血包价值为 2000，瞬移术平均价值更高些，其他道具的价值不好估计，要在实战中摸索）。具体来说，可对我方使用道具箱中任一道具的情况和不使用道具的情况（道具箱中最多三个道具，加上不使用的情况，计算复杂度最多增加为原来的四倍）分别找最优解，再选择其中最好的一种。（可设定一个阈值，如果使用道具的收益比不使用道具高不了多少，就先留着不用）此算法先按我们击球后的 Vy 值进行搜索，在判断对方的反应时还要进行一轮搜索，总共两层循环，时间复杂度高出一次方。

1.2算法流程图



1.3算法运行时间复杂度分析

储存基本数值：赋值操作，20 次左右
使用道具：循环，最多 3 次
捡道具：遍历桌上所有道具，最多 10 次
估计对方位置：条件判断，3 层循环
加速度选择：遍历可行速度范围，O（n）级别，n 是速度范围，数量级为 10^3 级别
一击必杀：判断，若能，为常数级别；不能，遍历可行速度范围

2 程序代码说明

2.1数据结构说明

所用到的基本数据结构有：线性数据结构
所用到的类有：①math
②python 内置基本类（列表、字典、元组等）

2.2函数说明

```
from table import *

# 发球函数，总是做为 West 才发球
# op_side 为对手方名称，以便从 ds 中查询以往记录数据
# ds 为函数可以利用的存储字典
# 函数需要返回球的 y 坐标，和 y 方向的速度
def serve(op_side: str, ds: dict) -> tuple:
    if (ds == {}):
        ds = {'pre':[] ,'now':[]}
    originalposition = (DIM[3] - DIM[2]) / 2
    originalvelocity = int((originalposition) / ((DIM[1] - DIM[0]) / BALL_V[0])) + 1
    ds['now'].append((op_side,originalposition, originalvelocity ))
    return originalposition, originalvelocity  # 返回球的 y 坐标，和 y 方向上的速度

# 打球函数
# tb 为 TableData 类型的对象
# ds 为函数可以利用的存储字典
# 函数需要返回一个 RacketAction 对象
def play(tb:TableData, ds:dict) -> RacketAction:
    # 设定字典
    if (ds == {}):
        ds = {'pre':[] ,'now':[]}
```

（道具说明：

SP 旋转球：用于抵消被用道具方施加在球拍上的加速的一半；

DS 隐身术：隐藏被用道具方跑位方向 1 次；

DL 掉血包：给被用道具方减血（减少体力值）；

TP 瞬移术：被用道具方可以移动一段距离而不消耗体力值；

AM 变压器：放大被用道具方的体力值损失；体力值损失增加 1 倍。

小组同学讨论，当有多个道具时，应根据各道具的实用性来设置使用优先级，并赋予各道具绝对价值，绝对价值将决定我们是否会为了捡这个道具而选择加速。事实上绝对价值的设置是一个难题，在后面的分析中我们会具体说到）

```
# 道具使用优先级，4 最优先
CardUsing = {'IL':4 ,'DL':4 ,'TP':4 ,'AM':3 ,'SP':2 ,'DS':1}
# 道具价值
CardValue = {'TP':2500 ,'IL':2000 ,'DL':2000 ,'AM':1500 ,'SP':1200 ,'DS':1000}
# 来球信息
pos = tb.ball['position']                # 来球位置
vel = tb.ball['velocity']                # 来球速度
# 对手信息
op_lif = tb.op_side['life']              # 生命
op_pos = tb.op_side['position'].y        # 位置
op_car = tb.op_side['cards']             # 道具箱
op_acc = tb.op_side['accelerate']        # 加速
op_act_car = tb.op_side['active_card']   # 用的道具
op_run = tb.op_side['run_vector']        # 跑位
# 我方信息
my_lif = tb.side['life']                 # 生命
my_pos = tb.side['position'].y           # 位置
my_car = tb.side['cards']                # 道具箱
# 球桌信息
tb_car = tb.cards['cards']               # 桌上有什么道具
tb_car_tic = tb.cards['card_tick']       # 道具沙漏
y = DIM[3] - DIM[2]                      # 球桌的宽
x = DIM[1] - DIM[0]                      # 球桌的长
# 分析出球速度
t = x / vel.x    # 球的飞行时间
vy_max_nor = int((3 * y - pos.y) / t) - 100    # 球向北击出的最大速度（正值）
vy_min_nor = int((y - pos.y) / t) + 100    # 球向北击出的最小速度（正值）
vy_max_sou = -(int((2 * y + pos.y) / t)) + 100    # 球向南击出的最大速度（负值）
vy_min_sou = -(int(pos.y / t)) - 100    # 球向南击出的最小速度（负值）
```

（这里我们求出了向南向北分别反弹一次、两次的临界速度，并试图通过加减 100 的方法避免因误差带来的反弹零次或三次的情况）

```
# 迎球
bat = pos.y - my_pos
# 用道具
card_use = None
card_side = None
if (len(my_car) > 0):    # 如果我方有道具
```

```
        card_use = my_car[0]
        for c in my_car:    # 就选出优先级最高的来用
            if (CardUsing[c.code] > CardUsing[card_use.code]):
                card_use = c
```

（如果我方有道具，先将使用的道具的默认值设为第一个道具，随后将我们所有的道具进行遍历，选出之前所设优先级最高的道具使用，这样做能使道具的利用率最大化，当然前提是我们的优先级设置合理。）

```
    if (card_use != None):    # 如果我方有道具，就选择被用道具方
            if (card_use.code in ['DS','IL','TP']):    # 隐身、加血、瞬移用于我们自己
            card_side = 'SELF'
        else:
            card_side = 'OPNT'
    # 捡道具
    pick = False  # 捡道具标志
    pick_car = []    # 把桌上所有应该捡的道具存入该列表
    car_pick = None    # 最终捡的道具
    if (len(tb_car) > 0):  # 如果桌子上有道具
        if (my_car.isfull()): # 如果我方道具箱已满，就和要溢出的道具进行价值比较
            for c in tb_car:  # 对桌上的道具循环
                if (CardValue[my_car[0].code] < CardValue[c.code]):  # 如果比要溢出的道具有价值就加入捡的列表
```

（此处考虑了两个道具的相对价值大小）

```
                    pick_car.append(c)
            if (pick_car != []):    # 如果有应捡道具在列表中
                pick = True    # 将捡道具标志设为 True
                for i in range(len(pick_car)):    # 用冒泡法将列表中的道具按价值从大到小排序
                    for j in range(i, len(pick_car)):
                        if (CardValue[pick_car[i].code] < CardValue[pick_car[j].code]):
                            pick_car[i], pick_car[j] = pick_car[j], pick_car[i]
        else:  # 如果道具箱没满，就将桌上所有道具存入列表
            pick = True    # 标记为 TRUE
            pick_car = tb_car    # 将桌上所有道具存入列表
            for i in range(len(tb_car)):    # 用冒泡法按其价值从大到小排序
                for j in range(i,len(tb_car)):
                    if (CardValue[pick_car[i].code] < CardValue[pick_car[j].code]):
                        pick_car[i],pick_car[j] = pick_car[j],pick_car[i]
```

（同样这里没有考虑道具的相对价值大小）

```
    # 加速
    if (op_lif > 0.35 * RACKET_LIFE):    # 如果对方生命大于 35%
        if (op_act_car[1] != CARD_DSPR):    # 如果对方没有使用隐身术
            if (op_pos >= ((5 * y) // 8)):    # 如果对方迎球位置在 5/8 处以上
                if (op_run >= 1):    # 如果对方向北跑位或不跑，就假设他向北跑位至可跑距离的 1/4 处
                    op_new_pos = op_pos + ((y - op_pos) // 4)
```

```
                else:    # 如果对方向南跑位，就假设他归中
                    op_new_pos = y // 2
            elif (op_pos <= ((3 * y) // 8)):    # 如果对方迎球位置在 3/8 处以下
                if (op_run >= 1):    # 如果对方向北跑位或不跑，就假设他归中
                    op_new_pos = y // 2
                else:    # 如果对方向南跑位，就假设他向南跑位至可跑距离的 1/4 处
                    op_new_pos = op_pos - (op_pos // 4)
            elif ((op_pos < ((5 * y) // 8)) and (op_pos > (y // 2))):    # 如果对方迎球位置在
5/8 到 1/2 之间
                if (op_run >= 1):    # 如果对方向北跑位或不跑，就假设他在 2/3 处
                    op_new_pos = (2 * y) // 3
                else:    # 如果对方向南跑位，就假设他归中
                    op_new_pos = y // 2
            elif (op_pos == y // 2):    # 如果对方迎球位置在中点
                if (op_run >= 1):    # 如果对方向北跑位或不跑，就假设他在 5/8 处
                    op_new_pos = (5 * y) // 8
                else:    # 如果对方向南跑，就假设他在 3/8 处
                    op_new_pos = (3 * y) // 8
            else:    # 如果对方迎球位置在 3/8 到 1/2 之间
                if (op_run >= 1):    # 如果对方向北跑位或不跑，就假设他归中
                    op_new_pos = y // 2
                else:    # 如果对方向南跑位，就假设他在 1/3 处
                    op_new_pos = y // 3
        else:    # 如果对方用了隐身术
            op_new_pos = y // 2
```

（以上 3/8、5/8 等数字是由小组成员通过预测对方常规跑位方向及距离的套路而确定的，即一步最优的思想。）

```
        # 遍历所有能达到的速度，并选出最优出球速度
        lif_comp = lif_pic_comp = (0,0)    # 前者是所有方案中的最好方案，后者是所有能捡到道
具的方案中的最好方案
        # 遍历所有朝南的出球速度
        for v in range(vy_max_sou,vy_min_sou): # 向南打球
            temp_lif_comp = temp_lif_pic_comp = (0,0)    # 前者存本次方案，若能捡到道具则也存
入后者
            temp_acc = int(v - vel.y)    # 本方案需要的加速度
            acc_vel = temp_acc + vel.y    # 加速后的实际速度（考虑到加速度被 int 会使得实际速
度与设想有偏差）
            if (acc_vel <= -(y + pos.y) / t):    # 如果要碰壁两次，op_bat_dis 为对方迎球距离
                op_bat_dis = pos.y + 2 * y + acc_vel * t - op_new_pos
            else:    # 如果只碰壁一次
                op_bat_dis = -pos.y - acc_vel * t - op_new_pos
```

（上式通过数学方法计算对方的移动距离关于我方加速后的实际速度和我方所处位置的数学函数）

```
            if (op_act_car[1] == CARD_SPIN):    # 如果对方使用旋转球，则需要将加速度放大一倍
                temp_acc *= 2
```

```
            my_de_lif = ((abs(temp_acc) ** 2 // FACTOR_SPEED ** 2) * (CARD_AMPL_PARAM if
op_act_car[1] == CARD_AMPL else 1))  # 我方加速耗血量
            op_de_lif = (abs(op_bat_dis) ** 2 // FACTOR_DISTANCE ** 2) * (CARD_AMPL_PARAM if
card_use == CARD_AMPL else 1)  # 对方迎球耗血量
            if ((card_use != None) and (card_use != CARD_AMPL)):  # 如果我方用了道具，就用它
的价值来抵消损耗的生命值
            my_de_lif -= CardValue[card_use.code]
            temp_lif_comp = (temp_acc,op_de_lif - my_de_lif)    # 以(加速度，对方耗血与我方耗
血之差)的形式记录本次方案
            # 如果要捡道具
            if pick:
                for c in pick_car:    # 遍历要捡的道具
                    A1 = (acc_vel * (-c.pos.x + pos.x) + vel.x * (c.pos.y - pos.y))
                    A2 = (acc_vel * (-c.pos.x + pos.x) + vel.x * (-c.pos.y - pos.y))
                    delta = (2 * abs(vel.x) * DIM[3])
                    if ((min(A1 % delta, -A1 % delta, A2 % delta, -A2 % delta) / math.sqrt(vel.x
** 2 + acc_vel ** 2)) < 1800):    # 如果能拿到就做记录
                        temp_lif_comp = list(temp_lif_comp)
                        temp_lif_comp[1] += CardValue[c.code]
                        temp_lif_comp = tuple(temp_lif_comp)
                        my_de_lif = ((abs(temp_acc) ** 2 // FACTOR_SPEED ** 2) * (CARD_AMPL_PARAM
if op_act_car[1] == CARD_AMPL else 1)) - CardValue[c.code]    # 我方加速耗血量，如果对方用变
压器就加倍
                        op_de_lif = (abs(op_bat_dis) ** 2 // FACTOR_DISTANCE ** 2) *
(CARD_AMPL_PARAM if card_use == CARD_AMPL else 1)    # 对方迎球耗血量，如果我方用变压器就加倍
                        if ((card_use != None) and (card_use != CARD_AMPL)):    # 如果我方用
了除变压器外的道具，就用它的价值来抵消损耗的生命值
                            my_de_lif -= CardValue[card_use.code]
                        temp_lif_pic_comp = (temp_acc,op_de_lif - my_de_lif)    # 存入本次方
案
                        if (temp_lif_pic_comp[1] > lif_pic_comp[1]):    # 若本次方案优于之前
的最优方案则替换之
                            lif_pic_comp = temp_lif_pic_comp
                        break    # 若已有能拿到的道具就退出循环
            if (temp_lif_comp[1] > lif_comp[1]):    # 如果之前的最优方案不如本次，则替换之
                lif_comp = temp_lif_comp
        if (lif_comp[1] > lif_pic_comp[1]):
            best1 = lif_comp
        else:
            best1 = lif_pic_comp
        lif_comp = lif_pic_comp = (0,0)    # 同上
        # 对向北出球的速度进行遍历
        for v in range(vy_min_nor,vy_max_nor):
            temp_lif_comp = temp_lif_pic_comp = (0,0)
```

```
            temp_acc = int(v - vel.y)
            acc_vel = vel.y + temp_acc
            if (acc_vel <= (2 * y - pos.y) / t):
                op_bat_dis = 2 * y - pos.y - acc_vel * t - op_new_pos
            else:
                op_bat_dis = acc_vel * t + pos.y - 2 * y - op_new_pos
            if (op_act_car[1] == CARD_SPIN):
                temp_acc *= 2
            my_de_lif = ((abs(temp_acc) ** 2 // FACTOR_SPEED ** 2) * (CARD_AMPL_PARAM if
op_act_car[1] == CARD_AMPL else 1))  # 我方加速耗血量
            op_de_lif = (abs(op_bat_dis) ** 2 // FACTOR_DISTANCE ** 2) * (CARD_AMPL_PARAM if
card_use == CARD_AMPL else 1)  # 对方迎球耗血量
            if ((card_use != None) and (card_use != CARD_AMPL)):  # 如果我方用了道具，就用它
的价值来抵消损耗的生命值
                my_de_lif -= CardValue[card_use.code] （这证明了我们后面的一个论点：道具价值
的设定与生命值是紧密相关的）
            temp_lif_comp = (temp_acc,op_de_lif - my_de_lif)
            if pick
                for c in pick_car:
                    A1 = (acc_vel * (-c.pos.x + pos.x) + vel.x * (c.pos.y - pos.y))
                    A2 = (acc_vel * (-c.pos.x + pos.x) + vel.x * (-c.pos.y - pos.y))
                    delta = (2 * abs(vel.x) * DIM[3])
                    if (min(A1 % delta, -A1 % delta, A2 % delta, -A2 % delta) / math.sqrt(vel.x
** 2 + acc_vel ** 2) < 1800):  # 如果能拿到就做记录
                        temp_lif_comp = list(temp_lif_comp)
                        temp_lif_comp[1] += CardValue[c.code]
                        temp_lif_comp = tuple(temp_lif_comp)
                        my_de_lif = ((abs(temp_acc) ** 2 // FACTOR_SPEED ** 2) * (CARD_AMPL_PARAM
if op_act_car[1] == CARD_AMPL else 1)) - CardValue[c.code]  # 我方加速耗血量
                        op_de_lif = (abs(op_bat_dis) ** 2 // FACTOR_DISTANCE ** 2) *
(CARD_AMPL_PARAM if card_use == CARD_AMPL else 1)  # 对方迎球耗血量
                        if ((card_use != None) and (card_use != CARD_AMPL)):  # 如果我方用了
道具，就用它的价值来抵消损耗的生命值
                            my_de_lif -= CardValue[card_use.code]
                        temp_lif_pic_comp = (temp_acc,op_de_lif - my_de_lif)
                        if (temp_lif_pic_comp[1] > lif_pic_comp[1]):
                            lif_pic_comp = temp_lif_pic_comp
                        break
            if (temp_lif_comp[1] > lif_comp[1]):
                lif_comp = temp_lif_comp
        if (lif_comp[1] > lif_pic_comp[1]):
            best2 = lif_comp
        else:
            best2 = lif_pic_comp
```

```
        #接下来从向北和向南出球的最佳方案中选出最佳方案
        if (best1[1] > best2[1]):
            best = best1
        else:
            best = best2
        acc = best[0]
    else:    # 如果对方生命值小于35%（35%是小组同学调试后得到的最优经验值）
        if (op_act_car[1] != CARD_DSPR):    # 如果对方没有使用隐身术
            if (op_pos >= ((5 * y) // 8)):    # 如果对方迎球位置在5/8处以上
                if (op_run >= 1):    # 如果对方向北跑位或不跑，就假设他向北跑位至可跑距离的1/4处
                    op_new_pos = op_pos + ((y - op_pos) // 4)
                else:    # 如果对方向南跑位，就假设他归中
                    op_new_pos = y // 2
            elif (op_pos <= ((3 * y) // 8)):    # 如果对方迎球位置在3/8处以下
                if (op_run >= 1):    # 如果对方向北跑位或不跑，就假设他归中
                    op_new_pos = y // 2
                else:    # 如果对方向南跑位，就假设他向南跑位至可跑距离的1/4处
                    op_new_pos = op_pos - (op_pos // 4)
            elif ((op_pos < ((5 * y) // 8)) and (op_pos > (y // 2))):    # 如果对方迎球位置在5/8到1/2之间
                if (op_run >= 1):    # 如果对方向北跑位或不跑，就假设他在2/3处
                    op_new_pos = (2 * y) // 3
                else:    # 如果对方向南跑位，就假设他归中
                    op_new_pos = y // 2
            elif (op_pos == y // 2):    # 如果对方迎球位置在中点
                if (op_run >= 1):    # 如果对方向北跑位或不跑，就假设他在5/8处
                    op_new_pos = (5 * y) // 8
                else:    # 如果对方向南跑，就假设他在3/8处
                    op_new_pos = (3 * y) // 8
            else:    # 如果对方迎球位置在3/8到1/2之间
                if (op_run >= 1):    # 如果对方向北跑位或不跑，就假设他归中
                    op_new_pos = y // 2
                else:    # 如果对方向南跑位，就假设他在1/3处
                    op_new_pos = y // 3
（这一段代码是根据对方跑位前的位置和跑位方向来确定跑位后的新位置）
        else:    # 如果对方用了隐身术
            op_new_pos = y // 2
        # 接下来判断是否可以一击必杀（打角）
        nor_dis = y - op_new_pos  # 打北角对方需要移动的距离
        sou_dis = op_new_pos - DIM[2]  # 打南角对方需要移动的距离
        nor_fly = sou_fly = con_fly = False  # 分别为打北角、打南角、打角的标记
        # 先判断北角
```

```
        if (int(((op_lif / RACKET_LIFE) * BALL_V[0]) * t) < nor_dis):
            best_acc = int(-(y + pos.y) / t - vel.y)
（小组考虑到直接将打北角当作反弹有风险，于是决定先打一次南边，再打向北角）
            if (op_act_car[1] == CARD_SPIN):
                best_acc *= 2
            nor_de_lif = (4 if op_act_car[1] == CARD_SPIN else 1) * ((abs(best_acc) ** 2 //
FACTOR_SPEED ** 2) * (CARD_AMPL_PARAM if op_act_car[1] == CARD_AMPL else 1))  # 我方加速耗血
量
            if (my_lif > nor_de_lif):（判断我方生命值是否能承受本次消耗）
                nor_fly = True
                nor_acc = best_acc
        # 再判断南角（判断南角同理）
        if (int(((op_lif / RACKET_LIFE) * BALL_V[0]) * t) < sou_dis):
            best_acc = int((2 * y - pos.y) / t - vel.y)
            if (op_act_car[1] == CARD_SPIN):
                best_acc *= 2
            sou_de_lif = (4 if op_act_car[1] == CARD_SPIN else 1) * ((abs(best_acc) ** 2 //
FACTOR_SPEED ** 2) * (CARD_AMPL_PARAM if op_act_car[1] == CARD_AMPL else 1))  # 我方加速耗血
量
            if (my_lif >sou_de_lif):
                sou_fly = True
                sou_acc = best_acc
        # 选出最佳一击必杀方案
        if nor_fly and sou_fly:
            con_fly = True
            if (abs(nor_acc) > abs(sou_acc)):
                acc = sou_acc
            else:
                acc = nor_acc
        elif nor_fly:
            con_fly = True
            acc = nor_acc
        elif sou_fly:
            con_fly = True
            acc = sou_acc
        # 如果生命值不能承受任何一次打角即不一击必杀，则继续之前的一步最优打法，代码跟 35%以
上的打法并无区别
        if not con_fly:
            lif_comp = lif_pic_comp = (0, 0)
            for v in range(vy_max_sou, vy_min_sou):
                temp_lif_comp = temp_lif_pic_comp = (0, 0)
                temp_acc = int(v - vel.y)
                acc_vel = temp_acc + vel.y
                if (acc_vel <= -(y + pos.y) / t):
```

```
                    op_bat_dis = pos.y + 2 * y + acc_vel * t - op_new_pos
                else:
                    op_bat_dis = -pos.y - acc_vel * t - op_new_pos
                if (op_act_car[1] == CARD_SPIN):
                    temp_acc *= 2
                my_de_lif = ((abs(temp_acc) ** 2 // FACTOR_SPEED ** 2) * (CARD_AMPL_PARAM if
op_act_car[1] == CARD_AMPL else 1))  # 我方加速耗血量
                op_de_lif = (abs(op_bat_dis) ** 2 // FACTOR_DISTANCE ** 2) * (CARD_AMPL_PARAM
if card_use == CARD_AMPL else 1)  # 对方迎球耗血量
                if ((card_use != None) and (card_use != CARD_AMPL)):  # 如果我方用了道具，就
用它的价值来抵消损耗的生命值
                    my_de_lif -= CardValue[card_use.code]
                temp_lif_comp = (temp_acc, op_de_lif - my_de_lif)
                if pick:
                    for c in pick_car:
                        A1 = (acc_vel * (-c.pos.x + pos.x) + vel.x * (c.pos.y - pos.y))
                        A2 = (acc_vel * (-c.pos.x + pos.x) + vel.x * (-c.pos.y - pos.y))
                        delta = (2 * abs(vel.x) * DIM[3])
                        if (min(A1 % delta, -A1 % delta, A2 % delta, -A2 % delta) / math.sqrt(vel.x
** 2 + acc_vel ** 2) < 1800):  # 如果能拿到就做记录
                            temp_lif_comp = list(temp_lif_comp)
                            temp_lif_comp[1] += CardValue[c.code]
                            temp_lif_comp = tuple(temp_lif_comp)
                            my_de_lif = ((abs(temp_acc) ** 2 // FACTOR_SPEED ** 2) *
(CARD_AMPL_PARAM if op_act_car[1] == CARD_AMPL else 1)) - CardValue[c.code]  # 我方加速耗血量
                            op_de_lif = (abs(op_bat_dis) ** 2 // FACTOR_DISTANCE ** 2) *
(CARD_AMPL_PARAM if card_use == CARD_AMPL else 1)  # 对方迎球耗血量
                            if ((card_use != None) and (card_use != CARD_AMPL)):  # 如果我方
用了道具，就用它的价值来抵消损耗的生命值
                                my_de_lif -= CardValue[card_use.code]
                            temp_lif_pic_comp = (temp_acc, op_de_lif - my_de_lif)
                            if (temp_lif_pic_comp[1] > lif_pic_comp[1]):
                                lif_pic_comp = temp_lif_pic_comp
                            break
                if (temp_lif_comp[1] > lif_comp[1]):
                    lif_comp = temp_lif_comp
            if (lif_comp[1] > lif_pic_comp[1]):
                best1 = lif_comp
            else:
                best1 = lif_pic_comp
            lif_comp = lif_pic_comp = (0, 0)
            for v in range(vy_min_nor, vy_max_nor):
                temp_lif_comp = temp_lif_pic_comp = (0, 0)
                temp_acc = int(v - vel.y)
```

```
                acc_vel = vel.y + temp_acc
                if (acc_vel <= (2 * y - pos.y) / t):
                    op_bat_dis = 2 * y - pos.y - acc_vel * t - op_new_pos
                else:
                    op_bat_dis = acc_vel * t + pos.y - 2 * y - op_new_pos
                if (op_act_car[1] == CARD_SPIN):
                    temp_acc *= 2
                my_de_lif = ((abs(temp_acc) ** 2 // FACTOR_SPEED ** 2) * (CARD_AMPL_PARAM if
op_act_car[1] == CARD_AMPL else 1))  # 我方加速耗血量
                op_de_lif = (abs(op_bat_dis) ** 2 // FACTOR_DISTANCE ** 2) * (CARD_AMPL_PARAM
if card_use == CARD_AMPL else 1)  # 对方迎球耗血量
                if ((card_use != None) and (card_use != CARD_AMPL)):  # 如果我方用了道具，就
用它的价值来抵消损耗的生命值
                    my_de_lif -= CardValue[card_use.code]
                temp_lif_comp = (temp_acc, op_de_lif - my_de_lif)
                if pick:
                    for c in pick_car:
                        A1 = (acc_vel * (-c.pos.x + pos.x) + vel.x * (c.pos.y - pos.y))
                        A2 = (acc_vel * (-c.pos.x + pos.x) + vel.x * (-c.pos.y - pos.y))
                        delta = (2 * abs(vel.x) * DIM[3])
                        if (min(A1 % delta, -A1 % delta, A2 % delta, -A2 % delta) / math.sqrt(vel.x
** 2 + acc_vel ** 2) < 1800):  # 如果能拿到就做记录
                            temp_lif_comp = list(temp_lif_comp)
                            temp_lif_comp[1] += CardValue[c.code]
                            temp_lif_comp = tuple(temp_lif_comp)
                            my_de_lif = ((abs(temp_acc) ** 2 // FACTOR_SPEED ** 2) *
(CARD_AMPL_PARAM if op_act_car[1] == CARD_AMPL else 1)) - CardValue[c.code]  # 我方加速耗血量
                            op_de_lif = (abs(op_bat_dis) ** 2 // FACTOR_DISTANCE ** 2) *
(CARD_AMPL_PARAM if card_use == CARD_AMPL else 1)  # 对方迎球耗血量
                            if ((card_use != None) and (card_use != CARD_AMPL)):  # 如果我方
用了道具，就用它的价值来抵消损耗的生命值
                                my_de_lif -= CardValue[card_use.code]
                            temp_lif_pic_comp = (temp_acc, op_de_lif - my_de_lif)
                            if (temp_lif_pic_comp[1] > lif_pic_comp[1]):
                                lif_pic_comp = temp_lif_pic_comp
                            break
                if (temp_lif_comp[1] > lif_comp[1]):
                    lif_comp = temp_lif_comp
            if (lif_comp[1] > lif_pic_comp[1]):
                best2 = lif_comp
            else:
                best2 = lif_pic_comp
            if (best1[1] > best2[1]):
                best = best1
```

```
            else:
                best = best2
            acc = best[0]
        # 跑位
        run = ((DIM[3] - DIM[2]) // 2) - pos.y
        # ds 数据导入
        ds['now'].append((tb,bat,acc,run,card_side,card_use))
        print((tb.tick, bat, acc, run, card_side, card_use))
        return RacketAction(tb.tick, bat, acc, run, card_side, card_use)

# 对局后保存历史数据函数
# ds 为函数可以利用的存储字典
# 本函数在对局结束后调用，用于双方分析和保存历史数据
def summarize(tick:int, winner:str, reason:str, west:RacketData, east:RacketData, ball:BallData,
ds:dict):
    if (ds == {}):
        ds = {'pre':[] ,'now':[]}
    ds['now'].append((winner,reason,west,east,ball))
    ds['pre'].append(ds['now'])
    ds['now'] = []
    return ds
```

2.3程序限制

1.道具优先级设置与溢出判断的失效、道具使用的浪费

这个优先级的设置并没有发挥到它的作用，即只要包内有了一个道具，我们就会立即把它用掉，道具并不会累积。这样用道具表面上是达到了百分之百的利用率，但实际上却是一种变相的浪费，比如有可能会在对方没有加速的时候浪费了旋转球，在对方移动距离很小的时候用上了变压器。

2. 道具的溢出判断自身的问题

此处只考虑了两个道具的相对价值大小。实际上我们还要考虑到为了捡新道具，我们还要因此作出加速，会有额外的体力消耗，所以应该用新道具价值减去额外损耗，然后再与将要溢出的道具价值进行比较，并且获取不同道具会产生不同的加速，从而损耗不同的生命值。这个地方其实是我们算法的难点之一，因为此处我们需要将道具价值跟加速损耗做差再跟道具价值作比较，所以我们必须保证道具价值跟加速损耗能达到一个合理的统一。一个改进的设想是我们应该在减去加速损耗时乘上一个参数，参数过小会导致我们为了一个价值高的道具损耗过多生命值，得不偿失；过大则会使我们彻底无视掉道具，所以我们要不断地调试来将参数换到一个最优值。很遗憾，这一设想最终没有得到实现。

3. 关于“由于直接将打北角当作一次反弹有风险，故决定先打一次南边，再打向北角”的考虑：

这样做虽然避免了 invalid bounce，但损失也是不可忽略的。其实我们不仅仅可以使球在南边反弹，也可以在北边某个十分接近北角的地方反弹，然后从中选出更优者。打一次

南角 accsou = int(-(y + pos.y) / t - vel.y)；打一次北角，设反弹处的横坐标为 bounce_x，由相似三角形的几何关系有（DIM[1]+abs(bounce_x)）/（DIM[1]+abs(bounce_x)）=(y-pos.y)/afterbounce_y，其中 afterbounce_y 是反弹后球又向南走的距离，在这个地方我们应调整使 afterbounce_y 最小，即球的落点更接近北角，再将它的最小值跟 y-pos.y 相加即球在 y 方向的总路程，打一次北角 accnor = int((y - pos.y+min(afterbounce_y)) / t - vel.y)）。然后在两方案中选出对我方损失最小的即速度改变量最小的一个方案。bestacc = accsou if min(abs(accsou),abs(accnor)) == abs(accsou) else accnor)

4.跑位策略漏洞

没有对“对方不动我方持续由于居中跑位消耗体力”的应对措施。这应当通过对 ds 的分析来实现，遗憾未能完成。

5.代码不够简洁

根据对方跑位前的位置和跑位方向来确定跑位后的新位置，生命值大于 35%与否，代码都是同一段。本可以将这一段代码放入一个判断跑位后位置的函数中，以此来增强代码的可读性，避免出现大段重复的代码。

6.我们未能给出可能解决方案的一些限制

①我方作为发球方时，捡拾道具功能失效

②误差积累导致自杀

3 实验结果

3.1测试数据

实验环境说明：

- 硬件配置： Inter(R) Core(TM) i5-6200U CPU@ 2.30GHz 2.40GHz
- 操作系统：win10 64 位操作系统
- Python 版本：python 3.5

本组采用了左右手互搏以及与提供的傻瓜算法对打的测试方法。将自己的最新算法不断与之前的老算法进行对战，对战时发现自己决策的漏洞，及时查漏补缺，提升算法，考虑细节。查看经过更新过后是否可以在总体上战胜之前的程序。在对战中，我们发现了两个问题：第一，我方打出的球会在 bounce 的临界值处出现反弹三次或零次的 invalid bounce；第二，我方会在血量足以加速的情况下出现零反弹的自杀情况，通过与大神的交流我们发现不止有我们一组出现了这样的自杀情况。并且我们还参加了两次热身赛，第一次热身赛遗憾我们的代码由于存在 bug 没能成功运行；第二次热身赛，结果为：

T_zxcvb
10 / 38
win for {'miss_ball': 7, 'invalid_bounce': 3}
lose for {'miss_ball': 13, 'invalid_bounce': 15}

3.2结果分析:

本算法由于算法复杂度比较低，运行时间在预期之中，主要运行时间用于遍历有效速度从而的得到最优解这一环节。

在经过多次测试和热身赛之后，我们发现了以下几个问题。第一，我方会在生命值符合要求的情况下不反弹直接自杀；第二，我们会因为未处理好 1、2 次打角的临界情况而出现不合规则的 0、3 次打角。第三，也是最匪夷所思的事情就是由于我们从不因为道具而改变线路，所以策略很大一部分没能得到实现。

在赛后，小组分析发现了不捡道具的原因：我们的思路是将两个元组进行比较（元组内存加速度，双方血量差），如果能捡到道具，就将道具的价值加到 lif_pic_comp 这个元组里，再和 lif_comp 进行比较，最后返回最优的 lif_comp。但是由于使用了连等，所以两元组等同于同一个元组（本应该是不同的），所以在改变 lif_pic_comp 时，另一个元组也同时改变，所以下边的比较都是无效的，也就没有得出我们想要的最优解。在改正代码后，小组成员发现当我们为非发球方的时候，能够正常捡道具，反之则不能。目前其他原因仍在探索中。

4 实习过程总结

4.1分工与合作

4.1.1 小组分工：

算法提出与改进：李能韬
代码撰写：覃程、史红毅
程序调试：杨佳琦、贾昊凝
报告撰写及跟进代码更新：顾书纶、彭路赢

4.1.2 合作过程：

- 2017.5.15 小组正式组建。拥有微信群。
- 2017.5.22 第一次组会，确定小组分工。



第一次组会：数算教室

- 2017.5.22 李能韬提出 play 函数的两个算法思路。
- 2017.5.25 道具刷新速率由 11.1 回合刷新一个改为每 1.11 回合刷新一个，应对道具刷新速度改变做出了如下调整：

  现在，有可能遇到道具箱溢出的问题。在原来的算法中加这一条：若道具箱已满，对于【不使用道具】且【击球后会获得道具】的结果，计算【道具收益】时，额外扣除掉最旧的道具被挤出道具箱带来的损失。当然，如果想更简单，可以直接忽略掉这样的结果，即默认：浪费道具总是不可取的。
- 2017.5.27 第二次组会 进一步明确算法思路



第二次组会：四教 502

会议议程：

1.梳理一次最优和两次最优的思路，写出之后分别与傻瓜算法比较，并且彼此比较。

2.讨论对于暴力打角（不加速）的应对措施：

出于消耗对方尽可能多体力的考虑，相信大多数的组都会选择采用打角的策略。如果对方是先移动的一方且采用暴力打角，则保持不动，就可以保证对方比我方有更高　的体力消耗；如果有加速等其他操作，则另作考虑。

3.关于跑位策略的讨论：

因为接球前的移动和接球后的跑位消耗体力值都是与移动距离的平方成正比的，所以跑位如果能够预先接近下一次来球的位置，就可以节省体力值。最好的结果是刚好移动到一半的位置，这样总共消耗的体力值只有傻瓜算法（不跑位，只移动）的 1/4. 但如果跑位方向预判错了，或者跑过头了，反而会消耗更多体力值。这也是算法的难点所在。

在写出一个能够预测对方回球落点的算法之前（也就是两次最优），暂定采用居中跑位，即每次跑至中点。

4.关于发球：

由于我们是居中跑位，所以发球点遍历整条边，分别算出一次碰壁和两次碰壁的情况，假设对方不加速，哪种情况回球最靠近中间就取哪种。

5．关于道具的使用时间：

加减血包没有延后使用的必要；隐身术由于在两次最优写出来之前本身就是居中跑位没有什么悬念用了也跟没用一样，所以也可以立即使用；瞬移术按说是用在最大移动距离时，但考虑到大家应该都是打角，那么从中点到角会有很多次，所以也可以立即使用；变压器应该是在对方最大跑动距离处使用；旋转球在对方给球加速时使用。

简单起见，先设成全部立即使用，在后期完善道具部分的时候再进行改进。

6.关于对方跑位的预测：

为了估计对方跑位和迎球消耗的 LP，需要估计对方的跑位位置。方案是根据对方的位置和跑动方向预测到一些点上。

如果对方迎球位置在 5/8 处以上
　　如果对方向北跑位或不跑，就假设他向北跑位至可跑距离的 1/4 处
　　如果对方向南跑位，就假设他归中
如果对方迎球位置在 3/8 处以下
　　如果对方向南跑位或不跑，就假设他向北跑位至可跑距离的 1/4 处
　　如果对方向北跑位，就假设他归中
如果对方迎球位置在 5/8 到 1/2 之间
　　如果对方向北跑位或不跑，就假设他在 2/3 处
　　如果对方向南跑位，就假设他归中
如果对方迎球位置在中点
　　如果对方向北跑位或不跑，就假设他在 5/8 处
　　如果对方向南跑，就假设他在 3/8 处
如果对方迎球位置在 3/8 到 1/2 之间
　　如果对方向北跑位或不跑，就假设他归中
　　如果对方向南跑位，就假设他在 1/3 处

这种方法建立在经验估计上，可能并没有足够的准确性，但由于判断步骤比较细致，误差会相对小。在后期的调试中，这些位置参数也可以做出调整。但考虑到其他组一定也会想到这种对付预跑位的方式，从而不按常理出牌，所以在两步最优算法会有更好的

应对。

- 2017.5.28 李能韬给出初步的道具价值：瞬移术是一个范围（625-4375），取接近平均的值；加减血包 2000；变压器 1500；旋转球 1200；隐身术 800。
- 2017.5.30 李能韬给出基于两步最优算法的跑位策略，但受限于编程进度，未能在热身赛之前实现。策略是：将我方位置、跑位方向和击球路径传给对方（一步最优），通过对方的回应计算回球的落点。如果我方预定跑位方向与我方下一步迎球方向相同，则跑位到现在位置和预定位置的中点；否则，跑位距离为 1，即做假动作。
- 2017.5.30 第一次热身赛。遗憾的是，代码无法运行。bug 是：位置坐标 pos 设置时只取了 Y 坐标，使用时却引用了 X 坐标和 Y 坐标。
- 2017.5.31 修改热身赛 bug，尚未解决的最大问题是自杀：一是打角存在误差，有时一次都不弹，有时弹三次，造成自杀；另一个是直线击球自杀。
- 2017.6.2 第二次热身赛。
- 2017.6.3 晚上在阴暗无灯的二教楼梯间艰苦卓绝地进行第三次组会。



第三次组会：二教楼梯间

本次组会，小组成员共同 debug，并且一起观看可视化复盘数据，努力解决当下问题：一仍然是自杀；二是在比赛中我方的算法完全呈现出暴力打角的状态，遍历寻找最优解的思路没有得到体现；三是跑位策略仍有不灵活之处，出现对方不必跑位，在某角持续打角，而我方每回合都要先居中再返回，并且打出同样的轨迹，直至体力耗至临界值以下。

问题二最快得到了解决，它是由于范围忘记加 range 的笔误导致的，故确实就是在打角。

问题一是一些误差导致的，误差产生的原因暂时未能分析出来，解决方案有两个：一，对速度进行微调，即弹一次的适当加大 Y 方向速度，弹两次的适当减小；二，对球桌大小进行微调，回合结束后再复原。决定采用第一种方案。

问题三，想法是用分析 ds 判断对方是不是在跑位，如果出现上述情况，则改变击球路径。

- 2017.6.4

将第一代遍历落点，根据每个落点确定四种方案（向南弹一次、向南弹两次、向北弹一

次、向北弹两次），在所有方案里找最优解的思路改进为遍历速度，就能得出轨迹。第二代能胜过第一代，但和傻瓜算法打，还是会出现自杀的问题，并且随着比赛进程，打角误差呈现积累。

与 hotel 组约战，200：0 绝地惨败。

目前最致命的问题就是，自杀问题依然没有解决。

另外，根据道具价值改变轨迹的思路并不能得到体现，因此较难捡到道具。所以就放弃了优化道具部分代码。

遗憾的是，两步最优算法没有来得及实现。

- 2017.6.5

正式比赛，首轮告负，无缘淘汰赛。

4.2经验与教训

本次大作业，我们小组最终的战绩不是很尽如人意。但是胜败乃兵家常事，我们是为学习而来，最重要的是实践的过程、合作的智慧、事后的反思与总结、日后的完善与提高；至于最终成绩，尽力而为便无愧于心，故而倒在其次。每一位小组成员在合作中都尽力完成了自己的职责，大家都不是程序方向的大佬，但能够踏踏实实从头做起，也会为了完成项目废寝忘食，由此看来，我们此次的大作业某种程度上看来也算是成功的，因为通过它，我们把课堂所讲和实际操练结合了起来，对于数据结构与算法有了更深入的认识，获得了学习和提高，这也是大作业的意义所在。

在比赛结束后，我们痛定思痛，对整个合作完成的过程做了一个回顾，又重新分析我们的代码，得出了许多经验与教训。

起初我们的设想还是较为完善的，但是后来，由于各种未预料到的阻碍，后来大量的时间精力被投入在攻克自杀问题和道具捡拾上，使得整体进程受到搁置，最后没能写好本应是我们算法核心的两步最优算法。虽然我们进行了大量的努力，这些问题仍迟迟没能得到解决，所以不管是热身赛、与 hotel 组的约战还是最终的比拼，我们都没能表现出最初设想应有的水平。

寻其原因，一部分是由于我们能力水平有限；另一方面，从合作过程中来看，前期的沟通交流相比后来不是很到位，成员对于分工看得过于绝对，也不利于团队的智慧得到发挥——代码中的问题难以被及时发现，这浪费了一定的时间，使得后来时间紧迫，很多问题来不及解决，一些设想也没能实现。大作业分工都是环环相扣的，负责调试、写报告的同学也必须时刻跟进代码，及时发现不合理的地方。就切身体验来看，等代码堆得很多时再看效率是很低的，通常就是顺着写代码的同学的思路被动地过一遍，很难发现其中不合理的地方，退一步说，即使发现，大段的代码已经写好，要改也并不是一件容易的事，常常牵一发而动全身。所以，在大作业的每一个环节，同学们都要积极参与，投入自己的精力与智慧。

胜利固然可喜，但失败也是人生路上必不可少的一种经验。相信通过这次的实践，我们小组的每个同学都有所收获，有所反思，都能完善自身，超越自我！

4.3建议与设想

组队方面，从防止大腿聚堆的角度来看，以后可以根据学号分组，可以起到随机分组的效果。

基础设施代码方面，希望代码建设小组能够更早开始开发与完善，尽量在大作业之初就有一套比较成熟的基础设施代码。当然，不可避免的会有需要修改的地方，希望老师可以在课程群里以群公告的方式通知每一位同学。

竞赛方面，我们认为分区赛制的公平性不如循环积分赛：首先，和超强算法同组的算法不易出线，而他们不一定比其他组的八强选手差；其次，A 算法输给 B 算法，B 算法输给 C 算法，A 却依然有可能把 C 完虐，因此将所有队放在一起比赛，才能最好地给算法的优劣排序。因此我们建议先大范围做积分赛，积分八强的选手进入淘汰赛。考虑到运行时间长，可以对同学意见进行投票，看看大家是否愿意助教事先运行比赛，课堂公布结果，在课堂上进行淘汰赛的比拼。

实习作业后续，如果时间允许的情况下，改善代码后，几队可以私下开展友谊赛，延续编程精神，培养兴趣，充实生活。

最后，拾人牙慧，送上对来年学弟学妹的祝福与展望：“希望学弟学妹能够不断超越，无畏艰难，奋勇前行，友谊第一，比赛第二，扬创新精神，展大将风范，挽狂澜于既倒，扶大厦之将倾，码实用创新之代码，任信息时代之栋梁！”

5 致谢

在本次大作业中，我们每个人都收获了很多。在此，我们衷心感谢一学期以来陈斌老师的悉心教导和对课程的精心设计；感谢基础代码搭建小组同学的辛苦付出；感谢超强助教天团的默默耕耘和及时反馈；最后感谢每一位小组成员所做出的努力！

祝陈斌老师数据结构与算法课程蒸蒸日上！

6 参考文献

无



PAPA

组长：徐晨雨

组员：董芮江，李晓婷，马俊贤，万长安，魏宇光

小组赛战绩：W3





Papa的聚餐/组会

数据结构与算法课程实习作业报告

徐晨雨*、李晓婷、董芮江、万长安、魏宇光、马俊贤

摘要：

我们的算法主要以价值判断为基础，通过对于下一步操作价值的比较择优进行击球、跑位，并在此基础上对一些细节进行调整。主要用到的数据结构为队列。在组内测试过程中，最终的代码对测试代码保持高胜率，至少是先手必胜、后手有几率胜的结果，在组间约战时，只有对 Oscar 组的测试代码毫无胜算，热身赛中由于提交的代码不够成熟，不具有参考价值。

关键词：

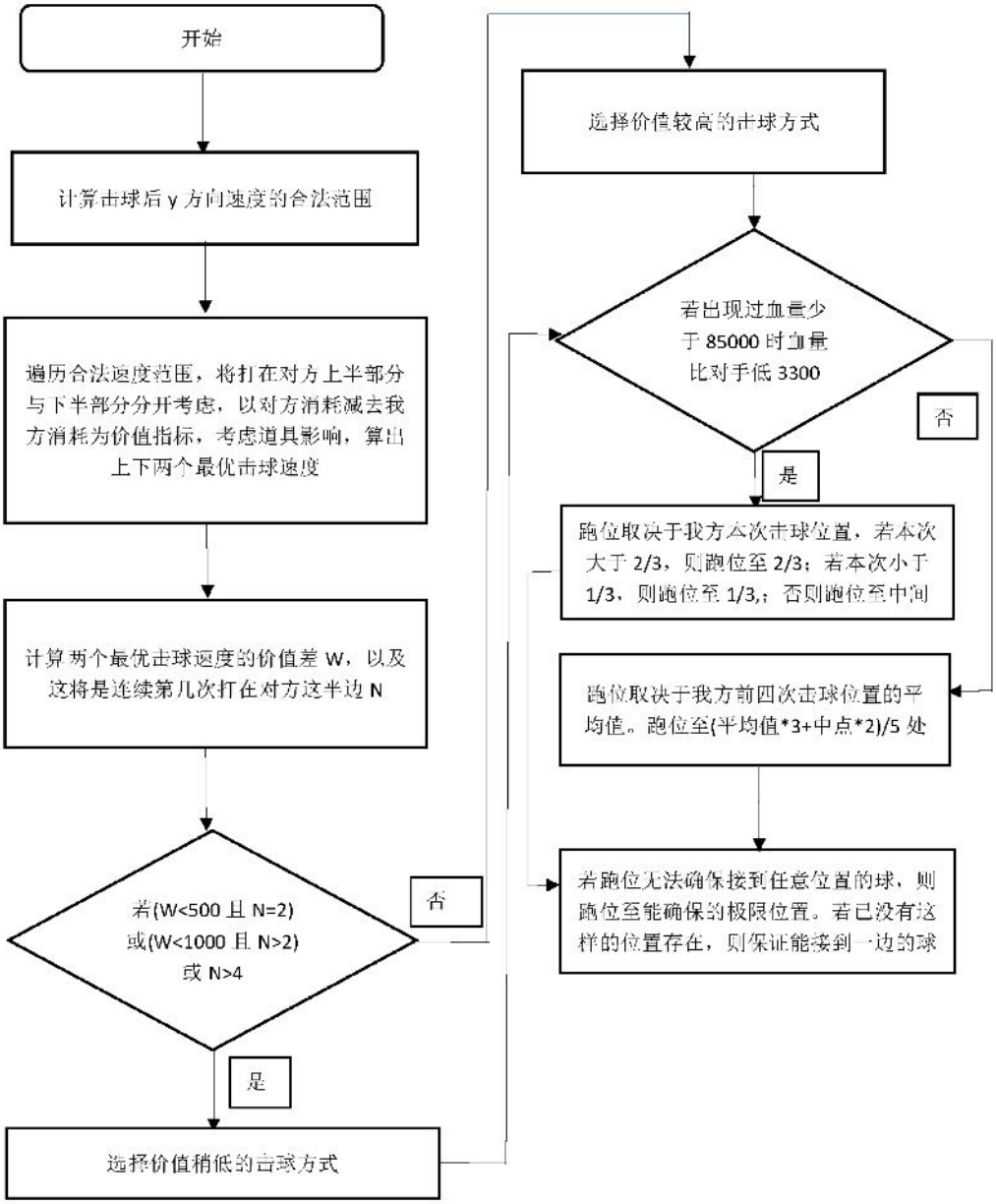
价值判断、队列。

1 算法思想

1.1 总体思路

算法总体分为三个板块。第一板块为计算最优加速度。通过计算我方本回合加速耗费体力与预估敌方耗费体力之差，取其差值最大所对应的加速度为最优加速度，达到一次最优。第二个板块为计算合理跑位值。这一板块分为两大种跑位策略。其一为一次最优跑位距离，通过缓存多次迎球位置，预估下一次迎球位置，得到最优跑位距离。其二为防即死跑位策略。当体力值降到一定值后，会出现迎球半斤小于跑位点和边界的距离，此时跑到预估迎球方向的最远可迎球点。第三板块为计算道具使用。通过人为赋予的道具收益权重，以一定优先级进行道具的使用。

1.2 算法流程图



1.3 算法运行时间复杂度分析

加速度计算复杂度分析

价值计算函数对于给定速度，复杂度为 O(1)。

遍历合法速度，与比较速度对应的价值，复杂度为 O(n)

2 程序代码说明

2.1 数据结构说明

队列：以队列 FIFO 的特点储存我方前四次的迎球位置

2.2 函数说明

2.2.1 发球函数（serve(op_side: str,ds:dict)）

由于可以任意选择发球速度与位置而不需要消耗体力，为增大对方接球消耗，我们选择使速度取到最大值 1666，即为极限弹两次，几乎完整走过三个 DIM[3]。随机选择从底端或顶端选择一个位置发球。

2.2.2 击球函数（play(tb:TableData, ds:dict)）

2.2.2.1 计算加速度函数(calcuBest_vel(tb,ds))

i.计算击球后 y 方向合法速度范围。(getAviliableV(ticks,pos,DIM=DIM))

球的落点应当在([DIM[3],3DIM[3]]∪(-2DIM[3],0)中。

考虑到边界可能出现误差导致 invalid bounce，下界均+1，上界均-1。

ii．选择本轮击球速度。

遍历正值速度与负值速度，调用价值判断函数，得到预估价值。

利用计算球落点函数，算出每一个速度值落点。将落在上半部分与下半部分的速度值分开，分别求出预估价值最大的速度值。

ds['bat'] 用于记录连续几次击打在对方一个半边（此处设为 n），以及是哪个半边。其代码在 play 函数中。记上下两个最优速度值的价值差为 w。若(w<500 且 n=2)或(w<1000 且 n>2)或 n>4，则选择两个击球速度中价值稍低的，否则选择较高的。

iii．价值判断函数(LifeD(tb,ball_velocity))

某一击球速度的价值=对方迎球消耗体力+对方跑位消耗体力（默认回中点）-我方加速消耗体力（均考虑道具使用）

这里若我方有旋转球，且遍历到的速度大于 1600，则将价值加 1500，若选取此速度，这一回合使用旋转球。

2.2.2.2 计算跑位位置函数(bestRun(tb,ds))

若本轮比赛中出现过自身血量低于 85000 且相比对手低 3300,的情况，则改用跑位策略二，否则选择跑位策略一。

i．跑位策略一

使用 ds['run']记录我方前四次击球位置（即对方前四次落点位置），求出其平均值，跑位至(平均值*3+中点坐标*2)/5 的位置。

ii.跑位策略二

直接根据我方本次击球位置决定跑位，若本次大于 2/3，则跑位至 2/3；若本次小于 1/3，则跑位至 1/3,；否则跑位至中间。

iii．防即死跑位策略

若以上跑位无法确保接到任意位置的球，则跑位至能确保接球的极限位置。先估计下一轮我方迎球前体力：体力为本轮迎球前体力减去本轮迎球，击球消耗体力（考虑卡片），若对方有减血包再减去减血包对应体力值，再减去可能出现的最大跑位消耗体力。根据所得体力计算接球范围，跑位至能确保接球的极限位置。

若体力过低，回中点也不能接到边缘球，则跑位至 100000 或 900000 中较近的一个，保证能接到一边的球。

调整跑位：若跑位超过 y 方向桌面长度的一半，则跑位回中点。若本轮我方使用旋转球，则回到中点。

2.2.2.3 卡片使用函数(activeCards(tb,run,vel))

卡片使用分为两种情况，自身体力大于一半则使用条件较严格，低于一半则条件较宽松。

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) i5-5200U　　RAM: 4.00GB
- 操作系统：（名称/版本）Windows 10 家庭中文版
- Python 版本：（版本号）Python3.6、pycharm

测试方法：

- 组内测试：

  每次所写的代码与之前所有可行代码进行对战，观察胜率，以优于 nobrain 系列为底线，同时之前的 T_1、T_2 系列也是重要的测试代码。这是我们最主要的测试方法，最后提交的代码也是以能够完胜大部分测试代码为原则选择出来的。

- 组间约战：

  主要与三支队伍的代码进行了约战，分别是 Quebec 组、Juliet 组以及 Oscar 组，对战 Oscar 组，我们的胜率为零，尽管当时他们用的也不是正式的代码，对其余两组胜率较为可观。

- 热身赛：

  两次的热身赛我们都参加了，第一次提交的代码为 T_nobrain、T_random、T_2，但是，由于 T_2 未及时修改的原因，在热身过程中出现了 bug，第一次热身赛的目的是想要获取其他小组的数据并试图对其进行数据分析，但是由于数据并不太好，所以没办法进行数学上的统计。第二次热身赛提交的代码为 T_800，当时的这个代码还没有添加道具系统，但是效果也还不错，在热身赛中胜率过半，利用这次热身赛的数据，我们详细观察了胜率极高的三组的打法，并针对性地对跑位策略做出了调整。

3.2 结果分析

普通的最优击球方式往往会集中打在对方一角，比较成熟的最优击球算法也倾向于打在上一次打到的某一角。我们的两种跑位策略均与前一次或前几次对方落点位置挂钩，大大降低了跑位造成的体力损失。这一策略所需时间极短，可以忽略。

在后期热身赛时期，我们发现多数对手的跑位策略倾向于跑位总长度的三分之一左右。我们的击球策略在我方连续击在对方一角时有转而击打另一角的倾向，这可以使对方不得不远距离跑位消耗较多体力。这一策略实际效果一般，比起每轮固定打另一边对角，可能未必有优势。

我们写了两套跑位系统，在己方占劣势（血量比对方少 3300）时换用另一套。这一策略可能并没有体现出效果，因为这个差距已经不能在后期被追回。如果改成负两场后换另一套体系，可能效果会好一些。这一策略的时间开销可以忽略。

主要的运行时间开销发生在对可行速度进行遍历的过程当中。

4 实习过程总结

4.1 分工与合作

在小组分工方面，原则上是每个人都参与到代码的建设当中，这也是第一次组会实际的情况，之后关于代码的具体编写，则由万长安、董芮江、魏宇光和马俊贤完成，数据测试由万长安、董芮江、徐晨雨完成，实习报告由李晓婷、董芮江和徐晨雨完成，这是大致的分工。

在交流的方式方面，前期主要是进行组会，所有的人都参与到讨论当中，后期则主要是在宿舍进行交流，算是一种小规模组会的形式。

前期组会共有两次，这两次小组全部成员都参与其中。

第一次组会：我们分析这次旋转乒乓球的平台代码，推导了在编写代码的过程中会用到的一些有关跑位、乒乓球落点、碰壁次数的基本的数学公式。除此之外，提出了最初的跑位思路：基于 $\{a^2+b^2 \leq (a+b)^2\}$ 这个数学公式，提出每次都把移动分成两段以从总体上减少移动的体力消耗。在讨论的过程中，马俊贤同学针对这个游戏消耗体力的特点，提出了如何应对打对角的问题，这也成为之后几天代码改进的重要标准。同时，决定写基础的代码，共两个，一为 T-random，在速度允许的范围内随机选择速度击球，跑位策略为在一个范围内随机选择，二为 T-nobrain，打对角，每次跑回球桌中点，这部分由万长安同学写成。另外，在第一次组会结束之后，董芮江写出了 T_1、T_2，这也是之后正式算法的基础。同时，魏宇光也写出了自己的代码，对于正式代码的改进有很大的帮助。

第二次组会：这次组会仅仅在第一次组会后的第三天，也是因为代码的初步搭建并不需要太长的时间。这次组会是在艺园举行的。周一道具系统上线了，我们初步讨论了关于道具的使用。我们的思路是首先对最优路径进行放宽，如果在可以接受的代价内能够捡到道具，那么这一路径就纳入实践路径的考虑范围内。我们给每一种道具根据它的杀伤力大小赋予了不同的价值，把价值纳入决策路径的计算中，每次取最优方案。对于道具的使用，因为对方的位置是不可知的，留着也很难判断究竟在什么时刻释放什么道具是最优的，因此我们讨论了很久之后决定采取即捡到即使用的原则。这次我们提出了新的跑位策略，即根据游戏进行的不同阶段分析出乒乓球落点的期望数据，然后根据期望值来进行跑位。



第二次组会照片

之后则是后期的小规模讨论：

5 月 24 日，徐晨雨提出建议，试图通过 FFT 与小波分析的手段，对历史数据进行分析，以精确分析对手的打法。起初，我们觉得这个方案是很有效的，只是可行性方面，由于自身水平的限制，再加上通过热身赛获得的数据不好，以及这个游戏本身随机性强、数据量少的特点，导致我们的这一“核心科技”完全失效，最终我们也放弃了这个想法。

5 月 25 日，魏宇光同学写出了一个算法，该算法可以估算对手下一步将会打哪个对角，从而做到可以减少跑位消耗，这个算法给我们之前的代码造成了很大的麻烦。

5 月 27 日晚，万长安同学在对代码进行修改的过程中实现了重大突破，开发出 T-800 系列代码，终于能够完胜练习代码 T-nobrain 系列以及之前的 T-1、T-2 系列。

第二轮热身赛结束后，万长安根据复盘的信息，重新考虑了击球的方式，分别在自己体力值的不同范围内选取不同的对角进行击打。另外，还有一个重要的问题，就是当自己的体力较小时，应该如何调整策略，这一部分也是由万长安完成的。

考虑到对于历史的使用，徐晨雨提出可以将多个代码集成到提交的代码当中，并根据对历史的分析选择相应的代码使用，但这个想法最终没能实现。之后，在进一步的改进中，我们给算法加上了道具模块、跑位模块等等，逐步完善了代码。

6 月 4 日，万长安和董芮江修改了跑位算法，将每次跑位跑中间的简易跑位策略与跑到三分之一处相结合，这样的策略起到了很好的效果，可以有效地应对之前我们的陪练算法。当天晚上，参赛代码的提交截止，尽管我们在不断地修改代码，但是最终还是选择了改进后的 T_800。代码的编写就此结束。

4.2 经验与教训

由于我们提交的代码在组内测试中可以完胜其他所有测试代码，所以这个代码的一个优势在于面对较弱代码时可以保持极高的胜率，这也是比赛时所呈现出来的。

这个代码的缺陷在于，进行价值判断的过程中，对于所有可行的速度区间进行了遍历，但是根据这个游戏的特点，这样的遍历有时会出现直接向对方中点击打的现象，这是我们不愿意看到的，这也是我们在最后一天测试代码是所发现的问题，但是限于时间因素，未能对此进行更改，这是一个遗憾。同时，在进行代码编写的过程中，过于依赖之前代码的框架，导致最后的这个缺陷，这在以后的工作当中也是需要极力避免的。

另外的一个缺陷在于我们面对这个比赛的策略，我们的思路只是赢下越多越好，并未考虑平局的情况，所以代码中完全没有出现尽可能地节省体力值这个目标，这也是最后失败的一大原因。

4.3 建议与设想

无论是在什么样的代码中，我们都可以看到更多地是在打对角或者对角附近，这样的情况在现实中出现是不太正常的，也可能是因为参数的问题，所以希望下一次的作业能够使得应对策略更加多样化。

有关最后的比赛过程，建议在可视化的时候严格按照比赛规则进行可视化，希望下次不会出现我们组这样胜场数排名分区第一却被淘汰的队伍。

通过对这个游戏的分析可以发现，这是一个非线性系统，排除随机性的影响，这种非线性是由于价值判断的过程而产生的，这样的情况在 T_calcu1 中表现得极为明显，故这种分析和模拟的方式可能会对其他涉及价值判断的领域有所启发，抑或其他领域当中已经成熟的理论可能会对这个游戏 AI 的开发有指导意义。

5 致谢

在完成这个实习作业的过程中，感谢陈斌老师以及助教老师们对于对战平台的搭建，另外，在组间约战的过程中，Juliet、Quebec、Oscar 组也对我们项目的进展有很大的帮助，尤其是在观察了 Oscar 组的打法之后，我们修改了自己的跑位算法，在此应当感谢。

6 参考文献

http://gis4g.pku.edu.cn/course/pythonds/



LIMA

组长：陶天阳

组员：刘启航，沈泽盛，冯曦林，黄佳磊，闫丽影

小组赛战绩：W4

数据结构与算法课程实习作业报告

季军败倒在我们的手下，亚军来找我们讨教经验，我们不是冠军，我们只是深藏功与名的迷之 LIMA！

陶天阳*，黄佳磊，沈泽盛，刘启航，冯曦林，闫丽影

摘要：

乒乓球大赛为不完全信息决策游戏，我们以设计战术的方式，结合博弈论与计算权重的思想设计了我们的算法。经过细致解读乒乓球大赛的原始代码并研究其规则，基于应用字典等多种数据结构，我们设计了此次比赛的算法并经过完善先后开发四代战术代码。LIMA3 为决赛对战函数，LIMA4 为我们经过完善的最终版本。LIMA4 由对方行为预测函数，与我方跑位与击球应对策略函数，道具决策函数，ds 分析函数组成。其中 ds 历史数据分析对方击球跑位道具濒死策略作为我方决策辅助工具，主体为人为设计的战术函数。总的体系是预测函数根据历史数据作出预测，决策函数根据预测函数进行决策。（原先的体系为先设计一步最优战术，再将一步优先战术视为假想敌设计二步最优战术。）

关键字：

乒乓球大战 历史数据 预测 收益分析 道具权重

1 算法思想

1.1 总体思路

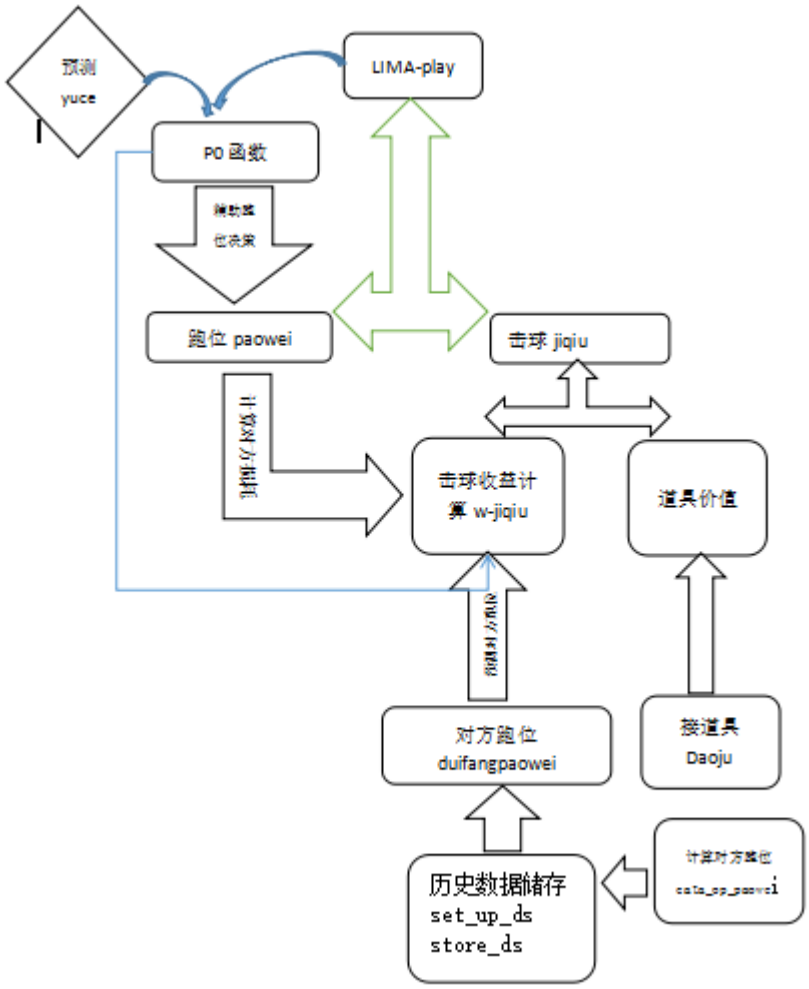
LIMA 算法并未涉及过多的数据结构，而主要采取推理—判断—决策的方式指挥球拍的行动。LIMA3 中包括了历史数据分析部分为决策的智能性提供了依托。算法总体由对方行为预测函数，己方行动收益计算函数，己方行动函数，道具权重函数组成。

我们的起始想法比较简单，即仅考虑收益最大化，不考虑对方的行动，但是经过算法逐步完善与实战经验的日渐丰富，我们开始考虑预测对方的行为来辅助我们的决策。站在对方的角度来想跑位和击球战术，我们假设对手都是最聪明的决策，即把他最应该采取的方式作为我们的预测的对方行动。接着我们把预测的对方行为视为有效信息补充进起始的不完全信息决策中，开始进行完全信息的决策。此时再设计收益估值函数，兼顾道具的权重收益，进而决定跑位和击球方式。

但是经过多次失败，我们发现，对手并不都是会非常机智的采取我们帮他们预测的方式去行动，比如有的组会跑位时固执的回到中间位置。这样我们的算法的亮点就失去了光彩。所作的最优决策也变得平常甚至亏损。因此，我们吸取经验教训设计了历史数据的记录函数，并在对战中参考历史数据，找到对方行动的规律再进行决策。

LIMA4 主要为预测-决策模式，不存在深度递归，不存在大量循环。一个算法中的语句执行次数称为语句频度或时间频度。记为 T（n）。算法的基本操作重复执行的次数是模块 n 的某一个函数 f（n），因此，算法的时间复杂度记做：T（n）=0（f（n））。每次调用程序时的计算量几乎是一个比较大的常数，只在道具卡出现时，击球部分会多出运算。因此，总体的时间复杂度为 0（n），n 为回合数。

算法流程图



2 程序代码说明

2.1 数据结构说明

应用了字典存储列表的能力来存储历史数据

2.2 函数说明

首先啊介绍主体函数 play，在 play 中用到的其他函数暂时仅说明其达到的效果，而具体实现方式将分列各个函数，逐一说明。

```
def play(tb, ds):
    if tb.side['life'] == 100000:
        set_up_ds(ds, tb)
        ds[tb.op_side['name']]['dui'] = 0
        ds[tb.op_side['name']]['cuo'] = 0
        ds[tb.op_side['name']]['o'] = 1
```

每次血量满时视为开局进行初始化

```
if abs(tb.ball['position'].y - 5e5) >= 4.5e5:# 距离中点大于4.5e5时，要判断在哪边
    oo = tb.ball['position'].y // 5e5
    if oo == 2:
        oo = 1
    if oo == ds[tb.op_side['name']]['o']:
        ds[tb.op_side['name']]['dui'] += 1
    else:
        ds[tb.op_side['name']]['cuo'] += 1
ds[tb.op_side['name']]['o'] = yuce(tb.ball['velocity'].y, tb.ball['velocity'].y
```

当球在某个角时先判断在哪一边的角，oo 的作用就是用来判断球在哪个角。oo 为 2 在 A 侧，否则在 B 侧，
ds[tb.op_side['name']]['o'] 用来储存对方接球球在的一侧。oo 为 2 和 oo 为一表示的都是 A 侧，oo 为 0 表示 B 侧，判断 oo 是否 ds[tb.op_side['name']]['o'] 相等，如果相等我方接球位置和对方发球位置在一侧则 dui 加一，否则 cuo 加一，这一部分在 p0 函数中用到。

```
x = tb.ball['position'].y - tb.side['position'].y
# xx:迎球阶段最短移动距离
xx = 0 if abs(x) <= 2000 else x - 2000 * x // abs(x)
active_card = tb.op_side['active_card']
byq = 2 if active_card[1] == 'AM' else 1
life = tb.side['life']
life -= (xx ** 2 // 4e8) * byq
```

这一部分计算到接球位置后血量及判断可用的道具卡是否为变压器。

然后调用 jiqiu 函数和 paowei 函数，做出我们的应对。计算我方击球损血量后再调用 store-ds 函数把这一回合内的数据都记录下来。代码如下。

```
vy = jiqiu(tb.ball['position'].x, tb.ball['position'].y, tb.ball['velocity'].y, tb.op_side['posit
          tb.op_side['run_vector'], tb.cards['cards'], life, byq, tb, ds)
# vy为我方出球时球y方向的速度
life -= (vy - tb.ball['velocity'].y) ** 2 // 400
pw = paowei(tb.ball['position'].y, vy, life, tb, ds) - tb.side['position'].y - xx
# paowei为本回合我方跑位的位移
store_ds(tb, ds, vy, pw)
# 调用记录数据的函数
return RacketAction(None, xx, (vy - tb.ball['velocity'].y) * (2 if active_card[1] == 'SP' else 1),
                    tb.side['cards'][0] if tb.side['cards'] else None)
```

以上就是我们的 play 主体。其中应用到的 jiqiu（x1, y1, v0, y0, d, cards, life, byq, tb, ds）函数和 paowei（y0, v0, life, tb, ds）在下面说明。以及在这两个函数中应用的其他函数将会一一解释。

以下为各个部分的代码说明。

(1)历史数据储存

store_ds(TableData,BigData,vy,paowei)

set_up_ds(BigData,tb)这里仅截取了部分代码

```
def set_up_ds(BigData, tb):  # 初始化datastore为一个字典，记录某一回合的历史数
    BigData[tb.op_side['name']] = {}
    ds = BigData[tb.op_side['name']]
    # 创建新的字典
    ds['HuiHe'] = 0
    # 表示回合数
    ds['Flag'] = False
    # 表示是否开始记录数据
    ds['JiQiu'] = []
    # 我方击球到达位置(y)
    ds['PaoWei'] = []
    # 我方跑位位置(y)
    ds['op-PaoWei'] = []
    # 对方跑位位置(y)
    ds['op-JiQiu'] = []
```

以字典的形式储存某一回合我方击球结束之前的历史数据，以要记录的项为 keys，以各个回合历史数据组成的列表为 values。储存的历史数据有我方击球到达位置，我方跑位位置，对方击球到达位置，对方生命值，对方击球后球的 y 方向速度，我方击球后球 y 方向的速度,对方道具卡，我方道具卡，对方道具使用情况，我方道具卡。每场比赛开始时开始记录。这些数据在后续的决策中都会发挥作用。

在此部分应用到 cata_op_paowei(y1,y2,l1,l2,v1,v2,run)函数来计算上一回合的对方跑位位置,返回值即为对方上一回合跑位的位置。

```
if n >= 1:
    if abs(ds['op-JiQiu'][n - 1] - ds['op-Ballv'][n - 1] * 1800) >= 500000:
        k = -1
    else:
        k = 1
    PaoWei1 = cata_op_paowei(ds['JiQiu'][n - 1], ds['JiQiu'][n],
                             ds['op-Life'][n - 1],
                             ds["op-Life"][n],
                             ds['Ballv'][n - 1], k * ds['op-Ballv'][n],
                             TableData.op_side['run_vector'])
    # 计算对方上一回合跑位
    ds['op-PaoWei'].append(PaoWei1)
    # 添加对方上一回合的跑位位置y
```

cata_op_paowei(y1,y2,l1,l2,v1,v2,run)部分代码如下，本函数旨在算上一回合对方跑位的位置 y,v1 为上一次我方击球后球的 vy,v2 为这一次对方击球后球的 vy。

```
def cata_op_paowei(y1, y2, l1, l2, v1, v2, run):
    # 本函数旨在算上一回合对方跑位的位置y,v1为上一次我方击球后球的vy,v2为这一次
    b = -2 * (y1 + y2)
    a = 2
    c = y1 ** 2 + y2 ** 2 + ((v2 - v1) * 1000) ** 2 + (20000 ** 2) * (l2 - l1)
    if (b ** 2 - 4 * a * c) >= 0:
        pos1 = (-b + (b ** 2 - 4 * a * c) ** 0.5) / (2 * a)
        pos2 = (-b - (b ** 2 + 4 * a * c) ** 0.5) / (2 * a)
    else:
        return 0
    if run == None:
        a = 1
    else:
        a = run
    if a * (pos1 - y1) >= 0:
        pos = pos1
    else:
        pos = pos2
    return int(pos)
```

计算的方法就是通过已知的对方两回合的血量差来已知对方上一回合总耗血，通过己方的上一回合击球速度方向位置，对方上上个回合的跑位（在前一回合算出）所以对方之前的所有行动都是可知的，于是计算出了上一回合对方跑位存入历史数据。

(3)我方跑位函数(需要使用 yuce，和 p0 在此一并解释)paowei(y0,v0,life,tb,ds)

目的:在一次迎球结束后跑到一个新的位置，使下一次迎球的消耗减少，利益最大化。最完美的跑位是跑位位置恰好是下一次击球位置的中点，但是由于本游戏是不完全信息决策，这种理想状态是达不到的。因此我们设计战术来尽量接近这一目标。

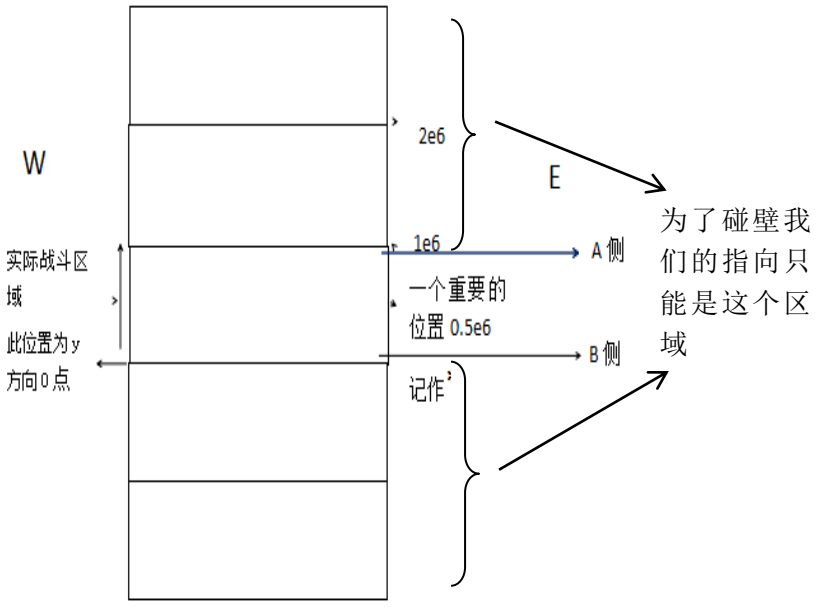
参数:我方击球时的位置，击球时的速度，我方击球后的血量，历史数据

返回值：击球结束后下一步跑位的具体位置。

PS:其中 zh 是把广义坐标转化为狭义坐标的函数，最初由沈泽盛分类讨论编写，后由组长一行秒杀：return abs((y+1e6)%2e6-1e6)

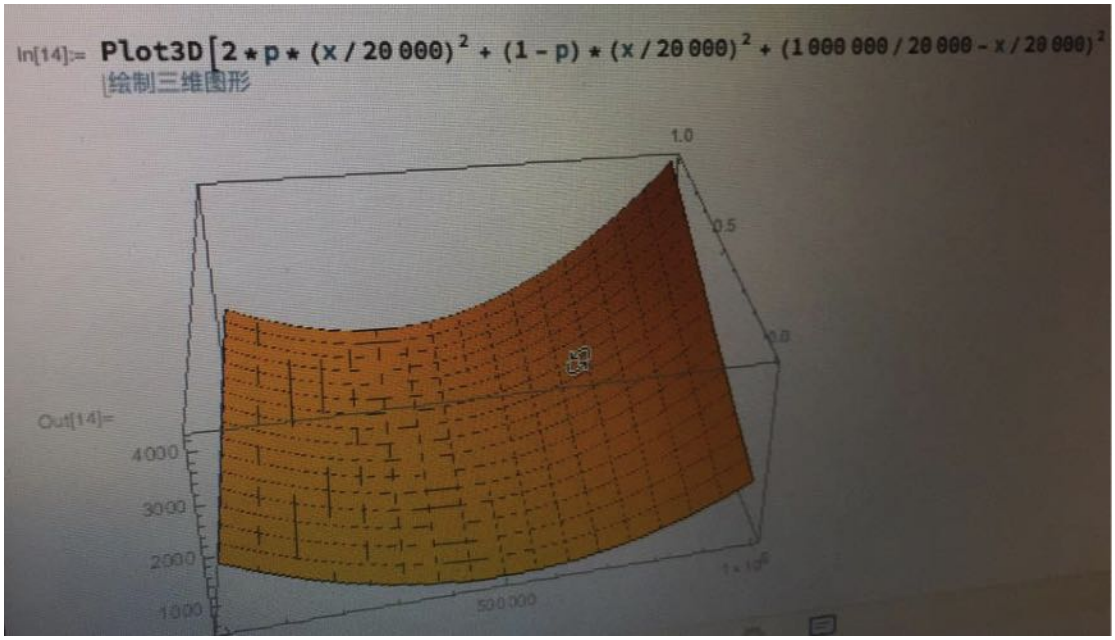
```
def paowei(y0, v0, life, tb, ds):  # 我方跑位函数
    yt = y0 + v0 * 1800  # 我方指向
    yl = zh(yt)  # 对方接球位置
    yi = yl + v0 * 1800  # 对方指向
    if yt >= 3.1e6 or yt <= -2.1e6:
        return 5e5
    p = p0(yl, v0, tb, ds)
    y = y0 / 2 + 5e5 * (1 - p)
    lm = life // 100 * 1800
    if abs(y - 5e5) > abs(lm - 5e5):
        y = int(5e5 + (y - 5e5) * abs((lm - 5e5) / (y - 5e5)))
    return y
```

可以结合以下图来理解这段程序



这里用到 p0 函数代码如下，其作用是返回一个概率值来表示我方接球时与对方发球打角时在同侧的概率。则计算 y = y0 / 2 + 5e5 * (1 - p)，将双方在一侧的概率考虑进去。在濒死是 lm 表示我们运动的最大半径，当理想的 y 到中点

的距离大于 1m 到中点的距离，我们将不得不重新选择 y 值。其中 y = y0 / 2 + 5e5 * (1 - p)是通过带参量的二次函数求极值得出的，以下为用 mathematica 画出的损血随 p 与 y 变化的函数图，上式为求得最小值时的曲线




```
def p0(y1, v0, tb, ds):
    # y1:对方击球位置
    # v0:对方接球速度
    if ds[tb.op_side['name']]['dui'] + ds[tb.op_side['name']]['cuo'] == 0:
        return 1 - yuce(y1, v0)
    else:
        p = ds[tb.op_side['name']]['dui'] / (ds[tb.op_side['name']]['dui']
                                             + ds[tb.op_side['name']]['cuo'])
        if yuce(y1, v0) == 0:
            return p
        else:
            return 1 - p
```

（4）击球收益估值函数 w_jiqiu(y1, v0, v1,life, tb, ds)

这是我们代码比较核心的部位，对于击球收益的估值在我们的击球决策中起至关重要的作用。击球受益=-我方击球损耗 w1+对方迎球损耗 w2-对方击球受益 w3。所以这部分以计算为主，分为三个部分。

1）计算 w1 = (v1 - v0) ** 2 / 400 ，即为我方接球改变速度所耗体力

2）计算 w2

```
yt = y1 + v1 * 1800
if yt >= 3e6 or yt <= -2e6 or (yt > 0 and yt < 1e6):  # 不合法
    return -10000000
y2 = zh(yt)
y3 = paowei(y1, v1, life, tb, ds)# y2为对方接球位置，y3为我方预测跑位位置
yy = duifangpaowei(tb, ds)
w2 = (y2 - yy) ** 2 / 4e8
```

yt 我方指向的位置，不能出界(这里的界限指的是报告之初的五个桌面拼成的大桌面）如果出界我告负。y2 为对方接球时的位置，y3 将用在 w3 的计算中。yy 是我们预测的对方跑位用到了一个新的函数 duifangpaowei（tb，ds）返回值是我们对对方跑位进行的预测。duifangpaowei（tb，ds）也在这一部分解释，如下是其代码

```
def duifangpaowei(tb, ds):
    if ds[tb.op_side['name']]['HuiHe'] > 3:
        return (ds[tb.op_side['name']]['op-PaoWei'][-1] +
                ds[tb.op_side['name']]['op-PaoWei'][-2] +
                ds[tb.op_side['name']]['op-PaoWei'][-3]) // 3
    else:
        return 5e5
```

当历史数据储量大于三时，即可取最近记录的对方三次跑位平均值来作为预测对方的跑位，数据不足时假设对方归中。

3）计算 w3，w3 的计算比较复杂，因为是计算对方的击球收益，我们对对方的行为只能预测。代码如下

```
w3 = 0
p = p0(y2, v1, tb, ds)
list1 = [0, 0, 1 - p, p, 0, 0]
for i in range(6):
    yi = y2 + v1 * 1800  # 对方采用idiot打法所打到的广义坐标
    ytt = 3e6 - i * (10 ** 6)  # 对方要打到的位置
    y4 = zh(ytt)  # 预测的我方再次迎球位置
    v2 = (ytt - y2) / 1800  # 预测的对方击球速度
    u1 = (ytt - yi) ** 2 / (1800 ** 2 * 400)  # 对方击球损耗
    u2 = (y1 - y3) ** 2 / 4e8  # 我方跑位损耗体力
    u3 = (y3 - y4) ** 2 / 4e8  # 我方迎球损耗体力
    # 接下来算我方再次击球损耗u4
    yi = y3 + 1800 * v2  # 算出我方再次击球采用idiot方法的坐标，而后选择较近的角打
    if yi >= 3e6:
        u4 = (yi - 3e6) ** 2 / (1800 * 2 * 400)
    elif yi <= -2e6:
        u4 = (yi + 2e6) ** 2 / (1800 ** 2 * 400)
    else:
        a = abs(yi)
        b = a % 1e6
        if b >= 5e5:
            u4 = (1e6 - b) ** 2 / (1800 ** 2 * 400)
        else:
            u4 = b ** 2 / (1800 ** 2 * 400)
    # u4 = 0
    w3 = w3 + (-u1 + u2 + u3 + u4) * list1[i]
    # 将w3按对方击各个角的概率加权求和
```

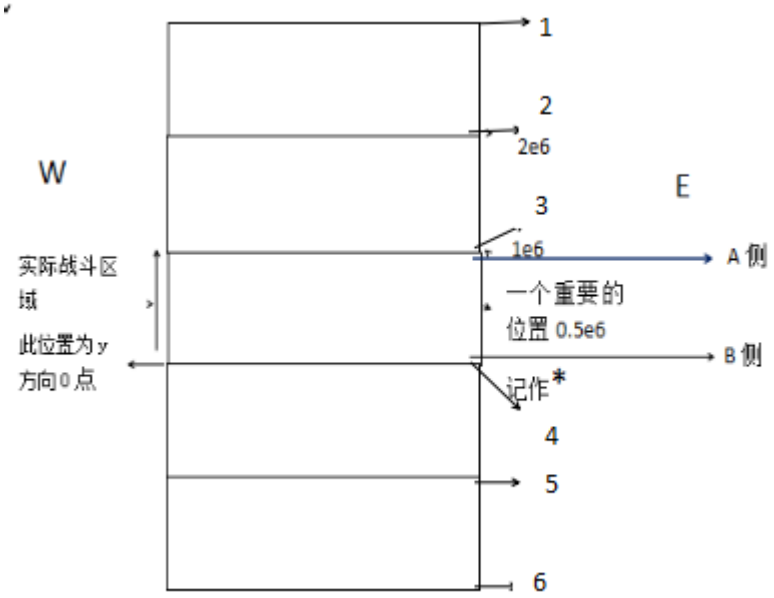
由于混沌原理，我们在预测对方跑位的基础上演变了两步，但我们的预测不一定靠谱，如果出了一点偏差将导致后面演变出更大的偏差，因此再往后预测将失去意义，而且对总体决策影响也不大了，所以我们直接假定为 idiot

（5）击球函数 jiqiu(x1, y1, v0, y0, d, cards, life, byq, tb, ds)

击球函数部分将用到估值的我方收益函数，同时考虑道具权重来进行击球决策。力求达到利益最大化的击球效果。以下为代码分步解读。

```
lw = []
lv = [(3e6 - y1 - 1) // 1800, (2e6 - y1 - 1) // 1800,
      (1e6 - y1 - 1) // 1800 + 1, -y1 // 1800,
      (-1e6 - y1) // 1800 + 1, (-2e6 - y1) // 1800 + 1]
```

lv 是六种可行打角的击球速度，分别指向 1,2,3,4,5,6，六个角，其中的+1-1 很重要，有一点差错将导致碰壁次数不对




```
for i in lv:
    lw.append(w_jiqiu(y1, v0, i, y0, d, life -
                      ((i - v0) ** 2 // 400) * byq, tb, ds))
for i in cards:
    l = daoju(x1, y1, i.pos.x, i.pos.y)
    lv += l
    for j in l:
        lw.append(w_jiqiu(y1, v0, j, y0, d,
                          life - ((j - v0) ** 2 // 400) * byq, tb, ds)
                  + dj[i.code])
return lv[lw.index(max(lw))]
```

第一个 for 循环把六个速度对应的收益算出，加入列表 lw 中，此时不接道具；第二个 for 循环是对出现道具的循环。一个道具出现时，有五种速度改变方式去击球，将这些速度加入到 lv 中，将五个新的收益数据与这个道具权重收益求和加入到 lw 中。然后计算下一个道具的对应情况。这里用到 daoju 函数用来计算为了得到道具可以采取的速度方案。

```
def daoju(x0, y0, x, y):
    k = 1000 / abs(x - x0)
    return [int(k * (y - y0)), int(k * (200 + y - y0)),
            int(k * (200 - y - y0)), int(k * (-y - y0)),
            int(k * (y - 200 - y0))]
```

lw 中的最大值对应位置数值在 lv 中对应的速度就是最优的速度解。

2.3 程序限制

如果游戏开始后不久，如果出现加血道具，可能会使血量重新满，使历史数据们重新初始化将会给预测带来麻烦甚至会报错。

3 实验结果

3.1 测试数据

实验环境说明：

- 硬件配置：4g
- 操作系统：Windows10
- Python 版本：Python3.5

我们没有专门的陪练算法或是测试程序，我们每次都是用之前的版本测试，因为最终版本 LIMA4 没有打过之前的 LIMA3，所以最终交的是 LIMA3（但 LIMA4 考虑更周全，算法更缜密，本文主要分析 LIMA4 的算法），附上每次测试代码都要用到的简短的 LIMA1:

```
from table import RacketAction

def serve(op_side,ds):
    return 500000,833#中位发球，碰2次发到角

def play(tb,ds):#tb是tabledata类型，ds是datastore类型
    #x:迎球阶段距离差
    x = tb.ball['position'].y-tb.side['position'].y
    #xx:迎球阶段最短移动距离
    xx = 0 if abs(x)<2000 else x-2000*x//abs(x)
    #y:无墙时碰到的位置
    y = tb.ball['position'].y+tb.ball['velocity'].y*1800
    #y1:y下面第一个边界(包括y)
    y1 = 1000000*(y//1000000)
    #y2:y上面第一个边界(不包括y)
    y2 = y1+1000000

    if y>3000000:
        v=(3000000-tb.ball['position'].y)//1800-1
    elif y<-2000000:
        v=(-2000000-tb.ball['position'].y)//1800+1
    elif y>=0 and y<1000000:
        v = (1000000-tb.ball['position'].y)//1800+1 if (1000000-y)<y else -tb.ball['position'].y//1800-1
    elif tb.op_side['run_vector'] and (500000-tb.op_side['position'].y)//tb.op_side['run_vector']>0 or not tb.op_side['run_vector']:
        v = (y1-tb.ball['position'].y)//1800+1 if y-y1<y2-y else (y2-tb.ball['position'].y)//1800-1
    else:
        v = (y1-tb.ball['position'].y)//1800+1 if tb.op_side['run_vector']==1 else (y2-tb.ball['position'].y)//1800-1

    return RacketAction(None, xx, v-tb.ball['velocity'].y, 500000-tb.side['position'].y-xx, None, None)

def summarize(tick, winner, reason, west, east, ball, ds):
    return
```

LIMA1 每次跑位回中，击球打到自己损血最少的角，并且无 bug，可以和被测试算法打很久，这样观看复盘数据时可以看到长时间的行为（LIMA2 更复杂且能胜 LIMA1 但不时会有 bug，所以后来不用做测试）

以下是第一次热身赛结果：（我们是 BZYNS）

第 0 个 [E.B]T_BZYNS-VS-T_AUGST-000
第 1 个 [E.B]T_BZYNS-VS-T_AUGST-001
第 2 个 [E.B]T_BZYNS-VS-T_AUGST-002
第 3 个 [E.B]T_BZYNS-VS-T_AUGST-003
第 4 个 [E.B]T_BZYNS-VS-T_AUGST-004
第 5 个 [E.B]T_BZYNS-VS-T_FANCY-000
第 6 个 [E.B]T_BZYNS-VS-T_FANCY-001
第 7 个 [E.B]T_BZYNS-VS-T_FANCY-002
第 8 个 [E.B]T_BZYNS-VS-T_FANCY-003
第 9 个 [E.B]T_BZYNS-VS-T_FANCY-004
第 10 个 [E.B]T_BZYNS-VS-T_FMQED-000
第 11 个 [E.B]T_BZYNS-VS-T_FMQED-001
第 12 个 [E.B]T_BZYNS-VS-T_FMQED-002
第 13 个 [E.B]T_BZYNS-VS-T_FMQED-003
第 14 个 [E.B]T_BZYNS-VS-T_FMQED-004
第 15 个 [E.B]T_BZYNS-VS-T_Hades-000
第 16 个 [E.B]T_BZYNS-VS-T_Hades-001
第 17 个 [E.B]T_BZYNS-VS-T_Hades-002
第 18 个 [E.B]T_BZYNS-VS-T_Hades-003
第 19 个 [E.B]T_BZYNS-VS-T_Hades-004
第 20 个 [E.B]T_BZYNS-VS-T_KILO-000
第 21 个 [E.B]T_BZYNS-VS-T_KILO-001
第 22 个 [E.B]T_BZYNS-VS-T_KILO-002
第 23 个 [E.B]T_BZYNS-VS-T_KILO-003
第 24 个 [E.B]T_BZYNS-VS-T_KILO-004
第 25 个 [E.B]T_BZYNS-VS-T_miaou-000
第 26 个 [E.B]T_BZYNS-VS-T_miaou-001
第 27 个 [E.B]T_BZYNS-VS-T_miaou-002
第 28 个 [E.B]T_BZYNS-VS-T_miaou-003
第 29 个 [E.B]T_BZYNS-VS-T_miaou-004
第 30 个 [E.B]T_BZYNS-VS-T_nobrain-000
第 31 个 [E.B]T_BZYNS-VS-T_nobrain-001
第 32 个 [E.B]T_BZYNS-VS-T_nobrain-002
第 33 个 [E.B]T_BZYNS-VS-T_nobrain-003
第 34 个 [E.B]T_BZYNS-VS-T_nobrain-004
第 35 个 [E.B]T_BZYNS-VS-T_November-000
第 36 个 [E.B]T_BZYNS-VS-T_November-001
第 37 个 [E.B]T_BZYNS-VS-T_November-002
第 38 个 [E.B]T_BZYNS-VS-T_November-003
第 39 个 [E.B]T_BZYNS-VS-T_November-004
第 40 个 [E.B]T_BZYNS-VS-T_PMNDS-000
第 41 个 [E.B]T_BZYNS-VS-T_PMNDS-001
第 42 个 [E.B]T_BZYNS-VS-T_PMNDS-002
第 43 个 [E.B]T_BZYNS-VS-T_PMNDS-003
第 44 个 [E.B]T_BZYNS-VS-T_PMNDS-004
第 45 个 [E.B]T_BZYNS-VS-T_random-000
第 46 个 [E.B]T_BZYNS-VS-T_random-001
第 47 个 [E.B]T_BZYNS-VS-T_random-002
第 48 个 [E.B]T_BZYNS-VS-T_random-003
第 49 个 [E.B]T_BZYNS-VS-T_random-004
第 50 个 [E.B]T_BZYNS-VS-T_SCRTS-000
第 51 个 [E.B]T_BZYNS-VS-T_SCRTS-001
第 52 个 [E.B]T_BZYNS-VS-T_SCRTS-002
第 53 个 [E.B]T_BZYNS-VS-T_SCRTS-003
第 54 个 [E.B]T_BZYNS-VS-T_SCRTS-004
第 55 个 [E.B]T_BZYNS-VS-T_shlcy-000

第 56 个 [E.B]T_BZYNS-VS-T_shlcy-001
第 57 个 [E.B]T_BZYNS-VS-T_shlcy-002
第 58 个 [E.B]T_BZYNS-VS-T_shlcy-003
第 59 个 [E.B]T_BZYNS-VS-T_shlcy-004
第 60 个 [E.B]T_BZYNS-VS-T_silly-000
第 61 个 [E.B]T_BZYNS-VS-T_silly-001
第 62 个 [E.B]T_BZYNS-VS-T_silly-002
第 63 个 [E.B]T_BZYNS-VS-T_silly-003
第 64 个 [E.B]T_BZYNS-VS-T_silly-004
第 65 个 [E.B]T_BZYNS-VS-T_vtest-000
第 66 个 [E.B]T_BZYNS-VS-T_vtest-001
第 67 个 [E.B]T_BZYNS-VS-T_vtest-002
第 68 个 [E.B]T_BZYNS-VS-T_vtest-003
第 69 个 [E.B]T_BZYNS-VS-T_vtest-004
第 70 个 [E.B]T_BZYNS-VS-T_wenge-000
第 71 个 [E.B]T_BZYNS-VS-T_wenge-001
第 72 个 [E.B]T_BZYNS-VS-T_wenge-002
第 73 个 [E.B]T_BZYNS-VS-T_wenge-003
第 74 个 [E.B]T_BZYNS-VS-T_wenge-004
第 75 个 [E.B]T_November-VS-T_BZYNS-000
第 76 个 [E.B]T_November-VS-T_BZYNS-001
第 77 个 [E.B]T_November-VS-T_BZYNS-002
第 78 个 [E.B]T_November-VS-T_BZYNS-003
第 79 个 [E.B]T_November-VS-T_BZYNS-004
第 80 个 [E.M]T_BZYNS-VS-T_Sheikah-000
第 81 个 [E.M]T_BZYNS-VS-T_Sheikah-001
第 82 个 [E.M]T_BZYNS-VS-T_Sheikah-002
第 83 个 [E.M]T_BZYNS-VS-T_Sheikah-003
第 84 个 [E.M]T_BZYNS-VS-T_Sheikah-004
第 85 个 [E.M]T_Unicorn-VS-T_idiot
第 86 个 [W.B]T_AUGST-VS-T_BZYNS-000
第 87 个 [W.B]T_AUGST-VS-T_BZYNS-001
第 88 个 [W.B]T_AUGST-VS-T_BZYNS-002
第 89 个 [W.B]T_AUGST-VS-T_BZYNS-003
第 90 个 [W.B]T_AUGST-VS-T_BZYNS-004
第 91 个 [W.B]T_BZYNS-VS-T_simple-000
第 92 个 [W.B]T_BZYNS-VS-T_simple-001
第 93 个 [W.B]T_BZYNS-VS-T_simple-002
第 94 个 [W.B]T_BZYNS-VS-T_simple-003
第 95 个 [W.B]T_BZYNS-VS-T_simple-004
第 96 个 [W.B]T_FANCY-VS-T_BZYNS-000
第 97 个 [W.B]T_FANCY-VS-T_BZYNS-001
第 98 个 [W.B]T_FANCY-VS-T_BZYNS-002
第 99 个 [W.B]T_FANCY-VS-T_BZYNS-003
第 100 个 [W.B]T_FANCY-VS-T_BZYNS-004
第 101 个 [W.B]T_Hades-VS-T_BZYNS-000
第 102 个 [W.B]T_Hades-VS-T_BZYNS-001
第 103 个 [W.B]T_Hades-VS-T_BZYNS-002
第 104 个 [W.B]T_Hades-VS-T_BZYNS-003
第 105 个 [W.B]T_Hades-VS-T_BZYNS-004
第 106 个 [W.B]T_KILO-VS-T_BZYNS-000
第 107 个 [W.B]T_KILO-VS-T_BZYNS-001
第 108 个 [W.B]T_KILO-VS-T_BZYNS-002
第 109 个 [W.B]T_KILO-VS-T_BZYNS-003
第 110 个 [W.B]T_KILO-VS-T_BZYNS-004
第 111 个 [W.B]T_miaou-VS-T_BZYNS-000
第 112 个 [W.B]T_miaou-VS-T_BZYNS-001
第 113 个 [W.B]T_miaou-VS-T_BZYNS-002
第 114 个 [W.B]T_miaou-VS-T_BZYNS-003
第 115 个 [W.B]T_miaou-VS-T_BZYNS-004
第 116 个 [W.B]T_nobrain-VS-T_BZYNS-000
第 117 个 [W.B]T_nobrain-VS-T_BZYNS-001
第 118 个 [W.B]T_nobrain-VS-T_BZYNS-002
第 119 个 [W.B]T_nobrain-VS-T_BZYNS-003
第 120 个 [W.B]T_nobrain-VS-T_BZYNS-004
第 121 个 [W.B]T_PMNDS-VS-T_BZYNS-000
第 122 个 [W.B]T_PMNDS-VS-T_BZYNS-001
第 123 个 [W.B]T_PMNDS-VS-T_BZYNS-002
第 124 个 [W.B]T_PMNDS-VS-T_BZYNS-003
第 125 个 [W.B]T_PMNDS-VS-T_BZYNS-004
第 126 个 [W.B]T_random-VS-T_BZYNS-000
第 127 个 [W.B]T_random-VS-T_BZYNS-001
第 128 个 [W.B]T_random-VS-T_BZYNS-002
第 129 个 [W.B]T_random-VS-T_BZYNS-003
第 130 个 [W.B]T_random-VS-T_BZYNS-004
第 131 个 [W.B]T_SCRTS-VS-T_BZYNS-000
第 132 个 [W.B]T_SCRTS-VS-T_BZYNS-001
第 133 个 [W.B]T_SCRTS-VS-T_BZYNS-002
第 134 个 [W.B]T_SCRTS-VS-T_BZYNS-003
第 135 个 [W.B]T_SCRTS-VS-T_BZYNS-004
第 136 个 [W.B]T_Sheikah-VS-T_BZYNS-000
第 137 个 [W.B]T_Sheikah-VS-T_BZYNS-001
第 138 个 [W.B]T_Sheikah-VS-T_BZYNS-002
第 139 个 [W.B]T_Sheikah-VS-T_BZYNS-003
第 140 个 [W.B]T_Sheikah-VS-T_BZYNS-004
第 141 个 [W.B]T_shlcy-VS-T_BZYNS-000
第 142 个 [W.B]T_shlcy-VS-T_BZYNS-001
第 143 个 [W.B]T_shlcy-VS-T_BZYNS-002
第 144 个 [W.B]T_shlcy-VS-T_BZYNS-003
第 145 个 [W.B]T_shlcy-VS-T_BZYNS-004
第 146 个 [W.B]T_silly-VS-T_BZYNS-000
第 147 个 [W.B]T_silly-VS-T_BZYNS-001
第 148 个 [W.B]T_silly-VS-T_BZYNS-002
第 149 个 [W.B]T_silly-VS-T_BZYNS-003
第 150 个 [W.B]T_silly-VS-T_BZYNS-004
第 151 个 [W.B]T_simple-VS-T_BZYNS-000
第 152 个 [W.B]T_simple-VS-T_BZYNS-001
第 153 个 [W.B]T_simple-VS-T_BZYNS-002
第 154 个 [W.B]T_simple-VS-T_BZYNS-003
第 155 个 [W.B]T_simple-VS-T_BZYNS-004
第 156 个 [W.B]T_wenge-VS-T_BZYNS-000
第 157 个 [W.B]T_wenge-VS-T_BZYNS-001
第 158 个 [W.B]T_wenge-VS-T_BZYNS-002
第 159 个 [W.B]T_wenge-VS-T_BZYNS-003
第 160 个 [W.B]T_wenge-VS-T_BZYNS-004

160 局 10 胜，交的是未改进的 LIMA1，只用了两个小时编写，有致命 bug，目的只是看看别人这么打

以下是第二次热身赛结果：（我们是 RABBIT）

第 0 个 [E.B]T_David-VS-T_RABBIT-001
第 1 个 [E.B]T_RABBIT-VS-T_ALPHA-000
第 2 个 [E.B]T_RABBIT-VS-T_ALPHA-001
第 3 个 [E.B]T_RABBIT-VS-T_ALPHA-002
第 4 个 [E.B]T_RABBIT-VS-T_ALPHA-003
第 5 个 [E.B]T_RABBIT-VS-T_ALPHA-004
第 6 个 [E.B]T_RABBIT-VS-T_Apollo-001
第 7 个 [E.B]T_RABBIT-VS-T_Apollo-002
第 8 个 [E.B]T_RABBIT-VS-T_AVCNA-001
第 9 个 [E.B]T_RABBIT-VS-T_AVCNA-002
第 10 个 [E.B]T_RABBIT-VS-T_AVCNA-003
第 11 个 [E.B]T_RABBIT-VS-T_AVCNA-004
第 12 个 [E.B]T_RABBIT-VS-T_eighth-000
第 13 个 [E.B]T_RABBIT-VS-T_eighth-001
第 14 个 [E.B]T_RABBIT-VS-T_eighth-002
第 15 个 [E.B]T_RABBIT-VS-T_eighth-004
第 16 个 [E.B]T_RABBIT-VS-T_excited-000
第 17 个 [E.B]T_RABBIT-VS-T_excited-001
第 18 个 [E.B]T_RABBIT-VS-T_excited-002
第 19 个 [E.B]T_RABBIT-VS-T_excited-004
第 20 个 [E.B]T_RABBIT-VS-T_harry-000
第 21 个 [E.B]T_RABBIT-VS-T_harry-001
第 22 个 [E.B]T_RABBIT-VS-T_harry-002
第 23 个 [E.B]T_RABBIT-VS-T_harry-003
第 24 个 [E.B]T_RABBIT-VS-T_harry-004
第 25 个 [E.B]T_RABBIT-VS-T_Hellofine-000
第 26 个 [E.B]T_RABBIT-VS-T_Hellofine-001
第 27 个 [E.B]T_RABBIT-VS-T_Hellofine-002
第 28 个 [E.B]T_RABBIT-VS-T_Hellofine-003
第 29 个 [E.B]T_RABBIT-VS-T_Hellofine-004
第 30 个 [E.B]T_RABBIT-VS-T_lkjhg-000
第 31 个 [E.B]T_RABBIT-VS-T_lkjhg-001
第 32 个 [E.B]T_RABBIT-VS-T_lkjhg-002
第 33 个 [E.B]T_RABBIT-VS-T_lkjhg-004
第 34 个 [E.B]T_RABBIT-VS-T_sabac-000
第 35 个 [E.B]T_RABBIT-VS-T_sabac-001
第 36 个 [E.B]T_RABBIT-VS-T_sabac-002
第 37 个 [E.B]T_RABBIT-VS-T_sabac-003
第 38 个 [E.B]T_RABBIT-VS-T_sabac-004
第 39 个 [E.B]T_RABBIT-VS-T_SWIFT-000
第 40 个 [E.B]T_RABBIT-VS-T_SWIFT-001
第 41 个 [E.B]T_RABBIT-VS-T_SWIFT-002
第 42 个 [E.B]T_RABBIT-VS-T_SWIFT-003
第 43 个 [E.B]T_RABBIT-VS-T_SWIFT-004
第 44 个 [E.B]T_RABBIT-VS-T_track-002
第 45 个 [E.B]T_RABBIT-VS-T_unique-000
第 46 个 [E.B]T_RABBIT-VS-T_unique-001
第 47 个 [E.B]T_RABBIT-VS-T_unique-002
第 48 个 [E.B]T_RABBIT-VS-T_unique-003
第 49 个 [E.B]T_RABBIT-VS-T_unique-004
第 50 个 [E.B]T_simple-VS-T_RABBIT-000
第 51 个 [E.B]T_simple-VS-T_RABBIT-002
第 52 个 [E.B]T_simple-VS-T_RABBIT-003
第 53 个 [E.B]T_simple-VS-T_RABBIT-004
第 54 个 [E.B]T_yzzxx-VS-T_RABBIT-000
第 55 个 [E.B]T_yzzxx-VS-T_RABBIT-002
第 56 个 [E.B]T_yzzxx-VS-T_RABBIT-003
第 57 个 [E.M]T_David-VS-T_RABBIT-000
第 58 个 [E.M]T_David-VS-T_RABBIT-002
第 59 个 [E.M]T_David-VS-T_RABBIT-003
第 60 个 [E.M]T_David-VS-T_RABBIT-004
第 61 个 [E.M]T_King-VS-T_RABBIT-000
第 62 个 [E.M]T_King-VS-T_RABBIT-004
第 63 个 [E.M]T_RABBIT-VS-T_Apollo-000
第 64 个 [E.M]T_RABBIT-VS-T_Apollo-003
第 65 个 [E.M]T_RABBIT-VS-T_Apollo-004
第 66 个 [E.M]T_RABBIT-VS-T_excited-003
第 67 个 [E.M]T_RABBIT-VS-T_King-001
第 68 个 [E.M]T_RABBIT-VS-T_learn-000
第 69 个 [E.M]T_RABBIT-VS-T_learn-001
第 70 个 [E.M]T_RABBIT-VS-T_learn-002
第 71 个 [E.M]T_RABBIT-VS-T_learn-003
第 72 个 [E.M]T_RABBIT-VS-T_learn-004
第 73 个 [E.M]T_RABBIT-VS-T_lkjhg-003
第 74 个 [E.M]T_RABBIT-VS-T_track-000
第 75 个 [E.M]T_RABBIT-VS-T_track-001
第 76 个 [E.M]T_RABBIT-VS-T_track-003
第 77 个 [E.M]T_RABBIT-VS-T_yzzxx-000
第 78 个 [E.M]T_RABBIT-VS-T_yzzxx-001
第 79 个 [E.M]T_RABBIT-VS-T_yzzxx-002
第 80 个 [E.M]T_RABBIT-VS-T_yzzxx-003
第 81 个 [E.M]T_RABBIT-VS-T_yzzxx-004
第 82 个 [E.M]T_Unicorn-VS-T_idiot
第 83 个 [W.B]T_ALPHA-VS-T_RABBIT-000
第 84 个 [W.B]T_ALPHA-VS-T_RABBIT-001
第 85 个 [W.B]T_ALPHA-VS-T_RABBIT-002
第 86 个 [W.B]T_ALPHA-VS-T_RABBIT-003
第 87 个 [W.B]T_ALPHA-VS-T_RABBIT-004
第 88 个 [W.B]T_Apollo-VS-T_RABBIT-000
第 89 个 [W.B]T_Apollo-VS-T_RABBIT-001
第 90 个 [W.B]T_Apollo-VS-T_RABBIT-002
第 91 个 [W.B]T_Apollo-VS-T_RABBIT-003
第 92 个 [W.B]T_Apollo-VS-T_RABBIT-004
第 93 个 [W.B]T_AVCNA-VS-T_RABBIT-000
第 94 个 [W.B]T_AVCNA-VS-T_RABBIT-001
第 95 个 [W.B]T_AVCNA-VS-T_RABBIT-002
第 96 个 [W.B]T_AVCNA-VS-T_RABBIT-003
第 97 个 [W.B]T_AVCNA-VS-T_RABBIT-004
第 98 个 [W.B]T_eighth-VS-T_RABBIT-000
第 99 个 [W.B]T_eighth-VS-T_RABBIT-001
第 100 个 [W.B]T_eighth-VS-T_RABBIT-002
第 101 个 [W.B]T_eighth-VS-T_RABBIT-003
第 102 个 [W.B]T_eighth-VS-T_RABBIT-004
第 103 个 [W.B]T_excited-VS-T_RABBIT-000
第 104 个 [W.B]T_excited-VS-T_RABBIT-001
第 105 个 [W.B]T_excited-VS-T_RABBIT-002
第 106 个 [W.B]T_excited-VS-T_RABBIT-003
第 107 个 [W.B]T_excited-VS-T_RABBIT-004
第 108 个 [W.B]T_harry-VS-T_RABBIT-000
第 109 个 [W.B]T_harry-VS-T_RABBIT-001
第 110 个 [W.B]T_harry-VS-T_RABBIT-002
第 111 个 [W.B]T_harry-VS-T_RABBIT-003
第 112 个 [W.B]T_harry-VS-T_RABBIT-004
第 113 个 [W.B]T_Hellofine-VS-T_RABBIT-000
第 114 个 [W.B]T_Hellofine-VS-T_RABBIT-001
第 115 个 [W.B]T_Hellofine-VS-T_RABBIT-002
第 116 个 [W.B]T_Hellofine-VS-T_RABBIT-003
第 117 个 [W.B]T_Hellofine-VS-T_RABBIT-004
第 118 个 [W.B]T_lkjhg-VS-T_RABBIT-000
第 119 个 [W.B]T_lkjhg-VS-T_RABBIT-001
第 120 个 [W.B]T_lkjhg-VS-T_RABBIT-002
第 121 个 [W.B]T_lkjhg-VS-T_RABBIT-003
第 122 个 [W.B]T_lkjhg-VS-T_RABBIT-004
第 123 个 [W.B]T_RABBIT-VS-T_David-000
第 124 个 [W.B]T_RABBIT-VS-T_David-001
第 125 个 [W.B]T_RABBIT-VS-T_David-002
第 126 个 [W.B]T_RABBIT-VS-T_David-003
第 127 个 [W.B]T_RABBIT-VS-T_David-004
第 128 个 [W.B]T_RABBIT-VS-T_simple-000
第 129 个 [W.B]T_RABBIT-VS-T_simple-001
第 130 个 [W.B]T_RABBIT-VS-T_simple-002
第 131 个 [W.B]T_RABBIT-VS-T_simple-003
第 132 个 [W.B]T_RABBIT-VS-T_simple-004
第 133 个 [W.B]T_sabac-VS-T_RABBIT-000
第 134 个 [W.B]T_sabac-VS-T_RABBIT-001
第 135 个 [W.B]T_sabac-VS-T_RABBIT-002
第 136 个 [W.B]T_sabac-VS-T_RABBIT-003
第 137 个 [W.B]T_sabac-VS-T_RABBIT-004
第 138 个 [W.B]T_SWIFT-VS-T_RABBIT-000
第 139 个 [W.B]T_SWIFT-VS-T_RABBIT-001
第 140 个 [W.B]T_SWIFT-VS-T_RABBIT-002
第 141 个 [W.B]T_SWIFT-VS-T_RABBIT-003
第 142 个 [W.B]T_SWIFT-VS-T_RABBIT-004
第 143 个 [W.B]T_track-VS-T_RABBIT-003
第 144 个 [W.B]T_unique-VS-T_RABBIT-000
第 145 个 [W.B]T_unique-VS-T_RABBIT-001
第 146 个 [W.B]T_unique-VS-T_RABBIT-002
第 147 个 [W.B]T_unique-VS-T_RABBIT-003
第 148 个 [W.B]T_unique-VS-T_RABBIT-004
第 149 个 [W.B]T_yzzxx-VS-T_RABBIT-001
第 150 个 [W.B]T_yzzxx-VS-T_RABBIT-004
第 151 个 [W.M]T_King-VS-T_RABBIT-001
第 152 个 [W.M]T_King-VS-T_RABBIT-002
第 153 个 [W.M]T_King-VS-T_RABBIT-003
第 154 个 [W.M]T_learn-VS-T_RABBIT-000
第 155 个 [W.M]T_learn-VS-T_RABBIT-001
第 156 个 [W.M]T_learn-VS-T_RABBIT-002
第 157 个 [W.M]T_learn-VS-T_RABBIT-003
第 158 个 [W.M]T_learn-VS-T_RABBIT-004
第 159 个 [W.M]T_RABBIT-VS-T_AVCNA-000
第 160 个 [W.M]T_RABBIT-VS-T_eighth-003
第 161 个 [W.M]T_RABBIT-VS-T_King-000
第 162 个 [W.M]T_RABBIT-VS-T_King-002
第 163 个 [W.M]T_RABBIT-VS-T_King-003
第 164 个 [W.M]T_RABBIT-VS-T_King-004
第 165 个 [W.M]T_RABBIT-VS-T_track-004
第 166 个 [W.M]T_simple-VS-T_RABBIT-001
第 167 个 [W.M]T_track-VS-T_RABBIT-000
第 168 个 [W.M]T_track-VS-T_RABBIT-001
第 169 个 [W.M]T_track-VS-T_RABBIT-002
第 170 个 [W.M]T_track-VS-T_RABBIT-004

T_RABBIT

31 / 170

win for {'miss_ball': 13, 'invalid_bounce': 18}

lose for {'miss_ball': 32, 'invalid_bounce': 107}

170 局 31 胜，交的是有 bug 的 LIMA2，其实此时 LIMA1 已经没有 bug 了，但为测试最新代码还是交了 LIMA2，结果经常 invalid_bounce，边角考虑不到位。

3.2 结果分析

第一次热身赛我们每次都回中间，而在第二次热身赛时我们稍微添加了智能的因素，站在对方的角度考虑打哪个角比较好，这样有些队的算法比较死，就每次被我们算到，我们在不用跑位的时候就不跑位，这样很快就拉开了血量的差距，但对于随机算法和智能算法还是不好应对。

最终算法会记录对方前三次跑位取平均，这样就不会出现对方一直不跑位的情况了，这是我们在热身赛中经常遇到的问题，因为那时别人记录了历史数据我们却没有。而对方打哪个角概率的评估使得我们不用每次都跑到中间，这样虽然每次都会损一点血，但不会出现总是白跑或者是一下跑 100 万的情况，所以之前算法中对方一直不动我们一直动的硬伤没有再出现了，一下损 2500 的情况也少了很多。

最终算法由于没有用到递归，用的是连续函数求极值而不是在一堆点里搜索，所以运行时间很短，除非是当道具权重设置得太小时，导致不去接道具，这样和自己算法打的时候会陷入双方不损血的死循环最后 time out，这样时间会稍长，但实战中不会遇到和自己一样的算法，所以不会出现这种情况。

3.3 经典战局

GOLF 作为东区冠军具有很强的实力，但却不小心被我们打败，其实也是因为他们的算法有些漏洞，而我们当时提交的算法虽简单却正好克他们，他们统计我们每次打上角还是打下角，而我们每次打损血少的角，这样在他们看来我们打的角是没有规律的，所以血量一直没有拉开差距。当我们双方血量都少于五万时，忽然他们就 miss ball 了。后来询问他们才知道，他们设定了当血量低于对方时放弃跑位到安全位置以减少跑位损血，但当时我们血量差距不大，他们随时会追上来，却直接放弃治疗了，我们打到一个他们跑不到的角他们就 miss ball 了

4 实习过程总结

4.1 分工与合作

1.最终小组分工

陶天阳：整个程序大体构架的建立，主要代码编写

黄佳磊：历史数据存储的代码
沈泽盛：计算迎球公式及实现代码
刘启航：道具算法公式和权重的计算
冯曦林：观看复盘数据总结经验
闫丽影：完成报告

2.小组活动记录

5.24 小组第一次组会

出席人员：陶天阳，沈泽盛，冯曦林，刘启航，闫丽影

讨论内容:确定小组头像，选择如下图片作为我们组的头像，展示我们组谦虚乐于学习的友好形象，初步确定小组分工。



（小组头像 LIMA）

陶天阳：代码编程　冯曦林：算法设计
沈泽盛：战略分析　闫丽影：撰写报告
刘启航：配合组长　黄佳磊：谱写队歌（666）

小组在清凉舒适优雅清逸的泊星地完成了第一次组会合影。



5.29 热身赛

出席人员：全体

内容：陶天阳组长临时编写了一个比较弱的对战代码，提交到助教处，用来

测试对手的战术，热身赛中我们采用保守的战略，没有展示我们的实力。代码中不包含核心科技。陶天阳组长代表我小组进行了报告。

5.30 端午节+第二次组会理教

出席人员：组长：陶天阳；组员：沈泽盛，冯曦林，闫丽影，黄佳磊

内容:此时还没有获得热身赛的结果，不能对其他组进行分析。但是我们已经认识到了时间的紧迫，开始了对大作业的真正研究。所以组长陶天阳给全体组员逐段分析 pingpange 和 table 的代码。逻辑清晰，富有耐心。同时组员们针对代码进行讨论。

分配阶段性小组任务：黄佳磊与组长负责代码编写，沈泽盛与冯曦林负责分析对战数据进行战略设计，刘启航配合组长。闫丽影记录。



（端午节的清晨组员们带着早餐在钻研代码）

成员编写了陪练代码，陪练代码逻辑简单，打角，归中，不接道具。

6.1 儿童节+第三次组会泊星地

全体组员出席讨论

因为第二天要进行第二次热身赛，而我们的战术代码尚未成型，所以全体组员齐聚泊星地讨论战术和代码。经过对第一次热身赛复盘数据的逐盘分析，我们的出了多数小组的跑位战术，并经过复杂有深度的计算设计了应对战术。最终讨论得出跑位与击球战略，汇总了各个成员的整体进度。最后几位男同学及一起实现代码。五位男生组员在麦当劳刷夜设计实现代码，至凌晨两三点，可以说是非常辛苦。最终在周五的第二次热身赛前提交了比较完善的 LIMA2。



（凌晨两点钟的麦当劳）

6.2,6.3，6.4 三天完善改进代码

第二次热身赛惨败后组长深夜思考到凌晨三点，最终决定推翻之前的体系，构建全新的体系。在这三天中五位战斗人员昼夜不停的更新改进算法，在战术上取得了突破性的进展，首先开发出了更为强大的 LIMA3（也就是最终击败 GOLF 的算法）。为了使算法更具智能性，黄佳磊同学对 LIMA3 进行了大面积的修改，增加了历史记录读取分析的功能，但是非常遗憾，没能在决赛之前完成 LIMA3 到 LIMA4 的进化，仍然有 Bug 没有解决。最终仍然使用 LIMA3 参加了最终的比赛。

6.5，决赛，惜败。

我们组阻止了其它三个区出现的全胜现象，导致前四名差距很小，分别是 12 分，10 分，10 分，9 分，我们为 9 分差一分出线。

虽然我组未能出线但是 LIMA 成功击败了 GOLF，最终 GOLF 成为决赛第三名。

3.小组交流方式

小组多次召开组会，面对面商量战术。在微信群中进行文本交互。男生组员因为宿舍距离较近，也经常在宿舍内讨论问题。

4.2 经验与教训

自从五月十五日大作业组员招募至今，我们数算 LIMA 小组已经共同努力了近一个月。数算作业在期末季给我们增加了别样的色彩。从起初的略微消极怠工，到第一次热身赛仓促出赛而别其他小组碾压，然后第二次小组赛仍有 bug，第二天开始构造全新的体系。在小组赛中 LIMA 成功击败 GOLF。但是十分不幸 LIMA 同时被两个 GOLF 的手下败将击败，与出线无缘。虽然我们小组没能成功出线，但我们小组是第一个打败 GOLF 的，而第二个和第三个分别是最终的冠军和亚军。我们小组成员的努力是不能忽略的，我们的实力也是不容小觑的。也是正因为我们组才证明了这个游戏是有环环相克的性质的。

在小组赛结束后组长立刻把代码发给了 GOLF 组让其找到失败的原因并作出代码调整，经过和 GOLF 交流发现我们的历史数据运用思想不同，他们记录哪个角，而我们记录是否作出损血最少的选择。这导致了游戏的不确定性变得很大

在比赛结束后我们也与 Oscar 组进行了交流，互相阐述了算法思路。我们了解到 Oscar 组（冠军组）的算法的最大特点就是深度优先搜索，不是由人来设计战术，而是由计算机计算各个区位的结果通过比较得出它的决策。而我们的战术无论是跑位还是击球都是先对对手进行预测，再结合自己的血量条件人为设计决策函数。也许我们的决策在某些情况下是无敌的，但是一旦预测失误，我们的战术就会平淡无奇而失去作用。

经过虚心学习，我们认识到，人脑设计的战术不能应对所有战场情况，而计算机却可以做到这一点。我们要致力于开发算法来实现计算机自身的智能。

4.3 建议与设想

本次实习作业游戏设计比较有趣，可玩性较高，道具的使用增加了游戏难度，在

游戏成型过程中广泛征集了大家的意见，非常民主。在这里略微提出几点小建议，建议设置复活赛，我的想法是随机分为四组后每组第一出线，其余组再随机分为四组进行比赛，每组第一出线。这样可以增加各组之间相遇的几率，每一组挑战的对手数量增加，也许更能反映真实战斗力。

5 致谢

首先感谢 LIMA 组的全体组员的通力合作，尤其是组长在我们团队中发挥了极其重要的作用。感谢所有为了设计算法烧脑的战斗人员。感谢老师和助教在实习过程中对我们的指导帮助。

6 参考文献

无



ZULU

组长：贺旎妮

组员：曲建，范锐聪，窦煜杰，唐彦涛

小组赛战绩：W5

数据结构与算法课程实习作业报告

窦煜杰 范锐聪 贺旎妮* 刘启源 曲建 唐彦涛

摘要：

本组算法的核心在两个预测，一个是预测对方的跑位，这样可以控制击球位置尽量多的使对方消耗体力；另一个是预测对方来球的位置，在己方跑位时节省体力。击球时采用的是打到对方桌面最边角位置的策略。道具策略比较简单，得到的时候就直接用。主要用到的算法有 if 条件语句判断和 for 循环语句，同时采用了类、字典等数据类型。

关键字：

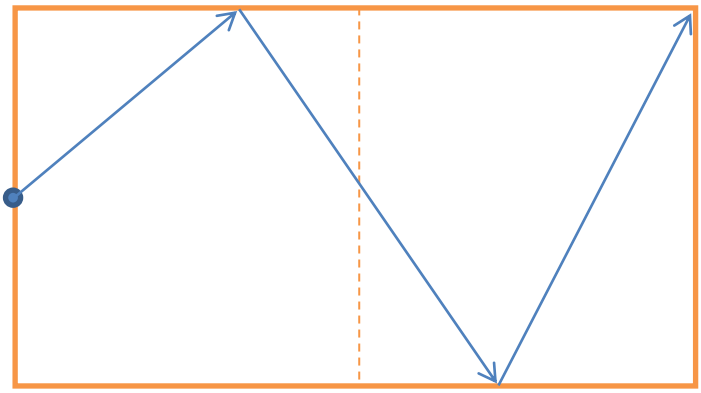
字典 if 条件判断 for 循环

1 算法思想

1.1 总体思路

1.1.1 发球策略

发球算法为固定从 west 正中 y 向速度为 1666 发球，这样发球经过上、下两次弹击会打到桌面上边角。下图展示了我们的发球策略。



1.1.2 接球策略

（1）迎球阶段：

使用球的位置和己方位置之差，就是要迎球的距离。

（2）击球阶段：

a) 使用道具的策略：

i. 如果自身的道具箱里有道具，则立即使用。

ii. 如果自身的道具箱里没有道具，则不使用。

b) 击球策略

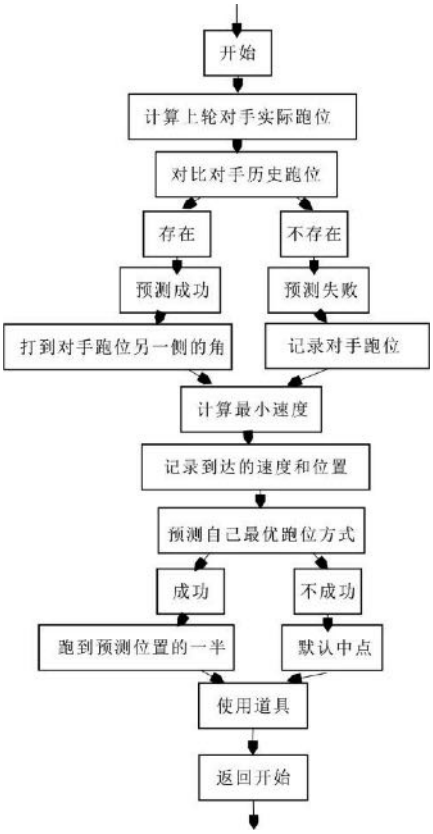
由于无法直接得到对方的跑位，需要通过计算对方在击球时的体力消耗和总体力消耗的差值，间接得到对方上一轮的跑位数据。在击球前会进行对方跑位的预测，并根据对方实际跑位进行修正。根据预测对方跑位的位置，且在球弹一下和两下间选择速度改变较小的方式，将球打在远离对方位置的一侧边角，使对方在击球段消耗尽量最多的体力。

1.1.3 跑位策略

由于从上一轮击球到下一轮击球期间跑位消耗的体力是反向抛物线的关系，即跑位停留在两轮击球位置的中间体力消耗最少。所以对下一轮击球位置的预判非常重要。

这里同样有一个预测机制，根据历史记录的信息判断对方来球的位置，选择等在上一轮击球位置和预测击球位置中间。并根据实际来球位置修正预测结果。最开始默认跑位为中点。

1.2 算法流程图



1.3 算法运行时间复杂度分析

程序的循环部分没有嵌套和数量减少，程序的时间复杂度为 O(n)。

2 程序代码说明

2.1 数据结构说明

我们主要采用字典、类等数据类型；主要利用字典作为对方跑位和击球历史存储库。

2.2 函数说明

我们定义的函数有：serve，valid_speed，get_fly_ans，以及 play。
serve 函数是发球函数，定义从中间发球，速度为 1666，这样球在反弹两次后触及对方上桌角。
valid_speed 控制球的速度，根据球所在的位置计算击球速度的上下限，避免出现 invalid_bounce 的情况。

get_fly_ans 判断击球后球落在对方边界的位置，为了方便起见，我们采取化曲为直的方法，将球经过的路径从镜面反射的曲线化为直线，对应到对方的落点。
play 是控制击球的函数。对方的体力消耗由跑位消耗和击球消耗组成，知道对方击球的体力消耗，就可以计算对方跑位的体力消耗，将这些数据存储起来，成为接下来的回合中预测对方跑位的依据：（1）通过预测对方跑位，计算球速，将球击到离预测对方落点尽量远的角，消耗对方更多的体力。（2）如果通过历史数据，能够预测对方击球的落点，我方把球跑位到该落点与现在位置的中点；如果没有历史数据，我方将球跑位到两板的中点。

2.3 程序限制

程序在最开始写的时候对战常出现 invalid balance，经检查应该是正好打在边角位置，所以有时没有反弹或反弹超过三次，经将程序内边界数据减小 1 就避免了这种边界情况。
后续的模拟对战中没有出现程序本身的 bug。

3 实验结果

3.1 测试数据

实验环境说明：

- 硬件配置：CPU Intel(R) Processor 5Y70 CPU @ 1.10GHz 1.30GHz/内存 4.00 GB
- 操作系统：windows 8 / 64 位操作系统，基于 x64 的处理器
- Python 版本： 3.4

本组没有参加第一次热身赛，除了正式的程序 zulu.py，第二次热身赛前用于测试的程序有 leg.py，参与第二次热身赛的程序 yzzxx.py，第二次热身赛后正式版之前的一版胜率较高的程序 genius.py，均用于内部测试提高。最后参与正式比赛的 zulu.py 可以完全战胜上述其他程序。
用于参加第二次热身赛的 yzzxx 胜率 34/170，排在所有程序第 15 位，具体胜负原因如下表：

	Miss ball	Invalid bounce
Win for	24	10
Lose for	51	85

热身赛结果反映出在 invalid bounce 上程序仍存在问题，所以这是后续修改中第一个改正的部分。
此外，经过复看和胜率较高的组的对战数据，可以看出胜率较高的组具有不同的策略（详见胜率排序.docx 的分析），可以直观看出的是程序的跑位并不是固定的，这也促使本组程序在后续修改中加入了预测功能。

3.2 结果分析

在和测试程序对战过程中，可以看到预测是很有效的，可以在体力上建立比较大的优势。
正赛中胜了 20 局，积三分，跟大佬们比还是被碾压了……

算法由于本身结构比较简单，时间消耗很小。

4 实习过程总结

4.1 分工与合作

	参与组会及讨论	代码	报告
窦煜杰	3		2.3 程序代码说明
范锐聪	1		4 实习过程总结
贺旎妮	3	Zulu.py	实习报告剩余部分
刘启源			
曲建	1	Leg.py	
唐彦涛			

由于小组人员组建的一些客观问题，我组的人数先期仅为四位，中后期增补到六位。所以在分工合作以及进度的安排中，面临一些额外的考验。很高兴我们在这个过程较为妥善得解决了这些问题。

首先，我们在小组成立后迅速建立了微信群，在线上确保了日常沟通、事项通知的及时性。随后，我组就本次实习的具体内容进行了大模块的划分，主要包括程序编写部分，中期事务部分、后期工作部分。

程序编写部分包括算法设计、代码编写，是本次实习的核心部分，也是最为考验的部分。所以原则上，我们全组成员都倾情参与，以组长贺旖旎同学、窦煜杰同学和　同学为主力军，其余组员协助，并积极寻求老师、同学的指点、帮助。

中期事务部分包括与老师助教的沟通，对实习、比赛进度的跟进，包括热身赛在内的相关事务的出席，这一部分主要由组长贺旖旎同学负责把握。

后期工作部分主体为报告撰写，根据报告的内容划分，原则上同样是全员参与，由组长贺旖旎同学负责统筹。

接下来，我组就实习的整体安排进行了时间进度的划分，确定第一周进行前期安排；第二周主力程序编写，在热身赛前完成初期版本；热身赛后的周末根据热身赛情况修改出正赛版本；正赛结束后进行报告撰写等其他后期工作。

第一周，我组在微信上进行了组员的确认后，确定了第一次线下组会的安排，并在线下组会上正式决定了以上所有的内容、进度安排，并对算法思想、代码编写进行了初步的讨论。随后，我组主要在线上进行相关讨论。

第二周，我组着力于本次实习的核心部分，全员投入代码的编写。由组长贺旖旎同学、窦煜杰同学和曲建同学为主力，进行了代码的主体编写，并且在过程中产生疑惑，遇到障碍，积极讨论并向老师同学请教。本周内我们进行了第二次线下组会，在这次组会上我们直接对代码进行了测试，对相关问题进行了面对面的讨论，得出了修改意见，随后在热身赛前主要由组长贺旖旎同学完成了热身赛版本的修改。热身赛后，组长贺旖旎同学带回了热身赛战报等相关信息。我们在最后的周末主要以线上讨论的形式，进一步完成了最终正赛版本代码的编写与提交。

正赛结束后，第三周我们根据正赛结果进行线上基础讨论，对本次实习进行了总结，并安排了报告的撰写，完成并提交。



（第二次组会照片）

4.2 经验与教训

得意之处：

1、 我组在组建后对相关内容的分工、进度的安排非常及时，客观上提升了工作效率；

2、 我组在实习过程中积极讨论，互相协作，在前期人数较少的情况下尽全力，最大限度地克服了人力不够充裕的困难，符合本次实习，锻炼我们团队协作能力的本质目标；

3、 我组在实习过程中遇到了一些困难，在这种时候我们总是正确地认识自我，正视困难的存在与不可避免，积极地寻求老师、同学等的指点与帮助，这符合实习过程中，本着以“实现自我成长与进步”为本质目的的原则积极开拓各种途径完善自我，工作中学习，学习为工作的精神。

经验教训：

1、 在小组组建的问题上，因为前期的沟通等问题，其组建有些延后、有些仓促，压缩了实际工作的可用时间，客观上给实习造成了困难；

2、 由于小组的组建较为仓促，所以组员的平均代码编写能力并不能够合理把握，及时抓住最大化利用效率的方式方法，所以在代码编写这一核心过程中，也体会到了“举步维艰”；

3、 小组成员中期出现了一些变动，针对新成员的加入与沟通跟进得有一定程度滞后，客观上牺牲掉了一部分理想效率。

4.3 建议与设想

1、 作为一门选课人数非常多、且选课学生所属年级、学院、专业非常分散的课程，在这一

庞大的人群中沟通不畅的问题是非常频发且种类繁多的，所以在这样一次大规模的实习安排中，直接的学生自行分组往往会出现一些冲突，一些疏漏，尤其时间上临近春季学期期末，客观上会有更多个体面临时间精力的窘迫，从而导致小组组建并不能保证及时与用心。基于此，我认为可以在学期中借助于课堂、作业、上机等等机会和形式，提前有一些小组化意识的渗透，从而打好选课同学互相沟通、了解的基础，类似“树莓”活动，这对实习时的小组组建一定很有帮助；

2、 就比赛参与的情况来看，小组赛和八强战的进行过于冗长，所以我们认为，可以寻求可靠的线上平台，对小组赛和淘汰赛的一部分进行课下、线上的直播等等，不必将整个比赛都放到课堂上来完成。

3、 最后，这样一次实习对数据算法技能的实践利用以及团队协作能力的实践、锻炼都是十分有意义的，在本次活动中，我组成员获益良多。



WHISKEY

组长：张容榕

组员：郑黛，吕品妍，计启迪，邢仕杰，谷鹤，郭晓彤

小组赛战绩：W6

数算大作业报告

漂移乒乓



Whiskey：

张容榕* 郑黛 吕品妍 郭晓彤 计启迪 谷鹤 邢仕杰

1. 算法思想

1.1 总体思路

我们以求局部最优解为主要的算法结构，一般最优化算法只能求局部最优值,很难求出全局最优值，所以有待改正。

1.1.1 发球算法策略

人站在场地中间发球，可以减少一次跑位，速度和方向皆是随机。（随机速度有上限和下限，以控制球碰壁 1-2 次）

1.1.2 跑位策略

原本血量是十万，当血量到 55000 以下时，有可能出现死角（由于血量越低，速度越慢），失误率更高。所以分两个时期，在一开始的时候，击球完毕不跑位，只根据敌方回球位置迎球。当血量达到 55000 以下时，每次打完球回到正中间，这样回球的时候期望距离最小。

1.1.3 击球加速策略

计算我方加速需要损耗的体力，和我方打过去以后对方损耗的体力。两者取差。每次取最大值（最大值意味着我方收益最大）作为发球速度。

1.1.4 道具策略

贪心法策略，即有道具就用。

1.2 算法流程图

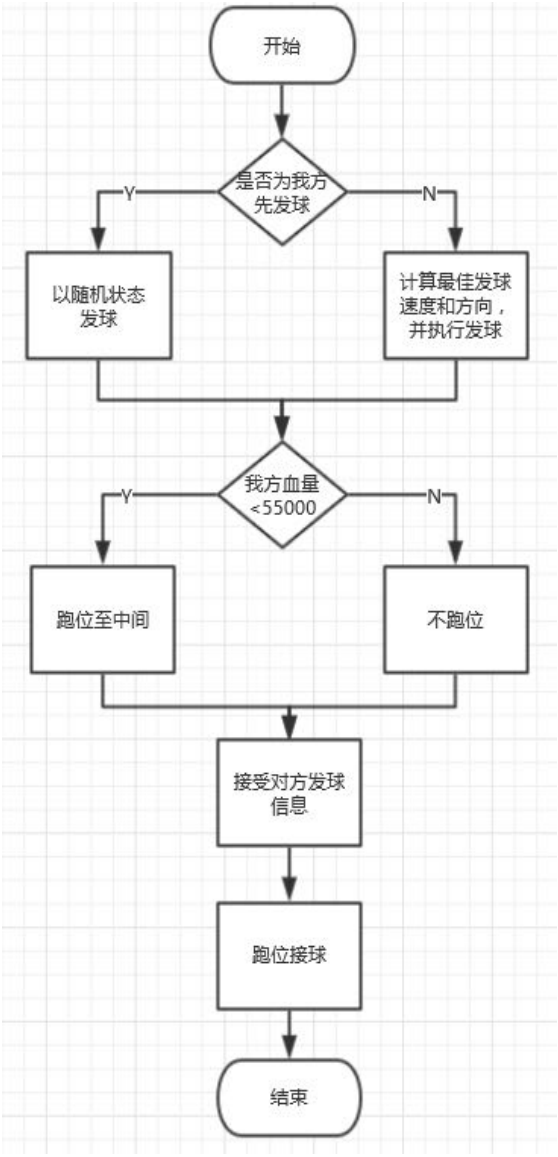


图 1 算法流程图

1.3 算法运行时间复杂度分析

时间复杂度主要体现在遍历所有的速度可能性，对于题目规定的设定条件，时间恒定，碰墙 1-2 次，则存在一个速度的上限和一个速度的下限。我们考虑两种极端情况，如图 2、3 所示，AB 和 CD 线段表示东西方球拍可以移动的范围。当在只碰壁一次的情况下，如图 2，由于时间恒定，当发球点无限接近于 A 点，接球点无限接近于 C 点，则会产生最慢速度，速度接近于 0。当在碰壁两次的情况下，如图 3，由于时间恒定，当发球点无限接近于 B 点，接球点无限接近于 C 点，则会产生最快速度，速度接近于 167。

遍历从-1667~1667，最差的情况下遍历 3334 次，最优的情况下遍历一次。综合分析，

时间复杂度为 O（n）。

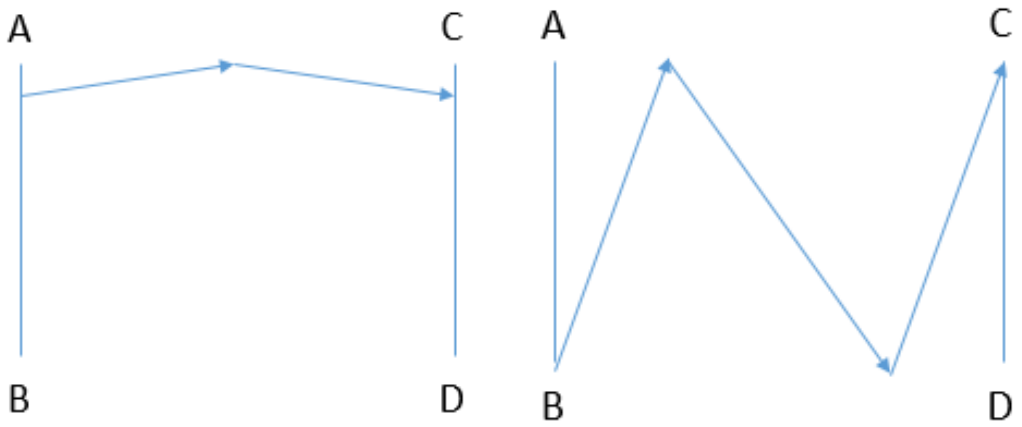


图 2 最慢和最快速度下的乒乓球移动

1.4 算法中 play 函数的接口和功能

在 play 函数中，为了获得对球的加速，我们的实现过程如下：首先遍历速度，找到最大速度 top_speed 和最小速度 min_speed；其次将 enemy_loss 和 my_loss 相减，根据遍历找出来最大的值，放入参数 gain 中；最后，找到对应的 speed，算出对球的加速 select_speed-ball_speed。

2 程序代码说明

2.1 数据结构说明

我们主要用到的是“树”的数据结构，实质上是从“树”的思想得到启发。我们发现了Minimax 博弈树，它是一棵包含了所有可能的执行步骤的树，其中每个节点可拥有任意数目的子节点。使用它能让双方能看到博弈的全部信息，可用于双方轮流博弈，每次只执行一步。应对于我组代码，主要呈现于每回对局结束计算双方差值，让我方占据有利条件，这在后面的代码中有具体说明。

博弈的任务是设计一个合适的估值函数对当前节点进行估值，并对树进行一定深度的搜索，当搜索达到一定程度时，对当前状态进行计算，再进行向上回溯，直到回到根节点。根据极大极小原理选择一个估值最好的节点为最佳着色节点，可以在一定程度上取胜。不过它的核心思想实际上是对博弈树节点的估值函数和博弈树搜索过程的结合。

列表：有时我们需要把一些东西储存起来，因为他们有某种直接或者间接的联系。很多人都知道数组的概念，但是数组的基本要求是所偶的数据类型必须一致，由于 Python 的变量没有数据类型，所以 Python 也没有数组，但是一个更为强大的列表就此诞生了。只需要中括号括起一堆数据，数据之间用逗号隔开就可以了。如果说数组是一个集装箱，那么我想列表就是一个工厂的仓库了。在本次编程中随处可见的列表以及它们多种多样的灵活应用就是很好的体现。

字典：类比《新华字典》，在 Python 中，字典的 key 就是新华字典中的字，value 就是字典对应的含义，Python 的字典在有些地方称之为 hash，有些地方称为关系数组。字典是 Python 中唯一的映射类型，映射是数学上的一个术语，指两个元素互相的对应关系。相比于列表，列表的索引和元素是没有关系的，但是字典却有，我们可以从基础代码中看到，比如 dict_side,dict_op_side.他们给 key 赋予了字面意思，给 value 赋予了实际操作，这样一来，我们想要知道我方的位置，就可以直接用 dict_side[‘position’]来表示，而不用调用 copy.copy(player.pos)这样较长的代码，用起来十分方便简单。

2.1.1 极大极小值搜索树

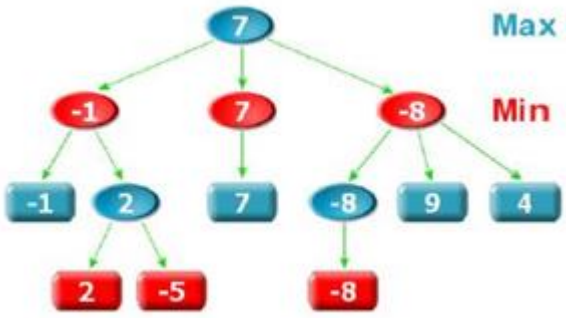


图 3 极大极小值搜索树示意图

极大极小值搜索树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自己的最大努力获得好结果。从一方来看，它的每一步行动总是在争取自己的最大利益，称为 MAX 过程，对应博弈树上的节点称为 MAX 节点，由于比赛过程中对对方的走法并不清楚，所以只能认为对方的每一下击球总是使耗损生命值最小，称为 MIN 过程，对应博弈树上的节点称为 MIN 节点，这样每一下击球都是 MAX 和 MIN 交替的过程。如果我方发现自己所有可能的行动派生的子节点中评估值最大的节点为 S'i，那么最大收益就为 V（S'i）-V（S），由此对应的行动就是单步搜索中所能找到的最优解。但如果不是单步搜索，也许会发现 S'所派生的子节点中的最小值为 V（S''i）小于其他节点所派生的子节点的最小值 V（S''j），这样可以设 S'j 对应的行动可作为自己的最佳行动，而这已经是最好的情况了。

但在极大极小搜索过程中，数据量多，会存在着一定程度上的冗余，减小冗余数据的量，是减小占用空间的必然做法，这样可以把 Alpha-Beta 剪枝算法应用到极大极小值搜索中，可以达到优化结构的作用。

2.1.2 Alpha-Beta 剪枝算法

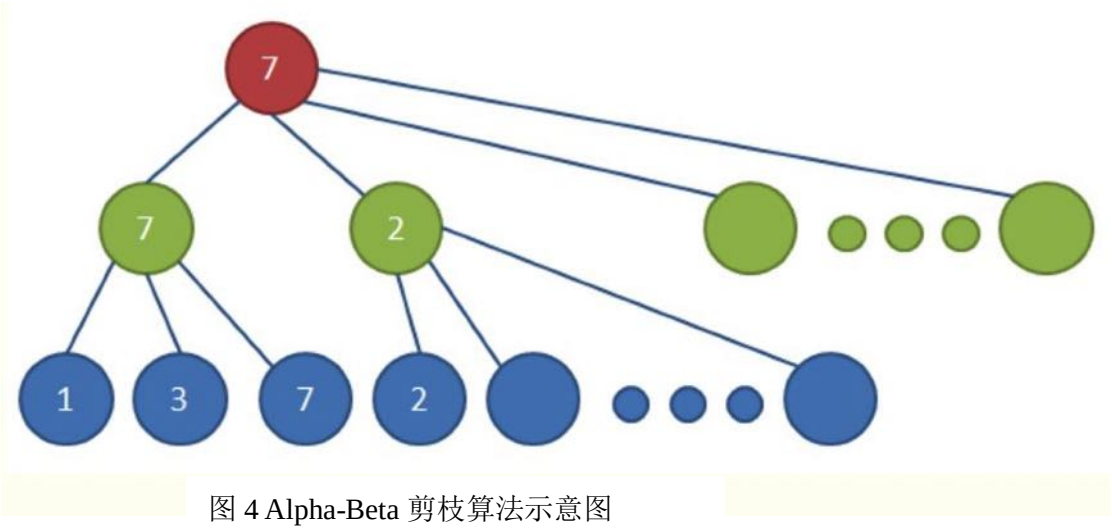


图 4 Alpha-Beta 剪枝算法示意图

Alpha-Beta 剪枝算法，是一个深度搜索算法，Alpha-Beta 剪枝的 Alpha 和 Beta 分别指的是 MAX 和 MIN 节点。因为预测的步数越多，生成的极大极小搜索树越庞大，消耗就会更大。为了提高搜索的效率，通过对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，达到优化。这样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。其将我方设为 MAX 方，因为它选择着法时总是对其子节点的评估值取最大值，即有利的着法，将对方设为 MIN 方，即选择对对方最为不利的着法。

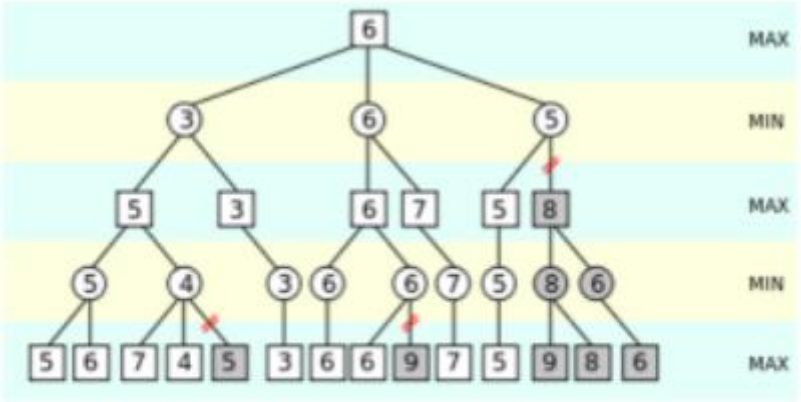


图 5 Alpha-Beta 剪枝算法示意图

如上图所示，在第二层，也就是 MIN 层，当计算到第三个节点的时候，已知前面有一

个 3 和一个 6，也就是最小值为 3。 在计算第三个节点的时候，发现它的第一个孩子的结果是 5，因为它的孩子是 MAX 节点，而 MAX 节点是会选择最大值的，那么此节点的值不会比 5 小，因此此节点的后序孩子就没有必要计算了，因为这个节点不可能小于 5，而同一层已经有一个值为 3 的节点了。其实这个图里面第三层分数为 7 的节点也是不需要计算的。这是 MAX 节点的剪枝，MIN 节点的剪枝也是同样的道理。

2.2 函数说明

2.2.1 模块说明

算法总共包含了 pygame、shelve、time、sys 四个模块，模块为包含并且有组织的代码片段，是一个保存了 Python 代码的文件。在 Python 中用关键字 import 来引入某个模块。

模块也是 Python 的对象，具有随机的名字属性用来绑定或引用。把相关的代码分配到一个模块里能让你的代码更好用，更易懂。下表为本算法包含的 pygame、shelve、time、sys 四个模块和几个常用的模块的简介。

表 1　算法模块简介

模块名称	模块简介
Pygame	Pygame 模块会自动导入其他 Pygame 模块，包括 Surface 函数和一些其他工具。它可以返回一个新的 Surface 对象。Surface 对象就是一个有确定尺寸的空图像，可以用来进行图像绘制和移动。常用的 pygame 模块如下： 1．pygame.locals 模块包括在自己的模块作用域内使用的名字（变量），还包括事件类型、键和视频模式等的名字。 2. pygame.display 模块包括处理 Pygame 显示方式的函数，其中包括普通窗口和全屏模式。 3．pygame.font 模块包括 Font 函数。字体对象用于表现不同的字体，可以用于将文本生成为可以在 Pygame 中用作普通图像的图像。 4. pygame.image 模块用于处理保存在 GIF、PNG 或 JPEG 等文件内的图像。pygame.image.load 函数用来读取图像文件，并且创建包括该图像的 Surface 对象。
shelve	shelve 类似于一个 key-value 数据库，可以很方便的保存 python 的内存对象，不像传统数据库一样，先取出数据，然后用这些数据重新构造一遍所需要的对象，而是在下次需要用的时候直接取出来。
time	time 有 2 种时间表示形式:

	1. 时间戳表示法。即以整型或浮点型表示的是一个以秒为单位的时间间隔。这个时间的基础值是从 1970 年的 1 月 1 号零点开始算起。 2. 元组格式表示法。即一种 Python 的数据结构表示。这个元组有 9 个整型内容。分别表示不同的时间含义。返回时间戳方式的函数主要有 time()，clock()等。Clock()用于返回当前程序的 CPU 执行时间。unix 系统始终返回全部运行时间；而 windows 从第二次开始都是以第一次调用此函数时的时间戳作为基准，而不是程序开始时间为基准。不接受参数。
sys	程序中使用的是 sys.argv 函数，作用是实现从程序外部向程序传递参数。在程序中判断了是否有参数传入。
os	os 模块是 Python 标准库中的一个用于访问操作系统功能的模块，os.listdir()函数获得指定目录中的内容，os.getcwd()函数获得当前的路径。
re	正则表达式（re）指定一个与之匹配的字符集合。本模块所提供的函数，将可以用来检查所给的字符串是否与指定的正则表达式匹配。 re.compile()用于把一个正则表达式 pattern 编译成正则对象，以便可以用正则对象的其他方法。其中正则对象会得到保留，这样在需要多次运用这个正则对象的时候，效率会有较大的提升。

2.2.2 基础代码相关说明

- 说明：该部分描述自编代码中涉及到的相关参数，以及调用的基础代码，大写部分例如 DIM、FACTOR_DISTANCE 等皆是基础代码中的实参，对此不多作赘述。蓝色部分为用到的基础部分相关代码，每段代码下方为用到的调用方法说明。

```
        dict_side = {
            'name': player.name,
            'position': copy.copy(player.pos),
            'life': player.life,
            'cards': copy.copy(player.card_box)}
        dict_op_side = {
            'name': player.name,
            'position': None if self.active_card[1] == CARD_DSPR else copy.copy(op_player.pos),
            'life': op_player.life,
            'cards': copy.copy(op_player.card_box),
            'active_card': self.active_card,
            'accelerate': None if self.active_card[1] == CARD_DSPR else
            (-1 if op_player.action.acc < 0 else 1),
            'run_vector': None if self.active_card[1] == CARD_DSPR else
            (-1 if op_player.action.run < 0 else 1)}
        dict_ball = {
            'position': copy.copy(self.ball.pos),
            'velocity': copy.copy(self.ball.velocity)}#copy模块知识
        dict_card = {
            'card_tick': self.card_tick,
            'cards': copy.copy(self.cards)}
```

tb.side['**cards**']：

Table 类中调用 side 字典中‘card’部分，浅拷贝 CardBox 类，返回我方道具箱信息。

tb.side['**life**']：Table 类中调用 side 字典中‘life’部分，返回该类中我方生命值信息。

tb.side['**position**']：Table 类中调用 side 字典中‘position’部分，返回该类中我方位置信息。

tb.op_side['**life**']：Table 类中调用 op_side 字典中‘life’部分，返回该类中敌方生命值信息。

tb.op_side['**position**']：Table 类中调用 op_side 字典中‘position’部分，返回该类中敌方位置信息。

tb.ball['**velocity**']：Table 类中调用 ball 字典中‘velocity’部分，浅拷贝该函数中 self.ball. velocity，返回我方球速度信息。

tb.ball['**position**']：Table 类中调用 ball 字典中‘position’部分，浅拷贝该函数中 self.ball.pos，返回我方球位置信息。

```
class RacketAction:  # 球拍动作
    def __init__(self, tick, bat_vector, acc_vector, run_vector, side, card):
        # self.t0 = tick  # tick时刻的动作，都是一维矢量，仅在y轴方向
        self.bat = bat_vector  # t0~t1迎球的动作矢量（移动方向及距离）
        self.acc = acc_vector  # t1触球加速矢量（加速的方向及速度）
        self.run = run_vector  # t1~t2跑位的动作矢量（移动方向及距离）
        self.card = (side, card)  # 对'SELF'/'OPNT'使用道具
```

最后需要返回的 RacketAction 函数信息，详见自编代码部分

2.2.3 自编代码部分

Serve 函数

- 说明：该函数为发球函数，总是做为 West 才发球，op_side 参数为对手方名称，以便从 ds 中查询以往记录数据。ds 参数为函数可以利用的存储字典，用以储存和对方战斗的信息 可作为发球依据使自己的算法不断变聪明。该函数需要返回球的 y 坐标，和 y 方向的速度。函数需要返回球的 y 坐标，和 y 方向的速度。
- 全代码：

```
# -*- coding: utf-8 -*-
import ...
# 发球函数，总是做为West才发球
# ds为函数可以利用的存储字典 储存和对方大战的信息 用于作为发球依据 可以使自己的算法不断变聪明
# 函数需要返回球的y坐标，和y方向的速度
def serve(op_side,ds):
    #发球策略：站在场地中间，节省一次跑位，随机一个速度发球
    pos=DIM[3]/2
    #计算速度的上下界
    top_speend=int((2*DIM[3]+pos)/(2*DIM[1]/BALL_V[0]))
    min_speend=int((pos)/(2*DIM[1]/BALL_V[0]))
    #随机速度方向
    speed_sign=random.randrange(-1, 2,2)
    speed=random.randint(min_speend,top_speend)
    return pos,speed*speed_sign
```

- 具体解释：

“pos=DIM[3]/2” 使球拍坐标处于中间位置

“top_speend=int((2*DIM[3]+pos)/(2*DIM[1]/BALL_V[0]))

min_speend=int((pos)/(2*DIM[1]/BALL_V[0]))”

“speed_sign=random.randrange(-1, 2,2)”随机速度方向

“speed=random.randint(min_speend,top_speend)”随机速度

以向上反弹为例：路线 ABC 的反弹可以用直线 ABC’表示；路线 ADEF 的反弹可以用直接 ADE’F 表示。（向下反弹方法亦然），以此计算最大速度和最小速度。

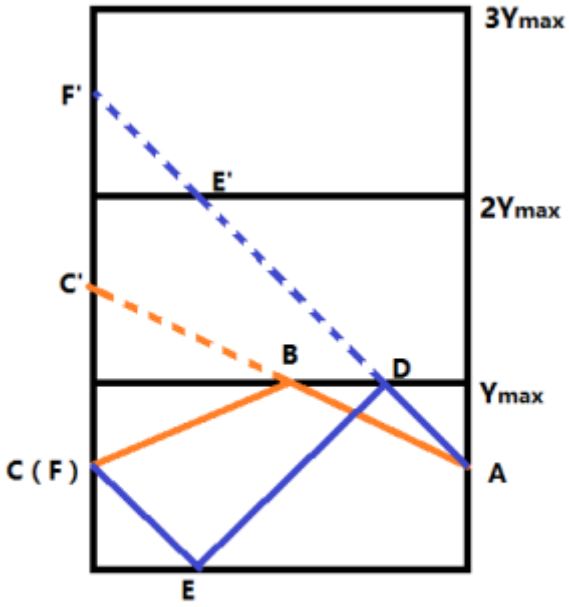


图 6 路径模式简化图

Play 函数

- 说明：该函数为打球函数，tb 参数为 TableData 类型的对象，为球桌态势，包含自身的位置，体力值，道具箱等信息，以及敌人的体力值，对方速度、跑位、道具、即将对我方实施的道具等信息，ds 参数为函数可以利用的存储字典，用以储存局部变量，函数需要返回一个 RacketAction 对象，包含了一个击球位置，对球实施的加速，以及跑位位置，对对方实施的道具。
- 全代码：

```python
# 打球函数
# tb为TableData类型的对象
# ds为函数可以利用的存储字典
# 函数需要返回一个RacketAction对象
def play(tb, ds):
    #贪心使用道具，有道具就用
    card_for_me=None
    card_for_enemy=None
    use_tp=0
    use_am=1
    if  len(tb.side['cards']) >0:#如果有道具
        active_card=tb.side['cards'][0].code#用
        if active_card==CARD_DSPR:# 隐身术：隐藏被用道具方的位置和跑位方向1次；param=0，位置乘以parm，方向乘以parm
            card_for_me=CARD_DSPR
        elif active_card==CARD_INCL:# 补血包：给被用道具方补血（增加体力值）；param=2000，life加上param
            card_for_me=CARD_INCL
        elif active_card==CARD_DECL:# 掉血包：给被用道具方减血（减少体力值）；param=2000，life减去param
            card_for_enemy=CARD_DECL
        elif active_card==CARD_TLPT:# 瞬移术：被用道具方可以移动一段距离而不消耗体力值；param=250000，只作用在迎球阶段
            card_for_me=CARD_TLPT
            use_tp=1
        elif active_card==CARD_AMPL:# 交压器：放大被用道具方的体力值损失；param=2；体力值损失增加1倍。
            card_for_enemy=CARD_AMPL
            use_am=2
    ##获取敌方信息和我方信息
    my_life=tb.side['life']
    not_know_enemy=0
    if tb.op_side!=None:
        enemy_life=tb.op_side['life']#敌军生命
        enemy_pos=tb.op_side['position'].y#敌军位置
    else:
        not_know_enemy=1#对方使用隐身卡等特殊情况无法获取敌军信息
    ball_speed=tb.ball['velocity'].y#当前球速度
    acc=0
    ##通过穷举加速度的可能性，按照她方当前位置和我方回球位置，计算每次我方loss和敌方loss，取最大值作为这次的击球策略。
    if not_know_enemy!=1:
        gain=0.0
        select_speed=0.0
        up_dis=DIM[3]-tb.ball['position'].y
        top_speed=int((2*DIM[3]+up_dis)/(2*DIM[1]/BALL_V[0]))
        min_speed=int(up_dis/(2*DIM[1]/BALL_V[0]))
        for speed in range(min_speed+1,top_speed-1):
            vir_dis=speed*(2*DIM[1]/BALL_V[0])-up_dis
            if vir_dis<DIM[3]:
                final_pos=DIM[3]-vir_dis#final_pos为敌军y坐标（距离下界的距离）
            else:
                final_pos=vir_dis-DIM[3]
            enemy_loss=abs(enemy_pos-final_pos)**2//FACTOR_DISTANCE ** 2*use_am
            my_loss=abs(speed-ball_speed)**2//FACTOR_SPEED ** 2
            if my_loss<my_life:##不能加速把自己加死
                continue
            tmp_gain=enemy_loss-my_loss
            if tmp_gain>gain:
                gain=tmp_gain
                select_speed=speed
        #向下发球
        down_dis=tb.ball['position'].y-DIM[3]
        top_speed=int((2*DIM[3]+down_dis)/(2*DIM[1]/BALL_V[0]))
        min_speed=int(down_dis/(2*DIM[1]/BALL_V[0]))
        for speed in range(min_speed+1,top_speed-1):
            vir_dis=speed*(2*DIM[1]/BALL_V[0])-down_dis
            if vir_dis<DIM[3]:
                final_pos=vir_dis #final_pos为
            else:
                final_pos=2*DIM[3]-vir_dis
            enemy_loss=abs(enemy_pos-final_pos)**2//FACTOR_DISTANCE ** 2*use_am
            my_loss=abs(speed-ball_speed)**2//FACTOR_SPEED ** 2
            if my_life<my_loss:
                continue
            tmp_gain=enemy_loss-my_loss
```

```
            if tmp_gain>gain:
                gain=tmp_gain
                select_speed=speed
        #确定最终的加速度
        acc=speed-ball_speed
    run=0
    #在出现55%的死角之前，不跑位，只根据敌方回球位置迎球。有死角之后，每次打完球回到正中间，这样回球的时候期望距离最小。
    if my_life<55000:
        run=500000-tb.ball['position'].y
        minus_life=(abs(run) ** 2 // FACTOR_DISTANCE ** 2)
        if minus_life>(my_life/2):
        print run,minus_life,my_life
            run=0
    print run,minus_life,my_life
    if use_tp:#使用了瞬移卡
        run=500000-tb.ball['position'].y
    print run
    if card_for_me==None and card_for_enemy==None:
        return RacketAction(tb.tick, float(tb.ball['position'].y - tb.side['position'].y), acc, run, None, None)
    elif card_for_me!=None:
        return RacketAction(tb.tick, float(tb.ball['position'].y - tb.side['position'].y), acc, run, tb.side, card_for_me)
    else:
        return RacketAction(tb.tick, float(tb.ball['position'].y - tb.side['position'].y), acc, run, tb.op_side, card_for_enemy)
```

- 道具部分相关代码解释说明：

card_for_me=None #对自身使用的道具

card_for_enemy=None #对敌人使用的道具

use_tp=0

use_am=1

if len(tb.side[**'cards'**]) >0: #如果有道具

active_card=tb.side[**'cards'**][0].code #用道具

if active_card==CARD_DSPR:# 隐身术

card_for_me=CARD_DSPR

elif active_card==CARD_INCL:# 补血包

card_for_me=CARD_INCL

elif active_card==CARD_DECL:# 掉血包

card_for_enemy=CARD_DECL

elif active_card==CARD_TLPT:# 瞬移术

card_for_me=CARD_TLPT

use_tp=1

elif active_card==CARD_AMPL:# 变压器

card_for_enemy=CARD_AMPL

use_am=2

- 发球部分相关代码解释说明：

my_life=tb.side['life']#从基础代码中的函数调取我方生命值

not_know_enemy=0#初始默认敌军信息可知

if tb.op_side!=None：

```
            enemy_life=tb.op_side['life']#获取敌军生命
            enemy_pos=tb.op_side['position'].y#获取敌军位置
        else:
            not_know_enemy=1#对方使用隐身卡等特殊情况无法获取敌军信息
        ball_speed=tb.ball['velocity'].y#当前我方球速度
        acc=0#加速度初始化为 0
#如果敌军信息可知
        if not_know_enemy!=1:
            gain=0.0
            select_speed=0.0
##向上发球部分
#球拍到上界的距离
            up_dis=DIM[3]-tb.ball['position'].y
#可执行的最大速度
            top_speed=int((2*DIM[3]+up_dis)/(2*DIM[1]/BALL_V[0]))
#可执行的最小速度
            min_speed=int(up_dis/(2*DIM[1]/BALL_V[0]))
#通过穷举加速度的可能性，按照地方当前位置和我方回球位置，计算每次我方 loss 和敌方 loss，取最大值作为这次的击球策略
#vir_dis 为从第一次碰壁起到终点这段时间 y 方向的路程
# my_loss、enemy_loss 分别是在某种假设情况下，我方和敌方的失血量，该计算公式由基础代码所给
#final_pos 为敌军 y 坐标（距离下界的距离）
# tmp_gain 为敌方与我方失血量之差
#gain、select_speed 分别为最大失血量之差以及最佳速度，在循环结束之前处于不断更新状态
            for speed in range(min_speed+1,top_speed-1):
                vir_dis=speed*(2*DIM[1]/BALL_V[0])-up_dis
                if vir_dis<DIM[3]:
                    final_pos=DIM[3]-vir_dis
                else:
                    final_pos=vir_dis-DIM[3]
                enemy_loss=abs(enemy_pos-final_pos)**2//FACTOR_DISTANCE ** 2*use_am
                my_loss=abs(speed-ball_speed)**2//FACTOR_SPEED ** 2
#注意自身加速是否会“自杀”
```

```
                if my_loss<my_life:
                    continue
                tmp_gain=enemy_loss-my_loss
                if tmp_gain>gain:
                    gain=tmp_gain
                    select_speed=speed
##向下发球部分
    #球拍到下界的距离
        down_dis=tb.ball['position'].y-DIM[3]
    #可执行的最大速度
        top_speed=int((2*DIM[3]+down_dis)/(2*DIM[1]/BALL_V[0]))
    #可执行的最小速度
        min_speed=int(down_dis/(2*DIM[1]/BALL_V[0]))
##同向上发球，穷举加速度的可能性，来获取最佳击球加速度
        for speed in range(min_speed+1,top_speed-1):
            vir_dis=speed*(2*DIM[1]/BALL_V[0])-down_dis
            if vir_dis<DIM[3]:
                final_pos=vir_dis #final_pos 为
            else:
                final_pos=2*DIM[3]-vir_dis
            enemy_loss=abs\
(enemy_pos-final_pos)**2//FACTOR_DISTANCE ** 2*use_am
            my_loss=abs(speed-ball_speed)**2//FACTOR_SPEED ** 2
            if my_life<my_loss:
                continue
            tmp_gain=enemy_loss-my_loss
            if tmp_gain>gain:
                gain=tmp_gain
                select_speed=speed
##确定最终的加速度
        acc=speed-ball_speed
```

- 跑位部分相关代码解释说明：

```
#初始化跑位距离
run=0
#在出现 55%的死角之前，不跑位，只根据敌方回球位置迎球。有死角之后，每次打完球回
```

到正中间，这样回球的时候期望距离最小。

```
    if my_life<55000:
        run=500000-tb.ball['position'].y
        minus_life=(abs(run) ** 2 // FACTOR_DISTANCE ** 2)
#如果移动需要损失的生命值超过当前生命值的一半，也不进行移动，节省体力。
        if minus_life>(my_life/2):
       print run,minus_life,my_life
            run=0
  print run,minus_life,my_life
#使用了瞬移卡的情况
    if use_tp:
        run=500000-tb.ball['position'].y
    print run
#使用了道具的情况
    if card_for_me==None and card_for_enemy==None:
        return RacketAction(tb.tick, float(tb.ball['position'].y - tb.side['position'].y), acc, run, None, None)
    elif card_for_me!=None:
        return RacketAction(tb.tick, float(tb.ball['position'].y - tb.side['position'].y), acc, run, tb.side, card_for_me)
    else:
        return RacketAction(tb.tick, float(tb.ball['position'].y - tb.side['position'].y), acc, run
```

3 实验结果

3.1 测试数据

实验环境说明：硬件配置为 I5-4200U/4G；操作系统为 win8；Python 版本为 2.7.3。

本次测试编写了一个 test.py 文件，该文件运行后可以进行 1000 次对局，其中对局双方为 west 和 east，且双方都采用 T_simple 算法，运行后可以得到这 1000 对局的数据信息：算法胜利次数，获胜次数中的三种获胜原因分别占有的次数，对局结束后双方平均剩余体力（生命），一次对局所用平均时间。

表 2 1000 次对弈结果数据汇总表

	获胜次数	因弹跳出错获胜次数	因体力耗尽获胜次数	因接球失败获胜次数	对局结束后剩余体力	平均一局所用时间
West	547	11	17	519	25401	79185tick
East	453	14	21	418	23173	

由数据可以看出，West 和 East 的胜利次数相差不大，接近 1:1，而双方获胜原因几乎都集中在因体力耗尽获胜，且双方因体力耗尽获胜次数占总次数的 90%。而对局结束后双方还剩余的体力都很多，说明输的那一方在还有大量体力没耗尽的情况下因为接球失败而落败，因此，T_simple 算法可能在处理接球上可能存在一些问题。另外，平均每局所用时间也较长。

3.2 结果分析

本次测试得出的结果如下：1.平均一局所用时间过长，建议优化算法，提高算法运行速度。2.由于获胜原因集中在接球失败上，可能是算法在接球处理上存在不足，建议优化算法对接球的处理，提高接球成功率。

3.3 热身赛结果分析

在热身赛中，我们的算法在对局过程中有失败也有胜利，胜利的原因大多是对手弹球次数出错，导致失败。而失败的原因大多是接球失败（miss_ball），这一点与测试的结果一致。整体上，失败的局数多于胜利，表明算法有待改进。

经过热身赛我们得到了宝贵的经验与教训，热身赛的结果再一次显示我们的算法在接球的处理上的短板，具体体现在与其他对手对战时经常因为接球失败而落败，因此加强算法在接球上的处理尤为关键。

因此我们利用自身对战数据进行可视化，进一步探究接球处理问题，发现了我们的算法在接球处理上存在的具体问题，主要体现为：当球进行最后一次弹射时，如果弹射点离两边边缘较远时，球拍能够稳定地接球。然而当弹射点离边缘很近时，球拍就不能接到球，进而导致失败。

由以下可视化对战的最终输球截图可以明显看出，未接到球的那一方都是在球在靠近墙壁边缘弹射后而未接到球导致失败。因此对于算法的优化可以考虑提前计算球的运动轨迹，当计算出球会在墙壁边缘进行弹射的情况时可以提前做出处理，从而避免因为球在靠接墙壁边缘接球失败的情况，进而提高接球率。

除此之外，算法在道具设计上采取了被动策略，即没有特意设计路线使球获得获取道具。

从可视化过程看到，道具利用率低，影响对战胜率，因此道具获取也是算法优化方向之一。

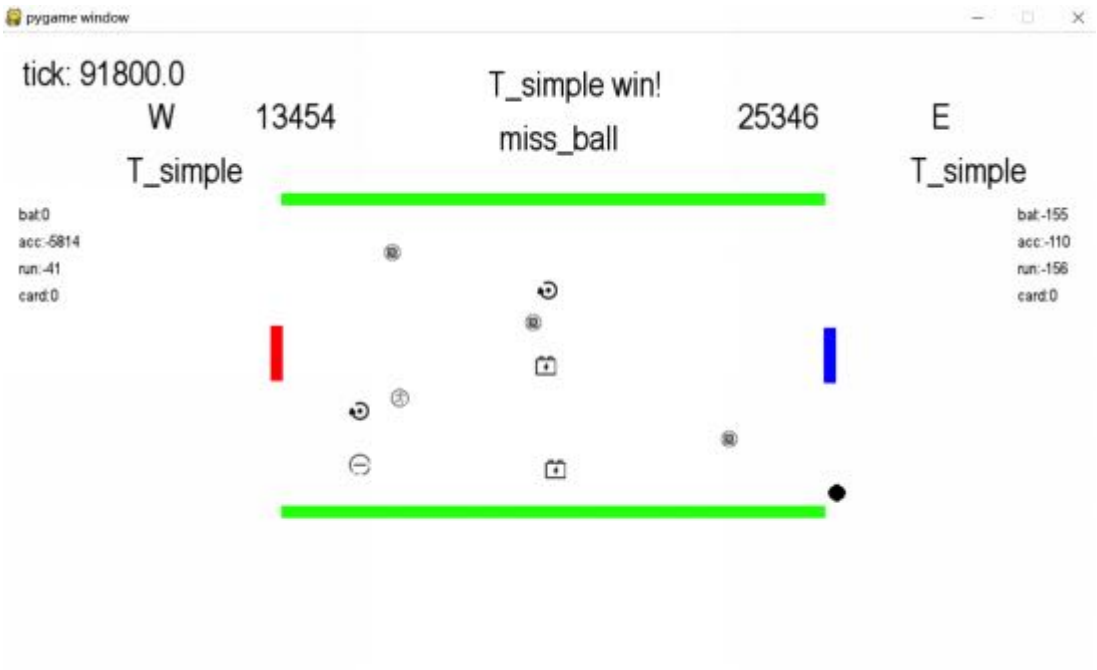


图 7 经典对战截图

3.4 测试程序算法

测试程序 test.py 的算法思路如下：

1. 建立一个 test 类，类包括 west_winnum、wmb_num、wib_num、wlo_num east_winnum、emb_num、eib_num、elo_num、time、west_life、east_life 等成员变量，这些成员变量分别代表着 west 和 east 胜利次数、胜利原因、胜利时双方剩余生命以及对局所用时间。

2. 定义一个 race 函数，函数功能与 pingpong 文件中的 race 函数类似，可以实现对局一次以及记录历史对战信息，另外该函数还增加了在对局结束后进行对局结果信息的累加处理。

3. 设置 while 循环执行 1000 次 race 函数，race 函数会将对局结果信息记录到 test 类的成员变量中，从而实现数据的记录。

4. 打印最终 1000 局对局的对战数据。

3.5 测试程序代码

```
#-*- encoding: utf-8 -*-
#注意由于本测试文件会运行1000次对局，所以最好将算法里和table文件中的print语句删除，防止大量输出信息造成测试速度下降
import ...
class test:
    def __init__(self):
        self.west_winnum = 0
        #记录West胜利次数
        self.wmb_num = 0
        # 记录West胜利原因为miss_ball的次数
        self.wib_num = 0
        # 记录West胜利原因为invalid_bounce的次数
        self.wlo_num = 0
        #记录West胜利原因为life_out的次数
        self.east_winnum = 0
        # 记录East胜利次数
        self.emb_num = 0
        # 记录East胜利原因为miss_ball的次数
        self.eib_num = 0
        # 记录East胜利原因为invalid_bounce的次数
        self.elo_num = 0
        # 记录East胜利原因为life_out的次数
        self.time = 0
        # 记录进行局数的所有时间
        self.west_life = 0
        #记录对局结束时West队所剩下的生命
        self.east_life = 0
        #记录对局结束时East队所剩下的生命

    def race(self,west_name, west_serve, west_play, west_summarize,
             east_name, east_serve, east_play, east_summarize, ):
        # 生成球桌
        main_table = Table()
        main_table.players['West'].bind_play(west_name, west_serve, west_play, west_summarize)
        main_table.players['East'].bind_play(east_name, east_serve, east_play, east_summarize)
        log = list()

        # 读取历史数据，文件名为"DS-<name>"
        for side in ('West', 'East'):
            d = shelve.open('DS-%s' % (main_table.players[side].name,))
            try:
                ds = d['datastore']
            except KeyError:  # 如果这个文件没有内容，说明球队尚未建立历史数据
                ds = dict()
            finally:
                d.close()
            main_table.players[side].set_datastore(ds)

        # 发球
        main_table.serve()

        # 开始打球
        while not main_table.finished:
            # 记录日志项
            log.append(LogEntry(main_table.tick,
                                RacketData(main_table.players[main_table.side]),
                                RacketData(main_table.players[main_table.op_side]),
                                BallData(main_table.ball),
                                CardData(main_table.card_tick, main_table.cards, main_table.active_card)))
            # 运行一趟
            main_table.time_run()

        # 记录最后的回合
        log.append(LogEntry(main_table.tick,
```

```
                    RacketData(main_table.players[main_table.side]),
                    RacketData(main_table.players[main_table.op_side]),
                    BallData(main_table.ball),
                    CardData(main_table.card_tick, main_table.cards, main_table.active_card)))

    # 终局，让双方进行本局总结
    main_table.postcare()

    # 最后，保存双方的历史数据，文件名为"DS-<name>"
    for side in ('West', 'East'):
        d = shelve.open('DS-%s' % (main_table.players[side].name,))
        d['datastore'] = main_table.players[side].datastore
        d.close()

    # 终局，保存复盘资料，文件名称记录了胜负及原因，对战双方名称，和选边
    d = shelve.open('[%s.%s]%s-VS-%s' % (PL[main_table.winner],
                                         RS[main_table.reason],
                                         west_name, east_name,))
    d['DIM'] = DIM
    d['TMAX'] = TMAX
    d['tick_step'] = main_table.tick_step
    d['West'] = west_name
    d['East'] = east_name
    d['tick_total'] = main_table.tick
    d['winner'] = main_table.winner
    d['reason'] = main_table.reason
    d['log'] = log
    d.close()

    # 每局数据记录并累加
    if main_table.winner == "West":
```

```
            self.west_winnum += 1
            if main_table.reason == "invalid_bounce":
                self.wib_num += 1
            if main_table.reason == "miss_ball":
                self.wmb_num += 1
            if main_table.reason == "life_out":
                self.wlo_num += 1
        if main_table.winner == "East":
            self.east_winnum += 1
            if main_table.reason == "invalid_bounce":
                self.eib_num += 1
            if main_table.reason == "miss_ball":
                self.emb_num += 1
            if main_table.reason == "life_out":
                self.elo_num += 1
        self.west_life += main_table.players['West'].life
        self.east_life += main_table.players['East'].life
        self.time += main_table.tick


import os

# 取得所有以T_开始文件名的算法文件名
players = [f[:-3] for f in os.listdir('.') if os.path.isfile(f) and f[-3:] == '.py' and f[:2] == 'T_']
i = 1000
t=test()
#进行1000次对局
while i>0:
 for west_name in players:
    for east_name in players:
        exec('import %s as WP' % (west_name,))
```

```
        exec('import %s as EP' % (east_name,))
        t.race(west_name, WP.serve, WP.play, WP.summarize, east_name, EP.serve, EP.play, EP.summarize)
        i = i - 1
#输出最终测试数据
print("in 1000 round,West win (%d)! for invalid_bounce (%d),for miss_ball (%d),for life_out (%d),\
East win (%d)! for invalid_bounce (%d),for miss_ball (%d),for life_out (%d),AVG_West:%s(%d) ,\
 AVG_East:%s(%d),平均时间: %d tick" %
        (t.west_winnum, t.wib_num,t.wmb_num,t.wlo_num,
         t.east_winnum,t.eib_num,t.emb_num,t.elo_num,
         west_name, t.west_life/1000,
         east_name, t.east_life/1000,t.time/1000))
```

4 实习过程总结

4.1 分工与合作

4.1.1 分工

编程组：张容榕 郑黛 吕品妍 邢仕杰

调试组：计启迪

报告组：郭晓彤 谷鹤 郑黛 张容榕 邢仕杰

4.1.2 组会记录

为了大作业任务的顺利进行，管理秩序的规范，和工作效率的提高，我们组考虑采用了这样多级而明确分工体系，同时有辅以线上及时讨论、线下组会交流的帮助，来构筑一个从上而下的小组分工体系。

在分工的过程中，我们主要分成了三个小组，编程组、调试组和报告组，而每个小组有都有自己的相关微信群，其中在做大作业的时候，为了使分工更明确，效率更高也交叉建立了各类微信群聊。在群里，每位组员都积极地讨论问题，都认真参与着每一项任务的完成，不分昼夜，从夕阳谈到朝阳，而其中每一个微小的成功也都饱含着每个人的努力和进步。



4.1.2.2. 组会讨论

单凭线上的讨论是明显不够的，小组也集中召开了两全组规模的线下组会，以及各小组

单独的小组会，而根据不完全的统计结果线下小组会的召开有十二次。下面主要对两次全体组员参与的大组会展开说明。

- 第一次组会记录：

时间：5.15

地点：理教 107

组会主要内容：

a) 互相认识，轮流自我介绍，同时介绍每个人的技能和长处，表决自己的分组意向；

b) 互相熟识并确立了分组，进行了初步的方法策略商讨；

c) 大组内的各小组进行相互认识介绍，同时针对不同任务设定不同的截止时间和初步规划。

d) 小组各成员登记信息，在组长处留下多种联系方式



- *第二次组会记录：*

时间：5.30

地点：泊星地咖啡厅

主要内容：

a) 各小组先对目前的工作阶段及所获进展进行总结；

b) 针对每一小组提出的存在的问题进行归纳和讨论，最后给出一致解决方案；

c) 每位组员针对程序思路提出建设意见，提供建议来完善算法结构；

d) 每一小组对之后的热身赛和正式比赛做准备，同时设置每一项工作的截止日期。



4.2 经验与教训

4.2.1 机器学习算法的失败

我们曾经试图做过机器学习算法，关于该算法，它是当今甚是流行的仿生算法，几个灰底圆中所标字母 w 代表浮点数，称为权重，进入人工神经细胞的每一个 input(输入)都与一个权重 w 相联系，正是这些权重将决定神经网络的整体活跃性。这是人工神经细胞激励函数的一种最简单的类型。

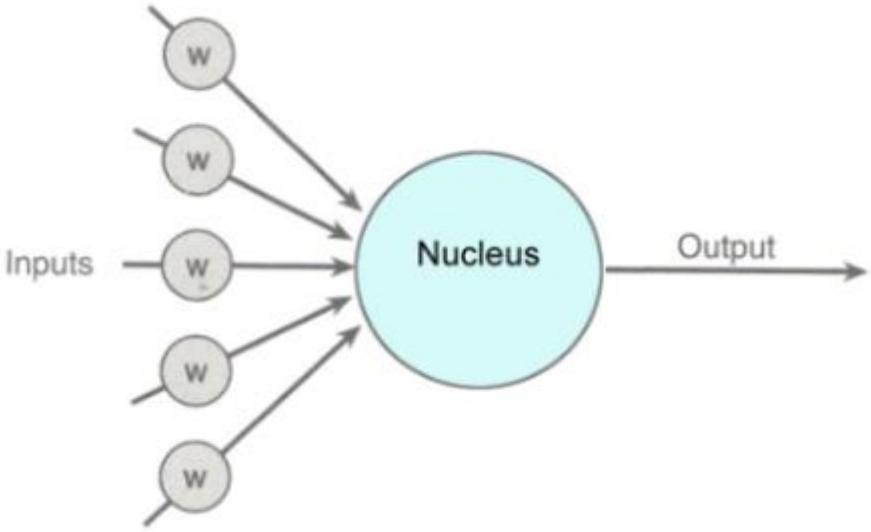


图 8 简单神经元模型

总之一个计算模型，要划分为神经网络，通常需要大量彼此连接的节点 （神经元），并且每个神经元，通过激励函数，计算处理来自其它相邻神经元的加权输入值，神经元之间的信息传递的强度，用所谓加权值来定义，算法还会不断自我学习，调整这个加权值。所以对于小规模的比赛，不能很好的发挥它的效果。

正因为如此，机器学习需要大量的对战经验，我们不能在短时内获得足够的资源，所以以机器学习算法作为主要策略实际上误差很大。当我们发现该算法效果并不能真正发挥神经网络算法的优越性时，我们选择了淘汰。

以下是机器学习的算法策略：

策略概述

前 6 轮和 6 轮之后的策略不同，这是由于 6 轮之后有机器学习（由于机器学习需要用到观测值拟合参数从而进行预测，所以前 6 轮如果用到机器学习可能有较大误差）。前 6 轮和 6 轮之后的策略不同之处有两点：1.前 6 轮假定对方击球后位置不同，而 6 轮之后根据机器学习结果预测对方击球后的跑位（不再假定对方位置不变）；2.前 6 轮我方迎球时回到球台中点，而 6 轮之后根据机器学习结果预测对方击球速度，从而预测对方击球后球的位置，我方迎球时跑到击球点和预测位置的中点（即不再跑到球台中点）。

前 6 轮的具体策略

1.接球策略：最大化对方的移动消耗和我方加速的消耗，并控制球反弹 1-2 次

$$Max \quad a_1[(v_0+v)t-y_1-y_2]^2-a_2v^2$$

$$s.t.y_1 \le (v_0+v)t \le y_1+w$$（用于控制球反弹次数）

其中 a1,a2 为系数，v0 是未击球时球的垂直速度，v 加速，t 每次球运行的时间，y1、

y2、w 的示意见下图。

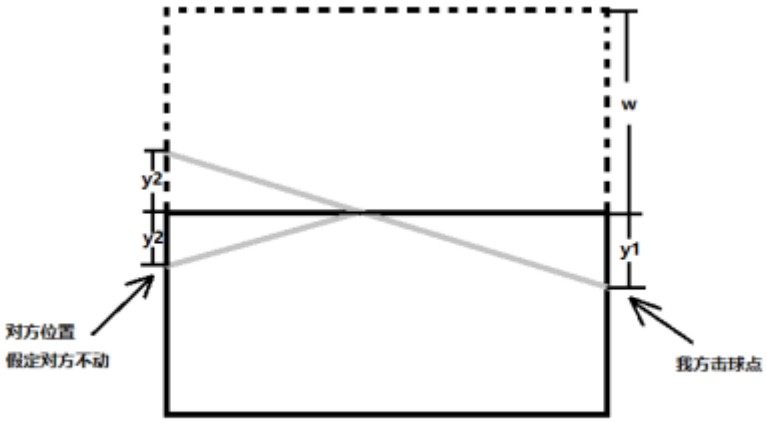


图 9 y1 y2 w 示意图

2.迎球策略（即我方发球后到对方接球前）：前 6 轮跑动到球台中点

6 轮之后的具体策略

预测 y_2 和 $v_{对方}$ 的方法：利用图片中已知的观测值分别对 y_2 和 $v_{对方}$ 作线性回归（或者其他回归），从而可以预测。

1.接球策略：最大化对方的移动消耗和我方加速的消耗，并控制球反弹 1-2 次（与前 6 轮不同的是 y2 来自预测值，而非假定不变）

$$Max \quad a_1[(v_0+v)t-y_1-y_2]^2-a_2v^2$$

$$s.t. y_1 \le (v_0+v)t \le y_1+w$$（用于控制球反弹次数）

2.迎球策略（即我方发球后到对方接球前）：根据预测的对方的球速 $v_{对方}$，预测对方击球后球的位置，从而在迎球时间里跑到击球点和预测位置的中点。

4.2.2 关于 Alpha-Beta 剪枝算法的缺陷

这个算法严重依赖于算法的寻找顺序，它是根据极大极小搜索规则进行的，但它如同极大极小一样，该算法最终也会找遍整个博弈树。如果程序总是能挑最好的着法来首先搜索，那么它的效率就会有所改善。

4.3 建议与设想

对课程建议：实习作业是一个愉快而有意义的过程，但关于分组，我们希望老师以后能够统计每次作业完成较好的人，从他们之中限定组长，避免学神扎堆或者是学渣集聚现象。

对学弟学妹的建议：面对大作业，一定要冷静下来认真分析算法的思想策略，可以分小

组负责不同的模块，最终进行整合。并且多吸取前人经验，多查资料，会有助于拓宽视野，给编程会带来极大的帮助。小组成员之间多讨论不仅可以可以给你新的启发，而且彼此会架起友谊的桥梁。另外老师每年的大作业都很新颖，相信完成它一定会让你们受益匪浅。

5 致谢

首先，感谢数据结构与算法课的主讲——陈斌老师，他给我们提供了一次好玩的实习机会，并始终鼓励着我们，陪伴着我们最终完成了一场难忘的算法大赛，在此也恭喜 Oscar 组获得了最终的胜利。

其次，感谢助教以及所有帮助设计的可视化对战程序同学，让我们从直观上感受到了乒乓球大战的激烈，从中也看出了四强组的算法模式精髓，让我们由衷的佩服，也惭愧自己算法的粗略。

虽然本次实习内容任务繁重，好在全组人员团结合作之下，完成了我们最终的算法大赛，虽然本组成绩比不上班里一些大神，但我们每个人都用心思考过，本组每个人都查阅了很多资料，无论是编程人员还是报告撰写者。在此感谢本组所有成员的辛勤努力和汗水。当然代码的顺利运行也要感谢所有帮助过我们的陈斌老师和助教团们，他们在算法的改进过程中提出了很多宝贵的意见，使我们的算法最终成熟。

在课程上，每次布置下来实习作业，陈老师、助教以及一些同学都会在群里为提问者耐心解答，有时候会忙到很晚，在此谢谢各位的辛苦指导。数算课让我们不仅收获了知识，更感受到了算法世界的奇妙之处，使我们认识了一门神奇的语言——Python，也让我们在编程思维的训练上有了很大突破。

在课程结束之末，我们由衷的感谢所有老师和同学们，是你们的精心点拨与热忱鼓励，使我们最终获得了人生中不可多得的宝贵经验。

6 参考文献

[1] 新浪博客.《极小极大搜索方法、负值最大算法和 Alpha-Beta 搜索方法》.http://www.cnblogs.com/pangxiaodong/archive/2011/05/26/2058864.html

[2]维基百科. https://zh.wikipedia.org

[3]脚本之家.www.jb51.net



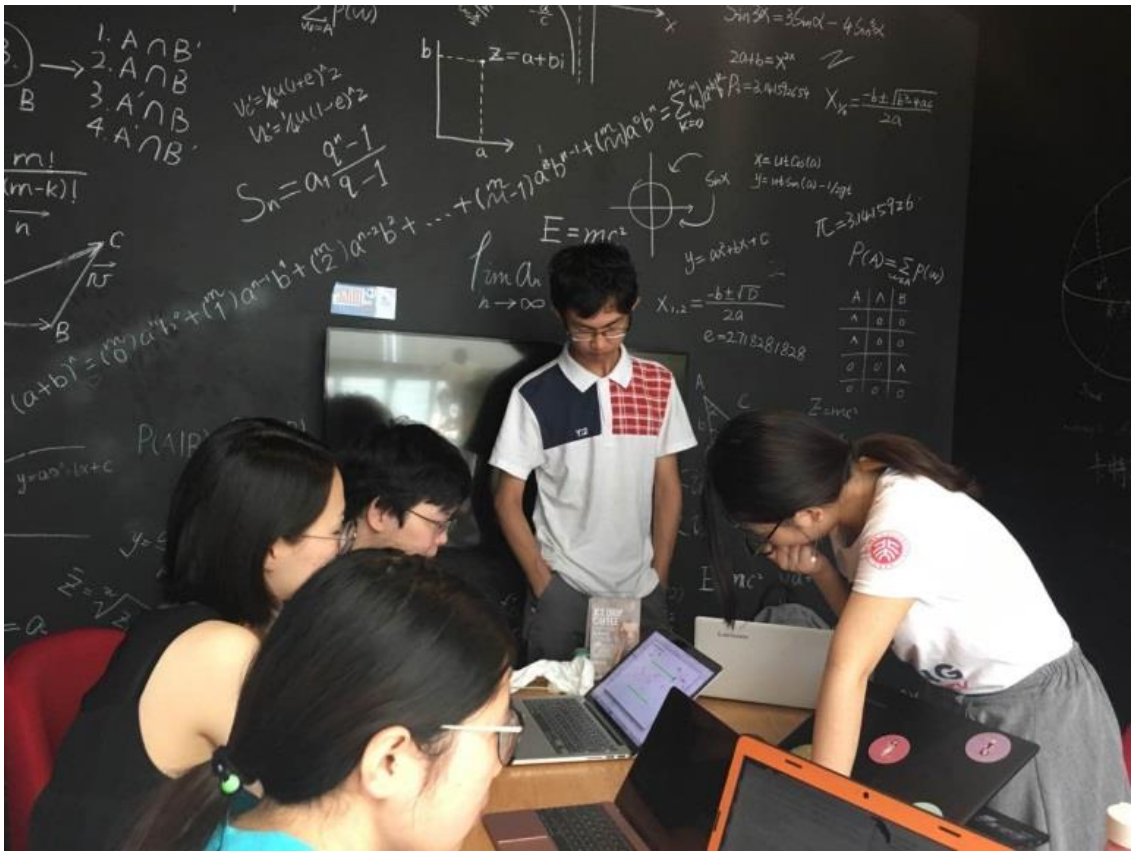
SIERRA

组长：张影

组员：黄殊晏，李沛泽，陈梦珂，高春芝，雷苗，杨阳

小组赛战绩：S3





Sierra

数据结构与算法课程实习作业报告

SIEERA 小组

（张影*，黄殊晏，李沛泽，陈梦珂，高春芝，雷苗，杨阳）

摘要：在乒乓球大战中，各组独立采用 Python 语言编写击球算法，通过与其他组的比赛确定输赢。由于这场比赛是不完全信息决策，双方互不知道对方的跑位和击球策略，因此本组先利用搜索树确定策略，又综合运用了线性数据结构以及多个函数库，考虑了道具使用、历史数据学习等多方面因素来编写算法。本报告就算法思想、程序代码、测试过程及实验过程总结等方面展开论述。

关键字：乒乓球大战 搜索树 线性数据结构 跑位 击球

1 算法思想

1.1 总体思路

我们分析具体问题认为赢球的关键在于解决以下三个问题：

1、 猜测对方跑位。

2、 道具的使用问题。

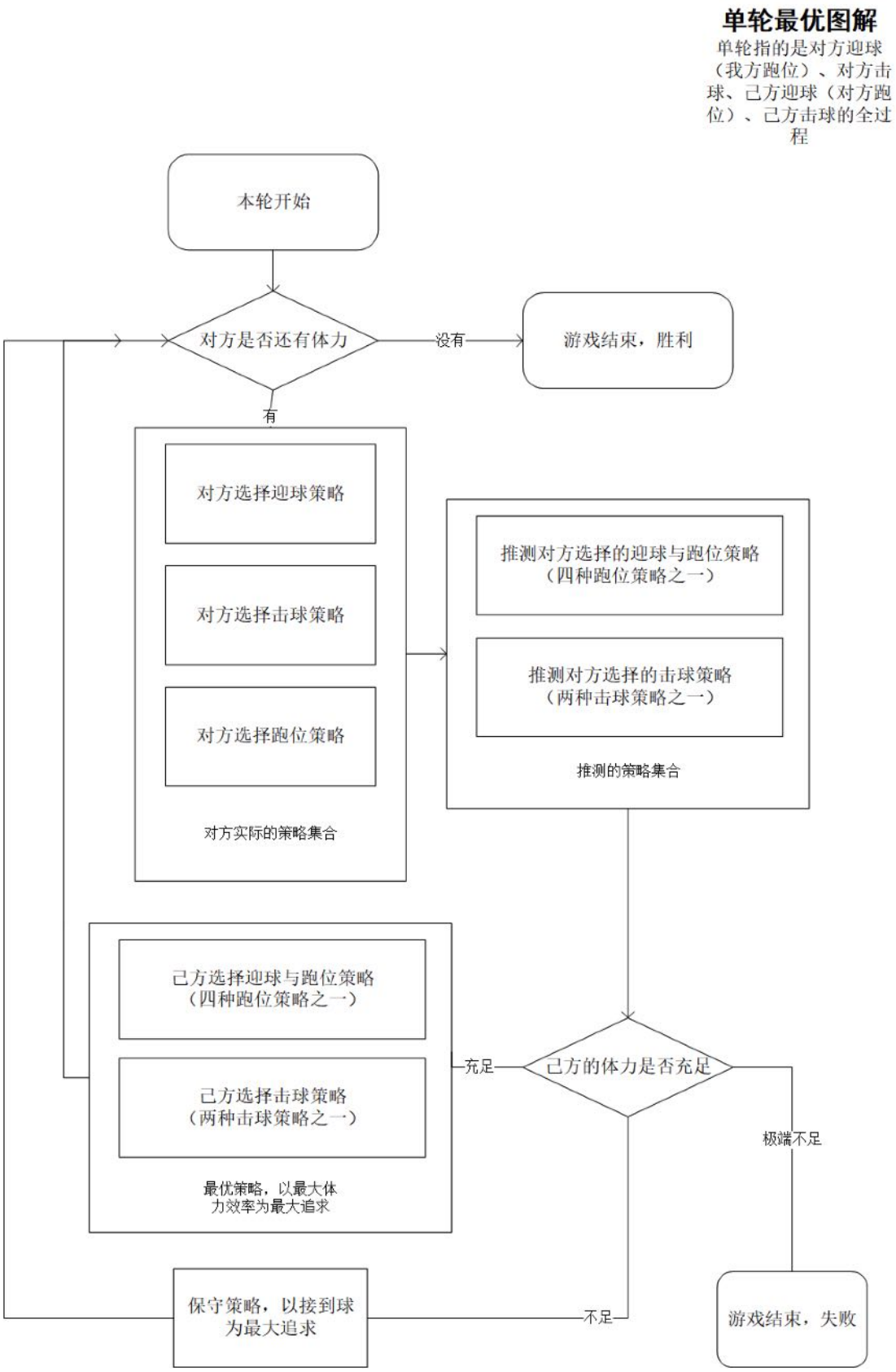
3、 如何最大限度的节省自己的体力并消耗对方的体能。

第一个问题，可以根据每局结束后对方的生命值损耗和对方的道具使用情况，来反推出对方的跑位，这就需要建立字典来存储每轮结束后对方的生命值损耗和道具使用情况，以供我们下一轮判断。

第二个问题，我们在最早的代码版本中，并未对道具的使用进行优化，只要球桌上有道具，我方就去取道具，取了道具就立即使用。在后续代码中对这一问题进行了优化，不同道具使用后的效果进行了一定的估计，并分别算出使用道具和不适用道具的收益，然后选出最有利于自己的决策。

对于第三个问题，我们根据博弈论的思想，假设双方都会做出理性决策，使自己的收益最大化，每一局都是这样，如果我们创建一个搜索树，树的节点保留每一回合双方交战的收益值之差（双方此回合消耗的生命值之差）。从树的底层开始搜索，如果每一层都选择我方受益最大的节点，由下往上类推到树的第一层，就可以得到我方博弈的一个最佳决策。搜索树的计算结果为我们的最终代码提供了很有意义的结果，譬如如何实现打角？我们最终代码的很多思想也都是基于搜索树。

1.2 算法流程图


单轮最优图解
单轮指的是对方迎球（我方跑位）、对方击球、己方迎球（对方跑位）、己方击球的全过程
本轮开始
对方是否还有体力
没有
游戏结束，胜利
有
对方选择迎球策略
对方选择击球策略
对方选择跑位策略
对方实际的策略集合
推测对方选择的迎球与跑位策略（四种跑位策略之一）
推测对方选择的击球策略（两种击球策略之一）
推测的策略集合
己方选择迎球与跑位策略（四种跑位策略之一）
己方选择击球策略（两种击球策略之一）
最优策略，以最大体力效率为最大追求
充足
己方的体力是否充足
极端不足
保守策略，以接到球为最大追求
不足
游戏结束，失败

1.3 算法运行时间复杂度分析

对 sierra 的时间复杂度分析：

由于我们已经基于 T-tree 估计出在展开的球桌上对方实现打脚的落点，所以我们在推测对方跑位计算我方回球的时候，用到一个 for i in range（6）的循环，虽然是循环，但是范围较小，所以时间损耗不大，基本上为 O（1）。

另外对于道具的计算中，我们用到了字典的遍历，时间复杂度为 O（1），单道具的最佳路径函数：single_besy（）中，又一次用到了 for i in range(leng)的循环，但是因为球桌上的道具不超过三个，所以 leng 也很小，时间复杂度也基本为 O（1）。

在跑位策略评估函数：scoretheo()和打击策略函数：scorehit()中，都存在一个两层的循环，但每个单层循环的循环次数都不超过 4，时间损耗上并没有很大

整体上，我们的算法使用了字典的遍历，还有一些循环搜索，但因为我们的循环次数很少，所以算法的计算速度仍然比较快。

2 程序代码说明

2.1 数据结构说明

所用到的基本数据结构有：

① 线性数据结构

② 树

所用到的类有：

① math

② Python 内置基本类（列表、元组、字典等）

③ time

④ scipy

⑤ numpy

⑥ table 内置类

2.2 函数说明

2.2.1Tree

为了找到一种消耗自己体力最少同时消耗对方体力最多的击球策略，我们小组使用了搜索树对双方击球策略进行了假设和分析。

我方使用三层树结构，设定 levels=2，树层数即为（levels+1）。我方考虑击球反弹的合法速度时，是将球台对称展开，一侧设置 8 个节点，即分为 9 份，一共是 18 份，18（branches）即为每层搜索分支数量。

```
from table import *

levels = 2  # levels+1 为树层数
divisions = 8
branches = (divisions + 1) * 2  # 每层搜索结果数
```

① targetdistribution(lst)函数

该函数生成 target 列表，用于存储 3 层的节点：

```
#  生成target列表
def targetdistribution(lst):
    # 3000000 ~ 1000000
    for i in range(divisions * 2):
        a = 2999999 - i*1000000/divisions
        lst.append(a)
    lst.append(1000001)
    # 0 ~ -2000000
    for i in range(divisions * 2):
        a = -1-i*1000000/divisions
        lst.append(a)
    lst.append(-1999999)
    return
```

② targettopos(tgt)函数

该函数用于将 target 转化为落点，返回 pos：

```
#  target转化为落点
def targettopos(tgt):
    pos = 0
    if 1000000 < tgt < 2000000:
        pos = 2000000 - tgt
    if 2000000 < tgt < 3000000:
        pos = tgt - 2000000
    if -1000000 < tgt < 0:
        pos = - tgt
    if -2000000 < tgt < -1000000:
        pos = tgt + 2000000
    return pos
```

③ targettovelocity(tgt, ori)函数

该函数用于将 target 转化为球 y 方向的速度，返回 vel：

```
def targettovelocity(tgt, ori):
    vel = 0
    if 1000000 < tgt < 2000000 or -2000000 < tgt < -1000000:
        vel = (tgt - ori) // 1800 + 1
    if 2000000 < tgt < 3000000 or -1000000 < tgt < 0:
        vel = (tgt - ori) // 1800
    return vel
```

④ callife(pos_a, pos_b, v_a, v_b)

该函数用于计算损失的生命值

```
def callife(pos_a, pos_b, v_a, v_b):
    lostlife = (pos_a - pos_b)**2 // FACTOR_DISTANCE**2 + (v_a - v_b)**2 // FACTOR_SPEED**2
    return lostlife
```

⑤ createtree(level, branch)

该函数通过递归生成搜索树

```
def createtree(level, branch):
    tree = list(range(branch + 1))
    if level == 0:
        tree = list(range(branch + 1))
    else:
        for i in range(1, branch + 1):
            tree[i] = createtree(level - 1, branch)  # 递归
    return tree
```

⑥ calleaf(tree, w_pos, e_pos, ball_v)

该函数通过输入搜索树、我方位置、对方位置、球速来计算底层叶值

```
def calleaf(tree, w_pos, e_pos, ball_v):
    for i in range(1, branches+1):  # 第一层
        tgt_a = targets[i-1]
        pos_a = targettopos(tgt_a)
        vdis_a = targettovelocity(tgt_a, w_pos)
        lifegain_a = callife(pos_a, e_pos, 0, 0) - callife(0, 0, vdis_a, ball_v)
        if -1000000 < tgt_a < 2000000:
            vdis_a = -vdis_a
        for j in range(1, branches+1):  # 第二层
            tgt_b = targets[j-1]
            pos_b = targettopos(tgt_b)
            vdis_b = targettovelocity(tgt_b, e_pos)
            lifegain_b = -callife(pos_b, w_pos, 0, 0) + callife(0, 0, vdis_b, vdis_a)
            if -1000000 < tgt_b < 2000000:
                vdis_b = -vdis_b
            for k in range(1, branches+1):  # 第三层
                tgt_c = targets[k-1]
                pos_c = targettopos(tgt_c)
                vdis_c = targettovelocity(tgt_c, pos_b)
                lifegain_c = callife(pos_c, pos_a, 0, 0) + callife(0, 0, vdis_c, vdis_b)
                tree[i][j][k] = lifegain_a + lifegain_b + lifegain_c
```

⑦ processminimax(tree)函数

该函数通过输入计算好叶值的搜索树，输出我方落点编号和对方落点编号，并进行 minimax 比较。如果将双方比作 A 和 B 两个人玩游戏，其中 A 希望能在游戏中获得最大值，B 希望获得最小值，双方进行博弈。在三层树结构中，底层是我方决策，取最大值；中层是对方决策，取最小值；上层仍是我方决策，取最大值。最后，输出我方最佳决策和相应的对方最佳决策，其中对方最佳决策用来判断跑位。

```
def processminimax(tree):
    # 底层：我方决策，取最大值
    for i in range(1, branches+1):
        for j in range(1, branches+1):
            tree[i][j][0] = -1000000
            for k in range(1, branches+1):
                if tree[i][j][0] < tree[i][j][k]:
                    tree[i][j][0] = tree[i][j][k]
    run_lst = list()
    run_lst.append(0)
    # 中层：对方决策，取最小值
    for i in range(1, branches+1):
        tree[i][0] = 1000000
        run_lst.append(0)
        for j in range(1, branches+1):
            if tree[i][0] > tree[i][j][0]:
                tree[i][0] = tree[i][j][0]
                run_lst[i] = j
    tree[0] = -1000000
    decision = 0
    # 上层，我方决策，取最大值
    for i in range(1, branches+1):
        if tree[0] < tree[i][0]:
            tree[0] = tree[i][0]
            decision = i
    return decision, run_lst[decision]
    # 输出我方最佳决策和相应的对方最佳决策，对方最佳决策用来判断跑位
```

创建 targets 列表以及 serve(op_side: str, ds: dict)打球函数：

```
# 创建targets列表
targets = list()
targetdistribution(targets)


def serve(op_side: str, ds: dict) -> tuple:
    return BALL_POS[1], BALL_V[1]


def summarize(tick: int, winner: str, reason: str, west: RacketData, east: RacketData, ball: BallData, ds: dict):
    return
```

⑧ play(tb, ds)

打球函数，通过创建搜索树、算出叶值来决定策略，并输出最佳策略。

```
def play(tb, ds):

    # 取出当前数据
    #   球落点
    w = tb.ball['position'].y
    #   原球速
    v = tb.ball['velocity'].y
    #   我方位置
    w_last = tb.side['position'].y
    #   对方位置
    e = tb.op_side['position'].y

    # 创建搜索树
    minimax = createtree(levels, branches)
    # 搜索树算出叶值
    calleaf(minimax, w, e, v)
    # 搜索树决定策略
    throw_choice, run_choice = processminimax(minimax)
    # 输出策略-【由于树中列表的决策编号从1开始，比targets中编号大1，故减去1】
    w_target = targets[throw_choice - 1]
    w_run = (targettopos(targets[run_choice - 1]) + w) // 2
    w_v = targettovelocity(w_target, w)

    return RacketAction(tb.tick, w - w_last, w_v - v, w_run - w, None, None)
```

经过测试，我们小组发现最佳击球策略是打对方死角，而对方最佳策略也是打我方死角，所以我方最佳跑位就是往中心跑。

但我们小组写搜索树和初版代码 T_young 的进程是同时展开的，重新审视搜索树的测试结果，我们发现搜索树 debug 之后打出来的球和 T_young 的结果几乎没有区别，只会打边线，走位走中间，反而有时候有少量极端的球的话会使其败给 T_young，因此我们在第二版 T_excited 代码中并未加上搜索树。

2.2.2Excited

①. serve(op_side: str, ds: dict)

发球函数，该函数返回球的 y 坐标和 y 方向的速度，在我们小组的代码中，每次发球返回固定的 y 坐标和 y 方向的速度。

```
from table import *
from scipy.optimize import fsolve

# 发球函数，总是做为West才发球
# ds为函数可以利用的存储字典
# 函数需要返回球的y坐标，和y方向的速度
def serve(op_side: str, ds: dict) -> tuple:
    '''发球，打到对方角上'''
    return 500100, 278
```

②. play(tb,ds)

打球函数，tb 为 TableData 类型的对象，ds 为函数可以利用的存储字典，函数返回一个 RacketAction 对象。

首先传入一些变量，比如接球时球的 y 方向速度、球桌高度、接球时的 x 方向位置等等。

```
def play(tb,ds):

    '''一些变量'''
    vb0 = tb.ball['velocity'].y #接球时球的y方向速度
    Y = DIM[3]                  #球桌高度
    xb0 = tb.ball['position'].x #接球时的x方向位置
    yb0 = tb.ball['position'].y #接球时的y方向位置
    t = tb.step                 #一轮的时间长度
    yp0 = tb.op_side['position'].y #对方跑位前y方向位置
```

在下图的道具使用部分，首先获取本轮正在起作用的道具，然后确定下一轮使用的道具，默认只要有道具就立即使用道具箱里第一个道具。

```
'''道具使用部分'''
card_code = tb.op_side['active_card'][1]
if tb.side['cards']:
    next_card_code = tb.side['cards'][0]
else:
    next_card_code = None
```

③. runtheo(theo_num, pos)

该函数用于根据历史数据猜测对方跑位，我方猜测对方可能的 5 种策略，分别是不跑位、跑底线中点、跑迎球中点、跑迎球 2/3 处和跑到离中点 2000 的位置，并应用到之后的击球策略中。

```
'''根据历史数据猜测对方跑位'''
# by lpz-跑位策略集(对方几种可能的跑位策略)-输入跑位编号和跑位前位置，输出跑位后位置
def runtheo(theo_num, pos):
    # 策略1：不跑位
    if theo_num == 0:
        return pos
    # 策略2：跑底线中点
    if theo_num == 1:
        return Y/2
    # 策略3：跑迎球中点
    if theo_num == 2:
        if pos < Y/2:
            run = (pos + Y) // 2
        else:
            run = pos // 2
        return run
    # 策略4：跑迎球2/3处
    if theo_num == 3:
        if pos < Y/2:
            return int(Y / 3 + pos * 2 / 3)
        return pos * 2 // 3
    # 策略5：跑到离中点2000的位置
    if theo_num == 4:
        if pos < Y/2:
            return Y/2 - 2000
        return Y/2 + 2000
```

④. scoretheo(poslst, runlst)

该函数用于对跑位策略进行评估，通过输入历史击球点和历史跑位，输出最佳跑位编号。我们小组设定了样本数为 9，若历史数据不够样本数，则历史数据中有一个 None 不可用，故 samplenum 加一。评估之后，储存各个策略打分(策略跑位和实际跑位之差的绝对值之和)，分数越低，策略表现越好。

```
# by lpz-跑位策略评估-输入历史击球点和历史跑位，输出最佳跑位编号
def scoretheo(poslst, runlst):
    totaltheo = 5  # 跑位策略总数
    samplenum = 9  # 取最近的样本数
    if len(runlst) < samplenum + 1:
        samplenum = len(runlst) - 3
    scores = list(range(totaltheo))
    for i in range(totaltheo):
        for j in range(samplenum):
            if poslst[-3 - j] and runlst[-1 - j]:
                # 第-3个击球点决定第-1个跑位
                scores[i] += abs(runtheo(i, poslst[-3 - j]) - runlst[-1 - j])
    return scores.index(min(scores))
```

对战初期，数据不够时，我方先不用历史数据猜对方跑位策略，而是主观进行猜测。如果对方使用隐身术，我方首先根据其位置猜一个跑位方向，进而猜测对方跑位策略。而当数据足够多时，则调用 scoretheo 函数猜测对方跑位策略，再根据跑位策略猜出对方本轮跑位后位置。

```
if tb.tick < 25200: #刚开始数据不够，先不用历史数据猜跑位
    if not tb.op_side['run_vector']:
        if yp0 < Y / 2:
            tb.op_side['run_vector'] = 1
        else:
            tb.op_side['run_vector'] = -1
    if tb.op_side['run_vector'] == 1:
        if yp0 < Y / 5:
            yp1 = Y / 2 - 100
        else:
            yp1 = int(Y / 3 + yp0 * 2 / 3)
    else:
        if yp0 > Y * 4 / 5:
            yp1 = Y / 2 + 100
        else:
            yp1 = int(yp0 * 2 / 3)
else:
    runtheonum = scoretheo(ds['op_pos'], ds['op_run_dis'])
    yp1 = runtheo(runtheonum, yp0) # 根据跑位策略猜出对方本轮跑位后位置
```

下面是我方击球部分，由于击球必须反弹 1-2 次，因此需要计算我方合法的速度范围。

```
'''计算合法的速度范围'''
v0up = (Y-yb0)/t       #向上，弹一次的速度下限
v1up = (2*Y-yb0)/t     #向上，弹一次的速度上限
v2up = (3*Y-yb0)/t     #向上，弹两次的速度上限
v0down = -yb0/t        #向下，弹一次的速度下限
v1down = -(yb0+Y) / t #向下，弹一次的速度上限
v2down = -(yb0+2*Y) / t #向下，弹两次的速度上限
range_up0 = list(range(int(v0up)+1,int(v1up)+1)) #向上弹一次的速度范围
range_up1 = list(range(int(v1up)+1,int(v2up)))    #向上弹两次的速度范围
range_down0 = list(range(int(v1down),int(v0down)))#向下弹一次的速度范围
range_down1 = list(range(int(v2down)+1, int(v1down)))#向下弹两次的速度范围
```

然后在所有合法速度中选取最节省自己体力同时最消耗对方体力的速度，此处需要注意的一点是：当前如果使用旋转球，发球时的速度要除以一个系数。

```
'''在所有合法速度中选取最节省自己体力同时最消耗对方体力的速度'''
def y1_up0(vy): #向上弹一次时，球到对面的落点y坐标与球y方向速度vy的函数关系
    return 2*Y - t*vy - yb0
def y1_up1(vy): #向上弹两次时，球到对面的落点y坐标与球y方向速度vy的函数关系
    return -2*Y + t*vy + yb0
def y1_down0(vy):#向下弹一次时，球到对面的落点y坐标与球y方向速度vy的函数关系
    return  -t * vy - yb0
def y1_down1(vy):#向下弹两次时，球到对面的落点y坐标与球y方向速度vy的函数关系
    return  t * vy + yb0 + 2*Y
spin_param = CARD_SPIN_PARAM if card_code == CARD_SPIN else 1
```

下面是一个最优化目标函数，即最小化自己的体力消耗减去对方下轮接球消耗的体力。然后对每个范围分别求最优的 vy 和对应的目标函数值，并在四个范围的 vy 最优值中选取目标函数值最大的一个。需要注意的一点是：在角上接球时，只允许弹一次。

```
#最优化目标函数：最小化自己的体力消耗减去对方下轮接球消耗的体力
def value_fun_up0(vy): #向上弹一次时的目标函数
    return ((abs(vb0-vy)/spin_param)**2// FACTOR_SPEED**2) * (CARD_AMPL_PARAM if card_code == CARD_AMPL else 1) - \
           (abs(yp1-y1_up0(vy))**2 // FACTOR_DISTANCE**2) * (CARD_AMPL_PARAM if next_card_code == CARD_AMPL else 1)
def value_fun_up1(vy): #向上弹两次时的目标函数
    return ((abs(vb0-vy)/spin_param)**2// FACTOR_SPEED**2) * (CARD_AMPL_PARAM if card_code == CARD_AMPL else 1) - \
           (abs(yp1-y1_up1(vy))**2 // FACTOR_DISTANCE**2) * (CARD_AMPL_PARAM if next_card_code == CARD_AMPL else 1)
def value_fun_down0(vy): #向下弹一次时的目标函数
    return ((abs(vb0-vy)/spin_param)**2// FACTOR_SPEED**2) * (CARD_AMPL_PARAM if card_code == CARD_AMPL else 1) - \
           (abs(yp1-y1_down0(vy))**2 // FACTOR_DISTANCE**2) * (CARD_AMPL_PARAM if next_card_code == CARD_AMPL else 1)
def value_fun_down1(vy): #向下弹两次时的目标函数
    return ((abs(vb0-vy)/spin_param)**2// FACTOR_SPEED**2) * (CARD_AMPL_PARAM if card_code == CARD_AMPL else 1) - \
           (abs(yp1-y1_down1(vy))**2 // FACTOR_DISTANCE**2) * (CARD_AMPL_PARAM if next_card_code == CARD_AMPL else 1)

min_vy_up0 = min(range_up0, key=value_fun_up0)
min_vy_up1 = min(range_up1, key=value_fun_up1)
min_vy_down0 = min(range_down0, key=value_fun_down0)
min_vy_down1 = min(range_down1, key=value_fun_down1)
min_val_up0 = value_fun_up0(min_vy_up0)
min_val_up1 = value_fun_up1(min_vy_up1)
min_val_down0 = value_fun_down0(min_vy_down0)
min_val_down1 = value_fun_down1(min_vy_down1)
if yb0 == Y or yb0 == 0:
    min_dict = {min_val_up0:min_vy_up0, min_val_down0:min_vy_down0}
else:
    min_dict = {min_val_up0:min_vy_up0, min_val_up1:min_vy_up1, min_val_down0:min_vy_down0, min_val_down1:min_vy_down1}
vy_best = min_dict[min(min_dict)]
```

⑤. hit_card_vy(card_pos,self_pos)

该函数用于吃道具，对于某一个桌上的道具，返回能够打到道具的所有球 y 方向速度，并对于球 y 方向速度 vy，计算目标函数值。

```
'''吃道具部分'''
def hit_card_vy(card_pos,self_pos):
    hit_vy = []
    xtime = abs(card_pos.x-self_pos.x)/BALL_V[0]
    hit_vy.append(int((card_pos.y-self_pos.y)/xtime))
    hit_vy.append(int((-card_pos.y - self_pos.y) / xtime))
    hit_vy.append(int((2*Y - card_pos.y - self_pos.y) / xtime))
    hit_vy.append(int((card_pos.y - 2*Y - self_pos.y) / xtime))
    hit_vy.append(int((2*Y + card_pos.y - self_pos.y) / xtime))
    return hit_vy

def cal_value(vy):
    if vy in range_up0:
        return value_fun_up0(vy)
    if vy in range_up1:
        return value_fun_up1(vy)
    if vy in range_down0:
        return value_fun_down0(vy)
    if vy in range_down1:
        return value_fun_down1(vy)
    return None
```

当球桌上有道具时，计算取每个道具的最优 vy，决定是否取道具以及取哪个道具。

```
if tb.cards['cards']:
    val_cards = {}
    for card_choice in tb.cards['cards']:
        card_pos = card_choice.pos
        valid_vy_val = {} #一个字典，存放可能打到该道具的合法速度及对应的目标函数值，value是vy，key是目标函数值
        for v in hit_card_vy(card_pos,tb.ball['position']):
            val = cal_value(v)
            if val:
                valid_vy_val[val]=v
        val_hit_best = None
        vy_hit_best = None
        if valid_vy_val: #合法速度不为空时，选出目标函数值最小的一个作为这个道具的最优速度
            val_hit_best = min(valid_vy_val)
            vy_hit_best = valid_vy_val[val_hit_best]
            if val_hit_best:
            tolerance = 2000 if card_choice == CARD_INCL or card_choice == CARD_DECL else 1000
            if card_choice == CARD_DSPR:
                tolerance = 0
            val_cards[val_hit_best-tolerance]=(vy_hit_best,card_choice)
    if val_cards and min(val_cards)<min(min_dict):
        vy_best = val_cards[min(val_cards)][0]
        tb.cards['cards'].remove(val_cards[min(val_cards)][1])
```

打道具一般而言会增大目标函数值(与不打道具的最优解相比)，因此需要对道具设置增大目标函数值的容忍度。但由于不同道具具有不同的效果，因此我方对道具设置了不同的容忍度。然后将道具、最优速度存入以对应目标函数值为 key 的字典，如果字典中最小的目标函数值小于不打道具的最优解，就可以取道具，并将选出的道具从当前桌上道具列表中删除。

⑥. decide_run(yb0)

跑位函数，如果本轮是会出现道具的上一轮，则跑到不到底线中点 20%处；如果本轮取完道具后，球桌上还有比较有价值的道具，就跑到不到底线中点 20%处；其他情况，一律跑到底线中点处。

```
'''跑位部分'''
def decide_run(yb0):
    card_tick = [int(3600*2.5*k) for k in range(1,16)] #会出现道具的时间
    for t in card_tick:
        if (tb.tick == t-1800 and tb.side['life']>55000):
            return int(0.8*(Y / 2 - yb0))
    if (CARD_INCL in tb.cards['cards'] or CARD_DECL in tb.cards['cards'] or CARD_TLPT in tb.cards['cards']
        or CARD_AMPL in tb.cards['cards']) and tb.side['life']>55000:
        return int(0.8 * (Y / 2 - yb0))
    return Y / 2 - yb0
```

⑦. cal_ynext(vy)

该函数用于计算球下一轮球的落点，并计算下轮对方接球时球的 y 方向速度以及上轮对方打出球时球的 y 方向速度。

```
def cal_ynext(vy):
    if vy in range_up0:
        return y1_up0(vy)
    if vy in range_up1:
        return y1_up1(vy)
    if vy in range_down0:
        return y1_down0(vy)
    if vy in range_down1:
        return y1_down1(vy)
    return None

op_in_vy = -vy_best if vy_best in range_down0 or vy_best in range_up0 else vy_best
op_out_vy = -vb0 if -vb0 in range_down0 or -vb0 in range_up0 else vb0
```

下面的部分是在字典中存储相关历史数据，如下图所示：

```
        '''存储数据'''
        if tb.tick == 0:
            '''当对方发球时，初始化'''
            ds["op_life"] = [100000]  # 该轮对方跑位前的体力值
            ds["op_pos"] = [Y / 2,cal_ynext(vy_best)]  # 该轮对方接球位置，多存一轮（对方下一轮的接球位置是我们本轮决定的）
            ds["op_in_vy"] = [None, op_in_vy]  # 该轮对方接到球的y方向速度，同理多存一轮
            ds["op_out_vy"] = [op_out_vy]  # 该轮对方打出球的y方向速度
            ds["op_run_dir"] = [None]  # 该轮对方跑位方向
            ds["op_card"] = [None]  # 该轮对方使用的道具
            ds["my_card"] = [None]  # 该轮我方使用的道具
            ds["y_ball"] = [yb0]  # 该轮我方接球位置
            ds["op_run_pos"] = [None]  # 上轮对方跑位后位置，需要计算
        elif tb.tick == tb.step:
            '''当我方发球时，初始化'''
            ds["op_life"] = [tb.op_side['life']]
            ds["op_pos"] = [999900,cal_ynext(vy_best)]
            ds["op_in_vy"] = [278, op_in_vy]
            ds["op_out_vy"] = [op_out_vy]
            ds["op_run_dir"] = [tb.op_side['run_vector']]
            ds["op_card"] = [None]
            ds["my_card"] = [None]
            ds["y_ball"] = [yb0]
            ds["op_run_pos"] = [None]
        else:
            ds["op_life"].append(tb.op_side['life'])
            ds["op_pos"].append(cal_ynext(vy_best))
            ds["op_in_vy"].append(op_in_vy)
            ds["op_out_vy"].append(op_out_vy)
            ds["op_run_dir"].append(tb.op_side['run_vector'])
            ds["op_card"].append(tb.op_side['active_card'][1])
            ds["my_card"].append(next_card_code)
```

我方可以根据对方上一轮的体力消耗计算其上轮跑位。因为对方上一轮和本轮的体力差，包括跑位、迎球、击球消耗的体力，已知我方击球速度和对方迎球打回来的速度，可以计算对方击球消耗的体力，将其减去之后，是对方跑位和迎球消耗的体力。

```
#基于对方上一轮的体力消耗计算其上轮跑位
life_dec = ds["op_life"][-2] - ds["op_life"][-1]  # 对方上一轮和本轮的体力差，包括跑位、迎球、击球消耗的体力
pos_before = ds["op_pos"][-3]  # 对方上一轮跑位之前的位置
pos_after = ds["op_pos"][-2]  # 对方上一轮跑位与迎球之后的位置
ball_vy_change = ds["op_out_vy"][-1] - ds["op_in_vy"][-2]  # 对方上一轮给球的速度
run_dir = ds["op_run_dir"][-1] # 对方上一轮跑位方向
if not run_dir: #如果上一轮使用隐身术，猜一个方向
    run_dir = 1 if pos_before <= Y/2 else -1
op_card = ds["op_card"][-1] #对方上轮使用的道具
my_card = ds["my_card"][-2] #我方上上轮使用的道具
op_spin_param = CARD_SPIN_PARAM if my_card == CARD_SPIN else 1 #上轮如果使用旋转球，发球时的速度要除以一个系数
#从对方总体力消耗中减去击球的体力消耗
life_dec -= ((abs(ball_vy_change)/op_spin_param)**2 // FACTOR_SPEED**2) * (CARD_AMPL_PARAM if my_card == CARD_AMPL else 1)
#去除加血、减血卡的影响
if op_card and op_card == CARD_INCL:
    life_dec += CARD_INCL_PARAM
if my_card and my_card == CARD_DECL:
    life_dec -= CARD_DECL_PARAM
#根据对方上轮跑位方向，得到合理的跑位范围
if run_dir == 1:
    try_min, try_max = int(pos_before), Y
else:
    try_min, try_max = 0, int(pos_before)
```

⑧. cal_life_dec(pos)

该函数用于计算跑位后位置与跑位、迎球消耗总体力的函数关系，通过反解得到对方上轮跑位后的位置，然后存入历史数据，将其用于之前的猜对方策略的函数中。

```
    #跑位后位置与跑位、迎球消耗总体力的函数关系
    def cal_life_dec(pos):
        if op_card and op_card == CARD_TLPT: #考虑使用瞬移术的情况
            if abs(pos-pos_before)<= CARD_TLPT_PARAM:
                calc = ((pos_after-pos)**2 // FACTOR_DISTANCE**2) * (CARD_AMPL_PARAM if my_card == CARD_AMPL else 1)
            else:
                calc = ((pos_after-pos)**2 // FACTOR_DISTANCE**2) * (CARD_AMPL_PARAM if my_card == CARD_AMPL else 1) \
                + ((abs(pos-pos_before)-CARD_TLPT_PARAM)**2 // FACTOR_DISTANCE**2)
        else:                   #其他情况
            calc = ((pos_after - pos) ** 2 // FACTOR_DISTANCE ** 2) * (CARD_AMPL_PARAM if my_card == CARD_AMPL else 1) \
                + ((pos-pos_before) ** 2 // FACTOR_DISTANCE ** 2)
        return calc - life_dec
    #反解出对方上轮跑位后的位置
    roots = [fsolve(cal_life_dec, guess, full_output=True) for guess in range(0,Y+1,10000)]
    #只有在合理跑位范围内的解才被接受
    acceptable = [res[0][0] for res in roots if res[2]==1 and try_min <= res[0][0] <= try_max]
    if not acceptable:
        run_pos = None
    else:
        #如果有多个解，取跑位距离更少的解
        run_pos = round(min(acceptable, key=lambda pos: abs(pos-pos_before)))
    #将计算结果存入字典中
    ds["op_run_pos"].append(run_pos)
return RacketAction(tb.tick, tb.ball['position'].y - tb.side['position'].y, (vy_best-vb0)/spin_param, decide_run(yb0), None, next_card_code)
def summarize(tick: int, winner: str, reason: str, west: RacketData, east: RacketData, ball: BallData, ds: dict):
    return ds
```

2.2.3Sierra

① 发球函数：serve()

```
5  def serve(op_side: str, ds: dict) -> tuple:
6      return 500100, 279#固定发球
```

每次发球的位置都是特定的，在这里并没有考虑使用历史数据。第一次发球的位置与速度要满足打角的最优策略。

② 目标点转落点函数：targettopos()

```
11 def targettopos(tgt):
12     pos = 0
13     if 1000000 < tgt < 2000000:
14         pos = 2000000 - tgt
15     if 2000000 < tgt < 3000000:
16         pos = tgt - 2000000
17     if -1000000 < tgt < 0:
18         pos = - tgt
19     if -2000000 < tgt < -1000000:
20         pos = tgt + 2000000
21     return pos
```

实际上，要满足撞击一次或者两次墙壁的条件，可以将原有的场地进行拓展，这里可以参照李逸飞同学在 table.py 中的相关阐述。

我们把不经过任何反弹而到达的地方成为目标点，这样方便我们在自己的代码中使用这样的信息。但是实际对战还是采用具体的在 y 轴上的位置的值的。于是我们就写了这样一个转化目标点为实际的落点的函数。

③ 实现速度函数：targettoveloctiy()

```
24 def targettovelocity(tgt, ori):
25     vel = 0
26     if 1000000 < tgt < 2000000 or -2000000 < tgt < -1000000:
27         vel = (tgt - ori) // 1800 + 1#这里为什么要+1#加1取整
28     if 2000000 < tgt < 3000000 or -1000000 < tgt < 0:
29         vel = (tgt - ori) // 1800
30     return vel
```

为了实现最优的打角的策略，我们需要计算出经过我们对球的加速之后的理想的能够打角的速度是多少。

这个函数输入我们方球拍的当前位置（暂时不考虑跑位与迎球），以及我们期望的球的目标位置，经过计算，输出前面所述的一个理想的速度。由于最终对战的时候要输出一个整数值，为了保证球不违反撞击一次或者两次墙壁的条件，我们做了一些取整数的考虑

④ 计算生命损失函数：calllife()

```
33 def callife(pos_a, pos_b, v_a, v_b):
34     lostlife = (pos_a - pos_b) ** 2 // FACTOR_DISTANCE ** 2 + (v_a - v_b) ** 2 //
       FACTOR_SPEED ** 2
35     return lostlife
```

这个函数很简单，输入某一方的初始位置以及变化后的位置，初始速度以及给球加速后的速度，然后计算生命损失的成本。这里没有分别讨论跑位以及迎球对生命损失的影响，因为我们认为，如果可以较好地推测对方击球过来时球的落点，那么最理想的方案是跑位的时候，先完成到球落点的一半距离，之后再完成另一半的距离。因此，我们假设对方采用这种最优的策略，与此同时，我们的迎球与跑位也采用这样一种思路。

⑤ 最佳正常回球函数：best_throw()

```
41      def best_throw(w_pos, e_pos, ball_v):
42          tgt_gain = []
43          for i in range(6):
44              #
45              gain = - am * sp**2 * callife(0, 0, ball_v, velocities[i]) + callife(e_pos,
                positions[i], 0, 0)#给对方最坏的跑位#我自己则只考虑速度
46              tgt_gain.append(gain)
47          best_gain = max(tgt_gain)
48          best_num = tgt_gain.index(best_gain)
49          return best_num, best_gain
50
```

按照在前面 targettopos()函数处的分析，如果要实现打角，那么可能会有 6 个目标落点。然而，这六个落点对于我们来说，价值是不一样的。其中必然有一个落点，使得我们体力的消耗最少，以及对方的体力消耗最多。此处也考虑到如果对方使用变速牌或者使用压力牌的影响，总的来说，我们是要克服对方使用卡牌对我们的影响而实现打角的。这里计算体力的消耗要调用前面的 calllife()函数。

⑥ 可以到达道具的所有目标函数：cardtotargets()

```
52      def cardtotargets(cardx, cardy, w_pos):
53          all_y = []
54          all_y.append(2000000 + cardy)
55          all_y.append(2000000 - cardy)
56          all_y.append(cardy)
57          all_y.append(-cardy)
58          all_y.append(-2000000 + cardy)#上面的tgt还没有考虑是否一定可以打得到
59          all_tgt = []
60          for i in range(5):
61              temp = 1800000 / (-wx + cardx) * (all_y[i] - w_pos) + w_pos#这里处理一条直线
62              if 999999 < int(temp) < 2999999 or -1999999 < int(temp) < -1:
63                  all_tgt.append(temp)
64          return all_tgt
```

继续我们的思路，如果将场地展开以更容易地考虑反弹问题，当我们要去主动获取某一个道具的时候，球拍位置与道具位置的连线必然与对球拍活动的直线有交点。显然地，其中有些交点是不合理的（会打到球台外）。于是排除掉这些不合理的目标位置之后，剩下来的就是可以考虑使用的目标位置。

⑦ 单道具最佳路径函数：single_best()

```
70      def single_best(w_pos, e_pos, cardx, cardy, cardcode, ball_v):
71          all_tgt = cardtotargets(cardx, cardy, w_pos)
72          if all_tgt:
73              all_val = []
74              leng = len(all_tgt)
75              for i in range(leng):
76                  temp = - am * sp**2 * callife(0, 0, ball_v, targettovelocity(all_tgt[i],
                    w_pos)) \
77                         + callife(e_pos, targettopos(all_tgt[i]), 0, 0)
                           #同样是选择损耗对方生命最多，节省自己生命最少的方法
78                  if cardcode == 'IL' or cardcode == 'DL':  #
                    加减生命#这里的权值是我们自己设定的
79                      temp += 3500
80                  if cardcode == 'TP':  # 瞬移
81                      temp += 1000
82                  if cardcode == 'AM':  # 变压器
83                      temp += 1600
84                  if cardcode == 'SP':  # 旋转球
85                      temp += 2000
86                  if cardcode == 'DS':
87                      temp += 0
88                  all_val.append(temp)
89              single_card_val = max(all_val)  # 最佳路径净价值
90              single_card_choice = all_val.index(single_card_val)  # 最佳路径位置编号
91              single_card_tgt = all_tgt[single_card_choice]  # 最佳路径目标
92              single_card_pos = targettopos(single_card_tgt)  # 最佳路径落点
93              single_card_v = targettovelocity(single_card_tgt, w_pos)#给出速度的值
94              return single_card_tgt, single_card_pos, single_card_val, single_card_v
95          else:
96              return None, None, None, None
```

前面的 cardtotargets()函数已经找出了所有可以考虑使用的目标位置，但是再考虑加上

使用各种道具的情况下，必然有一个最优的目标位置，相应地，也必然有一个对应的最优的目标落点。这里最优的标准和前面相似，也就是使得对方的跑位距离最长，以及我们自己加速生命值付出最少。除此之外，我们还对每一种卡牌的价值给一个权值，这里仅仅为了方便下一个函数的调用而做的工作。

⑧ 多道具最佳回球路径：multi_best()

```
102     def multi_best(w_pos, e_pos, ball_v, all_cards):
103         best_val = -10000000
104         best_tgt = None
105         best_pos = None
106         best_num = 0
107         for i in range(len(all_cards)):
108             tgt, pos, val, c_v = single_best(w_pos, e_pos, all_cards[i].pos.x, all_cards[i
            ].pos.y, all_cards[i], ball_v)
109             if val and best_val < val:
110                 best_val = val
111                 best_tgt = tgt
112                 best_pos = pos
113                 best_num = i#自己写的排序算法，其实可以通过已有的数据类型实现的
114         if best_tgt:
115             del all_cards[best_num]
116             c_v = targettovelocity(best_tgt, w_pos)#我们要给的速度
117             return best_tgt, best_pos, best_val, c_v, all_cards
118         else:
119             return None, None, None, None, None
120
```

在球台上有单个道具存在的时候，我们需要根据在每个单道具最佳路径中选择最佳的那个。于是这个函数调用前面的 single_best()函数。最终给出最佳道具的最佳路径的目标位置、球速以及生命值的获利。

⑨ 跑位策略集函数：runtheo()

```
148     def runtheo(theo_num, pos):
149         # 策略0：不跑位
150         if theo_num == 0:
151             return pos
152         # 策略1：跑底线中点
153         if theo_num == 1:
154             return 500000
155         # 策略2：跑迎球中点
156         if theo_num == 2:
157             if pos < 500000:
158                 run = (pos + 1000000) // 2
159             else:
160                 run = pos // 2
161             return run
162         # 策略3：跑迎球2/3处
163         if theo_num == 3:
164             if pos < DIM[3]/2:
165                 return int(DIM[3] / 3 + pos * 2 / 3)
166             return pos * 2 // 3
167
```

我们设想了四种跑位策略，分别是：

1.不跑位。这是因为对方有很大的可能采用打角的策略，这个时候，如果我们正在某个角的附近，就有不需要跑位与迎球的可能。

2.跑底线的中点。这是一个相对保守的策略，在我们的生命值比较少的时候，这个策略可以使我们最大限度地接到球。

3.跑到对方球预期的落点的距离的一半，这是生命值足够的情况下我们认为最好的选择。低生命值的情况下，首要考虑的是接到球，因此才有了策略 2。

4.跑到对方球预期的落点的距离的 2/3

实际我们基本没有使用第四种策略。

⑩ 打击策略集函数：hittheo()

```
169     def hittheo(theo_num, pos):
170         # 策略0：打原角落
171         if theo_num == 0:
172             if pos < 500000:
173                 return 0
174             else:
175                 return 1000000
176         # 策略1：打对角落
177         if theo_num == 1:
178             if pos < 500000:
179                 return 1000000
180             else:
181                 return 0
182 #
```

我们设想了两种打球策略：

1.打原角落

2.打对角落

⑪ 跑位策略评估函数：scoretheo()

```
184     def scoretheo(poslst, runlst):
185         totaltheo = 4  # 跑位策略总数
186         samplenum = 4  # 取最近的样本数
187         if len(runlst) < samplenum:  # 若历史数据不够样本数
188             samplenum = len(runlst)
189         scores = list(range(totaltheo))  # 储存各个策略打分，分数越低，策略表现越好
190         for i in range(totaltheo):
191             for j in range(samplenum):
192                 # 第-3个击球点决定第-1个跑位
193                 if runlst[-1-j] is not None:
194                     scores[i] += abs(runtheo(i, poslst[-2-j])-runlst[-1-j])  #
                        该式可以乘以与j有关的加权#跑得越长越费事
195         besttheonum = scores.index(min(scores))
196         return besttheonum
```

这个函数考虑了我们四种跑位策略，以及使用了以前记录的历史数据。我们计算每种跑位策略在 y 轴上的跑位距离，球从球场一端跑到另一端的时间是一定的，因此距离越短的跑位策略越好。基于这样的想法，我们找出之前四次采用不同的跑位策略的跑位距离的值中最小的那个，作为我们的本次的跑位策略，但这个函数是有缺陷的：

1.我们没有考虑存储并使用与一个对手对战的所有数据，而只是使用最近四次的数据。

2.我们没有考虑存储与某个特定算法对战的数据，并在下一次对战中使用。对我们来说，每一次对战都是全新的。

这两点是我们可能的改进空间。

⑫ 打击策略函数：scorehit()

```
199     def scorehit(poslst, hitlst):
200         totaltheo = 2
201         samplenum = 3
202         if len(hitlst) < samplenum:
203             samplenum = len(hitlst)
204         scores = list(range(totaltheo))#一个有二项的列表
205         for i in range(totaltheo):
206             for j in range(samplenum):
207                 scores[i] += abs(hittheo(i, poslst[-2-j])-hitlst[-1-j])
                    #同样的，一个回合的时间是一定的，所以路程越长越费事
208         besttheonum = scores.index(min(scores))
209         return besttheonum
```

为了找出本次最好的打击策略，我们小组设计了这样一个函数。这个函数的思路与上一个函数基本相同，不再赘述。

之后就是这些函数的应用：

1) 首先用于猜测对方的跑位迎球击球的策略。在我方生命值足够的情况下，我们假设对方

采取我们所认为的最优策略，包括跑位策略与击球策略，这可以通过把对方的相应数据输入我们的策略评估函数中来获取。

```
244     '''猜跑位'''
245     #   猜测对方跑位
246     if t/1800 < 10:#时间不足的时候，被迫调整的结果
247         theo = 1
248     else:
249         theo = scoretheo(ds['e_pos'], ds['e_run'])
250     e_guess = runtheo(theo, e)
251
252     #   猜测对方击球
253     if t/1800 < 10:
254         hitnum = 1#使用最稳妥的策略
255     else:
256         hitnum = scorehit(ds['w_pos'], ds['w_pos'])#使用最好的策略
257
258     #   对方是否使用spin道具
259     if e_use_card is not None and e_use_card == 'SP':
260         sp = 2
261     else:
262         sp = 1
263
264     #   对方是否使用ampl道具
265     if e_use_card is not None and e_use_card == 'AM':
266         am = 2
267     else:
268         am = 1#道具效果的体现
```

对方使用的道具的作用也会体现出来，这在最佳正常回球函数：best_throw()中实现。

2) 我们必然有一个较优的目标点，根据打角策略，这样的目标点有六个，那么对应的落点也有六个，实现到这六个落点的速度也有六个。后两个值靠调用目标点转落点函数：targettopos()、实现速度函数：targettoveloctiy()来实现。

```
270     '''设置正常目标'''
271     #   目标列表
272     targets = [2999999, 1999999, 1000001, -1, -999999, -1999999]
273     #   落点列表
274     positions = []
275     for i in targets:
276         positions.append(targettopos(i))
277     #   速度列表
278     velocities = []
279     for i in targets:
280         velocities.append(targettovelocity(i, w))
281
```

3) 调用最佳正常回球函数：best_throw()，输入我方当前的位置、调用前面猜测出来的对方的目标位置，以及球当前的速度，就可以求出一个最佳的目标点的名称（编号），以这个点的名称作为索引，可以得到一个最佳的目标点。在考虑到卡牌的问题的时候，情况会稍微复杂一点，调用多道具最佳回球路径：multi_best()，可以求出一个最佳的使用道具的目标点（生命值的使用效率最高）。对这个生命值使用效率与前面不适用道具的效率，进行对比，最终给出我们是否要去获取卡牌，获取哪一个卡牌的最优决策。

```
281
282     '''设置击球'''
283     # 最佳正常击球
284     norm_num, norm_gain = best_throw(w, e_guess, v)
285     w_target = targets[norm_num]
286
287     # 最佳道具击球
288     pick = False
289     if table_cards:
290         card_tgt, card_pos, card_gain, card_v, table__cards = multi_best(w,
            e_guess, v, table_cards)
291         # 我方取不取道具
292         if card_tgt and norm_gain < card_gain:#绝了，这才联通在一起
293             w_target = card_tgt
294             pick = True
295
296     # 最终击球决策
297     w_v = targettovelocity(w_target, w)#w是上一次的位置，返回理想的速度
```

4) 我们的思路有一个问题，这个问题也是值得被提出的。就是我们在对方使用了让我们的速度减半的道具之后，我们原有的正常策略就会被扰乱，而且会有不合法击球的风险，为避免这样的风险，我们进行了一些调整，但是结果并不理想。

```
297     w_v = targettovelocity(w_target, w)#w是上一次的位置，返回理想的速度
298     # spin道具保险
299     v_change = (w_v - v) * sp#v是上一次的速度，要考虑道具的影响
300     v_new = v_change + v #有bug,这里是不能返回一个理想位置的速度的
```

5) 关于收集到的道具的使用，我们的策略如下：如果是给自己补充生命值或者是消耗对方生命值的道具，我们获得之后就马上使用；瞬移卡则在我们需要跑位的距离超过一个预期值的时候使用；变压器与减速卡则在效益最高的时候使用。而在道具箱子满的时候，直接使用最先获取的那个；当自己的生命已经不足时，则立刻依次使用道具箱子里的道具。

```
339     w_use_card = None
340     if CARD_INCL in w_cards:#生命卡直接用
341         w_use_card = CARD_INCL
342     elif CARD_DECL in w_cards:#减对方生命的卡牌直接用
343         w_use_card = CARD_DECL
344     elif CARD_TLPT in w_cards and w_run - w > 250000:
345         w_use_card = CARD_TLPT
346     elif CARD_SPIN in w_cards and ((w_v*1800 + w > 2900000 and w < 100000) or
         (w_v*1800 + w <-1900000 and w > 900000)):
347         w_use_card = CARD_SPIN
348     elif CARD_AMPL in w_cards and ((w_v*1800 + w > 2900000 and w < 100000) or
         (w_v*1800 + w <-1900000 and w > 900000)):
349         w_use_card = CARD_AMPL
350     elif w_cards and w_life < 500000:#生命值小的时候，直接爆发
351         w_use_card = w_cards[0]
352     elif pick and len(w_cards) == 3:#满的时候，直接用
353         w_use_card = w_cards[0]
354
```

6) 我们还设计了一个跑位的体力保险值，当我们体力不足时，会有体力保险的脚本触发我们所有的低体力策略。

```
356     w_new_life = w_life - callife(w_last, w, v, v + v_change) - callife(
        1000000, 0, 0, 0)#留了一个路程的体力
357     if w_run >= 1800 * int((w_new_life / RACKET_LIFE) * BALL_V[0]):
358         w_run = max(1800 * int((w_new_life / RACKET_LIFE) * BALL_V[0]),
            500000)
359     if 1000000 - w_run >= 1800 * int((w_new_life / RACKET_LIFE) * BALL_V[0]):
360         w_run = min(1000000 - 1800 * int((w_new_life / RACKET_LIFE) * BALL_V[
            0]), 500000)
361
```

7) 最后要提一个猜测对方目标位置的函数：cal_run()

```
366     def cal_run():#猜测对方跑位
367         card_dc = 0
368         card_ic = 0
369         if ds['e_card'][-1] == CARD_INCL:
370             card_ic = 2000
371         if ds['w_card'][-2] == CARD_DECL:
372             card_dc = 2000
373         amp = 1
374         spin = 1
375         if ds['w_card'][-2] == CARD_SPIN:
376             spin = 4
377         if ds['w_card'][-2] == CARD_AMPL:
378             amp = 2
379         if ds['e_card'][-1] == CARD_TLPT:
380             a = 0
381         a = 2*amp / FACTOR_DISTANCE**2
382         b = -2 *amp* ds['e_pos'][-2]/FACTOR_DISTANCE**2 - 2 *amp* ds['e_pos'
            ][-1]/FACTOR_DISTANCE**2
383         c = amp*ds['e_pos'][-2]**2/FACTOR_DISTANCE**2 + amp*ds['e_pos'][-1]**
            2/FACTOR_DISTANCE**2 \
384             + spin*amp*callife(0, 0, ds['v_in'][-2], ds['v_out'][-1]) - e_ld
                - card_ic + card_dc
385         if b**2 - 4*a*c < 0:
386             run = None
387         else:
388             sol = numpy.poly1d([a, b, c])
389             run = None
390             if ds['e_pos'][-2] < 500000:
391                 for i in sol.r:
392                     if -100000 < i < 510000:
393                         run = i
394             else:
395                 for i in sol.r:
396                     if 490000 < i < 1100000:
397                         run = i
398         return run
```

显然对方的体力消耗我们是知道的，对方给球的加速我们也是可以算出来的，对方的迎球过程我们也清楚，因此根据体力的守恒，容易算出对方的目标位置。

以上就是我们最终提交的代码的主要思路。

2.3 程序限制

在 T_tree 的测试过程中，如果我方打对方原来的角落，对方不动，会相比于跑到中点有 1200 生命值的优势；不过如果我们猜中对方不跑位，从而把球打到对角上去，对方迎球将损失 2400 生命值，比跑中点多损失 1200。期间，tree 还出现过计算 bug，后来基本修正了。不过，我们小组发现搜索树整个操作就像一个黑箱一样，一定程度上说，里面的程序作出怎样的决策难以维护。

在 T_sierra 的测试过程中，除了在函数说明部分提到的那些问题，我们小组还发现存在一个吃道具出 bug 的小概率事件。如果球台出现一个道具，我方为了吃到道具可能有不只一种击球方式，但在极少数情况下，会吃不到道具，我方推测是取整存在问题，从而致使打偏。这种情况下，如果对方吃到了道具，对结果影响不大；如果对方不吃道具，而且球的运动路线与上轮相同，我方就会再次按照计算的结果去吃道具，这很可能会极大地消耗体力值，使我方处于劣势。但综合来看，这种情况发生的概率极其微小。

3 实验结果

3.1 测试数据

实验环境说明：

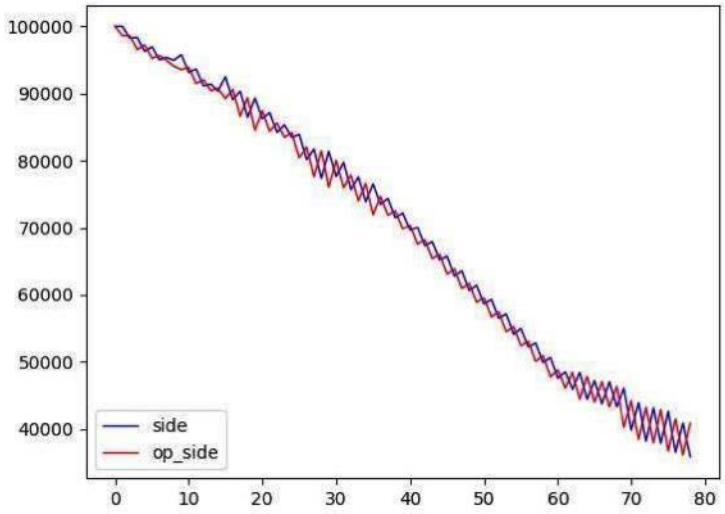
- 硬件配置：core i7 3.0GHz 双核 内存 8G
- 操作系统：macOS
- Python 版本：Python 3.5

在整个大作业的过程中，因为我们的代码在不断地更新，所以本组的测试方法主要就是将新代码和旧代码进行对打，以实现优化。在第一次小组会议之后，由于老师的代码还在进行持续性地修改，所以我们先在所提供的最新的示例代码基础上写出了一版最简单的算法，只考虑了一轮的情况，命名为 naive，可以稳定战胜 idiot。在几天之后我们完成了第二版算法，加入了道具的考虑，命名为 T_young。与此同时，我们也在进行另一种搜索树算法的尝试，并将其和 T_young 进行对打测试，结果发现搜索树 debug 之后打出来的球和 T_young 的结果几乎没有区别，只会打边线，走位走中间，反而有时候有少量球还会使其败给 T_young。其时临近热身赛，我们决定通过与其他组的实战，测试自己算法的优劣，并且比赛中要注意观察各个组的跑位模型和打球落点的模型，从对手身上学习经验，改进自己的算法。

而在随后的两次热身赛中，我们的战绩也相对不错：

热身赛	胜率	胜局	win for	lose for
第一次	111 / 151	111	{'invalid_bounce': 43, 'miss_ball': 68}	{'miss_ball': 40}
第二次	153 / 175	153	{'miss_ball': 110, 'invalid_bounce': 43}	{'miss_ball': 22}

在 5 月 29 日的第一次热身赛上，我们提交了代号为 T_ shlcy 的代码，晚上收到了测试数据和复盘数据的反馈。我们的胜率为 111 / 151，相对来说还不错。我们对对手的历史跑位进行了计算，尝试从中吸取经验教训，进一步优化自己的算法。下图是当时所做的体力值分析结果图，在大概 60 次的时候是一个临界点，之前我们一直占优势，以后就优势不明显了。



在 6 月 2 日的第二次热身赛上，我们提交了代号为 T_excited 的代码，晚上收到了反馈。我们取得了 153 / 175 的高胜率，战况喜人。随后我们再次对测试数据和复盘数据进行了分析，实现代码优化。

在正式比赛中，我们先是提交了 Sierra 的代码，但是我们并没有就此停止努力和尝试。最后时刻，新的代码有了突破性进展，将其和原来的 Sierra 进行了对战测试，取得了九胜一负的压倒性优势，最后我们提交了新的代码参赛。

在最后的比赛中，我们取得了 S 区第 3 的成绩，未能出线。因为付出颇多，所以还是感到很遗憾。

3.2 结果分析

我们的策略共分为猜跑位、猜击球、做出击球反应、跑位反应，并衡量是否收道具、是否用道具、猜对方是否收道具并做出反应几部分。在测试和热身赛的过程中，我们又持续不断地对这些策略进行优化，最终使它们发挥了良好的作用。

最初算法假定根据对方跑位猜一个对方下一轮的位置；然后要决定给球的速度，在能够弹 1 次或 2 次的所有整数速度里找了个最优值，目标函数是消耗自己的体力减去消耗对方的体力，追求最小化；道具默认使用道具库中的第一个。该策略在第一次热身赛中表现尚可。

第一次热身赛后，我们召开小组会议，一起细致分析测试数据和复盘数据的动画，总结我们输球的原因和改进策略。young 出问题实际是跑位有些错误，我们发现败局中我们的体力最后都比对方多，就是没处理好最后阶段的跑位，导致接不到角球，原来的策略是只往正向跑位，跑出了范围，由此设想，跑位跑到中间应该会好很多；此外，历史数据利用有问题，应该尝试用字典或者队列存取历史数据。跑位和历史数据策略的更新，让我们的算法进步了很多，在接下来的比赛中取得了良好的效果。第二次热身赛后，我们发现我们的败局中，有时猜测对方会跑位，实际它并没有跑位。lkjhg 这组引起了我们的关注，它比较狡猾，经常在角上不动，然后等我们把球喂到嘴里去。在此基础上，我们又对算法的策略进行了更新。原本历史数据用了很多轮，猜测有些滞后，在几个回合后才作出反应，后续只采用了三到四轮的历史数据，使得对方尽量少得用这个方法获利。最后，策略的优化在后期发挥了作用。

Sierra 算法的运行时间在我们的预料之中。搜索的结果数六个、搜索道具可能性十个以内、历史数据拟合三轮、搜索对方策略六个、对方道具可能性十个以内，每次运行解一个一元二次方程，基本不耗时间。而我们在探索过程中开发出来的 T_tree 算法比较耗时间，因

其搜索了 branches 的立方次，然后比较了 branches 的立方次，且没有用剪枝。

4 实习过程总结

4.1 分工与合作

小组名：SIERRA

本组成员：张影*，黄殊晏，李沛泽，陈梦珂，高春芝，雷苗，杨阳

合作交流方式：

1．微信群中进行沟通讨论，上传程序进行协同操作；

2．小组会议集中讨论，交流算法思路。

分工：

1．开发算法和编程任务：黄殊晏和李沛泽同学负责代码的撰写和优化。

2．小组报告撰写：陈梦珂同学负责算法思想阐述部分，杨阳和张影同学负责程序代码说明部分，雷苗同学负责测试过程部分，高春芝同学负责小组分工和实验过程总结部分。

微信讨论（节选）





第一次小组会议记录

记录人：张影

地点：二教三楼讨论区

参与人员：全体组员

时间：2017.5.18 上午



会议内容：

- 初步对实习作业进行了基本的分工；
- 对算法思路进行了探讨，确定了算法的基本思路，开始算法编写工作；
- 确定时间进度安排和下一步的工作，决定对贝叶斯决策、博弈论、蒙特卡洛方法、UCT 算法、剪枝算法、最大最小后悔值法和启发式算法等进行学习。

第二次小组会议记录

记录人：雷苗

地点：二教三楼讨论区

参与人员：全体组员

时间：2017.5.26 上午



会议内容：

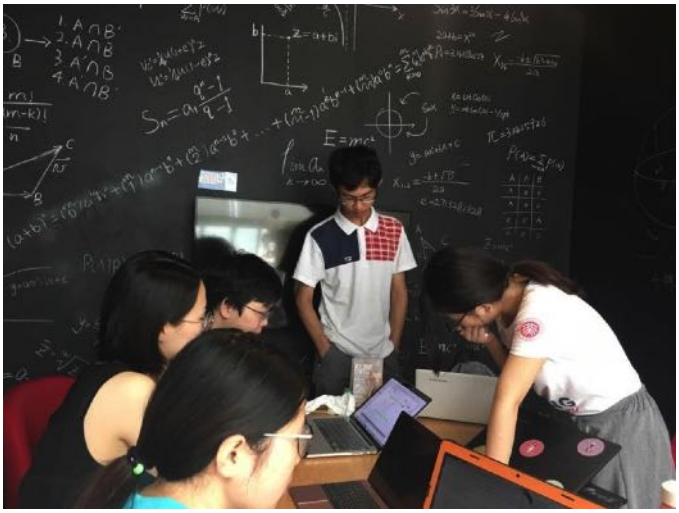
- 主要是对代码和对战策略进行进一步的讨论和优化，主要包括三个方面
  - ✓ 对于对手位置的猜测
  - ✓ 自身的跑位策略
  - ✓ 道具的使用策略

第三次小组会议记录

记录人：高春芝

地点：二教三楼讨论区

参与人员：全体组员
时间：2017.5.31 上午



会议内容：

- 对热身赛的结果进行了讨论和学习，就如何优化算法进行了讨论。
  - ✓ 分析了热身赛中输球的原因，并选择几次比赛进行可视化和观看；
  - ✓ 决定在历史数据的记录和跑位策略上进行改进。

4.2 经验与教训

这次实习大作业带给我们小组的收获颇多，一方面增强了与人合作交流的能力，一方面也提高了我们对于数算这门讲授内容的理解与应用。

在这次实习作业中，我们小组的代码存在的不足主要是：

1) 代码中有计算对方跑位的部分被放到了后面，使得一局结束后不能马上利用上一局的历史数据对对方的跑位进行猜测，上一轮的数据需要等到下一轮才能用到，历史数据的有效性和可利用性滞后了一局，一定程度上影响了我们队对手的预测，没能占得先机。
2) 在代码运行的开始我们对数据进行了初始化，这使得我们的每次比赛都相当于一次新的比赛，没有历史数据的留存，而对手一般都是从第一轮就开始记录，有大量历史数据可供利用，这在一定程度上也影响了我们对对手击球和跑位的预测。
3) 针对我们小组代码存在的问题，在分区赛结束后，我们对代码又进行了优化和改进，把计算对方跑位的代码提前，使得上一轮积累下的历史数据能够充分利用，但由于我们未能出线，因此也就没有与其他组再次对战的机会，但与我们优化前的代码相比试时，改进后的代码表现出更优的性能。

4.3 建议与设想

此次实习大作业在这么短的时间内能够成功进行可以说是十分成功了，各组的实力都十分强劲，虽然我们没能出线，但对于比赛我们还是十分认同的。关于实习大作业，我们小组有如下建议：

1. 延长实习大作业时间。此次实习大作业的代码完成时间主要是两周，时间比较紧，使得大家的准备可能不那么充分。适当延长比赛时间，或者在课程前期就开始实习大作业的

相关铺垫（比如期中作业等）可能会有更好的效果。

2. 关于比赛的赛制，建议采取积分排位，全部的小组进行排位赛，取前八名，因为分区比赛各区实力可能会有一定的偏差。此外，还可以引入复活赛等方式，使得比赛的形式更为丰富。
3. 在八强进四强、角逐冠亚军的比赛中，由于之前很多组遗憾出局，因此他们未到现场观看比赛，但我们觉得，最后几场比赛十分精彩，所以希望老师和助教能通过更多方式吸引同学们到场观看。我们建议，老师可以加强与下面观众的互动，比如增加观众讨论交流环节，必要的话可以作为实习大作业的加分项。这样的互动也可以加在前期的比赛中，那么大家在观看比赛时、或是等参赛队改进代码时不会因“不知所措”而离场。

对于以后想要选修这门课的同学们，我们小组认为这门课是一门非常有意义、有价值的课。在陈斌老师的课堂上，我们能在轻松愉快的课堂氛围中收获知识，还能得到自身编程能力的切实提高。尤其是在期末的实习大作业中，我们不仅在亲身实践中对于 Python 语言有了更深层次的理解，同时还增强了团队合作能力。此外，作为一场比赛，肯定会有胜负之分，由此又会导致分数上的差异，但这比起知识的收获来说是次要的，大家应当理性看待，胜败乃兵家常事，不要因此而忽视在这门课上所获得的宝贵的内容。

5 致谢

本次大作业在陈老师和各位助教的指导下，在全组组员的共同参与和努力下完成。十分感谢能有此次实习大作业的机会，使得我们小组的成员聚在一起，齐心协力，不断思考、学习和讨论，大家的思想相互碰撞产生出智慧的火花。在作业的过程中确实有过迷茫，但最终小组成员确定了策略和方针，共同完成了本次程序设计的实习作业。在这次作业的完成过程中，十分感谢老师和各位助教的指导，同时也感谢小组全体成员的努力。其中，尤其要感谢这次大作业的编程主力黄殊晏和李沛泽同学，他们废寝忘食地编写和优化代码，李沛泽同学更是连续 24 小时编写代码，两位同学对于我们小组代码的完成功不可没。

通过完成这次实习大作业，我们小组收益颇丰，在理解竞赛规则、编写代码的基础上，我们对于数算课堂中老师讲授的知识有了更深层次的理解和感悟；同时，这次大作业在增强了我们对于 Python 的理解的同时，也增强了我们的合作能力。再次感谢此次实习大作业，既为我们提供了展示自我机会，也为我们提供了锻炼自身能力的平台。

6 参考文献

[1] 分支界限法（剪枝法）学习：http://blog.csdn.net/u011846436/article/details/38554705

[2] α-β 剪枝算法：http://www.cnblogs.com/tk55/articles/6012314.html

[3] 最大最小准则：http://www.cnitpm.com/pm/15188.html

[4] 不完全信息博弈：百度百科 http://baike.baidu.com/link?url=wfIGE9wOznTxfMzaZGcwh2UEPL_jn5e0qr_RQKlwYzKi7RlEIZsMDaNFDTcWY0J7lFhCmgUxAlhJtllTHIQ07ZAEarUpCOdzLCFdpErDwgYnfWGg1oiYufGCLB1rn_pcZmlCfOuYFxvms8d2yF-ehK



BRAVO

组长：陈宇枫

组员：陈瑞，关汉岳，沈昕，王然，徐荣荣，杨涵

小组赛战绩：S4

Bravo 组数据结构与算法课程实习作业报告

陈宇枫* 陈瑞 关汉岳 沈昕 王然 徐荣荣 杨涵

摘要：

本组算法采用“老年人打角策略”，以保留体力为第一前提，在保留体力最大化的前提下寻求进攻，发球策略为将球发至对方桌面死角处，接球分为迎球击球两个过程，击球按不同情况共有八种应对方案，思想都为使球在桌边反弹后发射到对方桌面死角。主要采用的算法是 if 条件语句判断和 while 循环语句，同时采用了列表、字典等数据结构。

关键字：

列表 字典 类 if 条件判断 while 循环

1 算法思想

1.1 总体思路

本小组乒乓球比赛的算法总体思路是“老年人打角策略”，即以保留体力第一前提，在确保体力值最大化的前提下寻求进攻。主要采用的算法是 if 条件语句判断和 while 循环语句，同时采用了列表、字典等数据结构。以下是我们算法中采用的具体的策略。

1.1.1 发球策略

关于发球方面，我们采用的是一个静态方法。每次轮到我方发球的时候，都是从（-900000,200000）的位置处向上 45° 发射，这样会使得球到达对方场地的下方死角处。下图较为形象地展示了我们的发球策略。

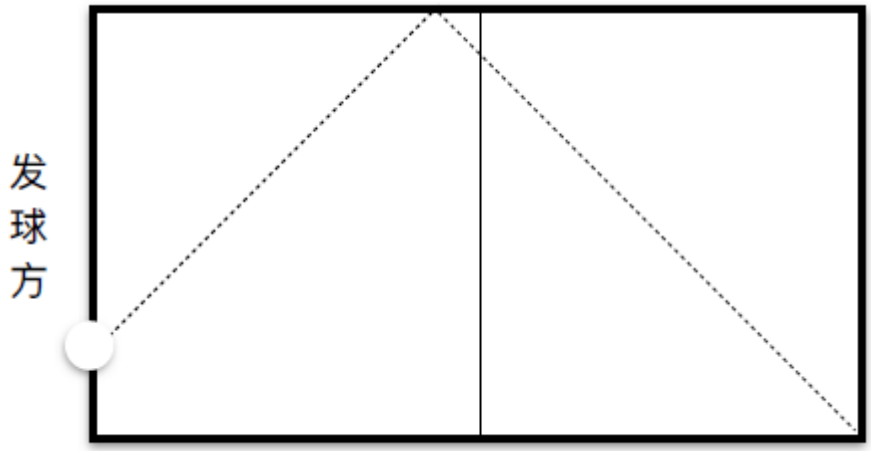


图 1 发球策略图示

1.1.2 接球策略

我们首先将接球分为两个过程，迎球与击球，分别采取策略进行。

（1）迎球：

需要跑位的距离使用球的位置减去我方的位置，直接迎球即可。

（2）击球：

a) 使用 card 道具的策略：

i. 如果自身的道具箱里有道具，且道具不为加血包，则立即使用。若道具为加血包，则在自身血量不足一定时使用，否则先不使用。

ii. 如果自身的道具箱里没有道具，则不使用。

b) 击球（加速度）策略

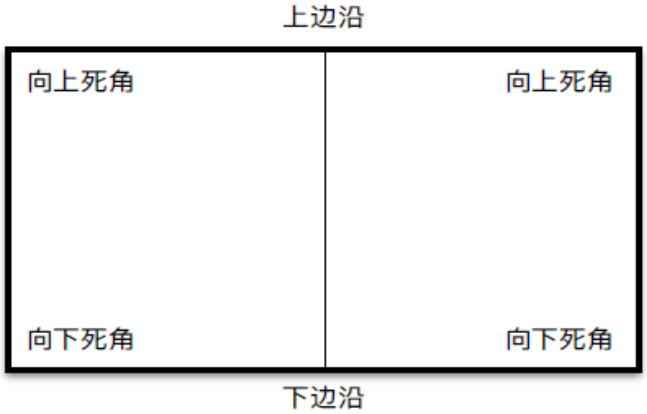


图 2 击球加速度策略

（3）具体策略

在不同的情况下，我们又提出了八种不同的方案进行应对，下表总结了八种不同的方案

表 1 八种接球策略

方案 1	通过在上边沿反弹一次，发射到对方的上死角
方案 2	通过在下边沿反弹一次，发射到对方的上死角
方案 3	通过在上边沿反弹一次，然后在下边沿反弹一次，发射到对方的上死角
方案 4	通过在下边沿反弹一次，然后在上边沿反弹一次，发射到对方的上死角
方案 5	通过在上边沿反弹一次，发射到对方的下死角
方案 6	通过在下边沿反弹一次，发射到对方的下死角
方案 7	通过在上边沿反弹一次，然后在下边沿反弹一次，发射到对方的下死角
方案 8	通过在下边沿反弹一次，然后在上边沿反弹一次，发射到对方的下死角

分别计算出每一种方案所需要的体力值（此处的体力值是包括了迎球、跑位所需的体力值以及考虑了若有道具使用的影响）计算后又会有以下两种情况：

a)当自身的体力值大于对方的体力值时，且上述方案中有所需体力值小于自身体力值的方案在满足所需体力值小于自身体力值的方案中，选出获得道具最多且消耗自身体力值最少的方案（按优先级排序），结束

b)当自身的体力值小于等于对方的体力值时，或者上述方案中没有所需体力值小于自身体力值的方案时

i.如果不给球加速度也可以有效击球，就直接接球不加速度，结束

ii.如果不给球加速度不能有效击球：若反弹数小于 1，则考虑方案一和方案六，选择体力消耗最小的且获得道具数最多，且发射和对方跑位相反的死角的方案（按优先级排序）；若反弹数大于 2，则考虑方案三和方案八，选择体力消耗最小的。

1.1.3 跑位

每次击球完之后，跑回中场位置【+-900000,500000】

1.2 算法流程图

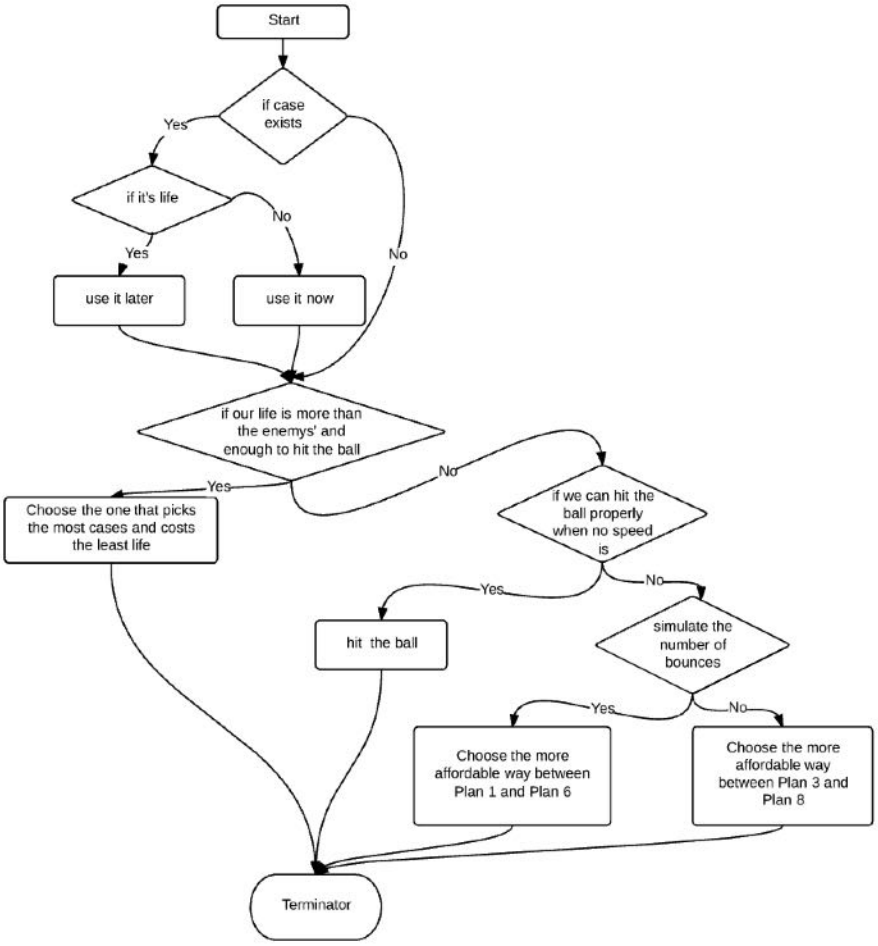


图 3 算法流程图

1.3 算法运行时间复杂度分析

此次我们的算法调用了 python 内置函数（如 sort 等），总体时间复杂度为 O(nlogn)

2 程序代码说明

我们的代码如下：

```
///
# coding=utf-8
from table import *
import operator
#  发球函数，总是做为 West 才发球
# ds 为函数可以利用的存储字典
#  函数需要返回球的 y 坐标，和 y 方向的速度
def serve(op_side, ds):
    return 200000, BALL_V[1]

def addCase(acc,case,run_dirc,b,cards):
    self_life = (abs(acc)**2 // FACTOR_SPEED**2)
    hit_cards = [card for card in cards if b.get_card(card)]
    case.append([acc, self_life,run_dirc,hit_cards])

#  打球函数
# tb 为 TableData 类型的对象
# ds 为函数可以利用的存储字典
#  函数需要返回一个 RacketAction 对象
# tick, tick_step, side, op_side, ball
#self.tick, self.tick_step,dict_side, dict_op_side, dict_ball, dict_card,player.datastore
def play(tb, ds):
    #  （加速值，加速自身所消耗的体力值，对方跑位方向，hit_card）
    case = []

    #  迎球
    distance = tb.ball['position'].y - tb.side['position'].y

    #  跑位—>场地中间位置
    run = DIM[3] // 2 - tb.ball['position'].y

    #  使用 card  策略是  只要有 card  就使用
    card = None
    while len(tb.side['cards']) > 0:
        card = tb.side['cards'].pop()
        card = card.code
        #  但是如果有  加血包就晚点使用
        if card == CARD_INCL and tb.side['life'] > RACKET_LIFE - CARD_INCL_PARAM:
            card = None
```

```
            break
        else:
            break

    # 加速因子
    spin=1
    # 如果对方使用旋转球 那么加速转球
    if tb.op_side['active_card'][1]==CARD_SPIN:
        spin=2

    # 变压因子
    ampl = 1
    # 如果对方使用变压器 那么考虑加倍掉体力值
    if tb.op_side['active_card'][1]==CARD_AMPL:
        ampl=2

    # 瞬移因子
    tlpt=0
    if card== CARD_TLPT:  # 如果碰到瞬移卡，则从距离减去 CARD_TLPT_PARAM 再计算体力值减少
        tlpt = CARD_TLPT_PARAM

    b = Ball(DIM, copy.copy(tb.ball['position']), copy.copy(tb.ball['velocity']))
    b.bounce_racket()

    # 目标：对方向上死角
    # 1 弹一下_先上弹
    acc = (DIM[3] - tb.ball['position'].y) // tb.step + 1 - tb.ball['velocity'].y
    b.update_velocity(acc, (None, None))
    addCase(acc*spin, case,tb.op_side['run_vector'],b,tb.cards['cards'])
    b.update_velocity(-acc, (None, None))
    # 2 弹一下_先下弹
    acc = (-DIM[3] - tb.ball['position'].y) // tb.step + 1 - tb.ball['velocity'].y
    b.update_velocity(acc, (None, None))
    addCase(acc*spin, case,tb.op_side['run_vector'],b,tb.cards['cards'])
    b.update_velocity(-acc, (None, None))
    # 3 弹两下_先上弹
    acc = (3 * DIM[3] - tb.ball['position'].y) // tb.step - 1 - tb.ball['velocity'].y
    b.update_velocity(acc, (None, None))
    addCase(acc*spin, case,tb.op_side['run_vector'],b,tb.cards['cards'])
    b.update_velocity(-acc, (None, None))
    # 4 弹两下_先下弹
    acc = (-DIM[3] - tb.ball['position'].y) // tb.step - 1 - tb.ball['velocity'].y
    b.update_velocity(acc, (None, None))
```

```
addCase(acc*spin, case,tb.op_side['run_vector'],b,tb.cards['cards'])
b.update_velocity(-acc, (None, None))

# 目标：对方向下死角
# 5 弹一下_先上弹
acc = (2 * DIM[3] - tb.ball['position'].y) // tb.step - 1 - tb.ball['velocity'].y
b.update_velocity(acc, (None, None))
addCase(acc*spin, case,tb.op_side['run_vector'],b,tb.cards['cards'])
b.update_velocity(-acc, (None, None))
# 6 弹一下_先下弹
acc = (- tb.ball['position'].y) // tb.step - 1 - tb.ball['velocity'].y
b.update_velocity(acc, (None, None))
addCase(acc*spin, case,tb.op_side['run_vector'],b,tb.cards['cards'])
b.update_velocity(-acc, (None, None))
# 7 弹两下_先上弹
acc = (2 * DIM[3] - tb.ball['position'].y) // tb.step + 1 - tb.ball['velocity'].y
b.update_velocity(acc, (None, None))
addCase(acc*spin, case,tb.op_side['run_vector'],b,tb.cards['cards'])
b.update_velocity(-acc, (None, None))
# 8 弹两下_先下弹
acc = (-2 * DIM[3] - tb.ball['position'].y) // tb.step + 1 - tb.ball['velocity'].y
b.update_velocity(acc, (None, None))
addCase(acc*spin, case,tb.op_side['run_vector'],b,tb.cards['cards'])
b.update_velocity(-acc, (None, None))



# distance 是迎球（受变压器影响） run 是跑位（受瞬移因子影响）
life= (abs(distance) ** 2 // FACTOR_DISTANCE ** 2) * ampl \
        +(max(abs(run)-tlpt,0)) ** 2 // FACTOR_DISTANCE ** 2

# 因为之前的 case 中只是考虑了旋转球的因素
i = 0
while i < len(case):
    case[i][1] = case[i][1] * ampl + life
    i += 1

# 当有优势的时候（自身 life 值 大于 对方 life 值）
# (加速值，加速自身所消耗的体力值，对方跑位方向，hit_card）
if tb.side['life']>tb.op_side['life']:

    # 按自身消耗体力的大小排序
    order_case=sorted(case,key=operator.itemgetter(1))
    # 过滤掉 自身所消耗的体力值>自身目前拥有的体力值 的情况
    order_case=filter(lambda x: x[1] < tb.side['life'], order_case)
```

```
        order_case=list(order_case)
        # def comp(x,y):
        #         if len(x[3])>len(y[3]):
        #                 return -1
        #         elif len(x[3])<len(y[3]):
        #                 return 1
        #         else:
        #                 # 加血包权值最重
        #                 if CARD_INCL in x:
        #                         return -1
        #                 elif CARD_INCL in y:
        #                         return 1
        #                 else:
        #                         return 0
        flt_case1=filter(lambda x:x[3]!=[] ,order_case)
        flt_case1 = sorted(flt_case1,key=lambda x:len(x[3]))
        flt_case1 = list(flt_case1)
        # 过滤掉 自身所消耗的体力值-对方消耗的体力值>自身目前拥有的体力值－对方目前拥有的体力值 的情况
        # flt_case2=filter(lambda x: x[1]-x[2]<=tb.side['life']-tb.op_side['life'], order_case)

        if flt_case1:

                return RacketAction(tb.tick, distance, flt_case1[0][0], run,None,card)
        elif order_case:
                return RacketAction(tb.tick, distance, order_case[0][0], run,None,card)
        # 采取保守进攻
        else:
                pass

    # 模拟不给加速度 球是否会按要求传递过去
    count_bounce, hit_cards = b.fly(tb.step, tb.cards['cards'])
    # 模拟结果发现 ：没有在墙上反弹
    if count_bounce <1:
        # 使用 case1 和 case6
        # 选择体力值消耗少的 case，若消耗相等，看得到的 hitbox
        if case[0][1]>case[5][1]:
                ans=case[5][0]
        elif case[0][1]<case[5][1]:
                ans=case[0][0]
        else:
                if len(case[0][3])>len(case[5][3]):
                        ans=case[0][0]
                elif len(case[0][3])<len(case[5][3]):
```

```
                ans=case[5][0]
            else:
                if case[0][2]>0:
                    ans=case[5][0]
                else:
                    ans=case[0][0]
        return RacketAction(tb.tick, distance, ans, run,None,card)
    # 模拟结果发现 ：反弹次数太多
    elif count_bounce >2:
        if tb.ball['velocity'].y>0:
            # 使用 case3
            return RacketAction(tb.tick, distance, case[2][0], run,None,card)
        else:
            # 使用 case8
            return RacketAction(tb.tick,distance,   case[2][0], run,None,card)
    # 模拟结果发现 ：反弹次数符合要求，那么保守进攻不提供加速度
    else:
        return RacketAction(tb.tick, distance, 0, run,None,card)

# 对局后保存历史数据函数
# ds 为函数可以利用的存储字典
# 本函数在对局结束后调用，用于双方分析和保存历史数据
def summarize(tick: int, winner: str, reason: str, west: RacketData, east: RacketData, ball:
BallData, ds: dict):
#def summarize(tick, winner, reason, west, east, ball, ds):
    Return
    ///
```

2.1 数据结构说明

我们主要采用列表、字典等数据结构；主要利用字典作为不同击球方案的存储库，同时利用字典这种数据类型作为历史数据的存储数据库。如：在 addCase 函数中，利用了 case 的 List 用于存放我们的 8 个方案，以从中筛选出所需方案。

2.2 函数说明

我们所编写的函数较少，主要以完成基本的接发球为主，附加将上述策略实现的过程。主要分为以下几个函数：

Serve(op_side , ds): 发球函数，采取静态策略，总是从（-900000,200000）的位置处向上 45° 发射，以使球到达对方场地的下方死角处。

AddCase(acc,case,run_dirc,b,cards): 集合策略函数，将不同的策略与采取该策略的得失添加到列表中，以进行策略筛选。

Play(tb,ds):打球函数，主要是计算每次球的位置，不同的击打策略所消耗的体力，对方可能消耗的体力以及道具的使用，具体策略第一部分已经详细描述，具体实现过程代码中已经详细标注，这里不再赘述。由于小组组员的年级问题，时间限制较多，没有对 ds 字典进行进一步利用。

2.3 程序限制

对于程序而言：目前只出现体力值不足时可能出现接不到球的情况。在实际对战中未发现程序出错情况。

对于算法而言：由于我们采取的简单打角策略，并以打出合法球为基准对速度区间进行了界定，并总结了 8 种方案并选取性价比最高的一种进行击球。其实限制较大，首先我们采取的 8 种策略其实本质相似，策略较为单一。其次，我们对对方体力损失的判断是基于对方跑回中间位置的判断，而这一点在实战中十分不稳定。最后，我们对道具没有刻意追求，所以战绩较为中规中矩。

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) i7-7700HQ CPU @ 2.80GHz /内存：8G
- 操作系统：win10 家庭版
- Python 版本：3.6.1（运行平台：PyCharm Community Edition 2016.3.2）

测试过程说明:

1. 我们首先将本组的算法与 idiot 算法进行简单的 5 次比赛测试，结果为全胜，没有任何 Bug 产生。
2. 其次我们自己和自己打，结果表明作为 West 方胜率稍高一点，也没有任何 Bug 产生。

除此之外，本组无其他测试。

3.2 结果分析

由于本组进行的测试有限，以及对热身赛的复盘数据总结不足，最后对算法策略没有进行大的调整。热身赛的胜率在 50%左右，从复盘数据来看，我们的算法优缺点较为明显：

优点在于：较为稳定，不会出现打出不合法球的情况，若是遇到对于跑位较差的队伍而言，能够发挥出较好的成绩，能将对方磨死；

缺点在于：策略较为简单，容易被针对，后劲不足，一旦被压制就几乎难以翻身，同时对历史数据的分析不足，策略也比较没有针对性。

对于结果，还是在我们的意料之中的，较为中规中矩。

4 实习过程总结

4.1 分工与合作

考虑到我们小组的同学都为大四的同学，大家的答辩时间各自不同，因此，为了减轻小组同学大的答辩负担，我们在完成具体的工作和小组分工方面采取了“共同讨论，分组负责”的做法，其中陈宇枫、陈瑞和关汉岳同学负责小组代码的完成与相关测试。沈昕、王然、徐荣荣和杨涵同学负责小组报告的完成。

在完成整个大作业的过程中，我们小组的成员采取线下讨论和线上微信群内交流相互结合的方式进行。其中，我们共进行了两次线下讨论，分别深入地讨论了小组的对战策略与代码测试等内容。值得一提的是，由于自身知识技能的局限性，我们还请了专业为计算科学的同学给我们提出参考性建议，通过线上视频远程对话讨论的方式，她指出了我们的不足之处，为我们小组的代码书写与报告的完成提出了很多建设性的修改意见。

同时，由于小组成员构成问题导致时间和能力有限，此次大作业完成的较为仓促，存在着许多的不足。



图 4 小组成员在讨论算法策略

4.2 经验与教训

首先确定算法的策略很重要，采取何种打法是最终比赛输赢的核心，一旦确定将想法翻译成代码就是一个相对简单的工作，所以大作业的关键就是策略。在考虑跑位时我组就原地待命还是跑回桌边中点处进行了反复讨论，也对总体思想是保守保全体力值还是主要关注积极进攻进行了探讨建议与设想。其次，热身赛很重要，经过热身赛的调整可以发现很多不足。比如我们就是因为对热身赛的复盘分析不足，导致从热身赛的 50%+ 胜率掉到最后的 40%左右。

建议学弟学妹在组队时平衡好各级各水平人员所占比例，多进行当面的线下讨论，提前作准备，为之后的期末季大波考试的复习预留出时间，可以请计算机专业相关的同学对作业提出参考意见，以便改进完善算法。多关注课程微信群内的动态，其他同学的一些发言、新奇的想法可能会为自己算法的设计提供灵感。

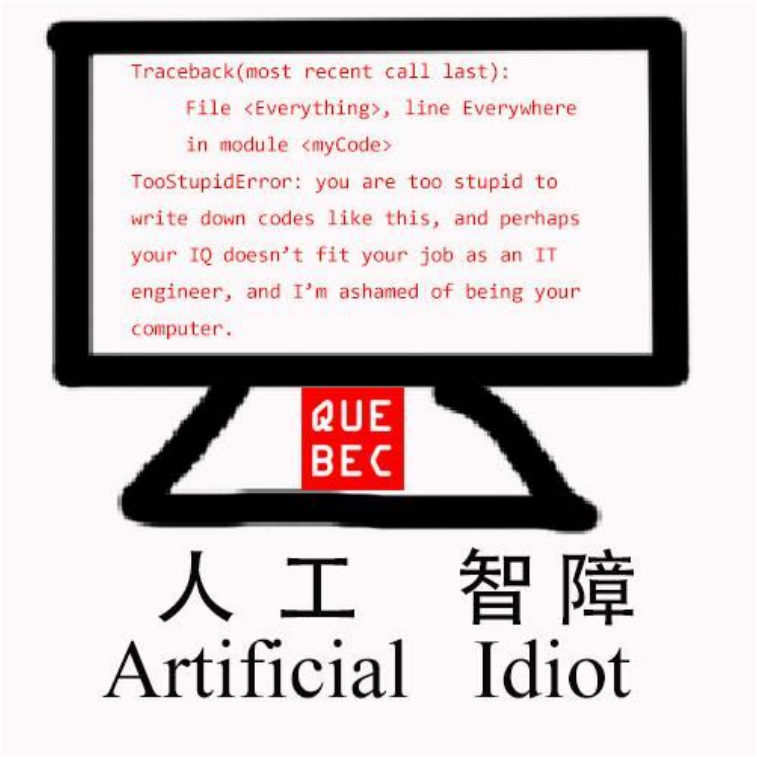
5 致谢

感谢参与本组讨论并为本组算法编写提出建议的相关专业同学，感谢老师与助教在群里为同学提出的问题提供了耐心的实时解答。

6 参考文献

Miller B N, Ranum D L. Problem Solving with Algorithms and Data Structures Using Python Second Edition [M]. Franklin, Beedle & Associates, 2011.

Magnus Lie Hetland［挪］. 司维等译. Python 基础教程（第 2 版 · 修订版）［M］，人民邮电出版社，2014



QUEBEC

组长：杨帆

组员：张昊，金恬，曹寒冰，姜河之是，胡若冰，叶勃

小组赛战绩：S5











Quebec

Quebec 小组

数据结构与算法课程

漂移乒乓实习作业报告

杨帆* 张昊 金恬 叶勃 姜河之是 胡若冰 曹寒冰

摘要：

我们组在算法讨论阶段，想到了很多种算法方案。最终我们使用了基于 pandas 的博弈树搜索，主要运用最大最小树搜索思想，在每一局中对己方和对方的跑位、迎球、击球和使用道具情况进行模拟，并根据生命值之差对模拟结果进行比较，并执行其中最优的策略。我们使用了搜索树进行搜索，并且使用了基于 pandas 的博弈树搜索算法给对方设置的估值函数，从数据上看是较优解。同时使用单回合体力消耗最大的原则推测对手的行为，有针对的进行决策。在代码运行方面，我们组的算法小组第三的成绩，虽然没有出线，但是收获很大。

关键字：基于 pandas 的博弈树搜索 估值函数 广度搜索 coding

1 算法思想

我们组一开始确定的是搜索算法和 Alpha-beta 剪枝，试图通过高效的算法（限定时间内搜索更加深入）和更准确的估值函数来获取优势。但是后来道具的加入使随机性不确定性增强，估值函数完全不可能在搜索深入后保持起码的准确性，甚至难以设计估值函数。我们放弃了深搜，决定转向广度搜索。我方给对方设置的估值函数，从数据来看是较优解，而我方由于比这个假想敌多搜索一层，并且有更优的跑位策略以及道具使用策略，可以获得比赛胜利。因此，下面就我们组考虑过的所有算法思路进行逐一介绍，主要对我们采用的算法策略分析。

1.1 总体思路

1.1.1 极大极小搜索&Alpha-beta 剪枝

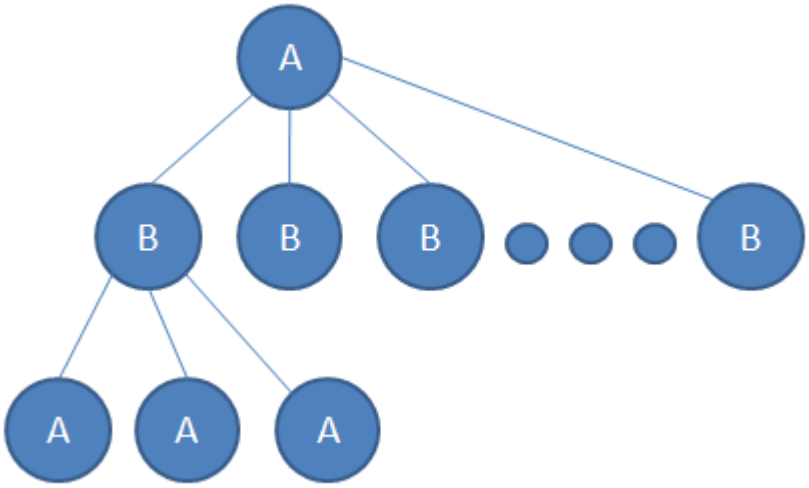
极大极小搜索：

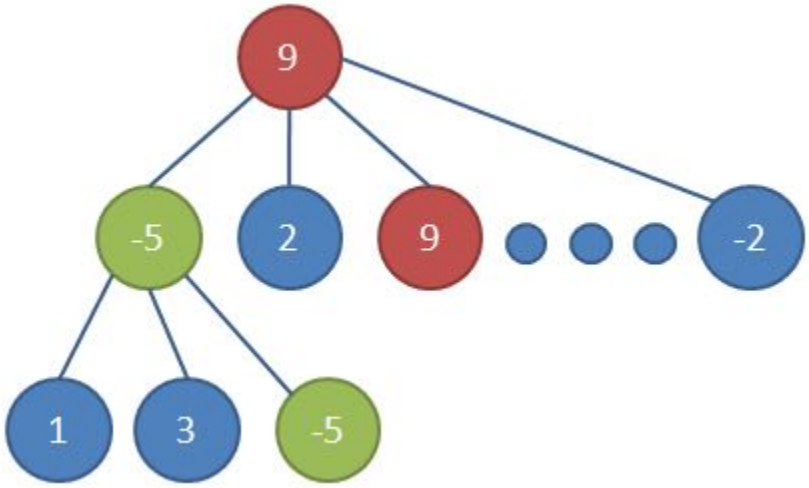
一般应用在博弈搜索中，比如：围棋，五子棋，象棋等。结果有三种可能：胜利、失败和平局。暴力搜索，如果想通过暴力搜索，把最终的结果得到的话，搜索树的深度太大了，机器不能满足，一般都是规定一个搜索的深度，在这个深度范围内进行深度优先搜索。

假设：A 和 B 对弈，轮到 A 走棋了，那么我们会遍历 A 的每一个可能走棋方法，然后对于前面 A 的每一个走棋方法，遍历 B 的每一个走棋方法，然后接着遍历 A 的每一个走棋方法，如此下去，直到得到确定的结果或者达到了搜索深度的限制。当达到了搜索深度限制，此时无法判断结局如何，一般都是根据当前局面的形式，给出一个得分，计算得分的方法被称为评价函数，不同游戏的评价函数差别很大，需要很好的设计。

在搜索树中，表示 A 走棋的节点即为极大节点，表示 B 走棋的节点为极小节点。

如下图：A 为极大节点，B 为极小节点。称 A 为极大节点，是因为 A 会选择局面评分最大的一个走棋方法，称 B 为极小节点，是因为 B 会选择局面评分最小的一个走棋方法，这里的局面评分都是相对于 A 来说的。这样做就是假设 A 和 B 都会选择在有限的搜索深度内，得到的最好的走棋方法。



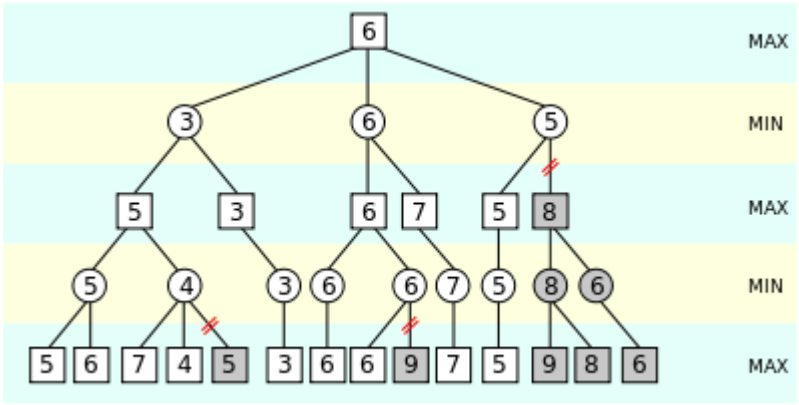


Alpha-beta 剪枝算法：

Alpha-beta 剪枝是一种搜索算法，用以减少极小化极大算法（Minimax 算法）搜索树的节点数，相当于极大极小值算法的优化。它是一种对抗性搜索算法，主要应用于机器游玩的二人游戏（如井字棋、象棋、围棋）。当算法评估出某策略的后续走法比之前策略的还差时，就会停止计算该策略的后续发展。

Alpha 和 Beta 分别指的是 MAX 和 MIN 节点。Alpha Beta 剪枝算法的基本依据是：棋手不会做出对自己不利的选择。依据这个前提，如果一个节点明显是不利于自己的节点，那么就可以直接剪掉这个节点。

如下图所示，在第二层，也就是 MIN 层，当计算到第三个节点的时候，已知前面有一个 3 和一个 6，也就是最小值为 3。 在计算第三个节点的时候，发现它的第一个孩子的结果是 5，因为它的孩子是 MAX 节点，而 MAX 节点是会选择最大值的，那么此节点的值不会比 5 小，因此这个节点的后序孩子就没有必要计算了，因为这个节点不可能小于 5，而同一层已经有一个值为 3 的节点了。



我们的问题如果用博弈搜索来解决，广度很大，深度不一定很大而且越深越不可靠。AB 剪枝优化基本是针对需要搜比较深的时候才有好的优化效果，而我们迫切解决的是子节点太多（广）的问题。

由于我们的问题连续性很好（状态改变较少时，下一步状态改变也较少），所以可以考虑用二分搜索的思想来进行优化搜索。由于下部节点更加不可靠，我们可以考虑采用估值上

部占比较大、下部占比较小的方式，（比如没向下一层乘以一个小于 0 的衰减因子），代替随机因子。也可能有其他方案。当确定了侧壁上反射的通用模型，就是两个相距为 d = 1800000 的带，都分布着间距为 3600 的点。每次发球都是两个点连线。有一确定算法可以把这个模型上的点（镜像点）映射为真实的点坐标。

1.1.2 蒙特卡洛搜索

在放弃了 AB 剪枝之后，我们的一个选择是蒙特卡洛树搜索。更确切的说是蒙特卡洛算法及其变种，包括信任度上限树算法 UCT 等可能更适用于我们的游戏的算法。正如前面所说，AB 剪枝对于我们的问题是不适用的，这时候本质上基于随机的蒙特卡洛树和能够频繁访问高效子节点的 UCT 对于我们需要解决的大广度可估值问题可能会有很大帮助。因为根据场地信息，每一次击球的可选范围都在数千的数量级上，而在当时我们还没有完全确定打角的必要性，非打角的防御策略还在开发，因此没有办法缩小搜索范围。

另一方面，UCT 算法可以说是兼顾了深度和广度两方面因素，对“最有希望子节点”的深搜结合我们的问题的连续性（在上个部分中也出现了连续性问题，事实上在第一次算法组的小组讨论中我们就利用镜像模型确定了在镜像区域内体力消耗和 y 速的连续相关），很有可能能够提高算法的效率和准确度。

1.1.3 启发式算法&估值函数

在考虑进行蒙特卡洛树搜索的同时我们也和自己的陪练进行了多次试验，这让我们想到了利用试验结果进行启发式方法尝试。因为打角的陪练往往比搜索到的不打角的解更有效，另一方面我们的非打角防御策略始终没有进展，所以一个经验的分析是打角不可避免。而另一个经验性的结论帮助我们把估值函数确定了下来。如果计算我们每次把球打回去对面的体力消耗，求最大值，那我们可能消耗了对方的体力但是自己损耗更大；把每次击球作为一个回合，计算我们的体力消耗和对面的消耗差；如果计算连续两个回合，就能形成一个循环，我方与对方各击球一次……依次类推，找到一个合适的回合数内的体力消耗差值，这就是我们的估值函数。这个可选的回合数将由我们的运算时间限制。

事实上在热身赛之前我们用模拟计算（即预先的经验）证明了打边的必要性。在每个接球回合中，球打过来的 y 方向速度已知，记为 v_0，到达的位置设为 y_0，根据老师提供的游戏代码，改变球速的体力耗费参数为 F_1=20，跑位体力耗费参数 F_2=20,000，根据先前的数据统计和计算，对手的跑位参数 k=0.67，单位球速改变引发的 y 方向位移单位 S=1,800，根据镜像方法，记 4 个镜像场地为-2H 到-H，-H0，H 到 2H，2H 到 3H，而原场地就在 0 到 H 上。

根据这样的数据，我们可以进行一次模拟计算：

减少体力值 lose= $(\frac{\frac{y-y_0}{s}-v_0}{F_1})^2$ = $(\frac{y-y_0-Sv_0}{sF_1})^2$

对方减少体力值

-2H<y<-H 时 $\mathrm{op\_lose} = K\left(\frac{A-y-2H}{F_2}\right)^2$；

-H<y<0 时 $\mathrm{op\_lose} = K(\frac{A+y}{F_2})^2$；

0<y<H 时 $\mathrm{op\_lose} = K(\frac{A-y}{F_2})^2$；

H<y<2H 时 $\mathrm{op\_lose} = K\left(\frac{A+y-2H}{F_2}\right)^2$

2H<y<3H 时 $\mathrm{op\_lose} = K\left(\frac{A-y+2H}{F_2}\right)^2$

在一个回合中对面体力损失和我方体力损失之差为二次函数，化简后为

f(v)=op_lose-lose= $\left(\frac{K}{F_2^2}-\frac{1}{s^2F_1^2}\right)y^2+Ay+B$ （A,B 为系数）

化简第一项得到

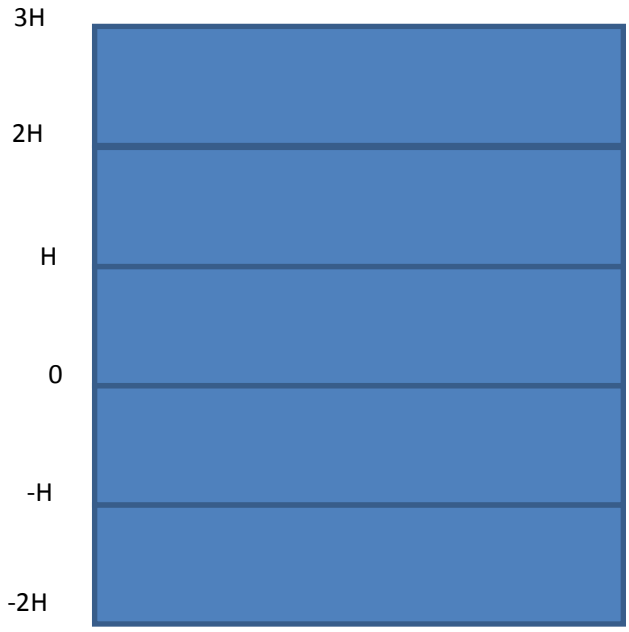
$\frac{1}{F_2^2}\left(K-\left(\frac{10}{18}\right)^2\right)$

因此二次函数开口向上，取最值必然在两边界，而反映在游戏中就是打边，即打边不可避免。而上面的推导过程中，我们使用了 lose-op_lose 作为估值，实质上就是一个回合的估值，这只是一种简化，而利用二次函数的性质是因为本质上是要讨论在区间上函数的最大值问题。在算法中我们使用了两个连续的回合进行累加，进行两层的最大最小值搜索。因此这就是我们的估值函数，计算连续两回合的体力差值作为估值。

1.1.4 基于 pandas 的博弈树搜索

速度策略（如何进行搜索）

我方在合法速度区间里取一系列速度，对于每一个我方选取的速度，替对方在对方的合法速度区间内决策出对对方最有利的速度返回。我方根据我方替对方决策（预测对方）的结



果，计算下一回合双方体力值耗损差值（估值函数）。取估值最大值所对应的速度作为我方此回合最终决策的回球速度。

跑位策略（基于搜索的结果，得到最佳的跑位）

当我们确定了 pandas 作为计划的一部分后，由于我们可以搜索两层，也就是可以得到预测对方打回的落点，因此我们决定将该落点和现在位置的中点作为跑位点。

在最后两天，我们观察了一些对局，发现这种跑位经常预测不准。我们参考了热身赛中某些强组的跑位方式，打算使用历史记录辅助跑位。考虑到最近的信息最为重要，我们使用了历史十次的位置加权平均的方式，权重使用 1-10。实际测试效果发现，对于轮流打边的算法对局，效果很差，因为会出现不明的浪费情况。后来我们又增加了跑位禁止越过中点的判断，效果仍然不理想。最终，没有采用。

此外，我们还打算使用安全区策略。所谓安全区，指的是体力值在 55500 以下，27500 以上，跑位到该位置可以保证绝对不会 miss_ball 的位置。当我们运算出的跑位点在该安全区范围之外时，即跑位到这一点。另外，还有所谓次安全区的概念，即体力在 27500 以下时，跑位到中间区域不能回到两边，而在某个位置可以保证至少保住一边。实际上，我们测试时发现，即使增加了安全区判定，也会出现 30000-40000miss_ball 的情况。我们组后来发现，这是因为我们是基于当前回合 tb 的数据判断的，而实际跑位时，我们可能经历了掉血包、加速的消耗，实际体力值要小于 tb 中的数据，所以这样判断出的安全区是一定偏大的，而偏大的安全区没有任何作用。考虑到时间紧迫，我们采用了减去一个常数（最终设定为 4000）

的方式，实际上这样也是不准确的，安全区偏小，这样就导致在接近 27500 时，转向了次安全区策略，导致出现失败。尽管如此，我们还是把安全区挂载到了 Series 和 Siderplus 上。

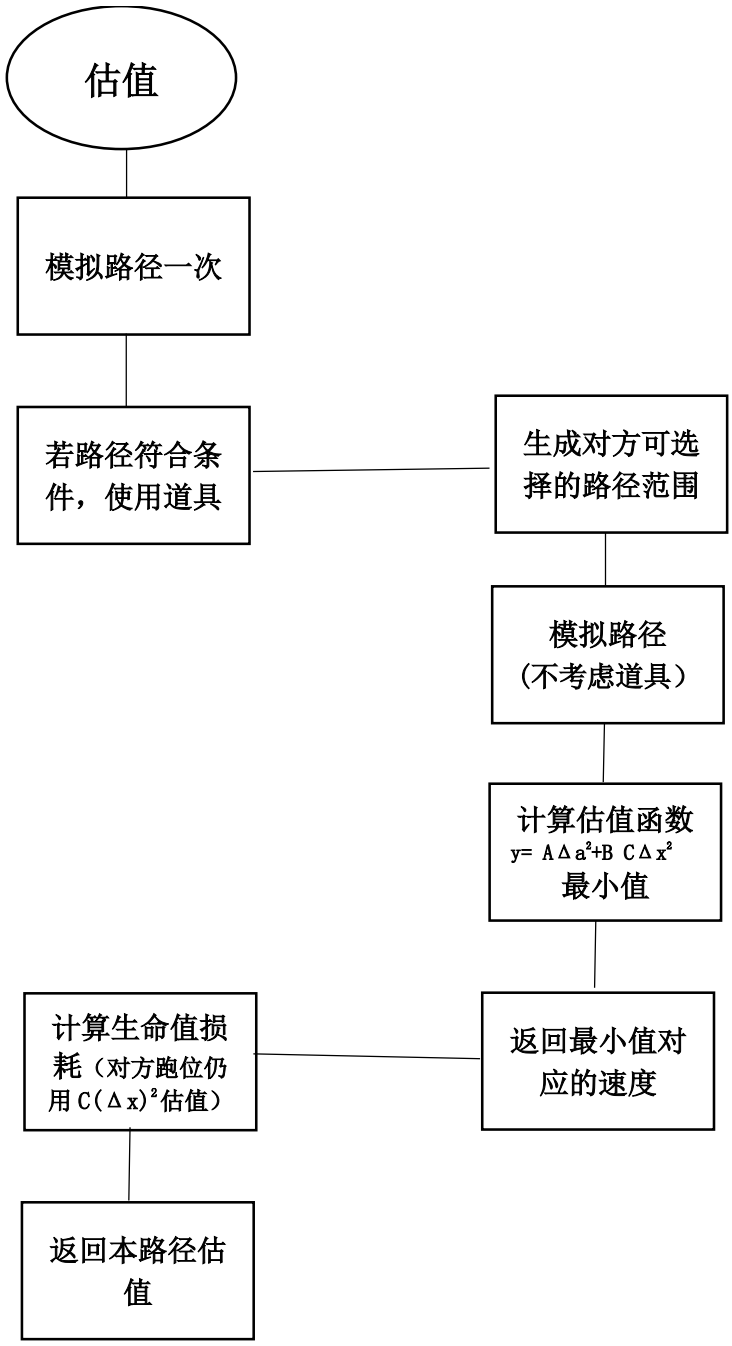
道具系统（根据路径上是否有道具，对估值进行加分）

①道具获取：将道具分为主动攻击型道具（旋转、变压、瞬移）和防守型道具（加血、减血）以及无用道具（隐身）。对能获取道具的路径（速度）进行一定的加分，在体力值额外消耗不小于预期收益的情况下选择获取道具路线。主动攻击型道具的预期收益会随着体力值下降而减小，这是出于避免出现“送死”情况而采取保守策略的考虑。

②道具使用：将道具分为主动攻击型道具（旋转、变压、瞬移）和防守型道具（加血、减血）以及无用道具（隐身）。主动攻击型道具的使用决策在估值函数中完成，并按照一定的优先级使用。防守型道具由于收益的固定，其使用决策在估值结束后决定。无用道具不考虑主动获取，并且及时抛出以空出道具箱。

③道具应对：道具应对包括两方面，一是对旋转和隐身的必要处理，以免对方使用时出现 bug；二是针对对方道具使用的决策变化。由于我方估值函数整体基于预期体力值消耗差，因此只需在体力计算时考虑道具使用即可。

1.2 算法流程图

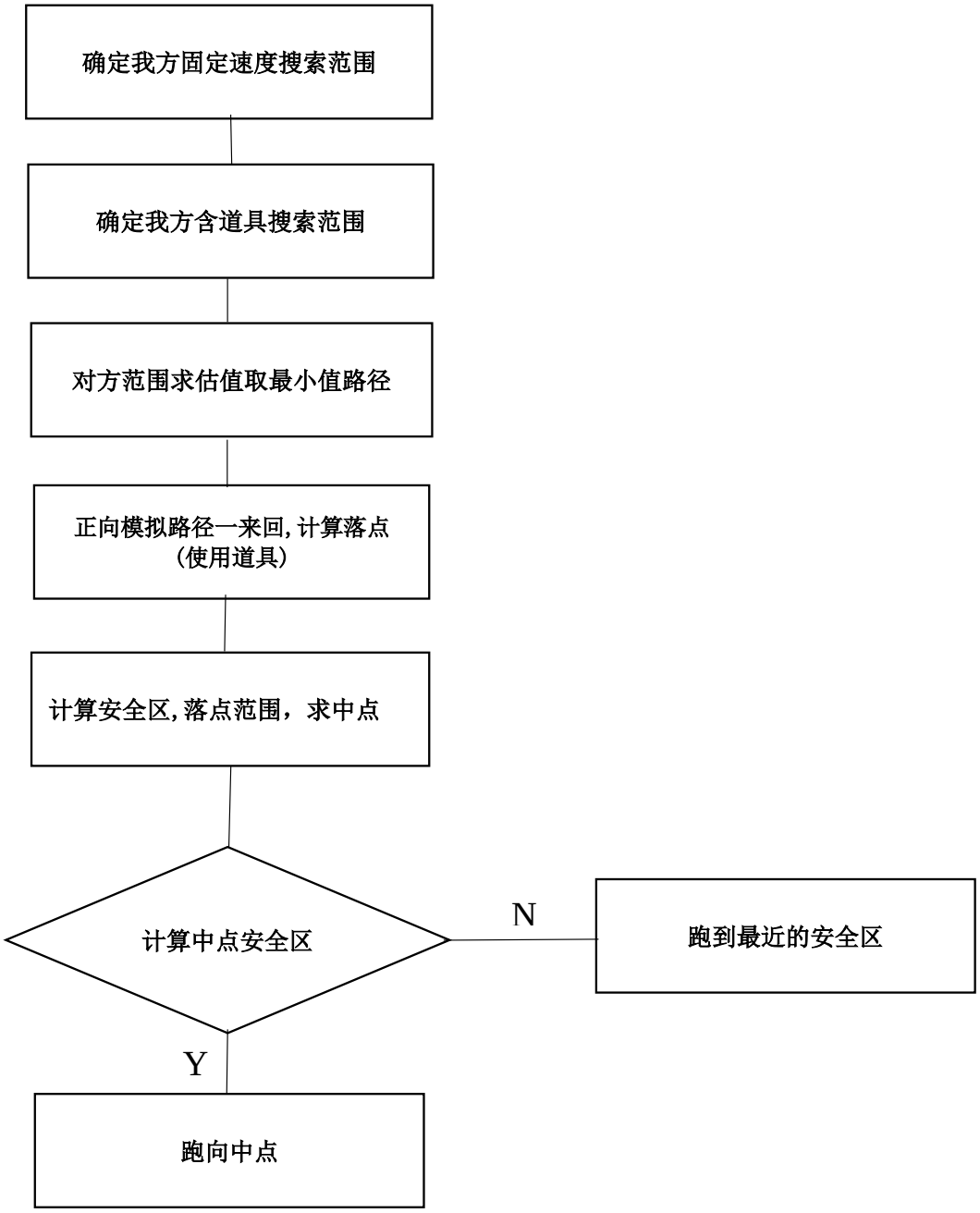


注：Δa：加速度

Δx：上次击球到下次击球的位移差

A、B：系统常数

C：我方设定常数，最后取值为0.67



1.3 算法运行时间复杂度分析

作为博弈树的极大极小搜索，算法的时间复杂度是 O(b^d)，b 为子节点数量，d 为搜索深度。这个游戏中，不考虑道具的“常规”情景，实际需要决策的量只有两个：出射速度和跑位位置。而我们的估值函数实际上把跑位和迎球部分合成，利用两端位置的确定性，得到一个估计值，所以实际上我们不需要考虑跑位，只需要判断速度。合法速度区间可以取值大约有 2200 个左右，我们计划采用等距取点法，将速度区间压缩为 500 个左右。我们初始计划是：需要搜索的树广度很大，因此不可能搜索地很深，可能也就 2 层左右。因此，我们抛弃了树形数据结构，期望用普通的函数调用降低代码维护成本。时间复杂度在 O(b^2)

而实际上，由于我们计划变更，使用了 pandas（效率高 Python 原生结构 2 个数量级）

以加速计算，实际运算时间在百毫秒级（每回合），实际并没有达到我们期望的高效程度。我们知道 pandas 虽然运算快，但是建立数组的过程相对较慢。基于这个认识，在 demo 开发中，我尝试将用到的 pandas 数组定义在 play 函数的外部，作为全局常量引用，不占用算法时间。但是实际的开发中，由于为了使用之前的辅助函数减少工作量，以及我们后来讨论时没有提及这件事，导致我们把生成数组的时间放在了 play 内部，实际相当于浪费了相当多运算时间，让进一步改进代码成为不可能的事。我们尝试使用 pandas 作为必杀器，但是并没有发挥其最大的威力。

2 程序代码说明

2.1 数据结构说明

元组：便于代码中一些数据处理，创建要调用的各种原始数据元组，代码中大多以 tb_开头。也有部分为了表示的更直观，用了其他命名，比如 cards 等。

Series：这是我们主要使用的数据类型，因此做一个较为详细的分析。Series 是 pandas 的数据函数类，是一种类似于一维数组的对象，它由一组数据（各种 NumPy 数据类型）以及一组与之相关的数据标签（即索引）组成。Series 的字符串表现形式为：索引在左边，值在右边。
我们之所以选择使用 Series 这一数据类型，主要考虑是其运行速度和对数据的处理能力。pandas 就是基于 NumPy 的一个 python 数据分析包，主要目的是为了数据分析。它提供了大量高级的数据结构和对数据处理的方法。
这里摘取书中一句话：”The Series is the primary building block of pandas and represents a one-dimensional labeled array based on the NumPy ndarray.” 大概意思是说：“说 Series 结构是基于 NumPy 的 ndarray 结构，是一个一维的标签矩阵。”感觉有一点想 python 里的字典类型，但比字典更加强大。

树：在估值函数中部分涉及，但是最终代码中没有体现。如果这是一个规则固定且简单没有随机性的游戏，比拼将集中于搜索的深度，就像棋类比赛。但由于道具的加入增加了随机性和策略多样性限制了搜索的准确性或者说难以实现，加之时间的限制，我们最终放弃了深搜，同时认定对手只会采取简单的策略判断并不善于获取道具及使用，由此也放弃了我们最开始考虑的一些搜索算法。

2.2 函数说明

```
主函数：
def play(tb: TableData, ds) -> RacketAction:
    """
    主程序 play，反复调用
    :param y0: int，打到我方底线时球的 y 轴坐标
    :param v0: int，打到我方底线时球的 y 轴速度
```

```
    :param p_v: Series，我方可能的出射速度
    :param y1: Series，打到对方底线时球的 y 轴坐标
    :param v1: Series，打到对方底线时球的 y 轴速度
    :param op_chosen_v: Series，对手最终决定的回球方式
    :param p_life1, op_life1: 对方上一轮迎球加速跑位加减血道具生效而我
方尚未作出迎球反应时双方体力值
    :param p_life2, op_life2: 决策后双方体力值
    :param path_assume: Series，我方的估值函数值
    :return: RacketAction，我方最终决定的回球方式
    """
```

创建过程中要调用的各种原始数据元组。（最开始全部使用 tb 来传输数据，后来为了调用方便和阅读直观创建了几个数据类；再后来认为没有必要，最终改为元组调用，然后每个子函数在函数开头自行命名变量赋值，方便直观。）

```
# 创建 ball_data 元组
b_d = (tb.ball['position'].x, tb.ball['position'].y,
tb.ball['velocity'].x, tb.ball['velocity'].y, tb.tick)
# 创建 player_data（迎球方）和 op_player_data（跑位方）元组
p_d = (tb.side["life"], tb.side['position'].y, tb.side['cards'])
op_d = (tb.op_side["life"], tb.op_side["active_card"],
tb.op_side['position'].y)
# 创建 cards 元组
cards = (tb.cards['card_tick'], tb.cards['cards'])
# 获取“安全区”坐标范围
safe_zone = get_run_side(tb)
#
————————————————————————————————————————————
————————————————————————#
y0, v0 = b_d[1], b_d[3]
# cards_available:桌面上现有的道具列表，元素为 Card 类
cards_available = cards[1]
op_active_card = op_d[1]
# 对方上一轮迎球加速跑位加减血道具生效而我方尚未作出迎球反应时双方体
力值
p_life1, op_life1 = p_d[0], op_d[0]
# 我方拥有的道具 list,为 CardBox 类
p_cards = p_d[2]
v00, v01, v02, v03 = ball_v_range(y0)
```

创建我方回球速度系列(p_v:pd.Series)。首先调用 ball_v_range 函数获取合法速度区间边界，再等距获得一系列合法速度。调用 ball_fly_to_card 函数获取能击中桌面上存在道具的速度二维列表，再将这些速度补充到 p_v 中。

```
p_v = pd.Series([i for i in range(v03,v02,20)] + [j for j in
range(v01,v00,20)])

# v_will_hit:list，其元素为 list，即为符合击中某道具要求的竖直速度群，
v_will_list 的元素和 cards_al 的元素是对应关系
v_will_hit = ball_fly_to_card(b_d, cards_available)
for vi in v_will_hit:
    p_v = p_v.append(pd.Series(vi), ignore_index = True)
# v_will_hit = []
# for card_i in cards_available:
#     v_will_hit.append(p_v[get_card(b_d, p_v, card_i)].tolist())
```

调用 ball_v_range 函数获得我方回球达到对方底线时的速度与坐标
(y1:pd.Series,v1:pd.Series)。调用 op_play 函数获取我方预测对方面对 y1,v1 时做出的回球决策，也即选择回球速度(op_chosen_v:pd.Series)。（预测中没有考虑对方使用道具以及获取道具的情况而仅仅考虑体力值消耗。这符合我们对一般对手策略的猜测。如果这是一个规则固定且简单没有随机性的游戏，比拼将集中于搜索的深度，就像棋类比赛。但由于道具的加入增加了随机性和策略多样性限制了搜索的准确性或者说难以实现，加之时间的限制，我们最终放弃了深搜，同时认定对手只会采取简单的策略判断并不善于获取道具及使用，由此确定了我方算法的合理性。）

```
# y1: Series 为到达对方时球的 y 轴坐标，v1: Series，为到达对方时球的 y
轴速度
# op_chosen_v 为对方回球的 y 轴速度，为 Series
y1, v1 = ball_fly_to(y0, p_v)
op_chosen_v = op_play(y0, y1, v1)
```

调用 p_life_consume 函数返回模拟决策后对应每种路径的我方的估值以及道具使用情况。调用 op_life_consume 函数返回模拟决策后对应每种路径的对方的估值。p_life2,op_life2,p_active_SPIN, p_active_AMPL, p_active_TLPT 都是 Series，其中 p_life2,op_life2 元素是 int 类型，p_active_SPIN, p_active_AMPL, p_active_TLPT 的元素是布尔值类型，表示 CARD_SPIN,CARD_AMPL,CARD_TLPT 在一系列 p_v 的模拟决策中道具使用情况。

```
# 计算双方决策后的“体力值”
# 对于我方而言，不等同于体力值，因为加入了获得道具的“加分”。而且为了
避免“吃老本”的情况，没有考虑加减血包的使用。
# 对对方而言，也不等同于体力值，因为是只考虑一小部分的体力估值
p_life2, p_active_SPIN, p_active_AMPL, p_active_TLPT =
p_life_consume(b_d, p_d, op_d, cards_available, p_v, op_chosen_v, y1,
v_will_hit)
# 补充：我的考虑是，如果我方使用旋转球，对方会采取简单反弹（不秒杀）或
者最小改变 min(ball_v_range)
op_life2 = op_life_consume(op_d, y0, op_chosen_v, y1, v1, p_active_SPIN,
p_active_AMPL)
```

利用之前调用 p_life_consume 函数和 op_life_consume 函数返回的下一回合双方体力消耗预期（以及估值加分，所以实际上还带有一定预测几回合之后的体力消耗预期的能力），计算得到估值函数 p_path_assume:pd.Series。使用 Series 的内置函数 argmax 获取 Series 里最大值的索引。设置我方具体打球方式的变量以及初始值。

```
# 计算双方决策后的“体力值”
# 对于我方而言，不等同于体力值，因为加入了获得道具的“加分”。而且为了避免“吃老本”的情况，没有考虑加减血包的使用。
# 对对方而言，也不等同于体力值，因为是只考虑一小部分的体力估值
p_life2, p_active_SPIN, p_active_AMPL, p_active_TLPT =
p_life_consume(b_d, p_d, op_d, cards_available, p_v, op_chosen_v, y1,
v_will_hit)
# 补充：我的考虑是，如果我方使用旋转球，对方会采取简单反弹（不秒杀）或者最小改变 min(ball_v_range)
op_life2 = op_life_consume(op_d, y0, op_chosen_v, y1, v1, p_active_SPIN,
p_active_AMPL)

# 获得我方估值函数值中的最大值对应索引
p_path_assume = (p_life2 - p_life1) - (op_life2 - op_life1)
index = p_path_assume.argmax()
# 返回具体打球方式
p_chosen_side = None
p_chosen_card = None
p_chosen_v = p_v[index]
```

对道具使用策略的补充，尽快使用加减血包，并扔掉不小心获取的隐身。

```
if p_active_SPIN is not None and p_active_SPIN[index]: # equal if
CARD_SPIN in p_cards and p_active_SPIN[index] :下同
    p_chosen_side = 'OPNT'
    p_chosen_card = CARD_SPIN
if p_active_AMPL is not None and p_active_AMPL[index]:
    p_chosen_side = 'OPNT'
    p_chosen_card = CARD_AMPL
if p_active_TLPT is not None and p_active_TLPT[index]:
    p_chosen_side = 'SELF'
    p_chosen_card = CARD_TLPT

# 如果没有使用道具而恰好有加减血包，现在是优先使用减血包，属于攻击性策略。
if p_chosen_side == None and p_chosen_card == None:
    if CARD_DECL in p_cards:
        p_chosen_side = 'OPNT'
```

```
        p_chosen_card = CARD_DECL
    elif CARD_INCL in p_cards:
        p_chosen_side = 'SELF'
        p_chosen_card = CARD_INCL
    else:  # 扔掉隐身术
        if CARD_DSPR in p_cards:
            p_chosen_side = 'SELF'
            p_chosen_card = CARD_DSPR
        elif CARD_SPIN in p_cards:
            p_chosen_side = 'SP'
            p_chosen_card = CARD_SPIN
```

调用历史数据，决策跑位。

```
# 跑位历史初始化
if tb.tick <= 1800:
    initial_ds(ds)
# 根据历史，加权进行跑位
save_catch_pos(tb, ds)
final_run_pos = get_run_pos(tb, ds)
# 判断安全区，如果在安全区外，则选择最近的一个安全区。
if safe_zone[1] == 2:
    if not(safe_zone[0][0] < final_run_pos < safe_zone[0][1]):
        final_run_pos = min(safe_zone[0], key = lambda pos: abs(pos -
final_run_pos))
elif safe_zone[1] == 3:
    if (safe_zone[0][0] < final_run_pos < safe_zone[0][1]):
        final_run_pos = min(safe_zone[0], key = lambda pos: abs(pos -
final_run_pos))
```

返回最终决策的 RacketAction。

```
return RacketAction(tb.tick, tb.ball['position'].y -
tb.side['position'].y,
                    (p_chosen_v - v0) * (2 if op_active_card == CARD_SPIN
else 1),
                    final_run_pos - tb.ball['position'].y,
                    p_chosen_side,p_chosen_card)
```

op_play 函数：

```
def op_play(y0:int, y1: pd.Series, v1: pd.Series):
    """
    面对(y1,v1)，对手的选择
```

```
    :param y1: pd.Series，打到对手底线时球的 y 轴坐标
    :param v1: pd.Series，打到对手底线时球的 y 轴速度
    :param op_v: Series，对手可能的回球方式
    :param op_assume: Series，对手可能的回球方式的估值函数值
    :return: int，对手最终决定的回球方式
    """
```

首先创建对方回球速度系列(op_v:pd.Series)。（这里是一个我们忘记解决的 bug，range 里 0,100,300,400 应该是 y1 对应的边界速度。但是由于边界速度是 Series 而不能简单套用在适用 int 的语句里面，所以暂时我并不知道如何修改。事实上，代码的实现过程中，最大的困难除了理清算法逻辑和函数调用问题外，就是如何实现将适用于 int 等一个数据的函数改为适用于 Series。这项工作耗费了大量的时间和信心。）创建对方回球速度系列的下一行代码调用了 op_assume_f 函数，这里面有（我们假设）对方用来决策的估值函数。这里首先对 op_v 使用了 apply 函数，这是因为 op_assume_f 函数的编写逻辑是适用于 int 的，而此处 op_v 是 Series。（这里简介一下 apply 函数，这是 Series 的一个内置函数，第一个参数为调用的函数名，后续关键字参数则会被传入此调用参数。apply 函数实现了将一个原本适用于 int 等单值的函数广播作用于 Series 里每一个元素的功能。）

如何将一个 DataFrame 压缩为一个 Series，Series 中每个元素都是被压缩轴上的最大值？我们通过使用匿名函数搭配 apply 解决了这个问题。idmax 和 swapaxes 是 Series 的内置函数。
（http://pandas.pydata.org/pandas-docs/stable/generated/pandas.Series.html 可以在这个网页上使用 Ctrl+F 查找，就可以找到函数详细介绍。）（被注释掉的代码犯了两个错误：压缩轴搞错和返回值搞错。事实上不必要使用 swapaxes，只要把作用轴换为 0 即可。没被注释掉的两行代码等价。）

```
# TODO 改系数，获得合理的等距 v 的 Series
op_v = pd.Series([i for i in range(0,100,20)] + [j for j in
range(300,400,20)])
op_chosen_v = (op_v.apply(op_assume_f, y1=y1, v1=v1, y0=y0)).swapaxes(0,
1).apply(lambda x : op_v[x.idxmax()],axis = 1)
# op_chosen_v = pd.Series(op_v[op_assume.argmax()])
return op_chosen_v
```

op_assume_f 函数：

```
def op_assume_f(op_v:int, y1: pd.Series, v1: pd.Series, y0:int)
->pd.Series:
    """
    只会想一层的可爱对手使用的估值函数
    对方只考虑我方为接此球移动消耗体力与每次击球所消耗体力的差值最大
    :param op_v: Series，对手回击后球的速度
    :param v1: Series，打到对方底线时球的 y 轴速度
    :param y1: Series，打到对方底线时球的 y 轴坐标
```

```
    :param y2: Series，打到我方底线时球的 y 轴坐标
    :return: Series，估值函数值（我方为接此球移动消耗体力与每次击球所消耗体力的差值）
    """
```

分析一下 op_assume_f 函数，将对方加速消耗体力与我方跑位迎球消耗体力差值作为（我们预测的）对方的估值函数，这是符合我们对对方的策略选取认知的。可以发现，当 op_v 看做一个 int 时，函数返回 Series 类型。这意味着，把这个函数 apply 在实际为 Series 的 op_v 上时，会形成一个 DataFrame。当把 op_v 看做 int 时，由 op_v 返回的 Series 对应了 y1:pd.Series,v1:pd.Series 的估值函数。但事实上，按照逻辑，我们想要的是当把 y1,v1 看做 int 时对应的 op_v:Series，得出他们共同确定的估值 Series，返回这个 Series 里最大值对应的 op_v 中的元素。

```
# 对方将球从 v1 加速至 op_v 所消耗体力
op_lose = (((op_v - v1) / FACTOR_SPEED) ** 2).apply(int)
# 我方将从 y0 跑位至 y0 和 y2 之间的某一点，并到 y2 处迎球
y2 = mirror2real(y1 + 1800 * op_v)[0]
# C 为可调试的常数，取值范围 0.5-1，
p_lose = (C * ((y0 - y2) / FACTOR_DISTANCE) ** 2).apply(int)
return p_lose - op_lose
```

```
p_life_consume 函数：
def p_life_consume(b_d:tuple, p_d:tuple, op_d:tuple, cards_available:
list, p_v: pd.Series, op_chosen_v: pd.Series, y1: pd.Series, v_will_hit):
    """
    根据我方此次决策，算出迎球+加速+跑位的总体力消耗(考虑道具)
    :param p_d: 决策前迎球方的信息
    :param op_d: 决策前跑位方的信息
    :param cards_available: list, 决策前桌面上可供获取的道具信息
    :param player_action: 我方此次决策结果 RocketAction 类
    :param bat_distance: 此次决策指定迎球的距离
    :param run_distance: 此次决策指定跑位的距离
    :return: 执行完决策后我方体力值消耗
    """
    # 我方拥有的道具 list,为 CardBox 类
    p_cards = p_d[2]
    p_life = pd.Series(p_d[0],range(op_chosen_v.size))
    # op_ini_pos 对手跑位前位置
    op_active_card, op_ini_pos = op_d[1], b_d[1]
    y0, v0 = b_d[1], b_d[3]
```

然后是对道具获取的处理，这里的估值加分需要在对局中调整。（可惜由于前期算法得

出以及代码实现和 debug 消耗了我们大量的时间精力，我们无力再去对参数进行细致调整，虽然这对于估值搜索算法而言本应该是极其重要的，会很大程度影响算法表现。）

```
# 以下对道具获取路径额外加分，对击中不同的道具的 p_v 路径给予不同的 p_life“加分”，以便估值函数能显示出走这条能获得道具的路更有益
# 由于最后计算 p 和 op 的 life 差值，op 的减分统一加在 p_life 上，PS：card_i 是 Card 类。
# 目前的加分是在体力 40000 以下就和体力成正比
for i in range(len(cards_available)):
    card_i = cards_available[i]
    if card_i == CARD_INCL or card_i == CARD_DECL:
        # CARD_INCL_PARAM 和 CARD_DECL_PARAM 都为 2000
        health_change = 4000
    elif card_i == CARD_TLPT:
        health_change = 1500 if(p_d[0] >= 40000)else(1500 * p_d[0] / RACKET_LIFE)
    elif card_i == CARD_AMPL:
        health_change = 1500 if(p_d[0] >= 40000)else(1500 * p_d[0] / RACKET_LIFE)
    elif card_i == CARD_SPIN:
        health_change = 2500 if(p_d[0] >= 40000)else(2500 * p_d[0] / RACKET_LIFE)
    else:
        health_change = 0
    # saved = p_life
    p_life[p_v.apply(lambda x: x in v_will_hit[i])] += health_change
    # print(p_life)
    # test = (p_life - saved)
    # # print(p_v)
    # print(p_v.apply(lambda x: x in v_will_hit[i]))
    # print(p_life[p_v.apply(lambda x: x in v_will_hit[i])])
    # # print(health_change)
    # print(test[test != 0])
```

此 TODO 基本完成，在 play 主函数里实现了根据历史位置加权影响的跑位策略和搜索的跑位策略。

```
# 我方决策跑位到 对方决策后在我方的落点 与 当下位置的中点
# TODO 跑位需要修改，加入随机因素，或者历史搜索
middle = (y2 + p_d[1]) // 2
```

下图是道具使用决策，在预期收益高于设置的阈值时，我方将调用道具。同理，这里的参数也应当根据实战调整。

```
    # TODO 道具的具体使用的阈值调整以及策略变动 注意！这里是道具使用！不是路径上的道具获取！
# 估值函数里不考虑加血包减血包的使用
# p_active_XXXX 为 Series，元素类型为 bool
# 旋转球的使用（如果秒杀则使用）
if CARD_SPIN in p_cards:
    p_active_SPIN = sec_kill(p_v, y0)
else:
    p_active_SPIN = None
# 变压器的使用（对方总移动超过一半）
if CARD_AMPL in p_cards:
    p_active_AMPL = (y1 - op_ini_pos).apply(abs) > Height * 0.5
else:
    p_active_AMPL = None
# 瞬移术的使用（跑位超过一半）
if CARD_TLPT in p_cards:
    p_active_TLPT = (middle - y0).apply(abs) > Height * 0.5
else:
    p_active_TLPT = None
```

设置我方回球动作（加速与跑位）以及移动速度，以便调用计算体力值损耗估值。

```
# player_action_bat = int(y0 - p_d[1]) # t0~t1 迎球的动作矢量（移动方向及距离）
player_action_acc = ((p_v - v0)* (2 if op_active_card == CARD_SPIN else 1)).apply(int) # t1 触球加速矢量（加速的方向及速度）
player_action_run = middle.apply(int) # t1~t2 跑位的动作矢量（移动方向及距离）
# player_action_card = (p_card_side, p_active_card)  # 对'SELF'/'OPNT'使用道具
```

下图是我方体力值的损耗估值计算过程。注意 run_distance 前后的注释，这里简单介绍一下 Series 的 where 函数。where 的第一个参数是一个与被作用的 Series 等 size 的 Series，元素类型为布尔值，如果布尔值为 True 则不改变被作用 Series，如果为 False，则将相对应的元素改为第二个参数的值。where 函数的作用有点像 if-else，是一个比较常用的函数。（起码这个程序里常用。）（另，跑位体力值损耗语句修改自老师源码。）

```

# 球拍的全速是球 X 方向速度，按照体力值比例下降
velocity = ((p_life / RACKET_LIFE) * BALL_V[1]).apply(int)
# bat_distance 为指定迎球的距离
# bat_distance = player_action_bat

# 如果指定迎球的距离大于最大速度的距离，则丢球，比赛结束
```

```
# if abs(bat_distance) > velocity[0] * STEP:
#     return False
# bat_distance.apply(abs).where(bat_distance.apply(abs) <= velocity *
STEP, 10000000)

# 按照迎球的距离减少体力值（考虑对方之前可能使用变压器道具）
# p_life -= (abs(bat_distance) ** 2 // FACTOR_DISTANCE ** 2) *
(CARD_AMPL_PARAM if op_active_card == CARD_AMPL else 1)
# 按照给球加速度的指标减少体力值（考虑对方之前可能使用变压器道具）
p_life -= (player_action_acc ** 2 // FACTOR_SPEED ** 2) * (CARD_AMPL_PARAM
if op_active_card == CARD_AMPL else 1)

# 如果指定跑位的距离大于最大速度的距离，则采用最大速度距离
run_distance = player_action_run.apply(sign) * \

player_action_run.apply(abs).where(player_action_run.apply(abs) <
velocity * STEP,velocity * STEP)
# \后面的代码作用于 int 时为 min(abs(player_action_run), velocity * STEP)

# 按照跑位的距离减少体力值（考虑我方可能使用瞬移卡道具）
param = 0
if p_active_TLPT is not None: # 如果使用瞬移卡，从距离减去
CARD_TLPT_PARAM 再计算体力值减少
    param = CARD_TLPT_PARAM

a = (run_distance.apply(int) - param) ** 2 // FACTOR_DISTANCE ** 2


b = run_distance.apply(sign) - param
p_life -= a.where(b > 0, 0)
# 返回执行决策后 p_life,p_card_side 和 p_active_card
return p_life, p_active_SPIN, p_active_AMPL, p_active_TLPT
```

op_life_consume 函数：

```
def op_life_consume(op_d, y0, op_chosen_v, y1, v1, p_active_SPIN,
p_active_AMPL)->pd.Series:
    """
    根据我方此次决策，算出迎球+加速+跑位的总体力消耗(考虑道具)
    :param p_active_SPIN: Series，不同路径使用 CARD_SPIN 的情况，元素为
bool
    :param p_active_AMPL: Series，不同路径使用 CARD_AMPL 的情况，元素为
bool
    :param p_active_TLPT: Series，不同路径使用 CARD_TLPT 的情况，元素为
```

```
bool
    :param op_d = (tb.op_side["life"], tb.op_side["active_card"])
    :return: 执行完决策后对方体力值
    """
    op_life = op_d[0]
    op_vy = op_chosen_v
    op_y = y1

    # 对方决策在我方处的落点
    y2, v2 = ball_fly_to(op_y, op_vy)
    # 对方跑位到我方落点与当下位置中某一点
    middle = Ct * (y2 - op_y) + op_y

    # 获取对方可能的决策结果 RacketAction(7200, Ct * (y2 - y0) + y0,
op_vy - v2, middle, None, None) 下文中 tick 没用，随便取一个合法值
    # 将迎球方动作中的距离速度等值规整化为整数
    op_player_action_bat = (Ct * (y2 - y0) + y0).apply(int)  # t0~t1
迎球的动作矢量（移动方向及距离）
    min_op_vy = ((Height - y1) // STEP).where(ball_fly_to(y1, v1)[0] >
(Height/2), - y1 // STEP)
    if p_active_SPIN is None:
        op_player_action_acc = op_vy - v1
    else:
        op_player_action_acc = (op_vy - v1).where((-p_active_SPIN),
min_op_vy).apply(int)
    # 这行代码是对我方使用旋转球的处理,相当于是从我们的视角替对方“纠
正”了自己的策略
    op_player_action_run = middle.apply(int)  # t1~t2 跑位的动作矢量(移
动方向及距离)
```

设置对方跑位和迎球动作。我们预测，当对方发现我方使用旋转时，采取在不出现 invalid bounce 前提下尽量少加速的策略。如果对方接不到某路径的球，那么必选此路径。人为将此路径估值变到足够大。再进行对方体力值损耗的计算。后文还有一些计算，最后返回一系列与我方回球速度相对应的路径对方体力损耗。

```
# 球拍的全速是球 X 方向速度，按照体力值比例下降
velocity = int((op_life / RACKET_LIFE) * BALL_V[1])

# TODO 如果指定迎球的距离大于最大速度的距离，则对方丢球，比赛结束，故
将此时对方迎球距离设为保证此路径估值最大。
op_player_action_bat.apply(abs).where(op_player_action_bat.apply(abs)
<= velocity * STEP, 10000000)

# 按照迎球的距离以及给球加速度的指标减少体力值(考虑我方可能使用变压器
```

```
道具）a,b 是为了简化式子的无意义临时变量
a = op_player_action_bat.apply(abs) ** 2 // FACTOR_DISTANCE ** 2 +
op_player_action_acc.apply(abs) ** 2 // FACTOR_SPEED ** 2
b = pd.Series(CARD_AMPL_PARAM, range(op_chosen_v.size))
if p_active_AMPL is None:
    op_life -= a
else:
    op_life -= a * b.where(p_active_AMPL, 1)
# 如果指定跑位的距离大于最大速度的距离，则采用最大速度距离
run_distance = op_player_action_run.apply(sign) * \

op_player_action_run.apply(abs).where(op_player_action_run.apply(abs)
< velocity * STEP,velocity * STEP)
# \后面的代码作用于 int 时为 min(abs(op_player_action_run), velocity *
STEP)

# 按照跑位的距离减少体力值（听说对方不用道具）
op_life -= run_distance.apply(abs) ** 2 // FACTOR_DISTANCE ** 2
# 返回执行决策后 op_life
return op_life
```

以下是辅助性的函数，实现以上主要函数所需的功能：
ball_fly_to（根据出射点坐标、出射速度，算出到达对侧位置速度）

```
以下是辅助性的函数，实现以上主要函数所需的功能：
def ball_fly_to(y: int, v: int) -> tuple:
    """
    根据出射点坐标、出射速度，算出到达对侧位置速度
    注意，虽然函数参数为 int，但是因为计算过程只涉及四则，y,v 皆可为
Series 并直接调用
    :param y0: 出射点坐标
    :param p_v: int，出射速度
    :param Y: int，镜像点 y 坐标
    :param count: 与桌碰撞次数，可正可负
    :return: 到达对侧位置位置、速度
    >>> ball_fly_to(0,50)
    (90000, 50)
    # Y 为没有墙壁时乒乓球到达的位置（镜像点）
    Y = y + v * STEP
    # 把镜像点转化为真实点
    yi = mirror2real(Y)[0]
    # 计算并更新 y 轴速度
    count = mirror2real(Y)[1]
    vi = v * ((count + 1) % 2 * 2 - 1) # 后面这一项，当 count 为偶数时
```

```
为 1，奇数时为-1
    return yi, vi
```

mirror2real（将镜像点映射为真实点）

```
def mirror2real(y_mr: int or np.array or pd.Series) -> tuple:
    """
    将镜像点映射为真实点.
    可以代入 int,np.array,pd.Series 等多种类型
    :param y_mr: 镜像点 y 坐标
    :return: 真实点 y 坐标数组（范围 0-1,000,000），碰撞次数数组
    """
    # 已完成测试
    if type(y_mr) is int:
        if y_mr % (2 * Height) < Height:
            y_real = y_mr % Height
        else:
            y_real = Height - y_mr % Height
        # 若球没有打在边界上
        if y_mr % Height != 0:
            count = y_mr // Height
        else:
            count = (y_mr // Height if (y_mr > 0) else (1 - y_mr // Height))
        return y_real, count
    elif type(y_mr) is np.array:
        # 以上代码用 np.array 改写形式
        y_real = np.where(y_mr % (2 * Height) < Height, y_mr % Height,
Height - y_mr % Height)
        count = np.where(y_mr % Height != 0, y_mr // Height,
                         np.where(y_mr > 0, y_mr // Height, 1 - y_mr //
Height))
        return y_real, count
    elif type(y_mr) is pd.Series:
        # 以上代码用 pd.Series 改写形式
        y_mr = y_mr.values
        # 复制 ndarray
        y_real = np.where(y_mr % (2 * Height) < Height, y_mr % Height,
Height - y_mr % Height)
        count = np.where(y_mr % Height != 0, y_mr // Height,
                         np.where(y_mr > 0, y_mr // Height, 1 - y_mr //
Height))
        return pd.Series(y_real), pd.Series(count)
```

ball_fly_to_card（根据桌面上现有的道具种类、位置和球所在的位置，计算吃到对应的各

个道具所需要的竖直速度）

```
def ball_fly_to_card(b_d: tuple, cards_al: list) -> list:
    """
    cards_al means cards_available
    根据桌面上现有的道具种类、位置和球所在的位置，计算吃到对应的各个道具所需要的竖直速度
    :param b_d: b_d = (tb.ball['position'].x, tb.ball['position'].y, tb.ball['velocity'].x, tb.ball['velocity'].y, tb.tick)
    :param cards_al: list，决策前桌面上道具的信息，元素是 Card 类
    :param height: 球桌宽度，DIM[3] - DIM[2]
    :param v_range: 满足规则（碰撞 1-2 次）的速度区间[v3,v2]∪[v1,v0]
    :return: 返回一个 list，其元素为 list，即为符合击中某道具要求的竖直速度群，vy_list 的元素和 cards_al 的元素是对应关系
    """
    # 已完成测试
    v_range = ball_v_range(b_d[1])
    # 吃到序号为 i 的道具需要的速度（所有可能的速度）保存在 v[i]中
    vy_list = [0] * len(cards_al)
    for i in range(len(cards_al)):
        vy_list[i] = fly_assistant(b_d, v_range, cards_al[i])
    # 返回吃到所有道具所需要的竖直速度，通过对应索引查找
    return vy_list
```

fly_assistant（根据某个确定的道具位置和球所在的位置，计算吃到它所需要的竖直速度）

```
def fly_assistant(b_d: tuple, v_range: tuple, card: Card) -> list:
    """
    根据某个确定的道具位置和球所在的位置，计算吃到它所需要的竖直速度
    :param b_d: b_d = (tb.ball['position'].x, tb.ball['position'].y, tb.ball['velocity'].x, tb.ball['velocity'].y, tb.tick)
    :param v_range: 满足规则（碰撞 1-2 次）的速度区间[v3,v2]∪[v1,v0]
    :param card: 是一个 Card 类
    :param x0: x0 = b_d[0]，接球时球的横坐标
    :param y0: y0 = b_d[1]，接球时球的纵坐标
    :param vx0: vx0 = b_d[2]，球的水平速度
    :return: 返回一个 list，元素为符合击中某道具要求的竖直速度

    """
    # 已完成测试
    x0, y0, vx0 = b_d[0], b_d[1], b_d[2]
    # 满足规则（碰撞 1-2 次）的速度区间[v3,v2]∪[v1,v0]
```

```
    v0, v1, v2, v3 = v_range
    # 要吃的道具共有五个可能的镜像点/真实点，置于列表 y 中
    y = [0] * 5
    y[0] = card.pos.y
    y[1] = 2 * Height + card.pos.y
    y[2] = 2 * Height - card.pos.y
    y[3] = -card.pos.y
    y[4] = -2 * Height + card.pos.y
    # 到达道具位置用时
    card_step = abs(x0 - card.pos.x) / vx0
    # 在能到达道具的一系列速度中挑选出属于区间 v_range 的，保存到列表中
并返回
    # 列表中的元素类型是 int
    return [vy for vy in
            filter(lambda v: (v >= v3 and v <= v2) or (v >= v1 and v <=
v0),
                   map(lambda x: (x - y0) / card_step, y))]
```

get_card（对于给定的球的数据 b_d 和给定的一个道具 card,返回一个布尔值，表示球能否吃到道具）

```
def get_card(b_d:tuple, p_v, card: Card, eps=2000) -> bool:
    """
    对于给定的球的数据 b_d 和给定的一个道具 card,返回一个布尔值，表示球
能否吃到道具
    :param b_d: b_d =
(tb.ball['position'].x,tb.ball['position'].y,tb.ball['velocity'].x,tb
.ball['velocity'].y,tb.tick)
    :param card: 是一个 Card 类
    :param eps: eps = 2000,表示能吃到道具的最近距离
    :param x0: x0 = b_d[0]，接球时球的横坐标
    :param y0: y0 = b_d[1]，接球时球的纵坐标
    :param vx0: vx0 = b_d[2]，球的水平速度
    :param vy0: vy0 = b_d[3],球的竖直速度
    :return: bool，True 表示能吃到给定道具，False 表示不能吃到
    """
    # 来自李逸飞学长的算法
    # 直线方程为 l:-v*x+u*y+v*x0-u*y0=0
    # card 经过多次对称后，位置为(x1,±y1+2*k*l)
    # 点到直线距离公式
dist=|-v*x1+u*(±y1+2*k*l)+v*x0-u*y0|/|self.velocity|
    # 记-v*x1+u*(±y1)+v*x0-u*y0=A1,A2,u*2*l 为 delta。则求最短距离，即
是求 A1%delta,A2%delta,-A1%delta,-A2%delta 中最小值，再除以
self.velocity 的模长
```

```
    x0, y0, vx0 = b_d[0], b_d[1], b_d[2]
    A1 = (p_v * (-card.pos.x + x0) + vx0 * (card.pos.y - y0))
    A2 = (p_v * (-card.pos.x + x0) + vx0 * (-card.pos.y - y0))
    delta = (2 * abs(vx0) * Height)
    # 实际上 p_v 是一个 Series，所以返回的也是元素为 bool 的一个 Series
    return (pd.DataFrame([A1 % delta, -A1 % delta, A2 % delta, -A2 %
delta]).apply(min) / ((vx0 ** 2 + p_v ** 2).apply(math.sqrt))) < eps
```

ball_v_range（根据我方出射点坐标，算出 y 轴可取速度的边界值）

```
def ball_v_range(y:int or pd.Series) -> tuple:
    # 已完成测试
    # v0,v1,v2,v3 是速度的范围边界:可取[v3,v2]∪[v1,v0]
    v0 = (3 * Height - y) // STEP
    v1 = (1 * Height - y) // STEP + 2
    v2 = (0 - y) // STEP
    v3 = (-2 * Height - y) // STEP + 2
    # 贴边打的情况算作反弹零次，需要排除
    # 只有 v2 才会和 y 同时取到零，v1 取到零的时候是不会贴边的
    if type(v2) is int:
        if v2 == 0:
            v2 = -2
    if type(v2) is pd.Series:
        v2 = v2.where(v2 != 0, -2)
    # 返回一个元组，依次为速度的四个边界值
    return v0, v1, v2, v3
```

sec_kill（根据我方出射点坐标速度，判断对方简单反弹时是否可以秒杀）

```
def sec_kill(p_v: pd.Series, y0: int) -> pd.Series:
    """
    根据我方出射点坐标速度，判断对方简单反弹时是否可以秒杀
    :param p_v:pd.Series，出射点坐标速度
    :param y0:int，出射点坐标坐标
    :return: pd.Series  元素为 bool 类型
    """
    # 已完成测试
    # 镜像点坐标。Y:pd.Series
    Y = y0 + STEP * p_v
    # 把镜像点 Y 转化为真实点，然后求合法速度区间
    y, count = mirror2real(Y)
    # 对方竖直速度范围 v_range: tuple 元素为四个 Series
    v_range = ball_v_range(y)
```

```
    # 返回 True or False，True 表示会被秒杀，False 表示不会
    op_v = p_v.where(count % 2 == 0, -p_v) # op_v = p_v if (count % 2 ==
0) else (- p_v)
    # 经测试，逻辑连接符可以用&|-表示与或非，但是似乎不可以用
and,or,not。注意优先级不同，必须加括号
    return -(((op_v >= v_range[3]) & (op_v <= v_range[2])) | ((op_v >=
v_range[1]) & (op_v <= v_range[0])))
```

get_run_side（获取本次跑位的边界点位置）

```
def get_run_side(tb:TableData):
    """
    获取本次跑位的边界点位置
    :param tb: 桌面数据
    :return: (跑位边界值元组，当前状态)。1 表示 55500 以上，无死角；2
表示 27700-55500，回中无死角；3 表示以下，回中也有死角。
    """
    # TODO 每局开始时调用，获取战场进度
    bat_s_max = int((tb.side['life'] / RACKET_LIFE) * BALL_V[0]) * 1800
    if bat_s_max > 1000000:
        return ((0, 1000000), 1)
    elif bat_s_max > 500000:
        # 不明原因出现安全区失效
        return ((1000000 - bat_s_max - 1, bat_s_max + 1), 2)
    else:
        return ((bat_s_max, 1000000 - bat_s_max), 3)
```

clear_ds（用于在回合开始时对 ds 进行初始化）、initial_ds（初始化 ds）

```
def clear_ds(ds:dict, *args):
    """
    用于在回合开始时对 ds 进行初始化
    :param ds: 传入 ds 字典
    :param args: 需要清除的 Key 值。如果为空，则默认为回到空字典。
    :return:
    """
    if args:
        for key in args:
            if key in ds:
                ds.pop(key)
    else:
        ds.clear()
```

```
def initial_ds(ds:dict):
    """
    初始化 ds 的
    :param ds:
    :return:
    """
    clear_ds(ds)
    # catch 保存对面落点信息
    ds['catch'] = []
    return
```

save_catch_pos（保存对对手的接球点纵坐标）

```
def save_catch_pos(tb:TableData, ds:dict):
    """
    保存对对手的接球点纵坐标
    :param ds: 要保存的字典
    :param op:
    :return:
    """
    ds['catch'].append((tb.ball['position'].y, tb.side['position'].y))
    if len(ds['catch']) > 10:
        # 最大队列长度为 10
        ds['catch'].pop(0)
    return
```

get_run_pos（得到跑位点。使用加权平均值）

```
def get_run_pos(tb:TableData, ds:dict):
    """
    得到跑位点。使用加权平均值，
    :param tb:
    :param ds:
    :return:
    """
    catch_list = [x_square(data[0]-500000) for data in ds['catch']]
    # 获取我们认为的最佳落点
    pos = 500000 + anti_x_square(np.average(catch_list, weights =
[x**1.5 for x in range(1,11)][0:len(catch_list)]))
    return (pos + tb.side['position'].y) // 2
```

一些基本的函数功能

```
def x_square(x):
    return x * abs(x)

def anti_x_square(x):
    return abs(x)**0.5 if x > 0 else -abs(x)**0.5
```

2.3 程序限制

最主要是问题是，由于参数调试不足，在比赛前算法算出来的最优解基本都是打单边，这样实际效果和暴力打角差别不大，优势不明显，反而造成了计算资源和时间的浪费。

其次，当此算法在东边的时候，经常发生吃不到道具的情况，这一点我们并没有找到问题原因，也没有提出解决方案，这也是我们算法和代码下一步优化的主要部分之一。

还有，我们尚未实现控制跑位，让体力值在 27500 以上不会出现自杀式跑位情况。这个是可以解决的，但当时时间有限，所以未能实现，我们只有一个简单的保护函数，但还未经过调试。

最后，由于时间限制，我们没有加入随机因素，这使得比赛过程中很可能被猜到策略而被针对。而在比赛中，我们可以看出，已经有几个组的算法具有了学习能力，因此添加随机因素是必要的。

3 实验结果

3.1 测试数据

实验环境说明：

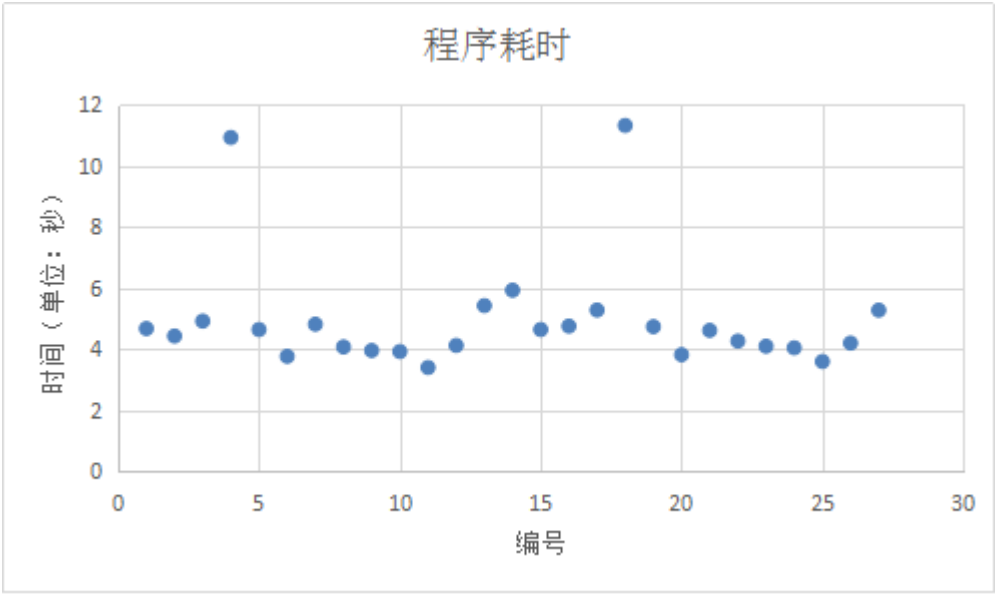
- 硬件配置：intel(R) Core(TM) i5-6200U CPU @2.30GHz 2.40GHz
  内存(RAM) 4.00GB
- 操作系统：Win10 64 位操作系统
- Python 版本：Python 3.6.1
- 测试方法：添加 docest 的测试样例+直接对战

doctest 模块会搜索那些看起来像交互式会话的 Python 代码片段，然后尝试执行并验证结果。我们为大部分的函数都添加了 docest 的测试样例（如图），方便检查函数的错误，在函数修改后也能很方便地调整相应参数。注释掉的部分就是 doctest 的测试部分，如果 doctest 通过，不会有任何输出，也可以加-v 参数来查看测试细节，这里就函数已经测试过了，所以运行不会有输出

```
'''
# 检查测试样例时使用
if __name__ == '__main__':
    import doctest
    doctest.testmod(verbose=False)
'''
```

```
def ball_v_range(y:int or pd.Series) -> tuple:
    """
    根据我方出射点坐标，算出y轴可取速度的边界值
    :param STEP: 1800 tick
    :param b_d[2]: b_d[2] = tb.ball['position'].y,接球时球的纵坐标
    :return: 可取速度的边界
    >>> ball_v_range(0)
    (1666, 556, -1, -1111)
    >>> ball_v_range(1000000)
    (1111, 1, -556, -1666)
    >>> ball_v_range(500000)
    (1388, 278, -278, -1388)
    >>> ball_v_range(pd.Series([0,1000000]))
    (0    1666
    1    1111
    dtype: int64, 0    556
    1      1
    dtype: int64, 0     -1
    1   -556
    dtype: int64, 0   -1111
    1   -1666
    dtype: int64)
    """
    # 已完成测试
    # v0,v1,v2,v3是速度的范围边界:可取[v3,v2]∪[v1,v0]
    v0 = (3 * Height - y) // STEP
```

对战测试，函数运行平均时间 4.9s 左右。



第一次热身赛结果：胜率为‘赛 150，胜 68’，这里用的代码是 T_Siderplus。平均局数：（依据是记录 shelve 的 log 条目数量除以 2，统计目的是控制对战总时间）

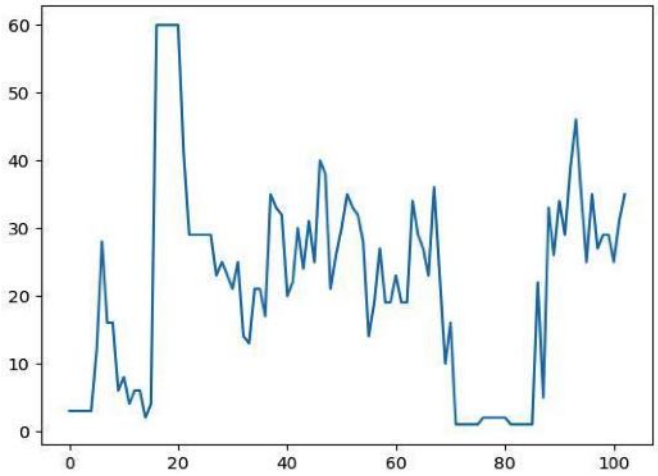


图 1：对战局数（横坐标：局数；纵坐标：回合数）

3.2 结果分析

热身赛：

第一次热身赛采取的代码是“T_Siderplus”，主要运用的算法是启发式算法，也就是根据预先的经验，比如打边，跑位到中间，以获得最小消耗。这个算法基本思路也就是交替的将球打到对方的上下两边，自己跑位到中间的位置，然后受制于当时编写代码时间，所以没有实现道具系统。

热身赛后，总结来看，胜率是“赛 150，胜 68”，主要来看我们的算法还是输的比例比较大，因此我们就各种输的原因做了分析：

1、 miss_ball

这也是我们主要败的原因，主要就是由于体能中降到 55%以下后，在桌面上就会存在死角，而我们的程序跑位是每次打完球就跑到中间，如果对方始终采取打单边的策略，那么我方生命值消耗就会比较大，而且我们的程序没有道具获取和使用，所以也就会造成生命值消耗的差距。

而在实际中，战胜我们的程序中也大多是由于以上两个原因，当然是以综合上面两种原因居多。

2、 invalid_bounce

我们的程序在控制速度方面，会有一个叫做 ball_v_range_up 的函数计算出可取速度的边界值，而出现“invalid_bounce”是我们预料之外的现象，后来我们考虑原因是对手使用了“旋转球”的道具，是我方加速度减半，而我们没有对此进行处理，所以可能导致我们的速度超出可取边界，因此出现了“invalid_bounce”。

针对具体对手，我们也进行了分析，大概分了三类，第一类是杀伤力相对较弱的，基本就是指哪些输给我们的，第二类是采取了打角的策略，但是比我们多了道具使用和获取，第

三类是存在一定学习能力的程序，而且对道具的使用和获取也很理性，比如我们认为最强的算法 T_silly，就已经可以根据我方的跑位一定程度上确定球的落点和道具的使用，以实现对我方最大的伤害。

于是我们大概做了下一步的工作安排：

1、针对对方使用“旋转球”的情况做出反应。

2、构造道具系统，包括道具的获取和使用。

3、不再采取简单打边，考虑通过计算估值函数的方式来优化算法。

根据上述的工作安排我们开发了 2.0 版的程序——“T_Series”，针对我们发现的问题做了集中的处理。

组内测试：

组内测试主要指的就是让“T_Series”和“T_Siderplus”对战和“T_Series”和“T_Series”自身对战，以实现对“T_Series”中估值函数各个系数值的确定和隐性“bug”的修改。

组内测试的开始由于参数值是随意设定的，造成的结果是“T_Series”对“T_Siderplus”的胜率并不理想（这里“T_Siderplus”已经增加了对旋转球的反应和道具系统），而且发现虽然我们是采取估值函数的方法来确定球的落点，球的落点还是固定在打一边的情况，而且道具获取效率也并不理想。

虽然 T_QUEBEC（也就是 T_Series）对于道具获取和使用要优于 T_Siderplus，而且使用了搜索，但是由于估值函数中存在的一些问题，T_QUEBEC 的体能消耗要远大于 T_Siderplus，因此会输掉比赛。

而这样的基于搜索估值的时间控制是一个很大的问题，如果考虑每一局最多打到 60 回合（基于第一次热身赛的分析结果），我们需要控制每次搜索时间小于 1 秒，虽然我方最终选择使用 pandas 库来实现运行速度的提高，但是最终也只能实现最多两层的搜索。另外，我们的 T_Siderplus 基于启发式的打角和跑位本身就有一定数学计算的合理性，所以，其中最终我们也没有实现“T_Series”对“T_Siderplus”较为理想的胜率。

结合这些，我们又提出了进一步的改进：

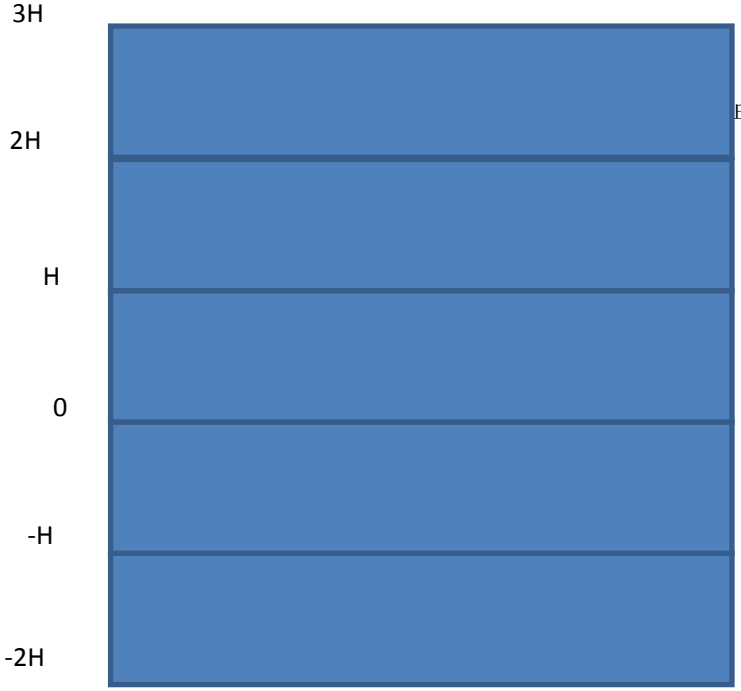
1.在体力值较少时，应该降低攻击性道具的估值（因为可能在使用之前就把自己搞死了）。

2.出现了试图吃道具但是吃不到道具的情况（道具路径获取不准确），连续几次尝试获取道具，但是都是擦过去的。

3.跑位不要用搜索，使用人工策略。

4.体力值下降到节点时要考虑选择性覆盖一边。

当然，以上几点最后由于时间问题没有完全实现，而且由于我们使用了 pandas 工具，本身代码维护成本很高，很容易出现各种 bug 和问题。





而对比“T_Series”和“T_Series”对战相对来说，持续时间会更长，基本可以将战局持续的体力值小于 40%的情况（如图）。

原因是关于跑位，55000 以下，其实还是有安全区的，我们的程序尽力保证在安全区内，至少能保证每一局最好能够持续到 25000 左右。

最后，由于我们程序代码量很大（695 行），我们难以完全排除 bug 的存在，我们觉得我们最好把最终代码用一个 try except 包起来，一旦出现 bug，就可以返回一个能快速算出来的，我们觉得没有任何问题的结果。

我们在组内测试的过程中发现的问题很多，还包括很多由于初期写代码对 pandas 使用不够熟悉造成的错误，对此我们在这一阶段也做了相应的修改调试，以实现更好的运行效果。

比赛：

由于我们对代码调试的复杂度没有足够的估计，没有预留足够的时间进行代码 bug 的修改，以至于我们在提交代码之前才发现代码有一个很严重的 bug，关于这个 bug 的存在原因和进一步修改的办法，在‘不足与反思’这部分里有具体分析，这里就不再展开。最后我们提交的代码是我们的‘陪练算法’——“T_Siderplus”，所以最后比赛结果确实不太令人满意。胜率是 38/150，没能小组出线，而由于我们提交的是很简单的“T_Siderplus”，因此对于比赛结果的分析意义并不大，所以这里就不再赘述。

4 实习过程总结

4.1 分工与合作

分工：

算法讨论：杨帆、张昊、姜河之是

代码实现：金恬、杨帆、张昊

代码测试：叶勃

实习报告：胡若冰、曹寒冰

合作方式：

线上讨论：微信 和 Coding（Coding 是一个面向开发者的云端开发平台，目前提供代码托管，运行空间，质量控制，项目管理等功能）结合

线下组会：如图所示

【第一次讨论】【2017.5.16.二教三层讨论区】【胡若冰拍摄】



【算法组讨论】【2016.5.24.中午 12 点二教三层讨论区】【胡若冰拍摄】



【全员参与准备热身赛】【2016.5.27 晚上 6.30~8.00】【一教】



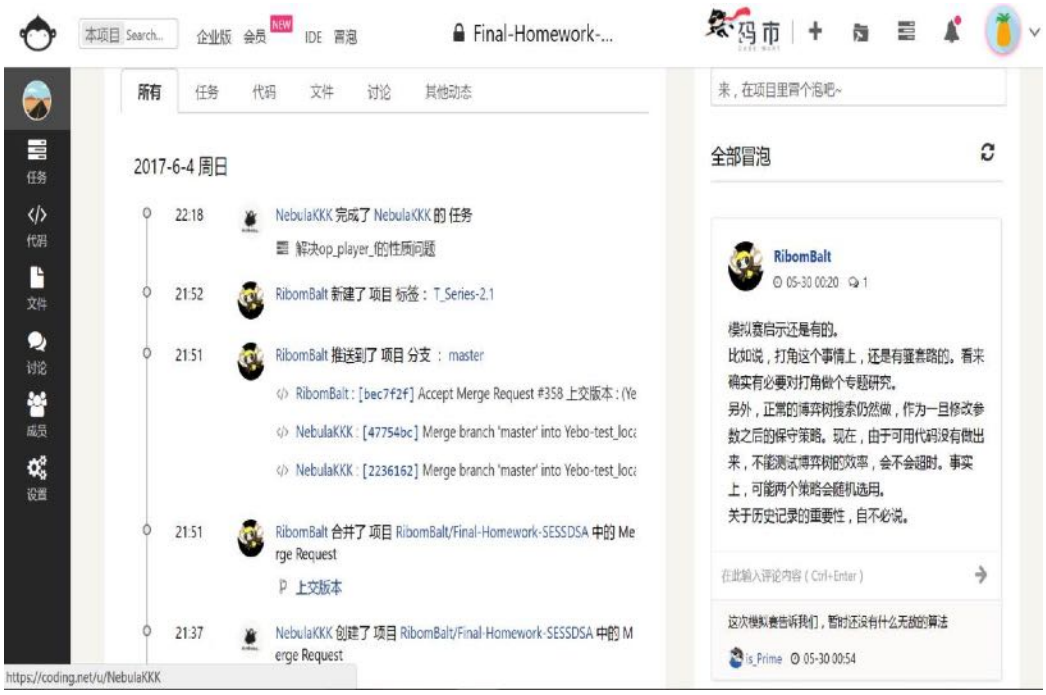



【赛前最后一次讨论】【2017.6.2.下午 3.00~5.00】【一教 112】【曹寒冰拍摄】






【Coding 讨论】截图：



4.2 经验与教训

作为数算大作业的首次合作，面对一个并不熟悉的“乒乓漂移对战”项目，在组织讨论方面，组长杨帆提出全组主要通过 Coding 软件沟通大作业上的问题，以解决“微信群找不到聊天记录”等问题。这一点也是我们组相较于与其他组的独特之处。在组长的带领下，全组员会在讨论版上接收到项目的最新进度，以免有组员“落下”，同时还能在创建的话题下评论自己的想法，对于效果而言，我们组觉得此软件在大作业进展和记录方面做出了重大贡献。

说到大作业，不得不提到我们的心路历程，可谓经历了“一波三折”。从开始算法提议时的“自信”到算法选择上的“纠结”到 debug 时近乎绝望的“失落和焦虑”再到最后的“安然”接受挑战。我们总结，主要是由于时间上的限制导致最后的结果并不是很令人满意，所以我们此次报告会着重对我们在大作业实施进程中出现的众多问题进行总结，并对之提出改进方案。

首先是“战略上的失策”。按照我们搜索的结果，由于参数的设置，最佳路径基本在“打边”而没有太大策略性。而由于这符合直觉的判断，导致“暴力轮流打边”成为很多组的主

要战略。但是根据估值函数来看，“只打一边”是较好的策略。当时我们组向老师提出过修改参数使“打边”不那么有利的方式以增强游戏的难度和可玩性，但是最终没有实现，这也使得搜索算法优势减弱。一般的搜索算法很难像非搜索算法一样灵活改变策略，除非加入学习、随机并且针对可能出现的非搜索算法写一些搜索体系外的应对策略。（由于时间精力和能力有限，我们没有按照这一思路实施策略）因此，搜索算法陷入尴尬的境地——发现大家都不按套路出牌。而事实上，从赢得比赛的角度，考虑到比赛本身对搜索的不利以及下文中会提及的代码实现以及维护的巨大困难，我们最佳选择可能是放弃搜索算法。但是当时我们不愿意放弃这一个非常有意思的算法（因为相较“暴力打边”等人为构思代码简单实现的策略，这是一个更加依赖于计算机“智力”而非人智力的算法），同时也寄希望于后期能够通过参数调整、加入随机来提高战斗力。但是我们低估了代码实现难度以及维护成本（从上文作为 1.0 正式版本的代码还存在各式 bug 中可见一斑），以至于搜索算法在完善核心参数调整上基本没有花多少时间。甚至由于现在已知和直到现在可能都还没发现的 bug，程序运行结果很差。

其次是组织和分工问题。我们前期（第一周）浪费了大量的时间，在讨论一些和本次没有什么关系的算法上。后期（第二周）我们定下了基本策略，并且使用了 pandas 搜索，而这个举措单从小组大作业的角度来说，是非常不明智的，使用一个我们都不熟悉的第三方库的结果是增加维护成本，并且我们在产出成果之前无法进行验证，也就不知道这个策略是否有效。事实上，我们不应该把这个大作业完全当成一次编程练习和学习，而应该融入我们的智慧;而我们实际上把完成代码放在了前面,默认了我们自己闭门造车出来的东西是有效的，而忽视了整个摸索问题本质的过程。这个过程，其实应该是老师的初衷，而我们的做法是背离这个初衷的。虽然说，赢得比赛本身没有太大的意义，作为一门课的课内比赛，过程中学到东西可能是更有意义的。但是既然是比赛，就应该多一些关注事件本身的意识，搞明白事件的要求并做好，即便可能方法不是那么“漂亮有逼格”。另外，我们死板的分工实际上还是带来了很多问题，有些同学负担过重。另外，人数越多，的确会带来越低的效率。如果把我们的团队拆成两组，相对独立开发，进行算法竞赛，这样的管理思路会更加高效。

然后是代码实现的过程中，由于能力有限、算法繁复不定、以及第三方库选择使用上这三个主要问题导致了超长代码成型时间和大量的 bug，大量的 bug 也让测试组花费了较久的时间去调试。进而不可避免的出现了超时问题。

最后一点也是最重要的是效率上的问题，期末忙、压力大也算是效率较低的主要原因。但是假如最后一天下午可以凑时间一起讨论写代码并持续三天，可能大作业完成的会更好。

4.3 建议与设想

给学弟学妹的一点建议：

组队方面，首先要充分了解自己的水平，也就是知道自己所擅长的，然后再是合理分组（这个非常重要！）。然后是不要误解“写报告会比编程、测试代码轻松”。其实，算法、写代码、测试代码都是不分家的。如果擅长写代码，却不会使用算法也不会顺利完成任务；如果只思考算法，却不考虑代码实现时的困难，就会造成像我们组后续工作繁复的后果。另外，良好的代码风格对多人合作的任务非常的有利，加上了文档之后，可以节省很多人费心理解同一段代码的时间，这样就能把精力更多地放在算法、策略本身上。

最后我想说，写报告并不像我们原先想的那么简单，反而我认为写报告是最难的。首先，时间上，和期末季紧张的复习撞车了；其次，你要清楚地概述大作业讨论过程的所有细节，也就是说所有的讨论你都要参与并紧跟进度，同时还要记录过程以便最后总结；最后建议大家要积极参与到活动中不要有任何“搭便车”的心态，因为这也算是大学时光为数不多有意

义的课程合作项目，就像“挑战杯”一样值得大家去体验过程，结果不重要。

另外，关于以后的大作业，我还有一点建议：我们的大作业在最后两周开始筹备，实际上，通选课考试也在最后一周，这样几个事情在一起就会非常忙，这可能会比较影响大作业的完成质量，希望以后的大作业的时间周期可以提前一周。另外希望大作业的内容及早讨论，因为时间比较紧的情况可能会出现某些设定失衡（比如一开始打边过强的情况，以及旋转球的秒杀 bug），当然，判断平衡性必须要做出来测试代码才能知道，所以这个问题也不太好提前解决吧，总之希望老师可以想想办法。

5 致谢

本次作业在组长杨帆的带领下，全员参与，使用 Coding 作为主要线上沟通平台，利用老师和助教讲解的知识和网络查找的资料完成。这次大作业让我们学到了很多，主要是对本学期学习的数据结构知识进行了综合利用，并用 python 语言编写了更加适用于本算法的函数，对本课知识的理解更进一步。

首先要感谢陈斌老师和助教的漂移乒乓创意、对战平台编写成员的工作，让我们有机会自己从零起步实现一个完整算法编写过程。同时他们的随时答疑和全程关注对于本组最终算法的产生与提高有着至关重要的作用，在此深表感谢。除此之外还要感谢 Quebec 全体成员的共同努力，金恬、杨帆、张昊、叶勃同学负责主要基础代码的撰写，胡若冰、曹寒冰主要负责写报告，姜河之是、张昊、杨帆主要对于算法思想提出一些建议，所有同学参与全局、控制整个实习作业的进度，每个人都从这次作业实习中学到了很多东西。

再次感谢陈斌老师和助教的指导和全体组员的勠力同心。

6 参考文献

[1]个人博客 极小极大搜索方法、负值最大算法和 Alpha-Beta 搜索方法 from http://www.cnblogs.com/pangxiaodong/archive/2011/05/26/2058864.html

[2]CSDN 博客 五子棋 AI 算法第三篇-Alpha Beta 剪枝 from http://blog.csdn.net/lihongxun945/article/details/50668253

[3]pandas 文档 pandas 0.20.2 documentation from http://pandas.pydata.org/pandas-docs/stable/generated/pandas.Series.html

附录：Quebec 组编码规范

杨帆

这一份是做树莓派的时候我们组的管理规范，我看了看主体不需要改动。重点就是合作代码，注释文档不要少，命名规范。doctest 要求，这次数值运算函数必须加测试样例，数据结构类不做要求。每个函数功能明确，去掉注释不超过 50 行（最多最多……也别超过 100 行）拒绝所有功能写到一个函数里……debug 太复杂太难。

0 概述

对于软件工程来说，编码规范就是生命线。即使是三个人的小组，也需要进行一定的规范，这个过程并不会增加过多额外的任务。下面的内容主要根据我对 Python 和软件工程的理解来写的，如有不恰当之处欢迎提出批评建议！

适用 Python 版本 3.5 以上

1 关于 IDE，与一些相关技巧

建议使用 PyCharm，除非你可以不借助 PyCharm 而完全遵守本协议。PyCharm 原生支持 PEP-8 协议，这是一个写 Python 约定俗称的协议，可以让你的代码更易读。所有违反 PEP-8 协议的内容都会标黄色提示！请务必修改代码直到大部分的黄色标记全部消失！特别是提示 Reformat Files 的地方！（快捷键：Alt+Enter）（比如：a=b -> a = b）

（例外：

1.提示“all tri-quotos should be used for docstring”时可以不用理会，具体见“关于文档”

2.有些黄色标记是无意义的，比如将==两侧的元素换换位置，这种可以不用理会

）

注意所有的黄色标记！

如果你想到要做一件事，但是现在有更紧急的事情，请在需要以后处理的地方加一行 TODO 注释。格式：

```
# TODO what you need to do
```

这样的注释在 PyCharm 中会高亮显示，你不会忘记它！

每行不超过 3/4 屏。如果代码太长，可以在适当地方断开，可能需要加上\

关于缩进，这个看黄色提示就可以了

2 关于文档与注释：（重点）

什么是文档：Python 中，在一个模块（一个文件名是合法标识符的.py 文件）、类或函数的开头用一个三行字符串表示文档。

文档的内容可以在 help 中显示出来。

例：

```
def func(a):
    '''
    xxxx
    '''
    # your codes
```

文档要求：
模块：写出模块的功能，有哪些类和函数
类：写出关于什么的类，有哪些成员变量和方法。
函数：函数的功能，参数内容，返回值是什么，什么情况下会抛出异常（Exception）

在我们的小型工程里，对模块和类暂时不做硬性要求，建议对复杂而有逻辑的类或模块添加文档。

所有的函数，必须有完整的文档！！！
完整文档的效果应该达到：别人看到这个函数文档，可以完全不看函数中的任何代码，就可以完全知道这个函数应该怎么用。所有错误要么在函数内部，要么在函数调用的过程中。

完整的函数文档应包含以下内容：
1.函数的功能。尽量用中文描述清楚。建议第一行一句话简洁概括，之后几行用几句话说清第一句没有解释清楚的部分。注意抛出异常的情况，至少占一行
2.函数参数列表。格式：:params name: xxx，后面是这个参数的类型与具体含义。参数的命名规范见“关于标识符的命名、变量”。PyCharm 中会自动生成这一部分的前缀。
3.函数返回值。格式：:return: xxx，后面是返回值的格式与具体含义。Pycharm 会自动生成。
4.doctest 测试样例*：虽然可以没有，但是建议添加。有点类似 IDLE 的操作，用一个>>> 作为开头（注意空格），然后后面是要测试的语句。下一行紧跟输出的结果。

```
def sum(a,b):
    '''
    返回 a 和 b 两个数的和。
    如果 a、b 有字符串，抛出 TypeError
    :params a: 第一项
    :params b: 第二项
    :return: 两个数的和
    >>> sum(1+2)
    3
    '''
```

*关于 doctest：import doctest 模块之后，在模块末尾输入 doctest.testmod()，可以对模块所有测试样例进行测试。如果全部正确，不会有输出。如果有错，则会仔细地输出错误提示。

保持代码和文档同步！也就是说，如果你 debug 时改了代码，功能或者参数变了，一定要同时改掉文档！

关于单行注释：处处有注释，# 开头一句话（建议#后紧跟空格）

不做硬性规定，只要保证你的代码出现 bug 后你根据自己写的#能快速 debug 就可以。

如果非要有个建议：每个 if-else 分支和 for/while 循环前加注释，长段顺序执行在每个小功能开始前加注释，不好分小功能（常常逻辑复杂）建议 2-3 行加一个注释

3 关于 debug：

PyCharm 中有 debug 单步执行功能，原则上自己函数出现的问题都可以通过单步执行发现。使用方法：

a.点击小虫子图标

b.程序会执行到出现未处理的异常或断点处。

c.下方六个并排按钮，第一个是执行下一步（平级）。第二个是进入某个函数的内部流程，第三个是运行到自己写的代码处，第四个我暂时不知道，第五个是执行到这个函数返回值后的一步，第六个是执行到你光标所在的那一句。视情况进行合理使用。

d.如果感觉在某个地方需要重点关注，可以在行数右面点击，出现红色的断点。

e.如果需要看某个时刻的变量情况，下面会有当前环境每个变量的值（str 类型），对于复杂类型，小三角可以看到成员变量。建议给自己的类都加上__str__方法，方便 debug

注意事项：

debug 功能和 doctest 不兼容。如果要断点测试，请把测试样例注释掉。这也是我没有强制要求 doctest 样例的原因之一。

如果要查看某一句的执行次数，我暂时没想到好办法。可以增加临时计数变量，或者 print 的方法

模块里可以增加主函数入口以供测试。具体而言：

```
if __name__=='__main__':
    xxxx # your codes
```

这样，如果执行这个模块，就会执行后面代码。而如果把这个模块 import 时，不会执行后面的代码。

同样的原理，不建议在模块中加入“游离”于函数和类之外的代码（除非用于初始化）

4 关于标识符的命名、变量：（重点）

标识符，就是一个变量的名字。模块名类名函数名，函数的参数名、函数内部临时变量的名字，都是标识符。标识符由字母数字和_组成，开头不能是数字

模块命名遵守标识符要求。

对类名建议用骆驼式命名，也就是第一个单词小写，后面大写，不加_。myClass。

对函数名建议用小写加_命名。add_last

（其实具体哪种格式命名不做硬性要求，推荐这两种，混用也无所谓，Python 命名本来就挺乱的= =）

对于类名、函数名、函数参数名，硬性要求：名字有意义。比如 add_last 一眼就看出功

能了

建议不用拼音，但是也不要用生僻英文单词。

类中如果定义了一些辅助用的、废弃但是有用的，不需要别人调用的函数，函数名前加一个_（比如_add）

注意：不可以和内置名相同！包括但不限于 str,list,tuple,dict,set,int,float,bool……以及你 import 进来的一切名字！

5 关于开发流程、时间控制

这块并没有想好，就是提个建议吧，可以再商量。

一般开发要先写需求书（我们已经定了，就是电子琴，要什么功能都写好了），然后写文档写架构。

架构就是整个程序的框架，包括写什么类写什么方法，写什么函数。这段时间我们不需要写函数的具体实现（当然比较确定的也可以先写），只需要写出声明和文档。大概花 1/4 时间。

之后就是具体的码代码了，因为有架构所以就是一个拼图的过程，比较简单机械，1/4 时间。

然后就是关键的 debug，必要时修正方案，修改文档，打回返工。我估计要花 1/2 时间。

中间零碎的时间就做点其他的，比如 readme 文档、报告等等。

当然我们不会这么机械，我们可能会把整个大任务打成几个小任务，然后对每个小任务做这个流程。

码了挺多字数，但是请务必看完，对我们的开发流程有益无害。我们的目标是全课程第一名！

【编者按】：Quebec 组组长杨帆，领导树莓派电子琴小组时就以对编码规范的高要求闻名。这一次的大作业该组很好地坚持了这一点。据编者所知，这一要求的确给组员带来了较好的编程习惯。编者认为这一文档对大家有一定参考意义，故收录于此。



NOVEMBER

组长：文家豪

组员：张晓华，王瑞姣，武心依，段会卓，王冠之，唐泱

小组赛战绩：S6

数据结构与算法课程实习作业报告

文家豪* 王瑞姣 武心依 张晓华 段会卓 王冠之 唐泱

【摘要】总体来说，我们的算法较为简洁，属于被动打法，尽量节省体力地跑位并击球，不去刻意追求道具的使用，重点放在如何把球接好，以防守为主；击球时采用打角策略，让对手消耗尽可能多的体力值。采用的算法结构和自定义类有字典、排序、控制流语句（条件 if、for 循环）等。最终结果败多胜少，程序还需要改进。

【关键词】 击球、边角、道具、跑位、条件判断、循环、排序

1 算法思想

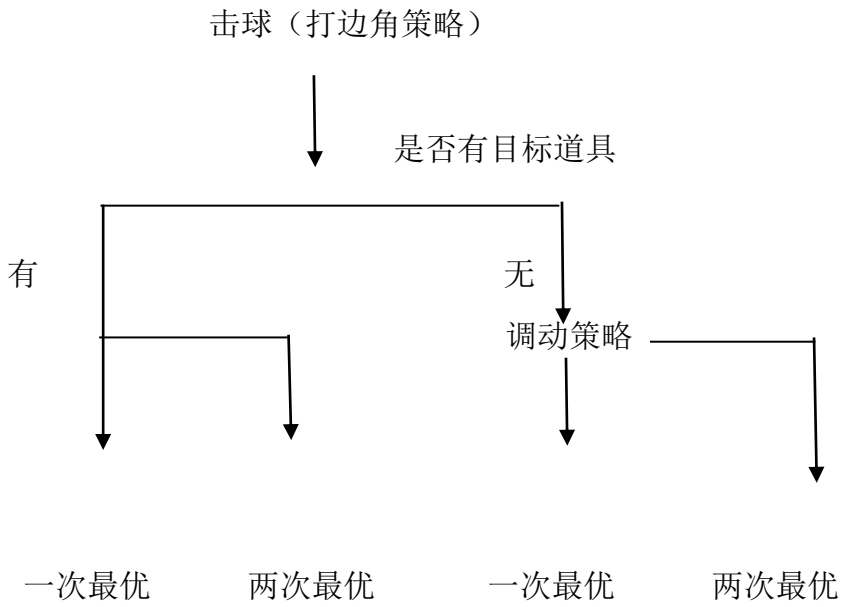
1.1 总体思路

本算法主要由三个函数组成，前两个函数着眼于如何击球，最后的主函数调用了击球函数，并给出了道具的使用策略。

我们的重点放在跑位和击球时，如何用最少的体力完成更多的击球，击球时尽量打边角使对方消耗尽可能多的体力，对道具没有过多的追求，采取的是防守型的打法。道具的获得采用的是直接计算路径的方式，因此是以判断语句为主。受我们思路的限制，我们的算法没有采用比较复杂的结构，框架主要以 if else 条件判断语句为主，同时也有 for 循环语句和排序。

根据这个思路本小组分为两组分别设计了各自的程序，最后通过比较两者，择优对其进行了优化作为小组的最终程序。

1.2 算法流程图



1.3 算法运行时间复杂度分析

因为在发球和跑位函数中主要是判断语句，时间复杂度为 1，算法的运行时间主要由道具函数决定，其大 0 数量级为 T(n)=0(n)。在实际运行中时间的消耗与我们的预期大致相符合。

2 程序代码说明

2.1 数据结构说明

主要数据结构和自定义类：字典的定义及使用、控制流语句（条件 if、for 循环）、排序等

2.2 函数说明

（1）击球函数(def interesting\def batting)：

interesting 用于根据对面发球判断击球的方位，batting 则用于推算以何种方式将球打出去。

总的思路是往边角打，让对方必须以最大的跑位去接球。往上角和往下角的选择就是判断往下角打需要加速耗费的体力减对方往下角跑位耗费的体力和往上角打需要加速耗费的体力减对方往上角跑位需要的体力哪个大，即使对手消耗比我方更多的体力值。这些判断均主要由 if else 语句实现。

（2）主函数（def play）：

play 的接口函数是 interesting 和 batting，包含了跑位和道具判断相关的函数。跑位部分就是通过对方的击球，直接位移接球，没有考虑复杂策略。

2.3 程序限制

本程序在边界问题的处理上存在缺陷，最终导致击壁次数超过两次或者少于一次，因 invalid ball 而被判负。这可能是在函数取整时不恰当而导致的。

3 实验结果

3.1 测试数据

实验环境说明：

a.硬件配置：处理器：Intel(R)Core(TM)i5O-4200U CPU @ 1.60Ghz 230GHz 已安装内存：4.00GB

b.操作系统：Windows10 专业版

c. Python 版本：Python Community Edition 2016.3.2

（1）测试数据及测试程序：

具体的测试数据在压缩文件夹中以单独的压缩包的形式给出。本小组采用的测试方法是将现有的最新版本的程序与历代的老版本进行比拼，将老版本作为测试程序，这些程序也在另外的文件夹中给出。

（2）测试结果：

新版本程序的胜率要高于老版本，其中出现了比较特殊的一个情况是双方对打时有时候会陷入互相打对角的循环，这样两方会待在对角上基本不动，反复击球直到时间用尽为止。

我组参加了第一次热身赛，共 150 场对局，其中 145 负，5 胜。胜和负的原因都全部是 invalid ball。

3.2 结果分析

本算法着重于击球打边角的策略起到了节省体力的效果，但是效果不如预期。算法的运行时间在预期之内，时间花销主要是计算跑位的环节。分析测试结果，出现的主要问题是边角处理问题，导致击壁次数超过两次或者少于一次。分析原因，可能是在取整 int 时，没有处理好。经过进一步讨论，我们设计了一系列解决方案，但最终效果都不理想。

4 实习过程总结

4.1 分工与合作

（1）分工与实行步骤：

唐泱同学主要负责实习报告的撰写，小组其余六人分成两个小组，分别独自编写程序。在程序初步完成之后，两组程序进行比拼，优势较大的一方作为初版。以出版程序为原型，三个人负责写代码，围绕的核心是策略的实现；另外三个人考虑合适的击球策略，包括负责建立自己的历史数据(ds)和利用这个进行分析。

（2）组会记录：

第一次组会：时间：5 月 26 日 地点：二教一层

内容：熟悉乒乓球比赛的基本代码与编程环境，讨论并确定基本策略，设计并规划未来策略的走向，确定初步的分工。

第二次：时间：5 月 27 日 地点：二教一层

内容：全组共同讨论并优化源代码，确保初级策略成功实现。

第三次：时间 6 月 2 日 地点 二教一层

内容：对程序跑位部分的代码进行删减，修改相应的系数，解决程序 invalid 的问题，初步对道具的使用进行构思与编程；对函数的 for 循环结构进行修改，试用二次函数，节省相应时间。

第四次：时间 6 月 4 日 地点：全家便利店

内容：对程序进行最后的优化和修改。

（小组讨论的记录照片保存在独立的文件夹）

4.2 经验与教训

（1）经验：在此次实习作业编写的工作中，本小组的分工较为明确，组员之间的配合较好，大家的积极性也被调动了起来，为程序的编写提供了很多策略。

（2）教训：由于时间紧张，大家准备的不是特别充分，整个程序没有用到复杂的算法结构，在面对多变的实战时，出现了很多缺陷，没有能够及时针对这些缺陷进行优化，导致我们最终没能小组出线。

4.3 建议与设想

（1）对学弟学妹的寄语：实习作业的编写需要平时积累的编程基础，所以在学习过程中必须认真听讲并完成每节课的作业，把基础打好。组队时，要互相帮助，

能者多劳，分工必须要合理，这样才能提高编程的效率。

（2）对实习作业的建议：组队时，希望能够根据平时作业的完成质量划分，避免强者聚集到同一组。希望老师能够花时间更加具体详细地介绍基础设施代码，让更多基础薄弱的同学理解，更好的参与到编程工作中。

（3）设想：可以根据本次实习作业的成功，开发一个电子小游戏。

5 致谢

感谢陈斌老师和助教们对我组编程的帮助，感谢陈老师深夜还在回答我们提出的幼稚的问题，感谢小组所有成员的努力和付出。



KILO

组长：邵俊宁

组员：张祖煜，刘天乐，黄丽潼，张溶倩，李然好

小组赛战绩：N3

数据结构与算法课程实习作业报告

（邵俊宁*，张祖煜，刘天乐，黄丽潼，张溶倩，李然好）

摘要：

我组的算法基于对游戏中获得的历史数据做概率统计的分析，预测对方可能选择的跑位、加速策略并对其进行评估，再结合双方体力值及道具情况，进而决策出能使对方消耗最大而己方消耗最小的跑位、加速的具体方案。我们通过自定义类 class data 来实现对历史数据 ds 概率统计的分析,data 的主要数据形式是一个字典，用来记录球速、落点位置、生命值、跑位位置、跑位距离、跑位方向分别与跑位、落点位置、加速、跑位位置组合时这些组合的不同取值出现的频数。为后续分析提供数据。

关键字：

自定义 data 类，概率统计，预测函数，评估函数，决策跑位函数，决策加速函数，决策道具使用函数。

1 算法思想

1.1 总体思路

本次乒乓球对战的大作业是来源于一项生活中常见的运动，但加入了一些场地、规则上的改变，并交给算法去实现比赛。人在乒乓球比赛中的取胜靠的是眼睛对球的位置、速度、方向的实时捕捉和对对手打法的经验感知，从而在球飞跃球场的过程中迅速做出回应方式的判断，达到的目的是使球到达对方最不易接到的位置。当交给算法去决定时，我们能掌握的

数据是球的实时信息和双方比赛的历史数据，这一点与人相同，而我们要做的是赋予它处理这些信息的能力。另外，大作业中还加入了道具元素和有限的生命值，算法需要综合考虑以上制约，最终得出一个能够使对方生命值消耗最大、我方生命值保存最多的迎球、跑位的具体行动方案。

我们把通过历史记录预测对方行为的部分称为预测部分，把综合考虑道具、生命值因素选择我方行为的部分称为决策部分。

（一）预测部分

我组的预测部分算法基于统计的思想。首先将位置、速度、加速度、生命值、距离等这些参数都离散化，具体来说是将其表示成占总份数的多少份的形式（如，球场的总宽度为1000000pace，将其分成 20 份，则当球落在 y 坐标 500000pace 处时就可以记录为 10）。这样做的目的是便于将球速、位置、生命值等这些参数出现的每一个份数值对应的球落点位置、对方跑位距离、对方跑位位置、加速度等这些参数出现的频数都记录在一个字典中，留待之后的概率分析。准备工作之后，进行对 ds 的处理。ds 是一个包含两个键值对的字典，分别记录我方上上回合看到的桌面信息 lala_tb，和我方上回合的桌面信息 last_tb。用 lala_tb 计算对方决策跑位时看到的状况，再结合我方这回合计算出的对方上上回合跑位，一起用于计算概率。用 last_tb 计算上个回合对方决策加速度时看到的状况，再结合本回合计算出的加速度，一起用于计算概率。

在概率计算时，首先把对方所处的状态的参数列举出来：

1.对方所处的位置

2.己方所处的位置

3.球的速度

4.对方的体力值

5.己方的体力值

6.己方跑位的方向。可以用{−1,1}来表示

7.己方加速度的方向。可以用{−1,1}来表示

不妨把这些参数全部离散化，然后用一个向量(n1,n2,・・・ ,n7) = n 表示然后对方的行为可以用一个三维向量 x = (x1,x2,x3)来表示。也将其离散化。所以对方在状态 n 的情况下采用行为 x 的概率用条件概率 P(x|n)来表示。我们可以采用极大似然估计来估计概率的值，也即是根据以往的频率来估计概率。所以我们可以把每一个点记录下来，然后计算频率来作为概率，估计对方下一次最大概率采取的行动。但是这样的话会有样本点不足的问题，在全空间的分布过于稀疏，从而导致概率估计不准的情况出现。所以我们需要根据概率计算公式对其进行化简。

这里我们采取贝叶斯分类的方法，具体如下。

一般而言，加速度和跑位应该是相互独立的。所以我们可以通过分解成两个问题来解决。即有：

$$P(x1,x2|n) = P(x1|n)P(x2|n)$$

于是空间就减小了一半，准确度提高了不少。此外，特征 n 也有一些参数可能是相互独立的。比如可以考虑：

$$\begin{aligned}&P(xi,n1,n2,n3,n4)\\&= P(n1|n2,n3,n4,xi)P(n2,n3,n4,xi)\\&= P(n1|n2,n3,n4,xi)P(n2|n3,n4,xi)P(n3,n4,xi)\\&= P(n1|n2,n3,n4,xi)P(n2|n3,n4,xi)P(n3|n4,xi)P(n4,xi)\\&= P(n1|n2,n3,n4,xi)P(n2|n3,n4,xi)P(n3|n4,xi)P(n4|xi)P(xi)\end{aligned}$$

显然这样空间大小并没有任何减少，问题依然存在。但我们考虑到参量之间的条件独立性的话，比如假设最简单的，全部相互独立，最后一行就可以写成：

$$P(n1|xi)P(n2|xi)P(n3|xi)P(n4|xi)P(xi)$$

我们只需要把每一项概率分别估计再相乘即可。可以看到在利用独立性简化之后，对于每一个 P，只要不多的样本就可以得到一个比较准确的估计，从而得出我们对整体概率的估计值。

针对当前的状态计算出对方不同跑位位置、球落点位置、跑位距离出现的频率并以（概率，值）的形式返回后，再对此概率分布做一个评估，看以上概率的效度如何，若好，就返回预测的行为；不好，就返回 None。

（二）决策部分

对己方跑位、迎球策略的决定基于以上对对方行为的估计，总体指导方针就是对不同方案计算后，选出（对方生命值损失 - 己方生命值损失）最小的方案。具体又存在如何选用道具和如何选择加速方式的考虑。

选用道具时：

1.如有掉血包、补血包、隐身术，就直接使用。

2.如我方跑位超过瞬移术的参数 250000，或游戏快结束（根据双方剩余血量判断），若有瞬移术就使用。

3.如加速超过 200，或游戏快结束，若有变压器或旋转球就使用。

4.如道具盒满，使用道具的优先级为：变压器 > 旋转球 > 瞬移术。

选择加速方式时，主要在打角、接道具、不加速几种方式之间选择。我们用打分（对方生命值损失 - 己方生命值损失）的方式对各种策略评估，选出最优。这里己方损失是加速的体力损失，对方损失是迎球的移动损失，对于接道具则有额外的分数加成。

1.对 8 种打角的方式一一计算评估，选取赚得最大体力的方法。

2.对接道具加速策略的评估:先初步选出对场上每一种道具既可以接到又不造成 invalid bounce 的加速度，再根据赋予不同道具的评估分数，选出最佳。

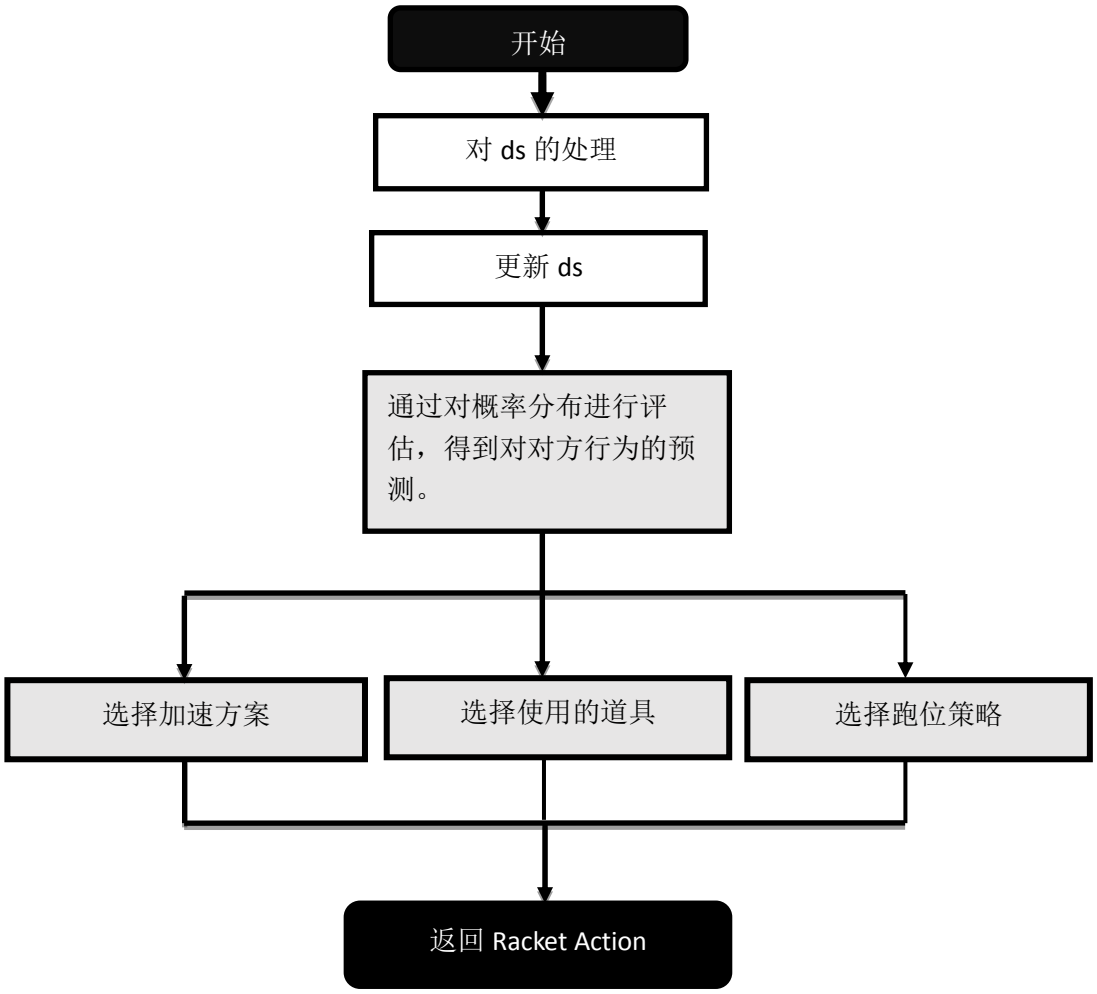
3.不加速时，由于没有己方的损失，一定是接到道具或打到角落的情况下取得极值。

从以上三个中再比较选出分数最高者。

在具体的 play 函数中，除遵循以上原则，我们还加入了随机应对算法，使己方的行为增加一定的随机性。

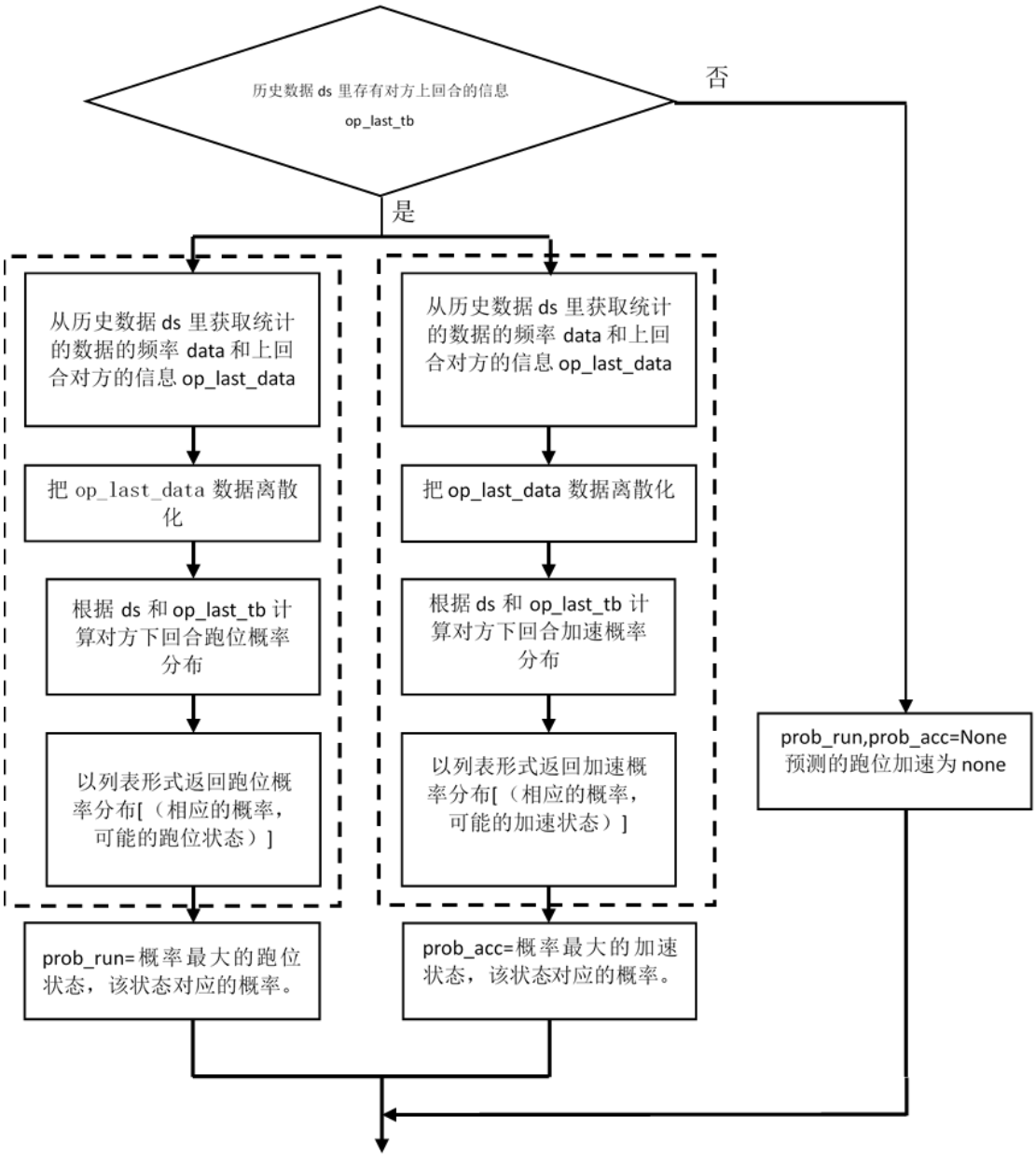
1.2 算法流程图

主流程图：



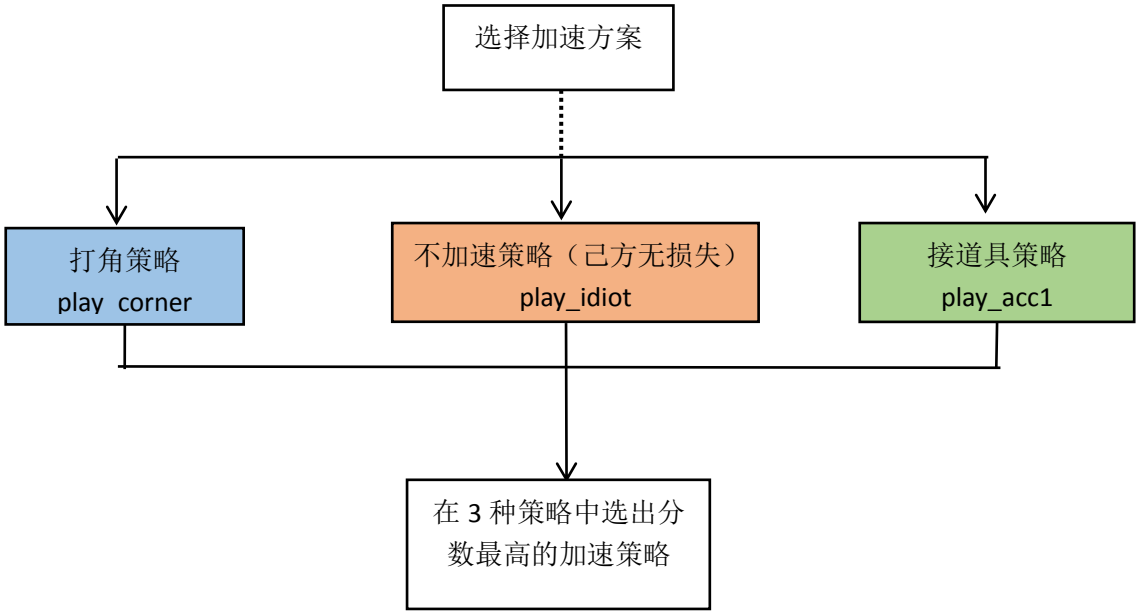
注：其中加灰色的几个步骤分别对应分图 1、分图 2、分图 3、分图 4.

分图 1：通过对概率分布进行评估，得到对对方行为的预测。



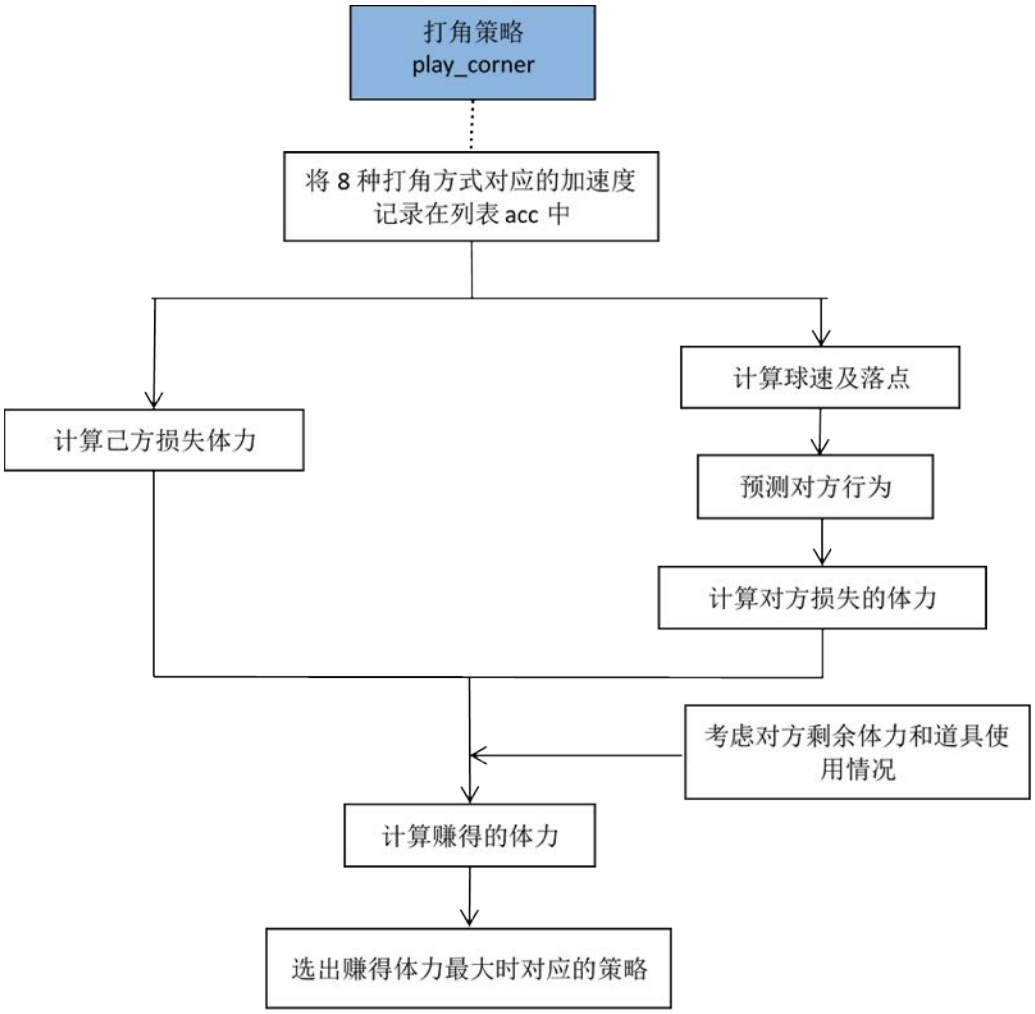
注：
左虚线框内代表 eval_prob_run(tb,ds)函数。
右虚线框内代表 eval_prob_acc(tb,ds)函数。

分图 2：选择加速方案。

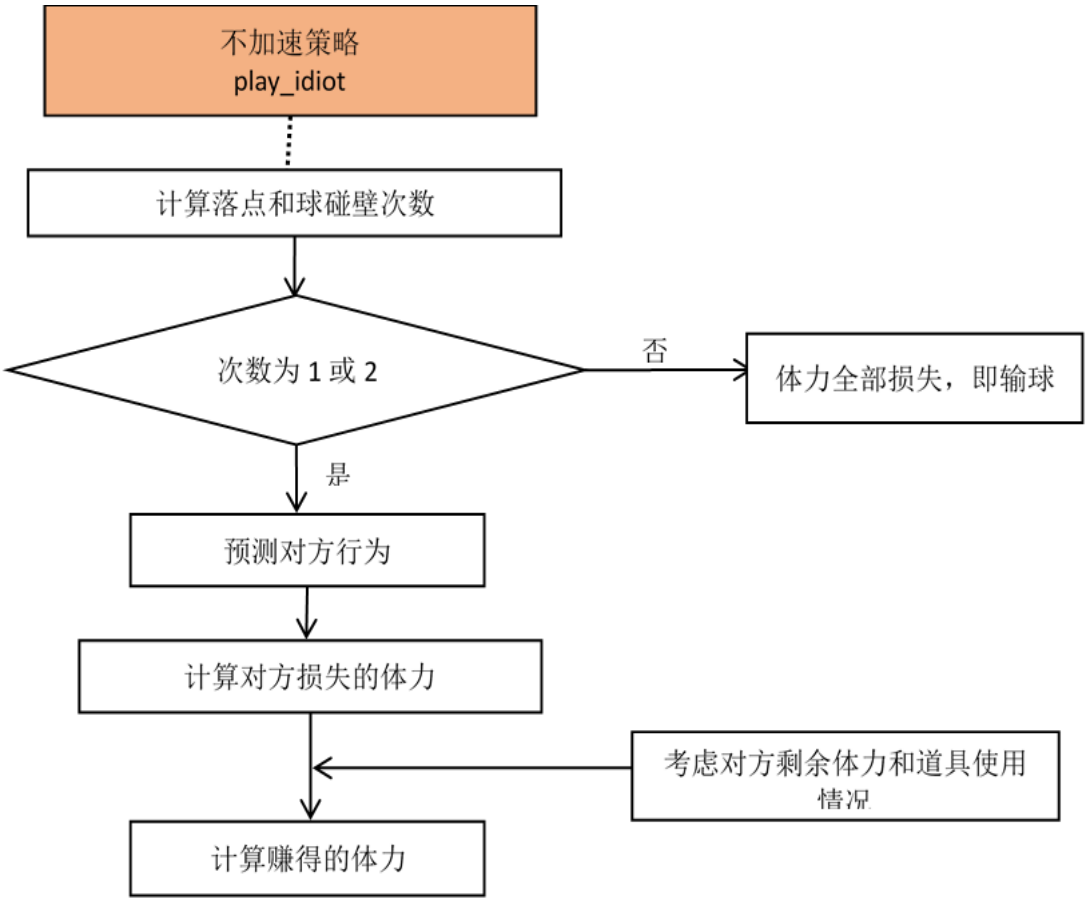


注：其中加颜色部分分别对应以下分图。

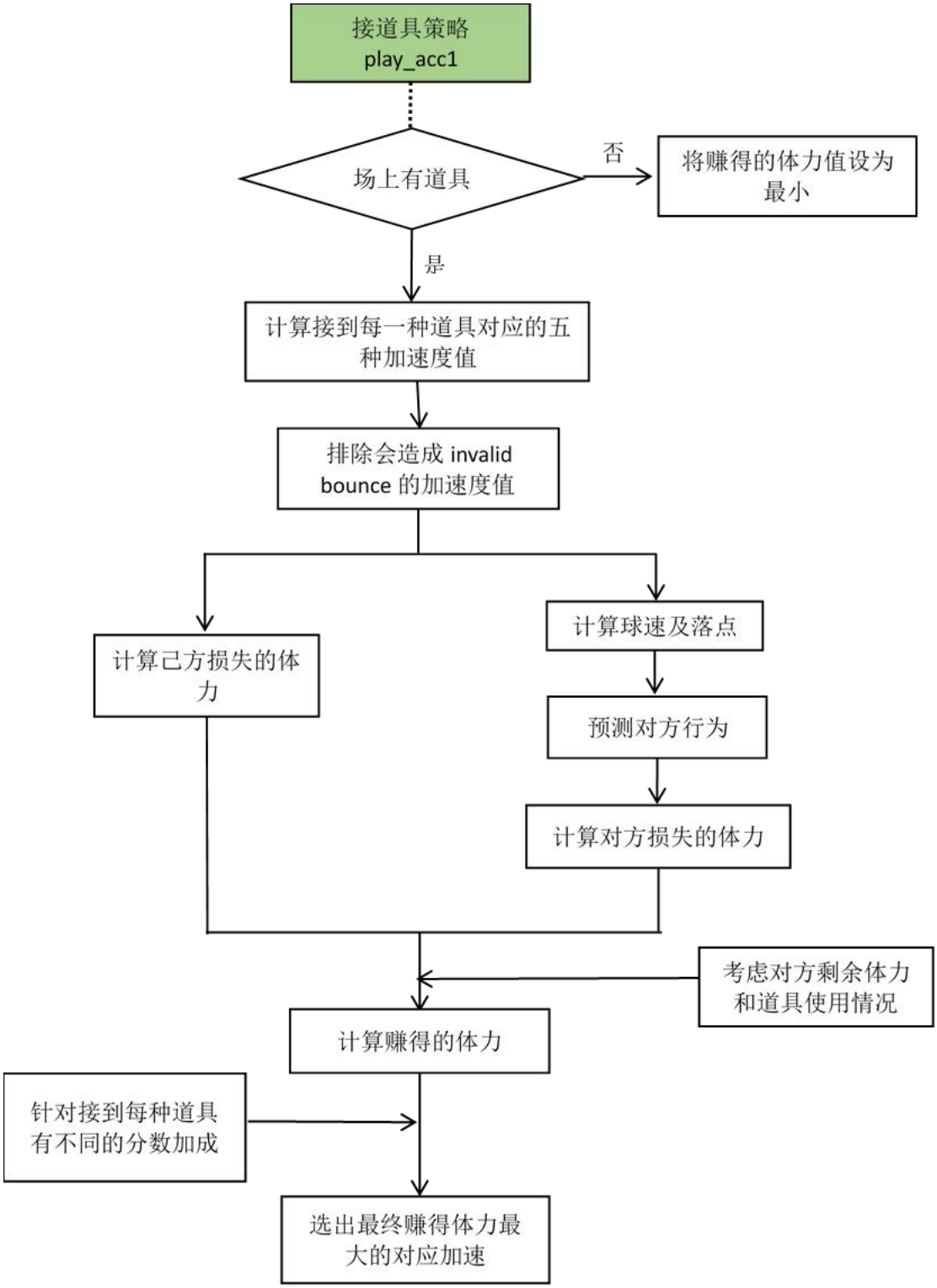
分图 2-1：选择打角策略。



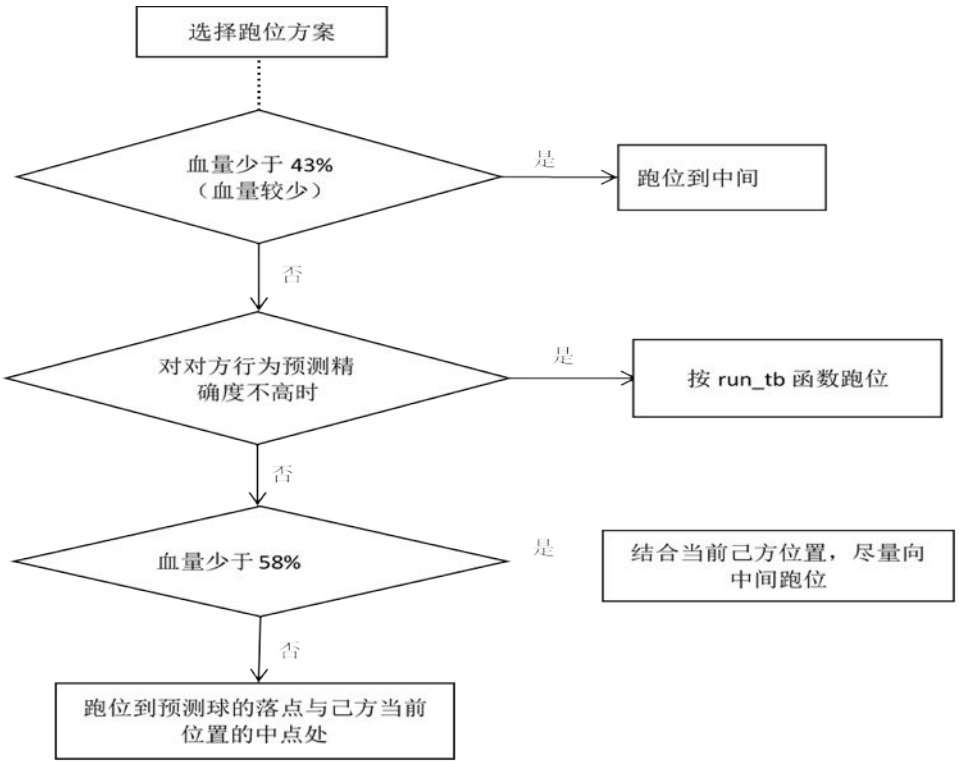
分图 2-2：选择不加速策略。



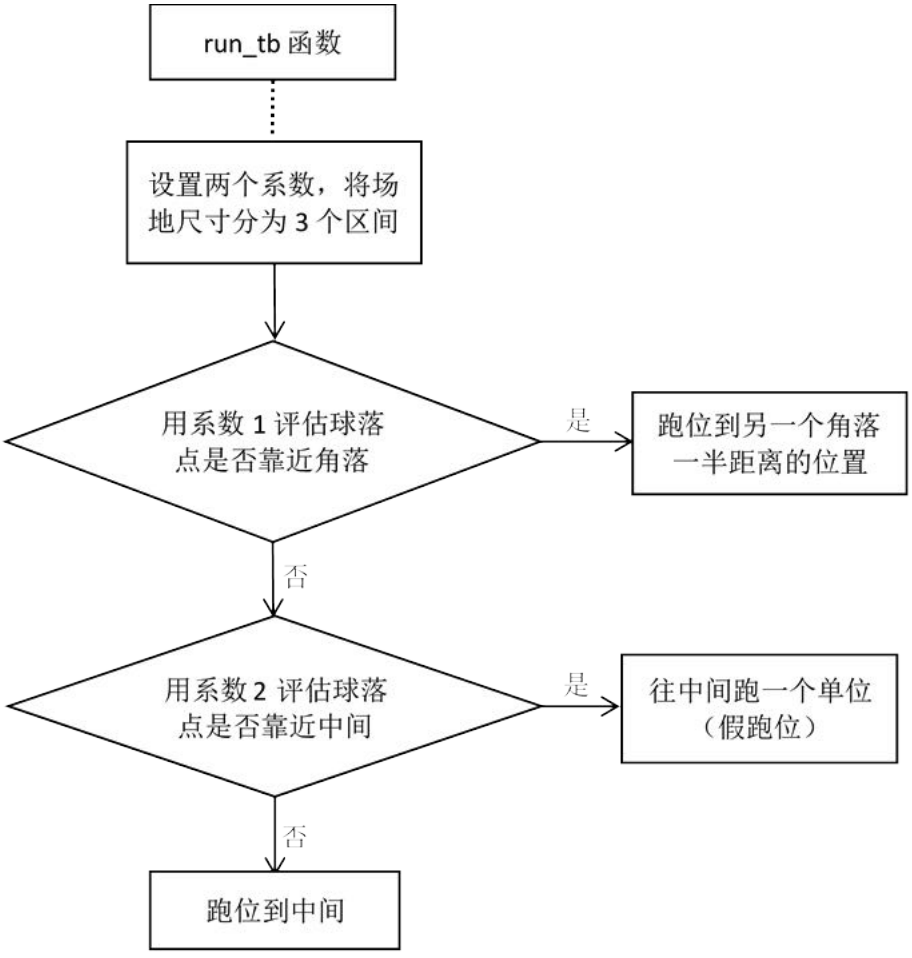
分图 2-3：选择接道具策略。



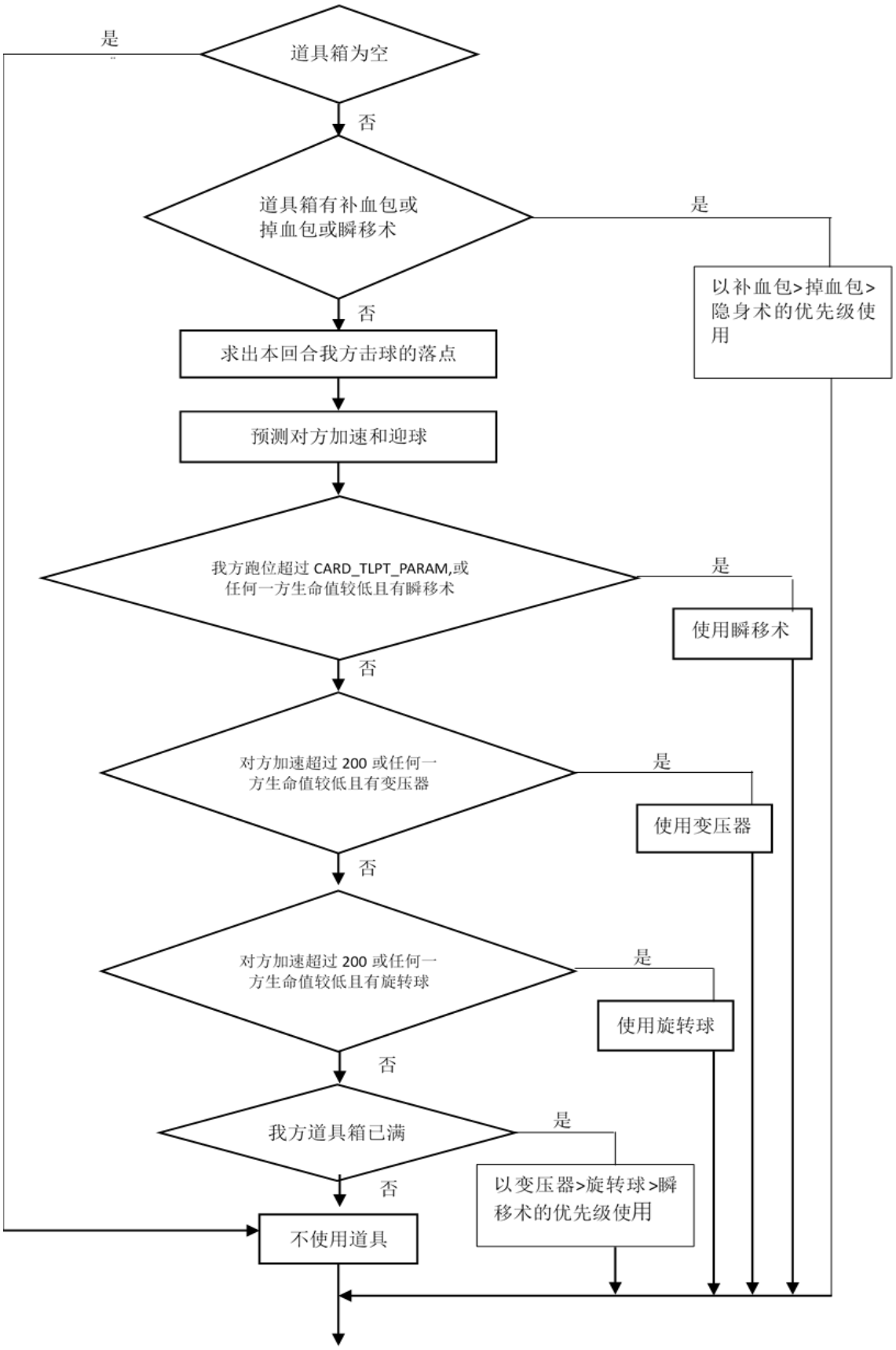
分图 3：选择跑位策略。



其中，run_tb 函数的分图如下。



分图 4：选择本回合使用的道具。



1.3 算法运行时间复杂度分析

考察 play(tb, ds) ，它的内容包括：

1．对 ds 的初始处理：O(1);

2．更新 ds 的函数，update_ds_tb(tb, ds)，包括 update_run,update_placement,update_acc,update_run_placement,这四个函数算法复杂度均为 O(n),所以,更新 ds 的算法复杂度也为 O(n);

3．随机应对学习算法：O(1);

4．调用预测函数和评估函数：均为 O(n);

5．加速决策函数 def_play_acc,包括：

打角方案 play_corner,O(n);

不加速方案,O(1);

接道具方案,O(n^2);

综上，其算法复杂度为 O(n^2);

6．跑位决策：O(1);

7．道具选择函数：O(n);

综上,play 函数的算法复杂度为 O(n^2).

2 程序代码说明

2.1 数据结构说明

class data 是一个字典，用来记录一系列频数，是 ds 的一部分。

```
class Data:
    def __init__(self):
        # 生成字典记录频数
        self.data = {
                    ('ballv', 'run'): {(m, n):0 for m in range(-VD//2, VD//2 + 1, 1) for n in range(-XD//2, XD//2 +1, 1)},
                    ('pos', 'run'): {(m, n):0 for m in range(POSD + 1) for n in range(-XD//2, XD//2 +1, 1)},
                    ('op_pos', 'run'): {(m, n):0 for m in range(POSD + 1) for n in range(-XD//2, XD//2 + 1, 1)},
                    ('life', 'run'): {(m, n):0 for m in (-1, 1) for n in range(-XD//2, XD//2 + 1, 1)},
                    ('op_life', 'run'): {(m, n):0 for m in (-1, 1) for n in range(-XD//2, XD//2 + 1, 1)},
                    ('direction', 'run'): {(m, n):0 for m in (-1, 1) for n in range(-XD//2, XD//2 + 1, 1)},
                    ('run_itself', 'run'): {(1, n):0 for n in range(-XD//2, XD//2 + 1, 1)},

                    ('ballv', 'placement'): {(m, n):0 for m in range(-VD//2, VD//2 + 1, 1) for n in range(POSD + 1)},
                    ('pos', 'placement'): {(m, n):0 for m in range(POSD + 1) for n in range(POSD + 1)},
                    ('op_pos', 'placement'): {(m, n):0 for m in range(POSD + 1) for n in range(POSD + 1)},
                    ('life', 'placement'): {(m, n):0 for m in (-1, 1) for n in range(POSD + 1)},
                    ('op_life', 'placement'): {(m, n):0 for m in (-1, 1) for n in range(POSD + 1)},
                    ('direction', 'placement'): {(m, n):0 for m in (-1, 1) for n in range(POSD + 1)},
                    ('placement_itself', 'placement'): {(1, n):0 for n in range(POSD + 1)},

                    ('ballv', 'acc'): {(m, n):0 for m in range(-VD//2, VD//2 + 1, 1) for n in range(-AD//2 - 1, AD//2 + 1, 1)},
                    ('pos', 'acc'): {(m, n):0 for m in range(POSD + 1) for n in range(-AD//2 - 1, AD//2 + 1, 1)},
                    ('op_pos', 'acc'): {(m, n):0 for m in range(POSD + 1) for n in range(-AD//2 - 1, AD//2 + 1, 1)},
                    ('life', 'acc'): {(m, n):0 for m in (-1, 1) for n in range(-AD//2 - 1, AD//2 + 1, 1)},
                    ('op_life', 'acc'): {(m, n):0 for m in (-1, 1) for n in range(-AD//2 - 1, AD//2 + 1, 1)},
                    ('direction', 'acc'): {(m, n):0 for m in (-1, 1) for n in range(-AD//2 - 1, AD//2 + 1, 1)},
                    ('acc_itself', 'acc'): {(1, n):0 for n in range(-AD//2 - 1, AD//2 + 1, 1)},

                    ('ballv', 'run_placement'): {(m, n):0 for m in range(-VD//2, VD//2 + 1, 1) for n in range(POSD + 1)},
                    ('pos', 'run_placement'): {(m, n):0 for m in range(POSD + 1) for n in range(POSD + 1)},
                    ('op_pos', 'run_placement'): {(m, n):0 for m in range(POSD + 1) for n in range(POSD + 1)},
                    ('life', 'run_placement'): {(m, n):0 for m in (-1, 1) for n in range(POSD + 1)},
                    ('op_life', 'run_placement'): {(m, n):0 for m in (-1, 1) for n in range(POSD + 1)},
                    ('direction', 'run_placement'): {(m, n):0 for m in (-1, 1) for n in range(POSD + 1)},
                    ('run_placement_itself', 'run_placement'): {(1, n):0 for n in range(POSD + 1)}}
                     # 这里的 direction 是我方跑位的方向。加速度方向影响应该不大，故而暂时不考虑。
                     # 这里是 x, a 本身的频率
```

data 是为了研究球速，我方位置，对方位置，生命值，跑位位置，跑位距离，跑位方向与我们需要决策的跑位，落点位置，球的加速，跑位位置之间的联系。

其内容是记录球速，落点位置，生命值，跑位位置，跑位距离，跑位方向分别与跑位，落点位置，加速，跑位位置配对时不同取值出现的频数。

我们观察这个字典中的一个元素：

('ballv', 'run'): {(m, n):0 for m in range(-VD//2, VD//2 + 1, 1) for n in range(-XD//2, XD//2 +1, 1)}

这一项是想研究 ballv 与 run 之间的联系。

这个元素的 Key 为('ballv', 'run')，value 为一个列表。

Value 列表的每一项又是一个字典。此字典的 key 为(m,n)，value 为对应的(m,n)出现的频数。元组(m,n)中，m 指的是 ballv(速度)离散化之后的值，n 指的是 run(距离)离散化之后的值。

data 中其他元素与之完全类似。

2.2 函数说明

1、 一系列将参数离散化的函数

posToInt 将位置离散化；

xToInt 将距离离散化；

vToInt 将速度离散化；

aToInt 将加速度离散化；

lifeToInt 将生命值离散化。

这五个函数类似，以 posToInt 为例说明。

【函数功能】将位置离散化，便于对历史统计数据的分析。

【函数接口】传参为这些变量的实际值，返回离散化之后的值。

【算法策略】将位置分为 20 份，求出具体的位置值对应在哪一份。

2、一系列用于更新 data 的函数

update_run 更新与跑位有关的频数；

update_placement 更新与球的落点有关的频数；

update_acc 更新与加速度有关的频数；

update_run_placement 更新与跑位距离有关的频数。

这四个函数完全类似，以 update_run 为例说明。

【函数功能】更新与跑位有关的频数。

【函数接口】传参为 self, r_ballv, r_pos, r_op_pos, r_life, r_op_life, direction, r_run，无返回值。

【算法策略】如 data 里与 run 有关的一项：

('ballv', 'run'): {(m, n):0 for m in range(-VD//2, VD//2 + 1, 1) for n in range(-XD//2, XD//2 +1, 1)}。(m.n)每出现一次，其对应的 value 便加一。

3、Leighton

【函数功能】用于计算出对方的跑位，迎球，和加速。

【函数接口】传参为 tb,ds，返回值为 x1(跑位), x2(迎球), acc_vector_real(加速)。

【算法策略】（1）首先通过 tb 中对方跑位，迎球的总距离和道具使用情况分别计算出对方加速前的速度和对方加速后的速度，最后结合我方道具的影响，计算出对方加速的真实情况；

（2）通过 tb 中生命值得信息求得对方生命值的减少总量，再结合（1）可算出对方因加速而消耗的生命值，从而得到对方因跑位和迎球所消耗的生命值；

（3）通过生命值的消耗和跑位迎球的总距离，以及讨论是否使用变压器，瞬移术，隐身卡的情况，得到对方跑位，迎球的信息。

4、一系列关于 tb 与 ds 的处理的函数

newtb

【函数功能】推算出下回合对方迎球时接收到的 tb 数据。

【函数接口】传参为 tb, acc, run, cardaction，返回值为下回合对方迎球时接收到的 TableData(tb.tick, tb.step, dict_side, dict_op_side, dict_ball, dict_card).

update_ds_tb

【函数功能】用于在计算出上一局对方的情况后，更新概率值。

【函数接口】传参为 tb,ds，无返回值。

【算法策略】此处对 ds 的结构做出说明：字典中有两个键值对，一个是'lala_tb'，另一个是'last_tb'。

'lala_tb'记录的是我方上上个回合看到的桌面信息，用于计算对方决策跑位时看到的状况（和我方这个回合计算出来的对方上上回合的跑位一起用于计算概率），即双方的体力值，球的速度，对方当时的位置（也就是我方上上上个回合球的落点），我方当时的位置；

'last_tb'记录我方上个回合的桌面状况，用于计算上个回合对方决策加速度时看到的状况，这一个和我们这个回合计算出来的加速度一起用于估计加速度相关的概率。

update_ds_op_tb

【函数功能】用于根据我们这局的决策算出下一局对方面对的情况后，更新对方面对情况的记录。

【函数接口】传参为 tb, ds, acc, run, cardaction，无返回值。

5、一系列预测对方行为概率的函数

eval_prob_placement 预测球落在不同位置的概率；

eval_prob_run 预测对方选择不同长度的跑位距离的概率；

eval_prob_run_placement 预测跑位到不同位置的概率；

eval_prob_acc 预测球不同加速的概率。

这四个函数类似，以 eval_prob_placement 为例说明。

【函数功能】预测球落在不同位置的概率。

【函数接口】 传参为 tb, ds, acc, run, cardaction，返回值[(probability, value)]为一个列表，列表里每一个元素为一个元组，元组第 0 个元素是估计出的概率，第 1 个元素是该概率所对应的位置。

【算法策略】这里用到的统计方法由刘天乐同学提出，涉及贝叶斯分类等，具体内容已在算法部分阐述。

6、一系列对上述概率值进行评估的函数

access_run_acc 评估对球的加速的预测是否可用；

access_placement 评估对球的落点的预测是否可用；

access_run_placement 评估对跑位距离的预测是否可用。

这三个函数类似，以 access_run_acc 为例说明。

【函数功能】评估对球的加速的预测是否可用。

【函数接口】传参为概率分布列表，若评估效果好，则返回预测的结果，否则返回 None。

7、serve

【函数功能】确定发球策略。

【函数接口】传参为 name, ds,返回值为 BALL_POS[1], 278。

8、一系列计算己方加速，跑位等决策的函数。

eval_acc

【函数功能】做出己方加速的决策。

【函数接口】传参为 tb, ds,返回值为生命值消耗最小的加速方案。

【算法策略】利用上述预测对方跑位概率的函数及对概率的评估的函数来估计对方跑位后的位置，再计算不同加速方案对应的生命值的损耗。

eval_run

【函数功能】做出己方跑位的决策。

【函数接口】传参为 tb, ds, acc, run, cardaction，返回值为跑位的决策 run_action。

【算法策略】利用上述预测对方落点概率的函数及对概率进行评估的函数，且考虑己方生命值的剩余量。

9、selectCard

【函数功能】选择道具。

【函数接口】传参为 tb, ds, acc, run, lst_a,返回值为本回合所使用的道具 selectcard。

【算法策略】如果没有任何道具，返回 None；

如果有掉血包、补血包、隐身术，则直接用；

如果我方跑位超过 CARD_TLPT_PARAM,或者任何一方生命值较低（游戏快结束），并且有瞬移术 就使用瞬移术；

如果对方加速超过 200，或者任何一方生命值较低（游戏快结束），并且有变压器，就用变压器；

如果对方加速超过 200，或者任何一方生命值较低（游戏快结束），并有旋转球，就用旋转球；

如果我方道具盒已满，（在不满足上述条件下的）优先级为变压器>旋转球>瞬移术。

10、play_acc

【函数功能】决策加速度。

【函数接口】传参为 tb, ds, prob_acc, prob_run, newrandom, 返回值为加速的最佳决策。

【算法策略】策略分为如下三种：打角，接道具，不加速。

先分别在三种内部评估，通过给每种做法打分，最终选择分数值（对方的生命损失-己方损失）最高的方式，己方损失是 acc 损失，对方损失是迎球的移动损失，对于接道具则有额外的分数加成。

再比较最优打角，最优接道具和不加速的方案，从中选出最终的决策。

11、play

【函数功能】选择调用以上函数，决策出最终球拍的行为。

【函数接口】传参为 tb,ds，返回值是球拍的行为

RacketAction(tb.tick,tb.ball['position'].y - tb.side['position'].y, accaction，runaction, None, cardaction)。

【算法策略】该函数包括:

对 ds 的初始处理和更新；

根据对方生命值与己方生命值的比例不同而采取的随机应对的算法；

对预测对方跑位和加速的函数(access_run_placement 和 access_run_acc)的调用；

对道具决策函数(selectCard)和加速决策函数(play_acc)的调用；

子函数 run_tb(tb), 该函数只根据此回合己方接球后的位置来决定跑位，设置两个比例将位置分为三个对称的区间，分别采取不同的跑位：靠近角落则跑位为到另一角落一半的距离；不靠近角落也不靠近中间则跑位到中间；靠近中间则往中间跑一格（假跑位）。

最后，根据 ds 得出的对球落点的预测，根据剩余血量，预测精准度选择合适的跑位策略。

2.3 程序限制

算法经过多次修改完善，对可能出现的 bug 做出了相应处理，在实际对战中没有出现报错的情况。

leighton 函数中解二次方程，由于生命值取整，该误差可能会导致求解对方上回合跑位时解出复数解，此时则返回 None，代表放弃此回合数据。

由于 table 对跑位的范围没有做出限制，对方跑位的结果可能跑出球桌，所以我们在统计跑位的函数中考虑到该极端情况，对跑位的数值进行限制。

在有可能出 bug 的部分使用 try 语句。

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 Core i5-5200U CPU @ 2.20GHz, 4.0GB RAM

- 操作系统：（名称/版本）Windows 10 64-bit
- Python 版本：（版本号）3.5.3

测试说明：

主要通过研发的算法互相对打，或者改进的算法和原算法对打计算胜率，并且分析复盘数据来测试。

3.1.2 测试程序：

1. 改写 pingpong 使其统计胜率

   race 函数增加返回值 main_table.winner 便于统计

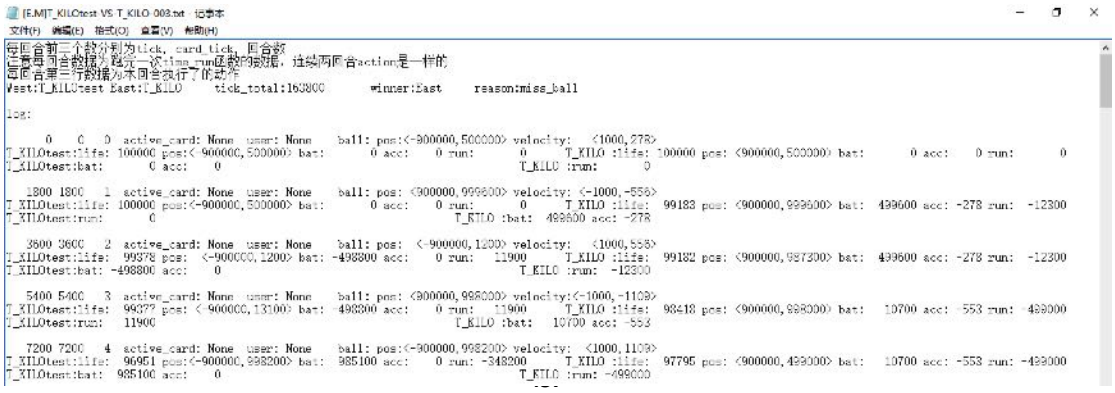
```
        for i in range(ROUND_NUMBER):
            winner = race(i, west_name, WP.serve, WP.play, WP.summarize, east_name, EP.serve, EP.play, EP.summarize)
            if west_name == 'T_KILO' and east_name == 'T_KILOtest':
                alls += 1
                if winner == 'East':
                    win += 1
            if west_name == 'T_KILOtest' and east_name == 'T_KILO':
                alls += 1
                if winner == 'West':
                    win += 1
    print(win, alls, win / alls, sep='\t')

```

   以及在最后加上统计胜负部分，并根据对战结果对原函数的参数进行调整。

2. 改进 reader 为 new_reader 使其输出的 txt 文件更易读

   新版优点：增加了每回合使用的道具，增加了每回合执行的动作避免没执行的动作影响可读性。以下为输出 txt 示例。



3.1.3 热身赛结果：

3.1.3.1 第一次热身赛结果

参加第一次热身赛时，我们的 KILO 算法还尚未加入学习算法，仅有接道具，打角落和其相应的评估函数。不过由于其他大多数组也没有完成代码，因此我们的函数表现尚佳，在半区内以赢 129 负 21 的结果名列第二。

下图为第一次热身赛的部分截图（本组为 KILO 组）

```
T_shlcy
 111

T_KILO
 129

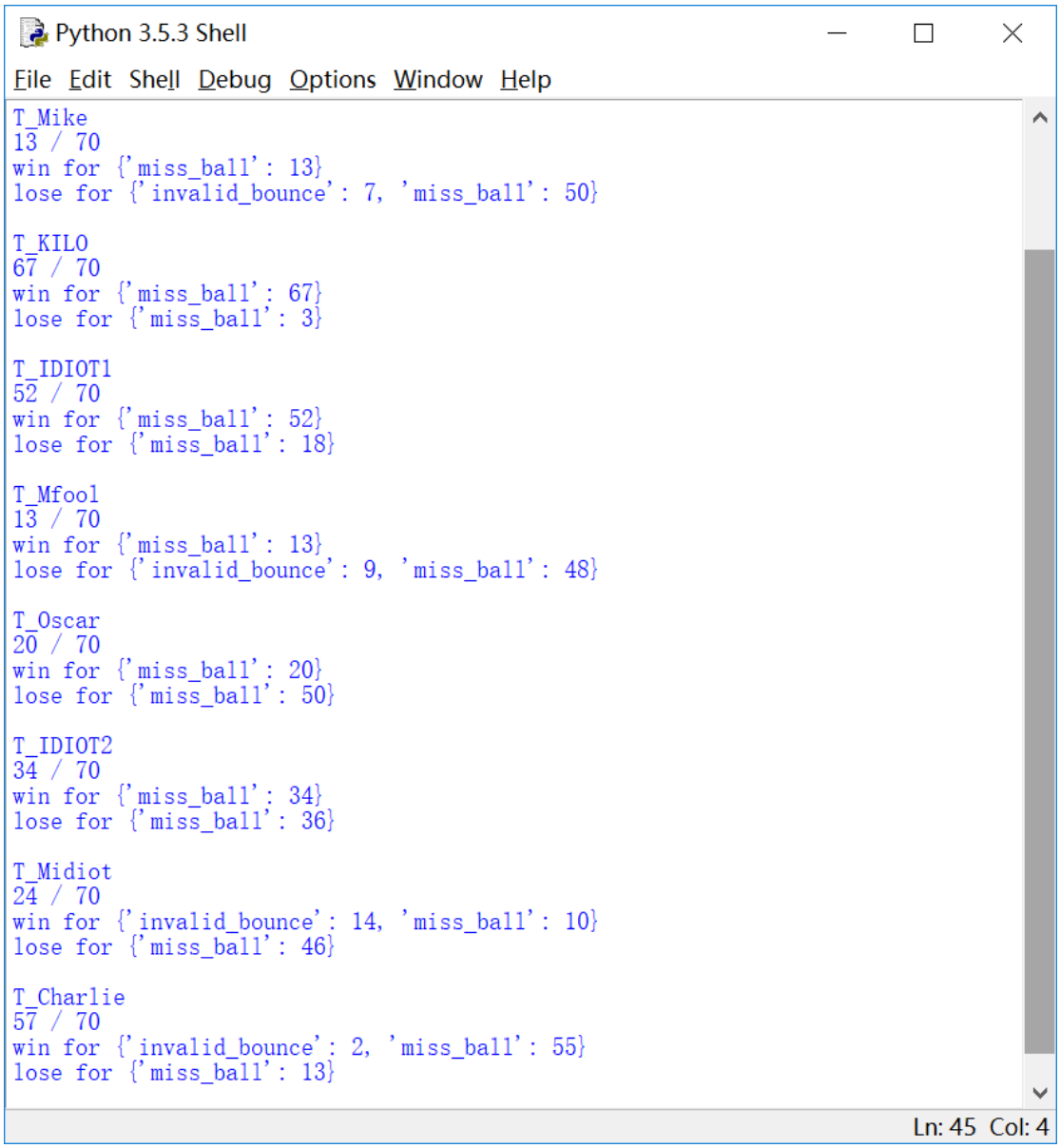
T_FANCY
 73
```

3.1.3.2MOCK 热身赛

本次比赛是由 Charlie 组的郭浩同学提议进行的内部比赛，有第一次热身赛中排名靠前的 Mike 组，Oscar 组，Charlie 组，Kilo 组参加，并由助教进行评测的一次热身赛。

由于参加这次热身赛时我们的学习算法已经完成，而 Oscar 组和 Mike 组还未加入学习算法，因此我们取得了比较好的成绩（赢 67 输 3）。

以下的 KILO，IDIOT1，IDIOT2 都是我们组参赛的代码



3.1.3.3 第二次热身赛

由于已经加入学习算法，与上次的算法差别不大。最终结果赢 155 输 15 局，在小组之间的对抗中输给 Oscar 组，与 Alpha 组战平。

下图为第二次热身赛的部分截图（本组为 lkjhg 组）

```
T_David
5 / 175
win for {'miss_ball': 2, 'invalid_bounce': 3}
lose for {'miss_ball': 125, 'invalid_bounce': 45}

T_AVCNA
128 / 177
win for {'miss_ball': 91, 'invalid_bounce': 37}
lose for {'miss_ball': 49}

T_zxcvb
10 / 38
win for {'miss_ball': 7, 'invalid_bounce': 3}
lose for {'miss_ball': 13, 'invalid_bounce': 15}

T_lkjhg
155 / 170
win for {'miss_ball': 111, 'invalid_bounce': 44}
lose for {'miss_ball': 15}

T_unique
38 / 170
win for {'miss_ball': 14, 'invalid_bounce': 24}
lose for {'miss_ball': 132}

T_rsbgd
0
win for {}
lose for {'miss_ball': 9, 'invalid_bounce': 1}

T_test
4 / 10
win for {'miss_ball': 4}
lose for {'miss_ball': 6}

T_King
69 / 170
win for {'miss_ball': 47, 'invalid_bounce': 22}
lose for {'miss_ball': 101}

T_ALPHA
157 / 185
win for {'miss_ball': 124, 'invalid_bounce': 33}
lose for {'miss_ball': 22, 'invalid_bounce': 6}
```

3.2 结果分析

3.2.1 暴力打角和接道具

由于在角落处己方加速度生命消耗往往最小而对方迎球消耗往往最大，因此暴力打角落和接道具就成了可能的最好加速度情况。我们对 8 种打角落和每个道具的 5 种方式进行评估，选出最好的一种。

在实战中（例如第一次热身赛），表现良好，可以战胜大多数未加入学习算法的函数。

由于打角方式和接道具方式都有上限，因此运算量每次几乎为常数级。

3.2.2 学习算法

学习算法采用朴素贝叶斯的数学手段，对对方的跑位和击球后球的落点进行估计，返回值供 acc 评估和 run 评估参考。在加入学习算法之后，跑位的效果明显提高：

当对方的算法未加入学习或者随机的情况下，对方落点和跑位的预测概率非常集中，即预测完全准确，在这种情况下，几乎可以大比分完胜。

当对方也为学习算法时，预测结果仍有很大的价值，大幅度提高跑位的准确性。

时间开销较非学习部分较大，改进这部分算法之后的时间开销仍然保证在可控的范围内，每回合耗时不超过 0.5s。

4 实习过程总结

4.1 分工与合作

在老师发布大作业组队通知 5 月 11 日当天下午，KILO 小组就在线上通过微信组建起来，6 名来自地空 16 级不同班级的同学为了大作业这个共同的目标聚集起来并一致推选出邵俊宁同学为组长。在小组正式组会之前，大家先在群里讨论了对大作业的想法，及时将课程网站上更新的资料上传共享，组长也督促了各位组员提前将大作业细则要求和代码吃透。

第一次组会：

定于 5 月 21 日下午在二教树下召开，小组成员全员准时到齐，并在经历了人群中搜寻观望、微信交流确定身份的小小坎坷之后成功碰头。由于此前男生组员和女生组员互不认识而内部熟悉的复杂关系，组会的第一件大事就是简单的自我介绍和互相认识，组长也初步了解了大家工作方向的意愿。之后我组就迅速进入了严肃活泼的正事儿讨论环节，经过大家对自己初期想法、疑问的交流，第一次组会主要得出了以下结果：

1) 打角策略。

2) 跑位应该要依赖于对对方策略的预测。

3) 道具一个回合大约可出现 20 个，道具问题我们主要考虑将其映射成体力值来加入计算之中。

4) 体力值减少以后，跑位会受到限制。这一点我们之前从未将之纳入考虑范围内，但是这在后期是十分重要的。

5) 对于对方行为的统计性预测可以用类似于分类对方性格的方式来进行（如保守型，攻击型）。其他的模型亟待思考。

6) 需要对任务进行一定的分工。具体分工在自愿原则上由组长分配。同时组长将设计一份时间表来督促任务的完成。

7) 我们的策略可以采用变化的形式以防止对方的预测。



合影 mark 面基成功

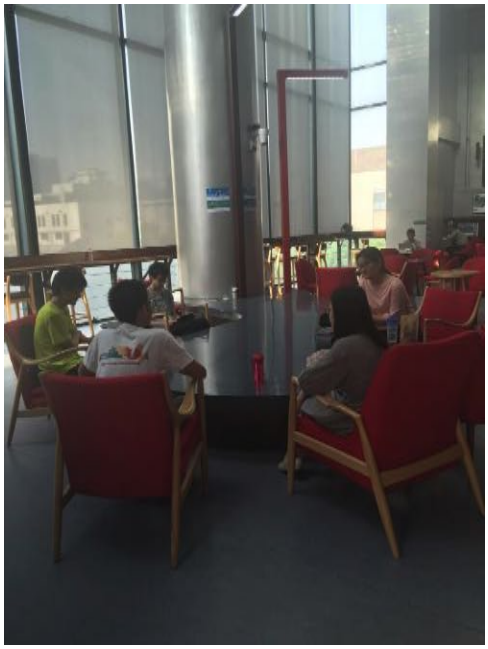


严肃活泼的讨论

第一次组会高效而愉快地结束了，组长及时将任务分配情况发到了群里。由于是第一次任务分配，也处于从头开始的初期阶段，每名组员都有或轻或重的编程任务，负责不同的模块，使大家能够迅速熟悉整个算法思想。

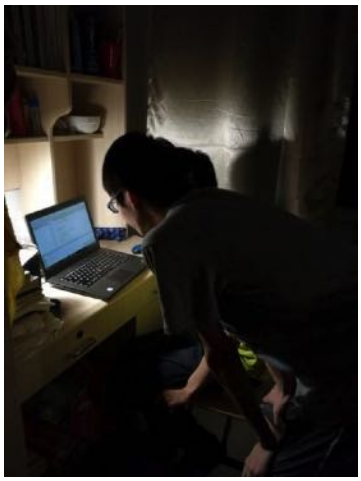
第二次组会：

经过一周，所有组员都在原定的 5 月 26 日的 DDL 前完成了自己的任务。于是，在 5 月 27 日我们又召开了第二次组会，本次组会由于某些原因，被迫定在早晨 7：30，地点依然是二教树下。非常令人感动的是大家带着睡意还是基本准时到达了，战胜起床气顺利进行了组会。会上总结了上周的工作进展，又正式做出了组内的分工：邵俊宁、刘天乐、张祖煜和黄丽潼负责代码编写，测试也主要由三位男生完成，张溶倩和李然好负责报告。另外，大家一起讨论了对算法的想法，主要是对道具的使用做了详细的考量。



清晨的树下

确定分工之后，代码组的工作节奏就迅速加快了，三名男生由于同住一个宿舍的地理优势，讨论交流更加方便。编程、测试的工作任重道远，大家都辛苦地不断调试改进，也在一次次热身对战中吸取经验，做出完善。



黑夜给了我黑色的眼睛，我却用它寻找 bug



……debug 到天明

分工合作：

- 算法思想：刘天乐
- 代码编写：

  张祖煜、邵俊宁、刘天乐

  黄丽潼：leighton(tb,ds)函数、selectCard(tb,ds)函数。

  张溶倩：play_acc1(tb, prob_run, newrandom)的一部分。
- Debug：张祖煜
- 报告书写：

  算法思想：张溶倩

  流程图：张溶倩（分图 2、分图 3）、黄丽潼（主图、分图 1）、李然好（分图 4）

  算法复杂度分析：李然好

  程序代码说明：李然好

  实验结果：邵俊宁、张祖煜、刘天乐

  分工与合作：张溶倩

  排版与修改：黄丽潼

4.2 经验与教训

尽吾志也而不能至者，可以无悔矣，其孰能讥之乎?

本次实习大作业我们小组做的比较好的地方就是每个人都很有责任感，虽然能力有限，但是每个人都尽自己的努力完成了分配的任务，没有拖欠任务、推卸责任的现象；小组的气氛也很融洽。

不足之处，时间安排不够合理，前紧后松。在编程的过程中刚开始大家都参与度很高，在较短的时间内完成了算法，在小组赛中表现优异；但是对程序和算法进一步的修改、思路的改进不足，导致在实战中惜败。

4.3 建议与设想

1、竞赛不分区进行，组与组间两两对战；或者保证每个分区的小组数相同，每个分区胜出小组占比相同，使竞赛更加公平，减少“运气”因素。
2、建议竞赛部分给分细化。没有出线的小组之间给分也应根据胜率有所差异。

5 致谢

感谢陈斌老师在数据结构与算法这门课上做出的贡献。
感谢冀锐助教为各个小组提供热身赛的机会。



JULIET

组长：朴健

组员：郭浩（城环），国和一，鲁尚宗，谢晓凡，杨欣宇，尹赣闽

小组赛战绩：N4

数据结构与算法课程 实习作业报告

（朴健* 1600012463，杨欣宇 1600012445，国和一 1600012407，鲁尚宗 1600012412，尹赣闽 1600012436，郭浩 1300011074， 谢晓凡 1600012418）

摘要：

漂移乒乓取胜的核心就是在保存自身体力的基础上尽量消耗对方体力。因此本组最终的算法思路就是在每一次击球中皆找到最优解，使得我方收益最大（对方的体力消耗减去我方的体力消耗再加上道具带来的体力收益达到最大值），主要用到了列表和字典的数据结构，实验结果为与 Idiot 的对战全胜，在热身赛中胜率为 50%，在小组赛中北区排名第四，无缘八强。

关键字：

编程应用，列表，循环

1 算法思想

1.1 总体思路

漂移乒乓的游戏规则是接不到球或者体力降为零时判负，在第一种情况中，拍移动速度受到体力值限制，体力值越低，移动速度越慢，会导致接不到球，而第二种情况中，体力高低直接影响了胜负，因此取胜的核心就是在保存自身体力的基础上尽量消耗对方体力。另外，发球策略为打角，随机数决定左右。

于此同时，游戏规则还有限制，必须保证乒乓球在墙壁上的反弹次数为 1 或 2 次，另外，还可以击球抓取各种道具，以实现各种功能。本组最终的算法的思路就是在每一次击球中找到最优解，使得对方的体力消耗减去我方的体力消耗数值最大，同时考虑到道具使用带来的体力收益。因此定义因变量“收益”：

“收益”= “对方体力消耗”-“我方体力消耗”+“道具带来的等价体力值收益”

以上函数的自变量为我方击球的速度，“对方体力消耗”为我方击球后造成对方跑位所消耗的体力，而“我方体力消耗”为我方击球为改变速度而消耗的体力，“道具带来的等价体力值收益”为每一次使用道具为我方恢复的体力或者给对对方带来的体力损失。每一次击球，对这个函数求最优解。自变量速度的区间为使得乒乓球在墙壁上的反弹次数为 1 到 2 次或者实现打角的速度范围，用 A 来表示。以 A 区间为定义域，求解因变量的最大值，以获得最优解。

定义域 A 为：

$$\left[-\frac{2h+x}{t},-\frac{x}{t}\right]\mathrm{U}\left[\frac{h-x}{t},\frac{3h-x}{t}\right]$$

收益的函数表达式为（分为三种情况）：

$$profit=\frac{(2h-y_0-v_yt-y_d)^2}{a}-\frac{(v_y-v_0)^2}{a},x\in\left[\frac{h-y_0}{t},\frac{2h-y_0}{t}\right)$$

$$profit=\frac{(-y_0-v_yt-y_d)^2}{a}-\frac{(v_y-v_0)^2}{a},x\in\left(\frac{-h-y_0}{t},-\frac{y_0}{t}\right]$$

$$profit=\frac{(2h+y_0+v_yt-y_d)^2}{a}-\frac{(v_y-v_0)^2}{a},x\in\left(\frac{-2h-y_0}{t},\frac{-h-y_0}{t}\right]$$

其中 v_0 为击球前的初始速度，v_y 为击球速度，y_0 为我方位置，y_d 为对方位置，a 为系数，$profit$ 为收益。

1.2 算法流程图

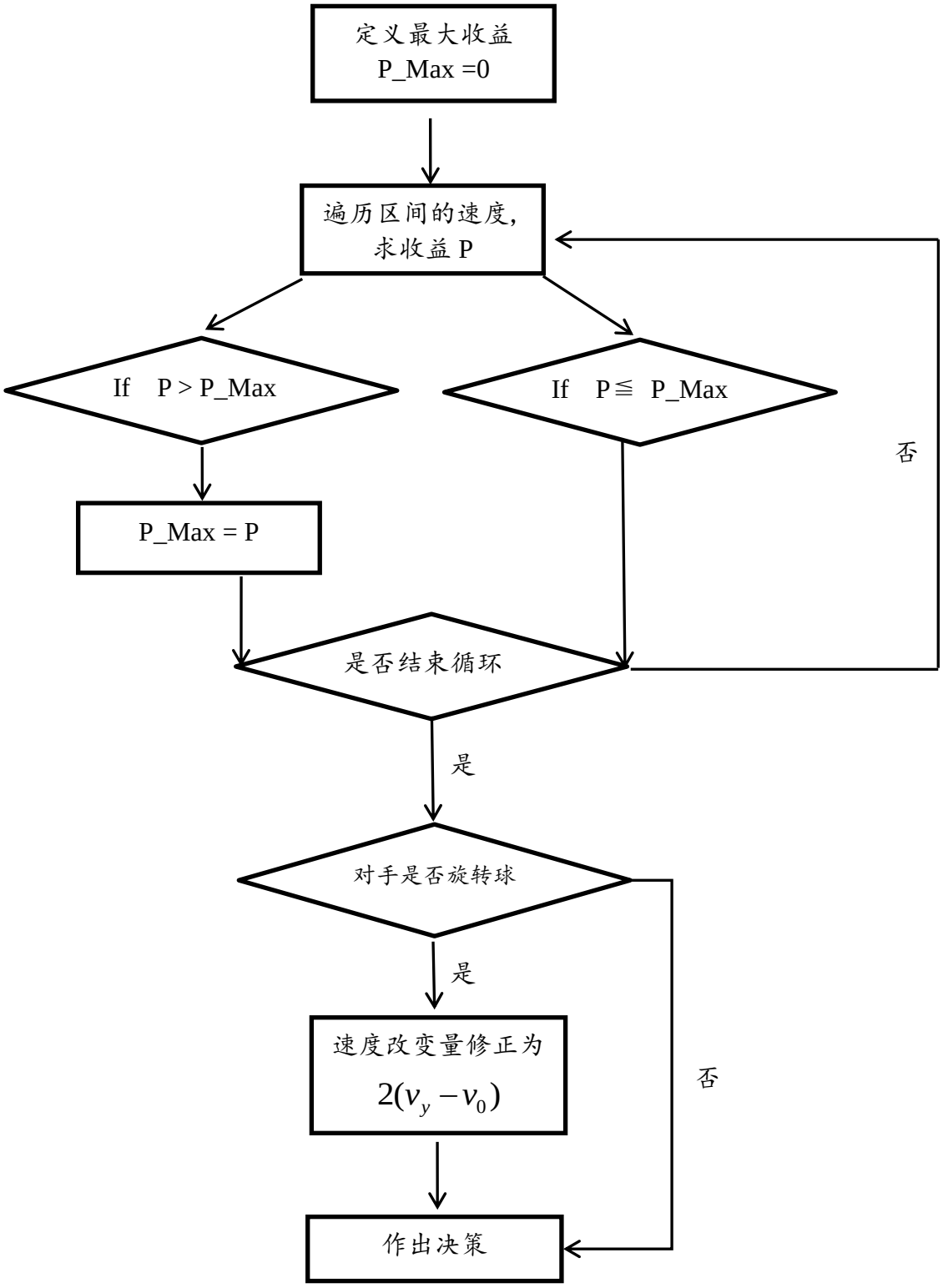


图 1.1 漂移乒乓算法流程图

1.3 算法运行时间复杂度分析

本算法时间主要用在速度区间内遍历全部的整数解上，平均一局 2~5 s，用单个 for 循环，复杂度为 O（n），n 比较大，因此时间较长。

2 程序代码说明

2.1 数据结构说明

用得最多的数据结构就是列表，因为这次作业是竞赛，实战性比较强，采用列表这种比较简单易行的数据结构反而效果更好。

2.2 函数说明

主要函数如下所示：

函数 1：Output_opposite（对方体力消耗）

根据对方跑位的距离计算我方击球后造成对方跑位所消耗的体力，，假设对方跑位到中间。

函数 2：Output_us（我方的体力消耗）

我方击球为改变速度而消耗的体力。

函数 3：Profit_Cards（试用道具带来的等价体力值收益）

针对该次使用的道具，计算我方恢复的体力或者给对方带来的体力损失。:

函数 4：DicProfit_Cards：可获得道具带来的等价体力值收益的字典

针对每一个可获得的道具，计算分别对应的等价体力值收益。

函数 5：Serve（发球函数）

发对角，用随机数决定左右。

函数 6：Play（击球函数）

引用 Output_opposite，Output_us，Profit_Cards，DicProfit_Cards 等函数计算收益的最优解，完成击球动作。

2.3 程序限制

最主要的问题是不能保证 100%捡到道具，丢包率 20-30%左右。

3 实习过程总结

3.1 组会记录

▲第一次组会讨论（2017.5.22）

今天我们在四教一个温馨的小教室讨论，留下了第一张合影。

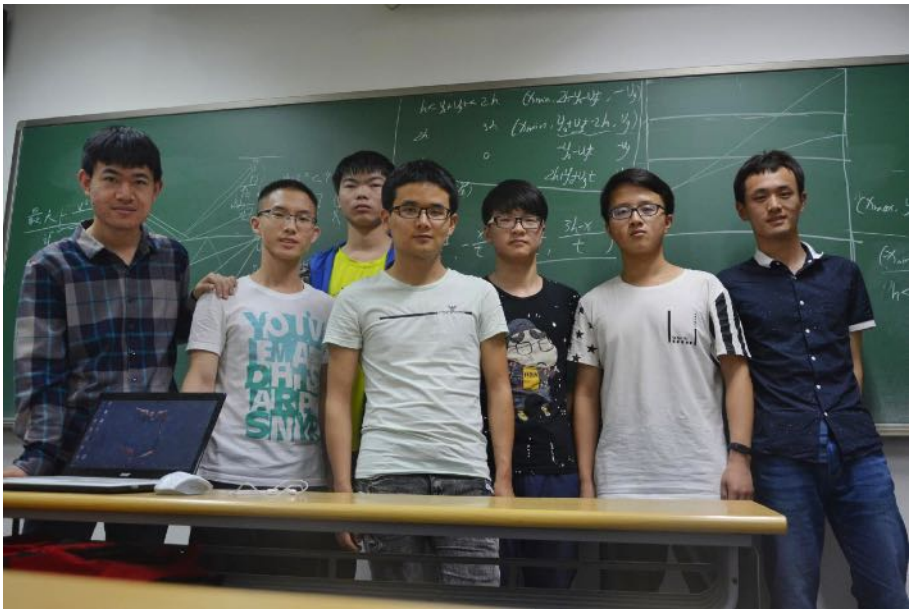


图 3.1 第一次组会的合影

主要讨论的是击球策略问题，组长带领小组成员进行了头脑风暴。



图 3.2 组长带领大家进行头脑风暴

如何在乒乓赛场上勇胜对手呢？考虑到体力降为负是被淘汰出局的重要条

件，我们权衡了战术——在每一个回合中尽量保证我方体力的损失少于对方体力的损失。

首先是发球，发球时要考虑下一轮接球的位置移动不能太远，因此小组成员一致决定站在中间位置发球。由于发球速度越快，在墙壁上反弹的次数也会越多，而速度太高会带来我方体力的消耗，因此我们决定以最小的速度，直接发到对方的边角处，同时满足了一次反弹的条件。于此同时，对方接球跑动的距离也相对较远。

然后开始接球环节，依然是以消耗对方体力并保存我方体力为原则，建立函数关系：

净胜体力=对方体力消耗-我方体力消耗+道具体力补偿

当然，接球要考虑来球的速度，以满足在墙壁上反弹 1-2 次的规定。

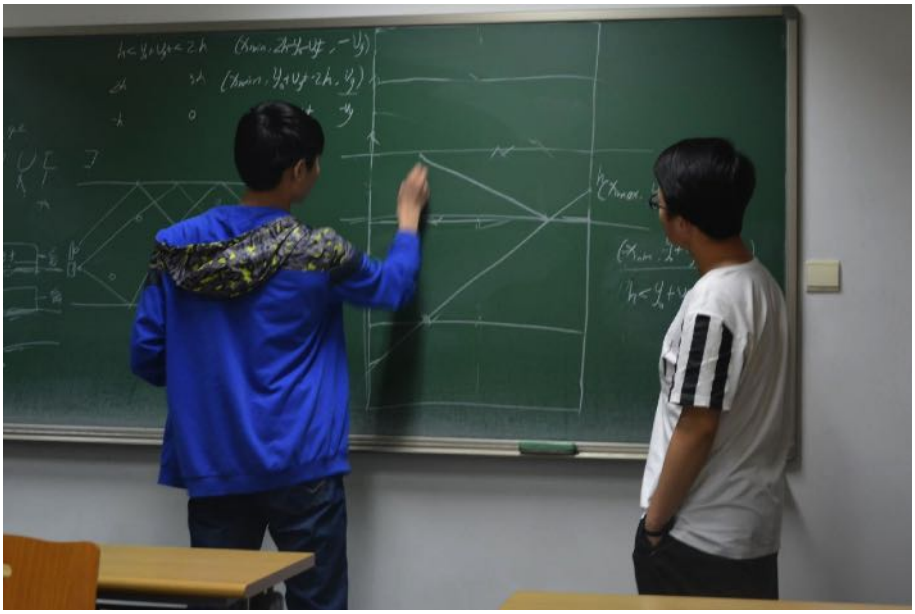


图 3.3 杨欣宇同学计算出了击球速度范围 A

击球可分为两种情况：1.抓取道具 2.不管道具

情况 1：对于此时出现在球桌上的 n 个道具，计算抓取每一种道具的击球路线，并求出净胜体力。这样有 n-2n（反弹 1 次或 2 次）种情况，选取击球速度符合范围 A 的情况，比较净胜体力的大小，选取速度符合要求的净胜体力最大的一个最佳道具。

情况 2：不管道具，直接算出净胜体力最大的最佳路线。（速度范围 A 为自变量变化区间，对函数求极大值即可。）

比较最佳道具和最佳路线中的最优解，并作出相应选择。



图 3.4 组员们进行激烈的讨论

由于我方并不知道对方发完球移动之后的准确位置，只知道对方的移动方向，因此情况变得十分复杂，但是后来组员一致同意，从平均情况来看，不管对方如何移动，我方假设对方不动作出的最优解依然有最佳的掉血效果。

因此，目前的问题就是对净胜体力的函数求极值。

而对于道具和接球之后跑位的策略，组员们一致认为除了补血包外，其余道具得到即用，而补血包留到关键时候应用。

接球之后跑位，为平衡边角情况起见，一致往中间位置跑位。

晚上，组长算出了求最优解的情况。

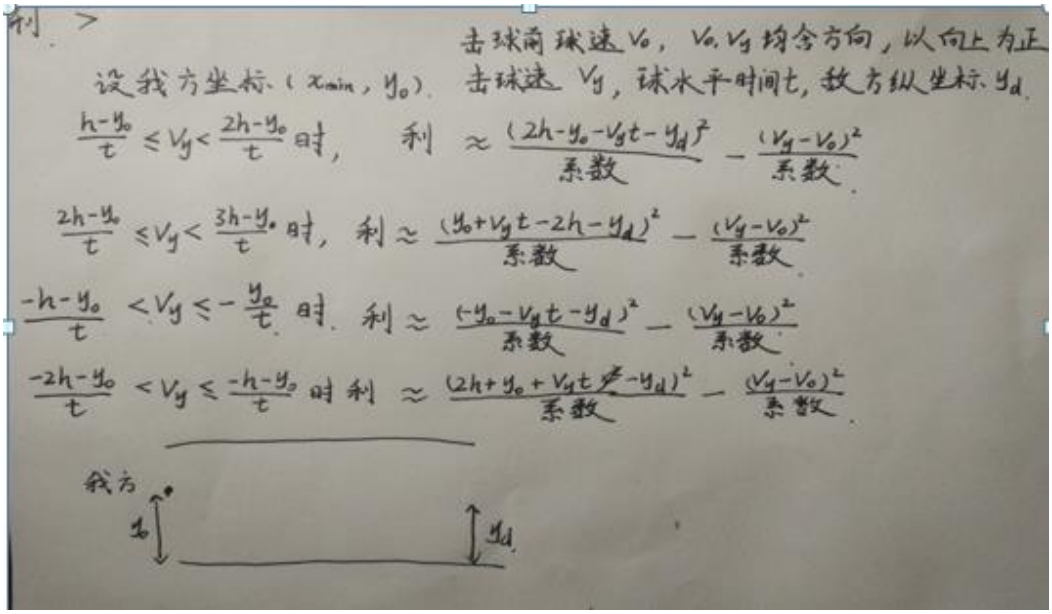


图 3.5 组长算出来的最优解

▲第二次组会讨论（2017.5.25）

今天我们主要讨论关于策略的问题。

每个人在上次讨论之后，就开始思考击球的策略。经过大家的讨论和交流，最后确定了两个策略，作为我们代码的基本策略：

1. 以防守为主，打角落。

2. 在捡道具过程中，进行遍历计算，以确定是否要捡道具、是否要对球进行加速等问题。



图 3.6 第二次讨论的场景

之后，为了大作业能够更有效地完成，组长进行了任务分配：将全组人员分为代码编写组（朴健、尹赣闽、杨欣宇）、代码测试组（国和一、鲁尚宗）、报告完成组（谢晓凡、郭浩）。分工完成之后，每个同学都开始去做自己的工作，大作业在顺利进行着，极大地提高了每个人的积极性及兴趣。

代码编写组：三个组员每人编写一个代码，用统一接口。然后三个代码取长补短、求同存异，最后编写出一个比较满意的代码。

代码测试组：配合代码编写组，及时对代码编写组编写的代码进行测试，并将发现的问题及时反映给代码编写组，与代码编写组成员进行讨论和交流，以解决问题。

报告完成组：对整个过程进行跟踪，及时收集数据信息，最后完成报告。

3.2 分工与合作

表 3.1 组内分工与合作的细节

姓名	分工
朴健*、尹赣闽、杨欣宇	代码的编写润色，及时解决测试组反映的问题。
国和一、鲁尚宗	代码的运行及测试，并及时将发现的问题反馈给代码编写组。
郭浩、谢晓凡	对整个过程进行跟踪记录，最后完成报告。

分工完成之后，代码编写组组员杨欣宇和尹赣闽开始初步的代码编写。经过一段时间的编写之后，两位同学各自完成了代码。两位同学将自己的代码发给了组长朴健。朴健在两位代码测试组组员国和一、鲁尚宗的帮助下，对这两位同学的代码进行了认真的分析，结果发现，杨欣宇同学所编写的代码只能体现思路，却无法正确运行；而尹赣闽同学所编写的代码可以正确运行，但乒乓球反弹次数会出错。综合了两个代码及自己的想法后，组长朴健编写出了第一代代码 Jack（2017.5.26）.（括号内日期表示代码编写成功日期，下同）

但是，在测试组测试过程中 jack 还是出现了一些问题。代码测试组组员及时将这个问题反映给了代码编写组。Jack 虽然可以正常运行，并且可以与老师给的模板代码进行对战，但其在捡道具方面有些问题：捡到道具只能靠运气，而不能进行运算，主动地去捡道具。关于这个问题，三位编写代码的同学进行了多次讨论，并对 Jack 进行了改进。在三位编写代码的同学及两位测验代码的同学鲁尚宗、国和一的努力下，第二代代码出世了！他就是 Harry（2017.6.2）.测试组及时对 Harry 进行了测试。

Harry 改正了 Jack 不能主动捡道具的毛病，但在代码测试组成员的测试下，发现 Harry 还是存在一些问题：Harry 捡到道具的概率为 70%~80%！这个问题可让代码编写组的成员挠破了头。为了解决这次的问题，代码编写组和代码测试组花费了较长的时间，不断对代码进行改进和测试，最终还是克服了困难，编写出了最终代码 Juliet（2017.6.4）！

Juliet 极大地提高了捡道具的准确度，并且与 Harry 相比，在给乒乓球加速方面节省了不少体力。这样的改进，使得 Juliet 在与 Jack 和 Harry 的对战中可以吊打 Jack 和 Harry。

代码编写和测试告一段落。接下来就是两位写报告的同学出场的时候了。

两位同学运用跟踪记录的数据，开始写报告，其间，还征求了代码编写组和代码测试组成员的意见及建议。最后，完成了报告。

3.3 经验与教训

1.分工有问题。虽说分工一定程度上提高了组员积极性和效率，但还是有些不完美。比如测试代码不需要两个人，测试代码很多工作是写代码的人顺便做的。而且应该也让报告完成组的参与代码编写和测试过程的讨论交流中。

2.写出代码后没有在其他方向上延伸，去突破原有思路。

3.4 建议与设想

▲建议：

1.其实代码想要运行比较简单，可以让每个人都想一种方法，写出代码，相互对战，以取得最优解。

2.不断更新代码给编代码带来麻烦，每改一次都要修改相应的部分。

3.布置的时间太晚，应对作业的时间太匆忙，之前并不了解机器学习方法的在一周之内 很难上手。

4.提前一个月布置，设置不同阶段的任务，一步步地完成作业。比如第一周先交思路，下一周交初步代码，之后不断地改进。

5.在组队过程中还是不可避免地出现了“大腿”抱团的现象。

6.竞赛的时候希望有几个讲解员，因为有的同学不太能看懂比赛如何进行。

▲寄语：

好好学习这门课吧，它可以让你受益终身！希望大家可以积极选这门课，因为这门课老师讲的好，助教负责任。

▲设想：

可以尝试编写一些简单的小游戏程序，然后进行游戏对战比赛。

4 致谢

感谢陈斌老师的辛勤授课，同时也感谢老师能给我们这样的机会，让我们体验自己编代码进行比赛的快乐。

感谢助教在此过程中的指导与包容。

感谢 Papa 组和 Oscar 组，提供他们不同版本的代码跟我们对战，使我们可以从中积累经验，对我们的代码不断进行改进。

感谢各位组员的辛苦付出，你们是坠吼的！

5 参考文献

1. www.gis4g.pku.edu.cn.
2. 数据与结构算法实验教程（地空版）



UNIFORM

组长：孟柳昳

组员：李昔筱，陆晓雨，唐晨晨，郑丁源，段麟飞，时雨晨

小组赛战绩：N5

Uniform 组漂移乒乓实习作业报告

孟柳昳（组长） 段麟飞 陆晓雨 李昔筱 时雨晨 唐晨晨 郑丁源

摘要

Uniform 小组在本次漂移乒乓竞赛中采用的主要是保存自身体力并尽可能消耗对方体力的“消耗战”策略。策略的应用主要体现在跑位与击球上，击球大部分情况选择打角策略，而实现跑位策略的具体的数据结构和算法则是利用博弈树与 Alpha-Beta 剪枝。算法的主体包括击球函数、道具函数、加速函数以及 play 函数等。本组的代码测试数据良好，在热身赛中获得 20%左右的胜率，热身赛的结果显示我方的击球函数控制较好，弱点在于跑位函数对边界状况的判断。之后小组对代码进行改进。小组在最终的比赛中获得小组第四未能出线。

关键词

数据类 博弈树 Alpha-Beta 剪枝

1 算法思想

1.1 总体思路

在回球策略上我方算法默认打到上角，若对手位于中线以上则打下角。计算比较打到上交或下角的不同坐标所需要的速度值后选择消耗体力最少的路径，若体力消耗太多则放弃进攻型策略，换另一角计算得到最节省体力的策略，并学习体力消耗较小的策略。

我方算法默认只有旋转（SPIN）、补血（INCL）、掉血（DECL）、变压器（AMPL）这四种道具是有用的，当遇到的是有用道具时计算四种获得道具的途径所消耗的体力值，只有当额外消耗的体力值大于预期收益时才选择获取道具的路径。掉血（DECL）和变压器（AMPL）这类 debuff 道具给对方使用，自己使用补血（INCL）。为探测对方的算法是否会将旋转（SPIN）道具使用在我方身上，我方第一次的旋转（SPIN）道具给自己使用之后则都用在对手身上。击球后从接球处回到中点，等待进行下一轮击球。

1.2 算法流程图

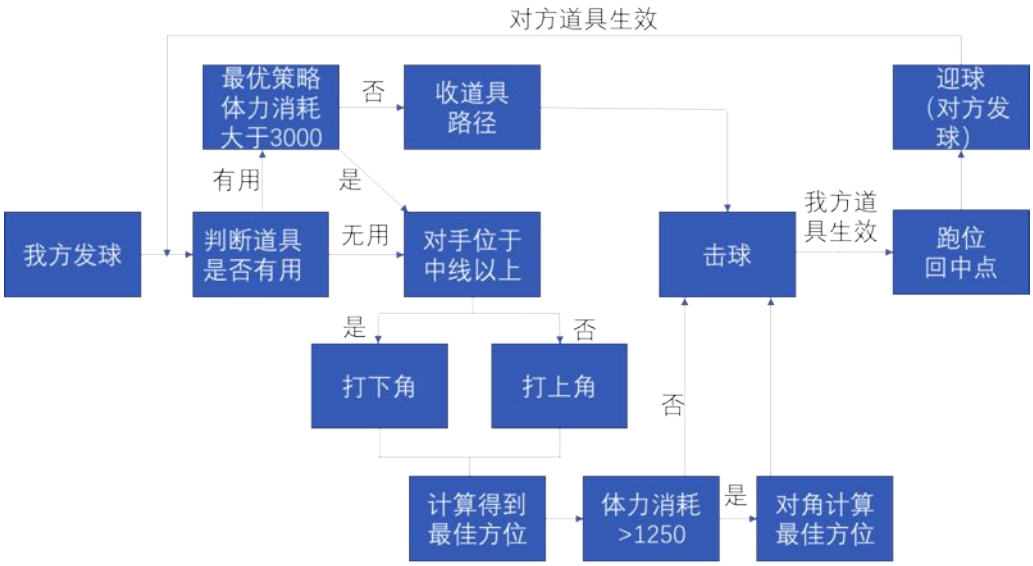


图 1.1 算法流程图

1.3 算法运行时间复杂度分析

对代码分析可知，循环部分为将首项与其余值进行比对得到最小值，执行时间是与问题规模 n 无关的常数，因此时间复杂度为常数阶 T（n）=O（1），时间效率比较高。

2 代码分析

2.1 数据结构说明

本次代码构建数据类 Nfunction，初始化过程中设定可用道具为：旋转（SPIN）、补血

（INCL）、掉血（DECL）、变压器（AMPL），其他均不使用；初始化“打角”策略中上角（远角）和下角（近角）的三种不同的纵坐标方位；初始化状态和文件记录。

```
def __init__(self):
    self.ratio = 1800 # pingpong会对serve的速度向量进行取整，在时间T内，y方向产生的最大误差为1800
    self.up = [DIM[3]*3 - self.ratio, DIM[3] + self.ratio, -DIM[3]] #打上角的三种方式的“上角”纵坐标，分别为[上下上、上、下上]
    self.down = [-self.ratio, -2*DIM[3]+self.ratio, DIM[3]*2]  #打下角的三种方式的“下角”的纵坐标，分别为[下、下上下、上下]
     self.mid = [-DIM[3]/2, -DIM[3] * 3/2, DIM[3] * 3/2, DIM[3] * 5/2]
    self.spin = 0  #初始化使用状态
    self.useful_card = [CARD_SPIN, CARD_INCL, CARD_DECL, CARD_AMPL]  #考虑使用的卡牌[‘旋转’，‘补血，’，‘掉血’，‘变压’]
    self.dic = None # 历史文件
    try: #write DS file
        f = open('DS_T_uniform', 'rb')
        self.dic = pickle.load(f)
        f.close()
    except:
        pass
```

2.2 函数说明

以下将通过对主要函数分析来具体分析本次代码：

1、击球函数

创建速度计算函数 cal_speed，利用的是 X 轴方向速度不变的原理。

```
#计算从spend发球，想要经过ppass需要的速度改变
def cal_speed(self, psend, ppass):   #psend:发球的位置坐标，ppass:想要经过的位置坐标
    t = abs(psend.x - ppass.x) / float(BALL_V[0])    #到达该点所需时间：利用横轴速度不变
    d = float(ppass.y - psend.y)     #两点在y方向上的距离（在没有墙的时候）
    return d/t   #需要的速度，pingpong在运行时会对这个速度进行取整
```

创建体力消耗计算函数 cost:

```
def cost(self, sp1, sp2, card):
    #如果被使用了“变压”，体力消耗*2
    return ((sp1 - sp2) ** 2 / FACTOR_SPEED ** 2) * (CARD_AMPL_PARAM if card == CARD_AMPL else 1)
```

击球函数默认打上角，当对手位于中线以上说明打下角对方消耗体力更多。应该选择下角；对于每一个可能的打角坐标，调用 cal_speed 函数分别计算打到该点时需要的速度值，存入列表 lists，再利用循环排序经过两两比较，得到击球前后速度增加值最小的打角坐标，说明此时在此击球策略下体力消耗最少，并记录下需要的速度。

```
min_change = abs(lists[0] - tb.ball['velocity'].y) #对球速列表排序，并取最小值计算加速最小值
spd = lists[0]
for item in lists:#再算一次，用最小加速和刚才的值比较，取最优情况
    change = abs(item - tb.ball['velocity'].y)
    if change < min_change:
        min_change = change
        spd = item
```

当打上角（或下角）体力消耗太多（>1250）时，则进行换角，并以同样的方式计算得到速度增加值最小的击球策略。

```
if self.cost(spd, tb.ball['velocity'].y, tb.op_side['active_card'][1]) > 1250:
    if hhit == self.up:
        hhit = self.down
    else:
        hhit = self.up
    for Loc in hhit:
        spd = self.cal_speed(tb.ball['position'], Position(tb.op_side['position'].x, Loc))
        lists.append(spd)
```

对于体力消耗较小（<250）的策略，调用 exec 函数进行记录。

```
if self.dic is not None and \
        self.cost(spd, tb.ball['velocity'].y, tb.op_side['active_card'][1]) < 250:
    exec(self.dic['read_data'])
    exec(self.dic['record_data'])

return spd
```

最后函数返回值为最优策略加速后乒乓球的速度。

2、道具函数

2.1 判断是否“吃”道具：eat_card 函数

首先与初始化有用道具列表进行对比，判断道具是否“有用”。如果不是，就不“吃”。所有的获得道具的路径可分为向上反弹一次、向上反弹两次、向下反弹一次和向下反弹两次。将每种路径所需要的加速后的速度值用 lists 列表记录。调用 cost 函数计算体力消耗值，利用循环排序找出体力消耗最小的最优策略，由于使用道具的收益约为 2000 左右，当体力消耗大于 3000 时，不“吃”道具。最终函数返回值为最优策略加速后的乒乓球速度。

```
min_cost = 3000 #物品的收益约在2000左右，消耗太大就不吃了
spd = None
for item in lists:  #找到捕捉道具的最小体力损耗
    cost = self.cost(item, tb.ball['velocity'].y, tb.op_side['active_card'][1])
    if cost < min_cost :
        min_cost = cost
        spd = item
return spd
```

2.2 判断给谁用道具：card_to 函数

如果是掉血（DECL）和变压器（AMPL），给对方使用；补血（INCL）给自己使用，而旋转（SPIN）第一次使用时给自己使用，目的是防止对手第一次使用在自己身上；从第一次以后，旋转（SPIN）使用在对手身上。

```
def card_to(self, card): #确定道具给谁用
    if card == CARD_DECL or card == CARD_AMPL:
        return 'OPNT'
    if card == CARD_DSPR or card == CARD_INCL or card == CARD_TLPT:
        return 'SELF'
    if card == CARD_SPIN and self.spin == 0: #第一次出现spin,先给自己用，想看对方是不是会默认spin给对手用，这样可以虚晃一枪
        self.spin = 1
        return 'SELF'
    if card == CARD_SPIN and self.spin == 1:
        return 'OPNT'
    return None
```

3、加速函数

创建 render 函数，策略是使用道具，调用 eat_card 函数获得最优体力消耗方案的速度值，如果使用卡片为 SPIN，则速度提升一倍。如果不使用道具，调用击球函数 hit，获得击球最佳策略的速度。

```
def render(self, tb: TableData):
    if self.dic is not None:
        exec(self.dic['read_data'])
        exec(self.dic['modify_data'])

    spd = self.eat_card(tb) #默认使用卡牌
    spin = 1.0              #默认无CARD_SPIN作用
    #如果被使用了CARD_SPIN，那将加速提高一倍以到达理想坐标
    if tb.op_side['active_card'][1] == CARD_SPIN:
        spin = 2.0
    #(默认)使用卡牌的情况
    if spd:
        return (spd - tb.ball['velocity']).y) * spin
    #如果不使用卡牌，返回hit函数决定的速度
    spd = self.hit(tb)
    return (spd - tb.ball['velocity']).y) * spin
```

4、play 函数

```
def play(tb: TableData, ds: dict) -> RacketAction:
    card = mf.use_card(tb.side["cards"])
    return RacketAction(tb.tick,   # 当前时间，照抄参数中的tb.tick即可
                        tb.ball['position'].y - tb.side['position'].y,   # 球拍的移动距离（有方向）
                        mf.render(tb),   # 球拍对球加速的速度（有方向）
                        mf.move(tb),   # 球拍击球后跑位的移动距离（有方向）
                        mf.card_to(card),   # 使用道具，对谁使用'SELF'/'OPNT'
                        card   # Card对象，使用什么道具
                        )
```

调用自定义类 render 函数进行使用道具或者不使用道具的最佳策略的选择，击球后调用 move 函数回到中点，进行下一轮击球。

2.3 程序限制

我们使用的函数中，跑位部分的代码比较粗糙，当检测到球出现的范围可能超出体力上限时会自动向球场的中间跑。这耗费了过多的体力。

3 实验结果

3.1 测试数据

1. 实验环境说明：

- 硬件配置：2.7GHz Intel Core i5/8GB 1867 MHz DDR3
- 操作系统：OS X EI Capitan 版本 10.11.2
- Python 版本：3.5

2. 测试方法和结果

利用本组算法与 T_idiot 对战，可稳定获胜。

3. 热身赛结果

热身赛中我们共进行了 170 次对战，其中 33 次胜利，胜率 19.41%。获胜的战局中有 9 次原因是对方未接到球，24 次原因是对方击出无效球；输掉的 137 个战局的失败原因均是我方未接到球。分析得知我方的击球函数较好，对速度的控制很稳定，但是跑位函数对边界状况的判断较弱，遇到对方的打角球时容易失

球。

表 3.1 UNIFORM 组参加热身赛结果统计表

结果 results	次数 times	原因 reason	次数 times
我方 West 获胜	12	对方失球 miss_ball	0
		对方无效球 invalid_bounce	12
我方 East 获胜	21	对方失球 miss_ball	9
		对方无效球 invalid_bounce	12
我方 West 失败	68	我方失球 miss_ball	68
		我方无效球 invalid_bounce	0
我方 East 失败	69	我方失球 miss_ball	69
		我方无效球 invalid_bounce	0

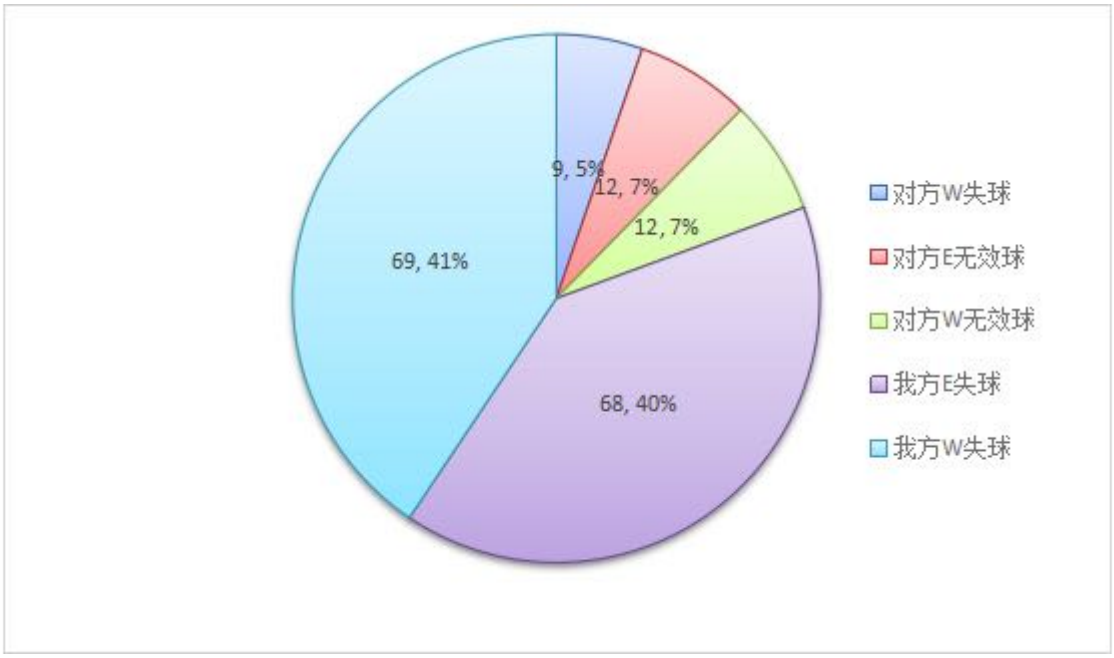


图 3.1 小组赛结果统计图

4. 选择热身赛的某场对战做展示：[W.M]T_SWIFT-VS-T_unique-001
   我方与 SWIFT 组对战，SWIFT 组获胜，原因是我方失球。

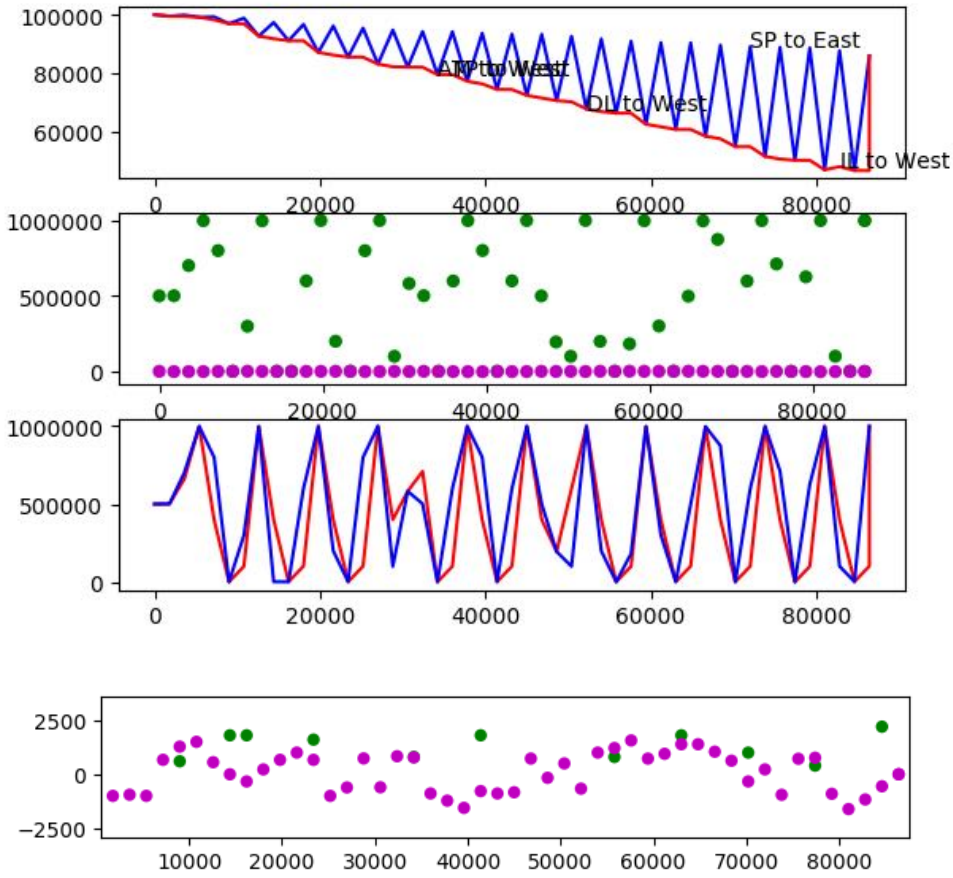


图 3.2 从上到下分别是体力图、球图、板子图和放大的球图。球图中，绿色为位置图，粉色为速度图。SWIFT 组为 West（红线），我方为 East（蓝线）。

5. 分区赛结果

北区分区赛中，我们战胜了 X-RAY 组合 TANGO 组，获得 6 分小组积分，胜率为 33.34%，排名北区第 5，未入围八强赛。

表 3.2 Uniform 组参加分区赛结果

积分	HOTEL [15]	KILO [12]	TANGO [0]	**UNIFORM [6]**	OSCAR [18]	JULIET [9]	X-RAY [3]
HOTEL	HOTEL	0	0	**0**	3	0	0
KILO	3	KILO	0	**0**	3	0	0
TANGO	3	3	TANGO	**3**	3	3	3
UNIFORM	**3**	**3**	**0**	**UNIFORM**	**3**	**3**	**0**
OSCAR	0	0	0	**0**	OSCAR	0	0
JULIET	3	3	0	**0**	3	JULIET	0
X-RAY	3	3	0	**3**	3	3	X-RAY

3.2 结果分析

1. 在测试、热身赛和实战对弈过程中，本算法的哪些策略起了何种作用？效果如何？

打角策略:由热身赛结果来看，我方胜利的 33 次中有 24 次是由于对方击出无效球，而无

效球的成因多为对方对于打角球难以应对，说明我方打角策略的实施效果较好。

博弈树策略：小组赛历次战败原因都是我方失球，即我方体力率先耗尽，说明我方的博弈树策略总是未能找到我方体力消耗较对方少的打法，该策略效果不佳。

αβ 剪枝：该策略主要目的是缩小搜索范围、节约计算成本，参考我组代码的运行时间可知该策略效果拔群。

2. 运行时间复杂度的分析详见第一部分。函数的具体分析详见第二部分。

主要的时间开销是计算体力消耗的 cost 函数，因为需要对每种可能的打角坐标计算速度值并循环排序，得到体力消耗最少的方案。eat_card 函数也需要较大时间开销，因为需要考虑所有可能获得道具的路径，并计算和循环排序每种路径所需要的加速度和体力消耗值，选出体力消耗最小的策略。

4 实习过程总结

4.1 分工与合作

Uniform 小组全部成员来自生命科学学院 13 级同学，在本次大作业期间都面临着毕业答辩的压力。为了高效利用时间，小组采用明确的分工，并且小组成员通过线上讨论（主要是微信群）和线下讨论的方式沟通各自的完成进度与成果。

小组成员通过组会和微信群的交流共同确定大致的游戏策略以及分工。具体分工如下：孟柳昳、段麟飞和陆晓雨同学主要负责完成代码，这其中孟柳昳主要负责迎球与击球、段麟飞负责道具的收集与使用、陆晓雨负责跑位以及学习等其他游戏策略代码的编写。另外四位同学：李昔筱、时雨晨、唐晨晨、郑丁源则负责小组报告的完成，李昔筱主要负责实验结果的分析和总结，时雨晨负责算法思想思想以及算法时间复杂度的总结，唐晨晨负责程序代码的各种说明解读与总结，郑丁源负责实习过程和实习经验教训的总结。

在整个完成实习的过程中，组员们始终通过微信群的讨论保持联系和工作进展的联系。线下的讨论总共进行过 3 次，并且由于实习的分工可以分成两个大组，因此代码组和报告总结组内部也进行过数次线下的讨论。



图 4.1 小组成员在讨论（孟柳昳摄）

4.2 经验与教训

整体来说我们小组最终的比赛结果只能说是中规中矩，对我们来说可能汲取的教训就是对整体游戏策略的讨论可能仍然不够深入。很明显，漂移乒乓游戏不能产生必胜策略，因此如何节省体力以及消耗尽可能多对手的体力是最优的策略。然而在最初构建代码的过程中，我们对这个小组这个策略实现的并不是非常理想。由于对对方体力消耗和己方消耗体力的预计对比需要用具体算法实现，因此如何利用算法准确的估计这两个值也是游戏获胜的关键。可以说游戏整体的策略是骨架，而代码则是血肉，将程序填充完成，两者相辅相成，关系密切。在认识到这一点后，我们小组对代码进行了改进，取得了不错的效果。

我们小组认为小组成员构建的策略以及对非完全可控因素的估计上是我们小组的亮点。

我们并没有局限于利用所有已知条件实现策略，而是敢于大胆地对对方策略也进行一定的预测和估计，因此代码实现后的竞争性大大提高。当然由于存在估计，其结果一定存在误差，在与测试机器人的的比赛中这种估计的误差无法被体现，因此代码的实战效果存在较大的不确定。对这个缺陷的改进需要我们设计更智慧的对手用于测试，当然这也将极大地增加小组的工作量。

4.3 建议与设想

这次的实习作业对我们小组的每个成员来说都非常具有挑战性，而由于作业的结果最终能以比赛的形式实现，因此实习作业也极大地调动了组内成员的积极性。这种竞赛的形式非常好，是今后实习作业应该继承的。让我们小组感到遗憾的是我们小组在小组赛就出局了。由于进入到淘汰赛的队伍都有一定的机会修改自己的代码，因此我们建议在赛程设置上可以考虑设置出线条件更为严苛的复活赛，给小组赛没有出线的其他小组修改和完善代码的机会。

我们注意到在实际比赛中同组的 Oscar 组的代码的运行时间远远高于其他组，在面对组内其他对手时也可以碾压般轻松地获得胜利。很明显，Oscar 小组的策略应当是考虑非常周全的，并且算法也较为复杂。因此以 Oscar 组为榜样，我们建议学弟学妹在进行类似的竞赛时应当充分地考虑迎战策略，并向一些其他相关专业的同学或老师咨询，完善自己的策略与代码，他们的参考意见也许将具有重大意义。

5 致谢

感谢参与本组讨论并对本组代码编写提供建议与帮助的信息科学学院 2013 级德语心同学。
感谢陈斌老师提供这样一个让我们小组发挥潜能的平台，也感谢老师对我们小组问题提供过的帮助。
感谢在班级微信群里为本组提供过帮助的各位同学。
最后感谢其他小组的同学与本组的精彩较量。

6 参考文献

http://www.cnblogs.com/mumuxinfei/p/4396339.html
http://www.cnblogs.com/mumuxinfei/p/4305981.html
http://www.cnblogs.com/mumuxinfei/p/4379595.html



X-RAY

组长：裴召文

组员：刘梓铭，孙宁远，刘坤维，姚明涛，谢东兴，杨露

小组赛战绩：N6





X-Ray

数算期末作业实习报告--XRAY 组

（裴召文*，杨露，刘梓铭，孙宁远，刘坤维，姚明涛，谢东兴）

摘要：

我们组的程序主要由迎球，道具，跑位三部分的算法组成，算法的主要的原理是基于估值计算和比较判断，在我方消耗较少的情况下增加对方的消耗，从而找出最优的加速、击打卡牌和跑位策略。

关键字：

估值；比较；判断

1 算法思想

1.1 总体思路

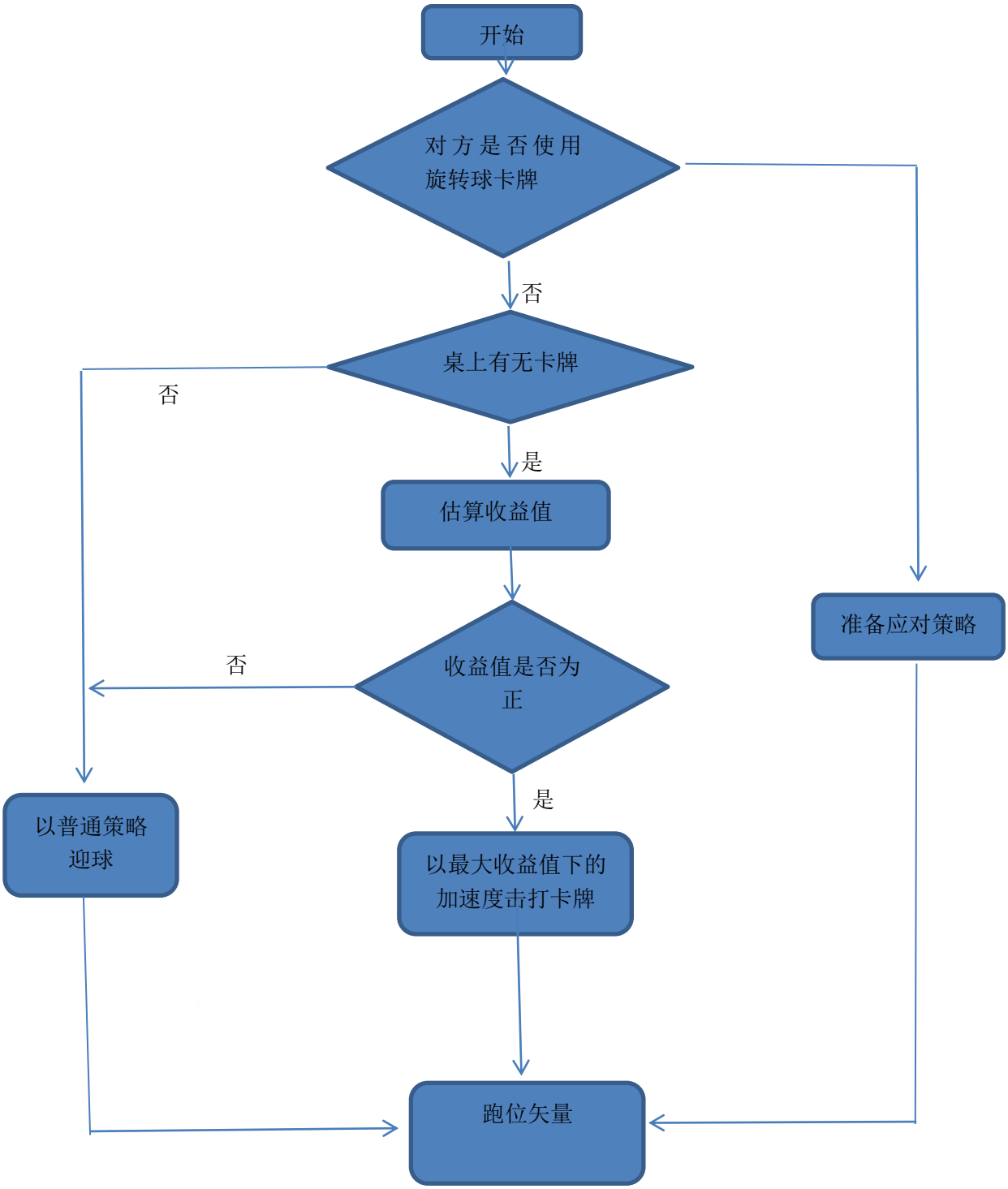
首先，在我们 XRAY 组的算法中，发球总的策略是：对于每一次击打，发球时都向对角处的方向发球。在接发球时由于对方不能跑，直接取最小值单独处理。对于迎球加速的策略，我们基于对方的位置，每次都把球击打到与对方位置相反的边角处，从而使对方在迎球时消耗较多的体力。

下面是道具的部分。首先是道具获取：游戏中的道具分为攻击型道具（旋转、变压、减血）和防守型道具（加血、瞬移）以及无用道具（隐身）。我们对每一种卡牌进的收益行了估值，对能获取道具的路径（速度）进行一定的计算，在体力值额外消耗和对方的损失不小于预期收益的情况下选择获取道具路线。然后是道具使用：防守型道具获取后立即使用，攻击型道具的使用决策在估值函数中完成，并按照一定的优先级使用。最后是道具的应对，我们主要考虑了对旋转卡牌的应对，以防止己方接球时出现 invalid_bounce 而出局。

上述的算法策略是我们在前两周的过程中逐步确立的，第一周确定并编写了发球函数部分、加速部分，第二周编写了道具部分的主要代码。关于跑位策略，开始时，我们采用每一次发球后都向中间位置跑位的原则；之后变化为基于 ds 的存储数据， 如果对方两次连续向我方同一边角打，选择不跑位；否则，向中间位置跑位。但是经过测试结果的分析，我们最终还是选择了每次跑中间的稳妥策略。

综上所述，我们组采用的策略是“暴力打角”策略，没有用到复杂的搜索算法和学习函数，因此使用的数据结构主要为列表和字典，算法就是估值与比较判断。

1.2 算法流程图



1.3 算法运行时间复杂度分析

我们组的算法的组成较为简单，没有应用复杂的数据结构，以估值计算和判断为主，主要代码的复杂度均为 O（1），还有一部分代码用到了循环语句，算法复杂度为 O（n）。总体来看，耗时极少，实际运行过程中，基本上都在 10^{-3} 秒以下。

2 程序代码说明

2.1 数据结构说明

算法中主要采用的数据结构和自定义包括字典和列表，字典中记录了对战中所需要的各种位置参数，列表中则分别存入了卡牌以及各种可能加速度。

2.2 函数说明

2.2.1 发球函数 serve

参数：ds，对方名称
返回值：球的纵坐标和加速度
功能及基本策略：使用最小的加速度，发到对方的边角处。

2.2.2 一般情况下的迎球加速度 accbase

参数：tb
返回值：球的加速度
功能和基本策略：基于对方的位置，每次都把球按照最小的消耗击打到与对方位置相反的边角处，从而使对方在迎球时消耗较多的体力。

2.2.3 击打卡牌的收益值计算函数 profit

参数：tb,card
返回值：为最优收益值及此时采用加速度
功能及基本策略：Profit 函数是当存在卡牌时，通过计算三种不同路径下（直接击打，一次反弹后击打，二次反弹后击打）击打卡牌时的加速度以及体力消耗，再结合对卡牌价值的估值、己方和对方的体力消耗，从而计算出击打卡牌的最优收益值以及此时采用加速度，采用的算法主要是估值计算以及比较判断。

2.2.4 打球函数 play

参数：tb,ds
返回值：一个 RacketAction 对象
功能及基本策略：Play 函数包含迎球策略，卡牌使用策略和跑位策略。迎球策略：调用 accbase 函数，此外还单独增加了应对接发球的情况。卡牌使用策略：则通过判断 profit 函数中的收益值正负选择是否击打卡牌，收益值为正则击打并使用卡牌，收益值为负则正常迎球，此外还增加了应对旋转球的策略。跑位策略：基于 ds 的存储数据，如果对方两次连续向我方同一边角打，选择不跑位；否则，向中间位置跑位。

2.3 程序限制

我们组的算法程序结构较为简单，在极端条件和边界条件下，程序未出现 invalid_bounce 等错误。

3 实验结果

3.1 测试数据

3.1.1 实验环境说明：

- 硬件配置：Intel(R)Core(TM)i5 - 6200U/4.00GB

- 操作系统：Windows10
- Python 版本：Python 3.5

3.1.2 测试方法：

主要有三种测试方法：

（1）主要是改进之前自己的代码（即陪练算法）进行对战；

（2）参加老师和助教组织的热身赛；

（3）和别的小组进行私下约赛。

3.1.3 测试结果：

（1）和自己之前的算法的对战：

一般每次算法改进之后均和之前的算法进行 300-400 次的对战，由于修复了之前算法的明显漏洞（下面会有说明），基本结果是可以获得全胜。

（2）热身赛：

第一次热身赛由于平台代码更新不及时，发球的 serve 函数少了一个参数，出现了 bug；第二次热身赛由于未分清 tb.op_side['active_card'][1].code 和 tb.op_side['active_card'][1]的区别，虽然在自己的电脑上跑着没问题，但是在助教的电脑上出了 bug，只获得了 15 组数据，都是失败。总结两次热身赛均出现 bug 的经验，一是没有关注平台代码的更新，及时修正自己的代码；二是代码某些具体的细节没有处理好。

（3）和别组私下约战：

首先是和室友所在的 Golf 组进行多次（百局左右）友谊赛，但每次都以失败告终！和 Remeo 组进行了一次友谊赛，可能是由于代码互相克制的原因，20 次对战取得了全胜。

测试方法	改进型对战	热身赛 1	热身赛 2	友谊赛 1（Golf）	友谊赛 2（Romeo）
数据组数	300-400	0	15	~100	20
胜负结果	全胜	bug	全败	全败	全胜

3.2 结果分析

3.2.1 基于对战结果的策略应用与改进：

（1）在与我组以前代码的对战中，我们基本修正了在复盘中出现的一些问题，比如说接球要单独处理，要应对对方使用旋转球，隐身道具和旋转球道具由于收益较小，暂时完全弃用，根据 ds 的记录进行跑位，对不跑位的打法进行克制。

（2）在热身赛中，发现本组的代码在某些情况下会出现加速消耗的生命值大于 5000 的情况，后经过检查代码，发现是绝对值处理出了偏差，没有注意负号。

（3）在和 Remeo 组的友谊赛中，我方获胜的原因主要就是，我方每次都使球达到离对方较远的角上，使对方迎球消耗的体力值较多，这说明我组的算法对对方算法起到了一定的克制作用。

（4）在和 Golf 组的比赛中，发现我组的跑位被对方克制，容易被对方来回调动，使迎球消耗的体力值较多。因此，在最后提交的比赛的代码，为了保持稳定，本组提交了始终跑位到中间的算法。

在与自己之前的陪练算法对战过程中，以及不同算法之间的对战过程中，可以发现，不同的发球策略之间有相互克制的效果，有的小组的算法是专制打角策略的，而打角策略又能克制其他算法，综合而言，打角策略可以说是一个构成简单，效果不错的“性价比高”的选

择。刚开始，纯粹的打角算法之间对弈，一般相互牵制，之后，卡牌的使用以及不同组跑位策略的改变使战局发生了变化，尤其跑位往往成为取胜的关键。关于跑位方面，我们改变纯粹的跑位中间位置策略，改为一个简易判断策略：当对方两次连续向我方同一边角打，选择不跑位；否则，向中间位置跑位。这个策略的选择使我们胜过纯粹中间跑位的算法；然而，面对学习型的跑位策略，我们的算法是无能为力的，并且会造成严重的消耗，这也使我们最终仍然提交了中间跑位的算法。

3.2.2 算法运行时间：

由于我组没有使用太复杂的结构，算法主要是以条件判断语句为主，所以算法消耗的时间较少，基本上都在 10^{-3} 秒以下。

3.3 经典战局：

在测试和比赛过程中，由于各方的基本策略多是打角，所以暂时没有特别经典的战局出现。

4 实习过程总结

4.1 分工与合作

4.1.1 小组分工：

成员名字	分工
裴召文	统筹安排，算法测试
刘坤维	部分代码编写
孙宁远	部分代码编写
杨露	报告撰写
姚明涛	提出部分算法和基本策略
谢东兴	提出部分算法和基本策略
刘梓铭	提出部分算法和基本策略

4.1.2 合作与交流:

主要方式：微信和会议

第一次组会:

时间：2017 年 5 月 22 日

地点：四教 405

主要内容：

(1) 题目分析与整理，熟悉对战平台的代码；

(2) 讨论分析发球、加速、跑位等基本策略；

(3) 初步确定小组成员的分工。



第二次组会：

时间：2017 年 5 月 24 日

地点：四教 406

主要内容：

（1）讨论修改加速及跑位策略；

（2）确定卡牌的击打、使用、应对策略。



第三次组会：

时间：2017 年 5 月 31 日

地点：四教 504

主要内容：

（1）对 ds 的使用进来讨论；

（2）讨论基于 ds 的加速策略和跑位策略。



以上三次组会是基本策略的讨论分析，此外，我们按照分工，组员之间进行了多次单独的交流。

4.2 经验与教训

在本次作业中，我们除了刚开始有了大致分工以外，剩下的每部分代码的分工都是在组会上再次分工的，虽然没有从开始就确定每个人具体要做的部分，但是这样也使大家在每一部分的策略选择以及改进过程中都认真投入，大家一起进行头脑风暴，增添了乐趣，达到了 groupwork 的效果。

但是本次工作中我们也有严重的不足之处：由于没有及时更改参数设置等原因，两次热身赛我们提交的代码都有 bug，这导致在正式赛之前我们获得的对战数据少之又少，在这种情况下，我们很难对代码进行有效的改进。唯一一点跑位策略的改变之处也因为私下对战的效果一般而被取消，事实上，之后我们想在保留判断型跑位策略的同时减少自身体力消耗，也因为时间有限而没有完成。

4.3 建议与设想

首先，在组队方面，在组员分配上还是存在着不合理之处，自由组队使高水平的“一流程序员”们聚集在一个组，组别之间实力相差较为悬殊，开放性的作业也使“抱大腿”、“外援”等现象频生。

其次，在赛制设定上，正式比赛前只有两次时间固定的热身赛，建议可以将此设计得更加灵活一些。避免出现因为没有及时更改参数设置，出现 bug 等情况。

在实际竞赛上，感觉还是可以设计得更紧凑一些，四个小时的竞赛感觉存在时间浪费，也使得过程不那么激烈。

另外，还希望可以公布前三名战队的代码，让大家都要学习的机会。

最后，想要对学弟学妹们说的是：欢迎来选陈斌老师的这门数算课，老师幽默风趣又可爱，微信群里还有红包不定时发送！

5 致谢

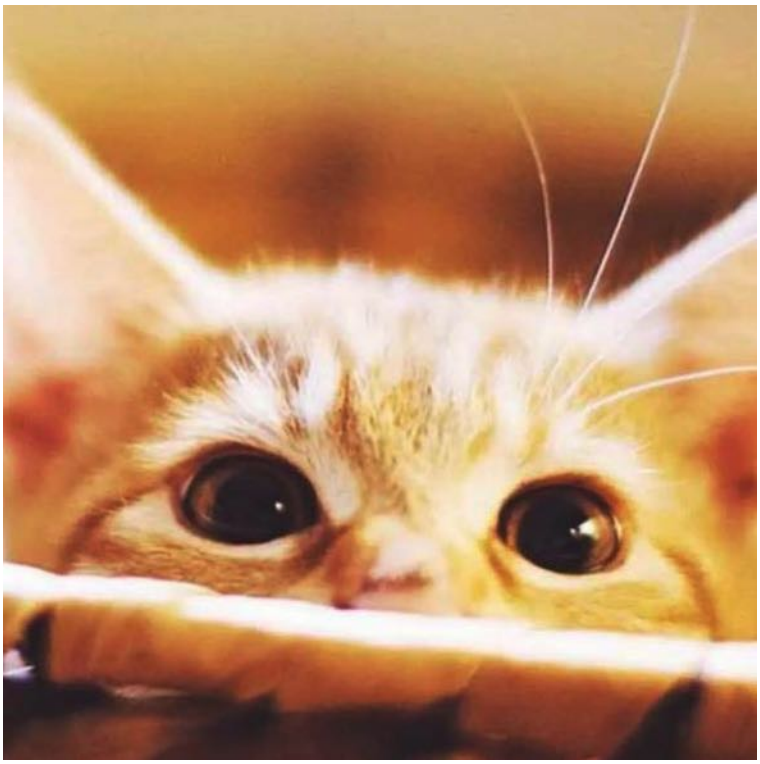
首先要感谢陈斌老师这一学期的教学和举行这次漂移乒乓的小组对战，让我们在学习课程知识的同时参与课外活动。同时，还要感谢老师和助教们对我们的问题的解答，帮助我们完善程序，对算法有了更清晰的认识。

感谢本次小组作业中每一位组员的认真努力，因为有了每一个人的参与，才有了现在的成果，虽然我们的算法算不上优秀，但是也汇聚了每个人的努力。同时感谢 Golf、Romeo 等组的支持，虽然总是被虐，但是也激励着我们做更多的改进。

6 参考文献

1 基于强化学习 TD 算法的乒乓游戏击球策略优化，陈功，周谊成，王辉，《电脑知识与技术》 , 2011 , 07 (28) :6926-6927

2 基于人工鱼群算法的乒乓球数据挖掘，孟宪明，凌培亮，《计算机工程》 , 2010 , 36 (5)



TANGO

组长：周云帆

组员：杨乐然，赵文哲，闻佳露，董子安，陈玄同，高温慧

小组赛战绩：N7

数据结构与算法课程实习作业报告

周云帆* 杨乐然 闻佳露 陈玄同 董子安 赵文哲 高温慧

摘要：主要使用贪心算法来对数据进行处理，即对每个子问题的解决方案都做出选择，把问题分为子问题，以局部最优解来替代全局最优解。实验结果在演练中 26 战 7 负，小组赛全败。

关键字：list、tuple、dictionary、greedy algorithm

ⅰ．思想

总体思路：

发球：发球位置选定中间，则对方消耗体力与我方消耗体力可以写成我方加速（v1）与对方加速（v2）的函数，取一个 v1,能使无论 v2 取何值，都能使对方消耗体力大于我方消耗体力

击球：击球过程中考虑对方消耗体力与己方消耗体力的差值的最大值，可以证明这样的情况一定在击中边角时实现，因此击球时根据对方跑位方向与其他桌面信息选择击球位置（当对方使用道具时将采用不同的防御措施）

跑位：跑位时初始设定为跑向中间，之后将根据 ds 来判断对方是否有特定的击球习惯（是否习惯于将球击向较为省力的方向）来改变跑位策略。

捡道具：是否捡拾道具将根据捡道具时双方消耗体力差与最佳方案双方消耗体力差来判断。捡道具时消耗体力差不能与最佳方案相差较大。因为已有的方案为打角，速度改变将使球的落点向中心偏离，一般情况下，对方消耗体力减少，净消耗的体力（对方消耗体力减去我方消耗体力）减少，因此这样损失的净消耗体力将近似为速差（捡道具所需的速度与原定最佳方案的速度）的函数，可以将可捡拾道具区间看作一个三角形（对于不同的道具这个三角形区间的大小将不一样（不同的道具对己方的益处将做不同的考量，主要视实验中的情况而定），若道具在在这个三角形区间，则捡拾（若有多个则判断优先级）并对击球速度作出修改。

使用道具：不同道具使用触发条件为：加血包，体力值低于 98000 则使用；**掉血包：**立即使用；变压器，计算消耗对方体力的期望值，若大于某一参数则使用；旋转球，立即使用；瞬移，若跑位距离大于一参数则使用；隐身术，若己方位置靠近中间则使用。

防御：若对方使用了旋转球或变压器，则击球速度为满足规定且所需改变速度最小的速度

算法流程图

若 ds 为空建立 ds



根据 ds 判断对方是否有极大省力位置（即是否选取较小的加速度）的倾向，判

断对方是否有跑位跑中间的习惯



计算跑位位置



计算最佳方案



判断是否捡道具



对方是否使用变压器或旋转球



判断是否使用道具



向 ds 中添加数据

算法时间复杂度分析

每次根据不同的情况（是否捡道具，对方是对否使用道具等）进行特定次数（一定范围）的计算

ii．程序代码说明

2. 1 数据结构说明

我们的核心就是贪心算法，一开始我们还设想能不能够用全局算法来进行程序的编写。后来发现太过困难，于是仍旧采用贪心。主要是根据每一步的变化来进行最优的求解，虽然精确度不是太高，但够用了。至于课程上的数据结构，我们只是稍微优化了一点，并没有进一步的扩展。

2.2 函数说明

函数 find_run:按跑中间计算，如果对方有跑中间倾向返回 True

函数 find_batting:统计对方 pos1 附近区域的频率，以调整跑位策略

函数 play:对战主程序

函数 cornor:得到打角所需的最小速度改变量及其所消耗的生命值

函数 get_min:得到三个数中间最小的

函数 use:给出使用道具的条件，在有多个道具满足条件的情况下，根据优先级判断何时使用

函数 batting:击球。经过我们组长强大的逻辑严密数学证明，在各个情况下打角总是最优我方的选择，于是我们毅然决然的使用了打角策略。为了让我们的 tangle 更加“智能”，我们还考虑到如果对方没有跑中间的习惯，我们按照对方总能采取最优跑位的情况计算，如果对方有跑中间的习惯，那么我们就采取打角策略。

函数 get_point:计算球的落点及速度

函数 isvaluable:计算是否捡道具，若在最优方案的相邻区域，则捡道具,有多个满足条件的道具，则按优先级确定

函数 count:计算碰壁次数

函数 waytosolveAM:如果对方使用变压器，则采取最省力的打角算法

2.3 程序限制

在算法的编写过程中，我们首先考虑了对方暴力打角的问题，并在算法中编入了应对暴力打角的函数，就是当对方专注于打某个角的时候，我们的跑位策略会选择待在那个角不动，这样我们便解决了暴力打角时体力消化过大的问题。但是我们在击球时选择的策略同样是打角，因为在组长周云帆的计算下，打角是最为优化的策略，也正是因为打角的策略，我们在面对道具旋转球的时候，速度会产生较大的误差，从而会出现 invalid bounce 即是击球超过两次的错误。因此只有在对方出现使用旋转球道具的时候，我们会在 1~2 回合内出现击球失误，从而输给对方。

iii. 测试数据

实验环境说明

硬件配置：GTX960m 显卡，四核 I5-6300HQ

操作系统：windows 10

Python 版本：3.5.3

iv. 实习过程总结

4.1 分工与合作

在我们小组中，周云帆，杨乐然，闻佳露，陈玄同，董子安负责程序的编写。赵文哲负责报告的撰写。大家主要还是通过组会的形式进行交流，当然在大家在网上的交流也是很重要的。





4.2 经验与教训

在实习过程中，我们觉得最重要的还是思路，如果一开始就有明确的思路无疑会对这次实习作业产生巨大的帮助。我觉得我们在这方面做的还是可圈可点的，首

先我们在第一时间就召开了第一次组会，随后在具体规则公布后又连续进行了多次，还算是比较迅速的有了自己的思路。在执行方面同样做的不错，大家都各司其职，一开始的模拟对战还算顺利，只是败给了 7 个对手。截止到这个阶段，大家都还是感觉不错的。但是后来的调试过程有点不足，在最后的对战上我们给出了一个不成熟的代码，落得惨败的结果，这应该是要引以为戒的，不要拿出不保险的方案，毕竟没有经过多次测试。所以最后结果不甚理想，大家也是心有不甘，本来有机会更进一步的。不过还是很开心，每个人在这次实习过程中都学到些东西，有的人多，有的人少，但终究是有所收获，足以。

4.3 建议与设想

在本次实习中，组队的安排有些不科学，很多“大佬”都抱团，这就有些降低比赛的竞争性了。建议可以按照历次作业的优秀程度进行分组，这样会更有看点，弱一点的同学也有机会学到更多。希望下一届的学弟学妹们可以比学长学姐们更上一层楼，用更加优化的算法解决问题，在合作中互相提高。而实习工作的后续也可以变得多样化，让对战富有更多的趣味。

v. 致谢

周云帆
杨乐然
闻佳露
陈玄同
董子安
赵文哲
高温慧
陈斌老师
还有各位助教

vi. 参考文献

Problem Solving With Algorithms And Data Structures Using Python, 2nd Edition
Learn Python The Hard Way, 3rd Edition

编后记

“树莓派、分形树、漂移乒乓算法……陈斌老师用丰富多彩的活动调动我们的积极性和创造性。其它课一般考卷交上就算结束，并不能留下什么；而在这里，我们一起写一部书，每个人都能在其中留下独特的一笔。我们欣赏百花齐放的美，也就更能意识到自己的一朵不可缺少……”

这是我写在学期总结中的一段话。的确，在拼接这些报告时我看到，很多想法重复出现，却也有不少小组有独到的闪光点。这本书中针对漂移乒乓的所有思想的集合是一个人的思维难以遍历的。

我也曾抱怨游戏规则的“不合理”，但回顾起热身赛组长发言的欢笑、正式比赛前明争暗斗的点滴，便能体会到这次活动的可贵。正因为此，我接下了编辑这本报告集的任务。

在合并各组报告时，我为每组制作了展现头像的封面；为单独提交照片的小组，制作了印在组封面背面的“照片墙”。由于各组目录大体相同，我删去了各组报告的目录。此外，我对竞赛的过程进行了整理。曾为小组编写过平台帮助文档的郭晴宇编写了规则和基础设施介绍；现任地空学生会宣传部长罗哲楷使用 Photoshop 为本书制作了主封面、三个章节封面和封底。在此感谢他们的大力支持。

我曾尝试过 Botzone 平台的 Ataxx、Reversi 两个机器博弈游戏，本学期还完成了程设的 Tetris 作业。漂移乒乓是一个机器博弈游戏，不必说重新创作一个，略加修改（比如两长边重合，使球台变为圆筒形，球拍在两底面圆周上运动），就又是一个新的游戏。面对一个机器博弈游戏，我们该采用什么方法获得它的一个相对非常好的策略，或者在不知道好的策略具体是什么的情况下编写程序，让程序总能选择好的策略？我们在思考这一问题时，也就走进了人工智能的大门。

2015 年数算黑白棋大作业是典型的棋类对弈游戏，各组算法几乎是清一色的估值函数加 Alpha-Beta 剪枝。今年则少有小组把这一问题看做有深度的搜索树，更多地通过实战摸索策略。但也有组对智能算法（如遗传算法）和机器学习（如神经网络算法）至少是作了尝试。人工智能领域其实是一片广阔的天地。

地空 16 级的代国峰同学，接触 Botzone 后对机器博弈产生兴趣，通过自己的努力把自己的 Ataxx 程序提升到全平台前三的水平。不知有多少同学今后涉猎到智能计算或机器学习领域时，会回想起地空数算的漂移乒乓大作业给了自己对这些领域的初步印象呢？

郭浩

2017 年 7 月 16 日于宿舍


PEKING UNIVERSITY
1898