SESSDSA 2018 实习作业
paper.io.sesseda 报告合集
第一册

陈斌、陈天翔、张赖和

摘要

本文档为关于 paper.io.sesseda 游戏（由 Xonic PC Game 1984 为基础进行改进的游戏; 以下称本游戏）的实习作业报告合集，全两册；本分册为第一册，内容为 SESSDSA 2018 实习作业 paper.io.sesseda 项目 (以下称本项目) 介绍和 N17 联盟各组期末报告。

本项目存储于以下 GitHub 仓库：

`https://github.com/chbpku/paper.io.sessdsa`

目录

第一部分　SESSDSA 2018 实习作业
paper.io.sesseda 项目介绍 **9**

第一章　本游戏基础设置 **13**

第二章　本游戏算法竞赛安排 **25**

第三章　核心游戏逻辑代码说明 **31**

第四章　循环赛工具说明 **41**

第五章　淘汰赛与半决赛脚本说明 **49**

第六章　可视化工具说明 **53**

第七章　数据分析用脚本说明 **63**

第八章　竞赛结果概述 **69**

第九章　特色战局选讲 **79**

第十章　结语 **91**

第二部分　N17 联盟各参赛队报告 **93**

第一章　N17_Alpha 报告 **95**

第二章　N17_Bravo 报告 **119**

第三章　N17_Charlie 报告 **149**

第四章　N17_Delta 报告 **177**

第五章　N17_Echo 报告 **199**

第六章　N17_Foxtrot 报告 **209**

第七章　N17_Golf 报告 **239**

第八章　N17_Hotel 报告 **265**

第九章　N17_India 报告 **295**

第十章　N17_Juliet 报告 **311**

第十一章　N17_Kilo 报告 **317**

第十二章　N17_Lima 报告 **347**

第十三章　N17_Mike 报告 **351**

第十四章　N17_November 报告 **357**

第十五章　N17_Oscar 报告 **373**

第十六章　N17_Papa 报告 **391**

第十七章 N17_Quebec 报告 **415**

第十八章 N17_Sierra 报告 **435**

第十九章 N17_Tango 报告 **449**

第二十章 N17_Uniform 报告 **465**

第二十一章 N17_Victor 报告 **491**

第二十二章 N17_X-ray 报告 **507**

第二十三章 N17_Yankee 报告 **539**

第二十四章 N17_Zulu 报告 **549**

第一部分

SESSDSA 2018 实习作业

paper.io.sesseda 项目介绍

SESSDSA 2018 实习作业
paper.io.sesseda 项目介绍

陈斌 *、陈天翔 **、张赖和 ***

* 北京大学地球与空间科学学院，北京大学遥感与地理信息系统研究所

** 北京大学生命科学学院

*** 北京大学数学科学学院

关键词: floodfill, matplotlib, multiprocessing, paper.io, python, tkinter

实习作业简要介绍

本部分将简要介绍 2018 春季学期北京大学地球与空间科学学院数据结构与算法课程的实习作业; 本部分的成立目的乃根据相应实习作业要求，建构合适的场地配置以供本游戏使用; 本项目被拆解成多个模块，透过引用关系以完成整体的架构设置。

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

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

本游戏简要说明

本游戏的游戏原型为 Xonic PC Game 1984; 与其他版本可能稍有不同，本游戏没有视野限制，玩家可以获取盘面上的所有信息，为完全信息决策游戏;

本游戏为回合制游戏，给定回合数和棋盘大小限制; 每局游戏由两个玩家参与，两个玩家分别控制自己的纸卷; 纸卷每回合移动 1 单位距离，并按游戏规则要求抽出纸带;

玩家可以透过控制纸卷向左转、向右转或直行等三个方向，让纸带与自己的已有区域形成新的闭合区域并填充，不断扩大领地;

游戏过程中，如果玩家的纸卷碰到边界，该玩家的纸带会断裂; 如果玩家的纸带被任何纸卷（包含自己或别人的）碰到，该玩家的纸带也会断裂;

游戏目的乃透过合法操作扩大自己的领地，使己方领地大小大于对方领地大小，或让对方的纸带断裂;

第一章　本游戏基础设置

1.1　本游戏及相关术语介绍

1.1.1　基础定义

本游戏每一局的对局结果称为一个战斗; 每局战斗由两个队参加，每队出场一个玩家或算法; 双方的目标是透过在场面上的移动以达到胜利条件; 战斗的胜负详见【本游戏基础设置 > 本游戏及相关术语介绍 > 判定: 战斗的胜负】;

1.1.2　战斗

每局战斗有两方参与，代号分别为先手方 (A) 和后手方 (B); 战斗初始阶段，系统会给定游戏总回合数、双方允许的总耗时以及以下初始属性:

1. 在地图西半部的正中间附近生成先手方纸卷，给定先手方纸卷的初始移动方向（东、南、西、北之一），并以先手方纸卷为中心生成一个 3*3 的先手方初始已有区域;

2. 在地图东半部的正中间附近生成后手方纸卷，给定后手方纸卷的初始移动方向（东、南、西、北之一），并以后手方纸卷为中心生成一个 3*3 的后手方初始已有区域;

相关数值请参见【本游戏基础设置 > 本游戏及相关术语介绍 > 棋盘】; 初始阶段结束后，战斗正式开始;

1.1.3 棋盘

1. 棋盘: 本游戏在具有边界的离散格点网上进行，默认设置下东西向格点数为 102，南北向格点数为 101，每方总可用耗时时间为 30 秒;

2. 格点属性: 每个格点有两个属性:

   - 领地属性: 先手方领地，后手方领地，无领地属性;
   - 纸带属性: 先手方纸带，后手方纸带，无纸带属性;

3. 上述' 正中间附近' 乃与正中间点距离不超过 3（8 邻域意义下），也即:

   (a) 先手方纸卷初始位置的横纵坐标范围分别为 22 至 28 与 47 至 53（两端均含，下同）内;

   (b) 后手方纸卷初始位置的横纵坐标范围分别为 73 至 79 与 47 至 53 内;

   (c) 纸卷初始位置坐标为 [i, j] 时，初始已有区域有 [(i-1) 至 (i+1), (j-1) 至 (j+1)];

   (d) 具体坐标意义可参见【本游戏基础设置 > 坐标系统】;

1.1.4 回合

1. 回合总数: 游戏的回合总数由系统给定，默认设置下为 2000 个回合;

2. 单个回合: 单个回合内，按顺序进行以下操作:

   (a) A 的一步:A 读入当前盘面和相关数据，返回左转、右转或直行的其中一个;

   (b) 系统按 A 的要求移动先手方纸卷，并进行相关判定、更新当前盘面和相关数据;

   (c) B 的一步:B 读入当前盘面和相关数据，返回左转、右转或直行的其中一个;

   (d) 系统按 B 的要求移动后手方纸卷，并进行相关判定、更新当前盘面和相关数据;

   以上步骤进行完毕后，一回合结束; 注意到每回合由两个盘面更改的要求，因此每回合将有两次更新数据; 但每回合之中，每个玩家仅能移动一次;

1.1.5 判定: 战斗的胜负

- 下列情况发生时，游戏立即结束，并对操作方判负，另一方判胜:
  - 己方纸卷超出棋盘;
  - 己方纸卷碰到己方纸带;
  - 己方纸卷在对方领地碰到对方纸卷;
  - 己方算法给出不合法操作;
  - 己方算法给出操作后可用耗时为负;
- 下列情况发生时，游戏立即结束，并对操作方判胜，另一方判负:
  - 侧碰: 己方纸卷在无领地属性格点碰触对方纸卷，且双方纸卷运动方向垂直;
  - 己方纸卷碰到对方纸带;
  - 己方纸卷在己方领地碰到对方纸卷;
- 下列情况发生时，游戏立即结束，并统计双方的领地大小，其中纸卷或纸带不计; 若双方领地大小不同，判领地大者为胜方，领地小者为负方; 否则判双方平手:
  - 对碰: 己方纸卷在无领地属性格点碰触对方纸卷，且双方纸卷运动方向相反;
  - 回合数耗尽;

1.1.6 判定: 标记和圈地

纸卷运动到每个格点时，若没有达成战斗胜负的条件，则进行以下判定:

1. 如果该格点的属性不是己方领地，则对该格点附加标记为己方纸带;
2. 如果该格点的属性是是己方领地，则按下列顺序进行圈地操作:
   (a) 首先将标记为己方纸带的格点的领地属性更改为自己的领地属性;
   (b) 将标记为己方纸带的格点的纸带属性移除;
   (c) 考察己方纸带与己方领地外缘所围成的区域（以闭包形式记录），将这些区域的所有格点的领地属性更改为自己的领地属性，完成圈地操作;

1.2 战斗的运行

1.2.1 系统设施组成

- 'match_core.py': 游戏执行逻辑; 战斗运行时调用的档案; 主管战斗的进行和判定规则;
- 'match_interface.py': 游戏界面辅助执行逻辑; 保存战斗过程等调用的档案; 主管战斗的记录;
- 'roundRobinWithScoreboard.py': 进行循环赛时使用的档案; 引用'match_core.py' 和各玩家的 play 函数使之进行游戏，并在两个队伍比赛完毕以后实时输出对局结果与记分板，并引用'match_interface.py' 在后台输出对局记录文件;
- 'roundRobin_multitask_database.py': 基于'roundRobinWithScoreboard.py' 进行改善的循环赛执行逻辑; 能利用服务器多线程的优势执行游戏;
- 'visualize.py':
  - 控制台复盘代码:**可读取*.zlog* 档案;** 将对局过程和结果进行可视化;
  - 单局战斗播放: 引用'match_core.py' 和各玩家的 play 函数使之进行游戏，游戏过程实时播放，并引用'match_interface.py' 在后台输出对局记录文件;
- 'compKnockoutScenario.py': 一对一连续多局战斗使用的脚本; 引用'match_core.py' 和各玩家的 play 函数使之进行游戏，并在每局结束时输出该局对局主要结果，并引用'match_interface.py' 在后台输出对局记录文件;
- 'compF17Quarterfinal.py' 等四个脚本:F17 / N17 八强淘汰赛/ 半决赛使用的脚本; 引用'compKnockoutScenario.py' 运行;
- 系统另有以下附加功能:
  - 'extract.py': 辅助打包.zlog 竞赛过程文件的脚本
  - 'bars.py' 与'dots.py' : 数据分析绘图使用的脚本
  - 'UPDATE_TOOL.py': 可连接 github 仓库;
  - 'glory_of_mankind.py': 可实行人机对战等附加功能;

1.2.2 参战方

各参战方提供一个参战函数; 该函数应有如下格式，具体要求见【本游戏基础设置 > 参赛代码要求相关】:

```
def play(stat, storage):
    pass # 此处编写你的函数

def load(stat, storage):
    pass # 此处编写你的函数

def summary(match_result, storage):
    pass # 此处编写你的函数

def init(storage):
    pass # 此处编写你的函数

def summaryall(storage):
    pass # 此处编写你的函数
```

1.2.3 对战平台运行方式

请按以下要求顺序执行:

1. 确认上述代码文件与存取参赛队伍的文件夹在同一目录且平级;
2. 将要参战的函数按要求存成.py 文件，并这些文件放入存取参赛队伍的文件夹;
3. 按实际需求运行'roundRobinWithScoreboard.py' 等档案的其中一或多个;
4. 附加功能 - 单局战斗回放: 开启'visualize.py' 并选取对应战局的.zlog 文件;
5. 附加功能如下 - 简易数据分析: 将'bars.py' 与'dots.py' 移入与对应竞赛结果的 log 文件夹同级目录后直接运行可得对应图表;

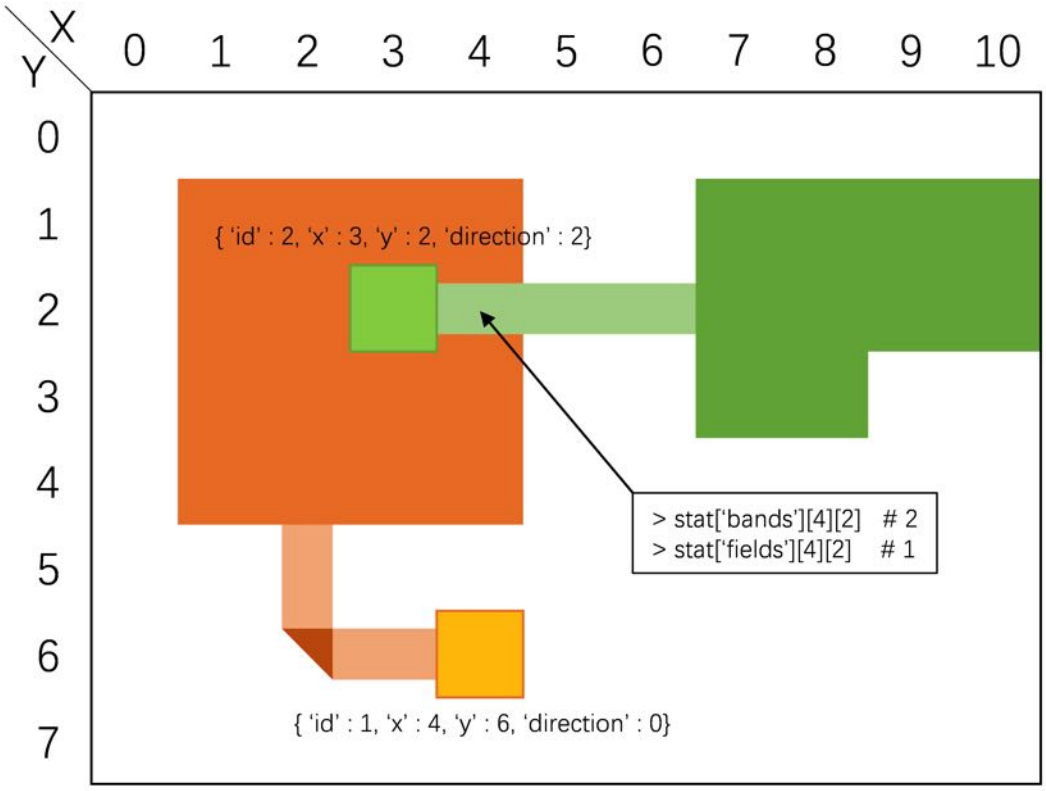
1.3 坐标系统

游戏数据中返回的横纵坐标均为整数，表示游戏地图的网格位置: 其中 x 坐标范围为 [0, 场地宽 -1]，y 坐标范围为 [0, 场地高 -1];

本文档中以东南西北作为绝对方向的指代，呈现为图形界面时，东南西北分别对应图形的右下左上四个方向; 其中 x 坐标增加的方向为东，y 坐标增加的方向为南;

fields 与 bands 二维列表中内容按列存储; 调用时第一个下标为横坐标，第二个下标为纵坐标;

下图为一个高 8 宽 11 的场地所指示的示意图:



1.4 对局结果 match_result['result']

长度为 2 或 3 的元组，记录了本次对局的结果; 其中当且仅当对局结束情况为需要统计双方领地的情况时（也即纸卷在无领地属性格点发生正碰，或回合数耗尽时）本元组长度为 3:

- match_result['result'][0]: 胜者编号:

  0: 先手玩家胜; 1: 后手玩家胜; None: 平局;

- match_result['result'][1]: 胜负原因编号:

  0 – WAL – 撞墙
  1 – TAP – 纸带碰撞
  2 – SID – 侧碰
  3 – FAC – 正碰，结算得分
  4 – CIT – 领地内互相碰撞
  −3 – END – 回合数耗尽，结算得分
  −2 – OVT – 超时
  −1 – ERR – 函数报错AI

- match_result['result'][2]: 双方领地大小;

  – 为长度 2 的元组;
  – 前一个元素 match_result['result'][2][0] 为先手方的领地大小;
  – 后一个元素 match_result['result'][2][1] 为后手方的领地大小;

1.5 参赛代码要求相关

1.5.1 代码编写的主要规范

- 编写的代码应能以 python 3.5.2 形式储存为.py 文件;
- 函数对外接口应命名为 play;
- 代码文件中应只包含 play 函数等函数，其中 play 为必需函数，其余为可选函数; 其余一切代码（不含注释部分）均应包含在这些函数内，包含导入库的语句;
- 比赛程序将按语句 action = play(stat, storage) 调用 play 函数，其中 action 为战斗中系统识别的方向变更要求，应等于参赛算法函数给出的返回值; 具体形态将于本章其它部分说明;
- 为维赛场整洁，禁用全局变量语句与 os, sys, threading, multiprocessing 等标准库;
- 不建议使用第三方库编写算法代码; 此举可能被视为 ImportError 而判负;
- 系统设施传递的参数均为标准 python 对象;
- 建议编写代码时有规范的格式和合理的注释，以便于小组的报告撰写;
- 对规则的最终解释权归本游戏技术组所有; 违规代码可能被取消参赛资格;

1.5.2 play 函数

编写的 AI 函数对外的接口应命名为 play，函数的接收参数包含游戏信息 stat 与存储对象 storage ，返回值为任意结果;

比赛系统将按 action = play(stat, storage) 形式调用，在每一回合开始时调用执行对应的 play 函数，决定函数控制的玩家方向的改变（左转、右转、不改变方向），随后朝改变后的方向前进一单位的距离;

若返回值为非空字符串且首字母大写为 L，对应左转操作; 若返回值为非空字符串且首字母大写为 R，对应右转操作; 若不符合以上两点，视为对应直行操作; 比赛核心逻辑中使用以下代码执行函数返回值操作:

```
action = play(stat, storage)
...
if isinstance(action, str) and len(action) > 0:
    op = action[0].upper()
if op == 'L':
    plr.turn_left()
elif op == 'R':
    plr.turn_right()
```

以下给出部分函数返回值与操作的相关对应和例子:

- 识别为“左转”: 函数返回'l', 'L', 'Left', 'LEFT', 'Legendary' ...
- 识别为“右转”: 函数返回'r', 'R', 'right', 'RIGHT', 'Robust' ...
- 识别为“前进”: 函数返回 None, 'iwannawin', [1, 2, 3], '0', 0 ...

1.5.3 load 函数（可选）

单局比赛开始前接收游戏信息 stat 与存储对象 storage;

可在此进行函数所需变量的声明与引用等; 本函数的用时计入玩家总耗时;

1.5.4 summary 函数（可选）

单局对局总结函数，每局比赛结束后运行一次;

可将总结内容记录于 storage['memory'] 关键字的字典中，内容将会在不同比赛间保留，比赛运行方保证对双方 storage 字典合乎逻辑的维护;

1.5.5 init 函数（可选）与 summaryall 函数（可选）

分别在与同一对手的多局比赛开始前与结束时各运行一次;

init 可用于初始化存储空间;summaryall 可用于对多局比赛进行整体总结;

1.5.6 游戏数据 stat

类型为字典，包含当前游戏状态信息关键字内容; 以下所提及的任何关键字 kw，对应的数据访问方式为 stat['kw']，部分内容详见稍后说明:

- size: 列表，包含游戏场景宽高
  类型:list[int]
  内容:[场地宽, 场地高]
- log: 运行记录
  类型:list[dict]
  内容: 对局开始以来每一回合的场地信息
- now: 当前场地信息（=log[-1]）

1.5.7 游戏数据（场地信息）

类型为字典，包含当前游戏状态信息，部分内容详见稍后说明:

- turnleft: 剩余回合数，按先后手排序
  类型:list[int]
  内容:[先手玩家剩余回合数, 后手玩家剩余回合数]
- timeleft: 双方剩余总思考时间（秒），按先后手排序
  类型:list[float]
  内容:[先手玩家剩余时间, 后手玩家剩余时间]
- fields: 纸片场地二维列表
  类型:list[list[int]]
  内容:fields[x][y] 返回坐标 (x, y) 点纸片领地归属，1 代表先手玩家，2 代表后手玩家，None 代表无纸片覆盖; 坐标含义详见【本游戏基础设置 > 坐标系统】部分
- bands: 纸带场地二维列表
  类型:list[list[int]]
  内容:bands[x][y] 返回坐标 (x, y) 点纸带领地归属，1 代表先手玩家，2 代表后手玩家，None 代表无纸带覆盖; 坐标含义详见【本游戏基础设置 > 坐标系统】部分

- players: 玩家信息列表，包含双方玩家信息，按先后手排序
  类型:list[dict]
  内容:[me, enemy]

- me: 自己控制的玩家信息

- enemy: 对手玩家信息

- band_route: 双方纸带行进方向
  类型:list[tuple]
  内容:[me 的纸带方向, enemy 的纸带方向]

1.5.8 游戏数据（玩家信息部分）

play 函数的游戏数据（玩家信息部分）包含当前游戏状态中一个玩家的状态，其中有:

- id: 玩家标记
  类型:int
  内容:1: 先手玩家;2: 后手玩家

- x, y:
  类型:int, int
  内容: 横、纵坐标; 详见【本游戏基础设置 > 坐标系统】

- direction: 数字标记的当前方向
  类型:int
  内容:0: 向东;1: 向南;2: 向西;3: 向北; 详见【本游戏基础设置 > 坐标系统】

1.5.9 存储对象 storage

存储对象 storage 呈现为字典形式，可供各函数自由使用与存储数据; 写入 storage 字典中的内容会一直保存至与同一对手的多局比赛与相关函数全部运行结束;

1.5.10 小组算法开发指南

系统将提供数个算法档案，可供参考和测试;

第二章　本游戏算法竞赛安排

2.1　算法竞赛规则

1. 竞赛项目:paper.io.sesseda;
2. 竞赛目标: 各参赛队提交算法，令算法指挥己方纸卷，使算法能自主改变纸卷在棋盘上的运动方向，利用棋盘态势信息计算纸卷走向，使其能透过圈到更多领地、碰断对方纸带等方式以取胜;
3. 竞赛场地:DELL Precision 5820 图形工作站，Xeon W-2135 CPU / 32GB RAM / 512GB SSD / NVIDIA GeForce GTX 1080 GPU; Ubuntu 16.04;
4. 竞赛环境: python 3.5.2;
5. 符合对决条件的两队称为一个竞赛对; 除友谊赛、冠军赛和季军赛等有附加说明的部分之外，竞赛对中将**进行至多 *20* 局 *GAME***，其中前十局由竞赛对的第一队为先手，第二队为后手，第十一局起先后手互换;
6. 每局战斗开始时均采用游戏的默认设置，也即赋予该局战斗设置与初始状态如下;
   (a) 每个局战斗有 2000 回合（最大可行动步数为双方各 2000，共 4000 步）; 各局战斗各队算法有总可用思考时间 30 秒（仅计入自身 play 与 load 函数的计算时间，可依需求分配到各步，2000 步共享）; 棋盘宽 102 高 101;
   (b) 先手方纸卷初始位置的横纵坐标范围分别为 22 至 28（两端均含，下同）与 47 至 53 内; 后手方纸卷初始位置的横纵坐标范围分别为 73 至 79 与 47 至 53 内;
   (c) 纸卷初始位置坐标为 [i, j] 时，初始已有区域有 [(i-1) 至 (i+1), (j-1) 至 (j+1)];
   (d) 双方纸卷的初始移动方向随机;

7. 每局战斗可能为一方胜利或平局，记入该竞赛对的总对决比分，竞赛对的胜负由该竞赛对的总对决比分唯一决定;

8. 若一竞赛对的 20 局战斗尚未全部进行完毕，而有其中一队累计胜出 11 局战斗时，此竞赛对剩余战斗将不执行，直接宣告该竞赛对由累计胜出 11 局战斗的队伍胜出;

9. 对规则的最终解释权归本游戏技术组所有;

2.2 赛前准备

- 人员:
  - 赛程技术 2 人（陈天翔/张赖和）;
  - 计分 2 人（张颢丹/滕沅建）同时负责发布截图到群里;
  - 解说 1 人（志愿者 1）;
  - 媒体 2 人（易超/志愿者 1）竞猜;
  - 拍照 1 人（冀锐）;
- 物料
  - 各组自备 A4 纸的组名牌子;
  - 投影双屏切换器和 HDMI 线;
  - 大大卷泡泡糖（100 个）;
  - 奖品水晶三维树内雕;
- 系统设置:
  - 建构文件夹，结构如下图:
    * root -> F17 -> E/W/S/N/E1W2/S1N2/E2W1/S2N1/E1W2S1N2/E2W1S2N1
    * root -> N17 -> E/W/S/N/E1W2/S1N2/E2W1/S2N1/E1W2S1N2/E2W1S2N1
  - 将各队代码放入对应的小组文件夹中

2.3 竞赛章程

1. 活动开始时间:2018 年 6 月 12 日下午三时十分（数算课上）

2. 开场

   - 老师介绍统计情况，竞赛规则;
   - 竞猜每个联盟的八强，其中猜中 5 个以上者分享 10 个大大卷，6 个以上者得到 1 个大大卷 + 分享 3 个纪念品（凭提交后的截图领奖）; 竞猜提交表单到循环赛正式开始时截止;
   - 技术组展示几个统计图表;
   - 计分组展示赛程计分牌;

3. 第一轮: 区内循环赛

   - N17 进行 N/E 两个区的循环赛; 计分组给出 N17-N1（N17 联盟东区第一名，以此类推）/N2/E1/E2; 解说同步采访出线组（下略此叙述）;
   - F17 进行 N/E 两个区的循环赛; 计分组给出 F17-N1/N2/E1/E2;
   - N17 进行 W/S 两个区的循环赛; 计分组给出 N17-W1/W2/S1/S2;
   - F17 进行 W/S 两个区的循环赛; 计分组给出 F17-W1/W2/S1/S2;
   - 计分组宣布两个联盟的八强结果;
   - 竞猜冠军亚军季军，猜中者得到 1 个大大卷 + 分享 3 个纪念品（凭提交后的截图和校园卡领奖），竞猜提交表单到八强淘汰赛正式开始时截止;
   - 进行技术分析（分 N17/F17 联盟: 终局面积 -步数散点图），部分战局演示;

4. 第二轮: 八强淘汰赛

   - 向各队取代码，放入对应的赛区文件夹;
   - 进行 N17 淘汰赛:
     - 赛局 E1W2: 东区第 1 名 vs 西区第 2 名
     - 赛局 E2W1: 东区第 2 名 vs 西区第 1 名
     - 赛局 S1N2: 南区第 1 名 vs 北区第 2 名
     - 赛局 S2N1: 南区第 2 名 vs 北区第 1 名

- 进行 F17 淘汰赛:
  - 赛局 E1W2: 东区第 1 名 vs 西区第 2 名
  - 赛局 E2W1: 东区第 2 名 vs 西区第 1 名
  - 赛局 S1N2: 南区第 1 名 vs 北区第 2 名
  - 赛局 S2N1: 南区第 2 名 vs 北区第 1 名
- 计分组宣布两个联盟的四强结果;
- 进行技术分析，部分激烈战局的演示;

5. 第三轮: 二分之一决赛

- 向各队取代码，放入对应的赛区文件夹;
- 进行 N17 二分之一决赛:
  - 赛局 E1W2S1N2: 赛局 E1W2 胜者 vs 赛局 S1N2 胜者
  - 赛局 E2W1S2N1: 赛局 E2W1 胜者 vs 赛局 S2N1 胜者
- 进行 F17 二分之一决赛:
  - 赛局 E1W2S1N2: 赛局 E1W2 胜者 vs 赛局 S1N2 胜者
  - 赛局 E2W1S2N1: 赛局 E2W1 胜者 vs 赛局 S2N1 胜者
- 计分组宣布两个联盟的冠军赛参赛者与季军赛参赛者;

6. 第四轮: 冠军赛和季军赛

- 向各队取代码，放入对应的赛区文件夹;
- 决出 N17 的第三名;（直播 3 局 2 胜）
- 决出 F17 的第三名;（直播 3 局 2 胜）
- 决出 N17 的冠亚军;（直播 3 局 2 胜）
- 决出 F17 的冠亚军;（直播 3 局 2 胜）
- 计分组宣布两个联盟的各赛局结果并由主办方颁奖;

7. 附加项目: 友谊赛

- 两个联盟冠军的友谊赛;（直播 3 局 2 胜）;
- 人类荣誉捍卫战 *5

2.4 赛制安排细节

- 区内循环赛中，每参赛对按对战结果给区赛积分，其中胜者 3 分，平局各 1 分，其余 0 分; 每区取积分前两名出线，是为八强; 若遇积分相同情况，以局均圈地大小多者位列高名次; 若仍无法比较，则进行加赛，其中加赛为骤死赛，先后手由系统决定;
- 第二轮及以后所有项目都不接受和局，遇和局情况则进行加赛;

第三章　核心游戏逻辑代码说明

3.1　核心游戏逻辑代码概述

match_core.py 是纸带圈地比赛的核心游戏逻辑代码，负责按照指定的比赛场地参数，接收双方指令进行对抗，并将比赛结果记录并输出;

模块主要对外开放的功能接口为 match 函数与 match_with_log 函数:

- match 函数输入双方玩家代码与比赛参数后会自动发起一场对抗，向双方函数传递参数，接收其返回值并更新比赛场地，并在结束后输出比赛记录对象;
- match_with_log 函数在返回对象的同时自动将其输出为导入的规范; 编写的玩家 AI 代码应包括必要的 play 函数及可选的其他函数; 导入的函数在执行时会挂载异常处理与超时处理系统，对于行为异常的代码将以比赛结果判负的形式显示。

3.2　总览

核心代码的逻辑主要可以分为框架逻辑、游戏执行逻辑与附加的计时模块; 在调用执行接口进行比赛时，各部分的职责如下:

1. 框架逻辑负责运行数据的传递与整合;
2. 游戏执行逻辑负责具体的游戏规则;
3. 计时模块评估玩家 AI 函数的功能。

之后将逐个介绍各部分所属函数的内容与实现细节; 各功能块和其所属的主要函数、类之间的调用关系如下图:

3.3 全局变量

核心代码使用以下全局变量存储游戏状态:

3.3.1 比赛场地参数

变量名	默认值	主要内容及意义
MAX_TURNS	2000	单局战斗单方最大步数
MAX_TIME	30	单局战斗单方总思考秒数限制
TURNS	[2000, 2000]	记录每个玩家剩余回合数
TIME	[30, 30]	记录每个玩家剩余思考时间
WIDTH	102	单局战斗场地宽度
HEIGHT	101	单局战斗场地高度
BANDS	[[None]*101 for i in range(102)]	记录格点的纸带属性
FIELDS	[[None]*101 for i in range(102)]	记录格点的领地属性
PLAYERS	[None, None]	双方玩家对象列表

3.3.2 参赛函数存储空间与跨比赛记忆

变量名	默认值	主要内容及意义
STAT	None	双方游戏信息
STORAGE	[], []	双方函数存储，可自由使用

3.3.3 其他

变量名	默认值	主要内容及意义
LOG_PUBLIC	None	全局比赛记录列表
FRAME_FUNC	lambda storage: None	逐帧处理函数接口
match.DEBUG_TRACEBACK	None	函数，用于报错输出存储变量

3.4 框架逻辑

框架逻辑负责初始化全局比赛环境，交替向计时模块传递双方函数，以及向双方 AI 代码及数个外界接口的 IO 功能; 其中 match 与 match_with_log 函数为整个模块的入口。

3.4.1 框架外部接口

框架函数具有以下形式:

match(name1, plr1, name2, plr2, k, h, max_turn, max_time)

此为整合的比赛函数，输入双方代码信息并设定比赛场地参数后，其将自动初始化并执行一场比赛; 参数意义如下:

- name1, name2: 先后手玩家分别的名称;
- plr1, plr2: 先后手玩家 AI 代码分别导入成的模块，包含 play 等函数;
- k, h: 控制比赛场地大小，生成的比赛场地网格长宽分别为 2k, h;
- max_turn: 比赛最大回合数，两方分别执行该数量回合后将结束比赛结算得分;
- max_time: 比赛最大思考时间（秒），任一方 AI 函数运行总时长超过该数则判负;

match 函数首先根据比赛参数初始化比赛环境，即对应的全局变量，此过程会调用 init_field 函数初始化比赛场地与玩家并初始化双方使用的存储空间，其中 memory 关键字传入全局保存的字典引用;

随后传入双方 AI 模块信息调用 parse_match 函数进行比赛，并返回记录比赛结果的元组，并在双方未直接分胜负的情形下调用 count_score 函数统计得分以判胜负; 最后调用双方的 summary 函数（如果存在），并将本次比赛结果记录打包输出为字典。

3.4.2 框架内部函数

init_field(k, h, max_turn, max_time)

初始化比赛空间参数全局变量; 将玩家对象 PLAYERS 中两个对象分别置于比赛场地左右部分正中附近的随机位置。

field_copy()

创建 FIELDS 与 BANDS 的元组拷贝，每回合执行一次

get_params(fields, bands, curr_plr=None)

从全局变量获取当前比赛状态，打包为字典传出。字典内容包含剩余回合数、剩余时间，并从参数输入获取纸片场地列表与纸带场地列表的深拷贝。

可选参数为玩家编号（0 或 1）时会额外返回标记每个玩家“自己”与“对手”的玩家信息引用，返回值将用于玩家 AI 读取与记录比赛状态; 否则会额外返回记录双方玩家纸带路径的数组，返回值将保存于比赛记录对象中，额外信息可用于可视化等。

parse_match(funcs)

在已初始化的场地中进行一次比赛，并返回标记比赛结果的元组。输入参数为长度为 2 的列表，其中两个元素分别为先后手玩家代码模块。

系统首先执行双方玩家模块的 load 函数，向其传递初始的 storage，若没有声明 load 函数则跳过该步，在该步报错或超时则直接判负。调用方式为作为参数传递给计时模块。

开始比赛后，比赛逻辑会循环指定的回合数次，轮流向计时模块传递先后手玩家的 play 函数及参数，并对其计时，在超时或报错时直接判负; 随后根据函数返回的操作符，决

定玩家 AI 代码控制的 player 对象是否左/右转，并调用 forward 函数使其前进一步。若该调用返回值为比赛结果（元组，不为 None）则直接返回该结果，否则向双方存储空间和公用的比赛记录中分别添加当前比赛场地状态，并继续循环; 循环结束后，返回“回合数耗尽”的比赛结果，待由外层 match 函数统计得分。

count_score()

统计得分函数，遍历 FIELDS 二维列表并统计双方玩家 id 出现次数，输出统计结果元组，分别为先手玩家得分、后手玩家得分。

3.5 游戏执行逻辑 – player 类

player 类集成了纸带圈地游戏的逻辑，通过数个代表玩家操作的函数连接了玩家 AI 的输出与比赛场地全局变量。

3.5.1 类变量

变量名	功能
directions	静态类变量 =[(1, 0), (0, 1), (-1, 0), (0, -1)]; 指示方向对应坐标变换
id	玩家标记，先手玩家为 1，后手玩家为 2
x, y	玩家在场地内坐标
direction	玩家面对方向
band_direction	纸带轨迹列表，记录纸带生成以来每格对应的玩家方向
field_border	列表，记录玩家到达过的区域的坐标范围

3.5.2 圈地原理

在代码实现中，领地与纸带的占地分别由 FIELDS 与 BANDS 表示，算法涉及到对二维列表进行遍历的 $O(n^2)$ 复杂度操作。为简化算法，增加性能，有必要对其进行优化:

A. 纸带 -领地转换:

每个玩家对象维护一个 band_direction 列表，包含生成纸带每一步时玩家的前进方向（0-4）。当玩家在自己领地内，无纸带时，该数组对应为空; 每当玩家前进一格走到领地外以后，铺设纸带的过程不仅会将 BANDS 列表中玩家当前坐标元素设置成玩家对应标记，还会向 band_direction 列表中加入当前玩家方向。

在玩家重新回到自己领地，将所有纸带占地转换为领地时，首先设置一个坐标 ptr 指向玩家这回合前进方向的上一步（这一步未添加新纸带），同时不断弹出 band_direction 列表末尾，并将 ptr 向每个元素对应方向的反方向不断移动 1 格;ptr 经过路径上的所有坐标都将移除 BANDS 数组内对应内容，并在 FIELDS 数组添加该玩家标记。

该算法将 $O(n^2)$ 的查询纸带占地操作简化为了 O(n) 的回溯操作，且在记录比赛信息时可以额外返回纸带的方向信息，便于可视化时对纸带方向进行显示。

B. 领地填色:

填色过程使用 floodfill 算法，并对其进行了范围优化与 python 化改进。floodfill 算法被使用于对一块封闭区域进行上色，本质为按次序遍历并标记图的节点并在到达边界时自动停止。在该填色情景下，需要填色的对象为非自己领地的格点，边界分为两种: 已有的自己领地格点与场地边框。所有与场地边框不相邻的区域均视为内部区域，需要进行上色。

注意到双方玩家互动涉及到的区域一般在很小的区域内，对整个比赛场地执行算法会有大量不必要的时间开销。为减少每次填色需要遍历的格点个数，以玩家为单位维护 field_border 列表 [x_min, x_max, y_min, y_max]，记录玩家本局到达过的横纵坐标的极值; 每当玩家坐标更新时，可能的影响范围扩大，该列表的值进行相应更新。

为忽略边界条件检查、减少额外的变量维护，填色操作的具体实现利用了 python 的数据结构进行简化，其大体流程如下:

1. 建立名为 targets 的 set，其初值为遍历 field_border 范围内所有 FIELDS 中该位置不为自己领地的坐标，包含了所有需要检查的坐标值
2. 在 targets 不为空时重复执行:
   (a) 建立名为 iter 的列表，包含 targets 中弹出的一个坐标，用于迭代一次 floodfill

(b) 建立名为 fill_pool 的空列表，记录该次填色需要上色的所有坐标

(c) 逻辑值 in_bound 设为 True，用于标记这次上色是否为内部区域

(d) 在 iter 不为空时重复执行:

i. iter 中弹出一个坐标记为 curr

ii. 在 curr 周围 4 个方向相邻的坐标内，如果 targets 包含该坐标，则将该坐标从 targets 中移除并添加至 iter

iii. in_bound 为 True 时: 如果 curr 落在 field_border 上，将 in_bound 设为 False; 否则将 fill_pool 中加入 curr

(e) 如果 in_bound 为真，则将 fill_pool 中所有坐标填充为当前玩家 id

其中涉及到的数据结构操作（除迭代式建立初始 set 外），包含 set 移除元素，list 在末尾添加、删除元素等，均为常数量级的操作; 总体访问的单元数不大于 5 倍 field_border 面积，复杂度为 $O(n^2)$。

3.5.3 类函数

注:player 类内无静态函数，且除初始化函数外均不接收外部参数; 以下内容省略 self

___init___(id, x, y, expand=1, init_direction=None)

在比赛场地内放置玩家对象并在 FIELDS 中覆盖对应位置的领地; 参数:

- id: 玩家编号（先手玩家为 1，后手玩家为 2）
- x, y: 初始坐标
- expand: 初始领地从玩家位置扩展距离，默认为 1，对应中心与玩家对齐的 3*3 领地
- init_direction: 初始方向，默认为 None 时初始方向随机

turn_left 与 turn_right

转向函数接口; 左转时方向设为 (当前方向 -1)%4; 右转时为 (当前方向 +1)%4;

forward

前进一格，并执行游戏主体逻辑。若触发游戏结束条件，该函数会返回一个元组; 否则返回值为 None; 比赛函数调用该函数时将捕捉返回值并决定是否结束比赛。内容:

玩家首先调用 directions[direction] 向当前方向前进一格，若前进后坐标超出场地边缘则返回对手获胜结果; 随后以当前坐标更新 field_border 记录的边界。

第二步是判断玩家间相撞的情况，在全局以 PLAYERS[2 - id] 的方式引用对手玩家。在对手玩家和自己的坐标重合时调用一系列判断返回相应游戏结果，其中自己领地外的情况需先调用 extend_band 更新纸带，保持纸带终点始终为自己的坐标。

随后，检查 BANDS 列表中当前位置是否为 None，若包含纸带，首先调用 update_field 函数更新场地，随后返回该纸带主人负的结果。

其余情况下调用 update_field 函数更新场地。

update_field

更新玩家在场地引起的效应。首先判断当前位置是否为自己领地: 若是自己领地，判断 band_direction 列表是否为空，并在其不为空的时候调用 check_field_fill 将纸带转换为领地，并进行填色; 否则调用 extend_band 函数延长纸带。

extend_band

延长纸带函数，将当前方向加入 band_direction 列表末尾，并将 BANDS 列表内当前坐标设置为自己的 id

check_field_fill

由领地外返回领地时执行的填色操作，分为“纸带转换为领地”与“填色”两步，详见“算法原理”部分说明。

get_info

返回玩家当前信息的字典，由 get_params 函数调用

3.6 计时模块

对于需要计时的玩家 AI 函数，框架逻辑中将函数与函数需要读取的参数统一传递给计时模块进行运行并计时。若运行中出现错误，计时模块能将异常捕获并输出给比赛流程函数结束比赛。内容:

3.6.1 TimeOut 类 (继承自 Exception)

自定义的超时异常，用于与函数运行报错区分。

3.6.2 函数 timer(timeleft, func, params)

计时模块的接口，包含返回值传递、异常处理、超时处理内容; 参数:

- timeleft: 输入函数最大可执行时间，使用时输入为当前玩家剩余时间
- func: 待执行函数，使用时输入为玩家 load 或 play 函数
- params: 函数调用的参数元组

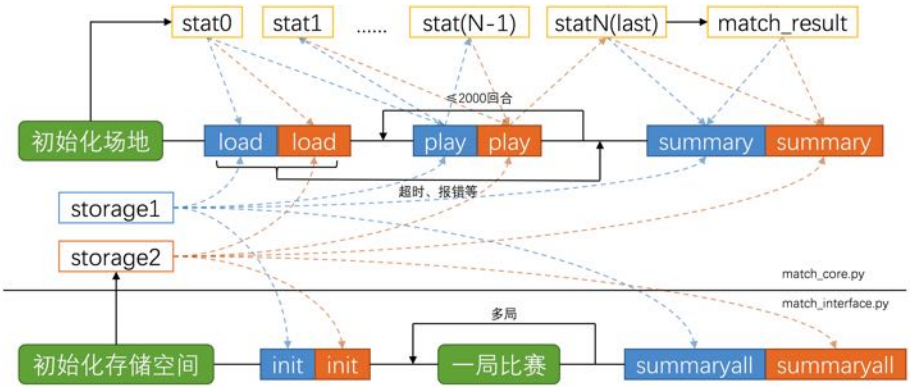
实现方式

运行 func(*params) 并在其前后使用 perf_counter 计时，若时间大于给定剩余时间则抛出 TimeOut 异常; 否则返回函数运行结果与时间消耗。

第四章　循环赛工具说明

4.1　循环赛工具概述

match_core 实现两方玩家进行一次比赛的功能，并可保存双方函数存储空间为全局变量 STORAGE，在多局比赛之间保留; 然而由于多局比赛时需要涉及到初始化存储空间、交换场地等高于单局比赛尺度的操作，match_core.py 中未对其进行直接实现; 循环赛工具 roundRobin_multitask_database.py 实现了此类功能; 下图为游戏执行逻辑的示意图:



进行多轮比赛时，比赛框架为参与者提供新接口 init 与 summaryall，两者均接收参数 storage，用于比赛前接收存储空间，进行数据导入与比赛总结等; 若此二个函数未定义或报错将被跳过，不影响比赛进程;

分组小组赛阶段采用了循环赛赛制，小组每个参与者都与其余所有对手进行比赛，并决出胜负; 单个分区的复杂度为 $O(teams^2 * h * w * rounds)$。

4.2 循环赛赛制介绍

4.2.1 比赛方式介绍

除一局比赛所需的参数 k=51、h=101、maxturn=2000、maxtime=30 外，多局比赛额外定义了比赛回合数参数 rounds=20。

对小组赛中每个对手组合 plr1 与 plr2，将进行至少 11 场、至多 20 场比赛，其中前 10 场比赛由 plr1 先手、plr2 后手，第 11 场起比赛由 plr2 先手、plr1 后手; 若任意玩家总获胜次数大于 10，则提前结束比赛。

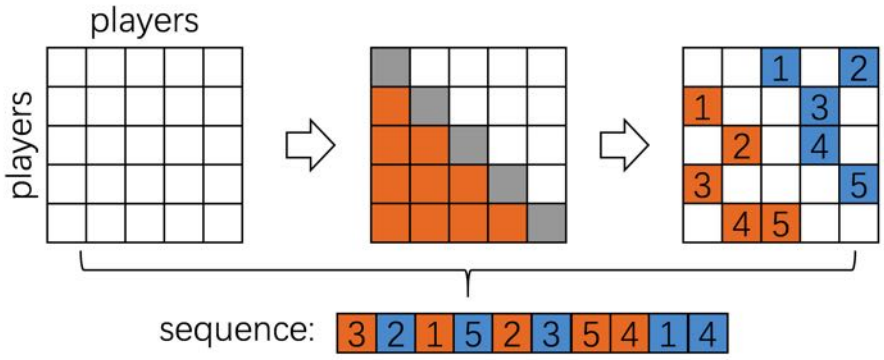
一个对手组合的胜负根据双方总获胜次数决定: 若一方获胜场次大于另一方，则获胜方（次数多者）记 3 分，另一方记 0 分; 若双方获胜场次相同则各记 1 分。

此外，为防止一方 AI 出现无限循环，每个对手组合的对决被设定了总时长阈值，其值为双方均达到最大思考时长且进行最大局数对决，并计入系统耗时的理论总时长。超过该时长的对手组合比赛将被强制结束，双方记 0 分。

4.2.2 实现细节

生成赛制前，比赛框架首先将读入比赛目录内的所有玩家代码文件，验证其合法性并将文件名加入合法玩家列表。合法玩家列表将记录为 players.txt，导入错误信息将被记录为 errors.txt。

在获取玩家列表后，比赛框架将以该列表分别作为两方玩家 plr1 与 plr2，进行一次倒三角形的遍历，随机将 (plr1, plr2) 或 (plr2, plr1) 加入赛程列表; 随后将使用 random.shuffle 打乱赛程列表，并对玩家列表按字典序排序，完成随机赛程的生成。赛制列表记录为 task_buffer 变量。随机赛程序列的生成方式图例如下:



4.3 代码结构

由于任务明确，本项目主进程中主体代码采取了顺序式而非模块化结构，拥有先后逻辑联系的代码块包含在 if ___name___==’___main___’ 语句之下; 由于多进程启动时需要独立的内存空间，进程函数及其依赖的库（match_core，match_interface）位置在主进程代码块之前; 项目结构如下:

1. 进程执行函数 process_task
2. if ___name___==’___main___’:
   (a) 设置相关参数
   (b) 初始化数据库、进程池、进程数据传输队列、比赛结果统计等
   (c) 读取玩家代码、生成赛程序列
   (d) 定义数个辅助函数 helpers
   (e) **进入主事件循环**
   (f) 总结比赛、打包比赛记录

以下将介绍主进程的常量与数据结构，及着重介绍辅助函数与主事件循环的内容，其余部分可分别参考相应章节。

4.3.1 设置参数

变量名	功能及意义
MATCH_PARAMS	传入 match 函数进行单局比赛的设置参数
ROUNDS	比赛局数（单向），每对玩家比赛最大局数为二倍该值，任一玩家获胜局数超过该值判为胜利
MAX_TASKS	最大并行进程数（设置为 cpu 核数 -2）
TIMEOUT	每个进程最大运行时间，由上述参数计算得到
MAX_PROMOTION	（显示）最大晋级人数，默认为 8，可读取命令行参数 2
TEAM	循环赛组别，读取命令行参数 1 得到; 默认为 8
LOG_FORMAT	与 TEAM 变量相关的比赛记录保存路径，在进程内调用比赛记录将保存为 log_format % (*names, index) 路径
AI_PATH	由 TEAM 变量生成的代码文件夹绝对路径

4.3.2 数据存储对象

变量名	类型	功能及意义
dataq	多进程队列	由各比赛进程传出的单局比赛结果，由主进程接收
db	SQL 指针	将比赛记录写入对应组别的数据库
running_tasks	列表	记录当前运行的所有比赛选手名称、进程及其启动时间
rounds_stat	字典	通过嵌套字典记录每个玩家组合当前比赛结果 关键字:(先手玩家名, 后手玩家名) 内容:0: 先手获胜次数,1: 后手获胜次数,None: 平局次数
results_stat	字典	通过字符串记录已完成的对局结果 关键字:(先手玩家名, 后手玩家名) 内容:”+”（先手胜）;”-“（先手负）;”x”（超时终止）

4.3.3 辅助函数 helpers

在进入主循环前声明了数个辅助函数，用于简化代码内容、明确代码逻辑。

flush_queue()

清空进程队列 dataq 内待处理的单局比赛结果，将结果累加至 rounds_stat 字典，并

计入数据库 match_raw 表

update_pair(names, flag)

在一对多轮对决结束（决出结果或超时终止）时运行，根据 rounds_stat 中相应关键字双向结果（name 与 name[::-1]）写入数据库并统计结果，将总结果写入 results_stat 相应位置，并将统计中间结果计入 match_result 表; 参数:

names – 对决双方玩家名

flag – 结束类型（”FINISH”或”TIMEOUT”）

visualize(file=sys.___stdout___)

将当前比赛过程可视化至屏幕，绘制表格，统计各玩家总得分并排序显示于屏幕。根据设置的 MAX_PROMOTION 参数将在该排名玩家后方画线; 参数:

file – 输出流，默认为标准输出（屏幕），可输出至文件。

4.3.4 主事件循环

主事件循环以时间驱动，以 time.perf_counter 获取当前时间。其单次循环的内容按顺序如下:

1. 清空并统计当前的比赛结果队列

2. 遍历 running_tasks 列表，对其中每个任务，如果其执行完毕就标记 FINISH，否则如果其启动时间与当前时间之差高于阈值就标记 TIMEOUT; 对有标记的任务对应的双方名称与标记传入 update_pair 函数汇总结果，并将其移除

3. 若 task_buffer 非空且当前运行任务数小于 MAX_TASKS，则取其中一个传入比赛进程运行，并将该赛程双方名称、该进程与当前时间加入 running_tasks 列表

4. 若距上次刷新可视化结果长于 0.5 秒，则刷新可视化界面并更新可视化计时

5. 在 running_tasks 列表为空时跳出循环

4.4 多进程结构

4.4.1 简介

注意到赛程中每个对手组合所进行的比赛是互相独立的，不受已完成的多轮比赛与结果的影响，在按照赛制运行比赛时采用多进程结构，可以充分利用比赛服务器多核 CPU 的计算力，大幅压缩比赛总时长。

4.4.2 结构与算法设计

由于赛程内比赛组数往往远高于 CPU 核数，需要有效的方式维护运行的线程，使比赛进程中并行任务数维持在合理的范围内; 此外，由于存在强制结束超时进程的需求，multiprocessing.Pool 不支持该应用情景，本项目使用了 multiprocessing.Process 启动每个比赛单元，并在主进程内使用列表结构配合 time.perf_counter 进行时间控制。

与多进程的维护相关的代码包含在主进程的主事件循环中。

4.5 比赛记录

循环赛中每一局的比赛记录将根据 match 函数返回值，输出为复盘记录 zlog 文件与终局情况 pkl 文件，其过程为使用 pickle 库将比赛记录对象序列化，之后使用 zlib 库对其进行压缩减少大小；同时比赛结果的简单统计也将保存在数据库中以备使用;

zlog 文件可用可视化工具 visualize.py 打开并查看复盘记录; 也可用于部分特征数据分析的可视化; 大小约 700-3000KB;pkl 文件可用于基于终局数据分析的可视化; 大小约 1KB;

4.5.1 数据汇总与可视化

主进程在比赛过程进行汇总统计比赛结果的任务，将由比赛进程经数据队列 dataq 传出的比赛结果计入 rounds_stat，并在每组玩家比赛结束后汇总 rounds_stat 内结果并计入 results_stat; 统计过程分别由辅助函数 flush_queue 与 update_pair 函数执行;

在汇总数据的同时，主进程每隔一段时间就执行可视化函数，将当前比赛过程显示于

控制台。可视化分为两个部分: 比赛状态表格与选手排名;

比赛状态表格（Status）：

比赛状态表格内每个单元格横纵轴分别代表先手、后手参赛玩家，按玩家名称字典序排序，行列表头内容相同，只显示行表头; 每行末尾显示了每个玩家当前的得分情况;

表格对角线显示组名，用于对照横纵坐标内容; 显示 p-q 表示正在进行比赛，其中 p 为该行玩家（先手）得分，q 为该列玩家（后手）得分; 若显示 results_stat 内符号说明多局比赛与统计已结束; 若显示空白则说明未开始比赛。

比赛得分按行统计，其中“+”记 3 分，“0”记 1 分，其余不计分。

得分排名（Ranking）：

所有参赛玩家将按得分进行排序，并显示相应的柱状图; 根据 MAX_PROMOTION 值将在相应排名处划线;

数据库记录

循环赛主线程初始化时建立了一个简单的数据库，其中包含两个表单:

- 原始数据:

  CREATE TABLE ‘match_raw‘ (‘plr1‘ TEXT, ‘plr2‘ TEXT, ‘winner‘ INTEGER, ‘type‘ INTEGER); 记录每局双方玩家名称、胜者编号（0 或 1，-1 表示平局）及胜利原因

- 统计结果:

  CREATE TABLE ‘match_result‘ (‘plr1‘ TEXT, ‘plr2‘ TEXT, ‘flag‘ TEXT, ‘win1‘ INTEGER, ‘win2‘ INTEGER, ‘tie‘ INTEGER, PRIMARY KEY(‘plr1‘,‘plr2‘)); 记录了每个对手组合最终的胜负场数;

依需求不同该部分可以扩充更多内容;

4.6 效果演示

运行效果如下图:

```
Status:
   +-----+-----+-----+-----+-----+-----+-----+
K: | -K- |  +  |  -  |  +  |  +  |  -  |  0  |  10
   +-----+-----+-----+-----+-----+-----+-----+
T: |  -  | -T- |  -  |  -  |  -  |  -  |  -  |  0
   +-----+-----+-----+-----+-----+-----+-----+
C: |  +  |  +  | -C- |  +  |  +  |  -  |  0  |  13
   +-----+-----+-----+-----+-----+-----+-----+
D: |  -  |  +  |  -  | -D- |  -  |  -  |  -  |  3
   +-----+-----+-----+-----+-----+-----+-----+
G: |  -  |  +  |  -  |  +  | -G- |  -  |  -  |  6
   +-----+-----+-----+-----+-----+-----+-----+
M: |  +  |  +  |  +  |  +  |  +  | -M- |  +  |  18
   +-----+-----+-----+-----+-----+-----+-----+
W: |  0  |  +  |  0  |  +  |  +  |  -  | -W- |  11
   +-----+-----+-----+-----+-----+-----+-----+


Ranking:
t_f17_menhera  |################################### 18
t_f17_charlie  |########################### 13
---------------------------------------
t_f17_whiskey  |####################### 11
t_f17_KizunaAI |################### 10
t_f17_golf     |############ 6
t_f17_delta    |###### 3
t_f17_Tango    | 0
```

第五章　淘汰赛与半决赛脚本说明

5.1　淘汰赛与半决赛脚本概述

本部分包含以下五个文件:

1. 多局单挑脚本 compKnockoutScenario.py
2. F17 淘汰赛使用的代码 compF17Quarterfinal.py
3. N17 淘汰赛使用的代码 compN17Quarterfinal.py
4. F17 半决赛使用的代码 compF17Semifinal.py
5. N17 半决赛使用的代码 compN17Semifinal.py

多局单挑脚本 compKnockoutScenario.py 主要设计提供于本竞赛的淘汰赛与半决赛执行; 本脚本呈现为单个函数形式，以下述函数定义语句起头:

def knockoutScenario(plrs, FOLDER):

本质上该函数内的代码可单独运行，即去除函数定义语句、调整缩进与变量 plrs, FOLDER 的输入后可直接运行; 为降低单个死循环引起的程序崩溃风险，并利于竞赛同时进行两个分区，且方便后续的晋级导致的参赛者调整，编写程序时将所有代码封装到该函数，方便竞赛过程中多次调用;

其余四个文件为竞赛用脚本，结构相同，均以提取参赛算法的函数起头，并在函数定义完后，按要求调用赛制要求的对应文件并执行，之后进行相应的赛局的结果显示;

竞赛用脚本写为四个文件乃方便操作和结果呈现; 总复杂度为 $O(teams*rounds*h*w)$。

5.2 多局单挑脚本的代码结构

5.2.1 函数输入

knockoutScenario 要求输入 plrs 与 FOLDER:

- plrs :

  应置入参赛代码的名称和函数，以供下述赛程创建语句使用:

  name1, func1, name2, func2 = plrs[0][0], plrs[0][1], plrs[1][0], plrs[1][1]

- FOLDER :

  应置入参赛代码所在的文件夹，方便存取对局结果文件等用途使用;

5.2.2 函数结构

具体结构上可分为以下四个部分:

1. 函数的首个部分为竞赛环境的一般设置，包含:
   - 屏蔽算法的 print 避免影响观赏;
   - 设置比赛参数;
   - 创建赛程;
   - 图形化计分表;
   - 比赛结果统计表;
   - 存储空间的初始化;
2. 定义 outputer 函数:

   outputer(match_result, i, GAMES, pairResult, name1, name2, x)

   outputer 函数于每局战斗结束时调用，接受当前战斗的相关参数（包含战斗结果，战斗局数编号，比赛结果统计表、参赛者名称与上一个图形化计分表等参数），使其能按图形化界面与相应格式打印比赛结果;

3. 根据赛程比赛的两个 for 语句，使用以下四个先前定义的函数完成比赛主体部分:

   match_core, match_result, save_match_log, outputer

   其中于第 1+GAMES 局起加入加入胜场数判断，若有参赛者已取得超过半数场次的胜利，则直接终止比赛程序;

4. 最后一个部分乃数据的总结、结果的整理与函数返回值; 为方便观赏，加入按 enter 键才开始新一个竞赛对的对决的功能;

5.2.3 函数内变量

函数内变量名的内容与意义如下:

变量名	类型	用于图形化积分表的变量
x	prettytable	双方游戏信息
STAT	[., .]	记录参赛对的双方分别累计的胜场数;
i	int	局数编号
winner	str	对应的胜方
state	str	终局状态
rmklst[flag]	str	终局原因
gameRounds	int	终局时进行的步数

5.3 竞赛用脚本的代码结构

由于主要运行程序包含于 knockoutScenario 内，竞赛用脚本仅需要按要求调用与给定输入即可: 提取参赛算法的 quarterfinal 函数要求输入指定文件夹，并返回对应的参赛队伍信息，以供竞赛使用;

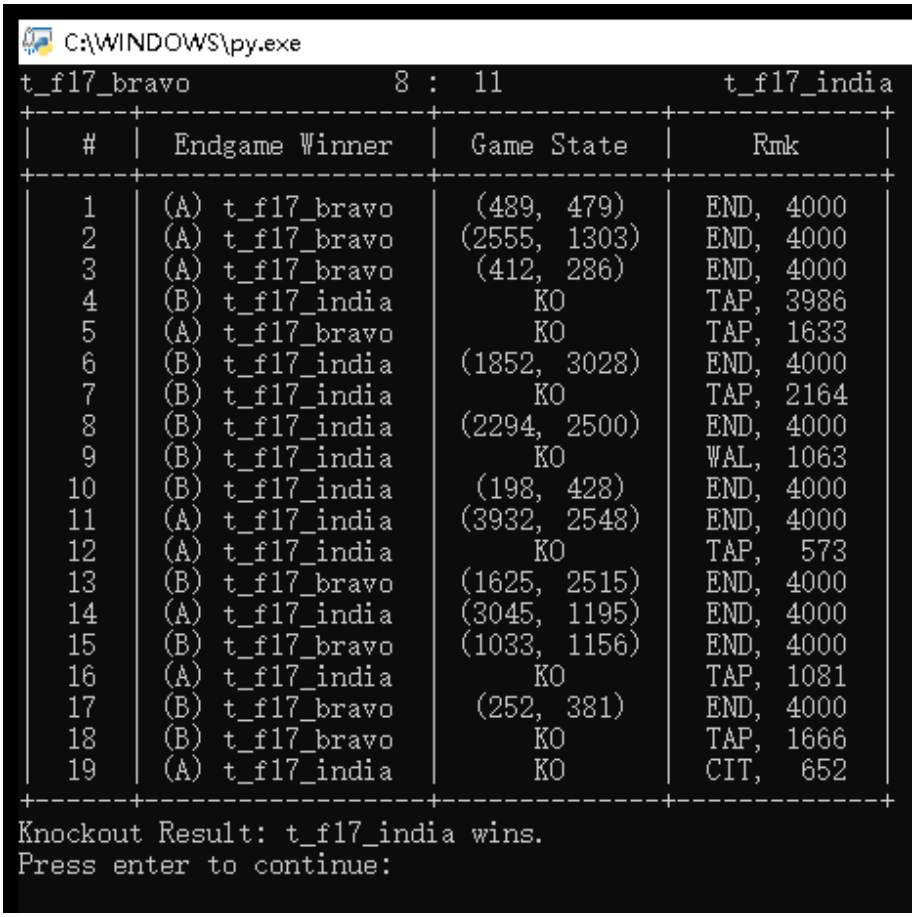
提取算法完成后，仅需要按以下方法找到指定文件夹并调用 knockoutScenario 返回竞赛结果即可:

FOLDER = os.getcwd()+'/F17/E1W2'

winnerA, loserA = knockoutScenario(quarterfinal(FOLDER), FOLDER)

5.4 效果演示

运行效果如下图:



其中第一行为当前累计总比分概览; 第二行起为表格，表格内的四列分别为:

该局编号; 该局获胜者; 终局状态; 终局原因与终局时进行的步数;

以上为运行至一个参赛对战斗完毕，程序暂停的画面，方便观赏，可输入 enter 键令程序继续运行下一个参赛对战斗的进程（或跳出，若程序全部运行完毕）

第六章　可视化工具说明

6.1　可视化工具概述

本项目含两个文件:

1. visualize.py: AI 对战与比赛记录读取的可视化工具

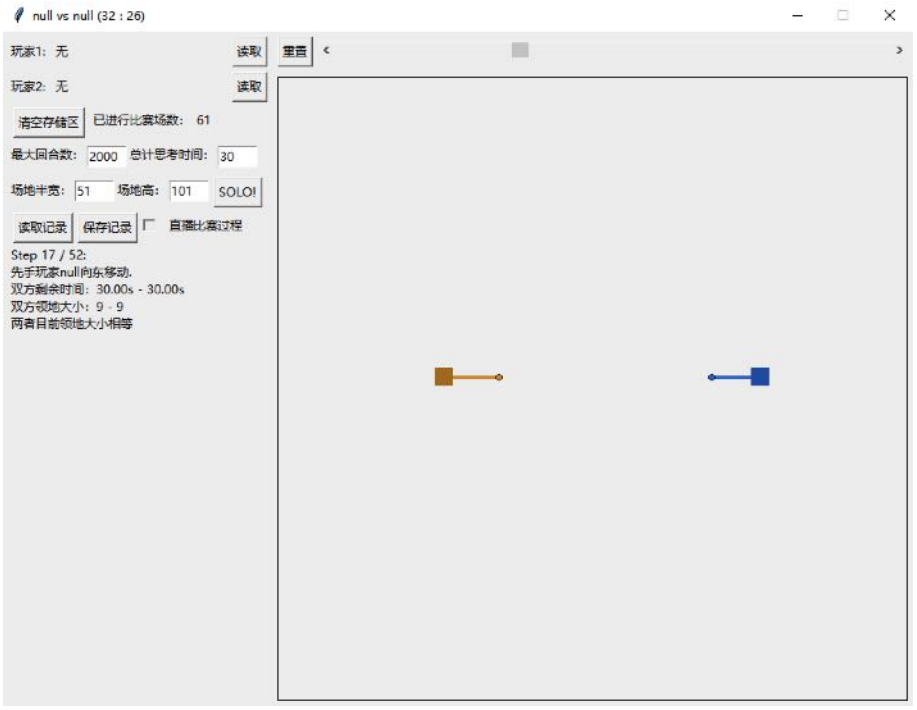
2. glory_of_mankind.py: 人机对战工具

本项目使用 python 内置库 tkinter 对纸带圈地对局进行可视化; 其设计理念为读取运行比赛所用参数，以及 match_core 生成的标准格式的每帧画面，并根据此调用 tkinter.Canvas 相关接口进行可视化显示。

根据不同的应用场景，不同的可视化程序进行了相应的功能与界面扩展。运行可视化程序需要包含 tkinter 库的 python3 环境及 match_core.py。

6.2　visualize.py

visualize.py 集成了 AI 对战与比赛记录 zlog 文件读取的功能，界面元素如下图:

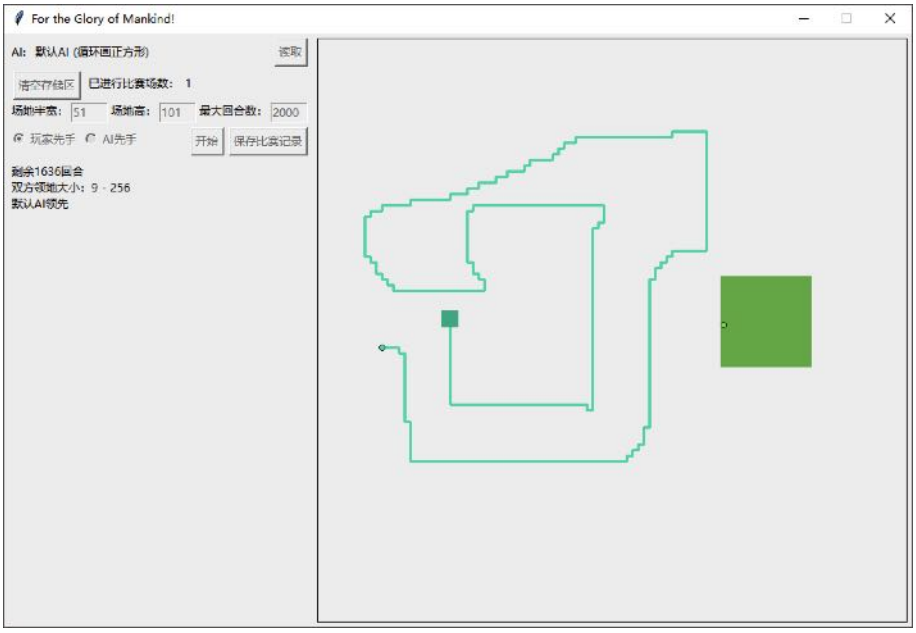


界面左侧为控制栏，包含比赛双方 AI 选择模块，比赛自定义参数设置模块，比赛记录存取模块，比赛信息等，上述模块会在后文进行详细介绍; “清空存储区”按钮将重置 match_core.STORAGE 变量（即双方 AI 存储区）并重置比赛场数等统计信息; “SOLO!”按钮将启动一场比赛; “直播比赛过程”选框勾选时双方比赛过程内每帧都将直接显示，否则将在比赛结束后整局读取记录;

界面右侧为显示栏; 右侧顶部为播放进度条，在读入比赛记录后可用，可以控制播放、暂停、时间定位等; 主体部分为可视化的比赛界面;

6.3 glory_of_mankind.py

人机对战工具实现了手动控制比赛一方与选定的 AI 模块进行对战的功能，并支持比赛记录的保存，名称 Glory of Mankind 取自游戏《NieR:Automata》; 其界面如下图:



界面左侧为控制栏，包含对战 AI 选择模块、清空存储区等内容，与 visualize.py 类似; 由于是人机对战，只需选择一个 AI 模块，并设置人类玩家为先手（左侧）或后手（右侧），另一方将使用特殊的 AI 模块（human_control）并使用“人类”作为名称; 此外，由于人类思考时间较长，比赛设置内移除了时间限制并重写了 match_core 内相关模块，人机对战比赛将不会出现一方超时的比赛结果;

界面右侧为显示框，显示了比赛进行（或结束）时的场景; 为保证窗口控制非阻塞，比赛与更新画面过程独立于主线程之外，故本程序与 visualize.py 在显示部分有少量区别;

6.4 显示模块（display_frame 类）

display_frame 类为封装的帧显示模块，通过 tkinter.Canvas 相关方法显示比赛进程中的场地内容; 其中玩家、纸带的显示与领地的显示采用了不同的策略，将详细介绍;

另外，由于涉及到 tkinter.Frame 布局，visualize.py 内的记录查看功能也内置于该类，将在后文相应部分进行介绍; 该部分只以 glory_of_mankind.py 中的实现为例，介绍基础的显示设施涉及的对象与过程;

在两个可视化程序中，display_frame 类均实例化为 display 对象，布局于窗口右侧;

6.4.1 设计理念与内容

Canvas 对象需要显示的内容包含如下两部分: 一是总量、拓扑结构等都不定，但在有限个固定位置内只有固定状态的双方玩家领地，二是每帧最多存在一个的玩家纸带以及只存在一个的玩家本体;

在本项目实际实现中，采用提前布局网格的方法显示领地，每个坐标都放置一个矩形对象，领地所属变动只需调用 Canvas.itemconfig 设置相应坐标矩形的颜色即可; 玩家与纸带分别使用椭圆对象与折线对象实现，可通过调用 Canvas.coord 改变其关键点更新位置;

display_frame 类包含的（与显示功能直接相关的）非函数对象如下表:

名称	功能描述
cv	Canvas 对象，用于显示及调用方法更新内容
names	该比赛参与双方名称
colors	根据比赛双方名称 hash 生成的对应领地颜色
bands	双方纸带 canvas 对象对应编号，用于 cv.coord 调用
players	双方纸带 canvas 对象对应编号，用于 cv.coord 调用
size	已生成的比赛场地网格宽高
pixels	记录比赛场地网格内容的列表，用于 cv.itemconfig 调用
last_frame	已渲染的帧内容，用于与新帧对照
default_frame	空白场景帧，在清屏后作为初始的 last_frame

函数对象如下表:

函数名称	接收参数	功能
___init___	窗口控件 root	将对象与 root 连接，并运行 _init_screen 函数
_init_screen	无	将控件 cv 布局于 root 内，声明数个初始变量
_setup_grid	网格大小 size	根据输入更新 pixels，并运行 _clear 函数
_setup_players	names	根据输入更新 bands 与 players
_clear	无	清除 pixels 内已有内容
_update_screen	cur_frame	更新 cv 绘制输入帧内容

6.4.2 初始化

显示模块在每次运行（开始比赛或读取比赛记录）时会首先进行初始化，依次运行 display._setup_grid 与 display._setup_players 函数，_setup_grid 内部又会调用 _clear 函数;

其中，_setup_grid 函数会检测输入参数 size，在其不等于 self.size 时重新布局 pixels; 随后运行 _clear 初始化屏幕并运行 root.update 刷新窗口;_clear 函数会根据 last_frame 遍历 pixels，将所有包含内容的领地重设为 root 背景色清空，并将 last_frame 指向 default_frame;

_setup_players 函数在 names 不等于已写入名称时运行，记录名称于 names 变量并删除已有玩家、纸带对象; 根据玩家名称 hash 生成主题颜色并记录，随后创建双方纸带与玩家对象并记录编号;

6.4.3 领地绘制

领地、玩家、纸带绘制，及文字信息更新均在 _update_screen 函数内实现;

在传入帧后显示新的领地状态具有 O(w*h) 的复杂度，需要检查并设置场地内每个格点坐标对应矩形为相应颜色，其中双方领地颜色在初始化过程中记录在 colors 变量内，空位置则通过调用 root['bg'] 获取背景色;

由于更新 canvas 对象需要耗费较多时间，此处采用了缓存加速的策略，将更新前的帧内容保存在 last_frame 对象内，并在遍历网格时将当前帧与上一帧进行对比，只更新颜色有变动的坐标，相比全部更新节省了大量时间;

6.4.4 玩家、纸带绘制

玩家通过 canvas 椭圆对象显示，每次更新帧时通过其游戏坐标与显示网格大小计算出椭圆对象在屏幕上实际的左上、右下坐标，并通过 cv.coord 更新; 绘制纸带过程使用了当前玩家坐标，并利用了 match_core 内额外传出的纸带方向序列，从玩家坐标开始回溯获得纸带上每个格点坐标; 此外，由于纸带上常出现连续方向，通过记录上一步方向，当接收相同方向时删除对应的中间节点，可以简化纸带关键点数以加速;

6.4.5 文字信息显示

在遍历检查领地的过程中，_update_screen 函数同时对比赛双方的总领地面积进行了计数，并将计数结果与其余所需的内容传递给生成文字信息的代码; 后者将根据双方玩家名称、方向、占地等生成对当前局势的描述，并显示于窗口左侧下部的信息栏;

6.5 通用功能模块

6.5.1 AI 读取模块（AI_selection 类）

AI_selection 类集成了玩家 AI 模块的读取与合法性检测，以及在 tkinter 窗口中布局的功能; 初始化 AI_selection 对象时需要指定其父 tkinter 控件，与显示的名称（AI，玩家 1，玩家 2 等）; 通过访问每个 AI_selection 实例的 AI_MODULE 与 AI_NAME 参数可以获得其成功读取的 AI 模块与对应名称;

其包含的 tkinter 控件包含可变显示模块名称的 Label 对象，与读取 AI 模块使用的按钮; 点击按钮将调用 tkinter.filedialog.askopenfilename 获取 py 文件的路径，尝试读取其内容并使用 exec 函数声明于一个类内部，并将该类写入 AI_MODULE 参数; 在成功读取后，AI_NAME 参数与相应的显示也将更新，同时将重新初始化 match_core 内部已有的存储内容;

6.5.2 比赛参数设置（checked_entry 类）

checked_entry 类为对 tkinter.Entry 的一个封装，执行从中获取合法输入文字，自动检查合法性与转化为相应类型的功能，用于设置比赛参数;

初始化 checked_entry 对象需要指定父控件 root，输入数值类型 type（int，float 等），默认值 default 与该输入框描述 text; 初始化函数会排列描述的 Label 与输入框 Entry 进父控件，并自动为 Entry 绑定合法性检查事件; 通过调用 get 函数可以获取其中已经输入的数据;

6.5.3 运行比赛（run_match 函数）

run_match 函数运行时会首先从几个输入框获取设置的双方参赛者模块与比赛参数，并更新一些简单的统计信息; 随后向显示模块传入比赛参数与选手名称进行场地初始化，最后开始比赛; 在比赛过程中使用了“控件锁”的机制防止误触造成逻辑错误; 比赛结果将被记录于全局变量;

glory_of_mankind.py 在导入 match_core 时通过重写其 timer 函数移除了超时判定; 此外，为防止 match_core 运行阻塞主窗口按键输入等，未将 match_core.match 函数直接在主线程中运行，而是放置于独立线程内;

6.5.4 即时显示

即时显示功能应用于 visualize.py 勾选“直播比赛过程后”，以及 glory_of_mankind.py 全程; 程序内通过将帧输出接口 match_core.FRAME_FUNC 指向 display._update_screen，实现即时显示功能;

visualize.py 在比赛运行期间是完全阻塞的，需要在 FRAME_FUNC 内增加调用 tk.update 以手动更新显示，封装为 update_hook 函数;

由于 glory_of_mankind.py 中调用帧输出接口也在外部线程内，在大块圈地时从主线程可以看到逐行遍历的“上色”过程;

6.5.5 控件锁

窗口内所有可操作的控件，包括输入框、按钮等，均被加入全局内名为 OP_WIDGETS 的列表，用于统一启用/关闭控件;widget_off 函数在每场比赛开始前运行，遍历列表内控件并将其 state 设置为 DISABLED; widget_on 函数在比赛结束后运行，遍历列表内控件，由于 Entry 与 Button 启用状态不同，其通过 try except 结构启用每个控件;

visualize.py 内进度条 ScrollBar 控件无 state 属性，其通过在拖动事件中判断比赛是否正在运行来决定是否执行功能;

6.5.6 记录保存

保存记录时调用了 tkinter.filedialog.asksaveasfilename 获取合法的保存路径，并将全局变量内的比赛记录对象使用 pickle 序列化后使用 zlib 进行压缩，并写入保存路径;

6.5.7 垃圾清理

由于比赛记录文件占用大量的内存，为防止自动内存清理不及时造成内存溢出，在启动比赛、读取比赛记录等关键操作前会调用 gc.collect 强制回收内存;

6.6 其他功能

6.6.1 记录读取与进度条

记录读取与进度条为 visualize.py 特有;visualize.py 的显示模块内增加了对查看比赛记录的支持，在 Canvas 对象上方增加了播放控制按钮与滚动条; 显示模块中新增的相关对象如下表:

名称	功能描述
playing_status	指示播放状态，0 为暂停/停止，1 为播放
frame_seq	帧序列，由 load_match_result 函数获取
frame_index	播放进度，指示 frame_seq 下标
match_result	比赛结果元组，由 load_match_result 函数获取

显示模块新增的相关函数如下表:

函数名称	接收参数	功能
button1_press	无	按钮事件; 根据当前播放状态决定播放/暂停/返回第一帧（若已播放至末尾）
scroll_option	可变参数 *args	滚动条事件，包括滑块与前后按钮; 只在非比赛运行且帧序列多于 1 帧时执行，运行时暂停播放事件，更新播放进度，渲染新的帧并更新滚动条位置
scroll_update	无	根据播放进度与帧序列总长度确定滚动条位置
update	无	播放事件; 只在 playing_status 为 1 时运行，增加播放进度并渲染对应帧，若播放至末尾则自动停止，同时通过调用 root.after 并传入 update 函数实现循环事件
load_match_result	比赛记录 log 与初始化标志 init	从 log 读取比赛结果、帧序列，初始化播放进度与相关控件，在 init 为真（加载比赛记录）时读取比赛参数进行初始化，并渲染第一帧
_update_info	当前帧 frame 与计数 field_count	封装的回合信息更新函数; 根据比赛是否运行决定显示剩余回合数或播放进度，其余类似人机对战程序

另外，可视化工具中为读取比赛记录功能新增了 load_log 函数以读取比赛记录文件; 读取的对象将被写入全局变量并传递给相应变量。

6.6.2 human_control 类

human_control 类为 glory_of_mankind 特有。人机对战程序中，人类方 AI 模块为特殊的 human_control 类，可以接收的鼠标键盘输入控制玩家行动，同时也执行了帧率控制的功能; 其内部的 control 参数指示了当前采用的控制模式（默认为键盘“key”，可选为鼠标模式）;

6.6.3 键盘控制

键盘控制模式为 glory_of_mankind 特有，为自动逐帧行进，涉及到的类变量有指示方向 op，延时设置 delay 与延时控制变量 old_timer，全局变量有键位对相应方向的映射 key_mapping;tkinter 窗口中绑定了按键事件 key_control 函数;

key_control 函数读取按键事件的 keysym 并在 key_mapping 内查找，若该键存在（方向键或 WASD）则将控制模式设置为“key”并传入相应方向至 human_control.op;

在 human_control 内，load 函数将 old_timer 参数初始化为当前时间;play 函数在控制模式为“key”时，首先进行帧率控制，在当前时间距 old_timer 小于 delay 时 sleep 相应时间差，随后刷新 old_timer 至当前时间; 随后，若指示方向存在，则通过计算指示方向与玩家当前方向之差决定输出转向方向，并将使用过的指示方向清空;

6.6.4 鼠标控制

鼠标控制模式为 glory_of_mankind 特有，为阻塞模式，涉及的类变量有目标位置 pos 及鼠标状态 mouse_holding，并在显示窗口 display.cv 上绑定了鼠标左键按下事件 mouse_down，抬起事件 mouse_up 与拖动事件 set_position;

mouse_down 函数将控制模式切换为“mouse”，并将 mouse_holding 设为 True，随后执行一次 set_position 函数;mouse_up 函数将 mouse_holding 设为 False;set_position 函数通过显示参数计算出鼠标点选位置的游戏坐标并传入 pos 参数;

human_control.play 函数内，当控制模式为“mouse”时，首先在鼠标抬起或点选位置与玩家位置重合时保持阻塞（若中途切换为键盘控制则递归调用一次自身跳出循环），随后根据当前玩家方向分情况讨论决定前往点选位置的前进方向，以实现只在按下鼠标时前进的功能;

第七章　数据分析用脚本说明

7.1　数据分析用脚本概述

本部分含两类:

第一类乃对于多个对局记录 的概览，要求按说明存放对局记录 pkl 文件; 此类脚本可直接执行以获得分析结果; 此类脚本包含以下两个文件:

1. bars.py;

2. dots.py;

两者均有多份直接存放防于竞赛配置文件夹副本 CompetitionResult 内，可直接于竞赛时，根据记录文件 pkl 自动执行使用; 两者均支援输入队伍代号以将相应队伍赋予特别标示的功能;

第二类乃对于单个对局记录 的特定特征数据分析的可视化; 此类脚本执行时需手动选区欲分析对局的 zlog 文件; 此类脚本包含以下四个文件, 存放于 ornamentalTools 文件夹:

1. lineChartAbsoluteFields.py;

2. lineChartDisDands.py;

3. lineChartDisScrolls.py;

4. lineChartFieldratio.py;

7.2 各脚本功能介绍

1. bars.py: 以二维带误差棒 (errorbar) 散点图形式，以参赛队为单位，表现组内所有参赛者在各局终局时的局均步数、局均领地面积、各局步数标准差、各局领地面积标准差，其中前两者表现为图上的点，后两者表现为两个误差棒; 所有数值为 float 形式; 复杂度为 $O(files * h * w)$;

2. dots.py: 以二维散点图形式，以各局战斗为单位，表现组内所有对局中，各局终局时的总步数与领地面积; 容易知道一局战斗会产生两个点，其中总步数部分相同，领地面积可能不同; 复杂度为 $O(files * h * w)$;

3. lineChartAbsoluteFields.py: 以折线图形式，以参赛队为单位，表现对局内各参赛者在各回合时的当前领地绝对大小; 所有数值为 int 形式; 复杂度为 $O(rounds * h * w)$;

4. lineChartDisDands.py: 以折线图形式，以参赛队为单位，表现对局内各参赛者纸卷在各回合时的与对方纸带的最小距离（四邻域意义下）; 所有数值为 int 形式; 复杂度为 $O(rounds * h * w)$;

5. lineChartDisScrolls.py: 以折线图形式，以参赛队为单位，表现对局内各参赛者纸卷在各回合时的与对方纸卷的距离（四邻域意义下）; 所有数值为 int 形式; 复杂度为 $O(rounds * h * w)$;

6. lineChartFieldratio.py: 以折线图形式，以参赛队为单位，表现对局内各回合时第一队领地大小与两队领地和的比值，为介于 0 与 1 之间的 float 形式; 复杂度为 $O(rounds * h * w)$;

7.3 多局分析脚本的代码结构

多局分析脚本的两个代码结构相同:

1. 依次调用欲进行分析的 pkl 文件以获取相应对局的终局结果;

2. 寻找参与该局战斗的两个算法，并计算达到终局结果时的进行步数与双方领地面积;

3. 存数据到列表;

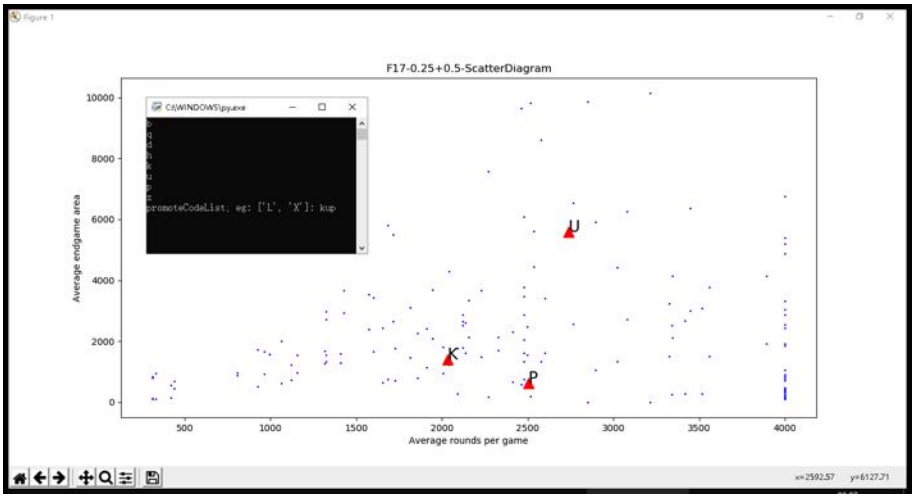
4. 反复进行前三步，得到所有符合条件的对局的信息

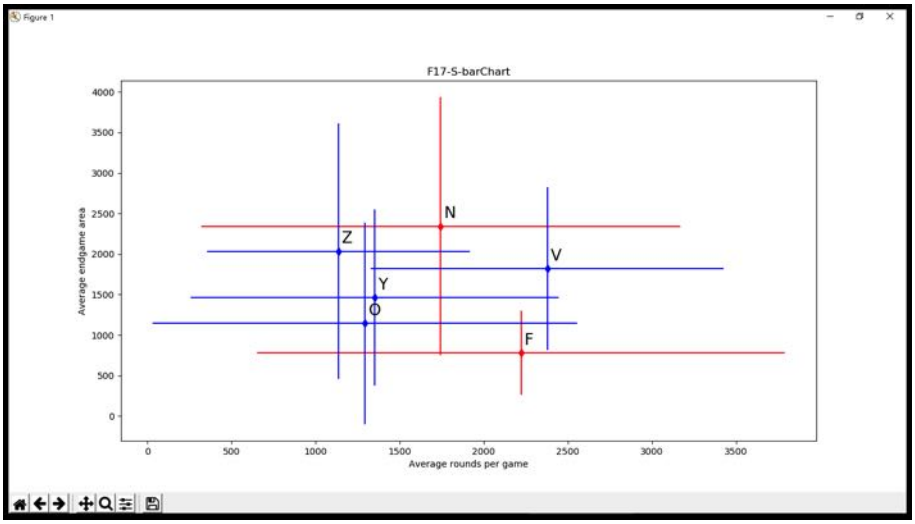
5. 对欲绘图的图标进行初始化;

6. 按要求标示各点与/ 或误差棒;

7.4 多局分析脚本效果演示

多局分析脚本运行效果如下图:





7.5 单局分析脚本的代码结构

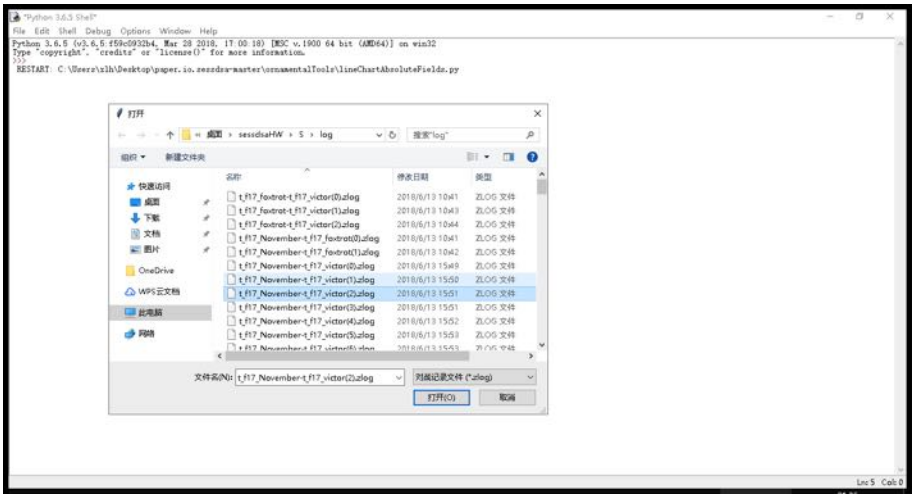
单局分析脚本的四个代码结构相同:

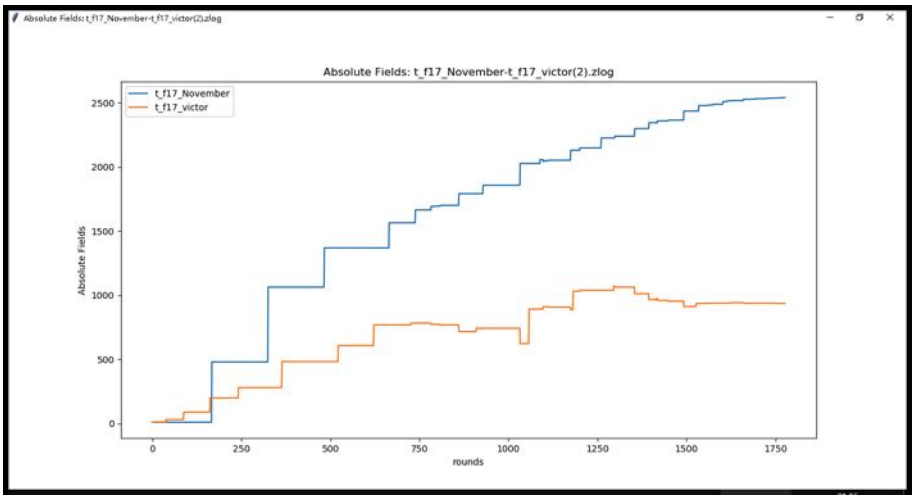
1. 手动选择及读入 zlog;

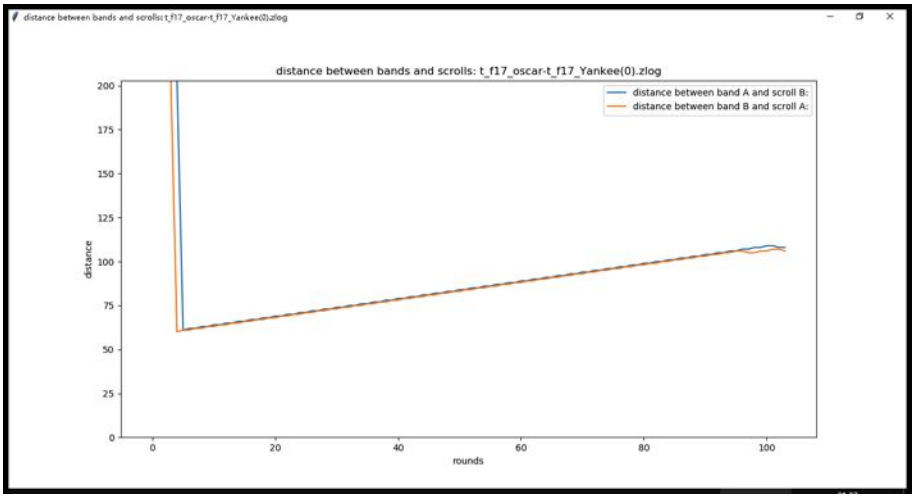
2. 计算绘图所需数值并按要求呈现图形;

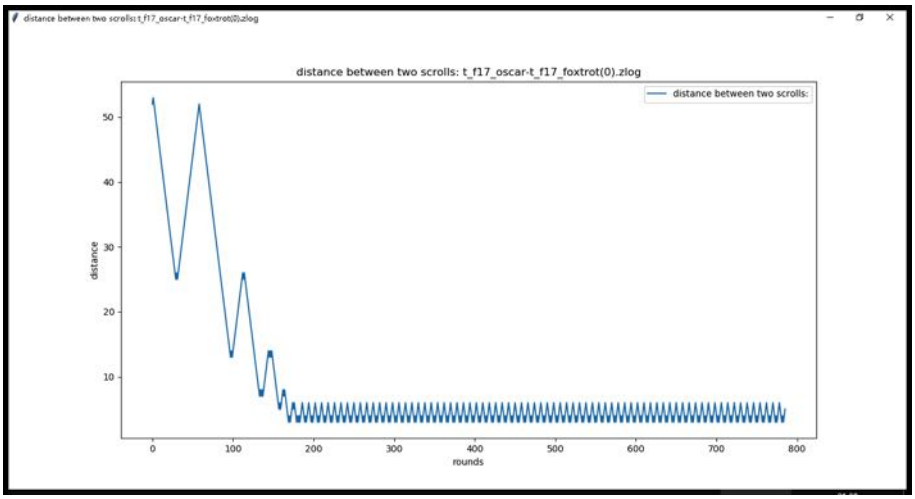
7.6 单局分析脚本效果演示

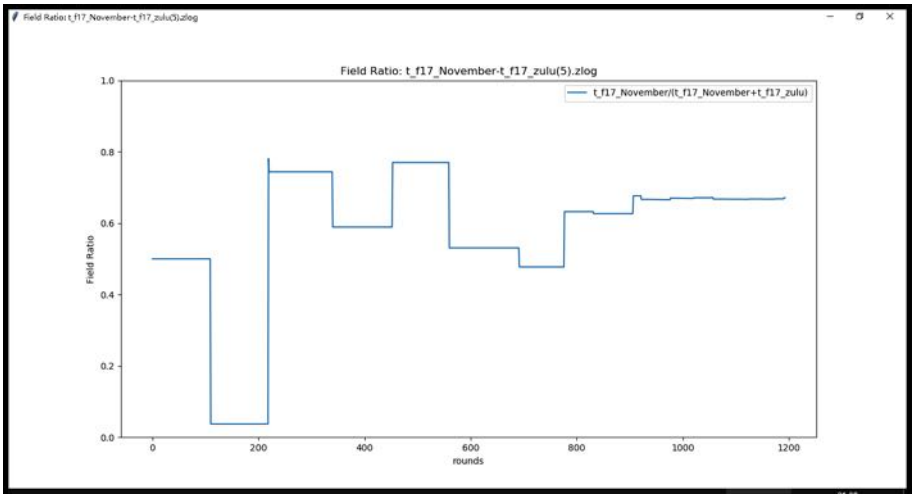
单局分析脚本运行效果如下图:






distance between bands and scrolls: t_f17_oscar-t_f17_Yankee(0).zlog
distance between band A and scroll B:
distance between band B and scroll A:
distance
rounds


distance between two scrolls: t_f17_oscar-t_f17_foxtrot(0).zlog
distance between two scrolls:
distance
rounds


Field Ratio: t_f17_November-t_f17_zulu(5).zlog
t_f17_November/(t_f17_November+t_f17_zulu)
Field Ratio
rounds

第八章　竞赛结果概述

8.1　N17 联盟循环赛分区结果公示

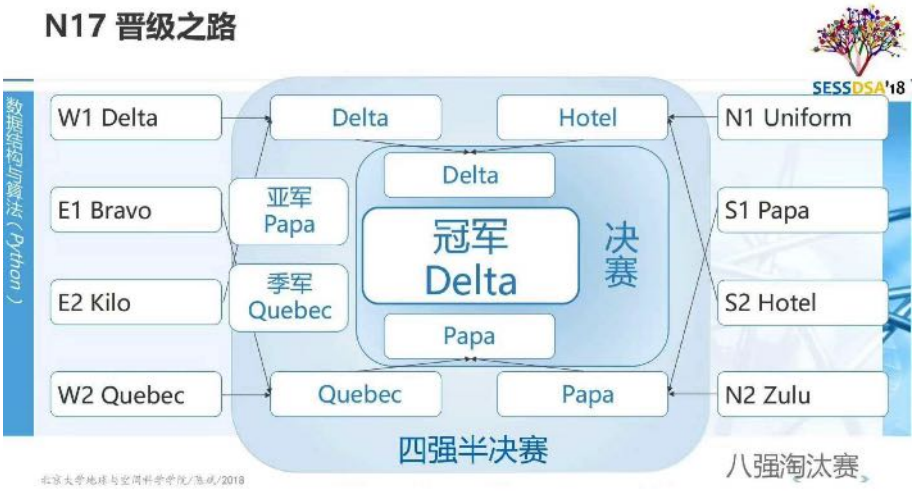
- Group N: ['n17-Charlie-苏文霖', 'n17-Lima-周忠鹏', 'n17-Oscar-丁力', 'n17-Uniform-席子涵', 'n17-Victor-王馨', 'n17-Zulu-周竞宇']
- Group E: ['n17-Alpha-王子瑜', 'n17-Bravo-张瀚垚', 'n17-Echo-向飞燕', 'n17-Golf-姜峰', 'n17-India-张楚珩', 'n17-Kilo-欧阳萌淞']
- Group W: ['n17-Delta-尹超', 'n17-Quebec-欧一', 'n17-Sierra-刘晟', 'n17-Tango-韩豫哲', 'n17-X-Ray-范家豪', 'n17-Yankee-拉毛切忠']
- Group S: ['n17-Foxtrot-彭晓韵', 'n17-Hotel-申景航', 'n17-Juliet-刁一飞', 'n17-Mike-贺海军', 'n17-November-杨少栋', 'n17-Papa-尹晨析']

8.2　F17 联盟循环赛分区结果公示

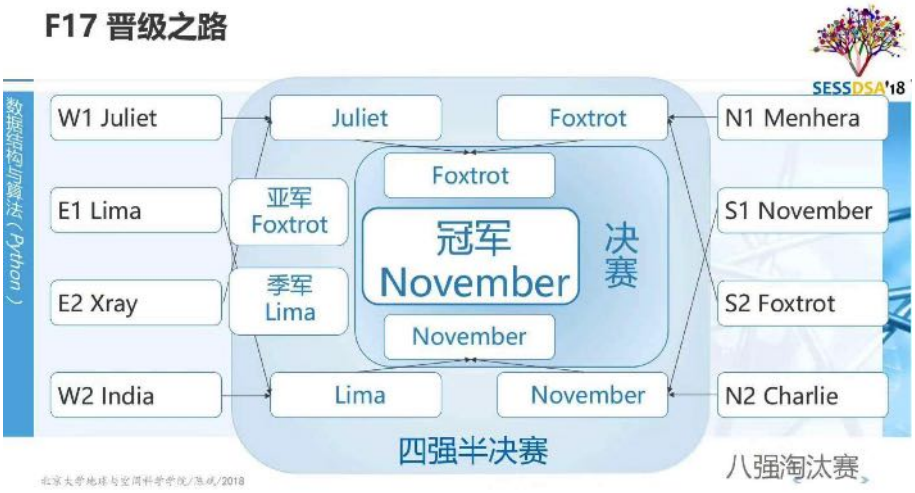
- Group N: ['f17-Charlie-张昀昊', 'f17-Delta-蔡家骥', 'f17-Golf-何为', 'f17-Kilo-姜金廷', 'f17-Mike-王禹菲', 'f17-Tango-杨一龙', 'f17-Whiskey-柯赵轲']
- Group E: ['f17-Echo-莫文韬', 'f17-Hotel-肖力哲', 'f17-Lima-周正清', 'f17-Papa-陈丹丘', 'f17-Romeo-项洋', 'f17-Uniform-余圣杰', 'f17-X-Ray-宣泽远']
- Group W: ['f17-007-刘一鸣', 'f17-Alpha-周子楠', 'f17-Bravo-李想', 'f17-India-黄荣', 'f17-Juliet-刘煜杭', 'f17-Quebec-汪昀鸿']

- Group S: ['f17-Foxtrot-张懿卓', 'f17-November-陈逸凡', 'f17-Oscar-黄赞佑', 'f17-Victor-伍峻琦', 'f17-Yankee-陈羲', 'f17-Zulu-夏一飞']

8.3 竞赛结果公示



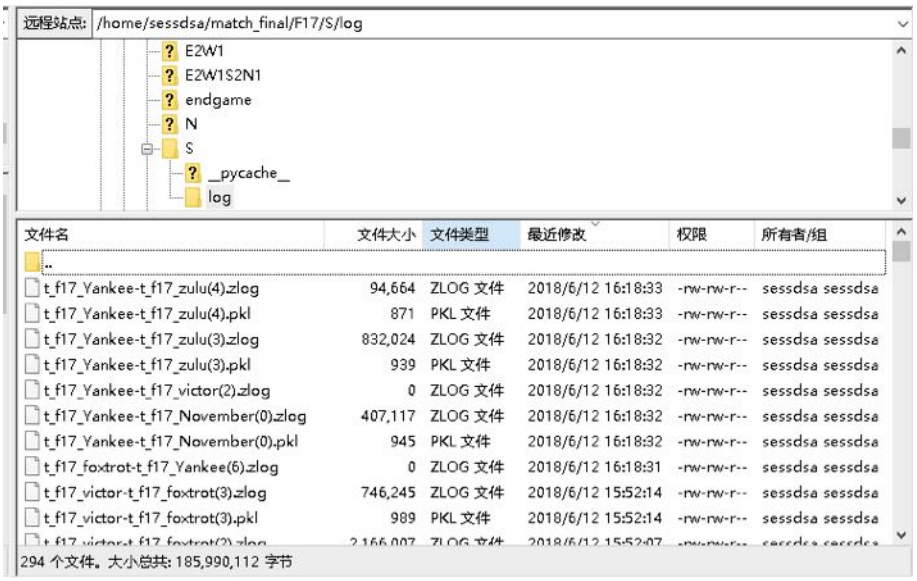
N17 联盟八强



F17 联盟八强

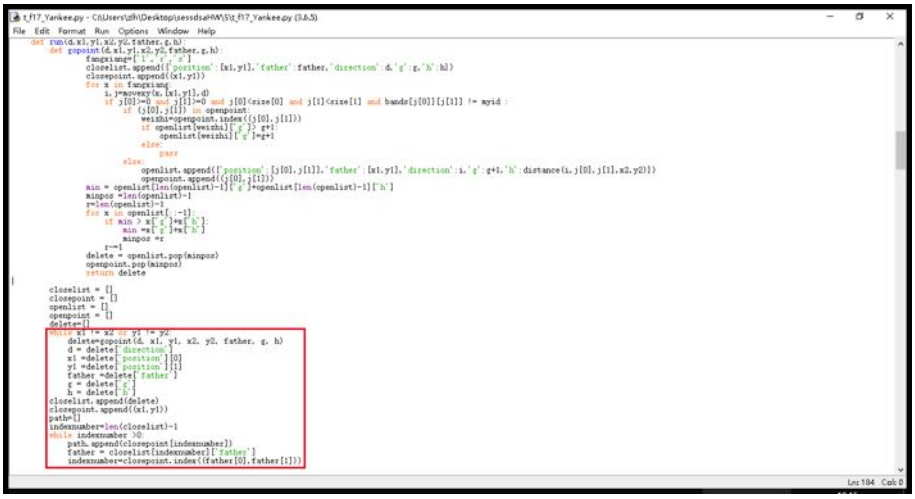
8.4 意外事件

本竞赛于 F17 联盟 S 区循环赛部分发生一意外事件，导致竞赛过程中该赛区无响应近半小时：



当时情况为 F17Y 的四个进程于竞赛中无响应，而区内所有其他比赛已完成比赛。赛中由主办方判定 F17Y 失去资格，手动终止该组剩余进程，扣除 F17Y 已完赛部分所分配的循环赛积分后，由 F17N, F17F 分列该区前两名。值得一提的是此二组分列为该联盟的冠军和亚军; 此事件也导致友谊赛与人机对战延后至 2018 年 6 月 15 日上午 8 时进行。

经技术人员赛后调试，发现无响应原因乃第 190 至 205 行（下图红色方框部分）的两个 while 循环，由于赛局特定条件触发该代码 delete 等列表中的赋值情况无法终止相应 while 循环，导致进程无响应：



技术人员在此呼吁编写代码时**谨慎检查代码中的 *while* 循环**，可用 for 循环或加入强制跳出条件; 以下分别以 while 和 for 语句示范最多进行 maxLoopsAllowed 次循环进程的代码样式，其中 (your criterions) 填入判据，(your treatments) 填入相应处理方式。

while 形式:

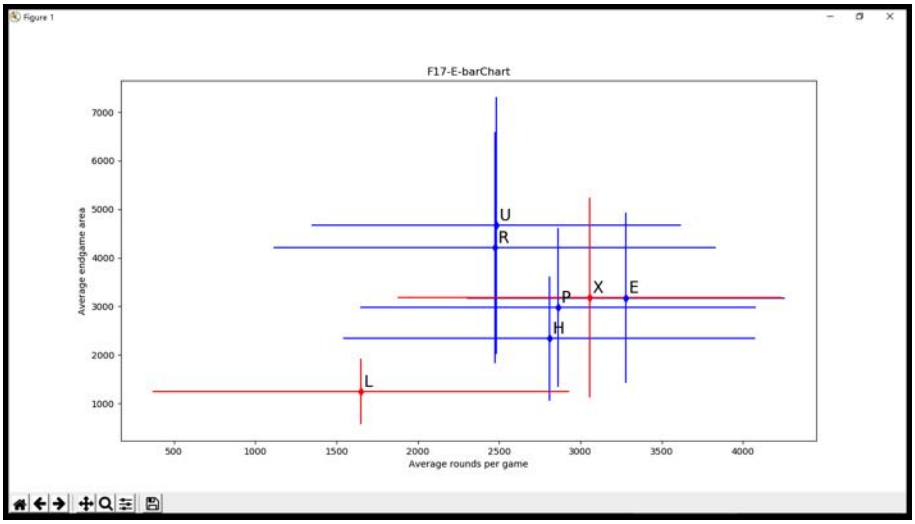
```
counter = 0
while (counter<maxLoopsAllowed) and (your criterions):
(your treatments)
```

for 形式:

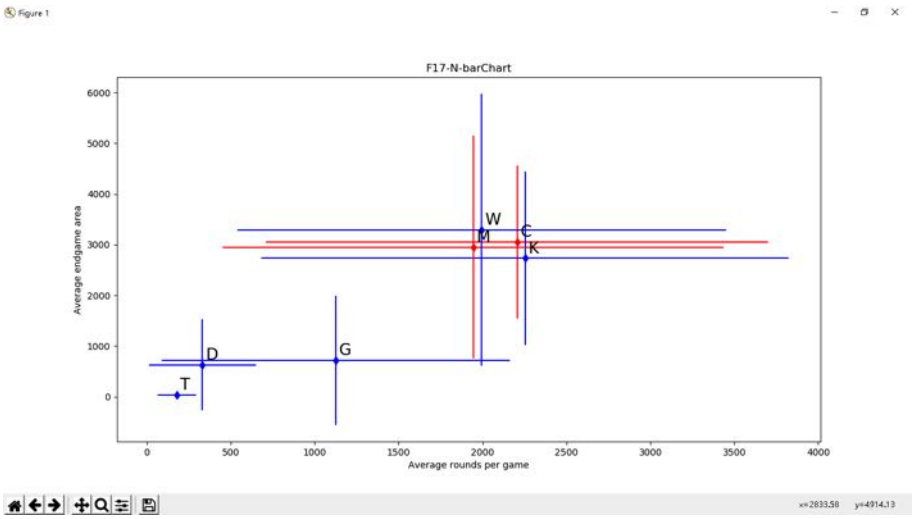
```
for counter in range(maxLoopsAllowed):
if (your criterions):
(your treatments)
```

8.5　循环赛得点分析

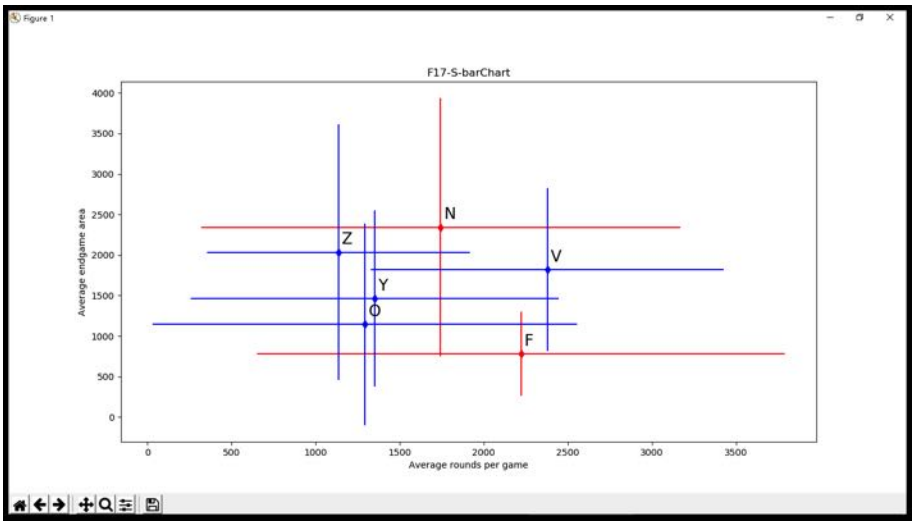
F17 分区 E 中，L 队相对喜欢攻击，倾向于短赛局内将对方淘汰：



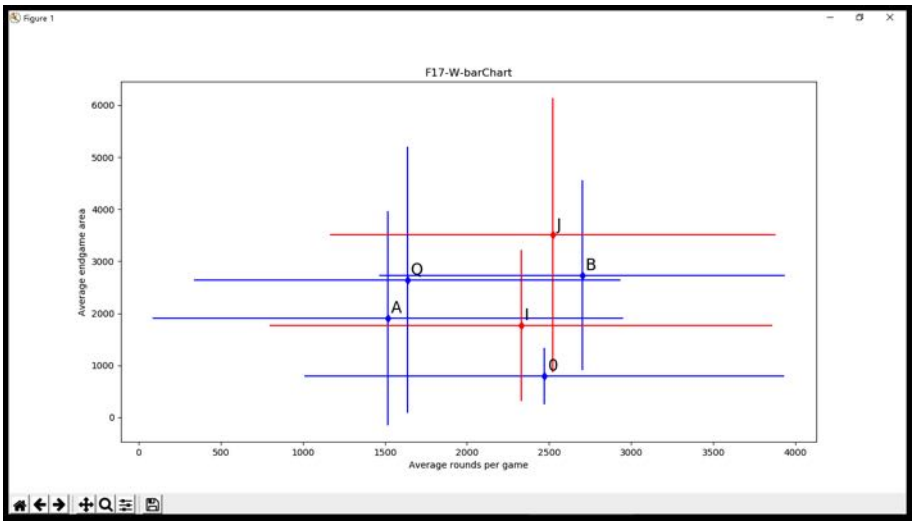
F17 分区 N 中，T 队欲遵循相似模式，然而复制结果失败：



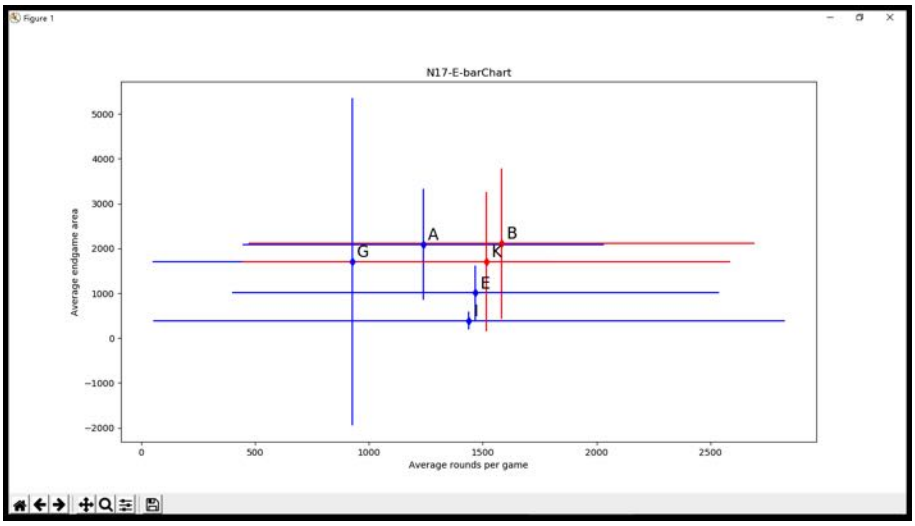
F17 分区 S 中出线了全联盟的冠军和亚军，两者分列为分区中局均圈地最多和最少的：



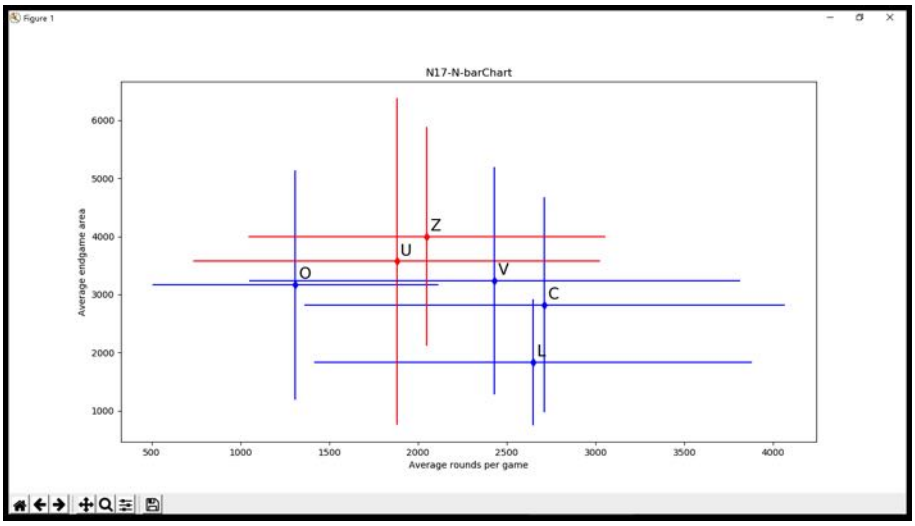
F17 分区 W 则稍显特征不明显：



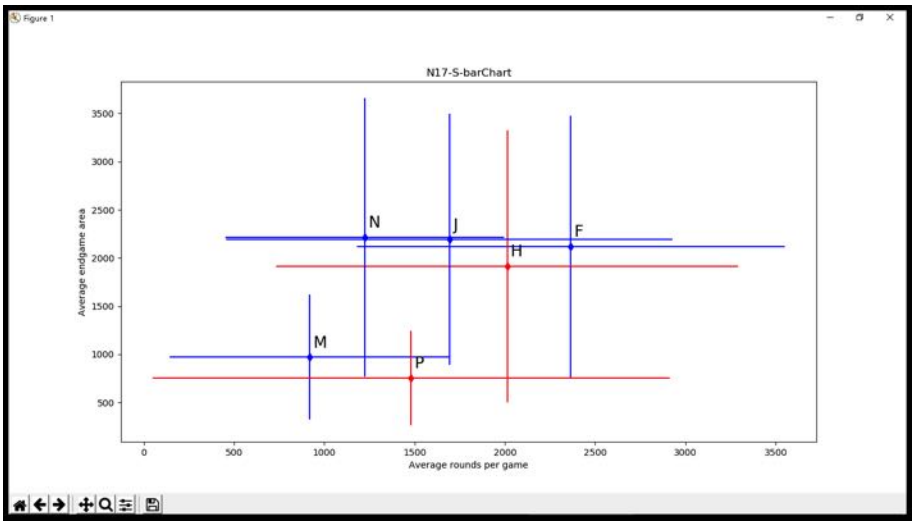
N17 分区 E 的战局激烈，实况中 B、K、I 三队均取得 12 分积分; 由于赛制规定 I 组无法晋级; 本区不甚喜欢圈地，且注意 G 组的各局圈地差距极大，是各图中唯一一张需要坐标轴负值标识的组：



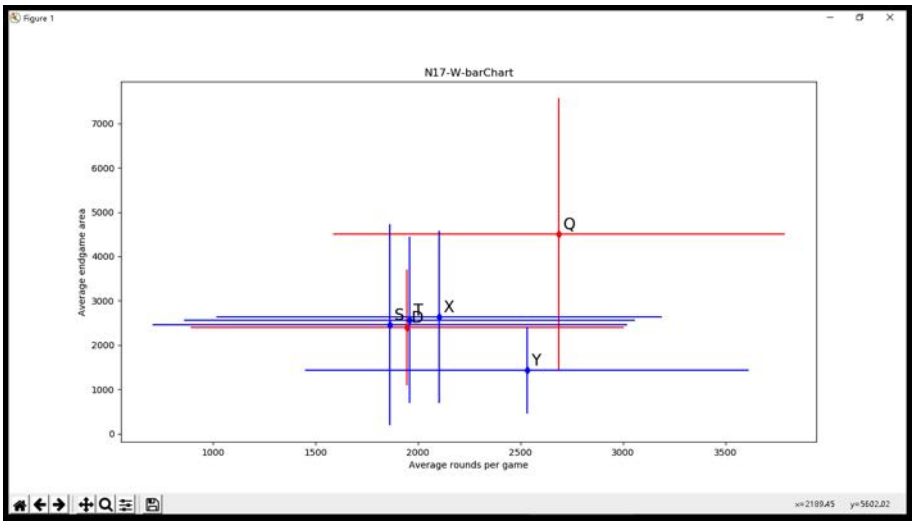
N17 分区 N 的各位喜欢圈地，局均回合数显著长于其他分区：



N17 分区 S 也是不喜欢圈地的组，其中由这里面最不喜欢圈地的 P 组取得小组第一，并于最终获得亚军：

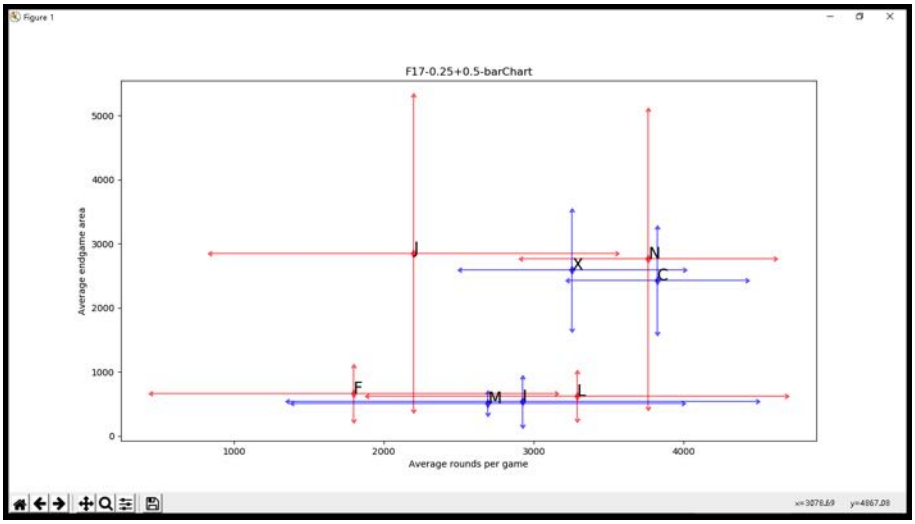


N17 分区 W 中独树一格的 Q 组以第二名出线并于最终取得季军，分区第一名则是与其他对手战得难分难舍且最后夺得冠军的 D 组：

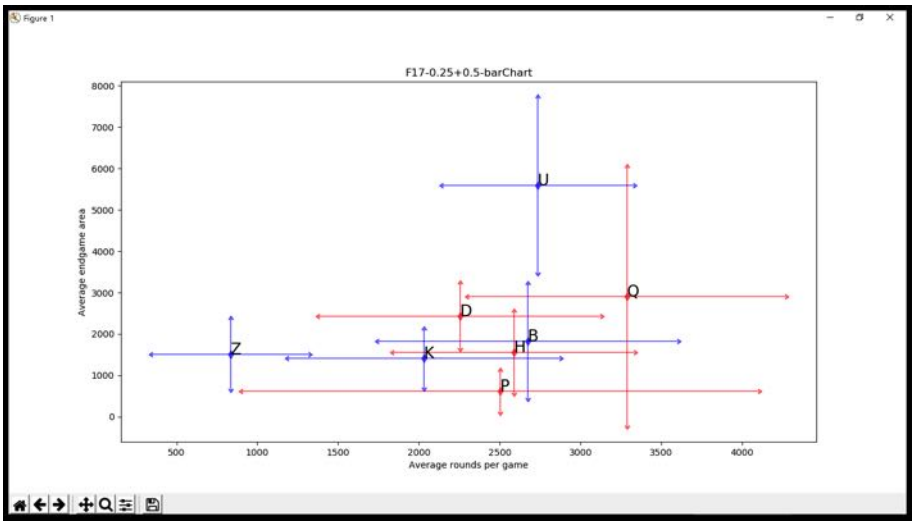


8.6 淘汰赛与半决赛得点分析

F17 中，能圈地的普遍成绩较好：



N17 中则以能持续战斗的参赛者取得较好成绩：



第九章　特色战局选讲

本部分将根据算法竞赛过程，选出几类兼具有趣、特别、有代表性但却不算罕见的几种竞赛过程类型，并对于各类型给出战局演示和解说。

由于总工作量过大，工作人员未能遍历所有竞赛结果；以下结果乃根据友谊赛、各类分析工作与战局记录抽查得到的总结，可能具有遗漏的情况，敬请见谅。

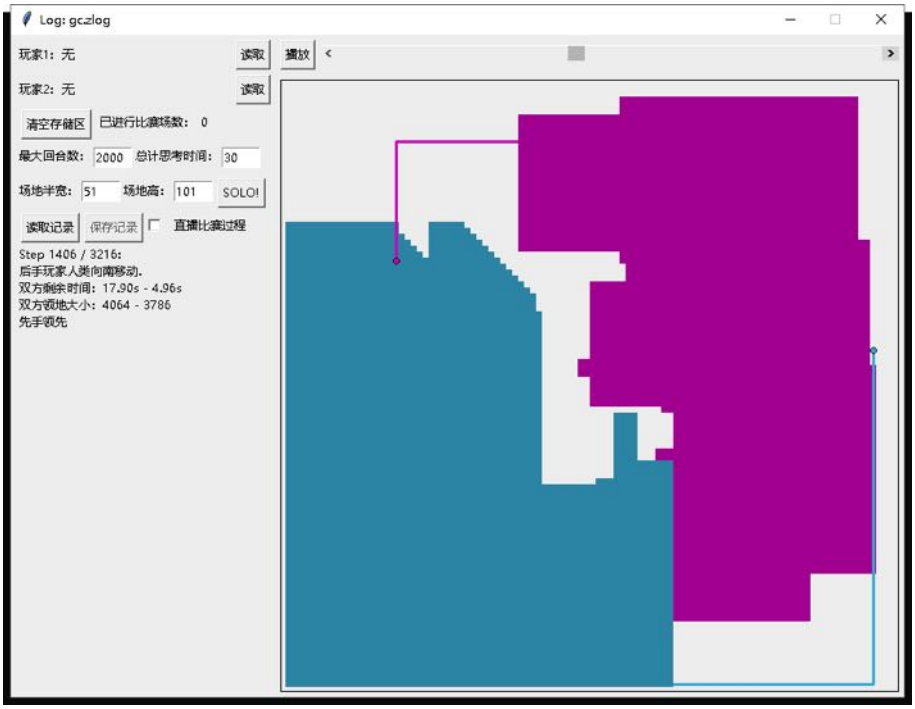
9.1　圈地为王

本类型为其中一方透过将对方领地完全包覆，致使对方失去所有领地的情况。

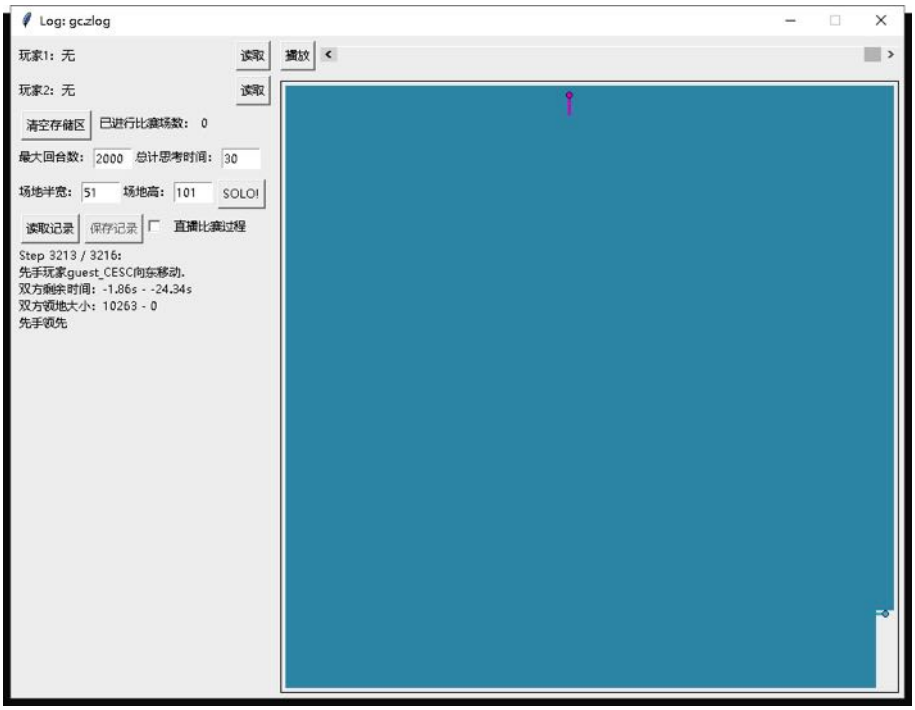
以下战局感谢北大城环数算课程提供参赛代码作为本战局先手 (此处以 CESE_guest 命名)，图上呈现颜色约为海蓝色；后手方为因为尴尬而不愿透露姓名的人类玩家，图上呈现颜色约为青莲色。

9.1.1　战局演示与解说

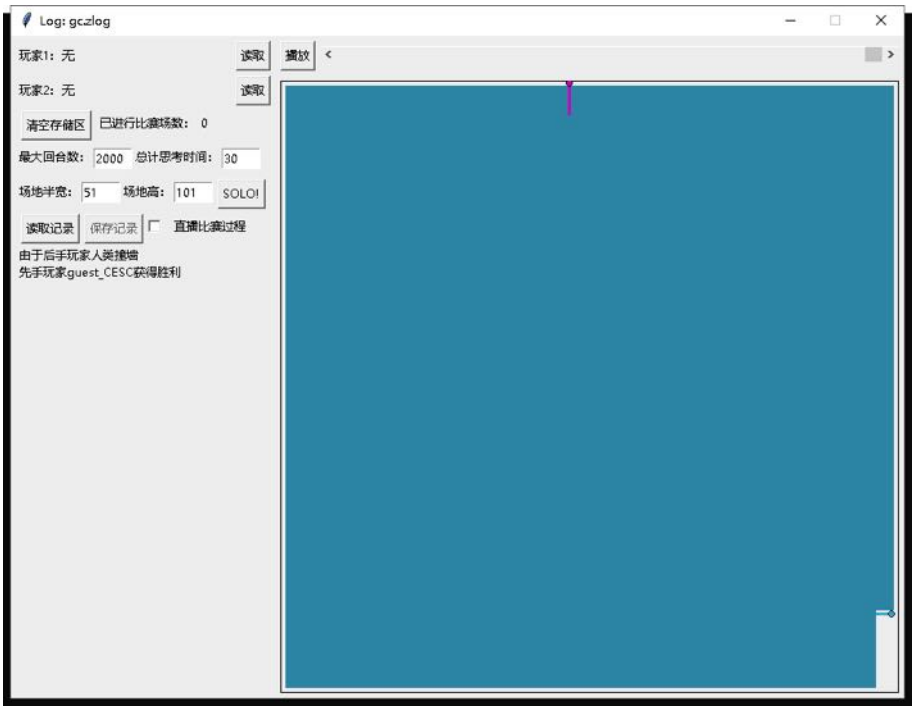
有的玩家一开始看起来很乖巧，好比这位海蓝色的玩家：



但是逮到机会就完全不给别人活路，一分地都不留：



青莲色的人类玩家内心大概是崩溃的，选择了一头撞上了豆 (bian) 腐 (jie)，结束了这样尴尬的局面：

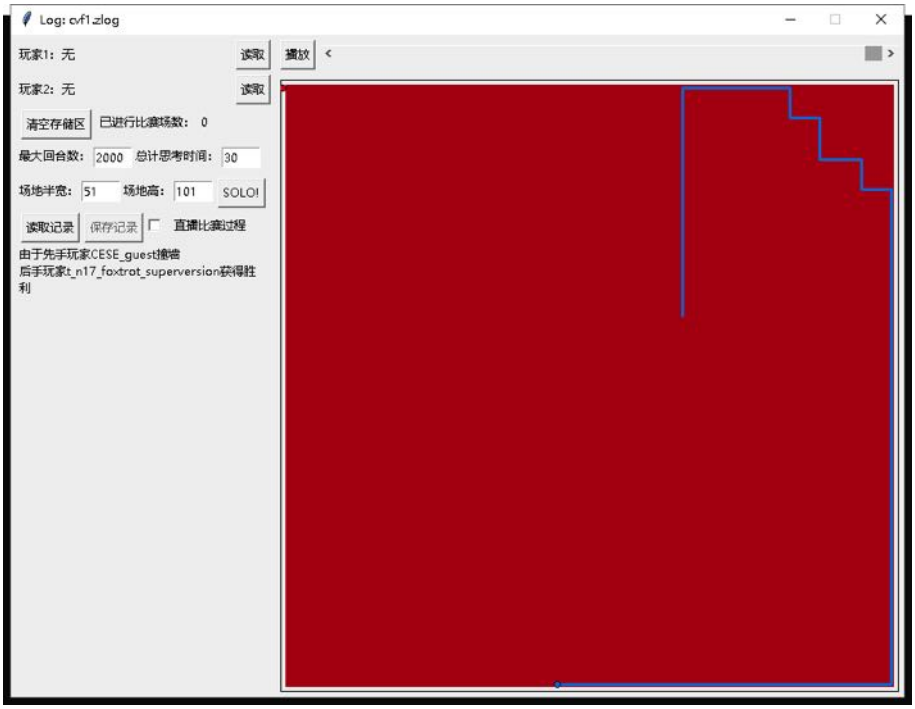


9.1.2 点评与附注

如果能够顺利达成此类情况，基本是必胜，仅仅需要在自己领地内游走，等待对方非自然死亡或游戏终局即可获得胜利——话虽是这么说，友谊赛内却存在这样一对选手：

先手圈起了全部 10302 个领地；此时后手非常紧张，由于失去了所有的领地，无家可归，只能在场地内无归属地游走，处于极度被动的地位。

先手估计是过于兴奋，把自己的脑 (zhi) 袋 (juan) 糊 (zhuang) 上了豆 (bian) 腐 (jie)，留下后手胜的局面、N 脸茫然的观众以及无限的遐想：



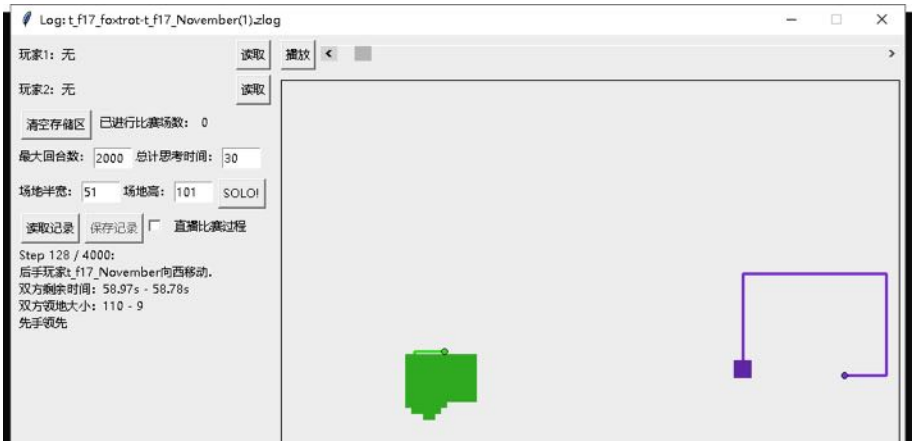
9.2 无限互博

本类型为双方由于算法设计上达到特定情况，双方距离非常近（四邻域意义下距离不超过四），且双方因为打算伺机攻击对方或不愿放弃该部分领地，致使双方在极小范围内反复游走的情况。

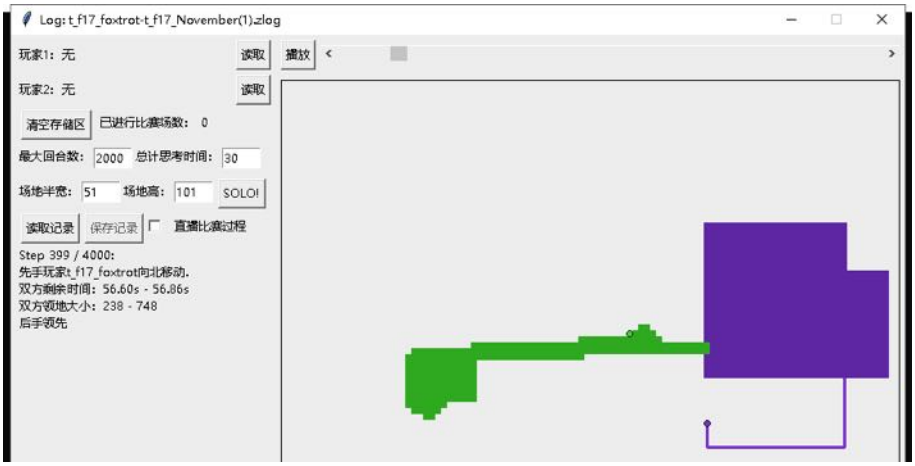
以下战局由草绿色的 F17F 为先手，楝色的 F17N 为后手。

9.2.1 战局演示与解说

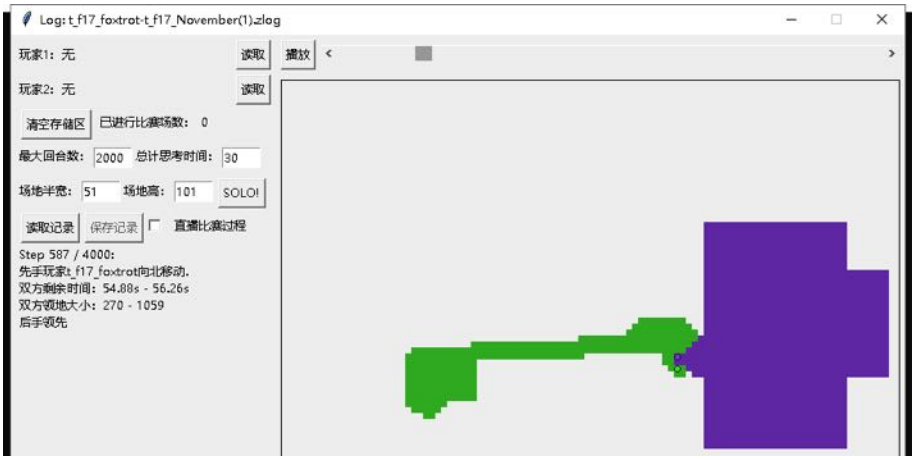
一开始看起来是一本正经的圈地：



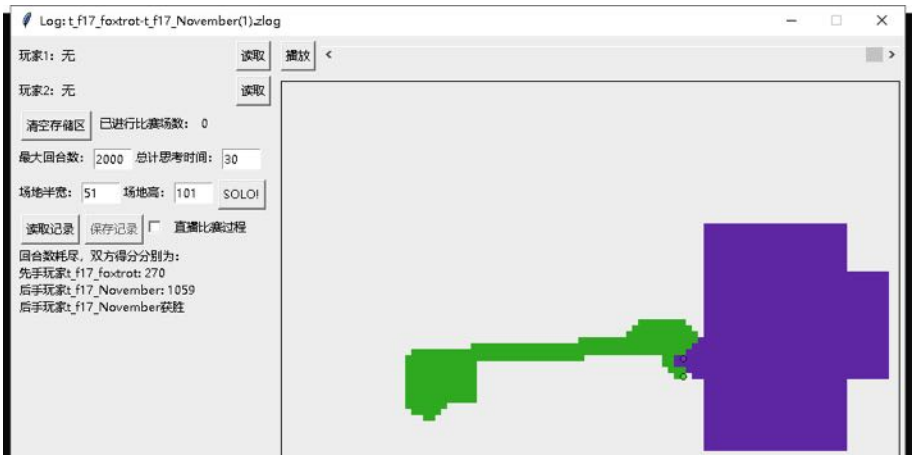
根据我们的了解，先手方 F17F 具有不算弱的攻击性；果不其然 F17F 主动靠近对方：



经过了一番近身格斗，领地分布变成了这样：

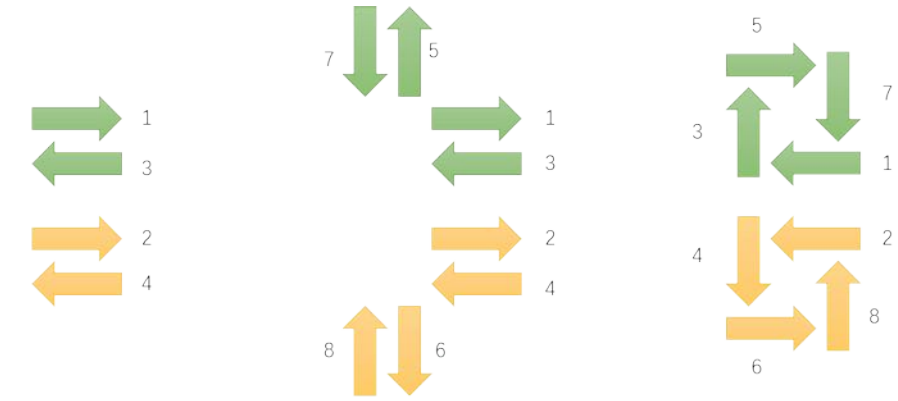


接着两位玩家持续了无止尽的循环，周期为 8（双方各 4 步），在观众看了 3400 多步的重复画面后，由系统宣布 4000 步耗尽，比赛结束：



9.2.2 点评与附注

本类型相对而言较为常见，主因是双方均打算伺机攻击对方或不愿放弃该部分领地，而进入此类循环，且一般以循环周期 4（反复横跳）或 8（无尽滑步、方形走位）显现：



反复横跳　　无尽滑步　　方形走位

另有其他周期，如反复横跳两单位长的 8 步周期、画 2*3 方格的 12 步周期等不同种类的情形; 容易知道，由于双方各需要移动偶数步以完成周期，此类周期应为 4 的倍数。

此类型若显现，以当时领地大者为胜（因后续无领地变更的机会），因此若领地大小处于劣势情形，宜避免此类型出现，可考虑放弃此小块领地，选择于其他区块占领无主区域，或趁对方不注意，抢占对方领地。

9.3 神仙打架

本类型为双方由于算法设计上达到特定情况，且均相对保守，时刻观察以下两个值：

1. 对方纸卷与己方纸带距离

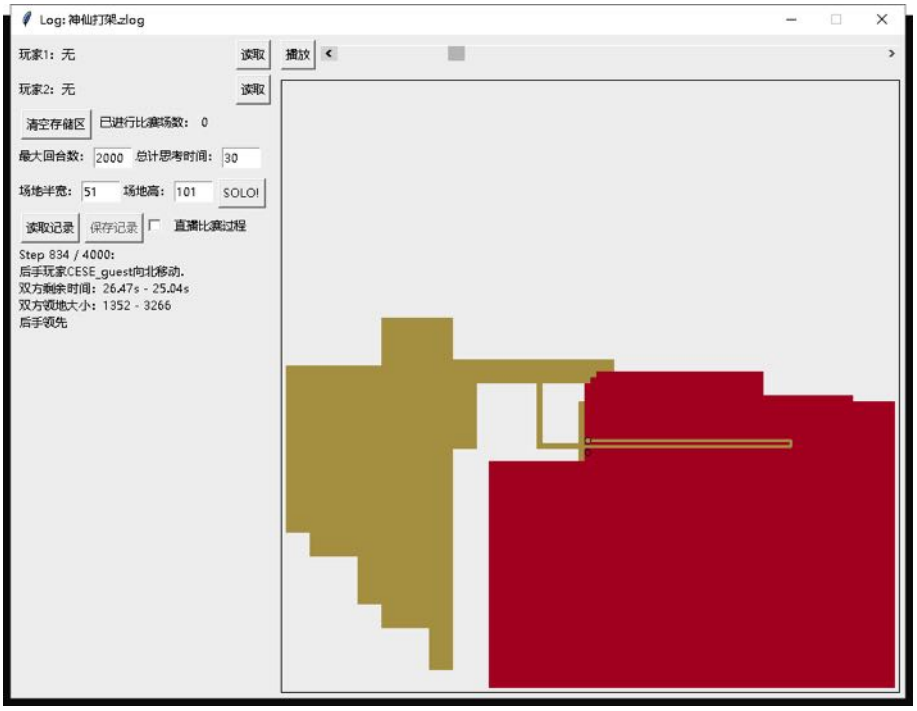
2. 己方纸卷与己方领地距离

并在此两个值达到警戒范围内，暂停当前想法，直接往己方领地前进的情况；由于轨迹多变，人类玩家多半不是看得很明白，并且由于相应的算法的游戏能力一般居于人类之上，而被人类称为神仙打架。

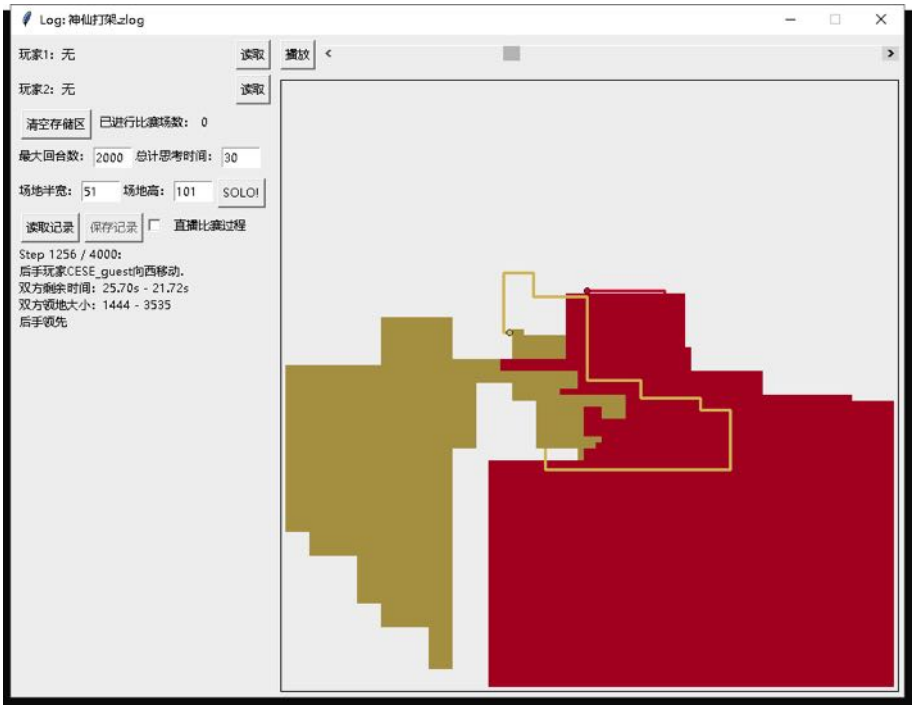
以下战局由沙色的 N17Fs 为先手，朱色的 CESE_guest 为后手。

9.3.1 战局演示与解说

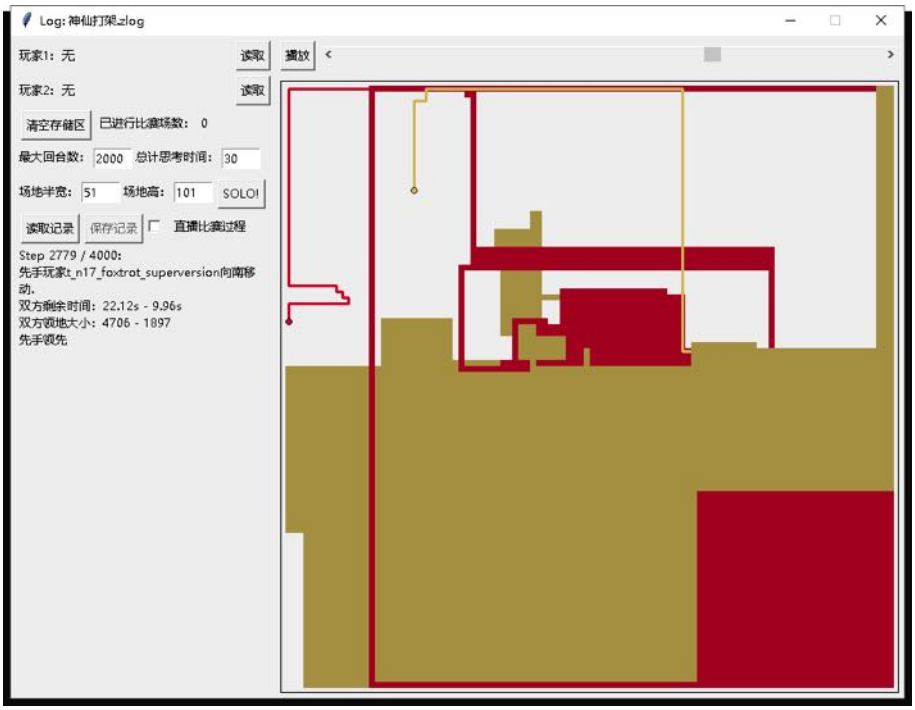
容易看得出来在几百步内双方已经有一些近距离格斗与抢占领地的情况：



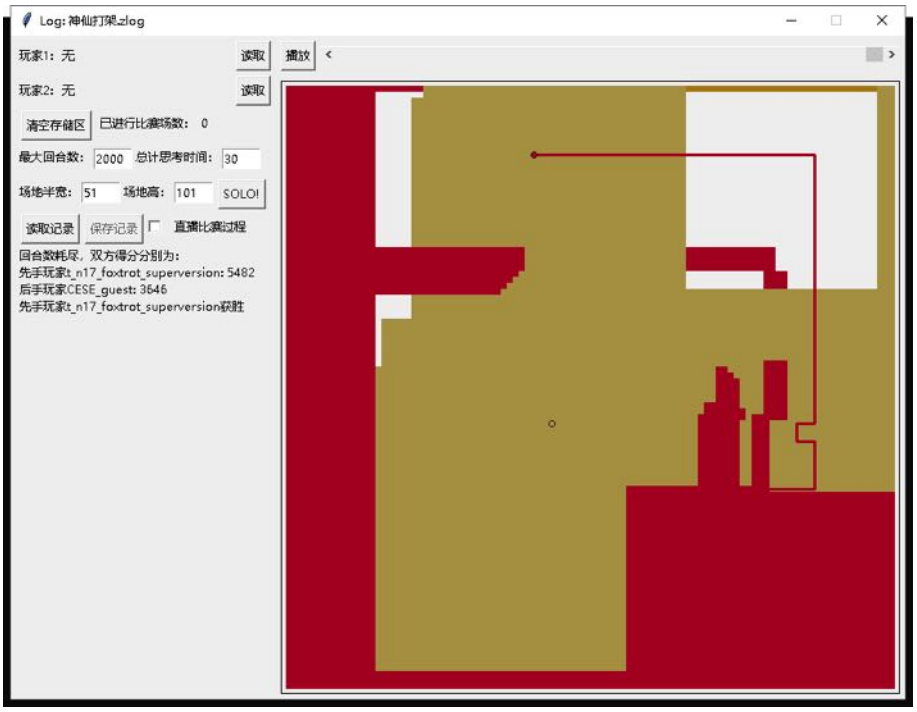
接着出现了一些谜一般的、贪心而胆小的轨迹：


Log: 神仙打架.zlog
玩家1: 无
读取
播放
玩家2: 无
读取
清空存储区
已进行比赛场数: 0
最大回合数: 2000
总计思考时间: 30
场地半宽: 51
场地高: 101
SOLO!
读取记录
保存记录
直播比赛过程
Step 1256 / 4000:
后手玩家CESE_guest向西移动.
双方剩余时间: 25.70s - 21.72s
双方领地大小: 1444 - 3535
后手领先

以及一些更迷幻的轨迹：


Log: 神仙打架.zlog
玩家1: 无
读取
播放
玩家2: 无
读取
清空存储区
已进行比赛场数: 0
最大回合数: 2000
总计思考时间: 30
场地半宽: 51
场地高: 101
SOLO!
读取记录
保存记录
直播比赛过程
Step 2779 / 4000:
先手玩家t_n17_foxtrot_superversion向南移动.
双方剩余时间: 22.12s - 9.96s
双方领地大小: 4706 - 1897
先手领先

最终由于后手没圈回去，回合耗尽，先手因占地较大而获胜：



9.3.2 点评与附注

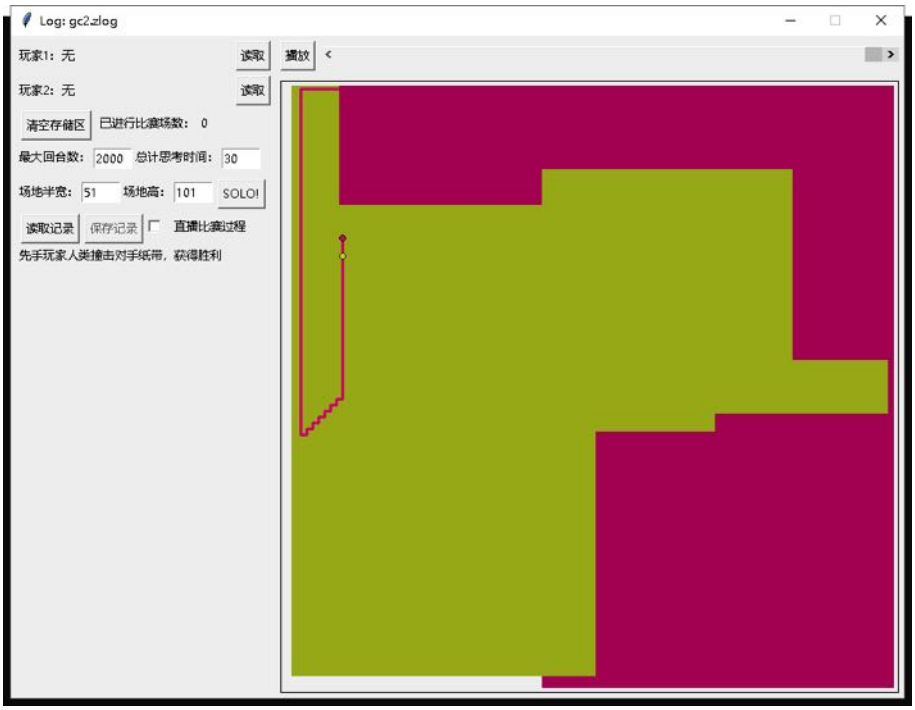
深究算法的话其实算是非常合理的选择，毕竟大家都想圈大的领地，只要不死肯定是越大越好; 然而对方也是这么想的，于是有一群人压在边界上，导致了虽然走过的轨迹很长但是却未能形成封闭图形，以致领地狭长而零碎。

以上述对局为例，后手大约再要 15 步能圈一个大的领地，并且如果能允许走这么多步的话，先手完全无法攻击后手，只能被圈在里面，之后甚至有被攻击致死的可能性; 整体而言后手算是输得相当可惜，这也告诉我们贪心还是有显著的缺点，特别是没有考虑到回合数等条件限制的情况。

9.4 人机对战结果

上部分也提到，人机对战中，若使用相对强的算法（本次人机对战中实际使用的有以下四个代码：N17Fs, F17N, N17D, CESE_guest）作为对手，人机对战中人类多半处于不利情况。

人机对战环节在数十局对局之后，人类终于拿下了其中一局，获得所有观众的喝彩与奖品一套：



第十章　结语

10.1　致谢

感谢本课程授课老师陈斌老师进行技术指导; 感谢北大城环数算课程的代码编写者提供参赛代码; 感谢本课程助教陈旭, 冀锐, 缪舜, 滕沅建, 易超, 张颢丹, 赵宇等提供协助。

10.2　设想与展望

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

10.3　主要参考资料

1. Flood fill, Wikipedia, `https://en.wikipedia.org/wiki/Flood_fill`, 2018-05-21
2. Matplotlib: Python plotting —Matplotlib 2.2.2 documentation, `https://matplotlib.org/`, 2018-06-01
3. 17.2. multiprocessing —Process-based parallelism —Python 3.5.5 documentation `https://docs.python.org/3.5/library/multiprocessing.html`, 2018-05-26
4. 数据结构与算法:Python 语言描述, 裘宗燕著, 北京: 机械工业出版社,2015 年 12 月
5. 25.1. tkinter —Python interface to Tcl/Tk —Python 3.5.5 documentation, `https://docs.python.org/3.5/library/tkinter.html`, 2018-05-24

第二部分

N17 联盟各参赛队报告

第一章　N17_Alpha 报告

王子瑜 * 马佳萌 蔡兴瑞 徐弘笛 金梦婷

摘要：函数运行是需要进行多次递归调用，分析目前盘面所处于的模块，然后运行对应模块的操作即可。作为一种思路的铺垫，Alpha-Beat 剪枝对博弈树可以对多步进行筛选，即设定参数 alpha、beta 分别为我方效用最大值和对方效用最小值，每方提出策略后使用简化版的结算函数进行盘面结算，计算双方效用并更新 alpha 和 beta。两者运用到了线性数据结构和树形数据结构。实验结果显示，模块操作可以有效分析目前场上的局势，Alpha-Beta 剪枝博弈树能动态计算出最优的行径路线。

关键字：函数模块操作 Alpha-Beta 剪枝博弈树，线性数据结构，树形数据结构

1 算法思想

1.1 总体思路

本次比赛最终提交的代码的总体思路是尽可能多地圈地，具体实现是分阶段进行游戏，在第一阶段将自己身后的地分四次圈起来（保证自己安全的情况下），在第二阶段从北向南或从南向北走小长方形逐渐圈地。该代码采取的数据结构主要是字典和列表，用于存储必要的信息以及判定是否安全等。

我们最初的想法是使用暴力搜索的办法，全局搜索不太可能实现，因为时间过长；但是如果是局部搜索，又可能因为敌方离我们太远而没有任何意义。所以我们之后形成了执行不同阶段策略的想法。我们分析得出结论：这个游戏前期重在圈地，后期要进攻与防守兼考虑。想要尽可能高效地圈地，要注意到高效和安全这两者呈反向关系，最高效的方法

就是沿着整张地图的边缘走，可是这也是最不安全的方法。所以我们必须在高效和安全之间做出权衡，采取了先将自己初始领地后方区域围起来的策略。然后我们需要进一步扩大领地，这个时候需要更加注重安全，所以效率上有所折扣，这一阶段采取的策略是从北向南或者从南向北不断地圈小矩形，进一步扩大自己的领地，同时需要防止敌人过来抢地。最后应该还需要有敌人和我方距离较近的时候的局部最优策略，从而在何种对峙阶段也能够攻守自如。

前期构想思路（备选方案）：Alpha-Beta 剪枝

首先外生给定我方、对方 AI 具有基本的自主意识，不会自动撞墙，从而给出在任意局势下某 AI 的可能行动集。然后采用了博弈树的思想：第 1 层为我方的可能行动，第 2 层为对方的可能行动，以此类推。递归深度为总层数，相当于预计多少“步”之后的盘面局势，并使用一定的效用函数对这种盘面局势进行估值，找出本回合中我方的最优方案。其中效用函数的规定为人工启发，结合小组成员在实战中的经验通过对双方面积、双方距离、距我方/对方领地距离、一方距另一方纸带距离、是否在我方/对方领地内、领地位置进行量化并加权求和，得出任一盘势的效用，并规定取得胜利的效用为无穷大，被战胜的效用为负无穷大。使用迭期望公式计算每种路径下的期望效用，计算在双方都具有不少于总递归深度阶理性的前提下每个子博弈完美均衡的均衡路径，并根据各期望收益选择出本回合最优行动。

为了节约运算时间，采取 Alpha-Beat 剪枝对博弈树进行决策筛选，即设定参数 alpha、beta 分别为我方效用最大值和对方效用最小值，每方提出策略后使用简化版的结算函数进行盘面结算，计算双方效用并更新 alpha 和 beta，若出现最大值小于最小值的情况就认为这种情况不会发生，从而避免了这之下的递归开销。

1.2 算法流程图

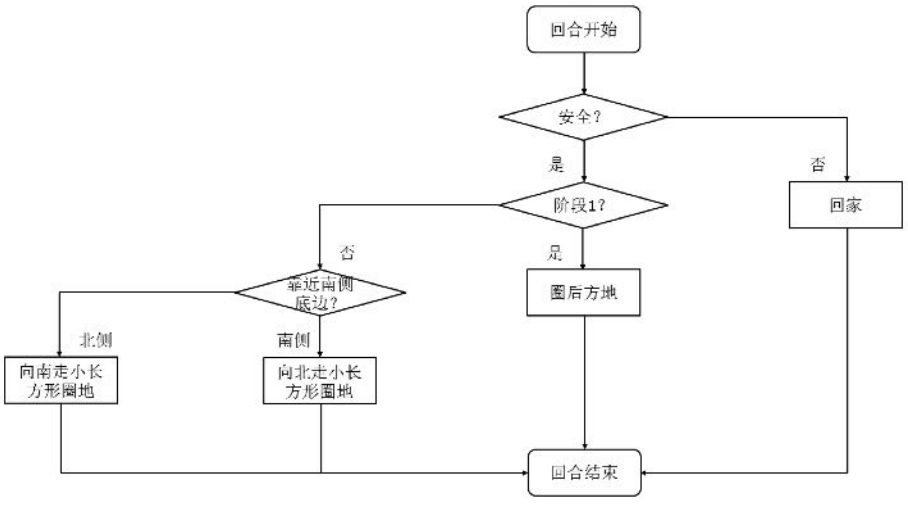


图 1 战斗函数

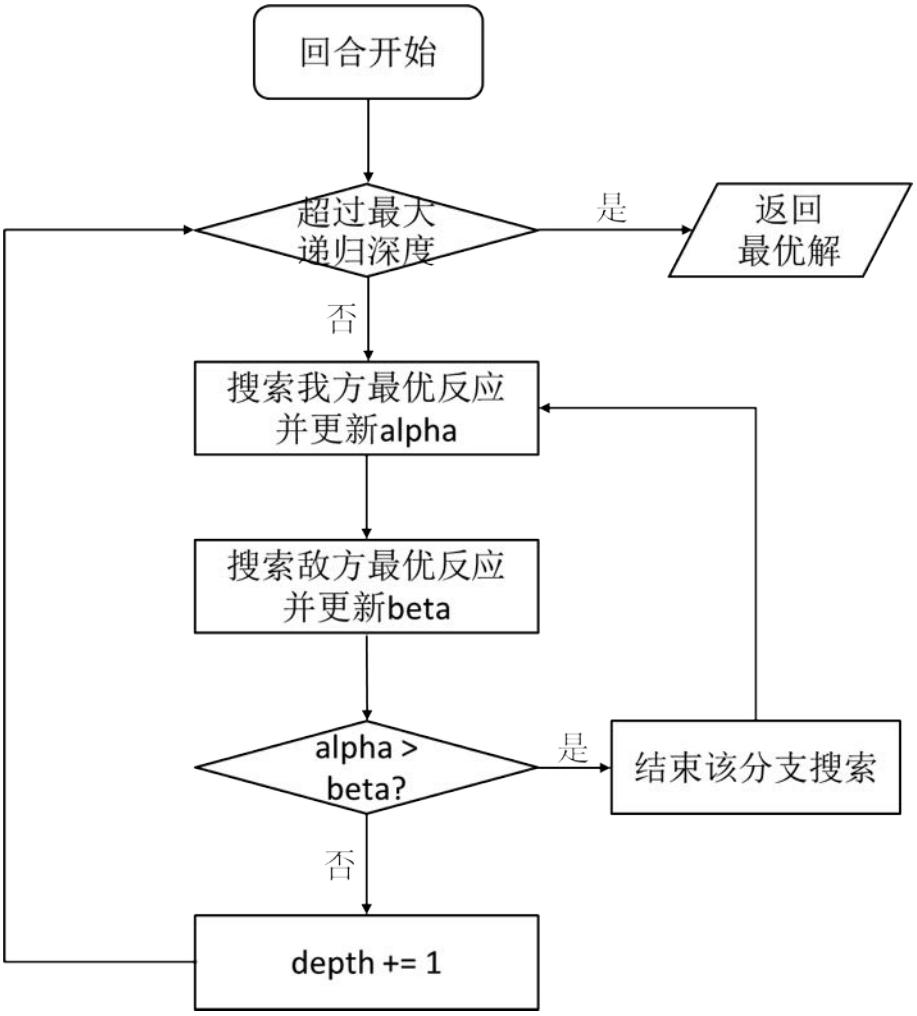


图 2 Alpha-Beta 剪枝博弈树算法流程

1.3 算法复杂度

算法运行时间的主要构成在于每次判定回家距离、敌人距离我的距离等等，因为都需要遍历一次棋盘，最终算法的复杂度是 O(n^2)。其他的部分都只是一些的条件语句，复杂度相对较低，为 O(1)。在实际运行过程中，我方走完 2000 步所花的时间仅 2-3 秒，远不及 60 秒的时间上限，不必考虑超时的问题。

而对于前期思考的 Alpha-Beta 剪枝的方法：

Alpha-Beta 剪枝算法用于减少搜索的节点数目，其效率很大程度依赖于节点的排列顺

序，理想条件下算法复杂度为 $O(b^{d/2})$，其中 $b(b \leq 3)$ 为经过可能性优化的枝条分叉数，d 为递归深度。递归深度越大，算法越智能，但是时间开销越大，所以搜索深度 d 会随局势不同进行调整。

2 程序代码说明

2.1 数据结构说明

战斗函数的算法中采取的数据结构主要是字典和列表，用于存储必要的信息以及判定是否安全等。而 Alpha-Beta 剪枝算法中采用线性数据结构和树形数据结构，并使用 Alpha-Beta 剪枝对树形数据结构进行优化。

2.2 函数说明

2.2.1 参数准备

在算法准备部分，初始化 storage 参数，以保证每一局开始前能够被调用。

```
def load(stat, storage):
    storage['first'] = True                 # 开局时调整位置方向
    storage['mark_x_1'] = None              # x坐标过程风向标
    storage['first_y'] = None               # y坐标过程风向标
    storage['decision'] = [True] * 100      # 流程判定
    storage['change'] = False               # 流程结束或开始
    storage['step'] = 0                     # 步数判定
    storage['inside'] = False               # 内外判定
```

```
def init(storage):
    storage['first'] = True                 # 开局时调整位置方向
    storage['mark_x_1'] = None              # x坐标过程风向标
    storage['first_y'] = None               # y坐标过程风向标
    storage['decision'] = [True] * 100      # 流程判定
    storage['change'] = False               # 流程结束或开始
    storage['step'] = 0                     # 步数判定
    storage['inside'] = False               # 内外判定
```

把所有的关键变量信息从 stat 中提取出来。包括棋盘的宽度（x 轴）、高度（y 轴），我方运行方向，敌我双方的坐标位置、领地占领情况等

```
chess_width = stat['size'][0]
chess_height = stat['size'][1]
my_name = stat['now']['me']['id']# = 1则为先手玩家 = 则为后手玩家
my_x = stat['now']['me']['x']# 我的坐标
my_y = stat['now']['me']['y']
my_direction = stat['now']['me']['direction']# 我的运行方向，是关键信息

enemy_name = stat['now']['enemy']['id']
enemy_x = stat['now']['enemy']['x']# 敌人的坐标
enemy_y = stat['now']['enemy']['y']
enemy_direction = stat['now']['enemy']['direction']

situation = stat['now']['fields']# 自己和敌方领地占领的情况
paper = stat['now']['bands']# 自己和敌人纸带占领的情况
```

2.2.2 判定行进过程有无危险

为了确保圈地过程中的安全，定义以下三个函数用来判定敌方能否击杀我方。首先，用 quick_home( ) 函数来遍历整个棋盘，求出当前位置回到自己领地的最快路径。分成在领地内、领地外两种情况，返回最短路径的长度 distance 以及应该回去的坐标点 target_x, target_y。

```
# 求当前情况回到自己领地的最快路径，返回值是回家的一个坐标点
    def quick_home(my_x,my_y,situation):
        # 如果在自己的领地内，就没有返回自己领地的最短路径，返回的坐标点设为None
        if situation[my_x][my_y] == my_name:
            return 0,None,None
        # 通过判定自己现在的位置回到自己领地的最短距离
        distance = None
        for x in range(0,chess_width):
            for y in range(0,chess_height):
                if situation[x][y] == my_name:
                    route = abs(my_x - x) + abs(my_y - y)
                    target_x = x
                    target_y = y
                    if distance == None or route < distance:
                        distance = route
                        target_x = x
                        target_y = y
        # 由于是不能直接180度掉头的，所以如果存在垂直领地出去的情况，回来需要多走一步
        # 保险起见，把回家的最短路径增加一个单位，确保不会被击杀
        shortest = distance + 1
        # 返回了最短路径的长度，以及应该回去的坐标点
        return shortest,target_x,target_y
```

为了在返回我方领地时实现效用最大化（圈出尽可能多的地），我们选择圈一个长方形的区域。因此，在判断敌方是否能够击杀我们时，不仅要判断敌方到我们现有纸带的距离（目前危险），还需要判断敌方是否到达我们返回领地的整条路径（安全隐患）。

在得知最短路径的基础上，分别利用 enemy_here( ) 函数和 safe( ) 函数，判断敌方到我方纸带和整条返回路径的最短距离 attack_route，以及当前位置是否安全。具体代码截图如下：

```
    def enemy_here(my_x,my_y,enemy_x,enemy_y,target_x,target_y):
        distance = None
        x1, x2 = sorted([my_x,target_x])[0], sorted([my_x,target_x])[1]
        y1, y2 = sorted([my_y,target_y])[0], sorted([my_y,target_y])[1]
        for x in range(x1,x2+1):
            for y in range(y1,y2+1):
                route = abs(my_x - x) + abs(my_y - y)
                if distance == None or route < distance:
                    distance = route
        attack_route = distance
        return attack_route

# 判断是否安全，是则返回True，不是则返回False
    def safe(my_x,my_y,enemy_x,enemy_y,situation):
        if situation[my_x][my_y] == my_name:
            return True
        tpl = quick_home(my_x,my_y,situation)
        shortest,target_x,target_y = tpl[0], tpl[1], tpl[2]
        attack_route = enemy_here(my_x,my_y,enemy_x,enemy_y,target_x,target_y)
        if shortest + 1 < attack_route:
            return True
        else:
            return False
```

以上三个函数将在每一局都被调用，确保我方的安全。

2.2.3 流程控制模块及对战初期策略

在对战的初期，我们认为由于双方距离较远、所属领地较少，敌方多半会采取保守策略，其进攻可能性较低。因此我方可以调整朝向（转到远离敌人的方向），先将自己身后的区域圈住，从而积累一定的优势。此阶段把身后领地均分为四块长方形，每次贴着棋盘的边界行走，圈出其中的一块长方形。由于每一阶段的任务不同，为算法添加一个便捷的“流程判定小模块”。change_value( ) 函数的功能就是判断下一步操作是否进入下一环节中。

```
def change_value(k):
    storage['change'] = True
    storage['step'] = k

# 如果change_value函数判定要进入下一个环节，在第二步的时候就会执行如下代码
# 函数的流程也会进入下一个阶段
if storage['change'] == True:
    number = storage['step']
    storage['decision'][number] = False
    storage['change'] = False
```

2.2.4 对战后期圈地策略

在对战的第二阶段，由于身后的地基本上都已经圈成功，所以开始逐渐朝向另一半（敌方的地盘）圈地。例如东面的地都被圈起来了，那么进攻方向就是西方；如果西面的地都被全起来了，那么进攻方向就是东方。算法的思路是不断向外圈出一个个小长方形（从背面到南面一路朝进攻方向圈小长方形，或者从西面到面一路朝进攻方向圈小长方形）。

首先在 storage 中储存一系列关键变量的值，利用字典来实现。‘count’表示向外进攻的步长为 5，即每次圈出的小长方形的一条边长；‘y_now’记录了当前纵坐标值；‘go_down’和‘go_up’用于判定是否处于由北向南、由南向北不断向外圈地的情况；‘inside’用于判定是否在一次进攻中已经回到过领地，可以辅助判定下一次进攻中的细节；‘safe’用于判定目前是否处于安全状态；而‘count_2’对旋转 180 度朝向的进攻方向细节进行控制。

```
while storage['decision'][20]:

    # 圈地完成，为第二阶段的一些变量赋值做初始准备
    # 在storage中储存一系列关键变量的值
    storage['count'] = 5
    storage['y_now'] = my_y
    storage['go_down'] = True
    storage['go_up'] = False
    storage['inside'] = False
    storage['safe'] = True
    storage['count_2'] = 0

    # 下一步进入下一个环节
    change_value(20)

    # 首先转向进攻方向
    if (my_direction+1) %4 == storage['attack_direction']:
        return "r"
    else:
        return "l"
```

然后在判定不出界、不受敌方威胁的前提下，不断向进攻方向圈出小块长方形的地（分成由北向南和由南向北两种行径方式，不断增加或者减小纵坐标的值）。一旦判定出危险，safe 的值就变成 False，并让坐标点在自己的领地内部直走。

2.2.5 Alpha-Beta 剪枝——一种思路构想

```
def getAction(state):
    # 函数中playerId 规定为1,2
    # 下一个状态可能的行动
    legalActions=getLegalActions(stat, 0)

    maximum=float('-Inf')
    # 初始化alpha beta
    alpha=float('-Inf')
    beta=float('Inf')
    maxAction=''
    #规定搜索深度
    depth = 10

    #个性化stat
    state = stat
    #写入field_border
    field_border = [None, [602, 0, 101, 0], [602, 0, 101, 0]]
    for i in range(602):
        for j in range(101):
            if field[i][j] == 1:
                if i < field_border[1][0]:
                    field_border[1][0] = i
                if j < field_border[1][2]:
                    field_border[1][2] = j
                if i > field_border[1][1]:
                    field_border[1][1] = i
                if j > field_border[1][3]:
                    field_border[1][3] = j
            elif field[i][j] == 2:
                if i < field_border[2][0]:
                    field_border[2][0] = i
                if j < field_border[2][2]:
                    field_border[2][2] = j
                if i > field_border[2][1]:
                    field_border[2][1] = i
                if j > field_border[2][3]:
                    field_border[2][3] = j
    state['field_border'] = field_border
    #写入band_direction
    band_direction = [None, [], []]

    enemyDirection = stat['enemy']['direction']
    for action in legalActions:
        currentDepth = 0
        currentMax = min_value(generateSuccessor(1, state, action, enemyDirection)\
                               , enemyDirection, alpha, beta, currentDepth)
        if currentMax > maximum:
            maximum = currentMax
            maxAction = action
    #print maximum
    return maxAction
```

getAction() 是主函数体，输入量为当前局势，规定了在计算最优解中需要的参数，包括 alpha，beta，depth 等，并将 stat 中的数据格式个性化为其他函数便于使用的格式，调用递归函数返回值为最优解。

```
def getLegalActions(state, playerId):                                    #这里认
    legalActions = []
    x, y = state[players[playerId]]['x'], state[players[playerId]]['y']
    # 方向数字对应前进方向
    directions = [(1, 0), (0, 1), (-1, 0), (0, -1)]
    for direction in directions:
        # 移动
        x += direction[0]
        y += direction[1]
        # 边缘检测
        if x >= 0 and x < WIDTH and y >= 0 and y < HEIGHT:
            legalActions.append(direction)
```

getLegalAction() 默认任何一方 AI 不会自动撞墙，从而缩小博弈树分支数量。

```
def min_value(state, enemyDirection, alpha, beta, currentDepth, depth):
    if state == True:
        return float('Inf')
    if state == False:
        return float('-Inf')
    # 初始化v
    v=float('Inf')
    # 对每个max分支求min 其中有多个Ghost 所有多个Ghost分支
    for pAction in getLegalActions(state, 1):
        v = min(v, max_value(generateSuccessor(2, state, pAction, enemyDirection)\
                             , pAction, alpha, beta, currentDepth, depth))
        # 若比alpha还要小了 就没搜索的必要了
        if v <= alpha:
            return v
        # 更新beta的值
        beta = min(beta, v)
    return v

def max_value(state, enemyDirection, alpha, beta, currentDepth, depth):
    if state == True:
        return float('Inf')
    if state == False:
        return float('-Inf')
    # 当前深度加一
    currentDepth = currentDepth + 1
    # 若当前状态已经赢了或输了 或者 已经到达了规定的深度
    if currentDepth == depth:
        return evaluationFunction(state)
    v=float('-Inf')
    # 对每个min分支求max
    for pAction in getLegalActions(state, 0):
        v = max(v, min_value(generateSuccessor(0, state, pAction, enemyDirection),
        # 若已经比beta要大了 就没有搜索下去的必要了
        if v >= beta:
            return v
        # 更新alpha的值
        alpha=max(alpha, v)
    return v
```

实现 Alpha-Beta 剪枝的代码块。

```
def generateSuccessor(playerId, stat, pAction, enemyDirection):

    #初始化state
    state = stat

    #读取state
    x, y = state[players[playerId]]['x'], state[players[playerId]]['y']
    fields, bands = state['fields'], state['bands']
    enemy_x, enemy_y = state[players[playerId]]['x'], state[players[playerId]]['y']
    fields, bands = state['fields'], state['bands']

    # 移动
    x += pAction[0]
    y += pAction[1]
    '''                                                    如果觉得对方可能撞墙，请使用
    # 边缘检测
    if x < 0 or x >= WIDTH or y < 0 or y >= HEIGHT:
        return False  # 撞墙死
    '''

    # 更新占有区域边界
    state['field_border'][playerId][0] = min(x, state['field_border'][playerId][0])
    state['field_border'][playerId][1] = max(x, state['field_border'][playerId][1])
    state['field_border'][playerId][2] = min(y, state['field_border'][playerId][2])
    state['field_border'][playerId][3] = max(y, state['field_border'][playerId][3])

    # 更新玩家碰撞
    if enemy_x == x and enemy_y == y:
        # 自己领地内击杀对手
        if fields[x][y] == playerId:  # 自己领地
            return True

        # 对手领地内送分
        if fields[x][y] == 1 - playerId:
            return False

        # 侧碰击杀对手
        if enemyDirection % 2 != pAction % 2:
            return True

        # 无人领地，计算面积
        return count(fields)[playerId] > count(fields)[1-playerId]
```

```
    # 更新纸带碰撞
    if bands[x][y] == playerId:
        # 撞自己的纸带
        return False
    elif bands[x][y] == 1 - playerId:
        # 撞别人的纸带
        return True

    # 更新场地
    state = updateField(playerId, fields, x, y)

    return state

def count(playerId, fields):
    res = [0,0]
    for x in range(602):
        for y in range(101):
            if fields[x][y] is not None:
                res[fields[x][y] - 1] += 1
    return res
```

```
def updateField(playerId, fields, x, y):
    '''更新玩家在场地引起的效应'''
    # 场地内检查填充情况
    if fields[x][y] == playerId:  # 在/回到自己场地
        if self.band_direction:  # 在回领地瞬间填充领地
            self.check_field_fill()

    # 场地外延伸纸带
    else:
        self.extend_band()

def extend_band(self):
    '''（在自己领地外）延伸纸带'''
    self.band_direction.append(self.direction)
    BANDS[self.x][self.y] = self.id

def check_field_fill(self):
    '''（返回自己领地时）检查填充'''
    # 1. 纸带转换为领地
    next_step = self.directions[self.direction]
    ptrx, ptry = self.x - next_step[0], self.y - next_step[1]  # 从上一步纸带位置回
    while self.band_direction:
        BANDS[ptrx][ptry] = None
        FIELDS[ptrx][ptry] = self.id
        next_step = self.directions[self.band_direction.pop()]
        ptrx -= next_step[0]
        ptry -= next_step[1]

    # 2. floodfill填充空腔
    targets = set(
        (x, y)
        for x in range(self.field_border[0], self.field_border[1] + 1)
        for y in range(self.field_border[2], self.field_border[3] + 1)
        if FIELDS[x][y] != self.id)
    while targets:
        iter = [targets.pop()]
        fill_pool = []
        in_bound = True  # 这次填充是否为界内填充
        while iter:
            curr = iter.pop()
            # floodfill
            for dx, dy in self.directions:
                next_step = (curr[0] + dx, curr[1] + dy)
                if next_step in targets:
                    targets.remove(next_step)
                    iter.append(next_step)

            # 判断当前点
            if in_bound:
                if curr[0] == self.field_border[0] or curr[0] == self.field_border[
                    in_bound = False
                else:
                    fill_pool.append(curr)

        # 若未出界则填充内容
        if in_bound:
            for x, y in fill_pool:
                FIELDS[x][y] = self.id
```

在比赛核心文件 match_core 的基础上进行删减的结算函数，用于模拟在某一方做出某行动后产生的新局势。

2.3 程序限制

首先，由于本算法假定在对战初期敌方进攻的可能性较低，我方可以调整朝向并将自己身后的区域都圈住。这是一种相对保守的策略，无法适用于所有情况。如果敌方恰好选

择了激进的“追杀”方式，我们就会处于十分被动的地位（一直在逃避敌人而无法高效率圈地）。

其次，在“2.2.2 判定行进过程有无危险”中，针对“返回领地的最短路径长度”以及“敌方击杀我方的最短路径长度”的算法不够完善。我们出于时间复杂度的考虑，采取了更易计算的二维平面两点的切比雪夫距离公式。但在实际情况中，最短路径可能比该值要大，譬如在返回领地时遇到障碍（自己的纸带），需要绕行躲避。对最短路径的估计偏差可能导致我们在与敌方近距离对抗时，低估了被击杀的风险。

3 实验结果

3.1 测试数据

一、实验环境说明：

- 硬件配置：（CPU/内存）Intel®Core™ i5-5200U CPU @ 2.20GHz (4 CPUs) 内存 8GB
- 操作系统：（名称/版本）Windows 7 旗舰版 64 位
- Python 版本：（版本号）3.6

二、测试方法

1、使用简单陪练算法 goround，测试初始圈地代码的可用性

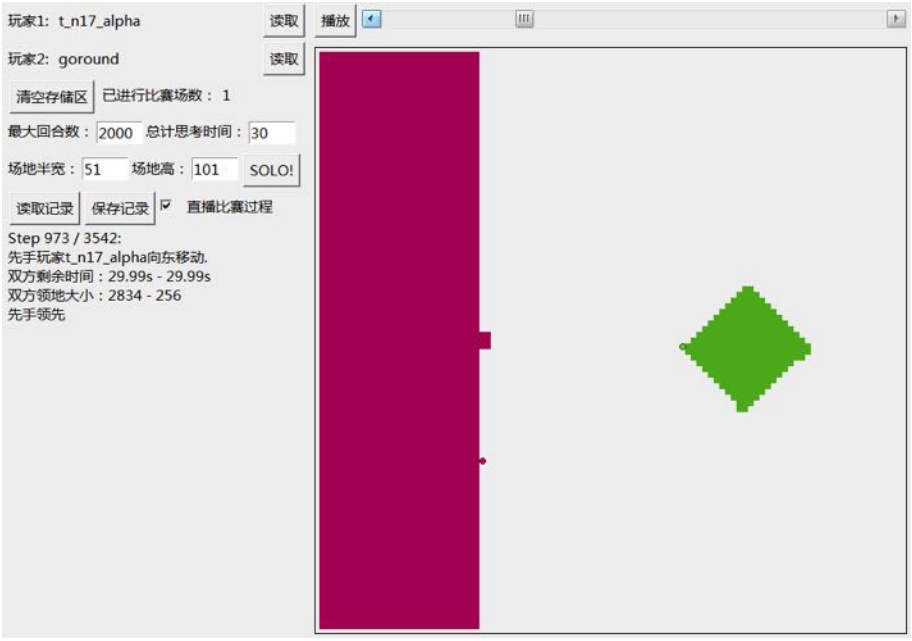
2、与技术组编写的 normal_wanderer 算法对战

3、小组循环赛，观察复盘数据

3.2 结果分析

1、根据小组决定的策略，我们的 AI 会首先在保证安全的前提下圈起在初始点所在垂直线到侧边边界的所有场地。在编写这部分代码的过程中通过一个简单的螺旋 AI 来测试可用性，最终实现了这样的圈地策略，并且能在每次对战中获胜。可以看出这样的圈地策

略效率大大高于简单的螺旋算法，并且在前期固定策略中用时甚至少于螺旋算法，后续加入条件判断后思考时间会有所增加。

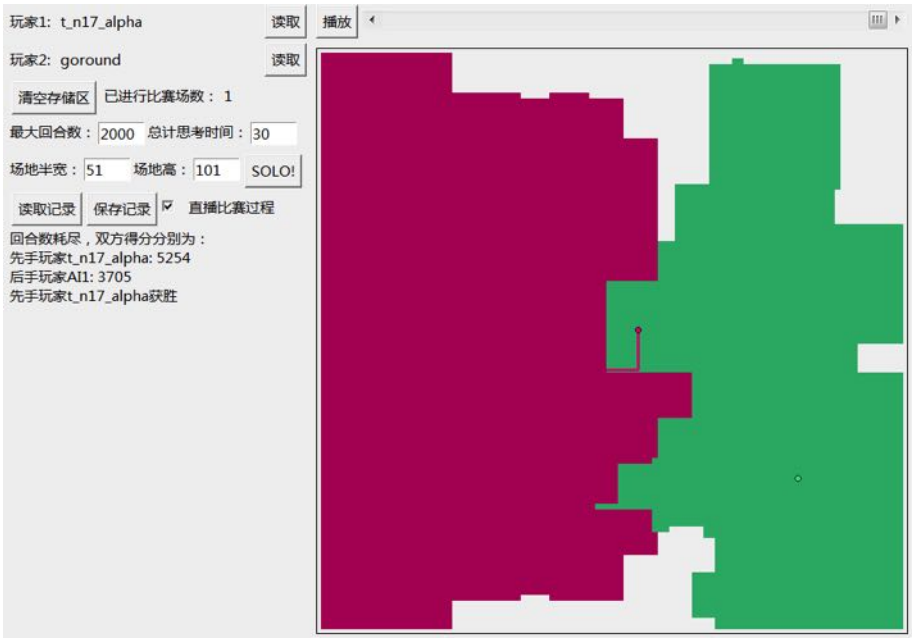


2、在和 normal_wanderer 的对战测试中，我们的算法在大多数情况下也可以取得胜利。在前半部分我们保持了高效率的圈地，可以领先对方 1000 格左右，并且在对方圈起部分靠近中央的领地再去圈边缘的地区时，可以趁机占领对方领地，使相对圈地效率提高一倍，往往可以在回合数耗尽时凭借领地面积优势获胜。

3、在小组循环赛中，我们的算法战胜了同小组的 golf，但在与其他小组的对战中遗憾落败，后续会在经典战局中选择案例进行详细分析。

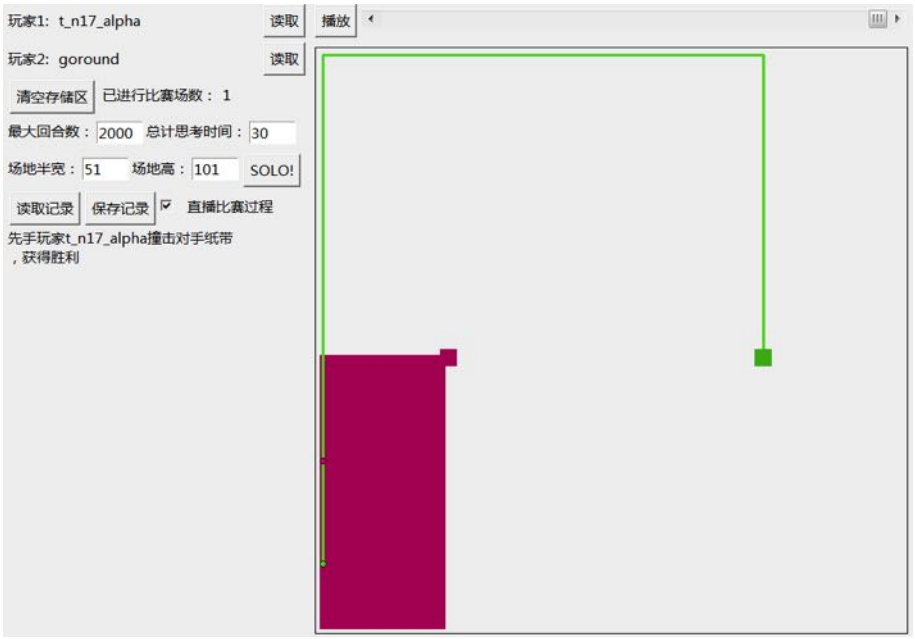
3.3 经典战局

1、N17_alpha vs normal_wanderer



在和 normal_wanderer 的这一局对战中回合数耗尽我方占地面积更大获胜，当对方的打法不太激进的时候，我们的策略能达到更大的圈地效率，从而占据优势。

2、N17_alpha vs N17_golf



Golf 组的策略主要是绕场一圈试图让对手无路可走，但由于我们的算法是要先圈后方的区域，所以这样做不可避免地要经过我方的领地，此时可以在自己领地内撞击对方纸带获胜，可以看出 golf 组的策略很有野心，但缺乏灵活性。

4 实习过程总结

4.1 分工与合作

一、小组成员：

王子瑜 *，徐弘笛、蔡兴瑞、金梦婷、马佳萌

二、合作交流方式：

1、在农园、最美时光、教学楼等地召开小组会议，集中讨论分析；

2、上课前后对有关问题进行交流；

3、微信群线上讨论，资料分享等。

三、分工：

1、策略设计：全体成员讨论制定；

2、程序开发：王子瑜同学负责主要代码的撰写，徐弘笛同学负责距离计算和危险情况判定部分，蔡兴瑞同学对其他的思路进行了探索，开发了一个独立的策略。

3、小组报告：蔡兴瑞同学负责算法思想和程序代码说明，徐弘笛同学和金梦婷同学负责程序代码说明，马佳萌同学负责实验结果和实习过程总结部分。

四、组会记录：

1、5 月 21 日

小组成员通过课程群互相认识，并由王子瑜同学担任组长建立小组微信群，Alpha 小组正式成立！然后大家商定各自熟悉竞赛环境和规则，尝试微信小程序中的“围城大作战”体会游戏过程。

2、5 月 29 日 14:00 理教讨论区

与会人员：王子瑜，徐弘笛、蔡兴瑞、金梦婷、马佳萌

第一次组会首先对整个大作业的基本情况进行了交流，然后大家分享了自己的想法和策略设想，将代码实现分成了进攻与防守两大部分，并且基本确定前期专注提高圈地效率，然后在占据领地面积优势的情况下找机会进攻，为整个大作业的实现奠定基调。

3、6 月 1 日 20:00 农园咖啡厅

与会人员：王子瑜，徐弘笛、蔡兴瑞、金梦婷

将策略分为几个阶段，开始撰写第一阶段圈起后方领地的代码，并对进攻和防守的具体情况进行了展开讨论。

4、6 月 2 日 9:00 最美时光

与会人员：王子瑜，徐弘笛、金梦婷、马佳萌

大家对防守部分代码编写时遇到的问题进行了讨论，设想了各种可能的情况，最终决定穷举判断的方式不太可取，暂定一个动态判断的方法。

5、6 月 2 日 20:30 农园咖啡厅

与会人员：王子瑜，徐弘笛、马佳萌

完成了第一阶段圈地策略的代码实现，并可以完全战胜螺旋圈地的策略。确定了下一阶段的走位策略，即身后领地圈好之后采用垂直进出领地的原则，通过对方到达自己领地的距离与自己回领地的距离比较判断危险，同理判断是否可以进攻。此次组会之后大致进行了代码撰写分工，最终完成了全部程序的编写。

4.2 经验与教训

这次大作业的完成是小组成员齐心协力的结果，从最初的头脑风暴，到一步一步形成丝路框架，再到具体的判断细节，每一次讨论都是激烈的思想碰撞，大家都从彼此身上学到很多，整个过程也很愉快而难忘。尽管在从无到有的探索中遇到了很多策略设想和具体实现上的问题，我们都可以做到积极贡献，排除困难，不断改进我们的程序。总体而言，我们先“发育”再“输出”的策略应该是比较稳定的做法，能够尽量先保持占地优势，然后再寻求时机攻击对方。另外从比赛结果和测试结果上看，我们的算法时间控制还比较好，首先没有出现过超时的情况，在很多情况下用时还短于对方。在具体情况判定上我们做到了尽可能设想到更多的情况，权衡利弊找到更合适的打法，为整个算法的可行性提供了保证。

当然，在完成整个大作业的过程中我们小组也有一些做的不完美的地方。例如最终的代码对于情况的判断不够完全，在第二阶段圈地的过程中返回领地的时候应当判断一下对手位置，以及返回路径中有没有可能经过对手领地，在这种情况下很有可能被对方击杀，在最终的复盘数据中也可以看到不少情况下都是因此落败。另外我们的算法非常注重圈地效率，但其实我们的设想也不够成熟，在后来的比赛中看到有其他小组先在一大片领地的外面拉出一条长长的边线，然后再从另一边绕过去，就能很快地在后面合适的时机圈起一大片地，而且这样避免了让大量纸带暴露在领地之外的危险。然后在算法实现上其实还可以引入更先进的数据结构，或者探索一些机器学习的方法，能够进一步提高运行效率和获胜可能。最后在小组分工方面也还可以做得更好，可能大家还不是很熟悉这种合作开发的方式，导致不能最大限度地发挥每一个人的能力。我们的同学提出了一些很好的想法，但由于时间原因没能最终整合到代码中，有一点遗憾。以后的小组作业中应当更加积极地沟通，也希望我们小组成员的友谊能够一直保持下去。

4.3 建议与设想

首先建议在大作业发布后尽快完成比赛环境以及重要参数的设定，反复修改会增加很多不必要的工作，这样可以帮助大家专心地开发自己的代码部分，当然技术组的同学已经非常优秀，在有限的时间内完成平台的搭建也很不容易。另外也希望老师能够在更早的时间发布大作业，因为多数同学对于这样的开发还比较陌生，如果有更多的时间相信能产生更出色的成果。热身赛是一个帮助大家完善策略的很好的平台，但可能有的小组在举办热身赛的时候还没有完成完整代码的撰写，如果可以开放一下热身赛复盘数据可能会对没能参赛的小组有很大的帮助。

有关本次实习作业的后续工作，我们可以继续完善一下危险判定的部分，以及尝试一些效率更高的算法和数据结构，另外还可以探索如何可以将没有利用起来的代码整合到一起，使我们的算法更加强大。

对学弟学妹当然是推荐选陈斌老师的数算啦，好好学 python 会在很多方面都有帮助。希望他们可以勇敢地去尝试，写代码这件事情其实也没有那么可怕。然后珍惜这次有趣的实习作业，相信自己和自己的队友，收获知识和快乐。

5 致谢

感谢陈斌老师的耐心指导与辛勤付出，感谢助教组和技术组的同学提供平台以及帮我们答疑解惑，感谢小组每一个成员的努力，感谢农园甜甜的桂圆红枣茶，以及感谢这一段美好的经历。

6 参考文献

1、数据结构与算法课程网站：http://gis4g.pku.edu.cn/course/pythonds/

2、github 网站：https://gitee.com/chbpku/paper.io.sessdsa

第二章　N17_Bravo 报告

张瀚垚　南天龙　符翊君　于明鑫　陈烨 [1]

摘要: 我们在数据结构与算法课所学知识的基础上，结合小组成员的游戏经验，实现了 Paper.io 纸带圈地游戏的 AI。我们采取前期尽量多圈地，后期在边缘立足防守，力图守住优势而获得胜利的游戏战略。其中采用隔数步重新根据我方和对手位置优化的圈地策略，配合使用队列数据结构储存未来操作进行圈地。而防守时首先移动靠近对方，然后与对方近距离绞杀，或者在边缘不断移动而阻止对方进入。同时配合以进攻函数，力图能够直接撞击对手取得胜利。我们通过将判定和实现分离，并辅以一个栈用于特殊情况下的优先级变换，来实现在不同情形下实现不同的策略函数。经过正式比赛的考验，我们虽然成功打入八强，但仍然发现我们的 AI 在防守状态判定上存在着严重的缺陷。

关键词：纸带圈地、数据结构、算法

1 算法思想

1.1 游戏思路战术设计——前期圈地，后期猥琐流

本组和许多其他小组相同，采取前期尽量大范围圈地、后期在边缘防守保证胜果的比赛思路。但是与大多数小组不同的是，我们的圈地方式依托于一条沿 y 方向延伸的边界，从这条边界上选取起始点和结束点，根据二者的坐标以及对手的位置，对最终一步圈地的“所得”做最优化。用来衡量“所得”的函数以圈地面积作为基础，同时加入对一些情况的奖励和惩罚。为了能够依托于沿 y 方向延伸的边界圈地，最初 300 回合之内我方需要尽量在 y 方向使区域延伸，而 x 方向的延伸长度在前期不很重要。

[1]按照学号排序

我们的防守判断基于对方的纸卷位置、纸带位置和与我方区域的距离变化趋势。在判断进入防守状态以后，我们将向对方的纸卷和纸带移动，力图能够碰撞击杀对方、或是减少对方在我方区域内能够停留的时间。如果成功把对方逼退，同时我方面积占优，则我方希望能够在边界与对方僵持从而耗尽回合数取得胜利。如果对方不断后退想要摆脱在边界上的纠缠，则我们希望可以通过蚕食对方领地让对方得不偿失。

我们的进攻策略是以进攻纸带为主，进攻纸卷为辅。如果确认对手已经无法回到自己区域，而我方与对方的纸卷距离比较近，为防止对方纸卷拼死一搏，我们会在本方区域内停留，直到对方纸卷走远。与进攻相伴的，还有回到本方区域的保命策略。如果对方与我方纸带的距离与我方的最小回复距离差距较小，那么我方将会采用最快回复策略回到本方区域，再做下一步打算。

总的来说，我方游戏的关键思路在于两点：1. 在前期纸卷距离较远时尽可能增大本方圈地面积；2. 在后期尽可能在边缘做好防守，保持住面积的优势。这一切需要在保证不被对方撞击纸带或侧碰纸卷的前提下进行，同时一旦有机会撞击对方纸带或侧碰对方纸卷，我方也会在保证安全的前提下去提前结束比赛。从我方对比赛的思路出发，可以将整个游戏分解成圈地、防守、进攻等环节，因此我们最终的团队分工也按照此原则来进行。

1.2 代码整体架构——根据场上局势选择行为模式

代码的总体架构需要实现根据场上不同情况，采取不同的模式、调用不同的函数来进行操作。同时，为了防止出现一些不合理的操作，需要在以上判断结束以后加入一个过滤器来“过滤”不合理的操作（Protect 函数）。我们采用的判断优先级如图 1 所示。其实现思路分为两部分：1. 通过将判断函数和执行函数分开，采用标识符的方法，判断函数决定出标识符后，传入 action 函数调用相应的执行函数，再将 action 函数的返回值传入 protect 函数中进行最后的检验（是否会撞墙、撞纸带等）; 2. 通过使用栈数据结构，保存上一步操作，来实现特殊情况下判断函数优先级的跨越。

其物理层次实现通过以下代码片段：

```
if storage['orderbuffer'].peek() == 'round':
                # 判断圈地标识符来确定对应的全局标识符状态
                if storage['state'] == 'Init':
                                storage = storage['InitStepJudge'](stat, storage)
                elif 'FirstStep' in storage['state']:
                                storage = storage['FirstStepJudge'](stat, storage)
```

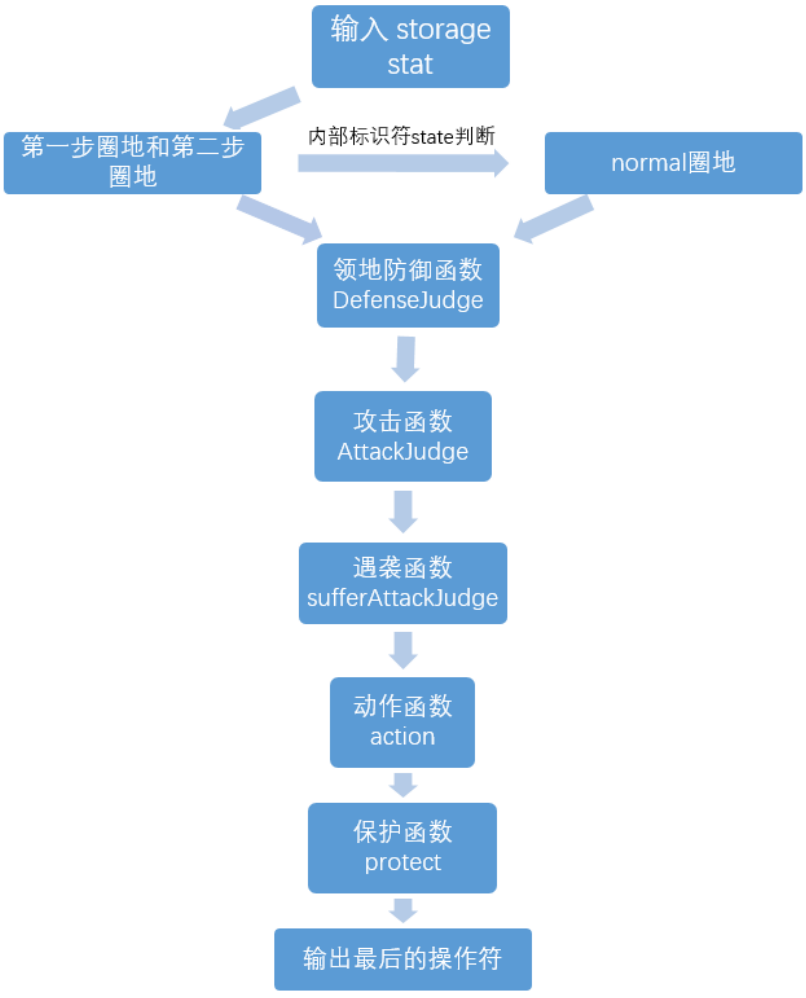
```
                elif 'Strip' in storage['state']:
                        storage = storage['MakeAStripJudge'](stat, storage)
                elif 'ThirdStep' in storage['state']:
                        storage['outstate'] = 'normal'
                # 纸卷和纸带遭受攻击函数的优先级最高
                storage = storage['JudgeDefense'](stat, storage)
                storage = storage['AttackJudge'](stat, storage, rmn, rmfm, rnbm, rnfn,
                                    rmbn, me, enemy)
                storage = storage['sufferAttackJudge'](stat, storage, rmn, rmfm, rnbm,
                                    rnfn, rmbn, me, enemy)

    else:
                storage['outstate'] = 'normal'
                storage = storage['JudgeDefense'](stat, storage)
                storage = storage['AttackJudge'](stat, storage, rmn, rmfm, rnbm, rnfn,
                                    rmbn, me, enemy)
                storage = storage['sufferAttackJudge'](stat, storage, rmn, rmfm, rnbm,
                                    rnfn, rmbn, me, enemy)

                # 圈地被打断后强行进入快速返回函数判断条件为上一步操作为攻击或逃跑
                if (storage['orderbuffer'].peek() == 'Attack' or storage['orderbuffer'].
                                    peek() == 'escape') \
                and storage['outstate'] == 'normal':
                        storage['outstate'] = 'FastReturn'
                # 快速返回执行完前不被圈地函数打断可以被攻击等函数打断
                else:
                        if  storage['orderbuffer'].peek() == 'FastReturn' and
                                    storage['outstate'] == 'normal'\
                                                and stat['now']['fields'][
                                    me['x']][me['y']] != me['id']:
                                        storage['outstate'] = 'FastReturn'
    return storage['Protect'](action(stat, storage), stat)
```

第一部分为常规流程。各判断函数之间的优先级为：纸卷纸带遭受攻击函数（sufferAttackJudge）> 攻击函数（AttackJudge）> 领地防御函数（DefenseJudge）> 普通圈地函数。各判断函数接收 storage 参数，返回值亦为 storage，在函数内部或对 storage 中 outstate 进行更改。通过按照优先级从低到高的次序依次调用各判断函数，来实现对 storage 中 outstate 的更改和覆盖，最终返回给 action 函数进行调用。protect 函数接收 action 函数的返回值并进行是否撞墙或撞纸带的判断，如果会撞墙或撞纸带则直接更改最终的走位，优先级最高。

第二部分为特殊情况下优先级的跨越。如：圈地函数内部标识符的连接和传递需保证不受外部的干扰、一旦第三步圈地过程被打断需立即回到领地且回到领地前不再调用圈地函数（前两部圈地过程无此要求）。故采用栈数据结构来保存上一步操作，通过上一步操作与第一部分结束后 outstate 标识符来共同决定是否对 outstate 标识符进行更改。



每走一步操作中调用函数的流程图

1.3 采用的数据结构和算法

在本次大作业中，我们采用了栈和队列两种数据结构，分别用于特殊情况下优先级的跨越和储存转向规划。这两种抽象数据结构帮助我们解决了函数优先级的强制转换，以及圈地函数中将目前回合的规划转化为未来的实际行动行动的问题。

我们采用了广度优先搜索 (BFS) 的算法来得到与某个区域和某个点的距离。我们采取的 BFS 和常规的写法有所区别。常规的 BFS 将待搜点放入一个队列中，再不断地进行搜索，但是考虑到游戏程序返回的地图并不是一个良好定义的图，直接对这个二维列表采取 BFS 可能会出现死循环等各种问题，最小距离点的递推式也较难求得。因此我们先将 x 和 y 的所有可能值按照与该点相应坐标之差的绝对值求出，然后再用一个嵌套循环，就实现了广度优先搜索——按照与起始点的距离来搜索的算法。一旦发现有一个点属于被搜寻区域，则中断循环而节省时间。

1.4 算法运行时间复杂度分析

策略中大多数判断和条件都是 $O(1)$ 复杂度，只有 ThirdStep 和 MinDistance 是 $O(n^2)$。运行一次 play 函数的时间复杂度是 $O(n^2)$。其中 n 是场地的宽度或长度。

2 程序代码说明

2.1 数据结构说明

使用栈实现特殊情况下优先级的跨越

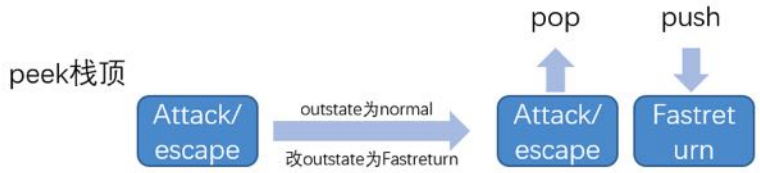
我们采用栈数据结构来保存上一步操作，通过上一步操作与第一部分结束后 outstate 标识符来共同决定是否对 outstate 标识符进行更改。

1. 圈地函数内部标识符的连接和传递需保证不受外部的干扰。当栈内 order 为'round' 时，表示正在进行前两步圈地，具体执行什么操作由内部标识符 state 决定。前两步圈地整体过程中可以被攻击和防守等行为打断，但危机解除后可以凭借内部标识符 state 继续原来的圈地策略。故栈内为'round' 时一直保留，直到前两步圈地完成后弹出，并压入'normal'。

2. 一旦 normal 圈地过程被打断需立即回到领地。如图 2，normal 圈地过程由于自身的特

殊性，每次圈地过程需要从领地内出发。故当栈内为'Attack' 或'escape' 函数且 outstate 标识符判断为圈地时，则将栈中 order 弹出并立即执行 FastReturn 操作。

3. 回到领地前不再调用 normal 圈地函数。原理同上，如图 3，栈内为'Fastreturn' 时保留操作继续执行，直到回到领地。



圈地函数被打断后返回本方领地的栈调用过程

实现回到领地前不再调用圈地函数的栈调用过程

使用队列保存已规划好的行动策略

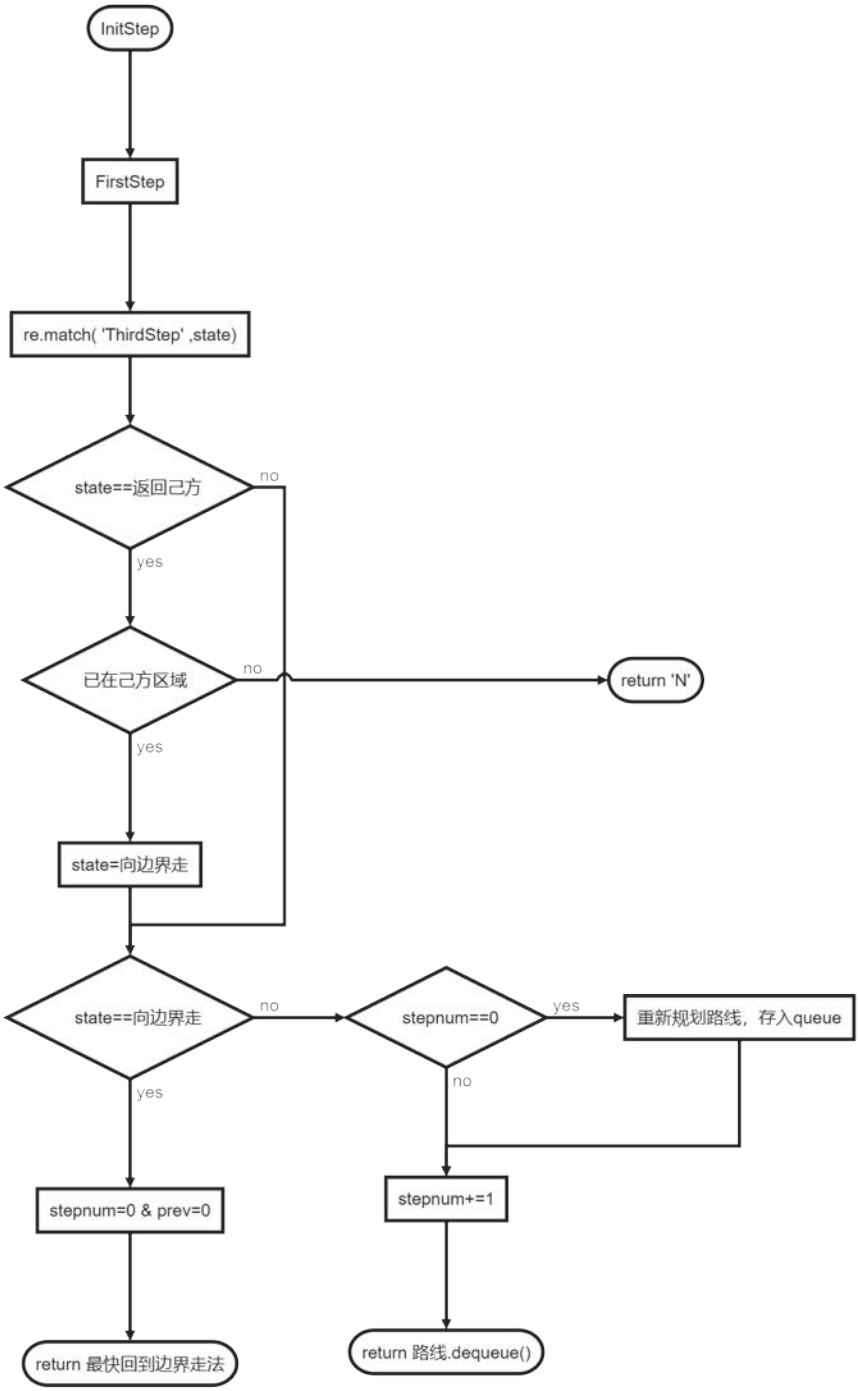
主圈地函数对未来的纸带转向做了规划，为此它返回一个队列于 storage 中，队列中存有之后的转向顺序，直到下一次更新圈地规划。随着圈地规划更新，storage 中的队列也更新，而每一步的操作都从队列中 dequeue 取得。这样就实现了每隔一定步数更新最优化圈地的行动策略。

2.2 圈地函数说明

圈地综述

圈地共分为两段，第一段是 FirstStep，仅在对战刚开始的阶段进行，主要目的是在 y 方向上占据较大的长度，以便进行第二段的圈地，同时值得注意的是，在此阶段，x 方向上的延伸长度并不十分重要。而第二段是在完成了第一步的圈地（FirstStep）后，在较长

时间内采用的策略。依据当前双方的状态，在当前己方的 y 方向的边界上选取圈地的开始和结束点，尽可能的使评估函数（Evaluate）取到最大值。



为了实际操作的简单，我们将第二段的圈地的操作分成了三个部分：ThirdStep1（在 x 方向上向外延伸）、ThirdStep2（在 y 方向上向下或者向下进行延伸）、ThirdStep3（结束圈地，向己方区域返回，并准备下一轮圈地）。同时由于圈地是一个续延时间较长的状态，并且为了节约时间，我们采用了每隔若干（实际设为 3）步进行一次规划，并将规划的结果以队列 queue 的形式储存在 storage['ThirdStepRoute'] 中，在接下来无需规划的步骤中，仅从队列中依次 dequeue 即可。

时间复杂度的分析：在第二阶段的圈地过程中，重新规划路线时：调用 ThirdStep，时间复杂度为 $O(n^2)$；无需规划时，仅仅 dequeue 一次，时间复杂度为 $O(1)$。

调整初始方向：InitStep

本策略用于游戏最开始，仅进入一次。我们的战略要求在最开始 200-300 回合之内圈出一条沿 y 方向延伸的长条。考虑到我们的圈地函数仅考虑到了 x 方向往对方优势的区域圈地，因此这个长条越长，随后我们的圈地函数就能运行得越顺畅。而 InitStep 的目的是将初始的随机给定方向调整至南/北（即 y 方向），从而作为圈地的开始。它的具体实现非常简单，如果本方先手，则将方向调至 N，后手则调至 S，这样的调整可以使得 FirstStep 中所有的转向都是左转，为策略编写提供便利。

圈出长条形区域：FirstStep

本策略用于 InitStep 结束以后，同样仅进入一次。在 FirstStep 中，我们根据对手与我方纸带的距离以及我方和我方区域的距离，规划出兼顾 y 方向延伸长度和圈地大小的长条。二者之间的权衡取舍通过 STRETCHAIM 参数来确定，即纸卷在南北方向移动到某一个特定的 y 坐标，或是对方到我方纸带的距离小于我方回到本方区域的距离加上一个安全阈值，就会转向从而在东西方向圈地。东西方向的扩展长度由距离所决定，与南北方向相同，如果对方纸卷靠近我方纸带，我方纸卷就会再次转为南北走向从而往我方区域回复。最终圈出一条 y 方向延伸的长条，其长度最大为 90（需要根据对方的位置来寻找安全长度），为我们之后采取的最优化圈地面积策略打下基础。

具体实施的伪代码和流程图如下：

```
def ThirdStep(storage, stats, stepnum):
    if simstate == 'ThirdStep1':
        #转向到向外的方向
```

```
    #未超出规划的步数时
        if simstate == 'ThirdStep1':
            # 未到达 farthest 时，直行
            # state 更改，进入 ThirdStep2 ，转向
    elif simstate in ['ThirdStep2u', 'ThirdStep2d']:
        if 入点被重新规划:
转向并回到              ThirdStep1
        else:
            #在状态中，直行ThirdStep2
            # 更改，进入，转向stateThirdStep3
    else:
        break
    return 规划的queue
```

主圈地函数：ThirdStep

本函数的目的是生成第三步圈地途中的前进路线，以队列形式返回，以便在后续取用，即每次仅有在 ThirdStep 被调用的过程中，Inpoint（入点）和路线才会被重新规划。ThirdStep 函数在重新规划的过程中，以 storage['state'] 的状态符记录当前圈地所处于的状态。

调用 Inpoint 函数进行规划，时间复杂度为 $O(n^2)$；第一步的转向事件复杂度为 O(1)；后面的循环由 num 进行次数的控制，在每次的循环过程中，仅有 if 的判断语句，和 queue 的 enqueue 操作，时间复杂度亦为 $O(1)$。所以 ThirdStep 总的时间复杂度为 $O(n^2)$。

选择“最优”的圈地结束点函数：InPoint

本函数是针对给定的评估函数（Evaluate）和己方边界，找到最佳的回到己方区域的入点坐标 InPoint。选择的基本原则是：再一次圈地中仅朝某一特定 y 方向延伸，以减小纸带的暴露长度和增大圈地面积。

分析它的复杂度，Inpoint 函数通过遍历，每次调用 $O(n)$ 的函数 Evaluate，所以时间复杂度为 $O(n^2)$。

```
def InPoint(storage, stats, simstate, prev):
    value, currentval, flag = -1000, 0, 'survive'
    inpoint = {'x': copy.deepcopy(storage['border'][stats['me']['y']]), 'y': stats['me']['
                                            y']}
```

```
    if simstate == 'ThirdStep1':
        if 在本轮圈地中，之前曾向上延伸:
            # 只在当前位置上方寻找入点，即:
            for i in range(0, stats['me']['y'] + 1):
                if 待评价行有己方区域:
进行规划
                if 规划值 > value:
                    value = currentval
                    inpoint =当前规划行边界坐标
        elif 在本轮圈地中，之前曾向下延伸:
只在当前位置下方寻找入点
        else:
在整个              y 的范围内寻找入点
    elif simstate == 'ThirdStep2d':
仅对当前位置下方进行入点搜索
    elif simstate == 'ThirdStep2u':
仅对当前位置上方进行入点搜索
    return inpoint
```

圈地效果评估函数：Evaluate

本函数针对给定的 StartPos（当前所在位置 $[x_1, y_1]$），StopPos（给定的返回位置 $[x_2, y_2]$），评估当前所给定的 StopPos 的优劣。当前所选择的评估标准：本次所能圈成的总面积减去所延伸距离的某个倍数（设置为 3），在反复的实验中我们发现，它可以较好的平衡圈地的面积与所需的步数。分析它的时间复杂度，通过遍历，每次调用 $O(1)$ 的函数 AddSquare，所以复杂度为 $O(n)$。

```
def Evaluate(stats, StopPos):  #圈地效率的评估函数
    Square = 0
    y1 = stats['me']['y']  #当前所在位置
    y2 = StopPos['y']   #待评估的纵坐标
    if 不合理的圈地情况（纸带会被对方撞到）
        return 0 （最小，必被排除）
    else:
逐个纵坐标遍历         :
            Square += AddSquare
            Square -=最远延伸距离  *3   #对过远的延伸进行惩罚，在 Square 上减去延伸距离的倍数（为可调
                                        参数）3
    return Square
```

圈地过程中最大延伸长度：extend

Extend 函数用于计算出对于指定的出发点（判断时自己所处的坐标），返回点以及敌方位置，自己最多可以在横向延伸的距离。

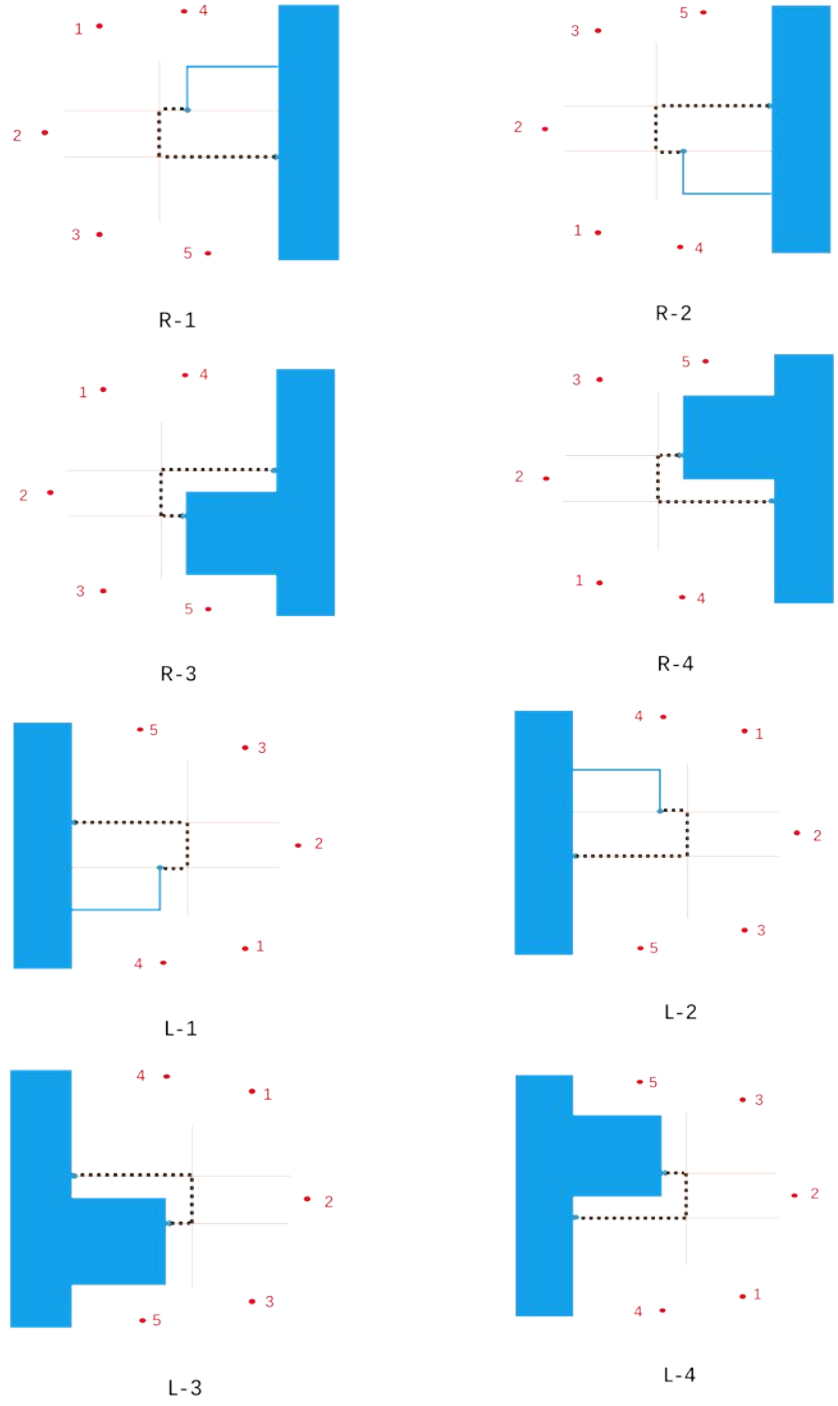
具体而言，根据自己和返回点、以及敌方的相对位置对基本情况进行分类：根据己方和返回点位置可以大致分为 8 类，如图 5 所示，每一类对应着 5 种敌方所处位置的分类。

对于每一种分类，我们可以大致估算出最大的延伸距离。因为所用的计算公式均具有很好的对称性，因此我们将这些公式先存储在一个字典中，并根据索引反复调用公式，而所用的语句格式更多是：

```
if 某情况:
    d =某公式
else 某情况:
    d =某公式
```

另外，在计算时很容易遇到在一中条件下一定无法返回指定点的情况，对于该情况函数中设定了“生存标识”flag，用于记录所有一定死亡的情况，对于这些情况我们指定评估时是最差的，从而淘汰掉这些目标点。函数中还设定了”转向标识”turn，用于记录目前状态是否需要转向以及需要如何转向。

可以认为在这个函数中，对公式的输入和变量的定义为 O(1)，大量的 if 语句同样是 O(1)，因此这样的估算函数没有使用任何循环迭代逻辑，时间复杂度为 O(1)，使用过程中仅占用部分存储空间。



extend 函数中考虑的 8 种情况示意图

计算给定圈地起始点、结束点和伸展距离的圈地面积：AddSquare

针对给定的 bound（所在行的边界 x 坐标），xmax（所能扩展到达的最大 x）；计算本行所能被圈入的面积。如果在当前行，xmax 在边界内部，则面积为 0；如若不然，则增加这个坐标的差值。显然，本函数时间复杂度为 $O(1)$。

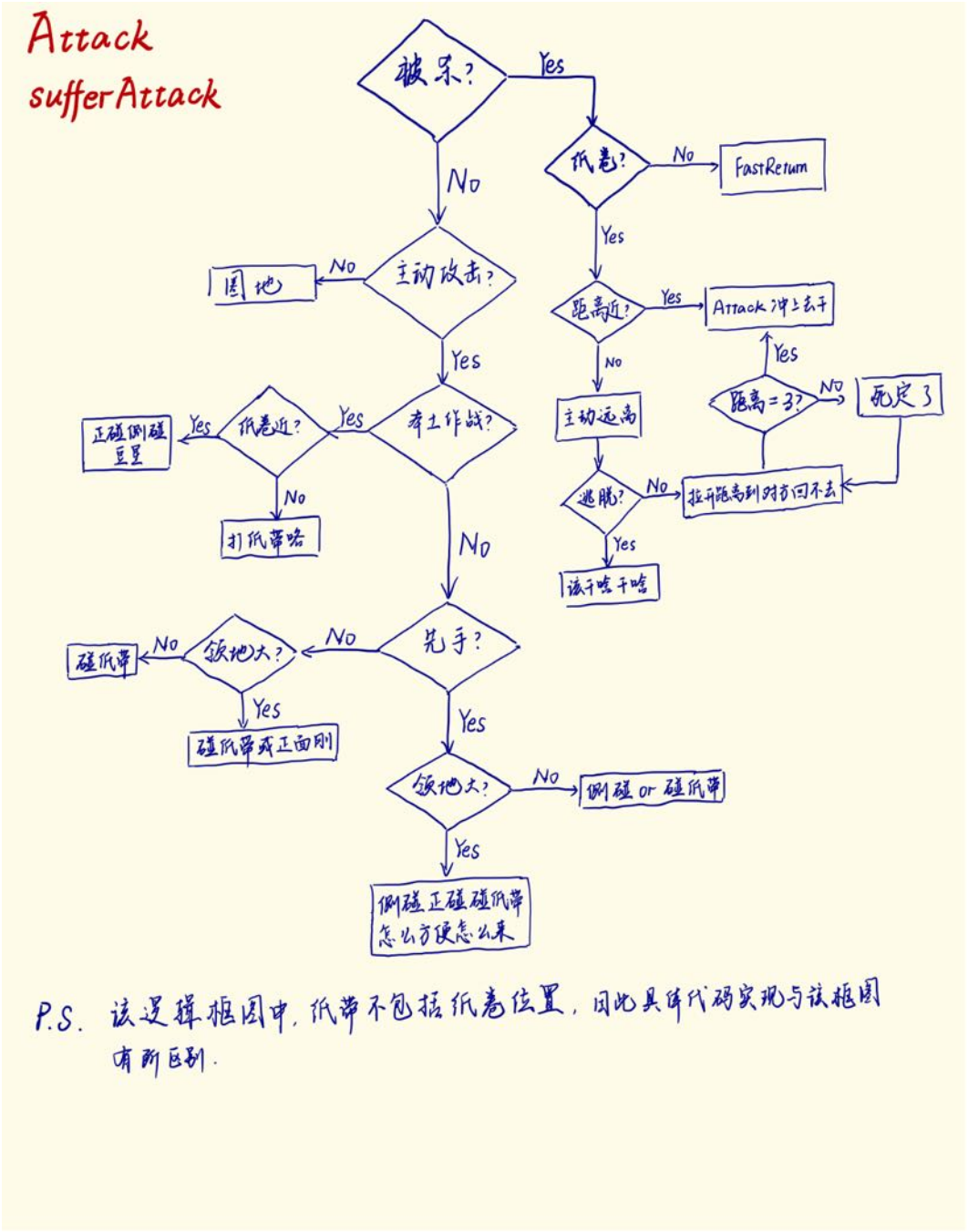
两次圈地之间的衔接：Return2Border

在一次圈地完成以后，由于我们的圈地策略限制，需要回到本方边界并调整方向，这是两次圈地中的“过渡期”。此处“边界”的定义仅仅是垂直于 x 轴的边界点集，而非四个方向的所有边界点集。因此，我们首先利用 BorderUpdateProcess 函数获得用列表表示的边界点集，随后采用 FastReturn 移动纸卷位置到这些点中最近的点处，最后再根据边界点集的延伸方向和纸卷的朝向来调整方向。最后实现回到边界的同时，将方向调整为对准对方初始位置（如果本方先手则为东，本方后手则为西）。

2.3 攻击函数说明

攻击判断函数：AttackJudge

AttackJudge 函数为攻击判断函数，接收 stat，storage 和敌我纸卷纸带相关位置参数。用 if 语句判断攻击条件（敌方回不去且己方距离敌方纸卷或纸带位置小于某一特定值）是否成立，如果成立，则将传入的 storage 中的 outstate 标识符更改为'Attack' 并返回，否则返回 storage 不做更改。攻击部分整体流程如图 6 所示。


Attack
sufferAttack
被杀?
Yes
No
纸卷?
No
FastReturn
Yes
主动攻击?
No
围地
Yes
距离近?
Yes
Attack 冲上去干
No
主动远离
Yes
距离=3?
No
死定了
逃脱?
No
拉开距离到对方回不去
Yes
该干啥干啥
本土作战?
Yes
纸卷近?
Yes
正碰侧碰豆豆
No
打纸带哈
No
先手?
No
领地大?
No
碰纸带
Yes
碰纸带或正面刚
Yes
领地大?
No
侧碰 or 碰纸带
Yes
侧碰正碰碰纸带怎么方便怎么来
P.S. 该逻辑框图中，纸带不包括纸卷位置，因此具体代码实现与该框图有所区别.

攻击模式流程图

攻击策略函数：Attack

Attack 函数为攻击函数，即满足攻击条件时具体采取哪种攻击策略。该函数不仅可以被 AttackJudge 函数调用，同时也可以被 sufferAttackJudge 函数调用，源代码如图。传入敌我位置距离等相关参数，通过调用 follow 函数和 wrestle 函数返回具体走位（'L'、'R'或'N'）。

```
def Attack(me, enemy, rmbn, rmn, rmbnpoint):
    # 本土作战
    if stat['now']['fields'][me['x']][me['y']] == me['id'] \
    or stat['now']['fields'][enemy['x']][enemy['y']] == me['id']:
    # 比较到敌方纸卷和纸带的距离
        if rmbn <= rmn:
            return follow(rmbnpoint, me)
        else:
            return follow(enemy, me)
    # 非本土作战
    else:
        if me['id'] == 1:
            s = AreaRecorder(stat)
            if s[0] > s[1]:
                if rmbn <= rmn:
                    return follow(rmbnpoint, me)
            else:
                return follow(enemy, me)
        else:
            if rmbn <= rmn:
                return follow(rmbnpoint, me)
        # 侧碰
            elif rmn == 3:
                return wrestle(me, enemy)
            else:
                return follow(enemy, me)
        else:
            return follow(rmbnpoint, me)
```

攻防基础走位函数：escape, follow, wrestle

escape、follow、wrestle 为进攻和防御过程中的基础走位函数。escape 函数的基本原则为：朝着敌人的反方向走，优先拉开对角线距离。因此需要传入敌我相关的位置和方向

信息，并返回具体走位（'L'、'R' 或'N'）。follow 函数的基本原则为：向敌人纸卷或纸带所在位置前进，优先缩小左右距离，再缩小上下距离。因此需要传入敌人纸卷和纸带的位置参数，并返回具体走位（'L'、'R' 或'N'）。wrestle 函数的基本原则为：将敌己距离缩短到 2*2 的方块内。由于己方轮回时，只有双方距离为奇数时才有侧碰攻击的机会，故 wrestle 函数只在双方距离为奇数时才能够调用并发挥作用。返回具体走位（'L'、'R' 或'N'）。

2.4 防守函数说明

防守判断函数：DefenseJudge

传入 stat、storage 参数。目的是判断是否进入防守状态或者进入快速回到领地的状态。在函数中设置了一个距离距离阈值 r（r 置为 25 单位），调用了 AreaRecorder 函数以及 DistanceRecorder 函数。在 storage 中加入了'warning' 标识，用于判断对手是否有纸带在我方领地中或者对手是否在我方领地中，其实现方法是：当'warning' 不在 storage 中时则将 storage['warning'] 置为 0，当对手进入我方领地后将其置为 1，当对手回到对方领地后将其置为 0。进入防守状态的条件为：我方领地面积大于对方是先决条件，对手的纸带或者纸卷在我方领地中或者对手纸卷和纸带都不在我方领地中且有靠近我方的趋势（通过比较上一步和这一步之后对手离我方边界最短的距离的大小关系来判断是否有靠近趋势，若减少则判定为有趋势，反正判定为没有），当自己不在己方领地内则进入 FastReturn 的状态；当自己在己方领地时则进入 Defense 状态。状态的改变通过改变 storage['outstate'] 的值来实现。若进入 FastReturn 的状态，则将 storage['outstate'] 置为'FastReturn'；若进入 Defense 状态，则将 storage['outstate'] 置为'Defense'。

防守策略函数：DefenseStrategy

传入 stat 参数。目的是采取防守措施，即靠近对手或者对手的纸带。其实现方法为分对手与自己的相对位置共计 8 种情况来分析，其中细分自己与对手的朝向来讨论，共计 128 种情况，这里对其中一种相对位置作简要分析：（对手在自己的正南方）

1、如果对手朝东方：如果自己朝东方，则返回'N'；如果自己朝南方或西方，则返回'L'；如果自己超北方，则返回'R'。

2、如果对手朝南方或者朝北方：如果自己东方，则返回'R'；如果自己朝南方，则返回'N'；如果自己朝西方，则返回'L'；如果自己朝北方，则随机返回'R' 和'L'。

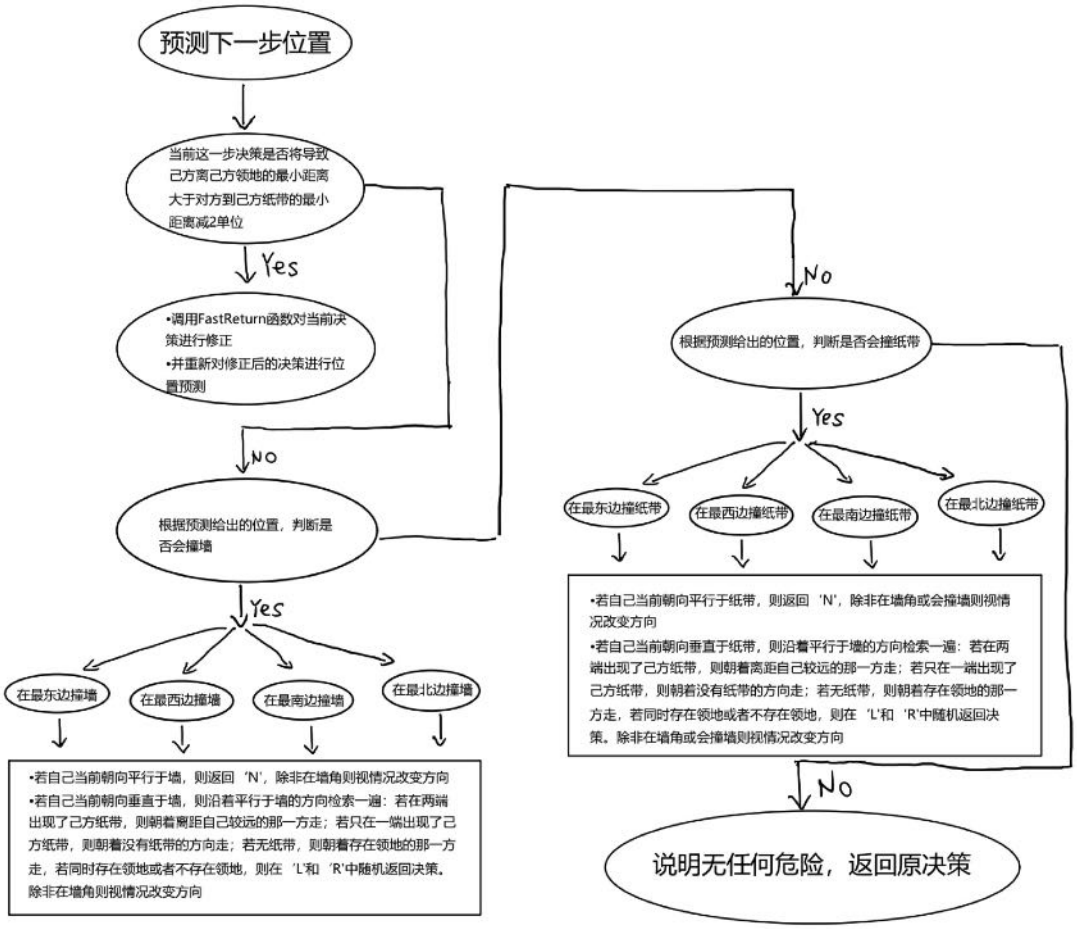
3、如果对手朝西方：如果自己朝北方，则返回’L’；如果自己朝南方或朝东方，则返回’R’；如果自己朝西方，则返回’N’。

防止被杀或撞墙：Protect

如图 6，传入 action、stat 参数，其中 action 为一个字符串，表征执行函数所返回的决策，可取值为’R’、’L’、’N’。目的是防止当前决策将导致自己有被敌方杀掉的、撞墙或者撞纸带。

首先结合自己当前的朝向以及传入的 action 给出的决策对下一步位置进行预测。然后调用 MinDistance 函数来判断自己到己方领地的最小距离与对手到我方纸带的最小距离减 2 的大小关系，若前者小于后者则表明安全，否则表明需要对这一步决策进行修正。修正方法是直接调用 FastReturn 函数来给出新的决策，并且将重新做出下一步位置的预测，在经过防撞墙和防撞纸带模块的过滤给出最终修正的结果。

防撞墙和防撞纸带的机制类似，触发条件为经过预测后的位点超出地图范围或者刚好在己方纸带上。然后对自己的朝向进行分类讨论：若为侧碰（即本身是沿着墙缘或者纸带边缘走位然后执行函数给出了’R’ 或’L’ 的指令导致下一步会撞墙或撞纸带），则修正为沿着墙缘或者纸带边缘行走；若为正碰（即本身的朝向垂直于墙或者纸带且执行函数给出了’N’ 的指令），则沿墙或者纸带方向遍历当前这一步所处的条带，若存在纸带，则向着没有纸带或者存在纸带且离自己距离较远的那一方进行修正，若不存在纸带且在一方存在己方领地，则向着存在领地的那一方进行修正，若不存在纸带且两边都有领地或者都没有两地，则随机修正。最后将修正完的 action 返回。



Protect 函数实现防止撞墙、撞纸带的判断流程

被攻击判断函数：sufferAttackJudge

sufferAttackJudge 函数为被攻击判断函数，其与其他模式之间的逻辑关系示意于图 2。它接收 stat，storage 和敌我纸卷纸带相关位置参数用 if 语句判断是否正在遭受攻击，如果遭受攻击则将传入的 storage 中的标识符更改为对应的策略，源代码如图。Fastreturn 为找最近的路径快速返回领地的函数，escape 为尽量远离敌人纸卷的函数，Attack 为攻击函数。

```
def sufferAttackJudge(stat, storage, rmn, rmfm, rnbm, rnfn, rmbn, me, enemy):
    # 如果敌方在敌方领地己方在己方领地且敌我距离小于等于 6 则调用领地边缘走位函数
    if stat['now']['fields'][enemy['x']][enemy['y']] == enemy['id'] \
    and stat['now']['fields'][me['x']][me['y']] == me['id'] \
```

```
    and rmn <= 6:
        storage['outstate'] = 'Walkfrontier'
    # 敌方到己方纸卷或纸带距离小于等于15 判定敌方有攻击意图
    # 己方到自己领地距离+4 大于敌方到己方纸卷或纸带距离之中较小的值则判定回不去
    elif rmfm + 4 > min(rmn, rnbm) and (rmn <= 15 or rnbm <= 15) \
            and stat['now']['fields'][me['x']][me['y']] != me['id'] :
        # 如果敌人在敌方领地内则快速回去
        if stat['now']['fields'][enemy['x']][enemy['y']] == enemy['id']:
            storage['outstate'] = 'FastReturn'
        # 如果敌人意图攻击己方纸带则快速回去
        elif rmn >= rnbm:
            storage['outstate'] = 'FastReturn'
        # 敌方也回不去且敌我距离为3 则发起反击
        elif rmn == 3 and rmn < rnfn and rmfm <= min(rmn, rnbm):
            storage['outstate'] = 'Attack'
        # 其他情况调用逃跑函数尽量远离敌方纸卷
        else:
            storage['outstate'] = 'escape'
    # 敌方有攻击纸带意图且己能回去由于纸卷所在位置也会判定为纸带( 故需排除不在己方领地内的情况)
    elif rnbm <= 6 and rmfm <= rnbm + 1 and rmbn > rnbm \
            and stat['now']['fields'][me['x']][me['y']] != me['id']:
        storage['outstate'] = 'FastReturn'
    # 敌方有攻击纸卷意图且己能回去，排除在自己领地内的情况
    elif rmn <= 6 and rmfm <= rmn + 1 \
            and stat['now']['fields'][me['x']][me['y']] != me['id'] :
        s = AreaRecorder(stat)
        # 领地大且距离为奇数就上吧
        if s[0] > s[1] and rmn \% 2 == 1 \
        and stat['now']['fields'][enemy['x']][enemy['y']] != enemy['id']:
            # 具体为正碰或侧碰交由攻击函数内部处理
            storage['outstate'] = 'Attack'
        else:
            storage['outstate'] = 'FastReturn'
    return storage
```

本方边界游走：WalkFrontier

该策略的目的是在本方边界上游走，用在与对方在边界僵持，双方纸卷距离非常靠近的情形下，防止对方进入我方领地而圈走我方地盘，同时防止我方由于离开本方区域而被撞死。首先通过 getFrontierDirections 得出目前位置往哪一个（绝对）方向走仍然是边界，

随后在这些方向里选取能够尽量靠近对手的方向。最后根据自己目前的方向，使用 turn 函数进行适当的转向。选择方向的优先级是：方向可以保证停留在区域内 > 可以在 y 方向靠近对手 > 可以在 x 方向靠近对手。有一些情况下，可能无论如何转向都无法保持在本方区域内，此时选择左转，是因为直接往前走大概率需要至少 3 步才能回到本方区域，而左转往往只需最多两步，这一步的差距往往就能决定是否被杀。

2.5 基础设施函数说明

点到区域间的距离：MinDistance

使用广度优先搜索算法来获取点到区域间的最小距离，但是写法上和传统的 BFS 不同，并不显式使用图和队列的数据结构，而是通过对二维列表中的点排序依照与目标点距离而实现。首先对所有的 x, y 值按照与坐标的差值绝对值从小到大排序，得到两个列表。任意一对坐标 (x, y)，横纵坐标在两个列表中下标之和越小，说明与目标点之间的距离越小。所以，对二者的下标之和采用一个循环，一旦发现属于目标区域的点就结束循环，这样可以较快的找到最小距离和对应的区域内点。这种写法比一般的 BFS 应用到这个问题上速度更快，平均来说，在距离较近时，这种改进版 BFS 比爆搜快 5-10 倍不等，与一般用队列实现的 BFS 时间速度相近。当距离较远时（临界值约为 140），本 BFS 可能会慢于爆搜，但是远快于用队列实现的正常 BFS。考虑到比赛中实际情况，在这一个问题中我们选择使用改进版 BFS。改进广度优先搜索代码如下：

```
def MinDistance(region, point, owner):
    mindis = MAXHEIGHT + MAXWIDTH - 2
    minpoint = {}
    posx = point['x']
    posy = point['y']
    # 对x, y 按照与目标点的距离排序，从而隐式实现广度优先搜索中的节点搜寻顺序
    xfilter = sorted(list(range(MAXWIDTH)), key=lambda x: abs(posx - x))
    yfilter = sorted(list(range(MAXHEIGHT)), key=lambda y: abs(posy - y))
    found = False
    #
    for i in range(MAXHEIGHT + MAXWIDTH - 1):
        if found:
            break
        for j in range(max(0, i - MAXWIDTH + 1), min(MAXHEIGHT, i + 1)):
            if region[xfilter[i - j]][yfilter[j]] == owner:
                if mindis >= abs(point['x'] - xfilter[i - j]) + abs(point['y'] - yfilter[j
```

```
                                    ]):
                    mindis = abs(point['x'] - xfilter[i - j]) + abs(point['y'] - yfilter[j
                                    ])
                    minpoint = {'x': xfilter[i - j], 'y': yfilter[j]}
                    found = True
    return mindis, minpoint
```

该算法由两次排序和一次二重循环所组成，Python 自带排序的时间复杂度为 $O(nlogn)$，而二重循环的最差时间复杂度为 $O(n^2)$，因此最差情况下这个算法的时间复杂度为 $O(n^2)$。但如果点和区域的（平均）距离较近，那么复杂度将会降低，因为不需要搜索到 n^2 数量级的点就已经达到了目标区域，从而通过 break 语句跳出了循环。对于它的实战应用，点和区域的距离往往比较近，此时实际时间复杂度低于 $O(n^2)$。

将绝对方向转化为相对方向：turn

绝对方向是指地图上的“东西南北”，相对方向是相对纸卷目前朝向的“前后左右”，在实际策略中，很多时候需要在二者之间做转换。因为策略考虑的是要往哪一个绝对方向移动，而实际输出的是纸卷要往哪一个方向转向。因此本函数输入 mydirection 和 targetdirection 两个参数，将目前的朝向根据目标方向转化为转向输出。

记录双方领地面积：AreaRecorder

传入 stat 参数，目的是实时记录双方的领地面积。为此采用遍历地图的算法，通过对每一个各点进行归属判断，同时分别用 sm，se 来记录双方领地面积。最后返回一个名为 a 的 list，其中第零个元素为 sm，即己方领地的面积；第一个元素为 se，即对方领地的面积。

记录对方纸卷和己方区域的面积：DistanceRecorder

传入 stat、storage 参数。目的是实时记录每一步对手离己方领地的最小距离。其做法为在 storage 中新建一个 key 为’memory’ 的字典，其中有一个 key 为’distance’ 的列表，用以记录每一步敌方离我方领地的最小距离。最后返回列表 storage[’memory’][’distance’]。

快速返回本方区域策略：FastReturn

传入 stat、region、point、owner、direction 参数。其中 region 为目标返回的区域，是一个二维列表；point 为自己的坐标字典；owner 为自己的 id；direction 为自己当前的朝向。其目的是为了直接向距离最近的点返回。

首先通过调用 MinDistance 函数来获得边界上最近点的坐标，然后依据最近点与自己的位置的相对关系分八种情况进行讨论：

1、若最近点在自己的正南方：若当前自己朝东，则返回'R'；若自己当前朝南，则返回'N'；若自己当前朝西，则返回'L'；若自己当前朝北，若自己不在最西北角且敌人在自己东边或者自己在最东北角，则返回'L'，反之返回'R'。

2、若最近点在自己的正北方：若当前自己朝东，则返回'L'；若自己当前朝西，则返回'R'；若自己当前朝北，则返回'N'；若自己当前朝南，若自己不在最西南角且敌人在自己东边或者自己在最东南角，则返回'R'，反之返回'L'。

3、若最近点在自己的正东方：若当前自己朝东，则返回'N'；若自己当前朝南，则返回'L'；若自己当前朝北，则返回'R'；若自己当前朝西，若自己不在最西北角且敌人在自己南边或者自己在最西南角，则返回'R'，反之返回'L'。

4、若最近点在自己的正西：若当前自己朝北，则返回'L'；若自己当前朝西，则返回'N'；若自己当前朝南，则返回'R'；若自己当前朝东，若自己不在最东南角且敌人在自己北边或者自己在最东北角，则返回'R'，反之返回'L'。

5、若最近点在自己的东南方位：不考虑其他因素，有两条回去的规则的折线路径。以自己与最近点连线为分界线，如果敌人在连线以上，则选择向南的路径；反之选择向东的路径；若在连线上则任意选择。根据每一步实时更新并选择路径。

6、若最近点在自己的东北方位：不考虑其他因素，有两条回去的规则的折线路径。以自己与最近点连线为分界线，如果敌人在连线以上，则选择向东的路径；反之选择向北的路径；若在连线上则任意选择。根据每一步实时更新并选择路径。

7、若最近点在自己的西南方位：不考虑其他因素，有两条回去的规则的折线路径。以自己与最近点连线为分界线，如果敌人在连线以上，则选择向南的路径；反之选择向西的路径；若在连线上则任意选择。根据每一步实时更新并选择路径。

8、若最近点在自己的西北方位：不考虑其他因素，有两条回去的规则的折线路径。以自己与最近点连线为分界线，如果敌人在连线以上，则选择向西的路径；反之选择向北的路径；若在连线上则任意选择。根据每一步实时更新并选择路径。

3 实验结果

3.1 测试数据

我们在 Intel Core i7-7700HQ CPU 的 Alienware 笔记本电脑上测试我们的 AI 程序耗时，对手是 N17-Hotel 组友情提供给我们的 AI。如果将圈地函数用技术组提供的 wanderer 中的 wander 函数来替代，那么 2000 步的耗时约为 8s。这 8s 即为代码架构、Protect 函数等部分的开销。如果加入圈地函数，那么耗时为 22-25s。对比对方的较低时间开销和相差不多的实际效果，我们在相同时间内达到的计算效果与对方还有一定差距。

3.2 结果分析

胜负原因分析

- 在我方先手的 34 局中，我方共胜 16 场，胜率 47.1%，其中，我方正碰对手面积占优取胜的有 4 场，侧碰对手纸带 5 场，对手进入我方区域被撞 3 场，回合耗尽我方面积占优的 3 场，对手自身失误 1 场。
- 在我方后手的 43 局中，我方共胜 29 场，胜率 67.4%，其中，我方正碰对手面积占优 1 场，侧碰对手纸带 21 场，对手进入我方区域被撞 2 场，回合耗尽我方面积占优 4 场，对手自身失误 1 场。

综合来看，我方的主要胜利原因在于攻击函数对于正碰、侧碰纸带、对手进入我方区域几种情形的计算较为精细，成功率很高，而且较高的圈地效率也使我方在回合耗尽的情形下仍能占到一定优势。

- 在我方先手的 34 局中，我方失利 18 局，其中：我方刚出己方区域即被侧碰的有 4 场；在圈地返回的途中出现诡异的转向，被对手撞纸带 4 场；在第一步圈地时自己绕死 1 场（bravo-echo 4）；报错 tuple out of range 有 2 场；被对方完全包围导致报错有 2 场；被对方完全包围导致被侧碰纸带有 1 场；第一步圈地未能及时回归己方 1 场；在角落被己方纸带包围无法返回 1 场（bravo-kilo 2）；回合耗尽面积较小 1 场。
- 在我方后手的 43 局中，我方失利 14 场，其中：我方刚出己方即被撞 4 场；在圈地返回的途中出现诡异的转向，被对手撞纸带 5 场；被对方完全包围导致报错有 5 场。

综合来看，我方的失利原因几乎没有出现面积不占优的情形，主要的原因集中在圈地状态的衔接，圈地与 protect 的衔接，以及对己方形成复连通区域的情形考虑不周等方面。这与我们分模块执行，整合不足有关。此外，还出现了一些我们在之前从未碰到的失利原因，由于数据不足，暂时还无法定位到 bug 的位置。

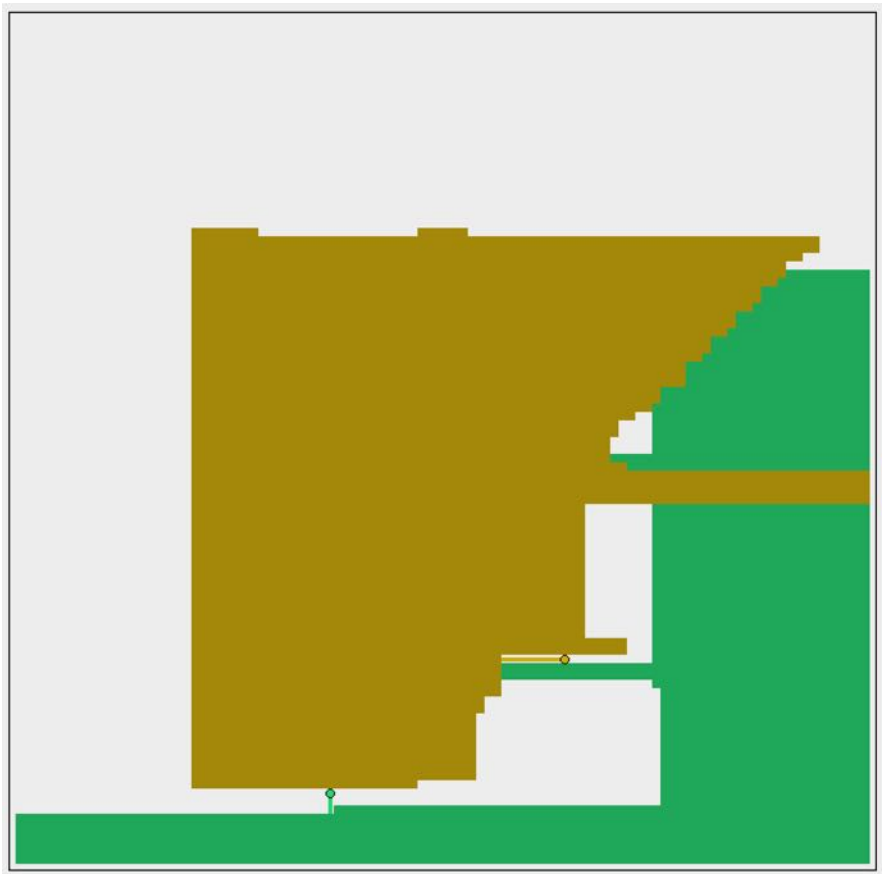
局中局势分析——预先定义的策略是否有效？

在局中，我方在边缘猥琐流与对方周旋的战略在约 1/3 的对战中均发挥了作用；可见，这一步战略的设计是较为成功的。在参与统计的 77 场比赛中，我方在所占面积这一参量上，可以在对决的大部分时间中占优的有 71 场，占比 92.2%，可以说，我方的圈地策略是非常行之有效的。

但是，我方的侧翼防守做得比较糟糕。在中后期圈地面积小于对手的几乎所有比赛中，都能看到区分防守状态和圈地状态的条件在实战中存在着缺陷。如对阵 Quebec 的比赛中，在比赛前期我方往往占据着不小的面积优势，然而当比赛来到中期，Quebec 倾向于在我方倾向于不加考虑的南、北两个侧面进攻我方领地，我方此时往往不能进入防守状态而去边缘阻止其圈地，而是继续在无关紧要的位置徘徊，最后后方被圈走一大块区域，而双方面积对比瞬间转为对我们不利的方向。

3.3 经典战局

四分之一决赛对阵 N17-Quebec

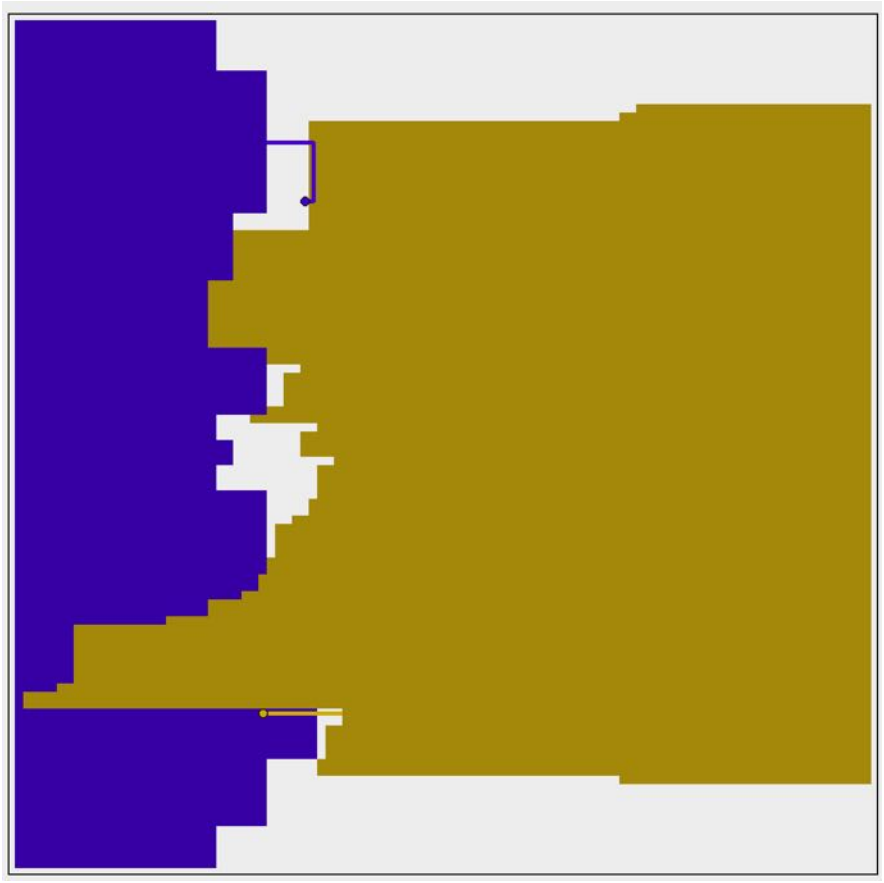


与 Quebec 的第二场比赛中第 1648 步的局势，其中黄色是我方

图 8 的状况集中体现了我们本次 AI 作业的优势和劣势。在游戏中期的第 1648 步时，我方通过最优化的圈地函数已经实现了巨大的面积领先优势，看起来局势将会朝有利于我方的方向移动。但我方的危机已经暗藏在其中，对手从我方区域的侧翼开始进攻，而我方 AI 却浑然不觉，依然在一个效率较低的地方进行圈地，而不是立马前往对手的纸卷位置进行防守。最终身后的领地被对方蚕食，而我方失败于 4000 步耗尽后面积较少。

总结这一场失败的比赛，我们并非没有考虑到需要在侧翼进行防守，也并非没有写出一个较为完善的防守策略函数。问题在于我方缺乏合理的防守判定函数，导致空有防守策略，却无法在每一个合适的时间进入这一策略，只能眼睁睁的看着对方将自己的地盘圈走。

另外一个问题在于，我方对对方领地的攻击性不够强。在这一局势当中，我方的另一种反制措施就是将对方右边的大片领地圈走，但我方的圈地函数并没有考虑到攻击性，只是在边界小打小闹，却坐视后院起火，而最终输掉比赛。



与 Alpha 的第六场比赛中第 1524 步的局势，其中黄色是我方

小组赛对阵 N17-Alpha

图 9 的状况其实和图 8 比较相似，反映着一模一样的问题。区别在于，此时我方有着更大的面积领先优势，而且我方区域的 y 方向宽度也比起图 8 中情况更宽，更有利于圈地。但我们仍然存在着相同的根本问题：防守不力。如果对手的策略包含随位置进行最优化的圈地策略，那么我方的上半部分区域会遭到对方的极大威胁。然而我方 AI 仍然对此浑然不觉，依然在效率不高的地方圈地而不是回去防守，圈地时对对方区域的杀伤也不大。幸好对手的 AI 选取了保守的圈地策略，在我方距离仍然很远时，就满足于一次较小面积的圈地

而返回本方区域，才不至于使我方损失巨大。因为对手的保守策略，我们最终赢下了这一局。但站在事后诸葛亮的角度，对手完全有机会利用我们的策略漏洞实现惊天大逆转。

4 实习过程总结

4.1 分工与合作

依照我们对游戏进行中不同模式的理解，我们对不同的模式下的策略进行分工。与此同时，我们还需要一个代码的整体逻辑架构来实现不同模式的转换和一些所有模式都需要用到的函数“基础设施”。

由于我们全组都不会使用 Git 或其他分布式版本控制系统，我们的代码库存于公共 OneDrive 上，OneDrive 文件夹在 Windows 系统上自动更新的特性让它对于我们这一个不很复杂的工程非常方便，只需要联网就自动更新其他人添加或更改过的代码，对合作效率的提高非常有利。

我们的合作方式以线下讨论为主，线上讨论为辅。由于打字的信息交流效率较低，我们大多数时间采取线下讨论的形式。线下讨论又分全体讨论和部分人讨论，后者主要是在分工时负责的部分需要联动配合的同学之间。我们小组为了本次实习作业，一共进行了 12 次全组讨论，分别在 5 月 25/27/29/31，6 月 1/5/7/8/9/10/11/12 日，非全组参加的讨论数量更多。其中临近截止时间的几天，小组 5 名成员几乎将所有精力都投入在本次作业上，其中最后一天为了优化算法、解决问题，全组成员先是在物院地下室几乎彻夜未眠后，又在二教 5 楼坚持到下午 2 点提交为止。我们小组选择的开会地点包括理教、二教、地学楼、经济学院楼、化学学院楼和物理学院楼（甚至还包括凌晨三点的松林包子铺），遗憾的是，因为全组成员讨论得过于入迷（认真脸），有很多次开会都忘记拍照片，因此只存有 12 次全组讨论中 5 次的照片。

6 月 12 日凌晨 4 点，组内成员在物院地下自习室小憩（请找出 4 位在睡觉的同学）

依次为 5.29, 6.05, 6.07, 6.10, 6.12 日组会合照

4.2 经验与教训

刚拿到这个大作业时，我们几乎束手无策，每个人的想法都不同。而在这种情况下，小组讨论便显得极其重要，它能帮助我们统一目标，明确分工，高效地完成各个模块，这是我们第一个非常宝贵的经验。这不禁令人想到经济学中的一个——比较优势，在完成这项大作业的过程中，每个人的擅长之处不尽相同，有人擅长于解决防守方面的问题，喜欢思考和分类讨论，但不擅长进行圈地效益分析，亦即他完成防守函数的机会成本更小；有人在绝对意义上在圈地与防守上均优于其他人，但他更擅长于对于圈地，也就是说让他来完成圈地的机会成本更小。在这种情况下，我们选择让擅长防守的那个人完成防守模块，另外一个人来完成圈地函数，这样的组合充分利用了比较优势，因此是有效率的。这也是分工明确带来的好处。第二条宝贵的经验便是勤于写注释，这有助于各个模块的交流、策略配合、后期润色代码以及 debug，当然也是有助于撰写书面报告的。正是这种优秀的书写代码的习惯帮助我们在短时间内以较高的效率完成了本次的大作业。第三条宝贵的经验则是在 debug 过程中，加入标识符增进函数之间的信息沟通以及及时打印状态标识符来帮助我们了解程序运行的进程，在我们提交代码的前一天晚上，我们通过这样的手续实现了将一个几乎残废的 AI 变成一个比较正常的 AI，可见优秀的 debug 方法非常有助于后期调试以及提高效率。第四条宝贵的经验是及时沟通减少重复工作的发生，并且细分函数，使

得函数的通用性大大提高，许多基础函数可以应用到更高层级的功能性函数中，既提高了代码整体的条理性和可读性，也减少了重复书写代码所浪费的时间。最后一条经验也是非常有借鉴意义的工作——通过标识符的方法促进函数之间的沟通，避免了大量使用 if 来进行 AI 的架构，提高了整体组装的效率和大大减小了后期维护于修改的难度。

在一路磕磕绊绊中，我们也得到了许多深刻的教训。在编写代码的时候，我们采用了大量的 if 结构，这给我们后续的修正带来了较大的阻力，正确的做法应该是在编写之前打好明晰的结构，并且在写的时候注意总结、提炼相似结构，这样能够提高可读性以及减少 debug 的阻力。其次，负责各模块的同学对于其他模块的了解程度不够，导致在最后整合时出现了交叉模块的联系困难以及搭建总框架时配合欠佳，这直接拖慢了进程，以至于最后两天的时间明显偏紧甚至导致了一些低级错误的发生。

我们认为，我们在游戏中采取的战术思想极大限制了我们的游戏打法，以至于我们面对某些类型的对手时候束手无策。我们的算法在各个队伍中属于相对复杂的算法，但实际效果并不是很好。我们的圈地策略限制在依托于垂直于 x 方向的边界，向东西两个方向圈地。这个策略适用于对手喜欢与我方正面缠斗时，一旦对方选择侧面迂回，我们很可能在短时间内丧失优势，自己的一大块地盘被别人悄无声息地圈走。时间的紧迫更加剧了这一点，我们单独测试防守策略和圈地策略时，往往能运转良好，但是在比赛当中需要二者结合时，需要在合适的情况下转换模式时，我们的 AI 显得迟疑不定，往往会在不重要的地方浪费时间，而不是去边界防守、白白被对手圈走一大块地（如四分之一决赛对阵 Quebec）。在路径选择的预期上，我们的防守算法大多是基于实时情况做出决策，而没有去进行预测和模拟，这导致在某些极端情况下无法及时回到安全的地方进行规避，并且没有考虑到圈地收益，于是防守算法比较粗糙，很多情况下会失效，这也直接导致后期博弈的低效率性，一旦游戏被拖入后期，往往上就会失去前期大面积圈地的优势从而输掉比赛。我们的整体策略显得有些单一、对不同类型对手的适应度较差，这是我们止步于八强的一个重要原因。同时，优先级的不够精细也是一个重要问题，经过大量的比赛，我们发现许多可以直接击杀对手的场景中因优先级的问题使得 Attack 模块被跳过，最终导致没能击杀对手。这说明我们的代码整体架构还不够完善，并不能在所有合适的时候都进入合适的策略模式。我们此次的游戏 AI 并没有参加热身赛，只是利用组员的私人关系和几个小组进行了友谊赛，因此无论是游戏整体思路，还是具体的函数实现，都存在较多想当然之处，与实际游戏情况不完全符合。因此，在之后类似的工程中，我们会更迅速地完成游戏思路的构造和代码的实现，并将后期的时间更多投入到优化游戏思路和解决代码问题当中。

4.3 建议与设想

我们由衷地感谢陈斌老师、技术组的同学和助教团队能给我们这样一个展示的平台，让非计算机专业的我们第一次体会到编写一个较大规模的程序是什么样的感觉。在我们实际体验代码编写的过程后，对本次大作业有一些愚见：

1、大作业实践总时长偏短，以至于在此时间段内需要投入极大密度的时间于这一作业上，可能会耽搁其他课程的进度。我们建议适当延长大作业实践的总时长，可以适当提前发布实践任务。对比信科的课程大作业，一般期中左右就已经布置下达，到期末季完成，时间相对更充裕。

2、我们在完成大作业的过程中，从与其他组的交流中学到了很多，但是热身赛的数量有限、时间固定，而且举办一次热身赛对技术组是极大的负担。我们认为如果有条件的话，可以提供一个 AI 的线上对战平台（类似于 botzone），只要将代码放上去就可以等待他人来挑战，这样更有利于我们在一次次对战当中进步。

我们衷心地希望陈斌老师的数据结构与算法课能够越办越好，期待以后出现更有吸引力的大作业形式！

5 致谢

1. 感谢陈斌老师、技术组的同学和助教团队，没有他们的创意和提供的基础设施，我们不可能从这次大作业中获得如此大的乐趣和满足感

2. 感谢 N17-Quebec、N17-Hotel 和 F17-Golf 组慷慨地与我们交换了 AI 源代码，使我们能够利用这三组设计良好的 AI 进行测试，对最后两天进度仍然较慢的我们提供了极大的帮助

3. 感谢我们的热心同学苏雨峰、郭一潇，以他们丰富的经验为最开始不知所措的我们指明了方向

4. 感谢paper-io.com网站提供的纸带圈地对战平台，为我们熟悉游戏规则、设计战术思路提供了极大的便利

第三章　N17_Charlie 报告

苏文霖 *，王颉，孙鎏磊，文飞黄，周宏伟

摘要：初步的算法设计以自保为根本目的，设计出来的 AI 片面地重视了安全性，忽视了其圈地效率。升级后的 AI 也是在保证安全的前提下，更加注重圈地效率。算法的结构十分简单，大量使用判断语句，然后分阶段地循环它们，借助步数的限制来终止 AI 的进程。总体说来，这个 AI 是一个保守一点的 AI，没有攻击性，只是在避开对手的攻击时完成圈地运动，最终还是要在耗尽步数时以圈地面积来衡量胜负。总体说来圈地效率中等，遇到攻击性较强的对手时容易死亡，因此在热身赛的时候排位在中游偏下的位置。总体说来，我方 AI 先手时更有优势，后手时更容易发生撞墙事故，决策十分迅速。

关键字：paper.io，小傻 AI，安全绕大圈策略，方形圈地

1 算法思想

1.1 总体思路

第一版：

采用的是在安全距离内进行圈地的保守的策略，策略的主要方案是假设我们初始位置在左侧，一开始就直接以画正方形的形向对面纸头的初始位置靠拢，即若初始方向是右，就直接出发，若是上下左，就以最快速度回到右边出发点出发（第一个蓝色箭头点）。出发后，边向右前进边探测对方纸头离我们纸头的 X 轴距离，直到这个距离等于我们已向右前进距离的三倍，就立即转头向上完成这个正方形，这样可以保证绝对安全（敌方纸头就算以最快速度向我们撞过来我们也够不着）。画完第一个正方形后回到第二个蓝色箭头点的位

置，之后重复这样的操作，知道连边长为一的正方形都画不出，即此时已经离对方纸头的 X 轴距离很近了（假设为第四个蓝色箭头点）。

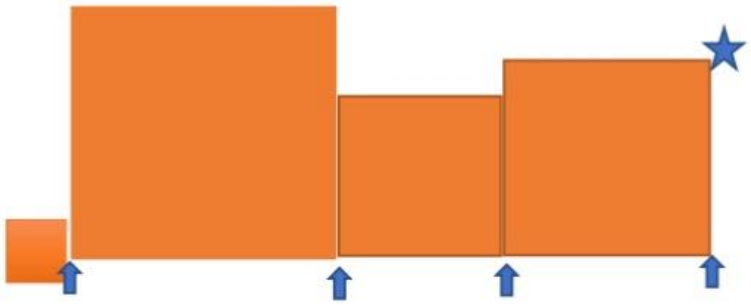


图 1-1. 初始圈地阶段

在第四个蓝色箭头点处，进行探测，如果方圆 3 个像素格以内没有敌方纸头，就下转开始画正方形，边长等于敌我纸头距离的三分之一（如果不是整数就取整）；如果 3 个像素格内有敌方纸头，就向上移动到五角星位置，再探测敌我纸头的位置，向上画正方形，边长取法跟在第四个蓝色箭头处一致。

图 1-2. 敌人进击时的圈地阶段

接下来的说明我们选取向下画方形的情况为例进行。此时发现我们所画的形状是不规则的，接下来的步骤就是把它补全。画完这个方形后我们在蓝色三角形处，随后以最快的速度前进到绿色三角形处向上补充，发现缺了用蓝色双向箭头圈起来的一块，补齐（如下图）。

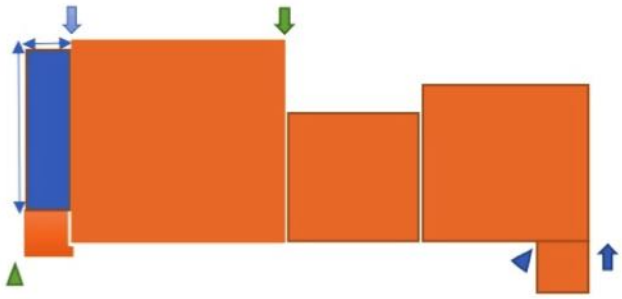


图 1-3. 补缺阶段

补充完毕后我们位于蓝色箭头处发现这条边有缺口，前进到绿色箭头处，准备向右补充发现右边有两个缺口，即以方形补充两次才能完整补齐，我们一次只补充一个缺口（如下图）。

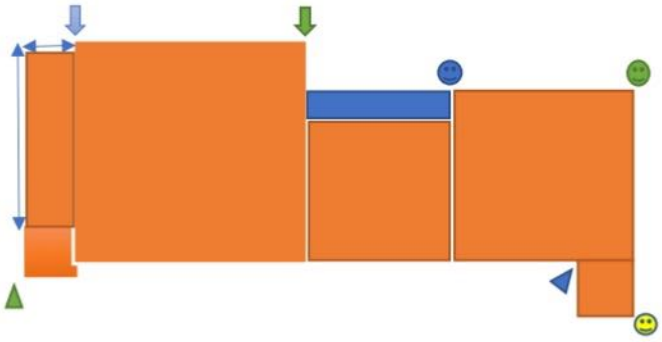


图 1-4. 上方补缺

补充完毕后我们位于蓝色箭头处发现这条边有缺口，前进到绿色箭头处，准备向右补充发现右边有两个缺口，即以方形补充两次才能完整补齐，我们一次只补充一个缺口（如下图）。

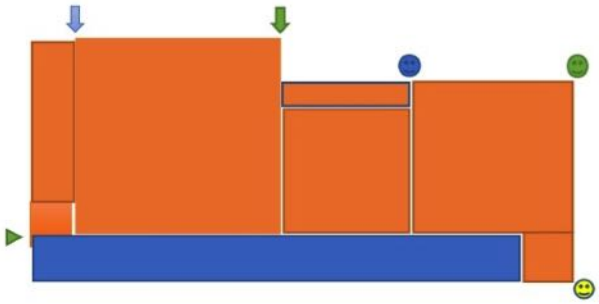


图 1-5. 下方的填充

由于只有一个缺口，用一个矩形就可以补齐，遂直接补齐，补齐后位于绿色三角处，发现这条边已经完整了，由于向左扩充不会有危险，于是我们直接扩充。

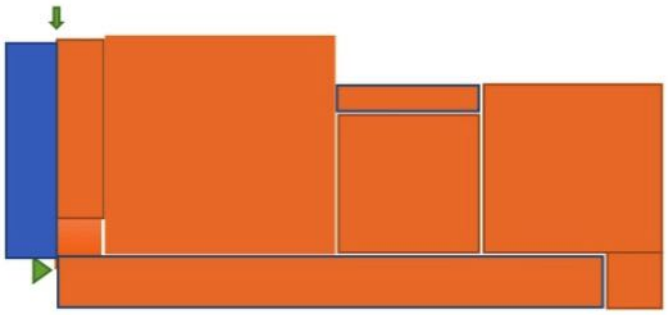


图 1-6. 扩张圈地阶段

补充完毕后我们位于绿色箭头处，发现这条边还有缺口，而且用一个矩形就可以补齐，直接补齐。

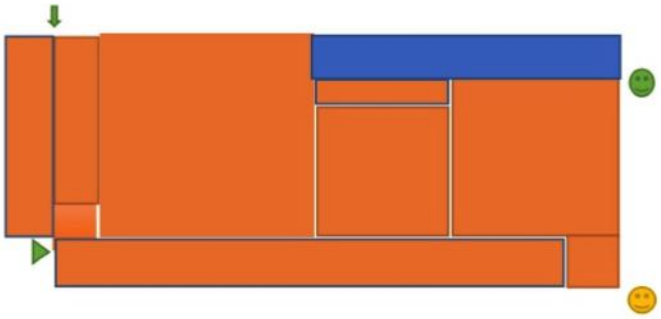


图 1-7. 继续补齐

此时我们位于绿色笑脸处，不向右扩充，直接前进到黄色笑脸处，发现左边这条边有一个缺口，补齐。照这样的步骤规律继续下去，在上，下，左边有缺口就补充，没缺口就画新的方形，最后把上下左的空白部分全部填满，一般这样的策略在填满之前绝对不止 2000 步，所以一旦出现 2000 步前就填满的情况，那就绕边长继续跑就好。以上就是我们的策略。

经过实际代码编写和实验，发现此策略尽管十分安全，不可能会被对手直接碰撞而死掉，但是发现这个策略存在着许多缺点，其一是圈地效率十分低下，其二是策略没有简单使用的且可有效重复的判定方法，代码实现非常困难。于是在后续，我们小组抛在第一种策略的基础上，重新制订了新的方法，进一步提高了圈地运动的效率。

版本二：

调整后的策略基本上还是采用敌我距离太近时就往已圈区域躲避的基本策略，但是对安全性的偏重有下降，并对初始圈地方式做了重新规划，主要的思路是在第一阶段和第二阶段尽最大的能力去圈地，因为这个过程对手也倾向于疯狂圈地，所以这个阶段稍微放松对安全性的关注会使得圈地的效果更好。

第一阶段起始就往北出发圈地，抵达边缘后折返回来进行圈地；

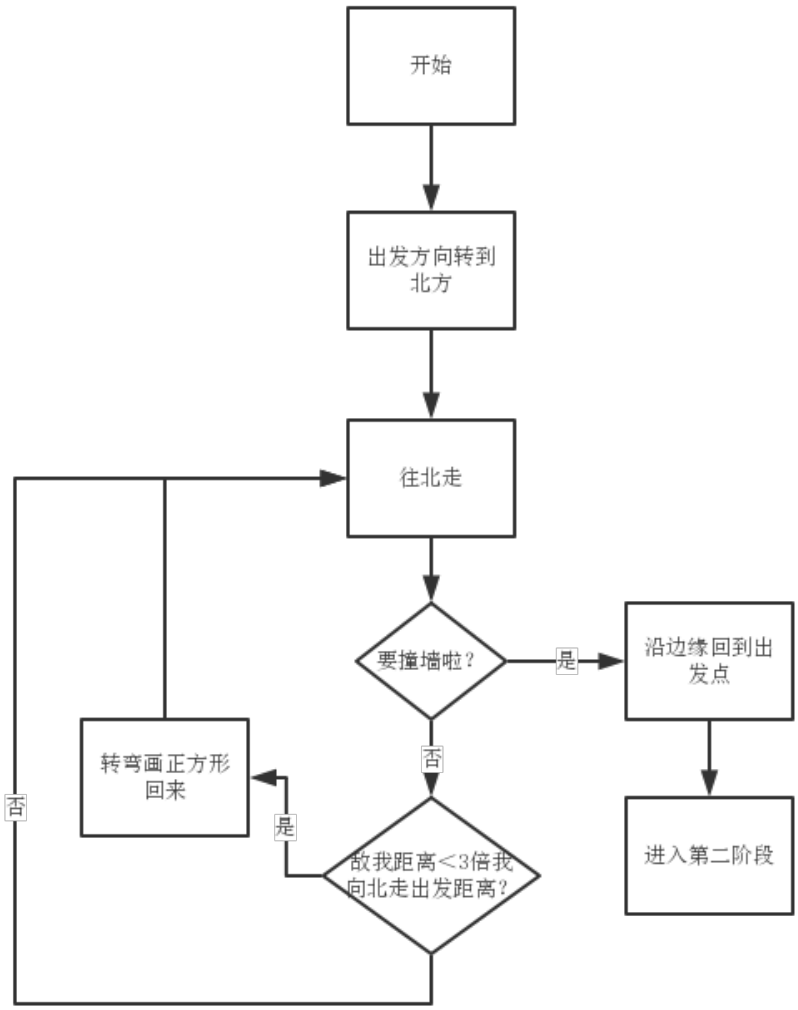
第二阶段向南出发在安全距离合适时向南边出发并进行圈地；

第三阶段继续向南出发进行圈地并且抵达边界；

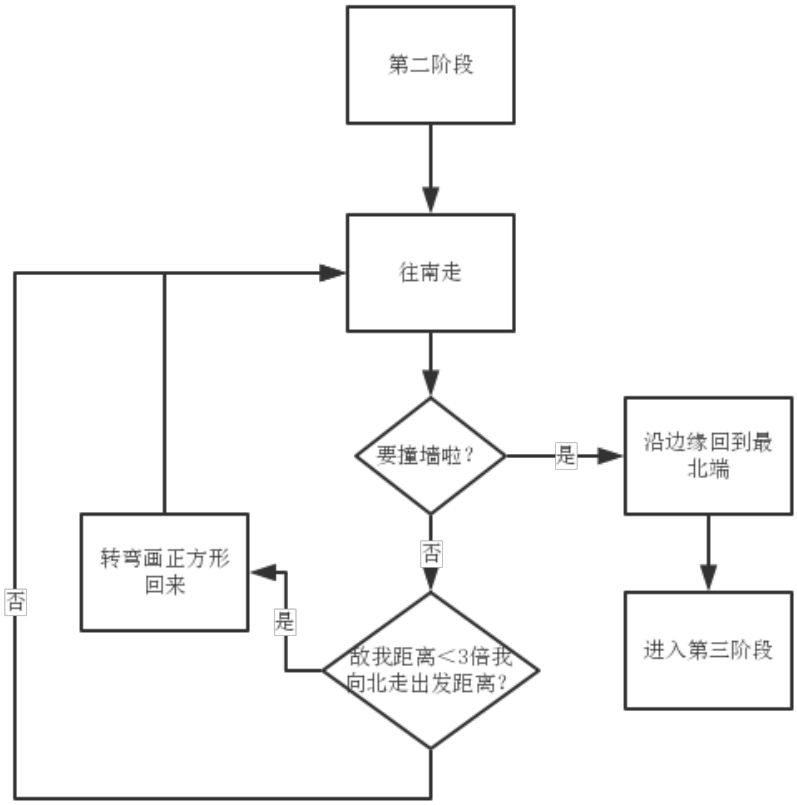
第四阶段沿边界画正方形，争取包围对手；

第五阶段沿边界尽力蚕食对方领地的同时，暗地里做将其包围的打算。

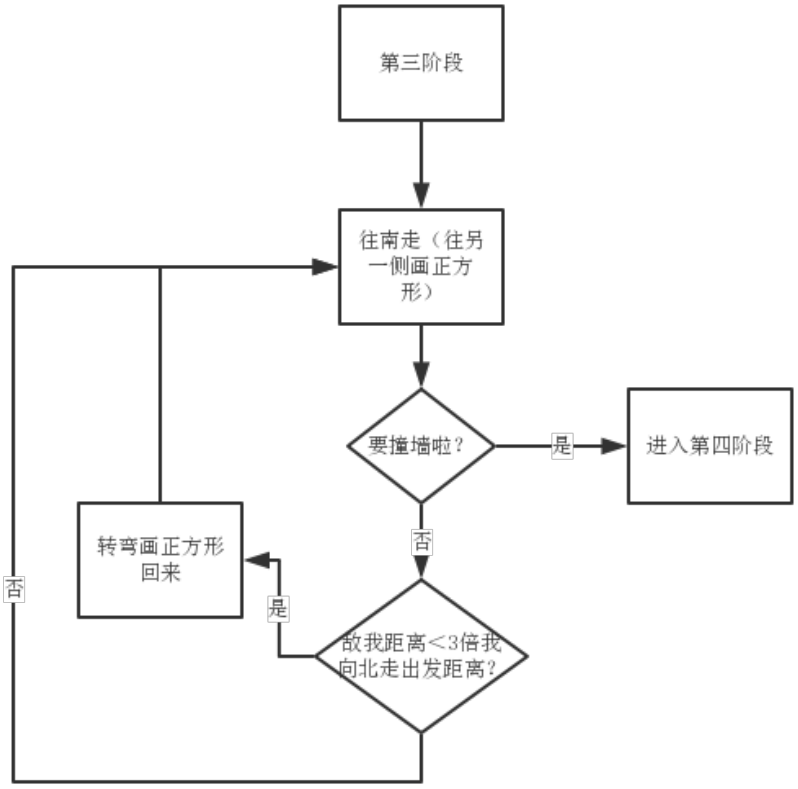
1.2 算法流程图



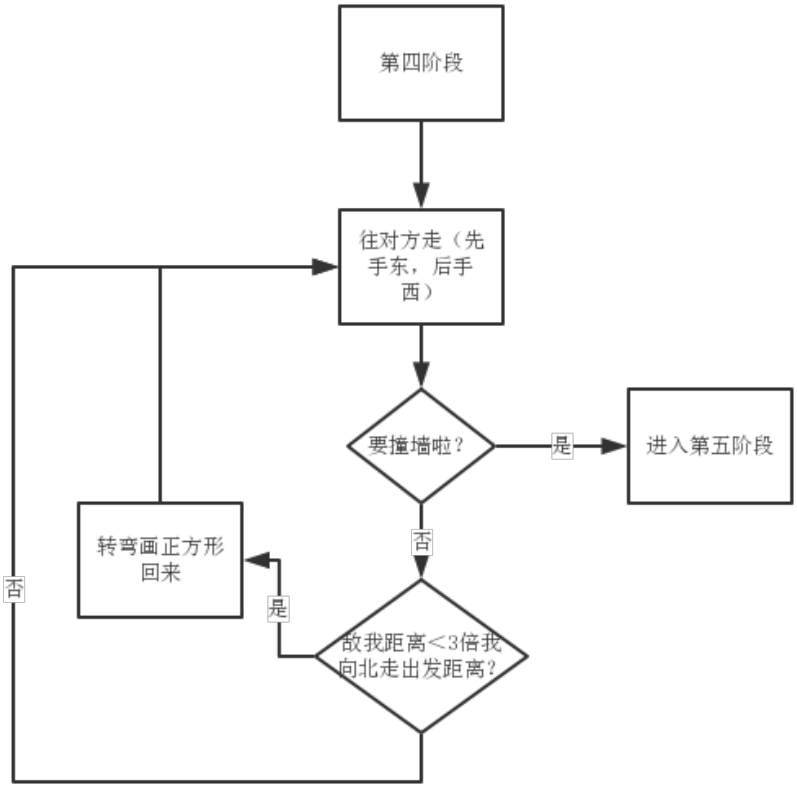
第一阶段


第二阶段
往南走
要撞墙啦？
是
沿边缘回到最北端
否
转弯画正方形回来
否
是
故我距离＜3倍我向北走出发距离？
进入第三阶段

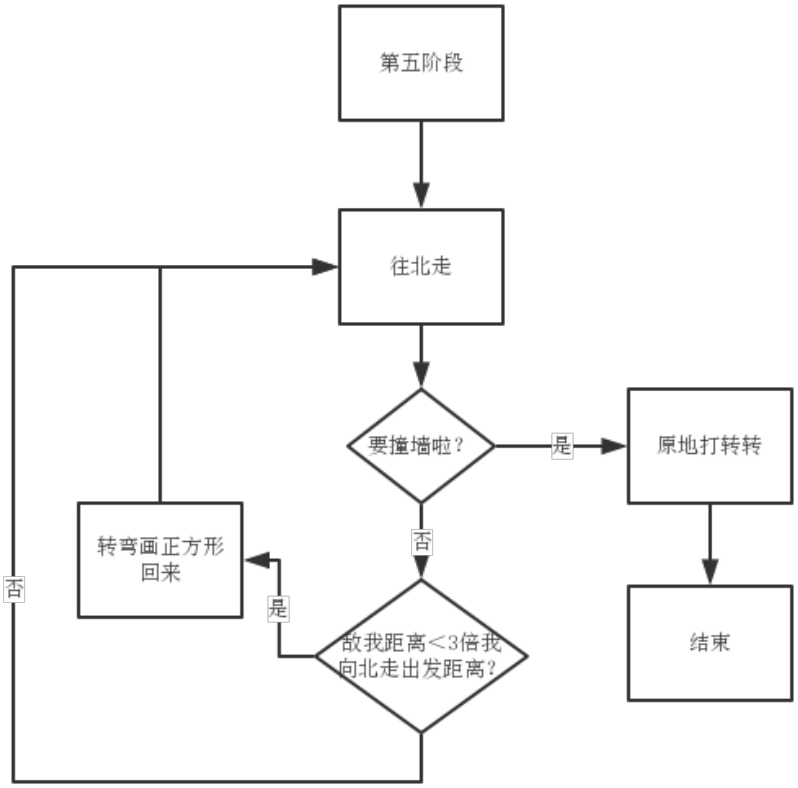
第二阶段


第三阶段
往南走（往另一侧画正方形）
要撞墙啦？
是
进入第四阶段
否
转弯画正方形回来
是
否
故我距离＜3倍我向北走出发距离？

第三阶段


第四阶段
往对方走（先手东，后手西）
要撞墙啦？
是
进入第五阶段
否
转弯画正方形回来
否
是
敌我距离＜3倍我向北走出发距离？

第四阶段



第五阶段

1.3 算法运行时间复杂度分析

从热身赛中的耗时情况可以看出，4000 步走完后，时间消耗在 0.01s 以内，说明算法运行的速度非常快，分析代码可以知道，这个 AI 里面没有复杂的数据结构，每一个循环都是相同的判别方法，可以估算出算法的时间复杂度应该是常数级的，即时间复杂度为 O（1）。时间额度基本没有耗费。

2 程序代码说明

2.1 数据结构说明

本小组设计的程序结构十分简单，程序的主题基本由 if 条件判断语句构成，虽然杀伤力不强，但是十分简洁高效。

2.2 函数说明

2.2

```
1.    def begin(field, me, storage):
2.        if storage['stepp'] == 0:
3.            storage['start_point'][0], storage['start_point'][1] = me['x'], me['y']
4.            # print (storage['start_point'])
5.
6.        # print(field[me['x']][me['y']])
7.        if not storage['start']:
8.            #对不同初始朝向确定转向
9.            #最远的一方（这里为南侧）需要多绕一下
10.            if field[me['x']][me['y']] == me['id'] and \
11.                            field[me['x']][me['y'] - 1] == None and \
12.                            field[me['x'] + 1][me['y']] == None and \
13.                            field[me['x'] + 1][me['y'] - 1] == None:
14.                # storage['stepp'] = 1
15.                return "L"
16.            if me['direction'] == 0:  #东
17.                return "L"
18.                storage['start'] = True
19.            elif me['direction'] == 2:  #西
20.                return "R"
21.                storage['start'] = True
22.            elif me['direction'] == 1:  #南
23.                return "L"
24.            else:  #北
25.                storage['start'] = True
26.
27.        #都回到最初的起点
28.        #初始化步数记录为（避免之后绕圈时自杀），进入画正方形函数（）  1first_round
29.        if storage['start'] == True:
30.            storage['stepp'] = 1
31.            storage['mode'] = 'first_round'
```

```
32.             return ""
```

随机刷的出发方向都转到往北方

图 2-1 向上出发

```
1.    def first_round(field, me, storage):
2.        storage['stepp'] += 1
3.
4.        #向上走，种情况判定开始转弯： 3
5.        #、敌方距离小于倍我方走出来的纸带长度 13
6.        #、北面撞墙则转弯 2
7.        #、走出的距离超过起始点坐标离边上的距离（防止左转之后撞墙） 3x
8.
9.        nexty = me['y'] + directions[me['direction']][1]
10.
11.        if not storage['temp'] and dist(me, storage['enemy']) <= 3 * storage['stepp'] +
                                          1 and \
12.                storage['stepp'] < min(storage['start_point'][0], len(field)-storage['
                                        start_point'][0]) \
13.                or nexty < 0 and me['direction'] == 3:
14.            # print ("here")
15.            #如果走到北边撞墙发生的转弯，则进入函数 modeback
16.            if nexty < 0 and me['direction'] == 3:
17.                storage['temp'] = False
18.                storage['mode'] = 'back'
19.                return storage['dir'][me['id']]
20.            storage['maxl'] = storage['stepp']
21.            storage['stepp'] = 0
22.            storage['count'] = 0
23.            storage['temp'] = True  #临时变量，记录是否发生第一次转向
24.            return storage['dir'][me['id']]
25.        if storage['temp'] and storage['stepp'] == storage['maxl']:
26.            storage['stepp'] = 0
27.            storage['count'] += 1  #这一部分会循环次（转次） 33
```

```
28.             if storage['count'] > 3:
29.                 storage['temp'] = False
30.                 return ""
33.             return storage['dir'][me['id']]
34.     #返回函数，和之后会调用 first_roundsecond_round
#基本思路是沿着边界回到起始点的位置
```

向北方画正方形，3 种情况判定开始转弯：1、敌方距离小于 3 倍我方走出来的纸带长度 2、北面撞墙则转弯 3、走出的距离超过起始点 x 坐标离边上的距离（防止左转之后撞墙）

图 2-2 first_round（左先手，右后手）

```
1.    def back(field, me, storage):
2.        idTurn = me['id']
3.        if storage['second']:
4.            if me['id'] == 1:
5.                idTurn = 2
```

```
            else:
                idTurn = 1
        if me['direction'] % 2:  #轴不出界 y
            nexty = me['y'] + directions[me['direction']][1]
            if nexty < 0 or nexty >= len(field[0]):
                return storage['dir'][idTurn]
        else:  #轴不出界 x
            nextx = me['x'] + directions[me['direction']][0]
            if nextx < 0 or nextx >= len(field):
                return storage['dir'][idTurn]

        if not storage['second'] and not storage['temp'] and me['y'] == storage['
                                            start_point'][1]:
            # print ("di")
            storage['temp'] = True
            return storage['dir'][idTurn]

        #返回之后进入到 first_roundsecond_round
        if not storage['second'] and me['x'] == storage['start_point'][0]:
            storage['stepp'] = 0
            storage['temp'] = False
            storage['mode'] = 'second_round'
            if me['id'] == 1:
                return storage['dir'][2]
            else:
                return storage['dir'][1]

        #返回之后进入到 second_roundthird_round
        #此时返回到了最北端
        if storage['second'] and me['x'] == storage['start_point'][0]:
            storage['stepp'] = 0
            storage['temp'] = False
            storage['mode'] = 'third_round'
            return storage['dir'][idTurn]

#往南走（先手顺时针，后手逆时针）
```

撞墙了往回走，走到初始点水平位置

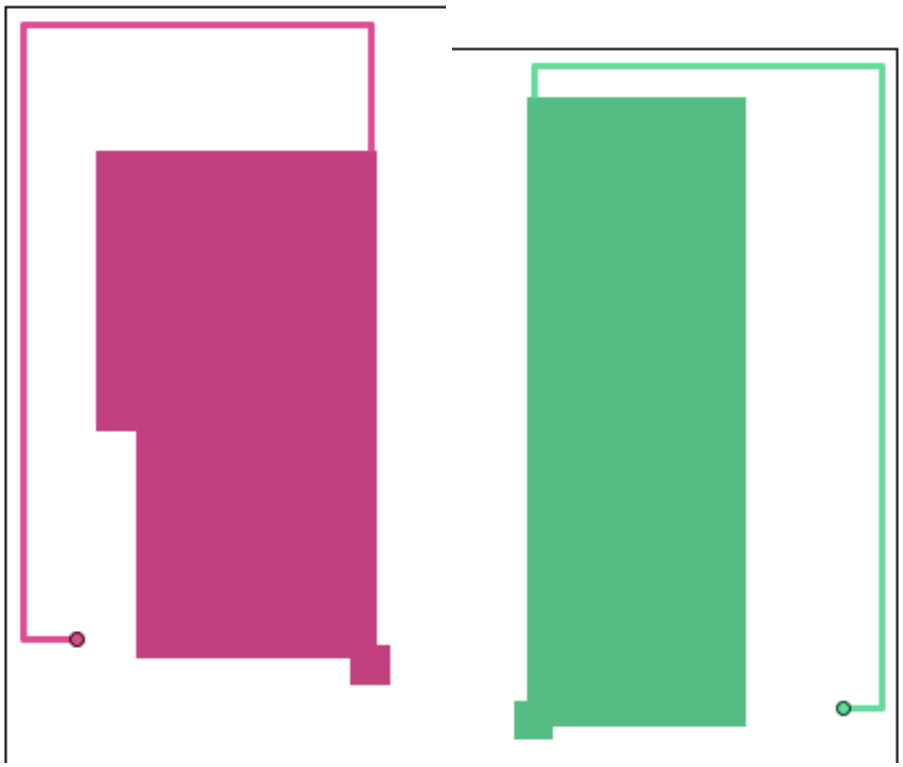


图 2-3 第一次往回走（左先手，右后手）

```
1.    def second_round(field, me, storage):
2.        # print(min(storage['start_point'][0], len(field)-storage['start_point'][0]))
3.        if me['id'] == 1:
4.            idTurn = 2
5.        else:
6.            idTurn = 1
7.        storage['stepp'] += 1
8.
9.        #撞墙判定为向南到头则转弯
10.        nexty = me['y'] + directions[me['direction']][1]
11.        if not storage['temp'] and dist(me, storage['enemy']) <= 3 * storage['stepp'] +
                                                1 and \
12.            storage['stepp'] < min(storage['start_point'][0], len(field)-storage['
                                               start_point'][0]) \
13.             or nexty >= len(field[0]) and me['direction'] == 1:
14.             # print ("here")
15.             if nexty >= len(field[0]) and me['direction'] == 1:
16.                 storage['temp'] = False
```

```
17.                 storage['second'] = True
18.                 storage['mode'] = 'back'
19.                 return storage['dir'][idTurn]
20.             storage['maxl'] = storage['stepp']
21.             storage['stepp'] = 0
22.             storage['count'] = 0
23.             storage['temp'] = True  #临时变量，记录是否发生第一次转向
24.             return storage['dir'][idTurn]
25.         if storage['temp'] and storage['stepp'] == storage['maxl']:
26.             storage['stepp'] = 0
27.             storage['count'] += 1  #这一部分会循环次（转次） 33
28.             if storage['count'] > 3:
29.                 storage['temp'] = False
30.                 return ""
31.             return storage['dir'][idTurn]
32.
#从最北端向南走（先手顺时针，后手逆时针）
```

往南边走

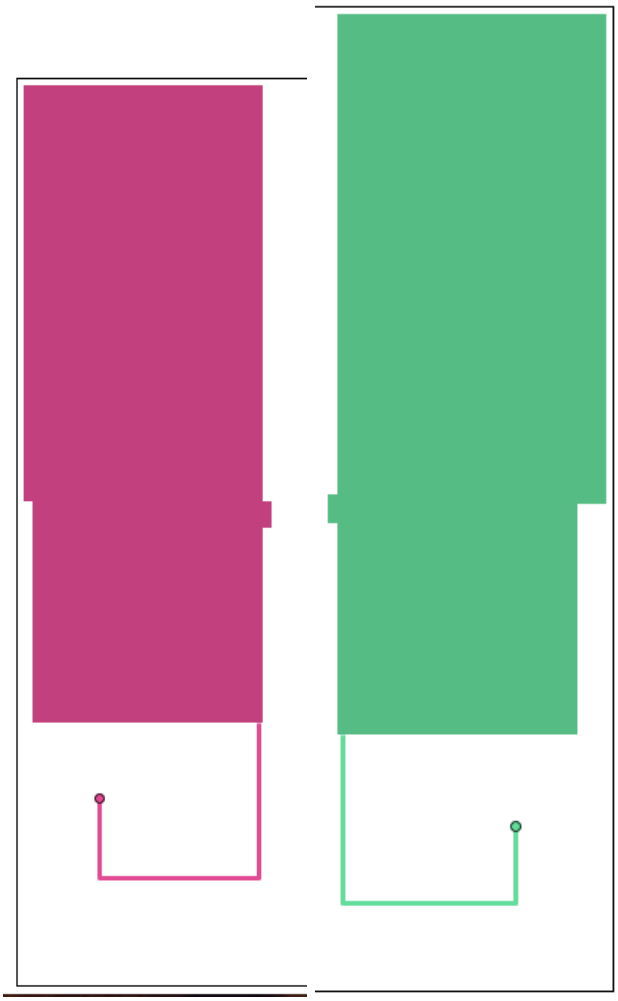


图 2-4 second_round（左先手，右后手）

def back(field, me, storage):

第二次调用 back，这时已经圈好了边上的地（先手左侧，后手右侧），回到北边顶端

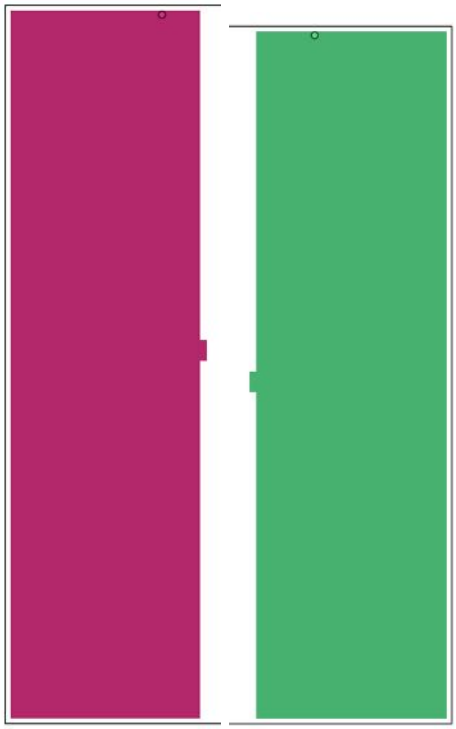


图 2-5 第二次 back（左先手，右后手）

```
1.    def third_round(field, me, storage):
2.        storage['stepp'] += 1
3.        nexty = me['y'] + directions[me['direction']][1]
4.        if not storage['temp'] and dist(me, storage['enemy']) <= 3 * storage['stepp'] +
                                                1 \
5.                or nexty >= len(field[0]) and me['direction'] == 1:
6.            # print ("here")
7.            if nexty >= len(field[0]) and me['direction'] == 1:
8.                storage['stepp'] = 0
9.                storage['temp'] = False
10.                storage['mode'] = 'forth_round'
11.                return storage['dir'][me['id']]
12.            storage['maxl'] = storage['stepp']
13.            storage['stepp'] = 0
14.            storage['count'] = 0
15.            storage['temp'] = True  #临时变量，记录是否发生第一次转向
16.            return storage['dir'][me['id']]
17.        if storage['temp'] and storage['stepp'] == storage['maxl']:
18.            storage['stepp'] = 0
```

```
            storage['count'] += 1  #这一部分会循环次（转次） 33
            if storage['count'] > 3:
                storage['temp'] = False
                return ""
            return storage['dir'][me['id']]

#先手往东，后手往西
```

从顶端往南边走，判定条件与 first_round 和 second_round 一样

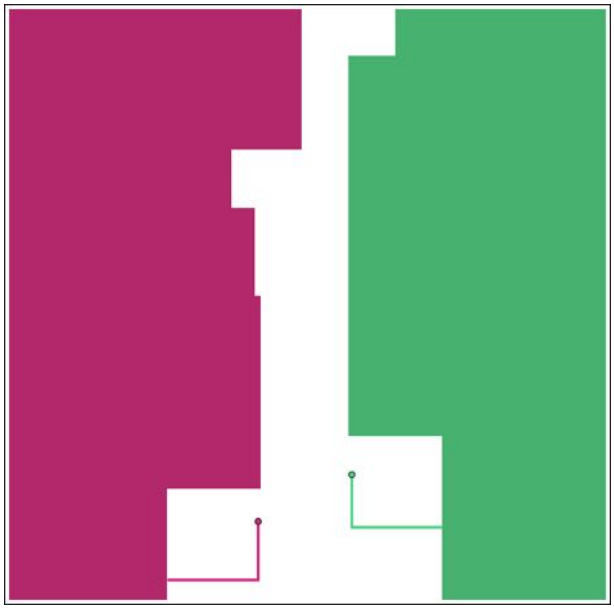


图 2-6 third_round（左先手，右后手）

```
    def forth_round(field, me, storage):
        storage['stepp'] += 1
        nextx = me['x'] + directions[me['direction']][0]
        if not storage['temp'] and dist(me, storage['enemy']) <= 3 * storage['stepp'] +
                                                1 \
                or nextx >= len(field) and me['direction'] == 0\
                or nextx < 0 and me['direction'] == 2:
            # print ("here")
            if storage['final']:
                storage['temp'] = False
                 # storage['turnover'] = storage['stepp']
                 storage['stepp'] = 0
```

```
12.                 storage['mode'] = 'fifth_round'
13.                 return storage['dir'][me['id']]
14.             if nextx >= len(field) and me['direction'] == 0 \
15.                or nextx < 0 and me['direction'] == 2:
16.                 storage['final'] = True
17.
18.             storage['maxl'] = storage['stepp']
19.             storage['stepp'] = 0
20.             storage['count'] = 0
21.             storage['temp'] = True  #临时变量，记录是否发生第一次转向
22.             return storage['dir'][me['id']]
23.         if storage['temp'] and storage['stepp'] == storage['maxl']:
24.             storage['stepp'] = 0
25.             storage['count'] += 1  #这一部分会循环次（转次） 33
26.             if storage['count'] > 3:
27.                 storage['temp'] = False
28.                 return ""
29.             return storage['dir'][me['id']]
30.
#往北走到头，然后在自己领地里转圈（因为的步数基本上走完了） 2000
```

走到头后，先手往右边画正方形，后手往左边画正方形

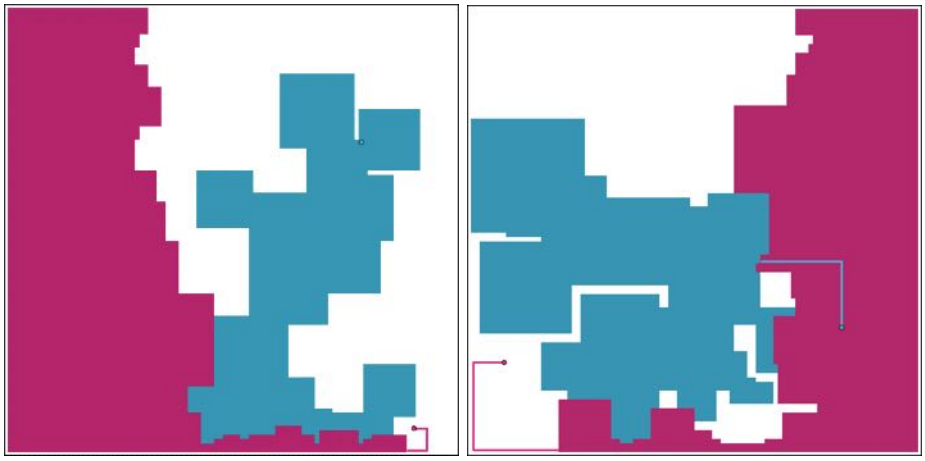


图 2-7 forth_round（左侧红色先手，右侧红色后手）

```
1.    def fifth_round(field, me, storage):
2.        storage['stepp'] += 1
3.
```

```
        nexty = me['y'] + directions[me['direction']][1]

        if not storage['temp'] and dist(me, storage['enemy']) <= 3 * storage['stepp'] +
                                                1 \
           or nexty < 0 and me['direction'] == 3:
            storage['maxl'] = storage['stepp']
            storage['stepp'] = 0
             storage['count'] = 0
             storage['temp'] = True  #临时变量，记录是否发生第一次转向
             return storage['dir'][me['id']]
         if storage['temp'] and storage['stepp'] == storage['maxl']:
             storage['stepp'] = 0
             storage['count'] += 1  #这一部分会循环次（转次） 33
             if storage['count'] > 3:
                 storage['temp'] = False
                 return ""
return storage['dir'][me['id']]
```

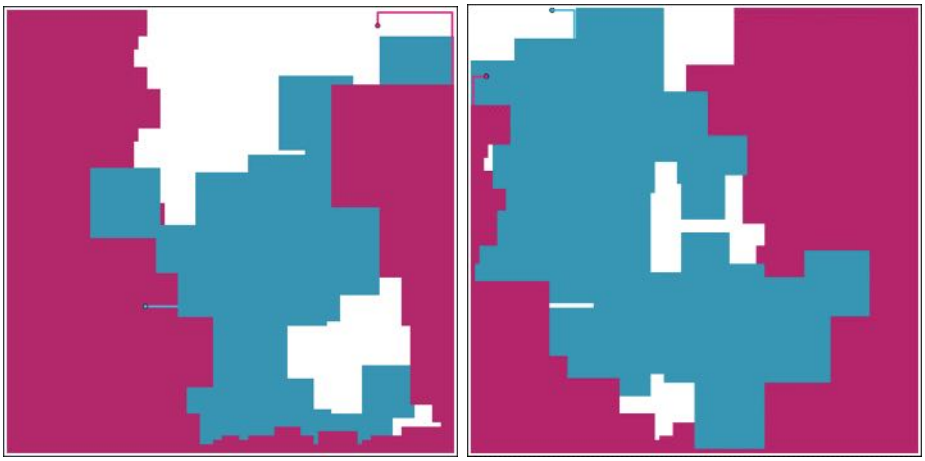


图 2-8 fifth_round（左侧红色先手，右侧红色后手）

2.3 程序限制

在对战的过程中可能会发生撞墙自杀的情况，说明算法中对边界的判断存在着一些遗漏；另外也存在着在别人领地内撞到别人纸带的情况，这是因为算法设计中对对手的信息，我们只是用了纸袋头的位置信息，对于纸袋的运行状态的其他细节没有在意。

3 实验结果

3.1 测试数据

实验环境说明：

- 硬件配置：（CPU/内存）1.8GHz Intel Core i5/8GB 1600MHz DDR3
- 操作系统：（名称/版本）macOS High Sierra
- Python 版本：（版本号）Python 3.6.3

基础框架搭建完成之后，采用人机对战以及与样例代码对战的方式进行测试，分析代码边界判定并进行修改。测试结果发现最初直接向对方领域进攻的方法很容易出现问题，结果表现较差，于是我们修改的策略，决定以圈地为主，从己方边界进行包围，再向对方领域包围，更改之后的策略从结果上明显优于之前的策略。

热身赛结果如下：

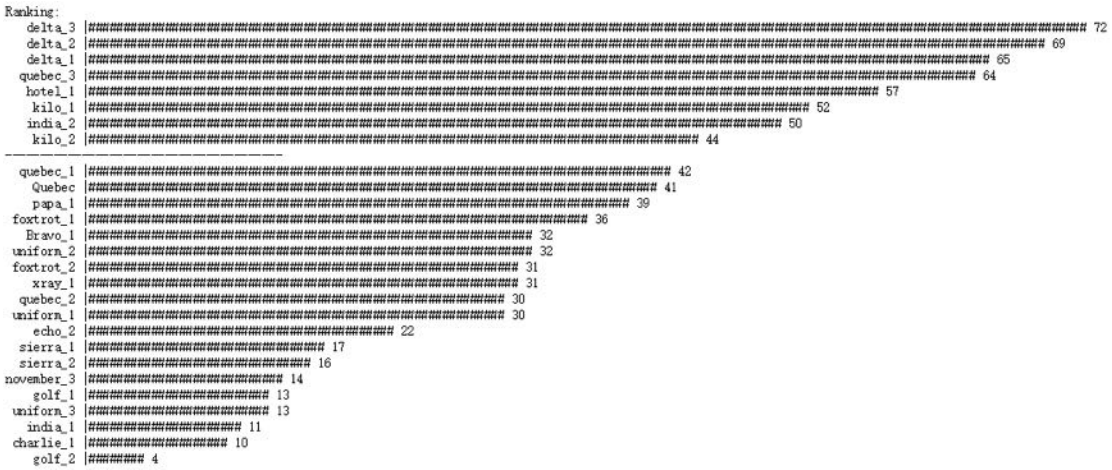


图 3. 热身赛结果

3.2 结果分析

1）初期自我对战结果

我们最初准备参与 6 月 3 日第一次热身赛，准备使用最开始的算法，直接先向对方领域进发，通过画正方形不断扩张领地，之后再回到自身地盘圈地。但是在比赛前夕，我们通过不断测试发现，这种算法存在一定问题，在很多情况下都会出现自己咬断自身纸带死亡。我们通过讨论认为自己的算法还存在不足，需要改进，为了不浪费时间，我们决定先不参加第一次热身赛，而是先自己改进算法。

2）热身赛对战结果

我们使用改进之后的算法参加了第二次热身赛，从结果上看，我们热身赛的结果并不太好。值得庆幸的是，热身赛并没有出现程序出错、超时等我们之前非常担忧的问题。我们小组在热身赛之后对于比赛结果进行了复盘，可以看出通过与其他小组算法的比赛，确实反映出了很多我们自己测试时没有看到的问题。比如在某些极端情况下会让己方自杀，第一阶段和第二阶段的算法判定上存在一些问题，在敌方距离过远的情况下会出现撞墙的情况，这都是我们在之前自己测试时没有发现的。这可能是因为我们对战使用的样例代码比较简单，确实不能反映出太多可能的情况。

另一方面，我们也发现我们的算法相对而言比较简单。于是我们决定首先先改进这一个算法，解决在热身赛中出现的问题，尽量能够做到不出问题；同时，我们也积极讨论，向计算机专业的同学请教，寻找增加机器学习内容的可能性。但是，由于我们准备的时间确实太短，而且之后也没有热身赛让我们检查错误。保险起见，我们最终还是放弃了全部重写策略的打算。于是我们最后针对自己的策略不断修改，花了很长时间不断测试检查 bug，最终提交的也是我们不断修改之后认为没有的问题的版本。

3）算法结果分析

因为我们采用的策略没有涉及到太多复杂的算法，所以基本不需要考虑超时的问题，运算时间方面是没有问题的。我们的策略主要是一个保守型的策略，最优情况下是要求敌方不可能在己方领地外杀死己方，主要难点在于边界判断以及撞墙的判定。最终结果还是基本符合预期的，我们比赛输的原因主要是判定错误导致的自杀，以及圈地面积不如对方。

3.3 经典战局

本部分我们选取了两个对我们的最终代码产生了深远影响的战局，可能在结果上并不算出色，也是我们珍贵的回忆。

首先是我们在与样例代码测试时，发现了一个问题，在某种极端条件下，算法跳出了往回画正方形的局面，而是直接从边缘出发包围了整个场地，从而获得胜利。

这种情况是我们之前并没有预料到的，但是却也给我们提供了一种思路：我们可以选择以更稳健的办法包围整个场地，这样可能会比我们现在的策略获得更好的效果。于是我们写出了一个包围场地的策略，但是，经过测试我们发现，这种策略很难保证自己纸带的安全。而如果一味地躲避对方的纸带，那么圈地的目标就很难实现。因此我们最终放弃了这个想法，但是这次尝试让我们发现了原来算法的一些问题，同时包围场地的算法躲避判定上对于原算法有了一些改进。

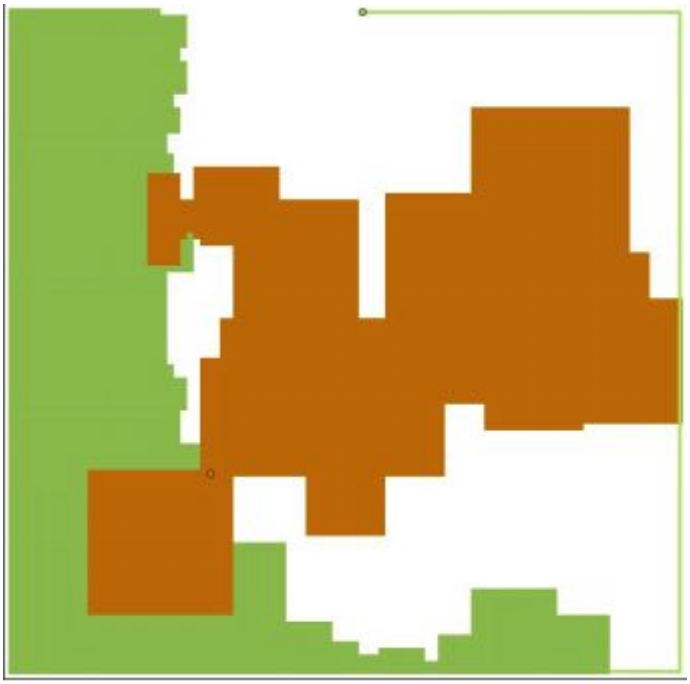


图 4. 战局中的一例

另一个战局我们想展示一个我们发现问题的过程。

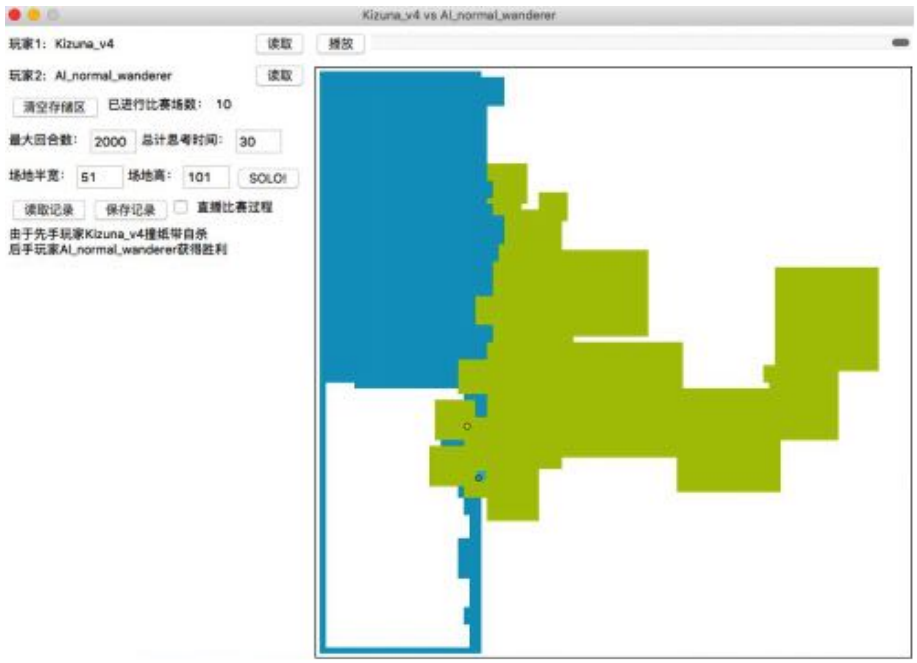


图 4. 战局中的一例

我们在对局中出现了这样一种情况：由于我们对于对方的策略并没有进行判定，因此当我们只顾着自己圈地的时候，如果对方将我们原来圈的范围切断了的话，我们自己画正方形的策略就会直接自杀死亡。对于这种情况我们并没有想到更好的办法，应对的方法是在判定对方距离时改成一个更加谨慎的条件，这在一定程度上减少了这种被切断情况的发生。

4 实习过程总结

4.1 分工与合作

分工:

孙鎏磊、文飞黄同学负责策略部分；

苏文霖、王颉同学负责算法实现部分；

周宏伟同学负责实习报告的撰写部分。

MAY 28 二教 514 9：30-10：30

进行了第一次组会，重点在于从一脸懵逼的样子里熟悉赛制和游戏规则，对小组成员进行了分工，简单地进行了策略讨论，在讨论中对游戏规则更熟悉，但是并没有得出成形的策略，要求策略组同学在第二天晚上研究出可行策略。

MAY 30 四教 405 9：30-10：30

继第一次组会，我们又进行了第二次组会，这次组会主要进行了 AI 策略汇报。经过策略组同学的思索，提出一种安全保守的策略，即在保证安全的距离内完成圈地，对方纸袋头入侵时，采取规避的策略，沿圈出的地的边缘逃离，在逃离的过程中完成小规模圈地或补缺动作，以螺旋渐进的方式完成圈地面积的扩张。

JUNE 6 四教 511 9：30-10：30

第三次组会主要是对第二次组会提出的策略进行探讨，代码组对算法实现提出了多个问题，策略组同学对策略进行了微调。

线上组会不计其数……

4.2 经验与教训

算法并非越复杂就越好，而是思路越明晰，算法越明确简洁才好。解决问题的关键在于对问题理解的深度，并不在于技术手段，优化问题的解决方法才依赖于技术手段。

原来听起来十分高大上的 AI 就可以由一堆判断语句制造出来，原来 AI 也不过如此，这说明有些概念听起来玄乎又玄，实际上其本质非常简单，应该从本质层面去认识事情，不要停留于表面和概念。

4.3 建议与设想

建议明年可以更早地确定下来大作业题目，多进行热身赛，阶段性地强制进行热身赛，鼓励各个小组多次参与热身赛，多次优化 AI 程序，使得最后的竞赛可以更加激烈。

5 致谢

感谢陈斌老师这一学期勤勉的教学、精彩的讲解和不输老司机的水群。

感谢冀瑞、陈旭、滕沅建、赵宇、缪舜、张颢丹、易超等助教的耐心指导和帮助。

感谢技术组大佬实力 CARRY，为大家搭建了简洁高效的对战平台。

感谢各位组员的热心参与。

第四章　N17_Delta 报告

段明阳、李文秀、汤玮辰、尹超 *、赵青

摘要：圈地、击杀、返回领地、领地内路径规划，四个主要部分共同组成了我们小组的纸带圈地算法，定义了 move_in_fields、enemy_returndistance、killenemy、cluster 等类用于实现领地内路径规划、最大面积圈地、最短路径返回领地、生成击杀路径等不同功能，turn_point 函数则确保了生成路径的安全性。经过上千次的测试后，撞上自己纸带和撞墙的 bug 已经成功避免，圈地效率与其他个小组相比也较高，能够有效判断必杀条件并生成击杀路径完成击杀。在与其他 AI 对战时，后手占优。

关键字：最大圈地面积、最短路径规划、路径安全性保障

1 算法思想

1.1 总体思路

整体算法由圈地算法、击杀算法、领地内行走、发现危险时最短路径返回领地，四个部分组成。

圈地算法限制（除第一次在场地中央圈出最大正方形外）单次圈地所走的路径最多有两个拐点，通过坐标变换使己方纸带头当前的行走方向为东，分为左拐一次、右拐一次、左拐两次、右拐两次，四种情况进行路径探测。分别计算四种情况下的所有可能路径的圈地面积与行走步数之比，选取比值最大的路径完成圈地。

击杀算法首先判断是否满足必杀条件，若满足则执行击杀动作。必杀条件判断过程如下：遍历敌方纸带中的每一点，判断自己到该点的距离是否大于敌方回到自己领地的距离，

如果是，则该点不满足必杀条件，计算下一个点；如果不是，则将该点设为目标点生成两条到达该点的不同路径，遍历两条路径中的所有点，若其上的点都满足在自己领地内或者敌方到该点的反杀距离大于我方所需的击杀距离，则满足必杀条件按之前所生成的路径完成击杀。另外，当不处于自己领地且敌我距离小于 2，或者处于自己领地内且敌我距离小于 5 时，都会搜索双方所有可能的行走方向，判断是否能够完成击杀。

当我方处于自己的领地内时，则按各个行走方向的不同，分别讨论出自己领地的最短路径，同时保证不会撞墙。

1.2 算法流程图

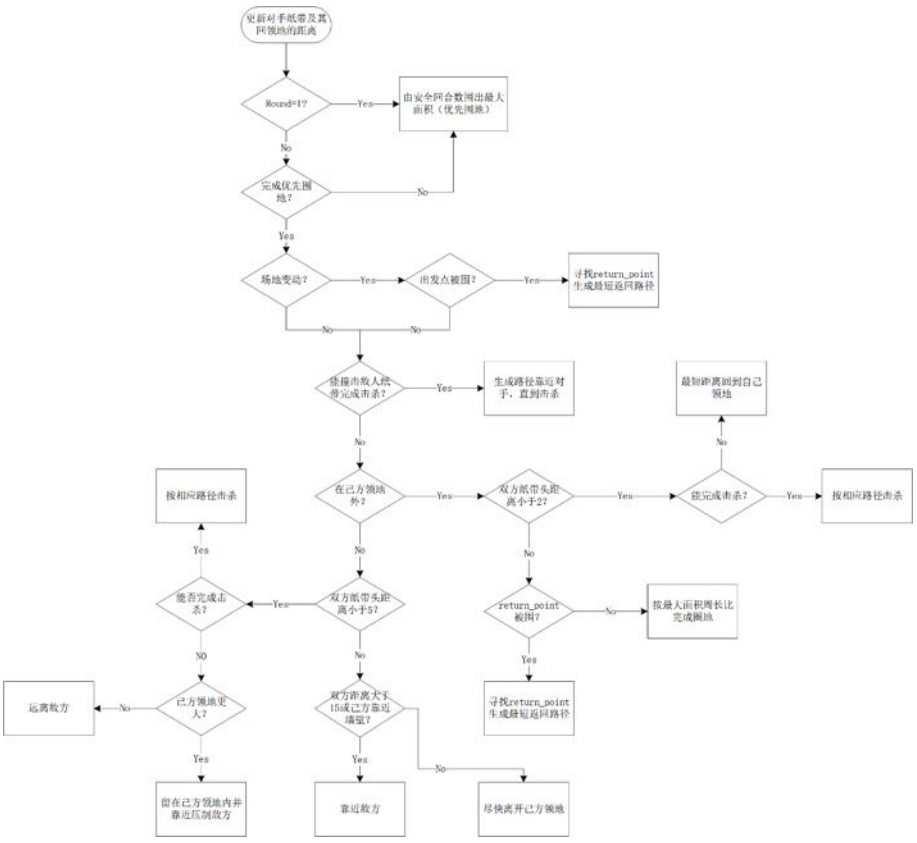


图 1：总体算法流程图

1.3 算法运行时间复杂度分析

除了开场围地之外每一局都使用 cankill 函数判断是否必杀，在寻找安全的最短距离击杀点时，会遍历地方纸带上的每一个点。生成两条击杀路径并比较击杀距离与反击杀距离，决定是否向击杀方向前进。如果没有必杀策略但己方领地面积占优则苟在己方领地，并尽量靠近对方纸卷，我们使用了 gouing 函数判断是否应继续待在自己的领地内下去，同时记录 gouing 函数返回 True 的次数称为 gouround，countarea 函数遍历了场地所有点，比较双方领地的面积大小，时间复杂度为 O(width*height)，由于遍历的点较多为了平衡时间消耗和决策正确性之间的矛盾，gouround=10 时调用一次 countarea。与敌方距离很近时，根据根据赛场局势的不同（出发点是否被围），时间主要消耗在用 go_out 函数和 min_return 函数生成最优的圈地路径和回自己领地的最短路径上。而当双方距离极近时，虽然遍历所有可能路径，但时此时的路径可能性小于某一常数，所以时间复杂度为 O(1)。

cluster 类中使用字典作为存储容器，并分别存放己方领地中每个 x 值对应所有 y 值，以及每个 y 值对应所有 x 值以及 x、y 边界。这降低了判断纸卷位置是否在边角，以及搜索、更新领地的时间复杂度。并且由于纸带的连续性，领地在相当的程度上也是连续的，故定义了两个不同函数 update 和 update_all 分别用于更新相连的领地和全域的领地信息。也减少了算法的时间消耗。

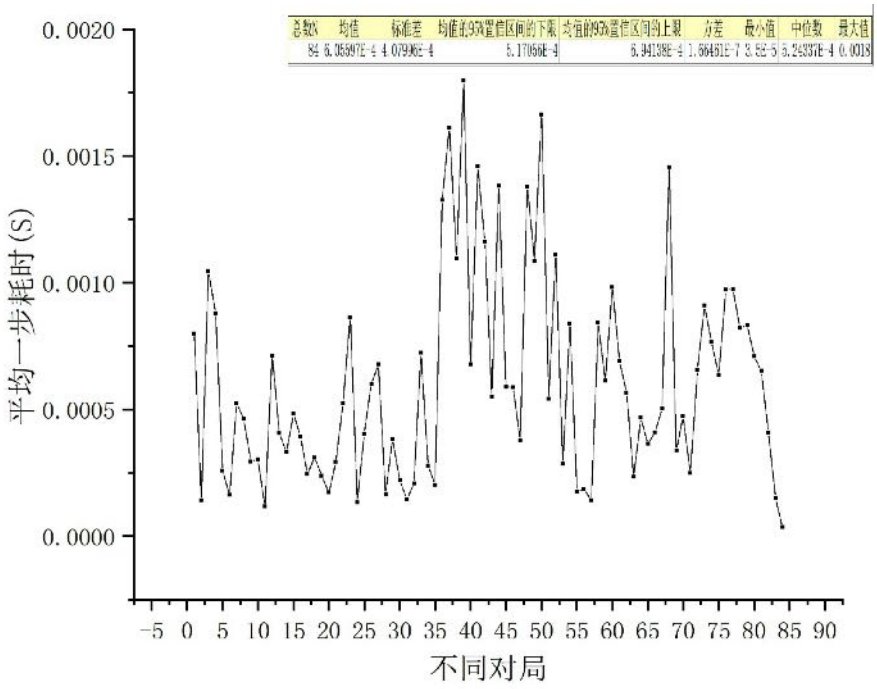


图 2：参赛代码在 6 月 12 日的对战中平均一步耗时

由于最终版本的代码只参与了 6 月 12 日的比赛和 6 月 15 日的友谊赛，且友谊赛的复盘数据并未公布，所以可以用于统计的数据较少，对于可以取得复盘数据的全部 84 场比赛进行统计。在所有比赛中我方代码耗时均控制在 2 秒以内，运行速度完全满足比赛规则的要求，对例如更新场地信息等可能需要进行大量数据点遍历的算法的时间复杂度完成了有效的控制。但是对战不同对手时的平均每步耗时仍有约 7 倍的差异，可见对战不同的 AI 代码时，所调用函数的频率分布有很大的不同，而不同函数的时间复杂度从 O(width*height) 到 O(1) 不等，故而时间消耗不同。对于统计的所有比赛而言，其平均值、中位数、标准差在同一数量级，平均时间消耗的振荡并未有远远超过平均值的情况发生。

2 程序代码说明

2.1 数据结构说明

定义如下类：move_in_fields，其中包含函数 resetfield 用于更新场地信息，infields 返回当前位置是否在自己领地内，setin_fields 用于更新 infields 的判断，以及三种走出己方领地的方式 rightout、leftout 和 straightout。

类 enemy_returndistance，用于计算敌方返回自己领地的距离。

类 killenemy，中有 kill 函数判断是否处于必杀状态；killpoint 函数用于寻找击杀点；cankill 函数判断能否达到必杀条件，若能则向该条件运行。类 gou 中包含了函数 countarea，其作用是当在己方领地内与敌人对峙多局时，计算双方面积大小，若自己占优则为了保证安全需待在自己的领地，否则远离敌人寻找其他出路。

类 cluster 中的函数 update 按广度优先搜索将相邻区域中属于自己领地的点加入 cluster，update_all 则没有联通的要求，将自己领地内横纵坐标的最大值和最小值分别存入 x_border，y_border 和 border。

```
def add(self,point): # 添加1个点到cluster
    [point_x, point_y] = point
    if point_x not in self.x:
        self.x[point_x] = [point_y]
    else:
        self.x[point_x].append(point_y)
    if point_y not in self.y:
        self.y[point_y] = [point_x]
    else:
        self.y[point_y].append(point_x)
def update(self,fields,start):
    self.x = {}
    self.y = {}
    self.amount = 0

    # 广度优先搜索生成相连的区域
    self.add(start)
    newlist = [start]
    while newlist != []:
        point = newlist.pop()
        self.amount += 1
        [point_x, point_y] = point

        # 把其相邻的加入cluster
        if point_x<self.width-1:
            if fields[point_x+1][point_y] == self.id and point_x+1 not in self.y[point_y]:
                newlist.append([point_x+1,point_y])
                self.add([point_x+1, point_y])

        if point_x>0:
            if fields[point_x-1][point_y] == self.id and point_x-1 not in self.y[point_y]:
                newlist.append([point_x-1,point_y])
                self.add([point_x - 1, point_y])

        if point_y<self.height-1:
            if fields[point_x][point_y+1] == self.id and point_y+1 not in self.x[point_x]:
                newlist.append([point_x,point_y+1])
                self.add([point_x , point_y+1])

        if point_y>0:
            if fields[point_x][point_y-1] == self.id and point_y-1 not in self.x[point_x]:
                newlist.append([point_x,point_y-1])
                self.add([point_x, point_y-1])
def update_all(self,fields): # 将自己的所有地盘，而不是连通的，存入self.x,self.y。用来找回家路径
    self.x = {}
    self.y = {}
    for x_co in range(self.width):
        for y_co in range(self.height):
            if fields[x_co][y_co] == self.id:
                if x_co in self.x:
                    self.x[x_co].append(y_co)
                else:
                    self.x[x_co] = [y_co]
                if y_co in self.y:
                    self.y[y_co].append(x_co)
                else:
                    self.y[y_co] = [x_co]
    self.x_border = {}
    self.y_border = {}
    for x_co in self.x:
        self.x_border[x_co] = [min(self.x[x_co]),
                               max(self.x[x_co])]
    for y_co in self.y:
        self.y_border[y_co] = [min(self.y[y_co]),
                               max(self.y[y_co])]
    self.border= {'xmin': min(self.x), 'xmax': max(self.x), \
                  'ymin': min(self.y), 'ymax': max(self.y)}
```

图 3：update 和 update_all 函数的具体实现

```
# 由self.x,self.y生成边界信息
self.x_border = {}
self.y_border = {}
for x_co in self.x:
    self.x_border[x_co] = [min(self.x[x_co]),
                           max(self.x[x_co])]
for y_co in self.y:
    self.y_border[y_co] = [min(self.y[y_co]),
                           max(self.y[y_co])]
self.border = {'xmin': min(self.x_border), 'xmax': max(self.x_border),
               'ymin': min(self.y_border), 'ymax': max(self.y_border)}
```

图 4：cluster 类中实现记录边界信息

2.2 函数说明

选取代码中功能最重要的如下四个函数进行说明，killenemy 类中判断必杀条件的 cankill 函数、完成圈地规划的 go_out 函数、发现己方被围时生成回到领地最快路径的 min_return 函数、保证盲走（不探测）情况下纸带安全的 turn_point 函数。

cankill 函数所需参数如下：stat、enemy_bands（敌方纸带）、enemyreturndistance（敌方返回自己领地的距离），其中需要从 stat 中获取以下信息：width, height = stat['size']、me = stat['now']['me']、enemy = stat['now']['enemy']、fields = stat['now']['fields']

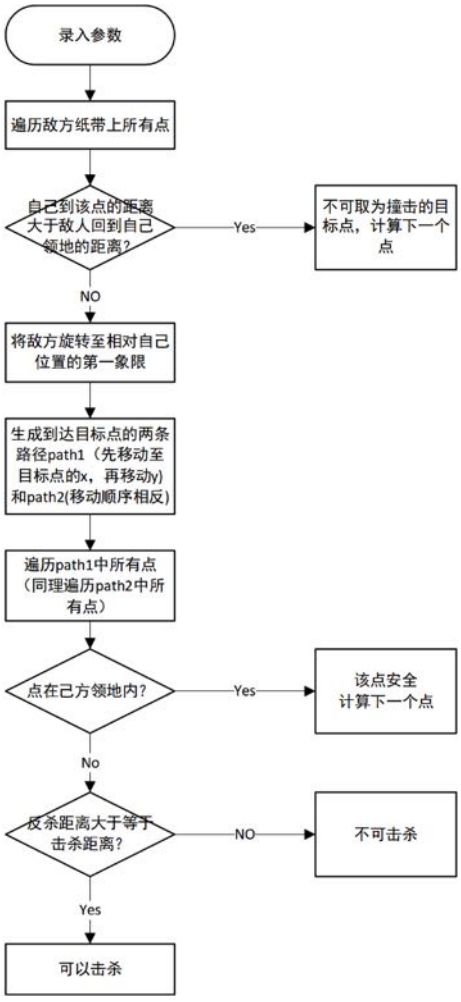


图 5：cankill 函数流程图

```
#将x,y绕x_0, y_0 逆时针旋转90*angle度，方便之后的讨论
def rotating(self, x, y, angle):
    if angle % 4 == 0:
        return x,y
    elif angle % 4 == 1:
        return y, - x
    elif angle % 4 == 2:
        return  - x,  - y
    elif angle % 4 == 3:
        return  - y, x
angle = 0
target_x, target_y = point_x - me_x, point_y - me_y
#通过旋转，将敌人位置转到相对自己位置的第一象限
if target_x >= 0 and target_y >= 0:
    angle += 0
elif target_x <= 0 and target_y >= 0:
    angle += 1
elif target_x <= 0 and target_y <= 0:
    angle += 2
elif target_x >= 0 and target_y <= 0:
    angle += 3
#旋转称新的位置和新的方向
newtarget_x, newtarget_y = self.rotating(target_x, target_y, angle)
newdirection = (direction - angle) % 4
```

图 6：rotating 函数将敌方旋转至我方第一象限的具体代码

```
#只考虑两条击杀路径
path1 = []
path2 = []
for i in range(newtarget_x + 1):
    path1.append(self.rotating(i, 0, - angle))
for i in range(newtarget_y):
    path1.append(self.rotating(newtarget_x, i + 1, - angle))
    path2.append(self.rotating(0, i , - angle))
for i in range(newtarget_x + 1):
    path2.append(self.rotating(i, newtarget_y, - angle))
```

图 7：path1 和 path2 两条路径的具体生成方式

在这段代码中，通过将敌方旋转至相对于我方纸带头的第一象限内，简化了由于双方相对位置不同带来的不同情况的处理过程，极大的提高了代码的凝练程度和可读性，便于功能的具体实现。通过坐标变化简化代码，这一思路将在后续的代码实现过程中起到重要的作用，并被反复运用。

go_out 函数所需参数如下：direction(我方当前行走方向)、me_field_x (己方领地所有横坐标的值) 、me_field_y （己方领地所有纵坐标的值）、me_field_x_border （己方领地横坐标上下限）、me_field_y_border（己方领地纵坐标上下限）、me_field_border（己方领地横纵坐标上下限）、enemy_x（敌方纸带头横坐标）、enemy_y（敌方纸带头纵坐标）、me_x（我方纸带头横坐标）、me_y（我方纸带头纵坐标）、width（场地宽度）、

height（场地高度）、turnleft（剩余回合数）、brave=1（勇敢因子，其值越高表现越激进，圈地效率越高，但有被击杀的风险，在竞赛过程中其值设定为 1）。

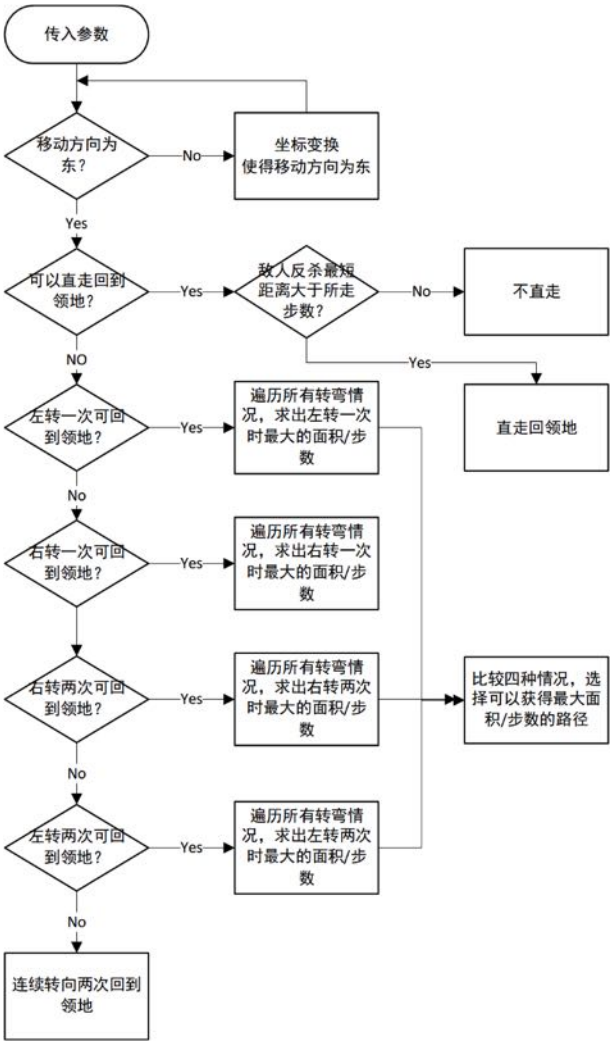


图 8：go_out 函数流程图

实际代码中依照左拐一次、右拐一次、右拐两次、左拐两次的顺序计算面积/步数的值，并进行比较，若新的比值较大则替换之前最大比值下找到的 max_y_return, max_x_return, max_x_turn。为了流程图的简介绘图时稍作修改，并未完全依照程序的实际执行方式逐步比较。

```
elif direction == 1: # x,y互换变为direction==0。行动中的左右转也互换
    new_border = {
        'xmin': me_field_border['ymin'], 'xmax': me_field_border['ymax'],
        'ymin': me_field_border['xmin'], 'ymax': me_field_border['xmax']
    }
    sleep, action, [x_return, y_return] = go_out(0, me_field_y,
            me_field_x, me_field_y_border, me_field_x_border,
            new_border, enemy_y, enemy_x, me_y, me_x, height, width,turnleft, brave)
    for index in range(len(sleep)):
        if sleep[index] == 'L':
            sleep[index] = 'R'
        elif sleep[index] == 'R':
            sleep[index] = 'L'
    if action == 'L':
        action = 'R'
    elif action == 'R':
        action = 'L'
    return sleep, action, [y_return, x_return]

elif direction == 2: # 坐标变换x'=-x,y'=-y 变为direction==0
    neg_border = {
        'xmin':-me_field_border['xmax'],'xmax':-me_field_border['xmin'],
        'ymin': -me_field_border['ymax'], 'ymax': -me_field_border['ymin']
    }
    sleep, action, [x_return, y_return] = go_out(0, neg_field(me_field_x),
            neg_field(me_field_y), neg_field(me_field_x_border,True), neg_field(me_field_y_border,True),
            neg_border, -enemy_x, -enemy_y, -me_x, -me_y,  -width, -height, turnleft,brave)
    return sleep, action, [-x_return, -y_return]

elif direction == 3: # 坐标变换x'=-x,y'=-y 变为direction==1
    neg_border = {
        'xmin':-me_field_border['xmax'],'xmax':-me_field_border['xmin'],
        'ymin': -me_field_border['ymax'], 'ymax': -me_field_border['ymin']
    }
    sleep, action, [x_return, y_return]= go_out(1, neg_field(me_field_x),
            neg_field(me_field_y), neg_field(me_field_x_border,True), neg_field(me_field_y_border,True),
            neg_border, -enemy_x, -enemy_y, -me_x, -me_y,  -width, -height, turnleft,brave)
    return sleep, action, [-x_return, -y_return]
```

图 9：坐标变换的具体实现

```
# 如果没找到一条路径（实际中几乎不出现），认为连拐2次能回家
if max_x_return == None:
    if me_y+1 in me_field_x[me_x-1]:
        return [0,'R',0], 'R', [me_x-1, me_y+1]
    else:
        return [0,'L',0], 'L', [me_x-1, me_y-1]

if max_x_return == max_x_turn: # 最佳路径为拐1次
    sleep = [max_x_turn - me_x - 1, 'R', abs(max_y_return - me_y) - 1]
    if max_y_return - me_y < 0:
        sleep[1]= 'L'
    if max_x_turn == me_x:
        return sleep[2:], sleep[1], [max_x_return, max_y_return]
    return sleep, 'N', [max_x_return, max_y_return]

# 最佳路径为拐2次
sleep = [max_x_turn - me_x - 1, 'R', abs(max_y_return - me_y) - 1, 'R',
                    max_x_turn - max_x_return - 1]
if max_y_return - me_y < 0:
    sleep[1], sleep[3] = 'L', 'L'
if max_x_turn == me_x:
    return sleep[2:], sleep[1], [max_x_return, max_y_return]
return sleep,'N', [max_x_return, max_y_return]
```

图 10：选择最佳圈地路径

四种路径均不可取则连续拐弯两次回到自己领地再次进行搜索，若找到了合适的路径，判断选取哪种情况的具体代码如图 9 所示。

min_return 函数所需参数如下：direction（己方当前行走方向）、me_id（己方表示 1 或 2）、me_x（己方纸带头横坐标）、me_y（己方纸带头纵坐标）、fields（场地信息）、

bands（双方纸带信息）、width（场地宽度）、height（场地高度），其中 width 和 height 均用于调用 cluster 类中的函数获取己方的纸带和领地信息。

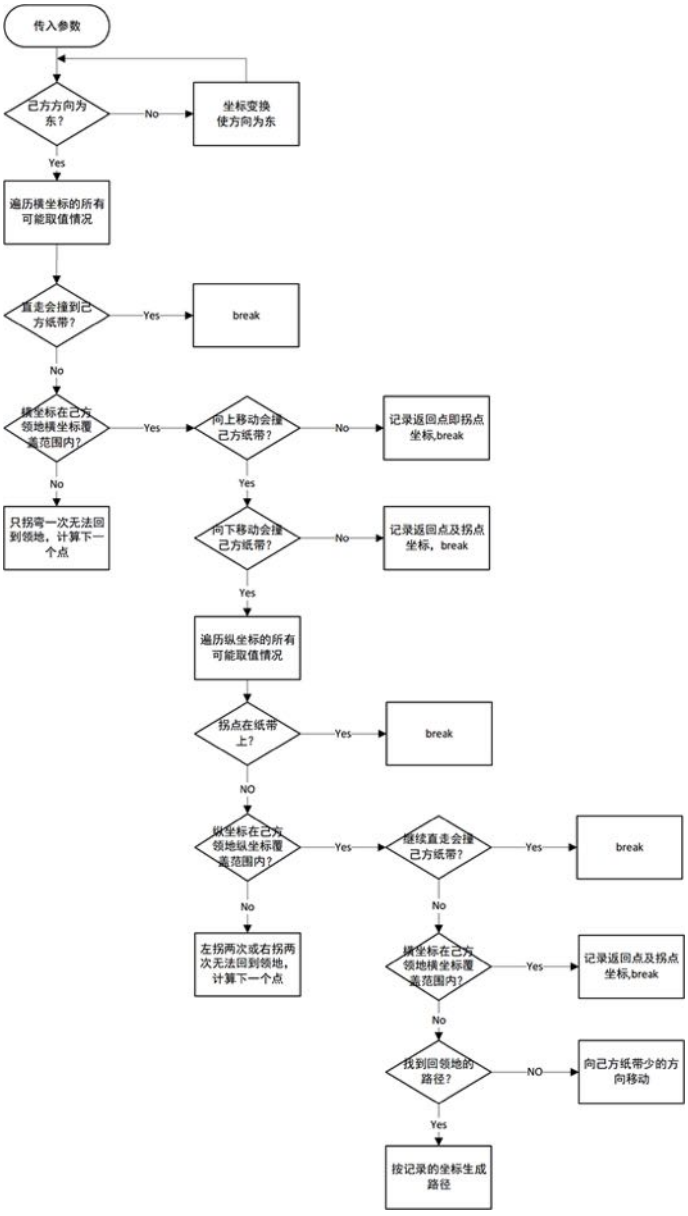


图 11：min_return 函数流程图

```
# 按me的direction分情况，direction != 0时用坐标变换变到direction = 0
if direction == 0:
    return min_return_helper(me_x, me_y,me_field.x,me_field.y,me_band.x,me_band.y, me_band.border, width, height)
elif direction == 1:
    new_border = {
        'xmin': me_band.border['ymin'], 'xmax': me_band.border['ymax'],
        'ymin': me_band.border['xmin'], 'ymax': me_band.border['xmax']
    }
    sleep, action, return_point = min_return_helper(me_y,
                        me_x,me_field.y,me_field.x, me_band.y, me_band.x, new_border, height,width)

    for index in range(len(sleep)):
        if sleep[index] == 'L':
            sleep[index] = 'R'
        elif sleep[index] == 'R':
            sleep[index] = 'L'
    if action == 'L':
        action = 'R'
    elif action == 'R':
        action = 'L'
    if return_point == None:
        return sleep, action, None
    [x_return, y_return] = return_point
    return sleep, action, [y_return, x_return]

elif direction == 2:
    neg_border = {
        'xmin':-me_band.border['xmax'],'xmax':-me_band.border['xmin'],
        'ymin': -me_band.border['ymax'], 'ymax': -me_band.border['ymin']
    }
    sleep, action, return_point = min_return_helper(-me_x, -me_y, neg_field(me_field.x),
                        neg_field(me_field.y),neg_field(me_band.x),
                        neg_field(me_band.y), neg_border, -width, -height)
    if return_point == None:
        return sleep, action, None
    [x_return, y_return] = return_point
    return sleep, action, [-x_return, -y_return]

elif direction == 3:
    neg_border = {
        'xmin':-me_band.border['ymax'],'xmax':-me_band.border['ymin'],
        'ymin': -me_band.border['xmax'], 'ymax': -me_band.border['xmin']
    }
    sleep, action, return_point = min_return_helper(-me_y, -me_x, neg_field(me_field.y),
                        neg_field(me_field.x),neg_field(me_band.y),
                        neg_field(me_band.x), neg_border, -height, -width)
    for index in range(len(sleep)):
        if sleep[index] == 'L':
            sleep[index] = 'R'
        elif sleep[index] == 'R':
            sleep[index] = 'L'
    if action == 'L':
        action = 'R'
    elif action == 'R':
        action = 'L'
    if return_point == None:
        return sleep, action, None
    [x_return, y_return] = return_point
    return sleep, action, [-y_return, -x_return]
```

图 12：坐标变换的具体实现

```
# 先直走，然后拐一次，从立刻拐开始，增加之前直走的步数搜索
x_turn = me_x
while x_turn <= min(x_max, me_x+min_len):
    x_turn_in_band = x_turn in me_band_x
    if x_turn_in_band and me_y in me_band_x[x_turn] and x_turn != me_x: # 直走到此撞纸带，结束循环
        break
    if x_turn not in me_field_x: # 在这里拐回不去
        x_turn += 1
        continue
    no_band = [True, True] # 左拐和右拐是否撞到纸带
    for dy in range(1,min_len + me_x - x_turn): # 拐之后直走的步数，从0开始递增搜索
        y_return = me_y + dy
        if x_turn_in_band and y_return in me_band_x[x_turn]:
            no_band[0] = False
        if no_band[0] and y_return in me_field_x[x_turn]:
            min_len = dy + x_turn - me_x
            min_x_turn, min_y_return = x_turn, y_return
            break
        y_return = me_y - dy
        if x_turn_in_band and y_return in me_band_x[x_turn]:
            no_band[1] = False
        if no_band[1] and y_return in me_field_x[x_turn]:
            min_len = dy + x_turn - me_x
            min_x_turn, min_y_return = x_turn, y_return
            break
    x_turn += 1
# 左拐2次，即在该处立刻右拐，直走一段后再左拐，再直走回家
y_turn = me_y-1
while y_turn >= max(y_min, me_y - min_len):
    if y_turn in me_band_x[me_x]:
        break
    if y_turn not in me_field_y:
        y_turn -= 1
        continue
    y_turn_in_band = y_turn in me_band_y
    for x_return in range(me_x-1, max( me_x+me_y-y_turn-min_len, x_min),-1):
        if y_turn_in_band and x_return in me_band_y[y_turn]:
            break
        if x_return in me_field_y[y_turn]:
            min_len = me_x - x_return + me_y - y_turn
            min_x_turn, min_x_return, min_y_return = me_x, x_return, y_turn
            break
    y_turn -= 1

# 右拐2次，即在该处立刻右拐，直走一段后再右拐，再直走回家
y_turn = me_y + 1
while y_turn <= min(y_max, me_y + min_len):
    if y_turn in me_band_x[me_x]:
        break
    if y_turn not in me_field_y:
        y_turn += 1
        continue
    y_turn_in_band = y_turn in me_band_y
    for x_return in range(me_x-1, max( me_x + y_turn - me_y - min_len, x_min), -1):
        if y_turn_in_band and x_return in me_band_y[y_turn]:
            break
        if x_return in me_field_y[y_turn]:
            min_len = me_x - x_return - me_y +y_turn
            min_x_turn, min_x_return, min_y_return = me_x, x_return, y_turn
            break
    y_turn += 1
```

图 13：拐弯一次和拐弯两次的途径探测过程

turn_point 函数所需参数如下：direction（己方当前行走方向），enemy_x（敌方纸带头横坐标），enemy_y（敌方纸带头纵坐标），me_x（我方纸带头横坐标），me_y（我方纸带头纵坐标），x_return（目标点横坐标），y_return（目标点纵坐标），brave=1

```
# 返回点，brave-1不死，没考虑边界
def turn_point(direction, enemy_x, enemy_y , me_x, me_y, x_return, y_return, brave=1):
    if abs(enemy_x - me_x) + abs(enemy_y - me_y) < abs(x_return - me_x) + abs(y_return - me_y):
        return None
    if direction == 0:
        if y_return <= me_y and x_return <= me_x: # return点在第三象限
            if enemy_x >= me_x and enemy_y > me_y: # enemy在第一象限，（以me为原点）
                turn = floor(( brave*(enemy_y-me_y) + me_x + x_return - abs(me_y-y_return) )/2)
                if turn < enemy_x:
                    turn = floor((brave*(enemy_x+enemy_y-me_y) + me_x + x_return -
                                  abs(me_y - y_return))/(2+brave))
            elif enemy_x > me_x and enemy_y <= me_y: # enemy在第二象限
                if enemy_y >= y_return:
                    turn = floor(( brave*enemy_x + me_x + x_return - abs(me_y-y_return) )/(2+brave))
                else:
                    turn = floor((brave * (-enemy_y + y_return) + me_x + x_return - abs(me_y - y_return)) /2)
                    if turn < enemy_x:
                        turn = floor((brave * (-enemy_y + y_return + enemy_x) + me_x + x_return
                                      - abs(me_y - y_return))/(2+brave))
            elif enemy_x <= me_x and enemy_y < me_y: # enemy在第三象限
                if enemy_x >= x_return:
                    turn = floor((brave * (-enemy_y + y_return) + me_x + x_return - abs(me_y - y_return)) / 2)
                else:
                    # 三种可能路径
                    len_ = min(abs(enemy_y-y_return)+abs(enemy_x-x_return),abs(me_y-enemy_y) + abs(me_x - enemy_x))
                    turn = floor((brave * len_ + me_x + x_return - abs(me_y - y_return)) / 2)
                    if turn - enemy_x < len_:
                        turn = floor( (-enemy_x*brave+me_x+x_return-abs(me_y - y_return))/(2-brave) )
            else: # enemy在第四象限
                enemy_return = enemy_y-y_return if enemy_x>=x_return else enemy_y-y_return + x_return-enemy_x
                enemy_re = enemy_y-me_y + me_x-enemy_x
                turn = floor((brave * min(enemy_me,enemy_return) + me_x + x_return - abs(me_y - y_return)) / 2)
                if y_return == me_y and enemy_x > x_return:
                    turn = enemy_y
                return turn if turn >= max(me_x,x_return) else None

        elif y_return <= me_y and x_return > me_x: # return点在第二象限，坐标变换x'=x-x_return, y'=y_return-y
            turn= turn_point(0, enemy_x-x_return, y_return-enemy_y,0,0, me_x-x_return, y_return-me_y,brave)
            return x_return + turn if turn != None else None
        elif y_return > me_y and x_return < me_x:  # return点在第四象限，坐标变换x'=x-me_x, y'=me_y-y
            turn = turn_point(0, enemy_x-me_x, me_y-enemy_y, 0,0,x_return-me_x, me_y-y_return,brave)
            return me_x + turn if turn != None else None
        else: # return点在第一象限，坐标变换x'=x, y'=y
            return turn_point(0, enemy_x,enemy_y,x_return, y_return, me_x, me_y, brave)

    elif direction == 1:
        return turn_point(0, enemy_y, enemy_x, me_y, me_x, y_return, x_return, brave)
    elif direction == 2: # x'=-x
        turn = turn_point(0, -enemy_x, enemy_y, -me_x, me_y, -x_return, y_return, brave)
        return -turn if turn != None else None
    else:
        turn = turn_point(0, -enemy_y, enemy_x, -me_y, me_x, -y_return, x_return, brave)
        return -turn if turn != None else None

storage['turn_point'] = turn_point

storage['last_in_field'] = [me_x, me_y]

storage['last_me_in_field'] = False
storage['last_enemy_in_field'] = True
```

图 14：确定路径中拐点的 turn_point 函数

2.3 程序限制

在经过多轮调试之后，基本杜绝了程序报错的发生。但是对于防止被击杀的的功能并不完善，比如我方被围的情况，只检测了已生成路径中的返回点和出发点是否被敌方占领，对于其他敌人蚕食我方领地的行为并没有相应的检测机制加以防范。另外只有在双方距离极近时探测了所有路径的击杀可能，此时胜负情况将会强烈的依赖于先前的局势，包括先后手、双方的领地和纸带的分布情况，应当在一定程度上增加搜索的距离，从而提高回旋的余地。go_out 函数和 min_return 函数探索的路径限制在拐弯最多两次的情况，相比所有可能的路径检测的范围还是太小，所以可能会找不到合适的路径，对后来的圈地造成不良影响或者给敌方提供击杀的机会。

3 实验结果

3.1 测试数据

实验环境说明：

硬件配置：（CPU/内存）i5—6300HQ 4G

操作系统：（名称/版本）Windows10

Python 版本：（版本号）3.62

测试采用人机对战时，通过人为制造满足必杀条件的情况诱导 AI 进行击杀动作，寻找能击杀但 AI 并不击杀的 bug；将 AI 逼近边界，测试 AI 能否规避撞墙。并在测试过程中发现程序报错和撞击自己纸带的 bug，并对这些情况进行改良。同时发现若代码侧重于攻击会在己方领地内增加过多不必要的步数，应在圈地与击杀之间进一步权衡。

采用相同版本的 AI 代码对战，可以发现其依赖于初始条件所产生的不同，并且发现了错失击杀机会和撞墙的 bug。同时我们意识到需要加入考量剩余回合数的问题，以防止操作完成之前，回合数耗尽判定结果。

采用不同版本的 AI 对战，便于比较不同代码之间的差异，包括圈地方式和击杀判断的改进，以及算法的时间消耗。同时，我们发现，在规划路径时简单的放弃有自己纸带的方向，不失为一种有效的策略，既可以简化算法，也可以避免发生围住自己纸带的情况发生，但是在一些特殊情况下可能会丧失制胜的机会。我们一共编写了 0.1，0.5，0.7，0.8，0.9，0.97，这几个版本的代码，分别对应三次热身赛中提交的 6 份代码，比赛结果如图 6 所示。

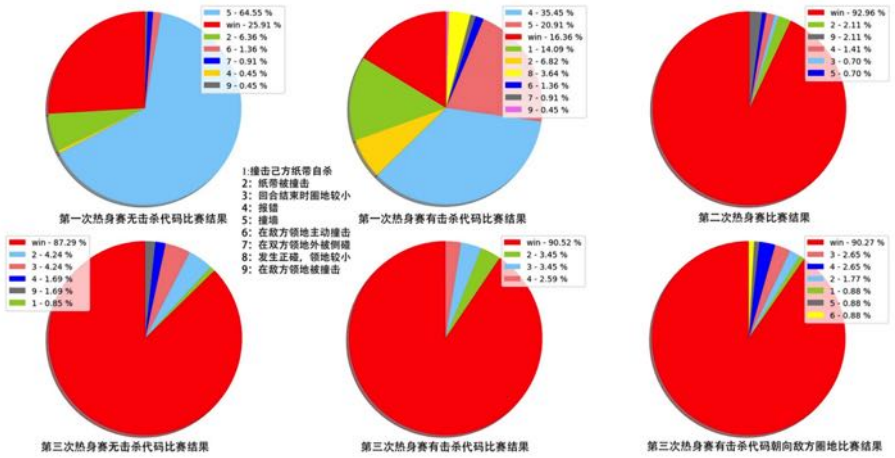


图 15：热身赛结果统计

3.2 结果分析

上图是对三次热身赛中我们小组进行的全部约 1500 场比赛的统计结果。在第一次热身赛中撞墙和击杀部分的代码中 bug 较多需要进行改进。第二次热身赛中代码仅包含改良后的圈地算法，胜率大大增加，可见取面积/步数最大的路径进行圈地是一种良好的策略需要保留，并尽可能改进。然而在对其中几盘复盘时我们发现，虽然我们的算法圈地效率很高，但假如对方一直在蚕食我方领地的话，最终面积大小反而可能会比敌方低。为此，我们考虑了将领地内的行走的算法在距离较大时改为向对方靠近，而不是找最短路径出领地（以牺牲圈地效率而增加攻击性）。在使用自己的代码测试时，攻击性较强的代码要明显由于原先的代码。由于我们不确定对战其他组的代码时攻击性较强的代码能否保持此优势，为了进行测试，我们在第三次热身赛的时候提交了三个版本的代码，分别是：delta_1—高效圈地；delta_2—高效圈地 + 击杀；delta_3—攻击性圈地 + 击杀。

第三次热身赛中，我们发现增加击杀代码后胜率有所增加，击杀代码需要保留。但同时在进行复盘时我们发现击杀部分代码存在一些 bug，包括报错和撞墙的情况复萌、将敌方出发点围住但并不击杀等，需要进行修复。在比较高效圈地和攻击性圈地代码时，我们发现 delta_2 和 delta_3 虽然胜率相近，但 delta_3 在和 delta_2 对战取得了较大优势。此外，delta_3 输的局中有相当部分是由于击杀部分代码的 bug 导致的（delta_3 更容易造成必杀情况），假如将此 bug 修复则可以进一步提升优势。最后，delta_3 的绝对圈地面积虽然比 delta_2 要低，但相对圈地面积反而比 delta_2 要高，即攻击性圈地可以虽然降低

了自己的圈地效率，但同时也能抑制对手的圈地。在第三次热身赛后，我们决定以 delta_3 为基础，进行修复和改进作为我们正式比赛的代码。

通过复盘我们还发现了一些其他需要规避的特殊 bug，例如：虽然避免了撞击自己的纸带，但是纸带围成了螺旋形状，实际上陷入了死地。这是由于每步探测之后规划的路径不同造成的，所以我们将算法改进为盲走 + 探测的模式。也就是在某些特殊点，例如：刚回到领地的点，刚出领地的点，依照先前规划需要拐弯的点，进行探测规划路径，之后除非满足必杀条件或者发现规划好的路径中的回归点被占等情况发生，就按该路径行走。这样既不会错过击杀的情况，也可以确保安全步数下的圈地效率，还可以减少多次探测带来的时间消耗。

3.3 经典战局

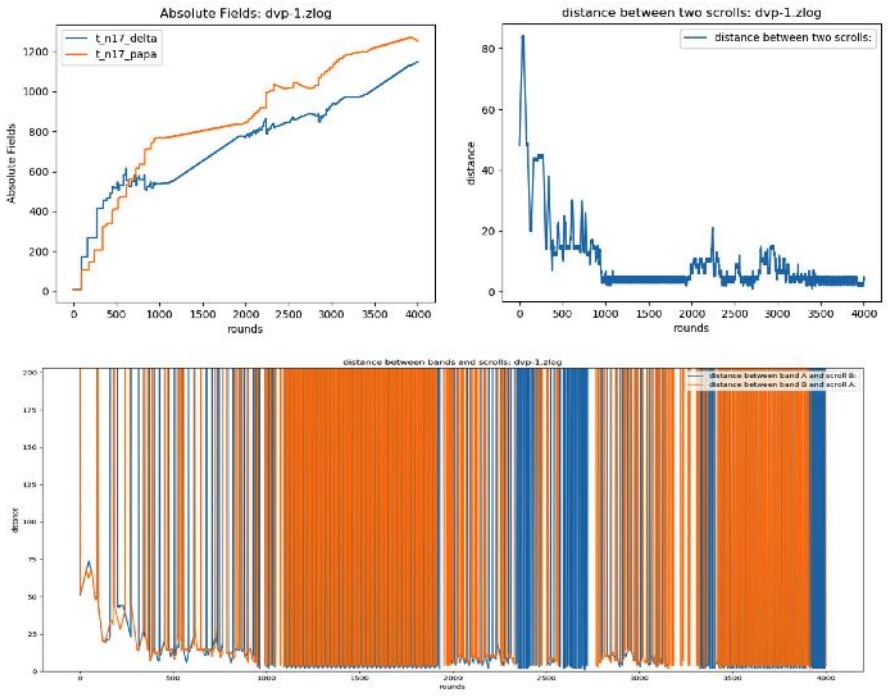


图 16：决赛 papa VS delta 第一场比赛分析

从复盘数据来看，在比赛前期双方还未进行交战时我方圈地效率更高，但是圈地面积在约 650 局时被敌方反超，这是因为我方靠近边界圈地效率受到限制，同时被敌方大量蚕食领地，说明我方的算法还可以增加一部分应对此种情况的代码。利用处于自身领地的优

势，即便无法击杀对方，至少可以逼迫对手离开自己的领地。在约 1000 局之后出现了激烈的缠斗情况，但双方均有效的躲避了对手的进攻并进行圈地，并未出现击杀的情况，虽然在缠斗过程中我方的圈地效率略占优势，但是由于前期领地大量流失，最终耗尽回合数，圈地面积略小于对手而落败。

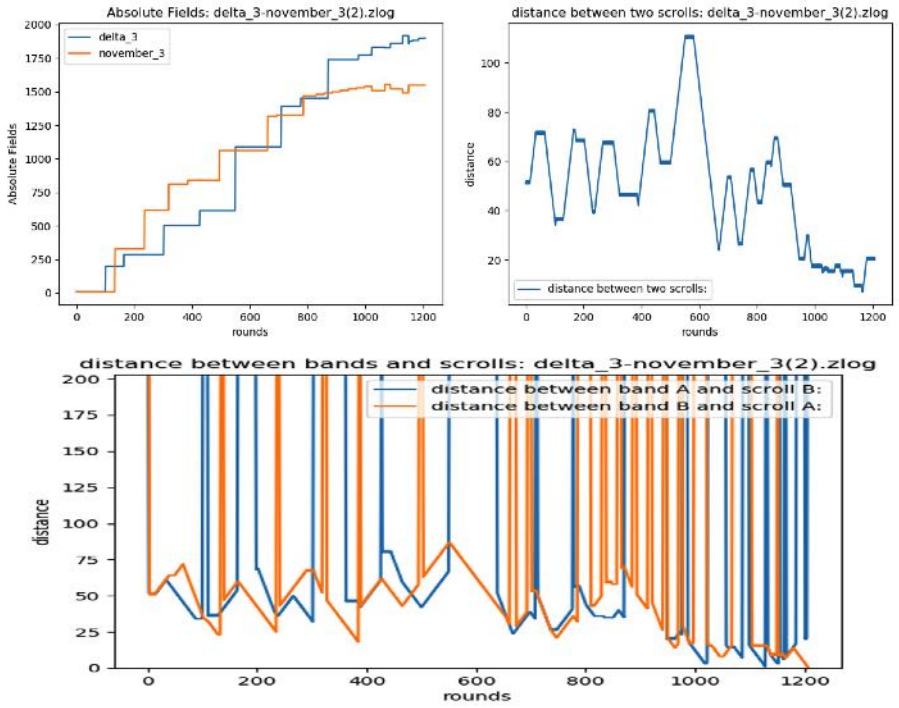


图 16：决赛 papa VS delta 第一场比赛分析

敌我双方的圈地效率不相上下，我方在圈地过程中不断靠近对方，这一策略在一定程度上限制了我方圈地的效率，但是提高了击杀对手的可能，在面对防御型策略的对手时比较有效。在约 550 局时我方利用双方距离较远，完成了一次大面积圈地，使得圈地面积反超对手。在约 1100 局时，对方进入并蚕食我方领地的过程中暴露了弱点给我方提供了击杀的可能，我方 AI 也抓住了这个机会，寻找到了合适的击杀路径，撞击敌方纸带取得胜利。可见撞击点在己方领地内即为安全的判定，发挥了良好的效果。

4 实习过程总结

4.1 分工与合作

尹超和汤玮辰同学负责了主要的代码编写工作，段明阳同学主要负责代码的测试工作，李文秀同学主要负责了热身赛复盘数据的统计工作，赵青同学主要负责报告的编写工作。大部分的算法和流程是在几次讨论过程中，大家集思广益所确定的。

交流方式包括在农园、理教、二教等地集合讨论；以及在微信群里沟通，上传已经完成的工作，并及时汇报进展。

第一次集合讨论时的照片，地点为农园咖啡厅，进行了初步的分工，确定了拐点最多为 2 个的路径规划方案。

第二次集合讨论的照片，地点在二教走廊，讨论决定了由探测结果确定盲走路径和步数，待完成盲走路径后再次进行探测，确定下一步的行动的算法，可以在兼顾时间消耗和行动的合理性。

第三次集合讨论没有照片，地点在理教走廊，确定了最终代码和部分报告内容。

4.2 经验与教训

我方代码的圈地效率高于大多数其他小组的算法，选择面积/步数最大的路径圈地及尽快走出自己的领地是明智的算法。在圈地和生成击杀路径的过程中通过限制拐弯次数小于等于 2，减少了可能路径的数量，从而缩小了算法的时间消耗；另外这也可以使得自己领地的形状较为规则。更新领地则有效地利用了圈地的规则和纸带的连续性，采用 cluster 类中的 update 方法即可解决多数领地更新问题，update_all 则用于全场地的领地更新，根据不同情况使用两种不同的函数，既可保证决策的正确性也可减小算法的开销。

对于路径的规划问题，解决方式仍然较为粗糙，舍弃计算较为复杂的情况，比如前进方向上有自己纸带等，但是其中可能含有合适的击杀路径或者回到自己领地的更短路径，这对我们进行进一步的优化产生了限制。我们小组由于运用了大量的启发性策略，导致整

局游戏用时平均在 2 秒左右，有大量的时间没有被使用。在时间充足时可以考虑使用时间复杂度更高但更有效的算法，比如 A 星算法，解决复杂情况下的路径规划问题。另外，还可以增加探测敌方纸带和领地分布情况地算法，考虑双方围地的一些局部以及整体结构，规避被围住的可能，并增加围住对方的几率。

在对正式比赛复盘时我们发现，假如采用攻击性更强的代码，可能可以提高对战 papa 队（冠亚军决赛对手）的胜率。而采用攻击性较弱但圈地更高效的算法，可能可以提高对战 hotel 队（半决赛对手）的胜率。该两组的策略实际上在第三次热身赛中有所展示，但出于保守起见，我们放弃了在冠亚军决赛和半决赛之前更换代码。假如我们在比赛时对对手的代码有一定的针对，或许可以进一步提高胜率。

4.3 建议与设想

对战平台前期频繁的调整，影响了代码的编写，希望能够直接开放较为完善的对战平台。另外对于复盘数据，希望能够增加按关键字搜索的功能，便于寻找所需地资料。这次实习作业的时间安排较为合适，不知能否够进一步细化报告和代码两部分的评分标准，用于指导代码和报告的编写工作。后续或许可以增加在线 AI 对战平台，每天开放几个小时给用户上传自己的代码，24 小时后可以获得与上传代码的全部用户的对战资料，这样就可以获得出热身赛外的其他数据用于改进自己的代码。

数据结构与算法这门课为我们提供了一个接触编程世界的优秀平台，还有更加广阔的空间有待我们的学习和探索，希望大家对于编程的学习不止于此。另外小组合作也是一个不同于大多数作业独立完成的机制，需要成员间通力合作，才能提高工作效率和质量。

祝愿选修这门课程的下期学生能够度过一段愉快的学习时光，并有所收获。

5 致谢

感谢组织这次实习作业的陈斌老师和翼锐、陈旭、滕沅建、赵宇、张颢丹、缪舜、易超几位助教。陈老师热心教学、知识渊博，给我们提供了良好的学习环境和内容，对这门课投入了很多地心血和精力，并对一些较为前沿地问题提供了简要的介绍和学习的资源，平时也给予了我们很多悉心的指导。各位助教也为这门课程投入了大量的时间，帮助我们更好的掌握这门课程，并且还对神经网络和遗传算法向我们进行了介绍，受益良多。

同时也要感谢编写对战平台的陈天翔、张赖和两位同学为这次实习大作业做出了巨大的贡献，几经改良后的对战平台已经十分的完善，对战平台的编写工作量远大于某一个小组的 AI 代码编辑工作，你们不吝自己的得失，高效地完成了基础代码的编写，选修为这门课程的所有同学提供了帮助。

也感谢各位组员的热心参与，使得我们小组顺利的完成了比赛并取得了优异的成绩。在本次前后约 1 个月的实习作业期间，大家分工合作，为本次作业做出了自己独特的贡献。交流沟通，及时反馈自己的工作进展，向组员寻求帮助也是重要的一环。

6 参考文献

北大人工智能实验室规划算法之 Alpha—Beta 剪枝.pptx

北大人工智能实验室规划算法之蒙特卡洛树搜索.pptx

第五章　N17_Echo 报告

向飞燕 * 张与之康峻侥尚昊成方鑫

摘要：本组在设计对战程序中，将任务分为三部分，即圈地（square）、思考（think）和进攻（attack）。其中 think 为程序的最重要部分，采用了 max-min 的比较策略，分析了本方和对方依次要选择的最优策略，用以判断。本程序中运用了树、栈、链表等多种数据结构，互相补充。算法主要采用了循环语句判断。在与 AI 的模拟战中，程序战绩优秀，但在课堂实战中，由于 max-min 没有完全构建，最终胜 6 局，未能出线。本报告将梳理对战程序的构思、要素、完成情况，以及小组工作的总结。

关键字：圈地 max-min 比较树

1 算法思想

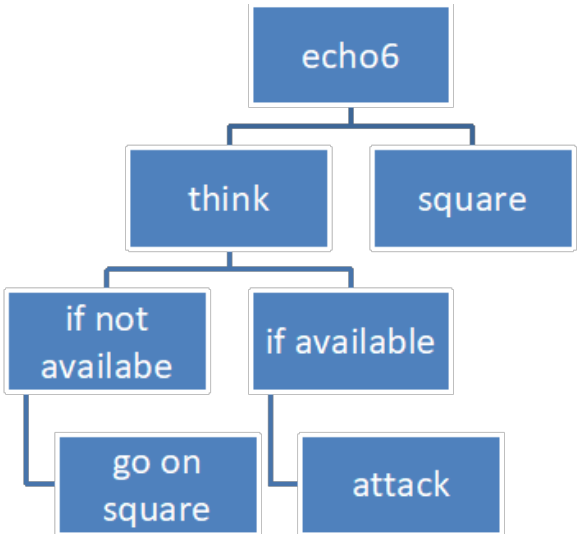
1.1 总体思路

纸带圈地游戏（paper.io，下简称圈地游戏）是一项规则简单但设计精巧的对战游戏，也是本次数算课程的大作业题目。圈地游戏目前没有明确的必胜策略，致胜除了要使自己的圈地范围尽量大之外，也要考虑到对手的运动，这便牵扯到了防守与进攻问题。本组认为，要想胜利，必须在圈地和防守进攻间达成相对平衡，这也是小组程序设计的根本思路。

为了达到平衡，本小组将程序思路分为三个模块考虑。首先是圈地（square），即传统的圈地策略，目标是使在离敌方较远时能够使领地最大。这主要通过判断语句实现。第二个模块是思考模块（think），采用 min-max 的比较思想，即通过树（三叉树）来分析每一轮本方和对方的最优策略。每方都会不断选择自己的最优策略，如果发现有机会进攻，则

本方将前进击杀；如果发现没有进攻机会且自身可能危险，则找寻最短的“回家”道路。本程序采取评分制（value）来分析进攻 -防守状态。进攻（attack）是指在短距离内击杀对方的策略。

1.2 算法流程图



1.3 算法运行时间复杂度分析

根据本组的设计思路，程序主要时间耗费在 attack 中的 max-min 部分。因为 max-min 在未优化时，如果本方和对方往下依次各搜索 n 步，则其节点数目是 3^{2n}c 个，数量巨大。因此在后续优化时，一定要有剪枝，这样才可以避免超时。

2 程序代码说明

2.1 数据结构说明

我们采取的基本思路是对每个局面进行一定程度的量化评分，然后对每次面临的三个选择之后的局面进行评分，选择其中分数最高的策略。（即如下的 next[‘value’]）

```
# 三个选择next[0],next[1],next[2] 分别对应不变，左转，右转，其中value是评价选择好坏的函数,初始设为0
next = [{}, {}, {}]
next[0]['x'], next[0]['y'], next[0]['choice'], next[0]['value'], next[0]['d'] =
myx + changex[dir], myy + changey[dir], 'N', 0, dir
next[1]['x'], next[1]['y'], next[1]['choice'], next[1]['value'], next[1]['d'] =
myx + changex[(dir - 1) % 4], myy + changey[(dir - 1) % 4], 'L', 0, (dir - 1) % 4
next[2]['x'], next[2]['y'], next[2]['choice'], next[2]['value'], next[2]['d'] =
myx + changex[(dir + 1) % 4], myy + changey[(dir + 1) % 4], 'R', 0, (dir + 1) % 4
```

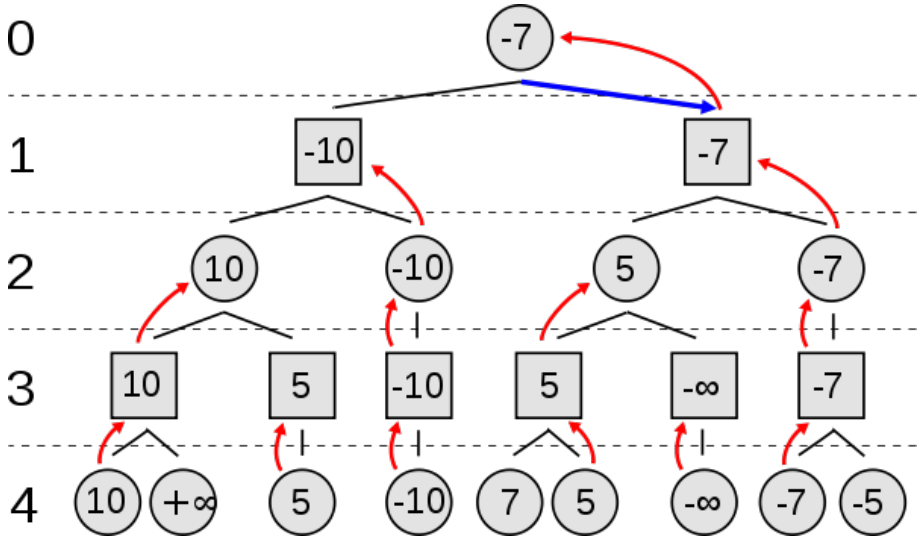
程序中主要应用两个数据结构：

一是对将每个局面搜寻安全的“回家”路径加入到一个队列中（由于敌方纸卷（enemy）可以攻击纸带的特点，所以即时一条可能路径需较多长度才能“回家”，但仍因为离 enemy 距离最远而成为最优解），因此我们需要逐一进行比对，确保存在安全的路径，而搜索选取 BFS 广度优先搜索模式。

二是建立的搜索树：在我们选择之后对方也同样有三个选择，对于 enemy 来讲，最优的策略自然是选择评分最低的路径（对我们来说是最差的结果），继续下去的选择无非也保持这样的循环。因此我们建立了一个三叉树，每个节点的子节点是其面临的三个选择后的局面，我们希望 2*n 层后我们的局面是最佳的！

但不幸的是，3^2n 量级的算法无疑太庞大了，因此面对这个问题，我们提出了两个解决方案：第一是在局面的初期，我们并不需要这么复杂的搜索，因为我们和地方的目的只有一个，即在安全地条件下圈最大面积的地，几乎不会出现对彼此的撞击和到对方领地内圈地等操作。因此我们只需要以正方形的形状在领地边缘圈地：即引入 square 模式；

二是对搜索树进行优化：一旦某一节点出现撞击或被撞击等情况，其所有的子树都并没有意义再进行搜索下去；还有对已经搜索的节点进行标记，如果在之前已经到达过并且围成的面积更大，则当前节点的子树叶无需再搜索。算法的基本图例如下（图片引用自维基百科）



2.2 函数说明

对存储函数 load 中的一些元素说明如下：重要的参考量包括 clock（我们并不希望路径形成过多的曲折，那样在围地面积少的条件下还更容易被攻击），因此我们更倾向于维持一个恒定的 clock；bandslist 和 opbandslist 是随时维护的我方和敌方的纸带列表，mode 是当前所处模式；turn 在 square 模式中判断当前是应继续向前还是回家中起作用, maxlen 是在 square 模式中选择的最大长度。

```
storage['angusclock'] = 'N'
storage['angusbandslist'] = []
storage['angusmode'] = 'square'  # 'think' 'attack''square'
storage['angustotallen'] = 0
storage['anguslen'] = 0
storage['angusborder'] = [1000, -10, 1000, -10]  # minx,maxx,miny,maxy
storage['anguscenter'] = [0, 0]
storage['area'] = 0
storage['angusopbandslist'] = []
storage['angustarget'] = [0, 0]
storage['angusturn'] = 0
storage['angusmaxlen'] = 0
```

我们设计了对一些局面的基本评分函数，例如：一旦当前局面存在一条路径使得我方到达对方纸袋的路径长度小于对方回家形成闭合回路所需的长度，那么局面估值变成 +100000，并且模式转换成“attack”。再如当在小范围博弈时，沿当前方向的行进长度“anguslen”超过 4 时，局面估值 -1000*len，若之后的转向与当前的时针方向 clock 不同

时，局面估值 -1000。

部分函数的接口见下。

```
def my_dist_to_opbands():
    minpath = [10000, 0, 0]
    for pos in storage['angusopbandslist']:
        if dist1(myx, myy, myd, pos[0], pos[1]) < minpath[0]:
            minpath[1], minpath[2] = pos[0], pos[1]
            minpath[0] = dist1(myx, myy, myd, pos[0], pos[1])
    return minpath
```

搜索到敌方纸带路径长度的函数。

```
def search_path(curx,cury,curd,curox,curoy,curod,curlist,curolist,curdepth):
    # accurate and violent search
    # dfs
    if curdepth >= 12:
        return (0,0)
    clist = curolist
    if stat['now']['fields'][curox][curoy] != opid:
        tu = (curox,curoy,curod)
        clist.push(tu)
    else:
        return (-10000,0)
    if stat['now']['fields'][curox][curoy] == myid and curox == curx and curoy == cury:
        return (100000,0)
    if stat['now']['fields'][curox][curoy] != myid and stat['now']['bands'][curox][curoy] == myid:
        return (-100000,0)

    snext = generate(curx,cury,curd)
    best = -1000000
    ci = 0
    for i in range(3):
        nx,ny,nd = snext[i]['x'],snext[i]['y'],snext[i]['d']
        val = op_search_path(nx,ny,nd,curox,curoy,curod,curlist,clist,curdepth+1)[0]
        if  val > best:
            best = val
            ci = i
    result = (best,ci)
    return result
```

极大极小值搜索，通过递归形式调用。

切换 attack 模式：

```
elif storage['angusmode'] == 'attack':
    for pos in storage['angusopbandslist']:
        if dist1(myx, myy, myd, pos[0], pos[1]) <  dist1(myx, myy, myd, storage['angustarget']
            [0],storage['angustarget'][1]):
            storage['angustarget'][0], storage['angustarget'][1] = pos[0], pos[1]
    ci = 0
    for i in range(1, 3):
        nx, ny, nd = next[i]['x'], next[i]['y'], next[i]['d']
        tx, ty = storage['angustarget'][0], storage['angustarget'][1]
        if dist1(nx, ny, nd, tx, ty) < dist1(next[ci]['x'], next[ci]['y'], next[ci]['d'], tx, ty):
            ci = i
    next[ci]['value'] += 20000
```

当然，在两方纸卷的距离特别小的时候，我们可以对搜索进行很大程度的简化，即我们只需要搜索终点，这通过对当前位置的附近一个网格的枚举即可做到，即：

```python
def short_search_path(x, y, d, id):
    # 在短距离找最快回家的路
    nx, ny = x, y
    cx, cy = 0, 0
    if stat['now']['fields'][x][y] == id:
        result = [0]
        result.append(x)
        result.append(y)
        return result
    minpath = 10000
    for i in range(20):
        for j in range(20):
            nx, ny = x - 10 + i, y - 10 + j
            if nx < 0 or ny < 0 or nx >= stat['size'][0] or ny >= stat['size'][1]:
                continue
            if stat['now']['fields'][nx][ny] == id and dist1(x, y, d, nx, ny) < minpath:
                minpath = dist1(x, y, d, nx, ny)
                cx, cy = nx, ny
    if minpath != 10000:
        result = []
        result.append(minpath)
        result.append(cx)
        result.append(cy)
        return result
    else:
        result = [10000]
        result.append(x)
        result.append(y)
        return result
```

2.3 程序限制

尽管我们做了很多优化，但是极大极小搜索还是相当耗时，在对方选择也比较稳妥从而盘数较多的时候，程序容易用尽时间。因此在对战的环节我们弱化了这一部分，最终结果也并不令人太满意。

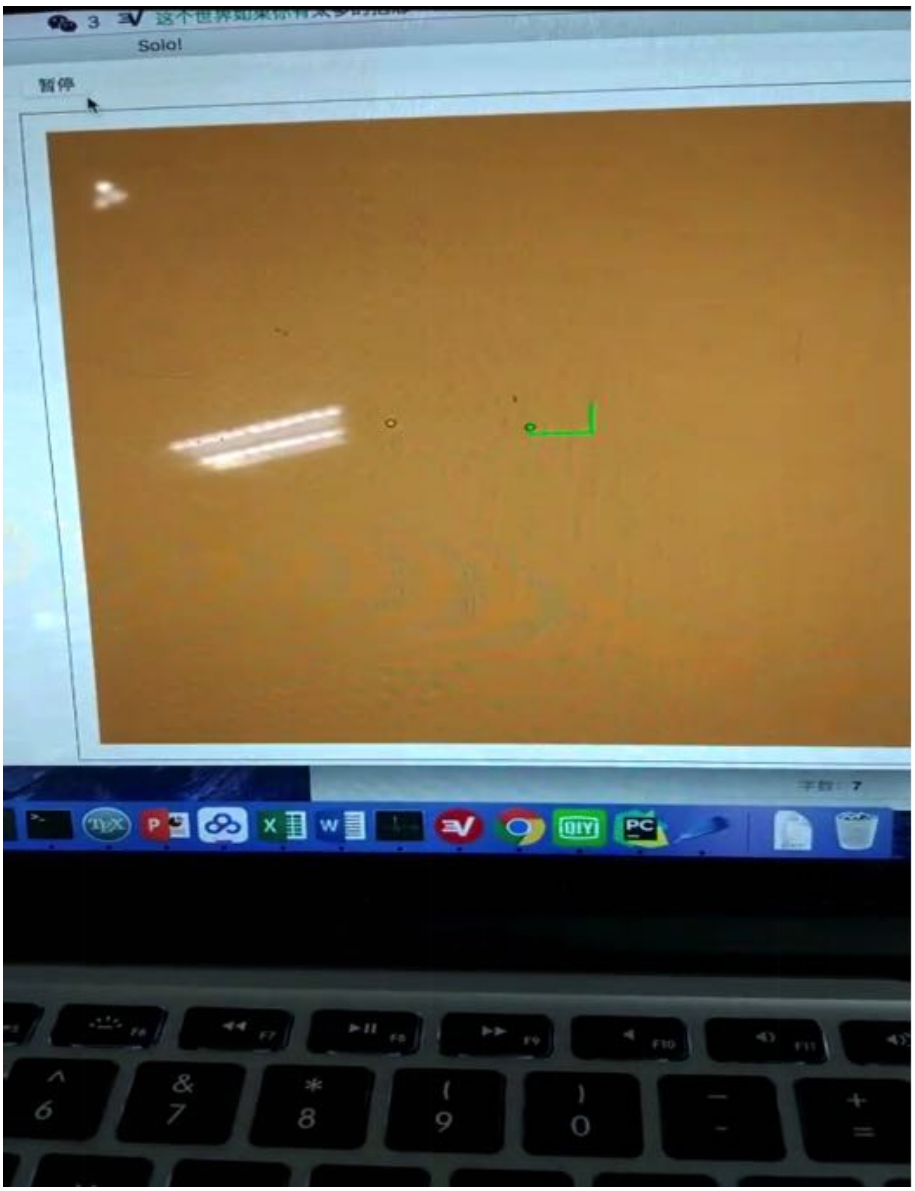
3 实验结果

3.1 结果分析

本组共参加了两轮热身赛以及正式比赛。第二轮热身赛参赛代码数量为 13，本组取得 5 胜 7 负的战绩，排名为第 9。第三轮热身赛参赛代码数量为 27，本组取得 6 胜 20 负，排名为第 23。正式比赛，本组位于 N17 的 E 组，在小组赛中取得了 2 胜 3 负的战绩，未获得出线资格。本组的代码相比其他组的代码，存在更多不完善的地方。

3.2 经典战局

我们某次的版本过分地追求了圈地面积，于是当遇到一个只具有在附近圈地而不具有主动攻击性的 AI 后出现了如下的局面（橙色为我方）：



4 实习过程总结

4.1 分工与合作

本小组组长由心理学院 15 级向飞燕同学担任。初期，由向飞燕、地空学院 14 级康峻侥和元培学院 16 级张与之同学各自独立编写代码，最终选择以张与之同学的代码为主要代码；中期，张与之同学负责代码进一步的编写与完善，康峻侥同学负责提供了对战思路的改进策略；比赛后，由地空学院 14 级尚昊成同学和元培学院 16 级张与之同学主要负责报告撰写，其他三人也有参与。

本小组共召开过三次线下讨论会，具体情况如下：

1.5 月 26 日：张与之、向飞燕、康峻侥、方鑫：初步讨论对战思路，启动程序设计

2.6 月 2 日：张与之、向飞燕、康峻侥：讨论各自撰写的代码，最终选择以张与之同学的代码为主要代码；进一步讨论算法的不足和可改进之处。（尚昊成因病请假，方鑫同学缺席）

3.6 月 12 日：全体同学。总结比赛结果，讨论报告内容，安排报告撰写。

4. 合作期间多次进行线上讨论。

小组分工有序，重点突出，其中张与之同学的贡献较大，经全体讨论，决定将组内加分奖励给张与之同学。

4.2 经验与教训

在赛后总结时，本小组认为整个实习过程最大的经验是充分发挥个人特长，分工明确。在贡献思路时，大家集思广益，共同构建 play 函数。临近热身赛时，让一名同学主要负责代码，这样可以保证思路的连贯性。同时为了平衡工作量，报告书写由另外的同学负责。这也符合小组合作的基本原则。

由于本组是 F17 组别，是跨年级合作，因此在日常沟通上存在一些延时和障碍。同时由于个人工作习惯不同，部分任务的分配不是平均，这些问题尽可能通过组内加分等调整。

4.3 建议与设想

本次实习作业是对 python 掌握程度和数据结构理解的综合考验，难度较大。在通力合作下，本组尽可能完成了任务。希望今后的大作业能够更加考虑零编程基础同学的适应强

度，同时对比赛难度进行分梯度设置。比如采用 AI 或学习算法的小组，与普通算法小组竞争，存在一定的不公平。

本组将在后续的学习中继续思考对战算法，力争解决耗时过多的难题。同时也为今后的 python 学习打下基础，希望大作业这一有益实践能对未来的学弟学妹有所帮助。

5 致谢

首先感谢陈斌老师一学期的精彩授课，通过寓教于乐的方式，让同学们能够对编程和数据结构有更感性的体会。感谢为课程建设付出的助教和本次大作业的技术组成员，为本次对战搭建了非常好的平台。感谢参与对战的所有同学，大家在学习过程中互相切磋，共同努力。

本组由两名 14 级地空同学，一名 15 级心理同学，和两名 16 级元培同学组成。感谢组长向飞燕的组织和统筹工作。感谢本组技术负责人张与之同学的辛苦付出，是本组对战最大的贡献者。康峻侥和尚昊成同学对本组的工作也付出了大量努力，体现了小组合作的成果。

6 参考文献

极大极小算法：`https://en.wikipedia.org/wiki/Minimax`

第六章　N17_Foxtrot 报告

高诗铠、彭金山、彭晓韵 *、王都越、王逸颖、魏静怡

摘要：我们的算法策略分为：攻击 (attack)、领地内游走 (wander) 和领地外圈地 + 防御 (walk)。每一步，算法会根据自己所处的状态、敌人的状态和场地情况，选择不同的应对策略。这里除了基本的线性数据结构，例如字典、列表、嵌套列表，还使用了 BFS 搜索图的思想和启发式算法。

算法首先判断“能否攻击”，如能必杀则击杀对方。我们希望首先能增加攻击和防御函数，但是考虑到攻击敌人的最短距离和回领地的最短距离难以用简单算法实现，所以在 supervision 版本中，试图使用 BFS 广搜算法（在算法竞赛这份代码仍有一些 bug，在当天下午 4 点才得以修复，没能参赛，不过所幸最后有机会参加了友谊赛）。之后算法根据是否在自己领地内划分为“wander”和“walk”两个策略。在 wander 时，我们希望自己能尽快走出自己的领地进行圈地，同时选择最优的方向走出。为了能在行走的方向选择上进行优化，我们使用了“启发式”思想，定义了启发式函数，对于不同格点的信息（属性，距离边界和敌人纸卷等）进行打分，对周围一定区域的格点分数累计，根据启发式函数的优先级进行判断，选择最优的方向。最后，我们对于代码中的一些细节进行了优化，在“walk”模式中强化了防御功能，最终得以实现代码。

热身赛中我们组的代码分别被 delta、hotel、kilo、quebec、india 打败，热身赛中没有达到小组出线。友谊赛中 Supervision 版本的代码与冠军 Delta 神仙打架，基本实现了我们希望代码实现的功能。

关键字：字典列表嵌套列表 BFS 搜索图启发式算法

1 算法思想

1.1 总体思路

1.1.1 superversion 版

算法总体的思路为：根据每一步自己所处的状态、敌人的状态和场地情况，选择不同的应对策略。应对策略分为：攻击 (attack)、领地内游走 (wander) 和领第外圈地 + 防御 (walk)。

算法特点是使用了 BFS 搜索和启发式算法。

为了为策略判断做准备，我们首先使用了 BFS 广度优先搜索算法，用于搜索最短距离和规划路径，分为“自己返回领地的距离和路径”、“敌人返回领地的距离”、“自己攻击对方的距离和路径”，以及“敌人攻击我的距离和路径”。每个格点的下一步有 3 种走法，遍历这三个方向，将符合条件的格点加入搜索队列，即可得到距离起点的距离和路径。为了节约时间，将搜索深度定义为 50。

在策略选择中，主要是通过简单的判断进行不同策略的切换。

首先判断是否能够攻击对方。先利用 BFS 中敌人的 enemy_return_bfs 函数，得到敌人的最短回程距离，随后调用自己 BFS 的攻击函数 me_attack_bfs，将自己的最远攻击距离设成敌人的最短回程距离（如果存在），尝试寻找自己的攻击路径。如果路径存在则立即进行攻击，并保存为“攻击”模式，持续攻击直至攻击列表为空。

接着判断下一步可能遇到敌人纸卷的情况，即纸卷相撞。遍历下一步可走的三个方向，若能撞到敌人，可分为头碰头直接撞击和侧碰的情况。若能侧碰，则我方赢，返回该方向；若头碰头直接撞击，分为在自己领地，在敌人领地和在无属性领地。在自己领地撞击则自己赢，返回该方向；若在无属性领地，且自己的分数（圈地面积）大于对方，则自己赢，也返回该方向；其他情况下跳过该方向继续遍历其他方向。

若不能攻击对方，则根据是否在自己领地，分为“wander”和“walk”。在自己领地 wander 时，有一系列需要遵循的原则。所以我们使用了“启发式搜索”的想法，运用 heuristic_comp 函数，设定规则，尽量寻找最优的方向。Wander 时，选方向的原则为：遍历周边 5*5 区域，优先寻找对手领地和空地，其次是靠近敌方边界，最后是离敌人纸卷较近。为了实现这些功能，在该函数内部再定义了三个函数：comp_grid_score、

comp_dist2border、comp_dist2enemy，根据上述原则设置优先级。

走出自己场地时，切换到“walk”模式，即领地外“圈地 + 防御”模式，该算法采取保守策略，每一步先计算是否能安全返回领地。遍历三个走向，对每个走向先使用了 me_return_bfs 函数和 enemy_attack_bfs，判断是否能找到安全逃回自己领地的路径，若能找到返回路径，则将该方向存入 walk_directions 列表中，如果在遍历三个方向后，发现有多种安全的走向，则使用另一个启发式搜索函数 heuristic_comp2 进行最佳方向判定。与 wander 模式中的启发式函数 heuristic_comp 相比，做了一些微调，第一原则不变，优先圈敌人的地和空地，但是把第二原则改成了“离自己边界近更好”，有利于防御敌人。根据启发式原则找到最优的下一步走向。如果在 Walk 模式中，出现了无论何方向走，敌人都可能攻击到自己的情况，则说明没有绝对安全的返回路径。此时，只能选择 walk_directions_bk 列表中具有最短返回距离的方向，尽快回到自己领地（虽然依然可能被攻击）。

该代码为了存储游戏基本信息，使用了一些基本的线性数据结构，例如字典、列表、嵌套列表等。除此以外，还使用了比较复杂的数据结构，主要运用了 BFS 搜索图的思想。

在几个主要的广搜函数中，本代码生成了用于搜索分析的队列，对已发现的顶点进行排列和存储。但是本代码没有先建立完整的图，而是对每一个队列中的顶点，遍历三个可选的方向，找到当前顶点对应的下一步顶点，再加入队列中。这样的策略类似“边建图边搜索”，在存储空间和运行时间上有一定优化。与课上 BFS 算法类似，我们也为顶点增加了 distance 属性，便于追踪路径。当首次找到终点（敌人领地）后，返回记录格点距离信息的二维列表与最短距离，实现所需功能。

该算法是与下述“普通版”算法同时推进的，但是功能上更为强大，细节更加完善。我们希望首先能增加攻击和防御函数，但是考虑到攻击敌人的最短距离和回领地的最短距离难以用简单算法实现，所以希望尝试 BFS 广搜算法。由于小组成员大多是“新手”，所以在 BFS 的细节上出现过一些 bug，使得该算法进度一直比较慢。进过努力，成功对 BFS 算法 debug 后，就需要进行策略的选择和实现。策略上基本比较清晰，主要还是攻击、防御、圈地三种。有了 BFS 算出的距离和路径，这些策略的实现并不是很难。为了方便策略状态切换，我们首先判断“能否攻击”，如能必杀则击杀对方。其他情况下，根据是否在自己领地内划分策略为“wander”和“walk”。和下述普通版代码类似，我们希望能在行走的方向选择上进行优化。所以，我们使用了“启发式”思想，定义了一些启发式函数，对于不同格点的信息（属性，距离边界和敌人纸卷等）进行打分，对周围一定区域的格点

分数累计，根据启发式函数的优先级进行判断，选择最优的方向。最后，我们对于代码中的一些细节进行了优化，在“walk”模式中强化了防御功能，最终得以实现代码。可惜的是，在算法竞赛前，该代码仍有一些 bug，在当天下午 4 点才得以修复，没能参赛，比较可惜。幸运地是，最后一节课上，老师让我们的代码和冠军们进行友谊赛，且该代码表现不错，也是对我们付出的肯定。

1.1.2 普通版

算法总体的思路为：根据每一步自己所处的状态、敌人的状态、场地情况，选择不同的应对策略。主要的应对策略分为“领地内游走和方向选择”wander，“正方形圈地”square、“自救”isolate 和“防御”chooseDirectionmargineperry。Wander 适用于在自己领地内的情况，square 适用于从自己领地出发，侵入敌人或者无标示地盘的情况。Isolate 适用于当自己领地被别人切割或占领，自己完全处于敌人领地时的情况。chooseDirectionmargineperry 适用于即将出界和走出领地时离对方太近的情况，采取防御策略尽快返回自己领地。

在 wander 时，我们希望自己能尽快走出自己的领地进行圈地，同时选择最优的方向走出。我们定义了“chooseDirection”函数，对以自己为起点，四个方向上的正方形区域（边长为自己最多能走的安全距离）进行遍历，对其中每个格点进行打分，出界扣分，在自己领地时分数不变，进入无属性领地时加 1 分，进入敌人领地加 5 分。对区域内所有格点的得分进行加和，再进行四个方向区域得分比较，可选定得分最高的区域所在方向为下一步走的方向。通过这个函数，可以使自己尽量走出自己的领地，且倾向于占领敌人领地，得到更大的圈地面积。

在 isolate 状态下（领地被对方切割，变为在敌方领地画纸带），我们希望自己能尽快返回自己的领地，使用了“chooseDirectionisolate”函数进行方向判断。与“chooseDirection”函数类似，对以自己为起点，四个方向上的正方形区域（边长为自己最多能走的安全距离）进行遍历，对其中每个格点进行打分，打分原则为，出界扣分，进入自己领地加 10 分，进入无属性领地加 1 分。对区域内所有格点的得分进行加和，再进行四个方向区域得分比较得到最优方向。在这种情况下，己方纸带会尽量向敌人领地外走，并尽量返回自己领地，从而实现“自救”。

在 square 模式下，进行正方形状圈地，正常情况下圈地的方向（storage['turn']）和边长（storage['maxl']）均在 wander 时确定，可以安全进行圈地。如果出现下一步的纸带属

性是己方，即领地被对方切割，则切换成 isolate 状态下，尽快返回自己领地进行自救。

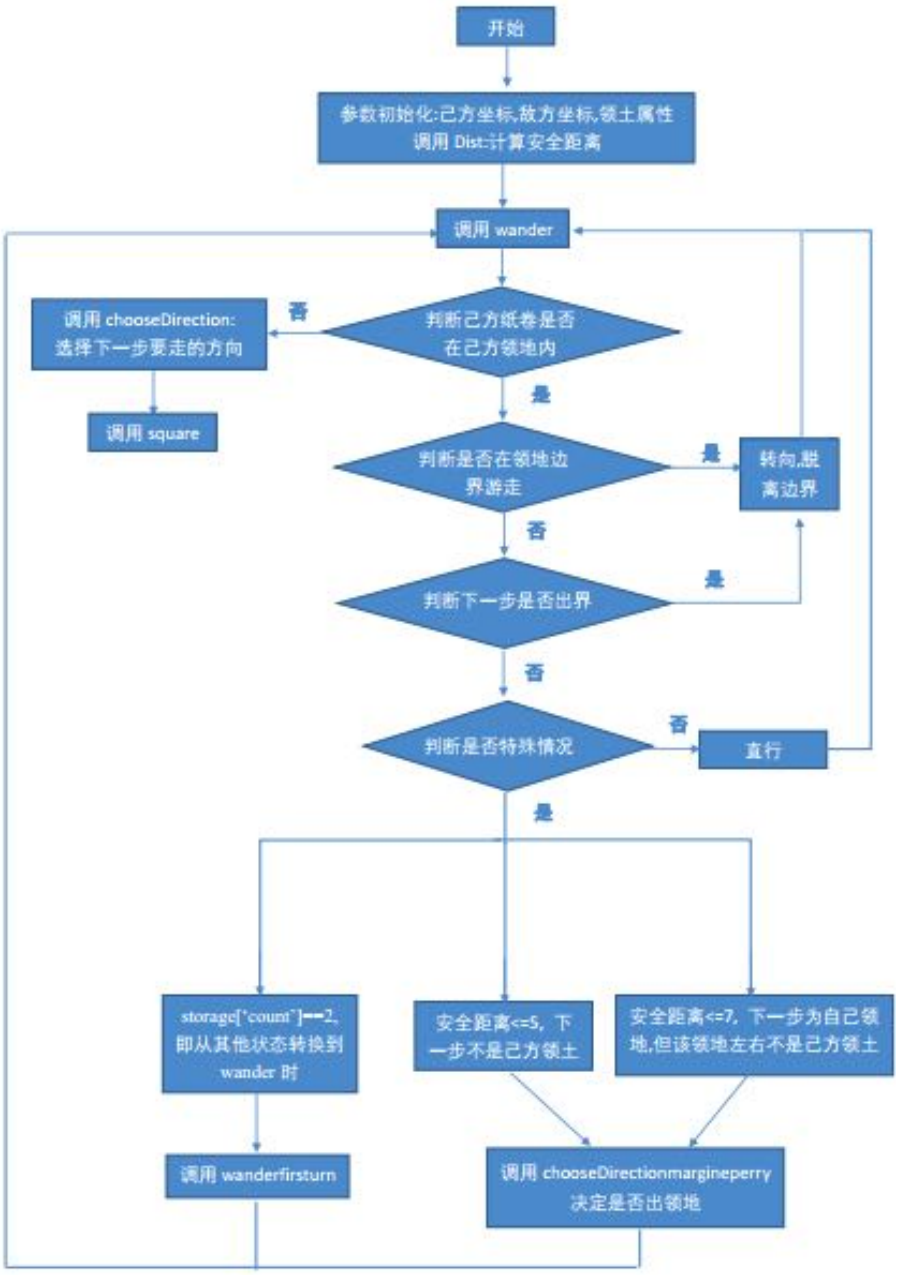
根据己方所处的状态，可以进行状态切换。初始状态是 wander, 在走出自己领地后，会切换成 square，square 画完回到自己领地后又回到 wander，在这些过程中如果出现领地被对方切割，自己不得不在对方领地画纸带时，状态切换为 isolate，尽快返回自己领地进行自救。

此外，还有快速返回自己领地的“chooseDirectionmargineperry”，在 wander 模式下调用，可以在自己即将出界和离对方太近时，立即采取防御策略，尽快返回自己领地。

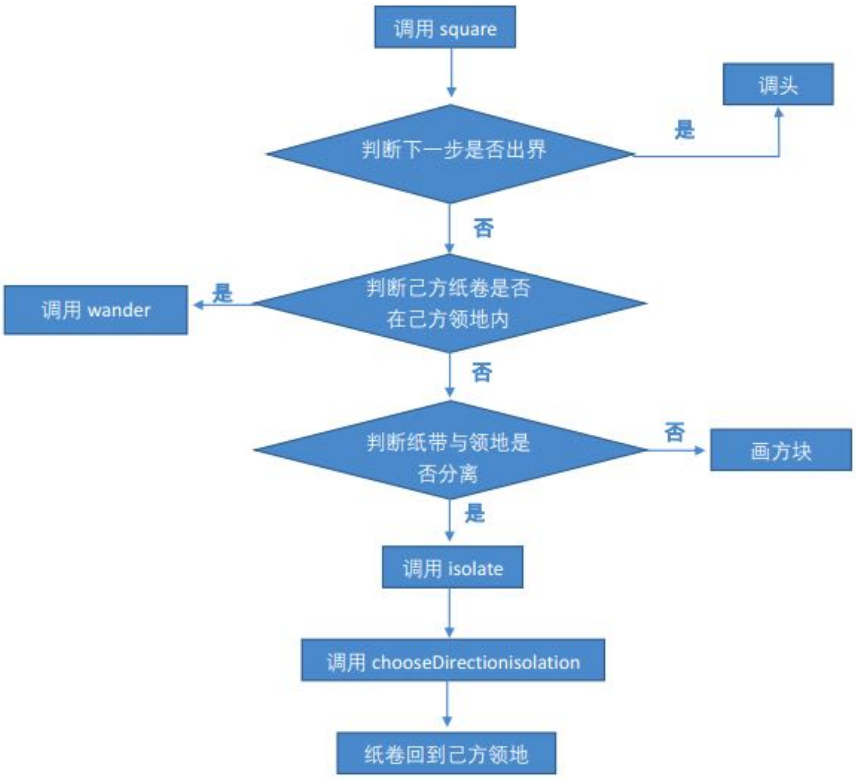
算法采用的主要数据结构基于字典和列表。例如 stat 和 storage 以字典形式存储了游戏中的基本信息和各种战略模式，field 和 band 等以二维列表形式存储棋盘上领地和纸带的归属状态。在“chooseDirection”函数中采用了二维遍历的思想，对以自己为起点，四个方向上的正方形区域（边长为自己最多能走的安全距离）进行遍历，对每个格点进行打分（类似启发式算法），累计后选择最优方向。

该算法最初有了“策略选择”的想法，收到 wanderer 的启发，考虑选择 wander、square 和 goback 三种状态。经过仔细分析，我们发现 goback 函数主要只是起到“掉头”和状态转换衔接的作用，于是我们删去这一函数，将其功能融入 wander 和 square 中。改变后，在这两个模式中，首先考虑是否需要防御返回（出界和离敌人近的情况），同时在 wander 即将切换成 square 时，将画 square 的方向和边长存入 storage, 进行衔接。另外，原始的 wander 并不能很好地选择最优行走方向，经常在领地中徘徊，所以我们增加了“chooseDirection”函数，对以自己为起点，四个方向上的正方形区域（边长为自己最多能走的安全距离）进行遍历，对其中每个格点进行打分，累计比较后得出最优方向，以期自己能尽快走出自己的领地进行圈地，同时选择最优的方向走出。另外，在几次热身赛中，我们发现，当自己领地被对方切割，使得自己不得不在对方领地画纸带时，容易死亡。所以我们增加了“isolate”自救模块，利用 chooseDirection 函数，会尽量向敌人领地外走，并尽量返回自己领地，从而实现“自救”。

1.2 算法流程图


开始
参数初始化:己方坐标,敌方坐标,领土属性
调用 Dist:计算安全距离
调用 wander
判断己方纸卷是否
在己方领地内
否
调用 chooseDirection:
选择下一步要走的方向
调用 square
是
判断是否在领地边
界游走
是
转向,脱
离边界
否
判断下一步是否出界
是
否
判断是否特殊情况
否
直行
是
storage["count"]==2,
即从其他状态转换到
wander 时
安全距离<=5，下
一步不是己方领土
安全距离<=7，下一步为自己领
地,但该领地左右不是己方领土
调用 wanderfirsturn
调用 chooseDirectionmargineperry
决定是否出领地

主图



分图 —-SQUARE 函数

1.3 算法运行时间复杂度分析

1.3.1 superversion 版

该算法主要采用的是 BSF 广搜算法和启发式搜索方法，这两个主要方法耗时最长。

BFS 广搜函数的算法主体是两个循环（while-for）的嵌套，while 循环图中每个顶点（O），for 循环每个顶点的边（E），所以复杂度一共为 O(V+E)。本算法中，每个顶点出发只能选择 3 种方向，为一个固定常数，所以经历的最多边数和顶点数成比例，所以 BFS 的复杂度与经历的顶点数同阶。如果遍历整个棋盘，则复杂度为 O（N*N），即 O（N2）。回溯寻找路径过程中，运用类似的想法，复杂度仍然为 O（N2）。本算法中，减少搜索距离至 50 步，可以减小绝对耗时，但是这个搜索距离也和场地长、宽成正比（如果长、宽减小，这个搜索距离也相应减小），所以算法复杂度还是 O（N2）。但是，由于缩小了搜索距

离，实际运行中，时间开销的确降低了不少，由原来容易超时（30s 以上，甚至 60s 以上）变成了保证 30s 以内，减少了绝对时间开销。

同时，我们在非攻击模式中，使用了启发式搜索方法，即对周围一定边长范围内的格点进行打分、比较，运行时间复杂度和搜索的范围边长相关。如果对整个棋盘进行搜索和运算，复杂度为 O（N*N），即 O（N2）。但是我们将搜索范围定为 5，其实是一个常数，所以从复杂度上讲，是 O(1) 的复杂度。

另外，在最初的信息获取中也用到了“遍历”操作。get_env_info(fields, bands) 函数遍历整个棋盘，获取格点属性（bands 和 fields 属性），同时更新某方最远达到的边界。所以对每个格点都进行了判断和计算，总耗时也达到 O(N2)。

综上，该算法整体的时间复杂度为 O(N2)。

1.3.2 普通版

本算法中，每一步运行时需做一系列状态判断，其中大部分都是简单的 if 判断，直接进行简单运算和判断，大多数操作时间复杂度为 O（1）。除此外，本算法最主要的耗时在确定方向的“遍历”操作中。在 wander 模式中，需要进行 chooseDirection 判断，而“chooseDirection”函数需要对以自己为起点，四个方向上的正方形区域（边长为 maxl, 即自己最多能走的安全距离）进行遍历。所以时间复杂度应为 O(maxl2)。理论上说，自己和敌人的距离是和棋盘长或宽成比例的，最大甚至可以达到对角线长度，所以，若认为问题规模 N 是场地的长或宽，那么时间复杂度可能达到 O（N2）。

2 程序代码说明

2.1 数据结构说明

2.1.1 基本型数据结构

在两个算法中，首先都使用了一些基本的线性数据结构，存储基本信息，例如字典、列表、嵌套列表等。

stat：游戏数据

类型：dict‘size’:int ,etc

内容：记录场地信息，运行记录，当前场地信息等

size：列表，包含游戏场景宽高

类型：list[int]

内容：[场地宽, 场地高]

log：运行记录

类型：list[dict]

内容：对局开始以来每一回合的场地信息

players：玩家信息列表，包含双方玩家信息，按先后手排序

类型：list[dict]

内容：[先手玩家信息, 后手玩家信息]，先手为 0，后手为 1

Bands：玩家纸带列表

类型：list[dict]

内容：[先手玩家信息, 后手玩家信息]，先手为 0，后手为 1

Storage：自设储存

类型：dict'mode'：wander/square，'turn'：'r'/'l'，'count'：int，'mark'：int，'direction'：int

内容：用于函数储存使用，'mode'对应纸卷运动状态：随意走动或者圈地；'turn'对应纸卷的转向；'count'对应圈地时的最大边长；'direction'对应当前纸卷前进的方向

Borders：玩家经历的最大边界列表

类型：list[list]

内容：[[leftborder, rightborder, upborder, downborder], [leftborder, rightborder, upborder, downborder]] 大列表中元素为两个子列表，分别记录己方或敌方数据。其中每个子列表有 4 个元素，分别记录该方的上、下、左、右经历区域的最远边界。

2.1.2 复杂型数据结构

superversion 版本中，使用了比较复杂的数据结构，主要运用了 BFS 搜索图的思想。

四个主要的广搜函数采用了“图”的思想。课上例子中首先建立了两个自定义“类”，分别记录顶点和图的信息，包括顶点间连接的边，顶点的前驱和后继等，从而建立了完整的“图”。接着生成一个队列，从起始顶点开始，遍历后继顶点，对已发现的顶点进行排列，决定下一个要探索的顶点。而本代码运用类似的思想，生成了用于搜索分析的队列（本代码使用了列表，功能类似），对已发现的顶点进行排列和存储。但是本代码没有先建立完整的图，而是对每一个队列中的顶点，遍历三个可选的方向，找到当前顶点对应的下一步顶点，再加入队列中。这样的策略类似“边建图边搜索”，在存储空间和运行时间上有一定优化。与课上 BFS 算法类似，我们也为顶点增加了 distance 属性，但是没有使用 color 属性，因为已经遍历过的顶点的 distance 非 None, 而即将遍历的顶点已经位于 wait_list 列表中，所以可以省去 color 属性。

探索到顶点后用四个方向向量作为建立“边”的工具，从起点开始，遍历经过四个方向后下一步的格点，从而实现对用队列（本代码使用了列表，功能类似）存储需要搜索的“顶点”（即格点），接着遍历其邻接顶点（即下一步所有可能到达的格点），经过简单规则判断（不出界和不撞纸带）后，将该顶点离起点的距离存储到 bfs_grid 二维列表中。当首次找到终点（敌人领地）后，返回 bfs_grid 二维列表与最短距离。与课上使用一个队列来存储要搜索的顶点不同，我们使用了两个列表，check_list 和 wait_list，在区分正在检验的顶点和下一步要搜索的顶点，有利于更好地定义“距离”。

2.2 函数说明

2.2.1 superversion 版

（1）基本型函数：

check_valid 函数用来检查是否出界及是否撞自己的纸带

check_valid_simple(next_x, next_y) 仅检查是否出界，在自己领地内行走时适用（不存在撞自己纸带的问题）

get_env_info(fields, bands) 函数：获取场地信息，遍历整个棋盘，获取双方纸带长度、地盘面积（即得分），并缩小 border 的边界至仅有“有属性条带”和领地的范围

（2）主要广搜函数：

本算法使用了四个主要的广搜函数，用于搜索最短距离和规划路径，分为 enemy_return_bfs、me_return_bfs、me_attack_bfs、enemy_attack_bfs。

enemy_return_bfs 函数用来找敌人返回敌人领地的最短路径。与课上提到的广搜思想类似，用队列（本代码使用了列表，功能类似）存储需要搜索的“顶点”（即格点），接着遍历其邻接顶点（即下一步所有可能到达的格点），经过简单规则判断（不出界和不撞纸带）后，将该顶点离起点的距离存储到 bfs_grid 二维列表中。当首次找到终点（敌人领地）后，返回 bfs_grid 二维列表与最短距离。

同理可以得到 me_return_bfs 函数，用来寻找自己返回自己领地的最短路径。

me_attack_bfs 用来寻找自己攻击敌人纸带的最短路径。与上述的广搜思想类似，用 check_list 和 wait_list 分别存储正在搜索的顶点和下一步需要搜索的顶点，这里需要剔除敌人返回领地路径上的点（比较保守的攻击策略），最后，当首次找到终点（敌人纸带）后，返回 me_bfs_grid 二维列表（存储了棋盘上顶点距离起点的广搜距离）和攻击位置 attack_position。

同理可以得到 enemy_attack_bfs 函数，用来寻找敌人攻击自己纸带的最短路径。

在这四个主要广搜函数的基础上，为了方便后续具体操作，继续定义了一系列函数，如下所示。

find_attack_path(end_x, end_y, bfs_grid) 函数用于返回自己的攻击路径。它读取 me_attack_bfs 函数的 me_bfs_grid 二维列表和攻击位置（即路径终点坐标）。接着从路径终点坐标开始，遍历三个方向回溯，遇到 distance 值为当前 distance-1 的格点即结束循环，将行走方向逆转后加入 attack_path 列表中，更新当前位置，进行下一轮搜索，直至遇到起点。此时返回 attack_path 列表，即为攻击路径。

walk_defend_bfs 函数在 me_return_bfs 函数基础上增加了判断，即能否被敌人攻击到，其作用是找到一条不被敌人攻击的安全返回路径。其整体结构与 me_return_bfs 函数类似，但是比 me_return_bfs 函数多读入 enemy_attack_bfs 函数的 enemy_bfs_grid 二维列表，对每一个格点，判断在 enemy_bfs_grid 二维列表中，它是否有非 None 的且小于自己后退距离的数值（说明在自己返回领地前敌人可能攻击到自己），依此进一步筛去一些格点。遍历完毕后，如果能找到安全返回的路径，那么返回终点的坐标，否则返回 None。

find_defend_path(end_x, end_y, bfs_grid) 函数用于返回自己的安全返回路径。它读

取 walk_defend_bfs 函数的返回值坐标，遍历三个方向回溯，与 find_attack_path 函数类似进行计算和搜索，最终得到 defend_path 列表。

（3）策略类函数（含启发式搜索函数）

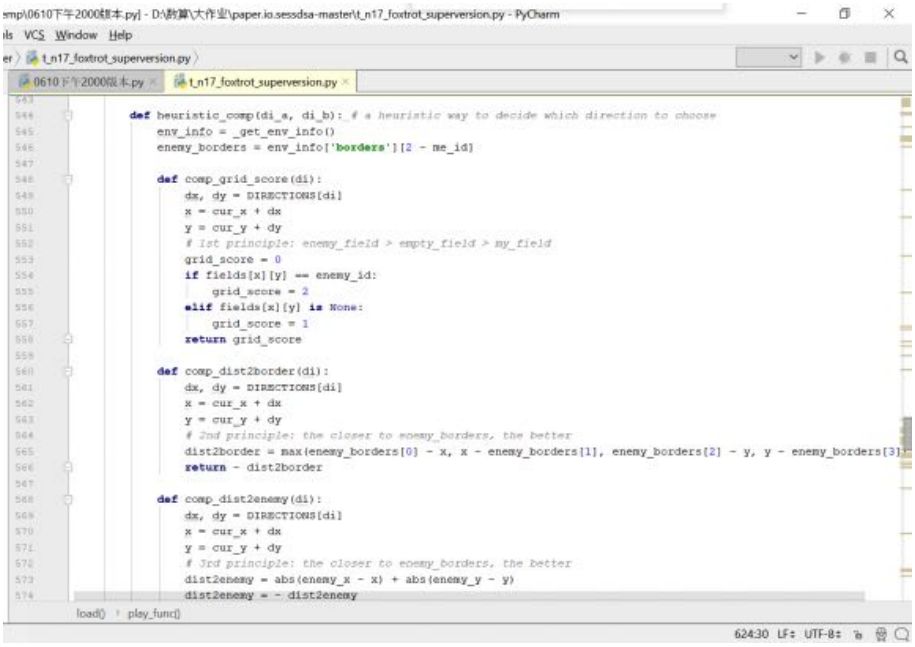
我们使用了 play_func(stat, storage) 函数，用于决定游戏中的具体策略。在读取了 stat 函数中各种基本信息后，根据不同模式 mode 执行操作。

在'attack_band' 模式下，使用 find_attack_path 函数的 attack_path 列表，一一弹出方向并返回，直至列表为空，列表为空时将 storage['check_attack_left'] 赋值为 0，可立即检查是否可以再次攻击对方。在 storage['check_attack_left'] 为 0 时，判断是否能攻击对方。首先利用敌人的 enemy_return_bfs 函数，搜索深度定为 MAX_ATTACK_DIST（最大攻击距离），得到敌人的 enemy_bfs_grid 和最短回程距离，随后将自己的最远攻击距离设成敌人的最短回程距离（如果存在），尝试用 me_attack_bfs 寻找自己的攻击路径。如果路径存在，切换 mode 为'attack_band'，存储 attack_path 列表至 storage 函数中，为攻击做好准备。否则 mode 不变。

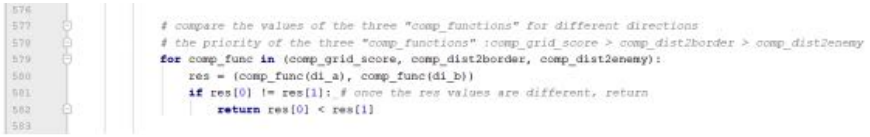
接着判断下一步可能遇到敌人纸卷的情况，即纸卷相撞。遍历下一步可走的三个方向，若能撞到敌人，可分为头碰头直接撞击和侧碰的情况。若能侧碰，则我方赢，返回该方向；若头碰头直接撞击，分为在自己领地，在敌人领地和在无属性领地。在自己领地撞击则自己赢，返回该方向；若在无属性领地，且自己的分数（圈地面积）大于对方，则自己赢，也返回该方向；其他情况下跳过该方向继续遍历其他方向。

若不能攻击对方，则固定模式为“wander”。在自己领地 wander 时，有一系列需要遵循的原则。所以我们使用了“启发式搜索”的想法，运用 heuristic_comp 函数，设定规则，尽量寻找最优的方向。

heuristic_comp(di_a, di_b) 是启发式的找寻方向的函数，原则为：优先寻找对手领地和空地，其次是靠近敌方边界，最后是离敌人纸卷较近。为了实现这些功能，在该函数内部再定义了三个函数：comp_grid_score、comp_dist2border、comp_dist2enemy，优先级为 comp_grid_score > comp_dist2border > comp_dist2enemy。



对于每一步决策，先遍历可走的三个方向，筛选掉出界的情况后，将可选的方向存入 wander_directions_bk 列表，接下来进一步考虑在踏出边界时，周围 5*5 区域的情况，如果有敌人存在，则不考虑该方向的走法，即在走出自己领地时会尽量躲避 enemy。用 wander_directions 列表存储经过两次筛选后的可选方向（即在 wander_directions_bk 基础上扣除领地外遇见敌人的方向）。如果筛选完毕后 wander_directions 为空（即无论哪个方向走都可能遇到敌人），则只能在 wander_directions_bk 中随机选定方向走一步。在其他情况下，对于 wander_directions 列表中的方向，再使用上述启发式的原则，使用 heuristic_comp(di_a, di_b) 函数，判断最优的行走方向并返回该方向。



Walk 是领地外“圈地 + 防御”模式，该算法采取保守策略，每一步先计算是否能安全返回领地。遍历三个走向，对每个走向先使用了 me_return_bfs 函数，判断是否能找到逃回自己领地的路径，若可以则将该方向加入 walk_directions_bk 列表中。同时注意敌人的情况，使用 enemy_attack_bfs 函数，判断敌方是否可以在自己返回之前攻击到自己，在敌人攻击不到自己时用 walk_defend_bfs 函数寻找返回路径，若能找到返回路径，则将

该方向存入 walk_directions 列表中。如果在遍历三个方向后，walk_directions 列表中元素大于 1，即有多种安全的走向，则使用另一个启发式搜索函数 heuristic_comp2 进行最佳方向判定。与 wander 模式中的启发式函数 heuristic_comp 相比，做了一些微调，第一原则不变，优先圈敌人的地和空地，但是把第二原则改成了“离自己边界近更好”，有利于防御敌人。根据启发式原则找到最优的下一步走向。

如果在 Walk 模式中，出现了无论何方向走，敌人都可能攻击到自己的情况（walk_directions 列表为空），则说明没有绝对安全的返回路径。此时，只能选择 walk_directions_bk 列表中具有最短返回距离的方向，尽快回到自己领地（虽然依然可能被攻击）。

2.2.2 普通版

本程序借鉴了 wanderer 的框架，由 play 和 load 两个主要函数组成，其中的具体细分函数和判定规则均和 wanderer 不同。

play 函数和 load 函数接受 stat 和 storage 两个参量，其中 storage 是我们自己保存、使用、更新的字典，其中存储了 mode,turn,count,direction 等的值。

在初始运行 load 函数之后，每一局运行 play，根据 mode 的值，来返回前进，左转或者右转。纸卷的运动状态有三种：领地内行走 wander，领地外圈地 square，领地外被别人包围时脱困 isolate，分别对应 mode 的三个值：’wander’, ‘square’, ‘isolate’。

当函数处在 wander 状态时，调用 wander 函数。首先判断纸卷是否在领地内，如果发现纸卷走出了领地，就要将模式转换为 square，决定圈地边长，选择圈地方向，进行圈地。如果纸卷还在领地之内，就要按顺序进行如下判断：

1. 如果发现纸卷贴场地边缘行走，则让纸卷转弯，比如自身 x 坐标 =stat[‘size’][0]-1，行进方向是向北，就要立刻左转；

2. 如果沿着既定方向，下一个格子出界，就要转弯，若左边格子是自己的领地，就左拐，反之右拐；

3. 如果现在在自己领地，下一个格子不在自己领地，并且与对方距离小于等于 5，就要调用 chooseDirectionmargineperry;

4. 如果与对方距离小于等于 7，下个格子是自己的领地但是那个格子左右都不是自己的领地，就要在当下提前转弯，同样调用 chooseDirectionmargineperry;

5. 如果 storage[‘count’]==2，就要调用 wanderfirstturn 返回’l’/’r’，否则直行。

在 wander 状态调用的函数功能如下：

chooseDirectionmargineperry，返回’l’/’r’，让纸卷在领地边界转到自己的领地内，防止离对方太近的时候出领地被攻击，而如果左右两边都不是边界，则调用 chooseDirection 函数。

wanderfirstturn 是进入领地转换为 wander 模式后调用的，它决定了最初的转向，首先按照既定方向调用 chooseDirection，边长参数是 60，确定一个 bestd，如果敌人离自己很近，则不能出领地，若 bestd 转向的格子是自己的领地则返回 bestd，否则转向自己的领地。经历转向后 storage[‘count’]=1，也就是说接下来都会在领地内直行。

当函数处在 square 状态时，调用 square 函数。这时应当已经调用了 chooseDirection（wander 到 square 的模式转换），chooseDirection 决定了纸卷是顺时针圈地还是逆时针圈地。当纸卷离开领地时，以领地为中心，接受的参数 bc 为象限边长分割四个象限，遍历每个象限获得象限评分：当某个格子是空领地，加 1 分，是对方领地加 5 分，是自己领地不加分，位于场地边界外面减 0.5 分，如果是敌人纸卷所在处就减 10000 分。若纸卷向东走，第一象限得分高，就返回’l’，其他同理，最后将返回值存储为 storage[‘turn’]。在 square 函数中按顺序进行如下判断：

1. 如果下一个格子在边界外，在就立刻返回 storage[‘turn’]；

2. 如果发现沿着前进方向下一个格子是自己的领地，就进行模式转换（到 wander），返回’‘；

3. 如果下一个格子是自己的 band，说明自己的纸卷已经被别人领地包围，将模式转换为’isolate’，返回与圈地的 storage[‘turn’] 相反的转向方向。

在经历了上述判断，才可根据安全距离画方块圈地。

当函数处在 isolate 状态时，调用 isolate 函数。进行如下判断：

1. 如果当前格子是自己的领地，则转换 mode 为 wander，直接运行 wander 函数，返回结果。

2. 否则的话，调用 chooseDirectionisolate 函数。

在 isolate 状态调用的函数功能如下：

chooseDirectionisolate：与 chooseDirection 类似，修改了格子评分的权重，超出边界的格子减 2 分，是自己的领地加 10 分，空领地加 1 分，是敌人领地不加分，每个象限的评分，返回前进方向上高分象限的方向’l’/’r’。

2.3 程序限制

2.3.1 superversion 版

该算法整体已经比较完善，兼有攻击函数、防御函数和启发式圈地算法。在和冠军们的对决中，还是发现了一些极端情况。

极端情况都比较统一，即当自己的领地被切割成两段，敌人包围了自己的纸带，容易被击杀。在程序中也确实发现了这种情况的存在。在 Walk 模式中，会出现无论何方向走，敌人都可能攻击到自己的情况（walk_directions 列表为空），此时没有绝对安全的返回路径。我们采取的策略是，选择 walk_directions_bk 列表中具有最短返回距离的方向，尽快回到自己领地，但依然可能被攻击。在这一点上需要改进。

2.3.2 普通版

在整个算法中，虽然在大多数情况下能保证避免敌人的击杀，但是有一些极端情况仍待

改进：

一. 在我方纸卷准备离开自己的领地时，如果与敌方纸卷距离过近（直线距离 <=5），会在出领地之前调用 chooseDirectionmargineperry() 函数进行回避，但仍旧有被击杀的可能。

二. 同样是 chooseDirectionmargineperry() 函数，由于判断条件不涉及到场地外，在我方领地为一条直线时，有概率因转向而撞墙。

三. 在我方纸卷准备离开自己的领地时，会直接开始判断，如果敌方在下一步占领了四周的领地，会导致周围环境的变化（领地归属），而我们缺少二次判断的机制，不能对这种环境的变化做出反应，因此极容易在这一步之后被击杀。

四. 当我方纸卷被圈入敌方领地时，纸卷的起点变成了一个小方格，对于此种情况，我

们构造了一个 chooseDirectionisolate() 函数来防止纸卷自杀，并向附近残存领地靠近，但遇到进攻性强的算法时，在回程的途中非常容易被撞纸带。

五. 当纸卷的前、左、右都是敌方领地时，会进行一个粗略的判断来防止被杀，但仍有概率被撞纸带或纸卷而死亡。

3 实验结果

3.1 测试数据

我们进行的测试有两种，第一种是利用搭建好的对战平台（solo.py 和 visualize.py ）对不同算法进行对比测试和可视化，其中包括与自己作战，以及我们编写的两个算法之间进行对战；第二种是参与课堂组织的热身赛进行同组之间的 PK。

进行数据测试的目的很明确，在对战中寻找并修复我方算法的漏洞，以及找寻敌方算法的弱点，加以针对。

热身赛环境说明：

硬件配置：（DELL Precision 5820 图形工作站，Xeon W-2135 CPU / 32GB RAM /512GB SSD / NVIDIA GeForce GTX 1080 GPU）

Python 版本：（Python 3.5.2）

Foxtrot1 和 Foxtrot2 两个算法基本相同，只有参数设置上的不同，下面是三次热身赛的结果：

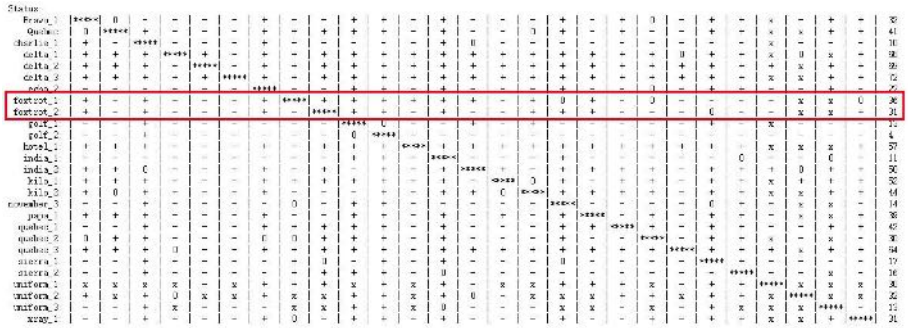
第一次：

代码 1 胜率 42%，代码 2 胜率 42%，两个代码 solo 时代码 2 更有优势。

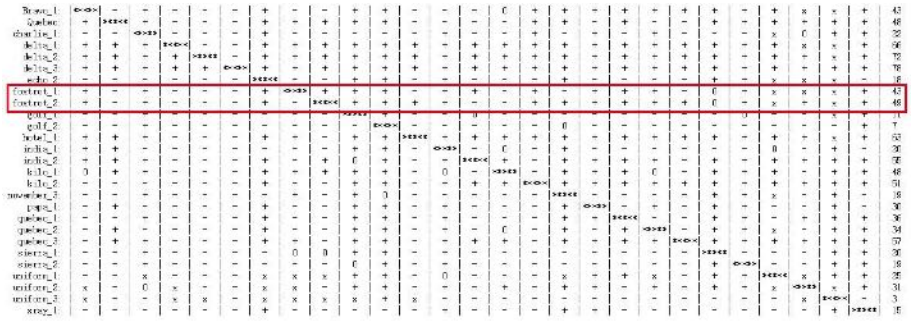
t_n17_GOLF_1:	*****	+	-	+	-	-	+	-	-	-	-	-	-	9
t_n17_GOLF_2:	-	*****	-	-	-	-	+	-	-	-	-	-	-	3
t_n17_delta_1:	+	+	*****	+	+	+	+	+	+	+	+	+	+	36
t_n17_echo_2:	-	+	-	*****	+	-	+	-	-	+	-	-	+	15
t_n17_foxtrot_1:	+	+	-	-	*****	-	+	+	-	0	-	-	+	16
t_n17_foxtrot_2:	+	+	-	+	+	*****	+	+	-	0	-	-	0	20
t_n17_golf:	-	-	-	-	-	-	*****	-	-	-	-	-	-	0
t_n17_hotel_1:	+	+	-	+	-	-	+	*****	-	+	+	+	0	22
t_n17_hotel_2:	+	+	-	+	+	+	+	+	*****	-	+	-	+	27
t_n17_india_1:	+	+	-	-	0	0	+	-	+	*****	-	+	+	20
t_n17_kilo_1:	+	+	-	+	+	+	+	-	-	+	*****	0	+	25
t_n17_kilo_2:	+	+	-	+	+	+	+	-	+	-	0	*****	+	25
t_n17_november_2:	+	+	-	-	-	0	+	0	-	-	-	-	*****	11

第二次：

代码 1 胜率 42.3%，代码 2 胜率 38.5%。



第三次：代码 1 胜率 54%，代码 2 胜率 61.5%。



在参加的三次热身赛中，由于 super 版的代码经常报错，我们提交的是普通版（newwanderer），普通版的获胜场次排位中上，获胜的原因主要是敌方纸卷自杀、撞墙，以及圈地面积更大，很少数情况下出现敌方纸带被撞的情况，这与我们的代码攻击性不强有关。输掉的原因主要是被撞纸带、在极端情况下自杀（见 2.3.2）。

solo 环境说明：

硬件配置：（Intel Core(TM) i7-6500U CPU/ RAM: 8.00GB）

操作系统：（Windows 10）

Python 版本：（Python 3.6）

分先后手，比试五场，测试结果见下表：

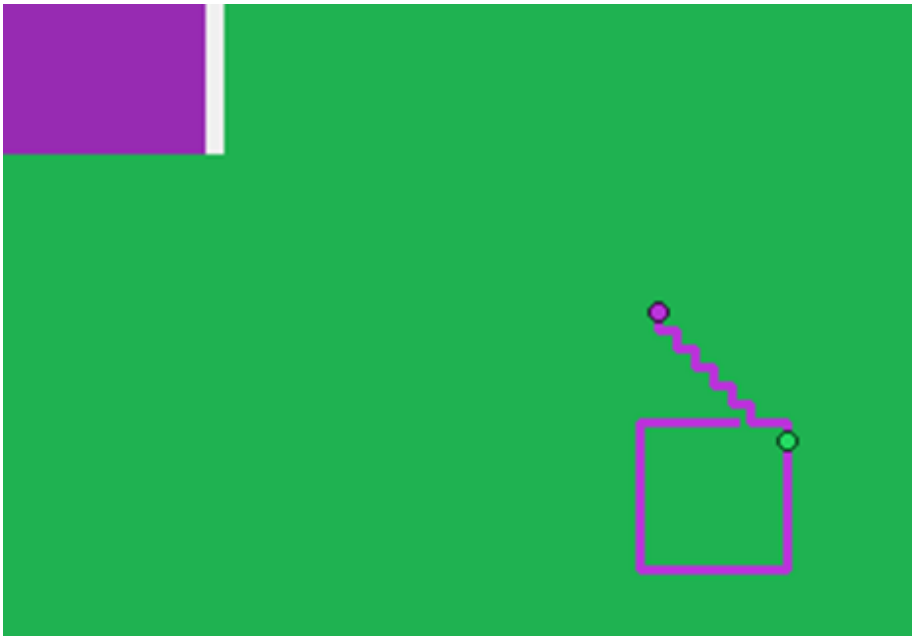
	SuperVersion	NewWander
SuperVersion	2:3	5:0
NewWander	0:5	4:1

下面是四种对局的结果分析：

Newwander 自我对战：先后手的差别比较大，不过数据量太小，无法做出更明确的判断；在比试过程中，有少量自杀的情况（领地被圈，见上图）

Superversion 自我对战：更多的情况是彼此防备，有两局出现一方的‘迷宫’被斩断，然后被撞击的情况（见上图）

N – S 和 S—N ：先后手差别不大，差距太明显，绝大部分情况都是 N 被 S 圈入领地，然后被杀或者自杀（见上图，是被杀的情况），针对这一情况，我们在 newwander 中添加了 chooseDirectionisolate() 函数，但效果不是特别明显。



3.2 结果分析

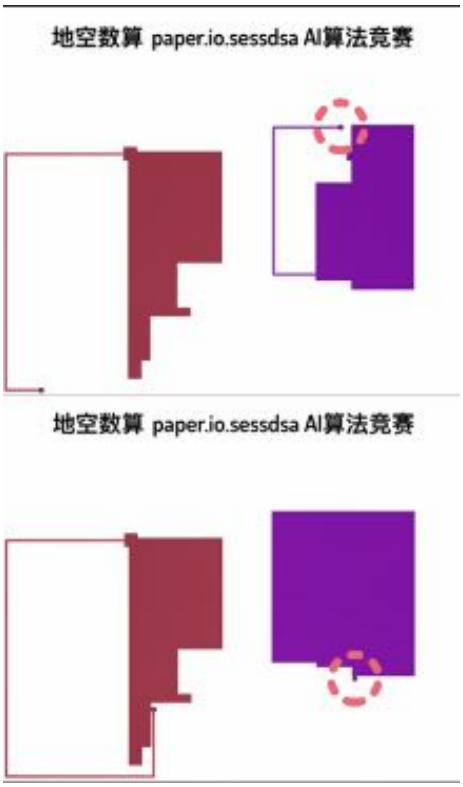
Supervision 版主要耗时来源于 BSF 广搜算法，算法复杂度是 O（N2），N 为场地的长或者宽；普通版最主要的耗时在确定方向的“遍历”操作中，算法复杂度是 O（N2）。实战对弈时，它们的运行时间都在预期中。

supervision 版本策略上主要是攻击、防御、圈地，对于不同格点的信息（属性，距离边界和敌人纸卷等）进行打分，对周围一定区域的格点分数累计，根据启发式函数的优先级进行判断，选择最优的方向。本代码一直在 debug，最后只参加了与冠军的友谊赛（策略和 debug 过程见 1.1.1 部分），最后策略按照我们计划实施。

普通版策略分为“领地内游走和方向选择”wander，“正方形圈地”square、“自救”isolate 和“防御”chooseDirectionmargineperry。对以自己为起点，四个方向上的正方形区域（边长为自己最多能走的安全距离）进行遍历，对其中每个格点进行打分。在几次热身赛实战中，我们发现，当自己领地被对方切割，使得自己不得不在对方领地画纸带时，容易死亡。因此我们增加了“isolate”自救模块，利用 chooseDirection 函数，会尽量向敌人领地外走，并尽量返回自己领地，从而实现“自救”。实战中自救策略成功实现。

3.3 经典战局

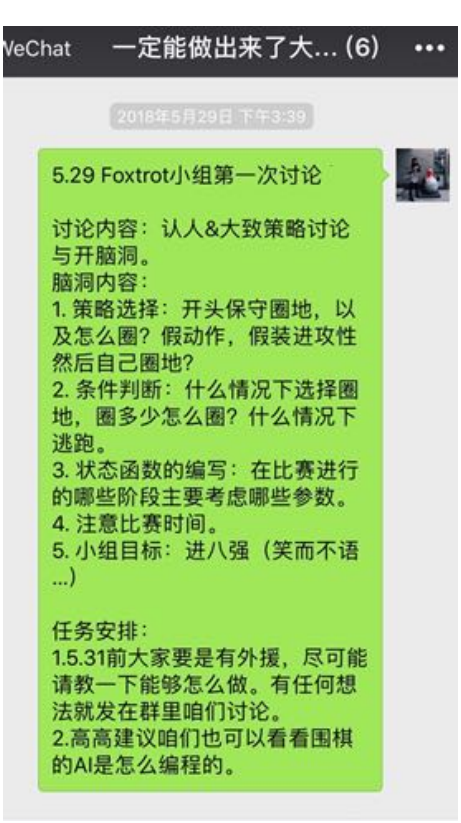
Foxtrot_superversion 在与 delta 的一场比赛以较大的优势获得胜利（5136:903），赛后分析这次比赛主要观察到 delta 算法圈地效率较低，其圈地过程中重复路径过多，如右图所示，在 supervision 完成大范围圈地时，delta 浪费一长段路径在其领地内部。而 supervision 最终的表现中效率要高出很多，在两方未开始相互占领彼此领地时，supervision 以 2495:1394 领先 delta，比 delta 高出约 80%。并且观察 delta 的实际表现，在面对对方领地、小块白地以及大块白地时，delta 会优先选择大块白地，即使这种操作会让其重复路径增加，而 superversion 在圈地时算法将寻找对手领地和空地作为同一优先级，并且在方向上优先选择躲避对方的方向，避免无谓的小范围试探性圈地，很大程度上提高了效率，使得两方在开始近距离相互占领彼此领地时，差距进一步拉开，中段时比分达 3470:1678，此时比 delta 高出 107%。到比赛末段，两个纸带不再近距离纠缠，重新进入大范围圈地状态，此前 supervision 的优势依旧存在，并且此时白地较少，相互占领的都是彼此的领地，supervision 的优势更加明显，最终以 5136:903 的比分赢得比赛。总结而言，supervision 在进攻和防守时做的有效但不纠缠，使得整体效率远高于 delta，逐步积累优势，最终大比分获胜。

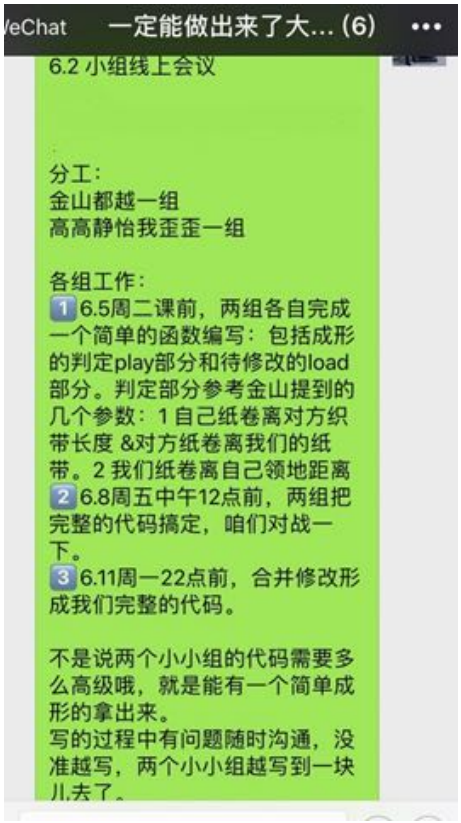


4 实习过程总结

4.1 分工与合作

从组队完成到最后提交代码，我们 n17_Foxtrot 一共开过四次线下和两次线上组会。这一部分我会先总结每次开会内容记录与合照，在最后总结小组分工与合作交流方式。





第一次线下和线上讨论没有照片，主要内容是：策略选择和分组编写代码。当时对代码的认识 too young too naïve，主要碰到的问题有：策略很多但是实现困难，主要使用的算法与数据结构模糊。所以为了解决问题我们各自寻找了外援，外援提到我们可以参考的方法有：蒙特卡洛树等。

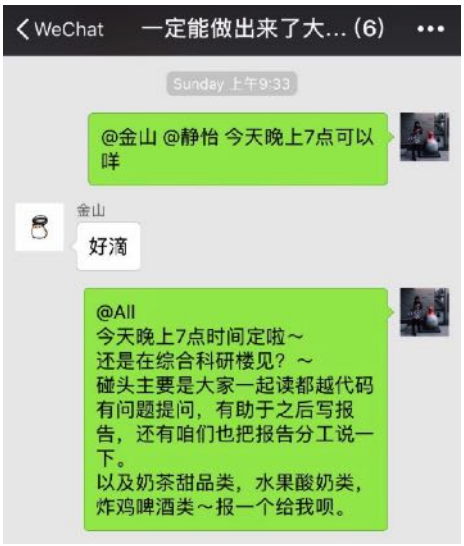
6.5 第二次线下小组会议，这一次小组有了重大进展，决定继续改进。

6-5 第二次线下小组会议 坐标：理教

6.7 的组会，我们意外发现综合科研楼休息室光线明亮、设施齐全、环境整洁、没有 WiFi。这一次的组会一直持续到晚上 11 点 30，主要内容是改进 AI 的自杀和圈地效率。

6.10 赛前最后一次小组线下会议，依旧那个教室，这次会议主要目的是大家一起读代码、改 bug、制定报告计划、喝快乐肥宅水。我们当时有两个代码，一个代码已经比较成熟，属于稳妥打发，参加了前两次的热身赛，一个代码刚开发，经常会报错，但是胜率非常高。计划最后 2 天一边改进后一个代码，一边完善前一个代码。

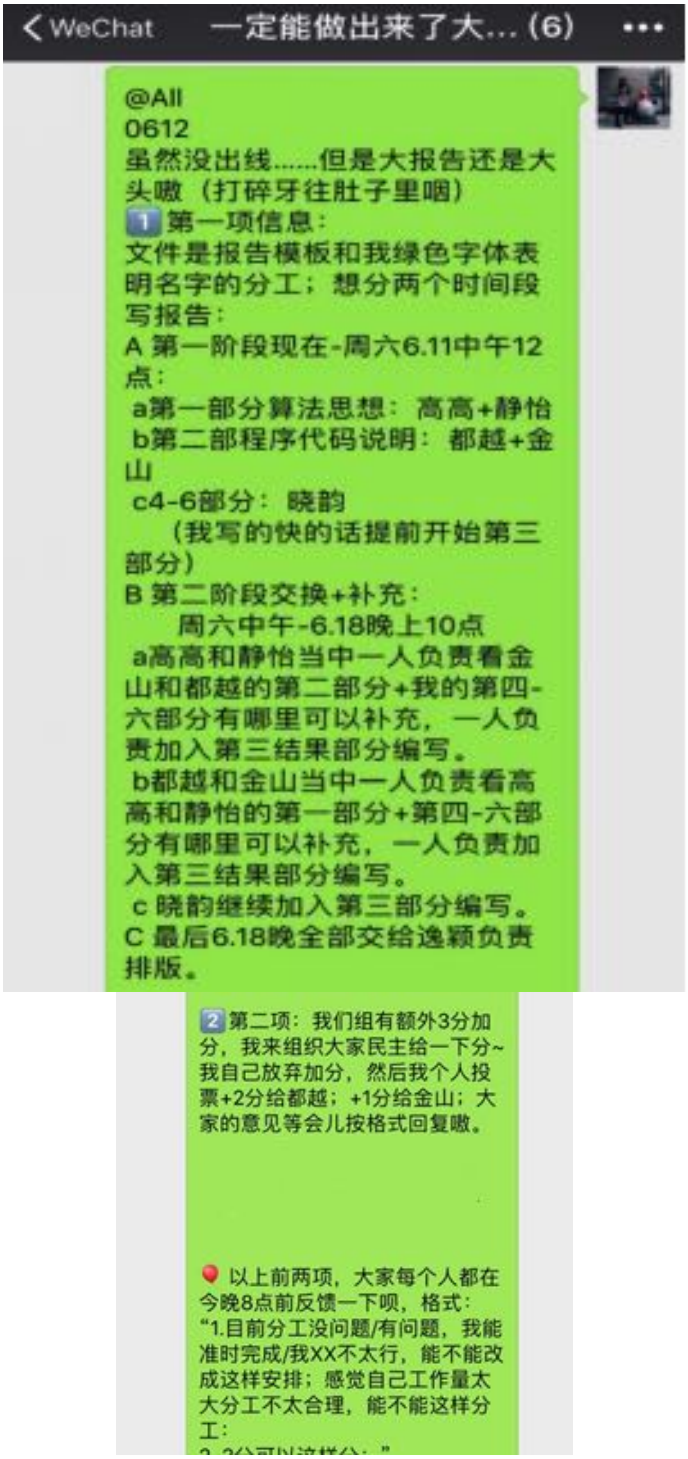
6-10 第四次线下小组会议
坐标：依旧是综合科研楼



6.12 比赛当天！没有合照 =_=，很可惜第二个代码在最后时刻没能完善成果，交了原来的版本 当天比赛后的线下会议主要关于三部分内容：小组报告分工与 ddl，小组加分商定、小组批评反思。

我们的分工总的来说就是：都越主负责代码的发开编写、金山负责和都越一起修改完善、静怡、晓韵和诗铠负责测试部分。晓韵另外负责会议组织记录、诗铠负责拍照。

合作交流方式主要是微信，大家都比较放得开，想到什么新的思路或者遇到新的问题都会在微信上进行交流。


WeChat
一定能做出来了大...(6)
@All
0612
虽然没出线.....但是大报告还是大头嗷（打碎牙往肚子里咽）
1 第一项信息：
文件是报告模板和我绿色字体表明名字的分工；想分两个时间段写报告：
A 第一阶段现在-周六6.11中午12点：
a第一部分算法思想：高高+静怡
b第二部程序代码说明：都越+金山
c4-6部分：晓韵
（我写的快的话提前开始第三部分）
B 第二阶段交换+补充：
周六中午-6.18晚上10点
a高高和静怡当中一人负责看金山和都越的第二部分+我的第四-六部分有哪里可以补充，一人负责加入第三结果部分编写。
b都越和金山当中一人负责看高高和静怡的第一部分+第四-六部分有哪里可以补充，一人负责加入第三结果部分编写。
c 晓韵继续加入第三部分编写。
C 最后6.18晚全部交给逸颖负责排版。
2 第二项：我们组有额外3分加分，我来组织大家民主给一下分~我自己放弃加分，然后我个人投票+2分给都越；+1分给金山；大家的意见等会儿按格式回复嗷。
以上前两项，大家每个人都在今晚8点前反馈一下呗，格式：
“1.目前分工没问题/有问题，我能准时完成/我XX不太行，能不能改成这样安排；感觉自己工作量太大分工不太合理，能不能这样分工：

4.2 经验与教训

工作经验最主要的就是多多沟通、互相了解、互相理解吧。从小组刚组队到后期代码开发完善、写报告，小组合作的侧重点也多有不同。

比较得意的部分就是整个过程中我们始终重视了以上侧重点，并且除开数算队友以外，也成为了比较好的朋友啦。刚组队时，四个大四学姐和两个大二的学弟大家对彼此的代码能力都不是非常了解，我们的熟悉方式是从一起读代码开始的。因为临近期末，大二的学习压力比较大，所以我们在时间安排上希望大家不熬夜，笃定地搞定大作业。很高兴最后的时间进度比较效率符合计划。希望改进的部分，可能是最后都越和金山的工作负担还是没能减轻很多，希望小组工作分配在与能力匹配的同时能够更平均吧。

4.3 建议与设想

这次的实习作业很有启发性和趣味性，各种基础设施代码不断在改进完善。可以改进的部分有：竞赛时进行在线直播，需要采访的话场外连线？不过这只是增加趣味性啦。

给明年学弟妹的寄语：不要怕不要方。有编程思路勤练手，把思路实现为 python 语言。

大胆设想：和 VR 结合使得人产生身为“纸卷”在一个巨大平面中圈地的视角。从第一视角玩儿，和我们现在等于俯视这个游戏场地的感觉应该会有不同

5 致谢

感谢都越和金山实力 carry，if 再多又怎么样，你们最棒了。

感谢清华大学计算机系的 csy 同学建议用蒙特卡洛算法，虽然我们没学会；感谢 csy 发来的一系列文献资料，虽然我们没读懂。

感谢信科的 gjy 同学友情指导，虽然我们没来得及把第二个代码的 bug de 完，然友谊赛神仙打架赛高！

感谢静怡孜孜不倦想出改进措施！感谢高高每次都提醒咱们拍照！感谢组长的珠荟啡！

感谢家属朱朱给我们照相！

感谢技术组孜孜不倦地优化基础设施！

6 参考文献

[1] 蒙特卡洛https://blog.csdn.net/z84616995z/article/details/50915952

[2] 《蒙特卡洛博弈方法》清华大学人工智能导论课件

第七章　N17_Golf 报告

姜峰 * 朱洁薇 张晓宇

摘要：本报告主要介绍 Golf 小组在数据结构与算法课程学习中的整体实习过程。我们组为纸袋圈地竞赛开发的算法主要思想即两个关键词：搜索和学习。最终由于强化学习算法效果不佳，选用常规策略算法参赛。常规策略算法依靠的数据结构是图 graph、线性表 list 和字典 dir，实时记录“攻击距离”、“安全距离”等数据实现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。主要算法为广度优先搜索，通过记录分析敌我双方参赛回合中的各项数据，包括纸带、纸带头、领地等，以此为基础来模拟对方的对战策略与己方的迎战策略，再通过对每一时刻当前我方被攻击、我方安全和攻击敌方的可能性进行比较，搜索得到最佳对战方案。整体来看对战过程，算法的出其不意使得其在与白痴算法的对战中均能取得极高的胜率，而且往往实现秒杀；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；但由于算法本身的缺陷，在最后的最终竞赛中未能取得胜利，遗憾无缘小组出线。强化学习目的在于训练出能对给定状态打分的神经网络，然后在每一步都选择使得下一步达到的状态的打分最高的行动。通过神经网络的自我博弈不断更新相关的参数，然后在实际的操作中我们发现神经网络太难收敛，最终训练效果并不好。

关键字：列表广度优先搜索强化学习

1 算法思想

1.1 总体思路

针对本次纸袋圈地竞赛，小组设计的算法思想主要包含两个关键词，其一是搜索，其二是学习。

1.1.1 常规策略算法

我们小组常规策略算法主要使用的数据结构是线性表 list 和字典 dir，主要应用为：

list：记录所有点距离纸带头的距离，更新 list 存储每一次探索的出发点；记录我方纸袋头与所有我方占领区域的距离，更新 list 记录最小值；

dir：存储是否可以“偷鸡”，根据 list 中存储的当前值判断下一步的策略；存储当前是在围领地中的哪个阶段（领地内、向外走、掉头、往我方领地里走）等。

总的来说，算法的最终实现都有赖于 list 的强大与便捷。

常规策略算法的主体是广度优先搜索算法。主要应用为：在选择对战方案的过程中，通过计算一方纸带头到另一方领地的距离，来估计敌方攻击我方需要的时间；通过计算自己纸袋头与所有自己占领区域的距离中最小值作为纸袋头距离自己占领区域的距离；通过比较沿着当前路径走下去，使得全边界相连所需要的距离与敌方攻击我方的距离，判断是否能实现“偷鸡”。

常规算法采用的主要策略可以概括为“偷鸡”，大体解释如下：

（1）在无干扰情形下，纸带头沿边界移动一周回到领地，将全部格点划为我方领地；

（2）当存在被敌方攻击的可能性时，估计　敌方攻击我方需要的时间；　我方纸带头回到我方领地的时间；　沿着当前路径走使得全边界相连所需要的时间，并进行比较。若 <= ，则不掉头，直接向前；若 <= , 则掉头，围一个两行的区域回到领地。

该算法实现依赖的主要函数如下：

（1）_headToAttacked(meOrEnemy, stat)

该函数用来计算一方纸带头到另一方占领区域的距离，来估计敌方攻击我方需要的时间。

（2）_headToHome(meOrEnemy, stat, whetherNeedPath)

该函数用来计算一方纸带头与所有自己领地的距离中最小值，来估计己方纸带头返回领地的时间。

（3）_checkWhetherCanTouJi(stat,storage)

该函数用来判断能否“偷鸡”。如果沿着当前路径走使得全边界相连所需要的距离小于敌方攻击我方的距离，就不掉头，直接向前。

在算法思路设计阶段，经过多次组会讨论，我们的思路可概括为“攻守兼备”，可分为攻击和圈地两部分。

（1）圈地部分：在保证不被敌方攻击的前提下尽可能扩大圈地面积。在走出领地后，若无被攻击的风险（被攻击距离 > 我方安全距离），则继续直行；若可能被攻击（被攻击距离 < 我方安全距离），则朝远离敌方的方向转弯。之后若无被攻击的风险，则继续直行扩大可能的圈地面积，若可能被攻击，则选择转弯返回领地。再之后若继续存在被攻击的可能，则保持直行返回领地；若无被攻击的风险，则继续拐向领地外扩大可能的圈地面积……以此类推。

（2）攻击部分：在判断攻击距离小于敌方安全距离时选择攻击敌方。

该思路的重点在于计算下列几种距离：

（1）被攻击距离：对方纸带头到我方纸带的最小距离

（2）我方安全距离：我方纸带头回到我方领地的最小距离

（3）攻击距离：我方纸带头到对方纸带的最小距离

（4）敌方安全距离：对方纸带头回到对方领地的最小距离

我们认为这是一个比较完善的思路，但在具体实现中遇到了很多问题，具体如下：

（1）四种距离的计算，包括在领地形状不规则时如何计算最小距离，是否需要得到领地边界，如何在计算距离时把不碰到己方纸带考虑进来等；

（2）纸带返回领地后如何选择下一个出口；

最终由于两位组员编程能力有限和强化学习算法效果不佳，我们没能实现这个算法，而是退而求其次，这就有了最后的“偷鸡”算法。

“偷鸡”算法实现的过程大致有两个阶段：（此为先手局情况，后手局将逆时针改为顺时针）

（1）在保证不被攻击的前提下，沿逆时针绕场一周并阶段性返回己方领地来圈地，最终回到己方领地围住全场；

（2）改进后首先朝着距离边界最近的地方走，围一个两行的领地再回来；然后先到达上边界后围一个两行的领地返回，再到达下边界后围一个两行的领地返回，再出发向下边界运动，沿逆时针绕场一周的方向并阶段性返回己方领地来圈地，最终回到己方领地围住全场，并每次判断能否“偷鸡”，即沿着当前路径走下去，使得全边界相连所需要的距离小于敌方攻击我的距离，若可以“偷鸡”则沿当前路径继续运动，否则掉头返回领地。

1.1.2 强化学习部分

强化学习的部分总体分为两部分，一是模型——一个输入对战状态，输出的预期回报的神经网络的训练，二是根据训练好的模型给出下一步的行动。

模型训练主要分为以下几个部分：

1、神经网络的搭建，对应代码 ___build_net(self)，网络的结构如下：

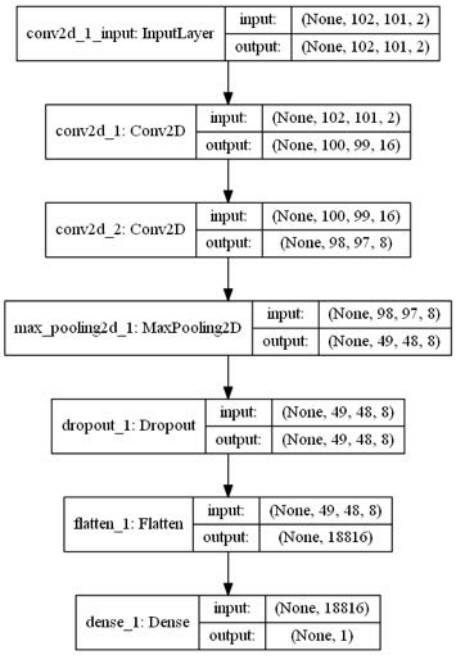


图 1 神经网络结构

（1）生成对战记录并打分，作为训练集，主要有以下几步：

(1) 生成环境，用于神经网络的自我博弈，得到不同可能的状态，对应的类为environment。

(2) 选择下一步，对应的函数为 choose_action(slef,env)：

根据当前状态，计算不同选择 (向左、向右、或向前) 获得的三个后续状态，利用规则判别得到相应的即刻回报，利用当前未进化完全的神经网络得到预计回报。相加，选择回报最大的那一步作为下一步返回。

(3) 生成训练数据的存储，对应函数为 store_transition(self,state,reward,state_)

存储的内容包括：当前状态，基于规则的打分，利用神经网络对最优步骤后的状态的打分，总分。其中基于规则的打分固定：如果死亡，打 -10000 分；如果杀死对方，打 10000 分，否则得分为自己已经圈出地盘的面积。利用神经网络的打分不固定，每当更新一次网络的参数更新一次。

（2）模型参数的优化，对应的代码部分是 learn(self), 参数优化使用的训练集为刚生成的 700 个数据和从所有之前的数据中随机选择 800 个，以希望在当前对战和历史对战之间达到一种平衡。采用随机梯度下降，速度快，避免落入鞍点的问题。

2、将训练好的模型用于实际应用。首先载入训练好的神经网络，对于可选的三个行动：向左、向右、不动分别生成行动后的三个局面，利用规则判别得到相应的即刻回报，利用神经网络得到预计回报。相加，选择回报最大的那一步作为输出。

1.2 算法流程图

主函数

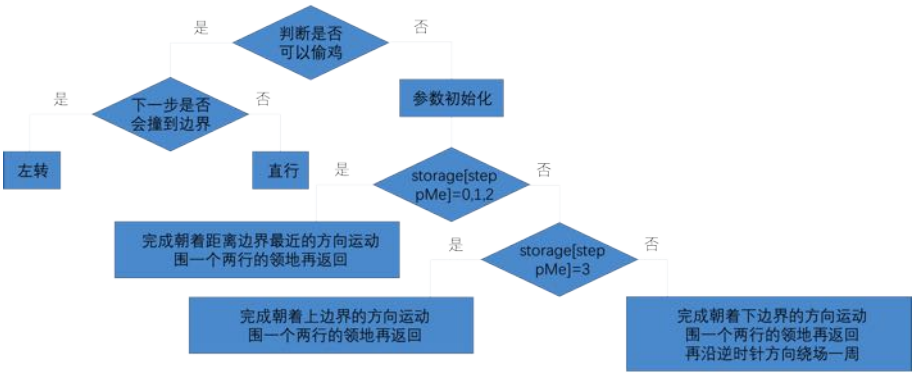


图 2 主函数算法流程图

_headToAttacked(meOrEnemy, stat)

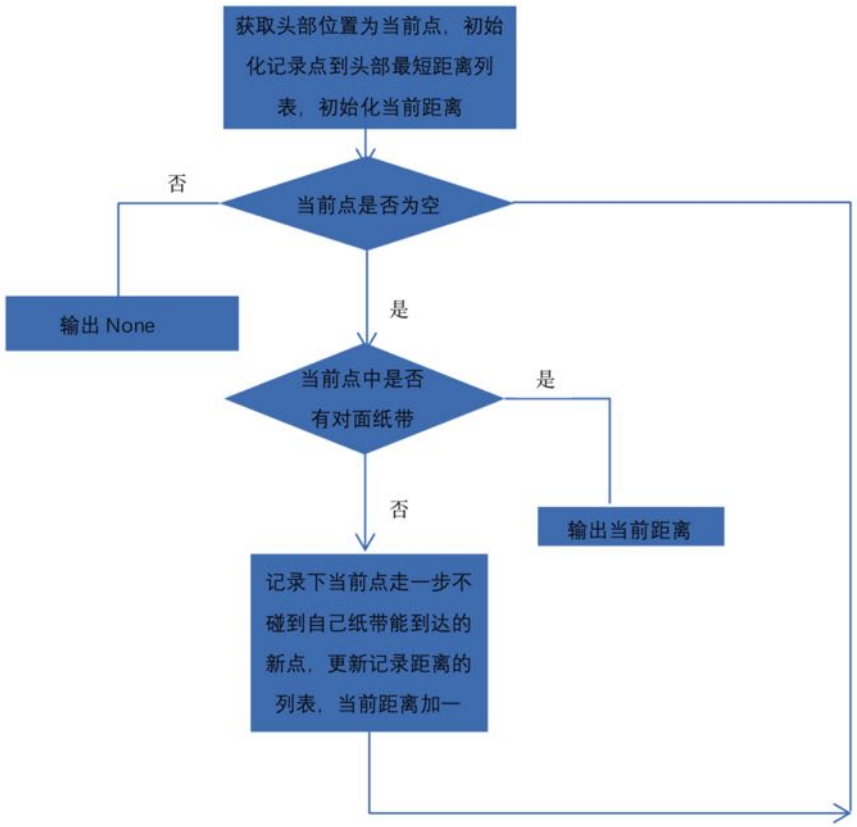


图 3 headToAttacked 函数流程图

_headToHome(meOrEnemy, stat, whetherNeedPath)

判断 fields 中所有点到 head 的距离中最小值，过于简单，无需流程图。

_checkWhetherCanTouJi(stat,storage)

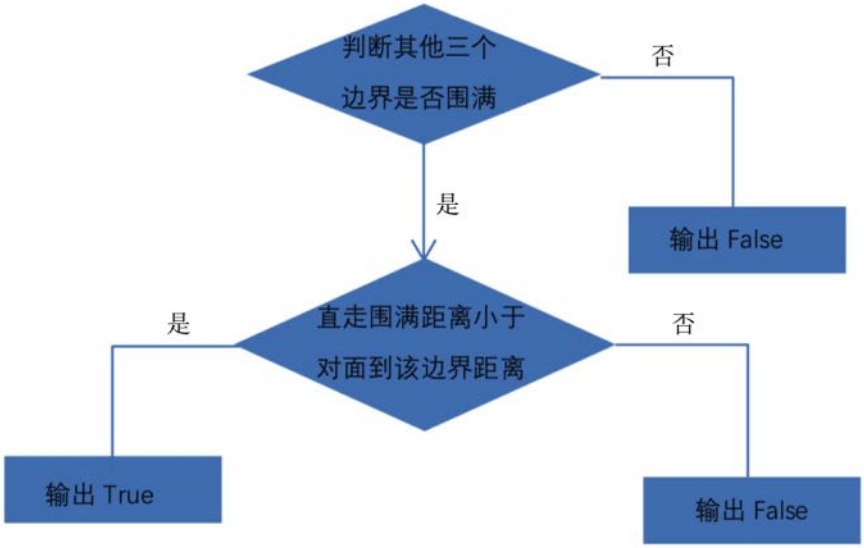


图 4 “是否能偷鸡”函数流程图

1.3 算法运行时间复杂度分析

我们组常规策略算法运行时间主要构成在于 _headToAttacked 函数。该函数采用宽度优先搜索，用 disMap 用来存储所有点距离头的距离，遍历 fields 内所有的点寻找距离表头一个单位距离的格点，故算法复杂度为 O(width*height)，width 和 height 分别为平台的宽和高。强化学习策略的时间复杂度为 $O(\sum_{l=1}^{2} M_l^2 K_l^2 C_{l-1} C_l)$ 其中 M_l 为第 l 卷积层的卷积核输出特征图的边长，K_l 为第 l 个卷积核的边长，C_{l-1},C_l 分别为第 l 层的输入、输出的通道数。

2 程序代码说明

2.1 数据结构说明

1、_headToAttacked(meOrEnemy, stat)

该函数主要采用了图的数据结构，用来计算一方纸带头到另一方占领区域的距离，来估计敌方攻击我方需要的时间。

该函数采用宽度优先搜索，disMap 用来存储所有点距离头的距离，初始化一个很大值在满足不碰撞到自己纸带的条件下，先找到距离表头一个单位距离的格点，存在一个 list 里面，并更新 disMap。接着从这个 list 中所有点出发，找距离该点一个单位距离且不是自己纸带，且 disMap 的距离大于等于 2 的所有点，存为下一次探索的初始点，并更新 disMap。依次下去，直到寻到的某个点是另一方的纸带区域。

2、_headToHome(meOrEnemy, stat, whetherNeedPath)

该函数主要采用了列表的数据结构，由于策略的路径较为简单，不存在回到自己区域碰到自己纸袋的情况。所以通过计算自己纸带头与所有己方领地的距离中最小值 minDis 来作为己方返回领地的“安全距离”。

3、主体函数

主体函数主要采用了字典的数据结构，用来存储当前我方纸带头所处的几个状态值，具体如下：

（1）’steppMe’用来存储当前运动的状态：0 表示朝最近边界运动；1 表示在最近边界处掉头；2 表示从最近边界处返回领地；3 表示向上边界运动；4 表示向下边界运动。

（2）’tempStep’用来存储当前是在围区域中的哪个阶段：0 表示在领地内；1 表示向外走；2 表示掉头；3 表示往自己领地里走。

（3）’ox’,’ oy’用来存储从领地出发的起始点的横纵坐标。

（4）’Touji’用来存储 _checkWhetherCanTouJi 的判断结果，即是否可以“偷鸡”

2.2 函数说明

1、_headToAttacked(meOrEnemy, stat)

该函数的功能是用来计算一方纸带头到另一方占领区域的距离，来估计敌方攻击我方需要的时间。

disMap：用来存储所有点距离头的距离

tempDis：用来存储每一次探索的初始点

reach：用来存储搜索到的点是否是另一方的纸带区域

该函数采用宽度优先搜索，初始化一个很大值在满足不碰撞到自己纸带的条件下，先找到距离表头一个单位距离的格点，存在 tempDis 里面，并更新 disMap。接着从 tempDis 中所有点出发，找距离该点一个单位距离且不是自己纸带，且 disMap 的距离大于等于 2 的所有点，存为下一次探索的初始点，并更新 disMap。依次下去，直到寻到的某个点是另一方的纸带区域，reach=TRUE。返回出发点到搜索终止点的距离。

```
if bands[headX][headY] == enemyId:
    return 0
while not reach and nowDis < width + high:
    tempDis = []
    for coord in tempNowDis:
        nx = coord[0]
        ny = coord[1]
        if nx + 1 < width and bands[nx + 1][ny] != headId and disMap[nx + 1][ny] == width + high:
            disMap[nx + 1][ny] = nowDis + 1
            tempDis.append([nx + 1, ny])
            if bands[nx + 1][ny] == enemyId:
                reach = True
        if nx - 1 >= 0 and bands[nx - 1][ny] != headId and disMap[nx - 1][ny] == width + high:
            disMap[nx - 1][ny] = nowDis + 1
            tempDis.append([nx - 1, ny])
            if bands[nx - 1][ny] == enemyId:
                reach = True
        if ny + 1 < high and bands[nx][ny + 1] != headId and disMap[nx][ny + 1] == width + high:
            disMap[nx][ny + 1] = nowDis + 1
            tempDis.append([nx, ny + 1])
            if bands[nx][ny + 1] == enemyId:
                reach = True
        if ny - 1 >= 0 and bands[nx][ny - 1] != headId and disMap[nx][ny - 1] == width + high:
            disMap[nx][ny - 1] = nowDis + 1
            tempDis.append([nx, ny - 1])
            if bands[nx][ny - 1] == enemyId:
                reach = True
    nowDis += 1
    tempNowDis = tempDis
return nowDis
```

图 5 headToAttacked 函数代码

2、_headToHome(meOrEnemy, stat, whetherNeedPath)

该函数的功能是通过计算自己纸带头与所有自己占领区域的距离中最小值来作为纸带头距离自己占领区域的距离。

minDis：用来存储自己纸带头与所有自己占领区域的距离，不断更新存储最小值。

coord：用来存储己方领地距离己方纸带头最近的点的坐标。

whetherNeedPath：返回是否需要返回一个回到己方领地的规划路径。

```
def _headToHome(meOrEnemy, stat, whetherNeedPath):
    # meOrEnemy :'me' or 'enemy'
    #stat:stat
    #whetherNeedPath:True or False
    now = stat['log'][-1]
    headId = now[meOrEnemy]['id']
    headX = now[meOrEnemy]['x']
    headY = now[meOrEnemy]['y']
    width, high = stat['size']
    fields = now['fields']
    minDis = width+high

    for x in range(width):
        for y in range(high):
            if fields[x][y]==headId:
                dis = abs(x-headX)+abs(y-headY)
                if dis<minDis:
                    minDis = dis
                    coord = [x,y]
    if not whetherNeedPath:
        return minDis
    else:
        for x in range(min(coord[0],headX),max(coord[0],headX)+1):
            for y in range(min(coord[1],headY),max(coord[1],headY)+1):
                fields[x][y]=headId
        return minDis, fields
```

图 6 headToHome 函数代码

3、_checkWhetherCanTouJi(stat,storage)

该函数的功能是判断是否可以“偷鸡”，即如果沿着当前路径走下去，使得全边界相连所需要的距离小于敌方攻击我方的距离，就不掉头，直接向前

connected：用来存储使得全边界相连的判断条件。以先手局为例，分别需要判断左边界、下边界、右边界以及上边界位于纸带头右边的格点是否都已属于己方领地。一旦有格点不属于，则返回 FALSE。

attackDis：用来计算敌方攻击我方的距离。在 connected 为 TRUE 的前提下，欲使得全边界相连，我方仍需运动 x 距离，若 attackDis+3<x，则判断可以“偷鸡”，将 storage 字典中’Touji’的值改为 TRUE。

```
def _checkWhetherCanTouJi(stat,storage):
    connected = True
    for i in range(width):
        if fields[i][0]!=meId:
            connected = False
    if connected:
        for i in range(high):
            if fields[0][i]!=meId or fields[width-1][i]!=meId:
                connected=False
    if connected:
        for i in range(x,width):
            if fields[i][high-1]!=meId:
                connected = False
    if connected:
        attackDis = min([abs(now['enemy']['y'])+abs(now['enemy']['x']-xx) for xx in range(x+1)])
        if attackDis+3 < x:
            storage['TouJi']=True
```

图 7 checkWhetherCanTouJi 函数代码

4、主体函数

主体函数的功能大致如下（以先手局为例）：

（1）首先判断是否可以“偷鸡”，若可以偷鸡，如果下一步会碰到边界就向左拐；如果下一步不会碰到边界就继续直行；

```
if not 'TouJi' in storage:
    storage['TouJi'] = False
if storage['TouJi']:
    directions = ((1, 0), (0, 1), (-1, 0), (0, -1))
    nextX = x + directions[dir][0]
    nextY = y + directions[dir][1]
    if nextX < 0 or nextY < 0 or nextX > width - 1 or nextY > high - 1:
        return 'l'
    else:
        return 'd'
```

图 8 主体函数部分代码

（2）若不可以“偷鸡”，设置‘Touji’初值为 FALSE。若未记录初始点’ox’,’ oy’，设置当前点为初始点。若未记录’steppMe’，设置初值为 0。若未记录’tempStep’，设置

初值为 0。

```
if not 'ox' in storage:
    storage['ox'] = x
if not 'oy' in storage:
    storage['oy'] = y
distanceWithWalls = [width - 1 - x, high - 1 - y, x, y]
if not 'steppMe' in storage:
    storage['steppMe'] = 0
if not 'tempStep' in storage:
    storage['tempStep']=0
```

图 9 主体函数部分代码

（3）storage[’ steppMe’]=0,1,2 时完成朝着距离边界最近的方向运动，围一个两行的领地再返回。

```
if storage['steppMe'] == 0:
    if min(distanceWithWalls) != 0:
        deltaDir = (distanceWithWalls.index(min(distanceWithWalls)) - dir) % 4
        if deltaDir == 0:
            return 'd'
        elif deltaDir == 1:
            return 'r'
        else:
            return 'l'
    else:
        storage['steppMe'] = 1
        return 'l'
elif storage['steppMe'] == 1:
    storage['steppMe'] = 2
    return 'l'
elif storage['steppMe'] == 2:
    ox = storage['ox']
    oy = storage['oy']
    if abs(ox - x) <= 1 and abs(oy - y) <= 1:
        storage['steppMe'] = 3
        return 'd'
    else:
        return 'd'
```

图 10 主体函数部分代码

（4）storage[’steppMe’]=3 时完成朝着上边界的方向运动，围一个两行的领地再返回。其中由 tempStep 用来存储当前是在围区域中的哪个阶段：0 表示在领地内；1 表示向外走；2 表示掉头；3 表示往自己领地里走。直到 storage[’tempStep’]=3 后回到领地。在 storage[’tempStep’]=0 时，用 headToAttacked 和 headToHome 存储 _headToAttacked 和 _headToHome 的返回值，并在每一步都比较 headToAttacked 和 headToHome。若不会碰到边界且 headToHome+3<headToAttacked，则继续向前，否则掉头并返回领地。

```
elif storage['steppMe']==3:
    if min(distanceWithWalls)!=0 and storage['tempStep']==0:
        deltaDir = (distanceWithWalls.index(min(distanceWithWalls))-dir)%4
        if deltaDir==0:
            return 'd'
        elif deltaDir==1:
            return 'r'
        else:
            return 'l'
    else:
        directions = ((1, 0), (0, 1), (-1, 0), (0, -1))
        nextX = x+directions[dir][0]
        nextY = y+directions[dir][1]
        if storage['tempStep']==0:
            headToAttacked = _headToAttacked('enemy',stat)
            headToHome=_headToHome('me', stat, False)
            if not (nextX<0 or nextY<0 or nextX>width-1 or nextY>high-1) and headToHome+3<headToAttacked:
                return 'd'
            else:
                if fields[x][y]!=meId:
                    storage['tempStep']=1
                return 'r'
        elif storage['tempStep']==1:
            storage['tempStep']-2
            return 'd'
        elif storage['tempStep'] == 2:
            storage['tempStep'] = 3
            return 'r'
        else:
            if fields[x][y]!=meId:
                return 'd'
            else:
                storage['tempStep']=0
                storage['steppMe']=4
                return 'r'
```

图 11 主体函数部分代码

（5）storage[’steppMe’]=4 时首先完成朝着下边界的方向运动，围一个两行的领地再返回。再沿逆时针方向绕场一周，若不会碰到边界且 headToHome+3<headToAttacked，则继续向前，若下一步会碰到边界，则左拐；若可能被攻击，则左拐掉头并返回领地。

```
else:
    if min(distanceWithWalls)!=0 and storage['tempStep']==0:
        deltaDir = (distanceWithWalls.index(min(distanceWithWalls))-dir)%4
        if deltaDir==0:
            return 'd'
        elif deltaDir==1:
            return 'r'
        else:
            return 'l'
    else:
        directions = ((1, 0), (0, 1), (-1, 0), (0, -1))
        nextX = x+directions[dir][0]
        nextY = y+directions[dir][1]
        if storage['tempStep']==0:
            if y==high-1 and x>1:
                _checkWhetherCanTouJi(stat,storage)
            headToAttacked = _headToAttacked('enemy',stat)
            headToHome=_headToHome('me', stat, False)
            if not (nextX<0 or nextY<0 or nextX>width-1 or nextY>high-1) and headToHome+3<headToAttacked:
                return 'd'
            else:
                if fields[x][y]!=meId:
                    storage['tempStep']=1
                return 'l'
        elif storage['tempStep']==1:
            storage['tempStep']=2
            return 'd'
        elif storage['tempStep'] == 2:
            storage['tempStep'] = 3
            return 'l'
        else:
            if fields[x][y]!=meId:
                return 'd'
            else:
                storage['tempStep']=0
                return 'l'
```

图 12 主体函数部分代码

2.3 程序限制

由于我们的常规策略算法思路较为清奇，在面对白痴算法时能迅速取得胜利，但在面对其他组的 AI 时局限性就非常明显了。其中最大的问题有两点：一是如果己方圈出的领地被对方截断，即回到原点后无法连成完整的一圈，不仅耗费了大量时间和回合数，而且圈地面积也非常小，这将我方置于非常不利的境地。二是我们的算法没有主动攻击对方的策略，这就使得我们在局面上非常被动。另外，我们的“偷鸡”算法的赢面就在于出其不意，而我们在热身赛中就派出了该算法参赛，其他小组在观看比赛回放后在自己的算法中也加入了“偷鸡算法”作为他们策略的一部分或者对我们设计了针对性的策略。

强化学习的策略部分，由于没有足够的时间和运气调整到合适超参、选择正确的给当前状态打分的方式（事实上这两者没有固定的标准，只能通过优化结果来慢慢尝试），也没

有足够的对战局数让参数尽可能优化（事实上在实际过程中最后损失函数下降越来越慢，这部分的问题相对于参数的选取并不算最严重），使得我们最后在实际网络训练的时候损失函数并没有降到合适的值，使得这个策略并没有取得很好的结果。

3 实验结果

3.1 测试数据

实验环境说明：

硬件配置：（CPU/内存）Xeon W-2135 CPU / 32GB RAM /

操作系统：（名称/版本）ubuntu 18.04

Python 版本：（版本号）python 3.5.2

本组算法在与白痴算法对战中保持全胜。

本组算法在最后一次热身赛中排名 N17 组第 15 名，成绩如下：

代码编号	获胜场数
Quebec_1	42
Quebec	41
papa_1	39
foxtrot_1	36
Bravo_1	32
uniform_2	32
foxtrot_2	31
xray_1	31
Quebec_2	30
uniform_1	30
echo_2	22
sierra_1	17
sierra_2	16
November_3	14
golf_1	13
uniform_3	13
india_1	11
Charlie_1	10
golf_2	4

3.2 结果分析

3.2.1 测试阶段一：和白痴算法对战

我们的常规策略算法在于白痴算法对战的过程中表现较好，能迅速绕场一周围住全场，取得 100% 圈地的胜利，且未出现被对方领地截断的现象。

下图为我方与白痴算法的对战，我方先手蓝色。

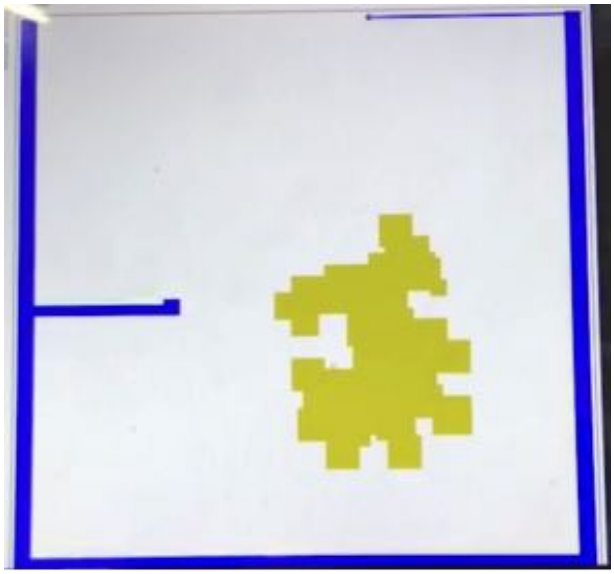


图 13 我方与白痴算法对战界面

3.2.2 测试阶段二：热身赛

我们组共参加了两次热身赛，最终的“偷鸡”算法参与了最后一次热身赛。

我们在热身赛中表现并不乐观，赢的场次很少，获胜的主要原因在于我方在未被对手攻击的情况下围住了全场。

下图为我方与 november 的对战记录，我方先手绿色，下一步我方即将围住全场获胜。

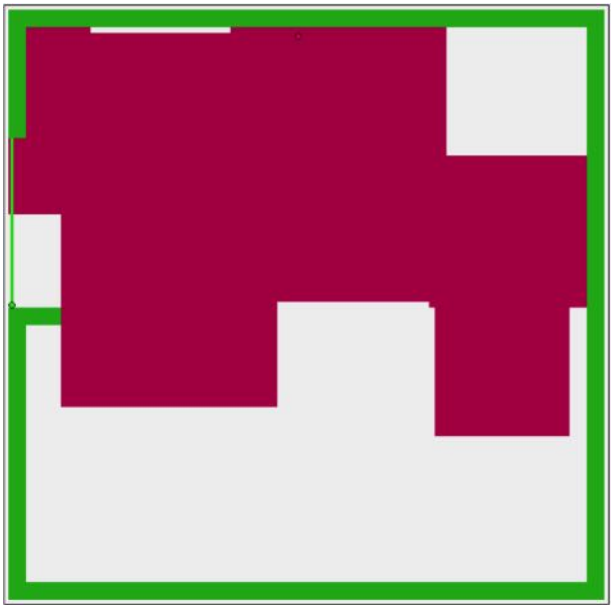


图 14 我方与 November 组对战界面

我方失败的主要情形如下：

1、对手撞到我方纸带获胜。由于我方在离开领地后纸带较长，且圈出的领地较窄，无法在短时间内快速回到领地，故被对手抓住机会撞击纸带获胜。

下图为我方与 delta 的对战记录，我方后手绿色。

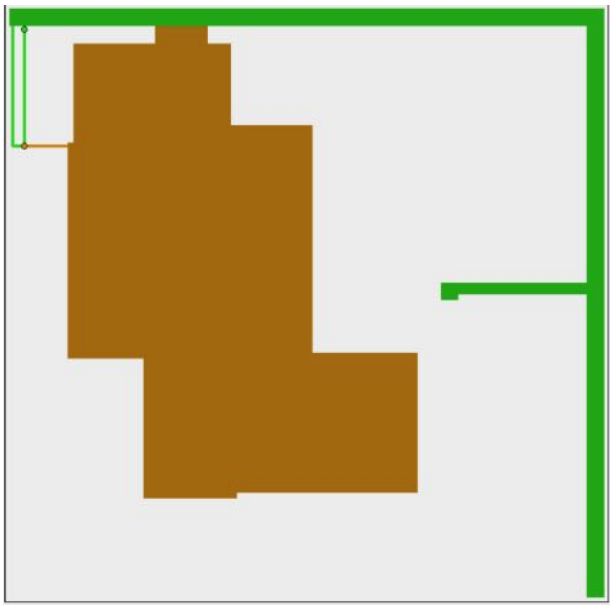


图 15 我方与 delta 组对战界面

2、领地被对手撞断。由于我方在未围住全场前，圈出的领地均为沿边界的窄矩形，无法形成大面积的领地，故我方领地极易被圈地能力强的对手撞断，导致我方无法回到领地形成包围态势。

下图为我方与 sierra 的对战记录，我方先手绿色。

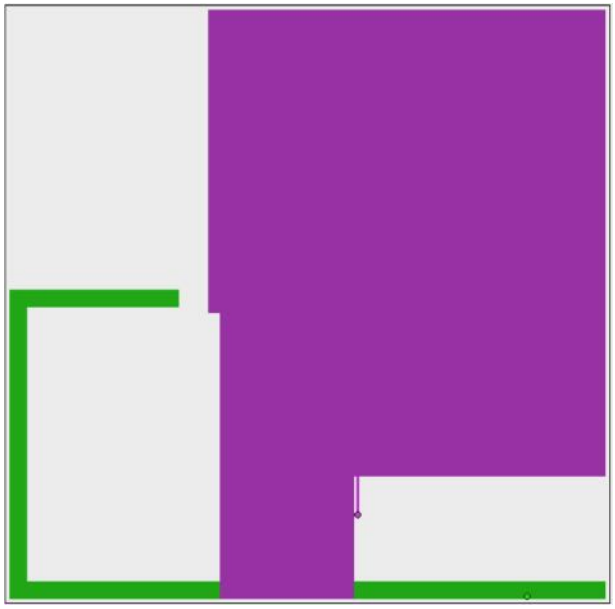


图 16 我方与 Sierra 组对战界面

3、我方思考时间耗尽。我方算法的时间复杂度为 O(width*height)，故在面对具有时间复杂度优势的对手时易耗尽思考时间而失败。

下图为我方与 hotel 的对战记录，我方后手。此时比赛结束，我方思考时间耗尽。

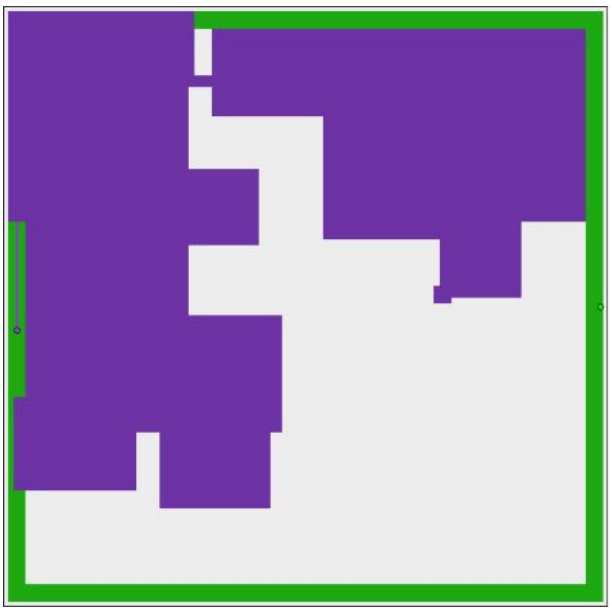


图 17 我方与 hotel 组对战界面

3.3 经典战局

4 实习过程总结

4.1 分工与合作

本小组组会包含一次正式线下组会和多次非正式线上组会。

4.1.1 线下组会

由于本小组组队时间较其他组而言较晚，我们在确认组队后不久（5 月 30 日）就开展了第一次组会。组会以线下方式进行，地点在元培学院宿舍楼地下讨论室，时间在 5 月 30 日下午两点半至五点半，组会内容以讨论基本策略和小组分工为主。

第一次组会（面基）中各成员互相认识，由组长姜峰先提出自己对于基本策略的想法，在白板上画图示意并进行阐释，认为主要需要写攻击函数和圈地函数两部分，针对纸带圈地游戏中可能出现的情况进行判断，从而控制我方纸带做出行动，给出了基本思路。组员朱洁薇和张晓宇对策略中的个别细节提出质疑，提出了纸带圈地游戏中可能出现的其他情

况。

由于组长姜峰还有使用机器学习算法完成此次实习作业的想法，最终决定由组员张晓宇和朱洁薇继续完善基本策略，由组长姜峰完成机器学习算法。

在两位组员继续讨论基本策略的过程中，组员朱洁薇提出可以针对对方纸带的行动进行博弈，只需判断何时需要进行转弯即可。在继续讨论中，张晓宇提出可以通过判断对方纸带头到我方纸带的最小距离与我方纸带头回到我方领地的最小距离的大小（圈地时防止被攻击）、我方纸带头到对方纸带的最小距离与对方纸带头回到对方领地的最小距离的大小（进行主动攻击）等几个双方相对距离的大小来决定我方纸带下一步如何行动。

基本策略仍然大致分为攻击和圈地两部分，分工决定由张晓宇负责圈地函数的部分，由朱洁薇负责攻击函数的部分，遇到问题时再进行讨论。

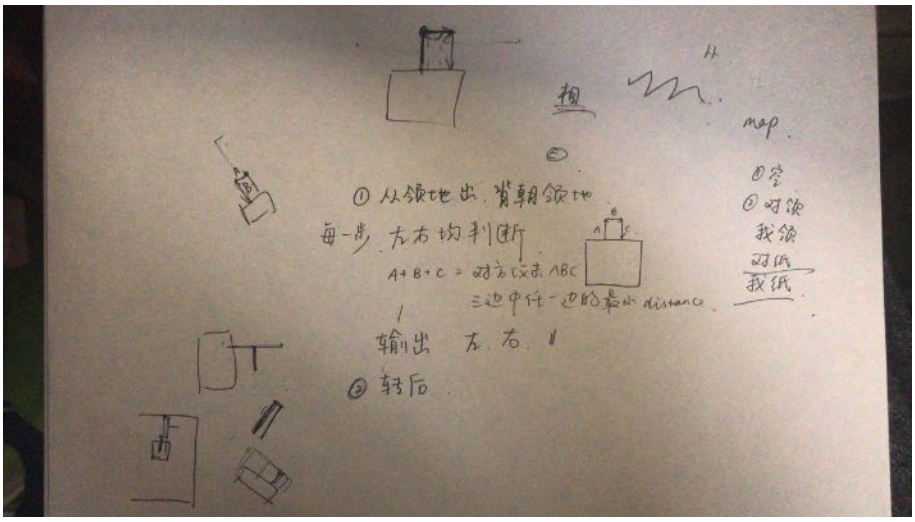


图 18 线下讨论思路笔记

本次小组会议内容大致如上。

4.1.2 线上组会

除了线下组会之外，本小组同样进行了多次线上组会，以微信群聊的方式进行，主要内容为交流两种算法的进展以及讨论进展中所遇到的问题。

4.1.2.1 第一次线上组会

第一次线上组会在 6 月 3 日，主要针对基础策略中遇到的问题进行讨论。

在之前线下组会提出了基础策略的基本想法后，两位组员在具体实现过程中遇到了一些困难。

首先遇到的问题是关于圈地函数的。在之前的线下组会中，我们只讨论了如何在圈地时避免攻击，但是没有讨论在纸带离开我方领地后如何进行圈地，即具体的圈地策略。比如在离开领地后默认的圈地形状，纸带在躲避攻击后何时选择返回领地能够使圈地范围最大化。

其次是在计算最小距离时，在领地形状不规则时如何计算最小距离，是否需要得到领地边界，这个问题在攻击函数和圈地函数中都遇到了。

针对第一个关于圈地函数的问题，讨论未得出一个最优圈地策略，即圈地形状的长宽比等。但是由于我们的圈地思路是根据对手行动判断何时需要转弯，圈地形状一定是不规则的，在没有判断到需要返回时就一直前进，我们认为应该会得到动态情况下最大的圈地面积。

针对第二个问题，组长姜峰提出不需要得到具体边界所占的坐标，领地外任意一点到领地的最短距离一定是到边界的最短距离，需要考虑的是如何确保在转弯过程中不会撞到我方纸带。

第一次线上组会内容大致如上。

4.1.2.2 第二次线上组会

第二次线上组会在 6 月 5 日，基本策略方遇到了问题。

由于两位组员之前没有实习作业的经验，也没有丰富的编程经验，无法完成完整的基本策略算法，因此决定调整分工，由两位组员提供目前研究出的策略思路，由姜峰具体实现。之后由两位组员完成实习作业报告。

组长认为两位组员的策略用到的实时信息太少，提出了一个新的基本策略思路，我们称为“偷鸡算法”。让我方纸带沿着边界圈出长条型的领地，同时注意避免被攻击，通过连接一圈领地占领整片领地。

第二次线上组会内容大致如上。

4.1.2.3 第三次线上组会

第三次线上组会在 6 月 9 日，主要根据目前机器学习算法和基本策略算法各自的进程

进行讨论。

机器学习算法虽已完成，但是经过训练，发现机器学习算法的效果并不是非常好，在剩下的两天时间内尽力对机器学习算法再进行调整和改进。

而我们的基本策略算法也面临比较复杂的问题。在我们的测试中，“偷鸡算法”取得了比较好的效果，但是“偷鸡算法”面临的问题在于，如果圈出的领地被对方截断，即回到原点后无法连成完整的一圈，不仅耗费了大量时间和回合数，而且圈地面积也非常小，将处于非常不利的境地。

最终我们选择通过再次在周围圈地来减小被截断的概率。

4.1.2.4 第四次线上组会

第四次线上组会在 6 月 11 日，机器学习算法在改进后的表现仍然不好，因此最终我们选择基本策略算法，此次组会主要希望通过研究热身赛复盘录像改进基本策略算法。

我们发现其他小组的策略与我们最初的基本策略思路相似，纯攻击算法和纯圈地算法都取得了非常好的效果，说明“平平淡淡才是真”。然而我们的“偷鸡算法”虽然在测试中取得了还不错的效果，实际对战中遇到了比较严重的问题，被截断的问题仍然无法避免，也没能做到非常好的防守，说明其他小组的算法远比我们使用的测试算法要“智能”。

然而此时已经没有充足的时间再转回曾经的思路，我们选择继续改进现有的“偷鸡算法”。

4.2 经验与教训

本次实习作业的结果并不理想。经过认真反思，我们认为有如下原因：

首先，是组队遇到的问题。由于大多数同学都在学期初确定了小组成员，彼此之间经过一学期的互相熟悉，在学期末能够更好地进行合作。而我们小组确定成员比较晚，彼此也不熟悉，因此在合作中还是遇到了一些问题。而且小组成员仅有三人，每人的工作量变大。

除此之外，小组成员中拥有编程经验的人太少。由于我们组编程经验丰富的同学只有组长姜峰一人，小组成员在完成算法时遇到了比较多的问题，剩余两人很难将自己的想法实现，从而无法比较两个策略的优劣，也耽误了较多的时间。

而且我们发现在热身赛后，很多其他小组在自己的算法中也加入了“偷鸡算法”作为他们策略的一部分。我们的“偷鸡”算法想赢就要赢在出其不意，在热身赛中暴露底牌将我们置于不利境地，让其他小组同学可以针对我们修改策略。

但最关键的原因还是我们自身的算法不够完美，考虑到的情况不够多，导致面对其他强劲选手时处于不利境地。

综上，我们小组失利既有主观原因也有客观原因。主观原因在于我们策略出现问题，不应该提前暴露底牌；客观原因在于我们本身缺少擅长写程序的组员，我们的“偷鸡”策略也不够完美。

但是在整个小组合作过程中也有以下方面做得比较好。

首先，在遇到问题时及时沟通。当两位组员在写算法过程中遇到解决不了的问题时，及时与组长进行沟通，通过讨论得出了解决方法。

其次，在分工出现问题时迅速进行调整。当发现两位组员无法完成基础策略代码的编写时，及时调整了小组分工，由组长负责代码，组员负责实习作业报告，确保了最后实习作业的顺利完成。

最后，小组成员都能够为小组着想，努力最大化整个小组的成果，主动担任自己能力范围内能够完成的工作，没有出现“free-rider”。

4.3 建议与设想

4.3.1 建议

组队方面，建议老师在日后可以尝试在中期退课结束后随机分组，从统计学意义上避免“大神抱团”的情况，同时也能够锻炼每个人与不熟悉的同学合作的能力。

基础设施代码方面，技术组成员非常好地完成了他们的工作，我们没有改进意见。

竞赛方面，如果能够按照助教的数量分成不同组，各组使用可视化界面进行竞赛，使得同组竞技的同学彼此能够直观看到比赛情况，可能比分组循环赛时只能看到比分的方式更能吸引同学的注意力，同学在比赛中的参与感更强。但是本次分组循环赛采取的方式确实能够节省更多的时间，能够更快的给出晋级结果，二者还需要老师和助教进行取舍。

4.3.2 设想

我们对于以后的实习作业有以下设想。

首先，可以尝试一次性提供 3-4 个选项，让选课的同学共同参与，投票决定。

除此以外，可以在学期初公开招募技术组、讲解组、志愿组等各小组成员，流程透明公开，增加选课同学的参与度。

还可以增加小组成员匿名互评环节，以尽可能防止“搭便车”行为出现，让每个同学都尽可能多的参与到实习作业中来。

4.3.3 寄语

对于选修明年课程的学弟学妹，我们有如下寄语：

愿你们始终保持自信，遇到困难不要轻言放弃，选择了就坚持到底；愿你们有问题及时求助，不要自己硬扛；愿你们遇到同样靠谱的队友，能够一起学习奋斗；愿你们享受过程，满载而归！

5 致谢

此次实习作业完成，非常感谢陈斌老师为机器学习算法提供的训练设备，虽然最后机器学习算法的效果不是很好，但是至少我们尝试过。

也非常感谢技术组同学搭建的基础设施，方便我们研究其他小组的策略，也方便我们自我反思与改进。

同样感谢其他小组的一些同学在我们提出问题时能够认真解答。

感谢地空数算大群让我们三个能够相遇并成为队友，收获了更多的友谊与感动。

感谢元培学院的讨论室，为我们提供了非常好的讨论场所和讨论环境，我们在讨论中收获良多。

感谢腾讯开发出的微信，让我们遇到问题能够畅通无阻、随时随地进行沟通。

感谢我们是三条单身狗，没有男/女朋友让我们分心，影响我们学习。

6 参考文献及附录

1.Open Course: `http://www0.cs.ucl.ac.uk/staff/d.silver/web/Teaching.html`

2.Keras Documentation: `https://keras.io/`

3.Mnih, V., Badia, A. P., Mirza, M., Graves, A., Harley, T., Lillicrap, T. P., Silver, D., and Kavukcuoglu, K. (2016). Asynchronous methods for deep reinforcement learning. In the International Conference on Machine Learning (ICML)

文件名	说明
t_n17_golf.py	常规算法
DQL.py	用来生成数据并训练神经网络
testNet.py	用来调用训练好的模型，对接比赛的接口

第八章　N17_Hotel 报告

申景航 * 周劲松 李思伟 程本夏 马艺铭

摘要：本组算法主要以 Attack 和 Farm 两大核心策略组成。当 Attack 判断可以击杀对方时，会按规划的进攻路线发动进攻；而在不能击杀对面时，采取 Farm 策略来进行圈地。在与其他 AI 对战时，基本能实现抓住对方的失误进行击杀，并保证一定的圈地效率。经统计数据分析，本 AI 的获胜方法主要以击败对手为主。

关键字：纸带圈地，纸带线段化，矢量计算距离，最大安全圈地距离，AI

1 算法思想

1.1 总体思路

本方的每个回合开始先调用 Attack 部分，将双方纸带线段化，用我方位置和线段的矢量乘积来计算进攻点和路径长度，并与敌方到其领地的距离大小来初步判断能否击杀对方。初步判断结束后，再用双方距离、领地大小、是否会自己碰到自己纸带等因素来检验是否要去选择实施进攻。如果不实施进攻，则调用 Farm 部分。

Farm 部分策略偏向保守，分为在领地外和领地内两部分。

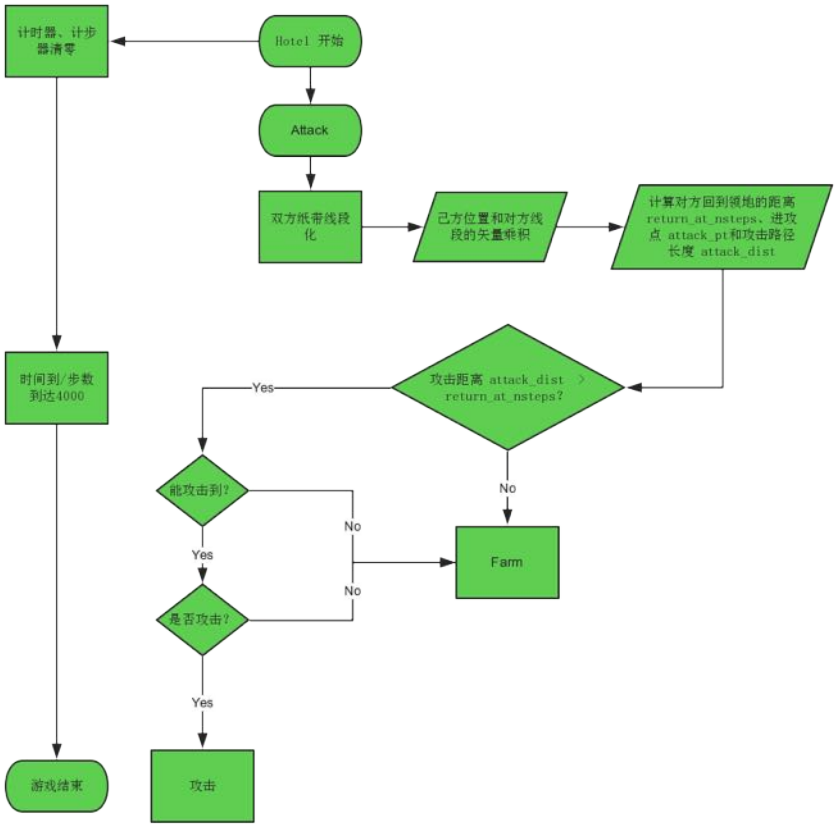
领地外：刚出领地时就根据双方位置进行圈地路线规划，并在第一次转弯前重新规划，判断能否扩展圈地面积。

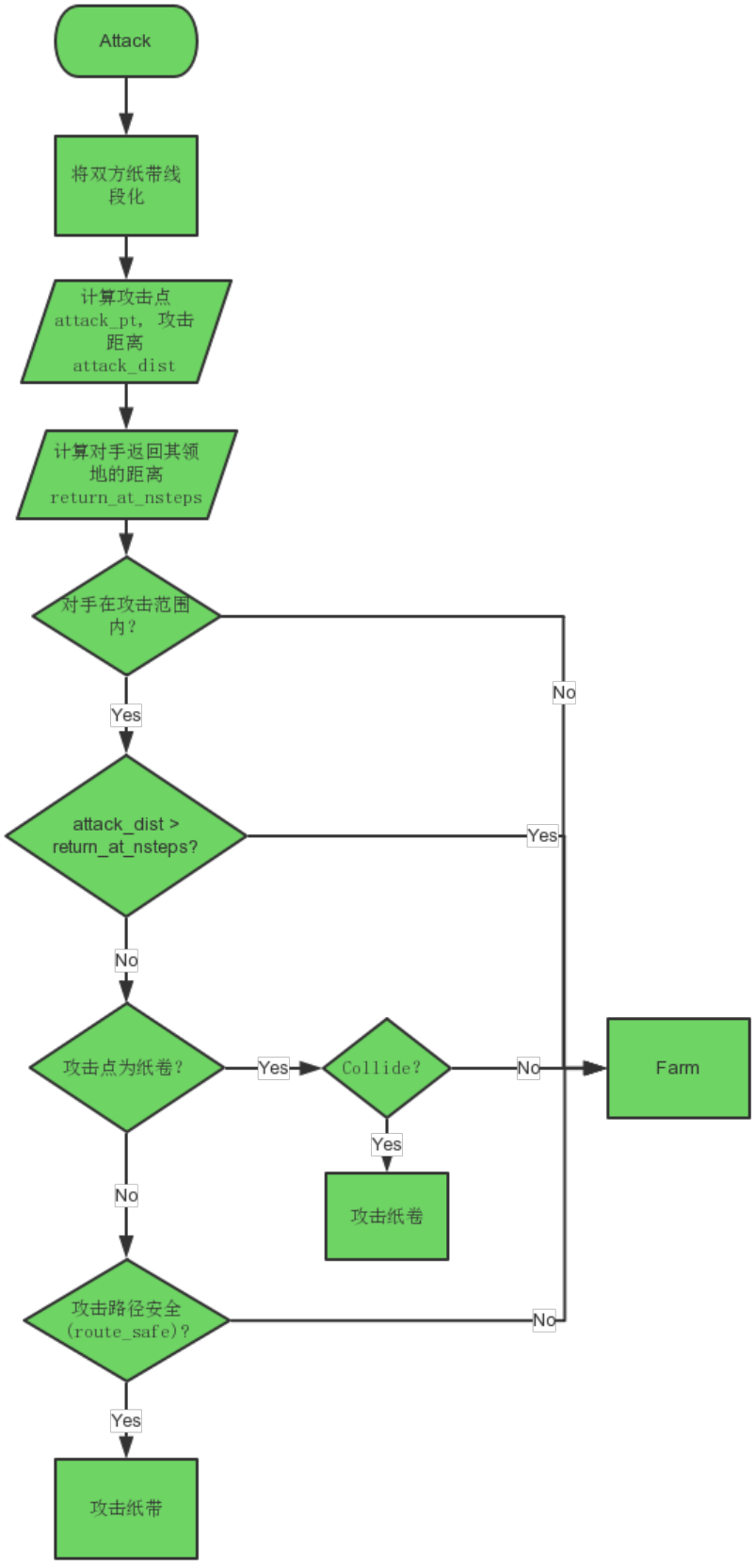
领地内：优先考虑双方是否进入近距离纠缠，如果距离小于 5 则不出领地；再考虑要抵抗敌方对自己领地的入侵，如果敌方在己方领地，则靠近敌方；如果双方没有以上冲突，

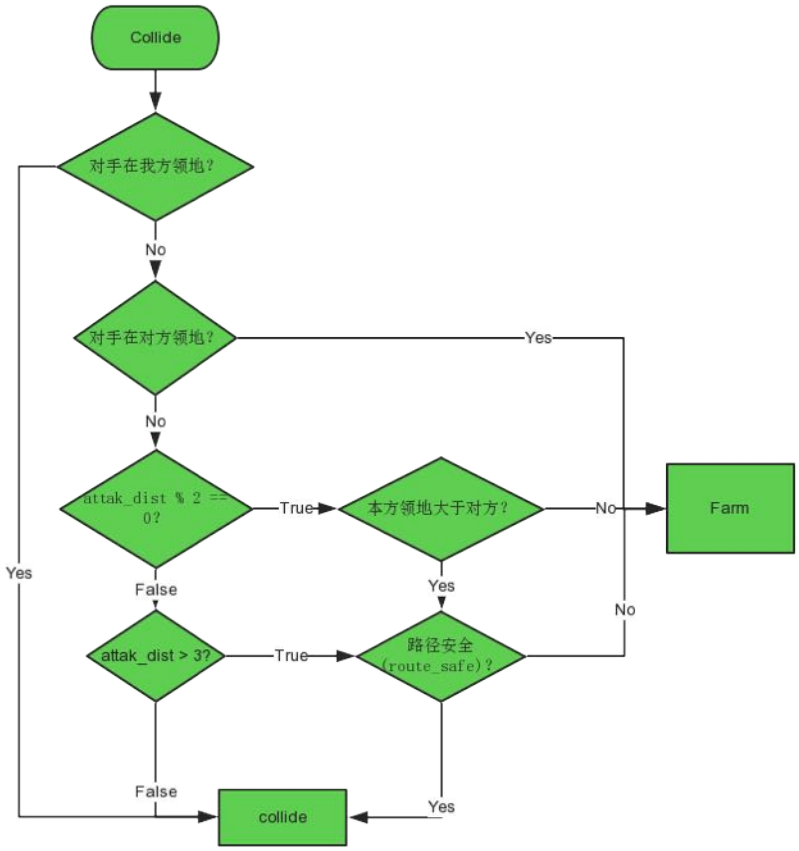
就在进入领地后通过连续转向来避免在领地内浪费过多步数；最后避免朝着前方已经全是自己领地的方向走。

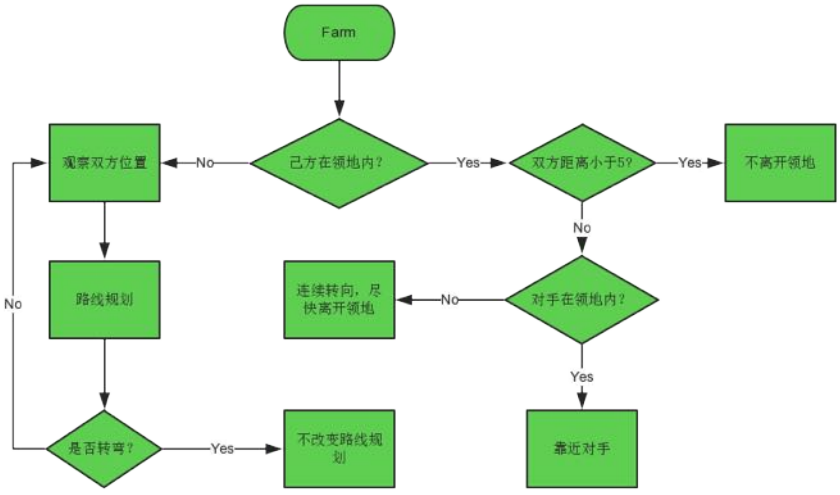
上述过程中都已经考虑并避免了撞到边界的问题。

1.2 算法流程图




Attack
将双方纸带线段化
计算攻击点attack_pt，攻击距离attack_dist
计算对手返回其领地的距离return_at_nsteps
对手在攻击范围内？
No
Yes
attack_dist > return_at_nsteps?
Yes
No
攻击点为纸卷？
Yes
Collide?
No
Farm
Yes
攻击纸卷
No
攻击路径安全(route_safe)?
No
Yes
攻击纸带


Collide
对手在我方领地?
No
对手在对方领地?
Yes
No
attak_dist % 2 == 0?
True
本方领地大于对方?
No
Farm
Yes
False
Yes
No
attak_dist > 3?
True
路径安全(route_safe)?
False
collide
Yes


Farm
己方在领地内?
No
观察双方位置
Yes
双方距离小于5?
Yes
不离开领地
No
对手在领地内?
No
连续转向，尽快离开领地
Yes
靠近对手
路线规划
No
是否转弯?
Yes
不改变路线规划

1.3 算法运行时间复杂度分析

代码中遇到的最坏情况为可以进攻到对手，且双方距离为偶数的情况。因为此时需要计算比较双方领地面积大小，而本组采用计算领地面积的方式为遍历整个场地来计数，时间复杂度为 O(n^2)。但这种情况发生概率较小，大部分时间都是进行简单的判断，时间复杂度为 O(1)。

2 程序代码说明

2.1 数据结构说明

算法中采用的主要数据结构是字典和列表，用于储存必要的信息、策略选择、矢量计算和等功能。

2.2 函数说明

2.2.1 基本信息和初始化

```
if 'Basic Informations:':
    '''
    常用信息的调用方式:
    me = storage['now']['me']
    enemy = storage['now']['enemy']

    注意:
    me['direction']是int
    directions[me['direction']]是单位向量

    攻击位置
    attack_pt   = storage["dist"]["me_attack"][0]
    攻击距离
    attack_dist = storage["dist"]["me_attack"][1]

    纸带信息:
    Band_segs_me    = storage['Band_segs']['me']
    Band_segs_enemy = storage['Band_segs']['enemy']

    宏观面积估计
    storage['score']['me']
    storage['score']['enemy']
    '''
```

2.2.2 定义基本函数和变量

```
if 'Functions':

    def distances(me, enemy):
        '''
        到敌方纸带的距离;
        调用方式:
        攻击位置
        pt   =   storage["dist"]["me_attack"][0]
        攻击距离
        dist=   storage["dist"]["me_attack"][1]
        '''
        me_attack = dist_pt_segs(
            [me['x'], me['y']], storage['Band_segs']['enemy'])
        enemy_attack = dist_pt_segs(
            [enemy['x'], enemy['y']], storage['Band_segs']['me'])
        storage['dist'] = {
            'me_attack': me_attack,
            'enemy_attack': enemy_attack
        }
        return storage['dist']
```

```
def count_score(fields, me, enemy):
    '''
    近似统计场上面积数，用于评判胜负.
    '''
    res = [0, 0]
    if fields[me['x']][me['y']] == me['id'] or fields[enemy['x']][enemy['y']] == enemy['id']:
        res = [0, 0]
        for x in range(6, WIDTH, 10):
            for y in range(5, HEIGHT, 10):
                if fields[x][y] is not None:
                    res[fields[x][y] - 1] += 1
    storage['score'] = {'me': res[me['id'] - 1],
                        'enemy': res[enemy['id'] - 1]}
    return storage['score']

def distance(p1, p2):
    '''返回两点间的折线距离'''
    return abs(p1[0] - p2[0]) + abs(p1[1] - p2[1])

def is_segment(p1, p2):
    '''判断两个点是否构成了一个线段，即是否在同一水平线/竖直线上'''
    return (p1[0]-p2[0]) == 0 or (p1[1]-p2[1]) == 0

def dot(p1, p2):
    '''向量的点乘'''
    return p1[0] * p2[0] + p1[1] * p2[1]

def cross(p1, p2):
    '''向量的叉乘，可以用来比较两个向量的方向是否逆时针'''
    return p1[0] * p2[1] - p1[1] * p2[0]

def dist_pt_segment(p, p1, p2):
    '''
    点p到线段p1->p2的最短驱离;
    返回值:距离与相应的点.
    key为pt与dist.
    '''
    if is_segment(p1, p2):
        seg = [p1, p2]
        if p1 == p2:
            return {'pt': None, 'dist': 233}
        ind = ind_dist(p, seg)
        direc = direct(seg[ind], seg[1 - ind])
        inner_product = dot(
            addition(seg[ind], p, -1), directions[direc])
        if inner_product >= 0:
            return {'pt': seg[ind], 'dist': distance(p, seg[ind])}
        else:
            pedal = addition(
                seg[ind], directions[direc], -inner_product)
            return {'pt': pedal, 'dist': distance(pedal, p)}
    else:
        return {'pt': None, 'dist': 233}
```

```
def dist_pt_segs(p, segs):
    '''
    寻找折线段上距离最短的点.
    返回一个字典,键为pt和dist;
    注意到这里的排序保存了线段顺序,即会优先从较早的线段处返回最小值!!!!!!
    '''
    try:
        dic = {dist_pt_segment(
            p, segs[i], segs[(i + 1) % len(segs)])['pt']: dist_pt_segment(
            p, segs[i], segs[(i + 1) % len(segs)])['dist'] for i in range(len(segs)-1)}
        dic = sorted(dic.items(), key=lambda e: e[1])
        # ind = dic[0][0]
        # return dist_pt_segment(p, segs[ind], segs[(ind + 1) % len(segs)])
        return dic[0]
    except:
        return (None, 233)
```

2.2.3 用到的所有决策

在我们的程序中，攻击部分 Attack 与圈地部分 Farm 构成了两个分立的模块。这两个函数的功能集成到了 Decision 函数当中，共同给出了每回合的最终决策。它是由如下的代码实现的:

首先将每回合必要的信息加载完成，包括更新纸带信息与进行攻击位置与攻击距离的计算。然后判断敌方是否在皮。当判定能够对敌方造成效果拔群的伤害时，直接执行攻击模块，勇猛地扑向对手；若不能立即击杀，则让 Attack 函数隐忍蛰伏，等待机会，执行 Farm 函数 (名称为 _play) 进行圈地。将 Decision 函数储存在 storage 中，并在主要的 AI 函数 play 中进行调用。我们的程序是模块化的，这样做的好处在于结构清晰，接口简单，易于维护与拓展，有利于我们的团队开发。事实上，在 Decision 这个大框架下，我们只需要在 AI 的 load 函数中分别设计并维护好 Attack 与 _play 两部分就可以轻松实现 AI 的更新优化。此外，我们添加了一些回合信息的打印，用于 debug 过程中一些最精细的数值改动。

```
def Decision():
    '''所有的决策汇总于此'''
    # 加载必要的信息
    S = Segments()
    me = stat['now']['me']
    enemy = stat['now']['enemy']
    fields = stat['now']['fields']
    D = distances(me, enemy)
    if Score_Counter:
        Score = count_score(fields, me, enemy)
    Att = Attack(me, enemy, fields)

    # 判断能否攻击成功，如果不能一击致命，则继续farm.
    if Att['Judge']:
        # 下面这些时DEBUG时用的~***********
        if Attack_Information:
            turn = 2000 - stat['now']['turnleft'][me['id'] - 1]
            print('\nTurn:\t', 2 * turn + me['id'] - 1)
            print('Attack:\tPlayer %d' % me['id'])
            print('Pos:\t\t', S.Pos)
            print('Band:\t\t', S.Band_segs)
            print('Att_aim:\t', storage['dist'])
        return Att['decision']
    else:
        return _play(stat, storage)
storage['Decision'] = Decision
```

2.2.4 定义纸带的 ADT

由于我们组写 Attack 同学的电脑比较低配, 他想尽可能地避免在回合中进行遍历一类的操作. 为此, 我们组的同学实现了一个尽可能少计算量计算距离的方案.

在实现 Attack 模块的过程中, 有几个关键的节点: 首先是碰撞模块, 即判断能否通过碰撞取得胜利, 并且给出路径. 其次, 要计算若干个距离, 来判断是否有机会截杀敌方. 对于我方进行攻击的情形, 需要计算的距离包括: 我方攻击敌方纸带的最近距离及相应攻击位置; 在这个攻击距离下敌方是否能够丢盔弃甲逃回领地; 攻击路径有没有可能没敌方截断等. 归纳一下, 我们要求的由点到纸带的距离, 以及点到区域的距离.

为了避免遍历过程的过高计算量, 包括确定纸带与领地范围所需的遍历, 以及计算距离所必要的遍历, 我们采取了一些有趣的小技巧.

首先计算点到纸带的距离. 我们的方案是将纸带线段化, 并在每回合开始时动态地维护纸带的拐点列表 Band_segs['me'] 以及 Band_segs['enemy']. 这个部分我们单独实现在名为 Segments 的模块当中, 并在每回合开始时进行调用. 特别地, 为了检验这个计算过程是否正确, 我们写了一个测试函数 _test, 用于每间隔若干回合就进行以此输出. 我们将 _test 函数的开关放在了 load 顶部的 parameters 当中.Segments 模块代码如下:

```
class Segments:
    '''
    每回合将类型实例化时,
    自动更新线段化的纸带Band_segs;
    通过storage['Band_segs]调用运算结果.
    '''

    def __init__(self):
        # bands = stat['now']['bands']
        fields = stat['now']['fields']
        me = stat['now']['me']
        enemy = stat['now']['enemy']

        self.Pos = self._init_Pos(me, enemy)
        self.Band_segs = self._init_Band_segs(fields, me, enemy)

        if Turn_Information:
            self._test(stat, me, enemy)

    def _test(self, stat, me, enemy):
        '''用于打印回合信息到控制台'''
        turn = 2000 - stat['now']['turnleft'][me['id'] - 1]
        timeleft = {
            'me': stat['now']['timeleft'][me['id'] - 1],
            'enemy': stat['now']['timeleft'][enemy['id'] - 1]
        }
```

```
    if turn % TURN == 9999:
        print('\nTurn:\t', 2 * turn + me['id']-1)
        print('Time:\t', timeleft)
        # 在此添加其它检验对象
        print('Pos:\t', self.Pos)
        print('Band:\t', self.Band_segs)
        print('Score:\t', count_score(
            stat['now']['fields'], me, enemy))

def _init_Pos(self, me, enemy):
    return {
        'me': (me['x'], me['y']),
        'enemy': (enemy['x'], enemy['y'])
    }
```

```
def _init_Band_segs(self, fields, me, enemy):
    # 在执行_init_Field_segs时,对Band_segs清空
    '''Band_segs[0]不在领地内,这是纸带转折点构成的列表'''
    Band_segs = storage['Band_segs']
    segs_me = Band_segs['me']
    len_me = len(segs_me)
    if fields[me['x']][me['y']] == me['id']:
        segs_me.clear()
    elif len_me < 2:
        segs_me.append((me['x'], me['y']))
    elif me['direction'] == direct(segs_me[(-2) % len_me], segs_me[(-1) % len_me]):
        segs_me.pop()
        segs_me.append((me['x'], me['y']))
    else:
        segs_me.append((me['x'], me['y']))

    segs_enemy = Band_segs['enemy']
    len_enemy = len(segs_enemy)
    if fields[enemy['x']][enemy['y']] == enemy['id']:
        segs_enemy.clear()
    elif len_enemy < 2:
        segs_enemy.append((enemy['x'], enemy['y']))
    elif enemy['direction'] == direct(segs_enemy[(-2) % len_enemy], segs_enemy[(-1) %
        len_enemy]):
        segs_enemy.pop()
        segs_enemy.append((enemy['x'], enemy['y']))
    else:
        segs_enemy.append((enemy['x'], enemy['y']))

    return Band_segs
```

进行线段化并且维护的好处在于，我们无需每回合遍历场地信息，只需每回合花费很小的代价维护信息，并且只需遍历所有的线段，通过研究线段的两个端点与我方纸卷的位置关系，就可以得到相应的距离，并且计算出攻击位置。在纸带上存在较长线段时，避免了大量的重复计算，能够有效地提升效率。然而在极端鬼畜的情况下，会与通常的遍历算法耗费同等时间。

对于单独线段的情形，由函数 dist_pt_segment 完成计算。

对于一般的复杂折线段情形，则要通过 dist_pt_segments 来寻找攻击位置与攻击距离. 特别地，我们利用了排序过程的稳定性, 会优先攻击敌方纸带较早的位置。

2.2.5 Attack 和 Judge_Collide 函数

Judge_Collide 函数

这个部分是 Attack 模块中最早实现的部分之一，通过了高能纸卷对撞机的无数次检验。我们小组的成员很早就注意到了开局距离的奇偶性对游戏过程巨大的影响，因而很早地完成了针对敌方纸卷进行攻击的 Judge_Collide 函数，特别是在初始距离为奇数，具有侧碰优势的情形。

```
def Judge_collide(me, enemy):
    '''
    判断能否通过纸卷对撞消灭敌人;
    返回字典,key为Judge与Decision;
    Judge取bool值,取真值时不要怂~
    Decision取值为前进或左右转向;

    Collide 在Attack函数中, 调用时已经满足如下条件:
        1. 敌方无法返回己方领地
        2. 双方处于距离30以内
        3. 敌方纸卷处于其纸带上距离我方最近的点
        4. 根据sorted函数的机制, 我们计算得到的attack_pt优先取在敌方较早的线段上
    '''
    dist = distance([me['x'], me['y']], [enemy['x'], enemy['y']])
    fields = stat['now']['fields']

    # 领地内抵抗入侵
    if fields[enemy['x']][enemy['y']] == me['id']:
        if not return_at_nsteps(dist, enemy, fields):
            direc = direct(me, enemy)
            if me['direction'] == direc:
                decision_value = ''
            elif(me['direction'] + 1) % 4 == direc:
                decision_value = 'r'
            elif(me['direction'] - 1) % 4 == direc:
                decision_value = 'l'
            else:
                return {'Judge': False, 'decision': ''}
            return {'Judge': True, 'decision': decision_value}

    # 若我方回合时,双方距离为偶数,则无法在领地外通过对撞击杀敌人.
    elif dist % 2 == 0:
        # ***判断领地大小***
        if Judge_score(me, enemy):
            # 己方领地大于对方时,直接撞上去
            return route_safe(me, (enemy['x'], enemy['y']), storage['Band_segs']['me'])
        else:
            return {'Judge': False, 'decision': ''}

    else:
        # 己方领地小于等于对方时:
        if dist > 3:
            return route_safe(me, (enemy['x'], enemy['y']), storage['Band_segs']['me'])

        elif dist == 3:
            if addition([me['x'], me['y']], directions[me['direction']], 3) == [enemy['x'],
                enemy['y']]:
                # 指向敌方时,左转
                decision_value = 'l'
            elif me['x'] == enemy['x'] or me['y'] == enemy['y']:
                # 不指向敌方时,直走
                decision_value = ''
```

```
            else:
                # 距离为sqrt(5)时,朝向敌方前进
                direc = direct(me, enemy)
                if me['direction'] == direc:
                    decision_value = ''
                elif(me['direction'] + 1) % 4 == direc:
                    decision_value = 'r'
                elif(me['direction'] - 1) % 4 == direc:
                    decision_value = 'l'
        elif dist == 1:
            # 距离为1时,直接撞上去.
            direc = direct(me, enemy)
            if me['direction'] == direc:
                decision_value = ''
            elif(me['direction'] + 1) % 4 == direc:
                decision_value = 'r'
            elif(me['direction'] - 1) % 4 == direc:
                decision_value = 'l'
    if Attack_Information:
        print('Player %d:\tCOLLIDE!' % me['id'])
    return {'Judge': True, 'decision': decision_value}
```

Attack 函数

Attack 函数的前身是针对敌方纸带的攻击函数，我们将其与 Judge_Collide 函数的逻辑合并到了一起。

其中主要包括了一个攻击范围的判定，一个能否再敌方返回前击杀的判定，以及击杀路径不会自杀的判定。程序会根据以上这些判定信息来决定是否展开攻击，以及攻击纸带还是纸卷。

```
# 加上不自杀的判断
# 更新了Attack中判断的逻辑，原先版本可能不能正确处理对撞情形
def Attack(me, enemy, field):
    '''Attack 假定了 玩家Farm的过程是安全的'''
    me_attack = storage['dist']['me_attack']
    attack_pt = me_attack[0]
    attack_steps = me_attack[1]
    # 以下包含了撞击纸卷与咬断纸带两个部分
    if attack_steps < ATTACK_RANGE:
        if not return_at_nsteps(attack_steps, enemy, field):
            # ******************************此处添加了不会撞自己的判断
                ****************************************
            if Attack_Information:
                print('Player %d:\tYou have no way to escape!' % me['id'])
            if attack_pt != (enemy['x'], enemy['y']):
                return route_safe(me, attack_pt, storage['Band_segs']['me'])
            else:
                return Judge_collide(me, enemy)
        else:
            return {'Judge': False, 'decision': ''}
    else:
        return {'Judge': False, 'decision': ''}
```

值得一提的是，通常情况下，上述的第二个判定需要一个特殊的距离：敌方返回领地的距离，并且将其与我方攻击距离进行比较。然而这个距离是点到区域的距离，严格的计算是必然要通过遍历来进行的。这与我方猪队友电脑的能力不符。因而我们找到了一套特

殊的方法：我们不需要真的取计算距离，而是只需判定再一个给定的距离下，敌方能否返回。这样的修改使我们能够避免遍历，大幅提高时间效率。具体方法是，再给定一个距离 dist 后，判断敌方纸卷与纸带初始顶点的距离；判断敌方纸卷距离为 dist 的邻域的边界上是否有敌方领地；为保险起见，还判断了 dist/2 邻域边界的情形。除了一些极端支离破碎的鬼畜情形，这种判定方式都是有效的。

```
def return_at_nsteps(steps, plr, field):
    '''粗略地判断玩家plr能否恰好在n步是返回领地,复杂度为O(4n)'''
    # 更新了一条逻辑：判断steps大于敌方纸带初末距离
    if plr == stat['now']['enemy']:
        if steps > distance(storage['Band_segs']['enemy'][0], storage['Band_segs']['enemy'][-1]):
            return True
    # 进攻时有1步的先手。
    n = int(steps) - 1
    x = plr['x']
    y = plr['y']
    # direction = plr['direction']
    # 方形范围四个顶点
    p_forward = addition((x, y), directions[3], n)
    p_left = addition((x, y), directions[2], n)
    p_right = addition((x, y), directions[0], n)
    p_back = addition((x, y), directions[1], n)
    lis = [p_forward, p_left, p_right, p_back]
    for i in range(1, n):
        lis.append((p_forward[0] - i, p_forward[1] + i))
        lis.append((p_left[0] + i, p_left[1] + i))
        lis.append((p_right[0] - i, p_right[1] - i))
        lis.append((p_back[0] + i, p_back[1] - i))
    lis = [p for p in lis if 0 <= p[0] < WIDTH and 1 <= p[1] < HEIGHT]
    for p in lis:
        if field[p[0]][p[1]] == plr['id']:
            return True
```

```
    # 保险起见，在加一次判定。
    n = n // 2
    p_forward = addition((x, y), directions[3], n)
    p_left = addition((x, y), directions[2], n)
    p_right = addition((x, y), directions[0], n)
    p_back = addition((x, y), directions[1], n)
    lis2 = [p_forward, p_left, p_right, p_back]
    for i in range(1, n):
        lis2.append((p_forward[0] - i, p_forward[1] + i))
        lis2.append((p_left[0] + i, p_left[1] + i))
        lis2.append((p_right[0] - i, p_right[1] - i))
        lis2.append((p_back[0] + i, p_back[1] - i))
    lis = [p for p in lis if 0 <= p[0] < WIDTH and 1 <= p[1] < HEIGHT]
    for p in lis:
        if field[p[0]][p[1]] == plr['id']:
            return True

    # 若以上两圈均无领地，则判定无法返回。
    return False
```

2.2.6 游戏基础函数 _play(stat,storage) (用于实现最基础的判断以及圈地和领地内行为)

这一部分决定了纸卷在圈地上的行为方式，具体分为领地外和领地内。

当己方纸卷处于领地外时，按领地外部分来进行决策：首先要初始化一些计数器的值，并记录刚出领地的原点坐标。

```
# 初始化转弯方向（默认左转）、计步器、和记录转弯次数的计数器
if not 'turn' in storage:
    storage['turn'] = '1'
if not 'steps' in storage:
    storage['steps'] = 0
if not 'definedirection' in storage:
    storage['definedirection'] = 0
if not 'startx' in storage:
    storage['startx'] = 0
if not 'starty' in storage:
    storage['starty'] = 0

# 当转弯计数器为零时开始计算安全范围内走的长度，经过一次计算的结果最大值的最小值为玩家距离//5
if storage['definedirection'] == 0:
    # 避免重复进入
    storage['definedirection'] = 1
    # 录入起始点坐标
    storage['startx'] = stat['now']['me']['x']
    storage['starty'] = stat['now']['me']['y']
    storage['length'] = storage['playerdistance'] // 5
    storage['secondlength'] = storage['playerdistance'] // 6
    big = 0
    small = 0
    x = storage['startx']
    y = storage['starty']
    bigx = x
    smallx = x
```

在领地外的决策，最优先是避免撞墙。如果是刚出领地，则进行第一次路线规划，通过比较左转或右转带来的领地面积优劣来决定选择哪条路线。

```
        # 考虑是否在边界
        # 如果在左侧边界
        if stat['now']['me']['x'] == 0:
            storage['length'] = storage['playerdistance'] // 5
            # 如果朝南，左转
            if stat['now']['me']['direction'] == 1:
                storage['turn'] = '1'
            # 如果在西北角，且朝西，左转
            elif stat['now']['me']['y'] == 0 and stat['now']['me']['direction'] == 2:
                storage['turn'] = '1'
            # 其余右转
            else:
                storage['turn'] = 'r'
        # 如果在右侧边界
        elif stat['now']['me']['x'] == (len(stat['now']['fields']) - 1):...
        # 如果在上侧边界
        elif stat['now']['me']['y'] == 0:...
        # 如果在下侧边界
        elif stat['now']['me']['y'] == (len(stat['now']['fields'][0]) - 1):...
```

```
        # 不在边界
        else:
            # 安全范围内走的长度改为玩家距离//5，这种距离对方向有一定要求，无论什么方向都要避开对面；
            # 如果迎着对面，安全长度就为玩家距离//6，可以通过比较圈地收益，避免在边界附近走边界，来提高效率
            #如果自己朝东
            if stat['now']['me']['direction'] % 4 == 0:
                #自己在敌人北侧
                if stat['now']['me']['y'] <= stat['now']['enemy']['y']:
                    # 考虑边界的算法
                    while bigx < min((x + storage['length']), (len(stat['now']['fields']) - 1)):
                        bigy = y
                        while bigy > max((y - storage['length']), 0):
                            if stat['now']['fields'][bigx][bigy] == stat['now']['me']['id']:
                                big += 1
                            bigy -= 1
                        bigx += 1
                    big = min((storage['length'] + 1), len(stat['now']['fields']) - 1 - x) * min(
                        (storage['length'] + 1), y) - big
                    while smallx < min((x + storage['secondlength']), (len(stat['now']['fields']) - 1)):
                        smally = y
                        while smally < min((y + storage['secondlength']), (len(stat['now']['fields'][0]) - 1)):
                            if stat['now']['fields'][smallx][smally] == stat['now']['me']['id']:
                                small += 1
                            smally += 1
                        smallx += 1
                    small = min((storage['secondlength'] + 1), (len(stat['now']['fields']) - 1) - x) * min(
                        (storage['secondlength'] + 1), len(stat['now']['fields'][0]) - 1 - y) - small
```

```
                    if big >= small:
                        storage['turn'] = '1'
                    else:
                        storage['length'] = storage['secondlength']
                        storage['turn'] = 'r'
                else:...
            # 朝南
            elif stat['now']['me']['direction'] % 4 == 1:...
            # 朝西
            elif stat['now']['me']['direction'] % 4 == 2:...
            # 朝北
            else:...
```

在行进路线中，每走一步，计数器加 1，如果前方是墙，计数器直接加满来避免撞墙。

```
# 每走一步，计步器加1
storage['steps'] += 1
# 防止撞墙
nexty = stat['now']['me']['y'] + \
        directions[stat['now']['me']['direction']][1]
if nexty < 0 or nexty >= stat['size'][1]:
    storage['steps'] = storage['length']
nextx = stat['now']['me']['x'] + \
        directions[stat['now']['me']['direction']][0]
if nextx < 0 or nextx >= stat['size'][0]:
    storage['steps'] = storage['length']
```

第一次转弯前，判断能否扩大圈地范围，以提高圈地效率。转弯后，计数器清零并开始按规划好的路线决策，不再做变换。

```
# 走过一条边，转弯计数器加1，转四个弯清零
if storage['steps'] >= storage['length']:
    # 第一次转弯
    if storage['definedirection'] == 1:
        newdistance = abs(storage['startx'] - stat['now']['enemy']['x']) + abs(
            storage['starty'] - stat['now']['enemy']['y'])
        # 如果距离变大
        if newdistance // 5 > storage['length']:
            # 如果南北向
            if (stat['now']['me']['direction'] % 2) == 1:
                # 在上下界，转弯
                if nexty < 0 or nexty >= stat['size'][1]:
                    storage['steps'] = 0
                    storage['definedirection'] = (storage['definedirection'] + 1) % 5
                    return storage['turn']
                # 不在上下界，修改
                else:
                    storage['length'] = newdistance // 5
            # 如果东西向
            else:...

        # 距离没有变远
        else:
            storage['steps'] = 0
            storage['definedirection'] = (
                                          storage['definedirection'] + 1) % 5
            return storage['turn']
    # 不是第一次转弯
    else:
        storage['steps'] = 0
        storage['definedirection'] = (
                                        storage['definedirection'] + 1) % 5
        return storage['turn']
```

在领地内，先对领地外计数器清零，领地内策略优先考虑避免撞墙和双方边界纠缠的情况以确保安全；再判断敌方是否在侵占自己领地来实施抵抗；当双方的情况缓和时，采用连续转弯来快速投入到圈地中，避免在领地内浪费过多步数。最后用 Foresee 函数避免纸卷朝着前面都被自己占过的方向走。

```
    # 在field内计步器和计数器清零，防止撞墙，直走
    if stat['now']['fields'][stat['now']['me']['x']][stat['now']['me']['y']] == stat['now']['me']['id']:
        storage['steps'] = 0
        storage['definedirection'] = 0
        if stat['now']['me']['direction'] % 2:
            nexty = stat['now']['me']['y'] + \
                    directions[stat['now']['me']['direction']][1]
            if nexty < 0:
                if stat['now']['me']['x'] == 0:
                    return 'r'
                else:
                    return 'l'
            elif nexty >= stat['size'][1]:...
        else:  # x轴不出界
            nextx = stat['now']['me']['x'] + \
                    directions[stat['now']['me']['direction']][0]
            if nextx < 0:
                if stat['now']['me']['y'] == (len(stat['now']['fields'][0]) - 1):
                    return 'r'
                else:
                    return 'l'
            elif nextx >= stat['size'][0]:...

    # 优先考虑双方是否进入近距离纠缠，如果距离小于5则不出领地；
    if storage['playerdistance'] <= 5:
        right = (stat['now']['me']['direction'] + 1) % 4
        left = (stat['now']['me']['direction'] + 3) % 4
        # 执行左转、右转、执行后到达的位置
        nextxl, nextyl = stat['now']['me']['x'] + directions[left][0], stat['now']['me']['y'] + \
                         directions[left][1]
        nextxr, nextyr = stat['now']['me']['x'] + directions[right][0], stat['now']['me']['y'] + \
                         directions[right][1]
        nextxs, nextys = stat['now']['me']['x'] + directions[stat['now']['me']['direction']][0], \
                         stat['now']['me']['y'] + directions[stat['now']['me']['direction']][1]
        # 同时确保不出界
        if 0 <= nextxs < stat['size'][0] and 0 <= nextys < stat['size'][1] and \
                        stat['now']['fields'][nextxs][nextys] == stat['now']['me']['id']:
            return ''
        elif 0 <= nextxl < stat['size'][0] and 0 <= nextyl < stat['size'][1] and \
                        stat['now']['fields'][nextxl][nextyl] == stat['now']['me']['id']:
            return 'l'
        elif 0 <= nextxr < stat['size'][0] and 0 <= nextyr < stat['size'][1] and \
                        stat['now']['fields'][nextxr][nextyr] == stat['now']['me']['id']:
            return 'r'
        else:
            pass
    else:
        pass
```

```
# Hail Resistance!
# 在自己的领地内要坚决抵抗敌军入侵！！！
if stat['now']['fields'][stat['now']['enemy']['x']][stat['now']['enemy']['y']] == \
        stat['now']['me']['id']:
    # 如果我们在西边
    if stat['now']['enemy']['x'] > stat['now']['me']['x']:
        # 如果我们在北边
        if stat['now']['enemy']['y'] > stat['now']['me']['y']:
            # 如果东西向距离大于南北向距离
            if (stat['now']['enemy']['x'] - stat['now']['me']['x']) > \
                    (stat['now']['enemy']['y'] - stat['now']['me']['y']):
                # 就要转方向朝东
                if stat['now']['me']['direction'] == 3:
                    return 'r'
                elif stat['now']['me']['direction'] != 0:
                    return 'l'
            # 如果南北向距离大于东西向距离
            elif (stat['now']['enemy']['x'] - stat['now']['me']['x']) < \
                    (stat['now']['enemy']['y'] - stat['now']['me']['y']):...
        # 如果南北向相等
        elif stat['now']['enemy']['y'] == stat['now']['me']['y']:
            if stat['now']['me']['direction'] == 1:
                return 'l'
            elif stat['now']['me']['direction'] == 3:
                return 'r'
            elif stat['now']['me']['direction'] == 2:...
        # 如果我们在南边
        else:...
    # 如果我们在东边
    elif stat['now']['enemy']['x'] < stat['now']['me']['x']:...
    # 东西向相等
    else:...
```

```
# 回到领地后执行连续转弯动作
if not 'double_turn' in storage:
    storage['double_turn'] = 0
if 'double_turn' in storage:
    if storage['double_turn'] == 1:
        storage['double_turn'] = 2
        # 判断是否转弯后会撞墙
        if storage['turn'] == 'l':
            dir = (stat['now']['me']['direction'] + 3) % 4
        else:
            dir = (stat['now']['me']['direction'] + 1) % 4
        # 下一步的位置
        nextx, nexty = stat['now']['me']['x'] + \
                       directions[dir][0], stat['now']['me']['y'] + \
                       directions[dir][1]
        # 假如下一步会撞墙那么调转储存中的方向
        if nextx < 0 or nextx >= stat['size'][0] or nexty < 0 or nexty >= stat['size'][1]:
            if storage['turn'] == 'l':
                storage['turn'] = 'r'
            elif storage['turn'] == 'r':
                storage['turn'] = 'l'
        return storage['turn']
    else:
        pass
```

```
    if len(stat['log']) > 2:
        # 判断上一步是否在领地外
        last_field = stat['log'][-3]['fields']
        lastx, lasty = stat['log'][-3]['me']['x'], stat['log'][-3]['me']['y']
        if last_field[lastx][lasty] != stat['log'][-3]['me']['id']:
            storage['double_turn'] = 1
            # 调转方向
            if storage['turn'] == 'l':
                storage['turn'] = 'r'
            else:
                storage['turn'] = 'l'

            # 判断是否转弯后会撞墙
            if storage['turn'] == 'l':
                dir = (stat['now']['me']['direction'] + 3) % 4
            else:
                dir = (stat['now']['me']['direction'] + 1) % 4
            nextx, nexty = stat['now']['me']['x'] + directions[dir][0], stat['now']['me']['y'] + \
                           directions[dir][1]
            # 假如下一步会撞墙那么调转储存中的方向
            if nextx < 0 or nextx >= stat['size'][0] or nexty < 0 or nexty >= stat['size'][1]:
                if storage['turn'] == 'l':
                    storage['turn'] = 'r'
                else:
                    storage['turn'] = 'l'
            return storage['turn']

    return ''
else:  # 在field外farm field
    if storage['double_turn'] == 2:
        storage['double_turn'] = 0
    return farm(stat, storage)
```

```
    # Foresee 预见自己前进的路线是否已经被自己占过了
    if stat['now']['me']['direction'] == 0:
        sx = stat['now']['me']['x']
        myfield = True
        while sx < stat['size'][0] and myfield:
            if stat['now']['fields'][sx][stat['now']['me']['y']] != stat['now']['me']['id']:
                myfield = False
            sx += 1
        if myfield:
            if stat['now']['me']['y'] == 0:
                return 'r'
            else:
                return 'l'

    if stat['now']['me']['direction'] == 2:...

    if stat['now']['me']['direction'] == 1:...

    if stat['now']['me']['direction'] == 3:...
```

2.3 程序限制

本程序经多次调试、修改，基本上不会有报错现象发生。

3 实验结果

3.1 测试数据

实验环境说明：

硬件配置：Intel Core(TM) i5-3337U CPU @ 1.80GHz

操作系统：Windows 10

Python 版本：Python 3.6

最初先实现了 Farm 部分的功能，采用官方 AI wanderer 进行对战演练，测试其领地外圈地效率。之后添加了领地内的行为策略，采用新版本与旧版本之间进行对战演练，测试其领地内策略的效果。最后为了测试 Attack 函数，单独编写了绕场地一周的版本，用于调试其中的漏洞。在测试中，本组不断完善了如何避免撞墙与最大安全距离问题。

本组参加了三次热身赛。三次提交的代码中，都在测试其 Farm 部分的圈地效率功能，而没有加入 Attack 部分的函数。由于热身赛中其他组超时及报错较多，没有能够及时有效地反馈出圈地效率上的问题。三次热身赛结果如图 1，图 2，图 3 所示。

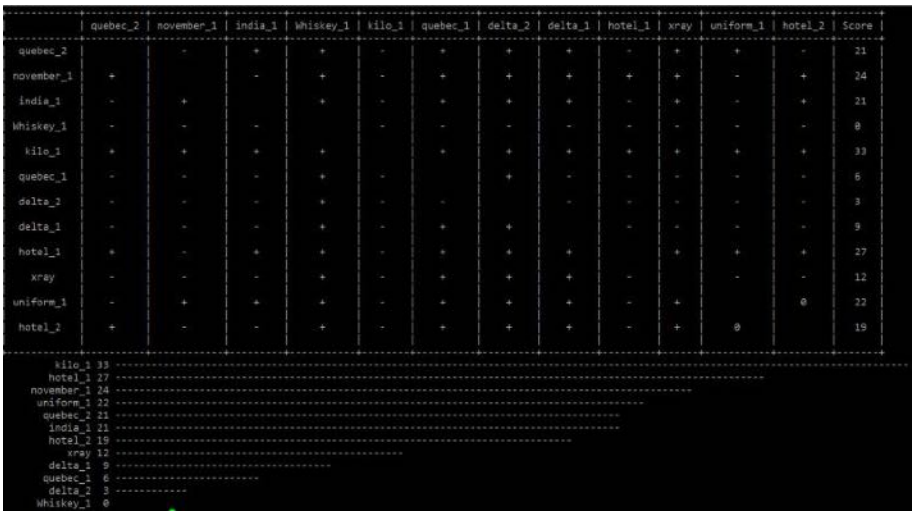


图 1 第一次热身赛情况及结果

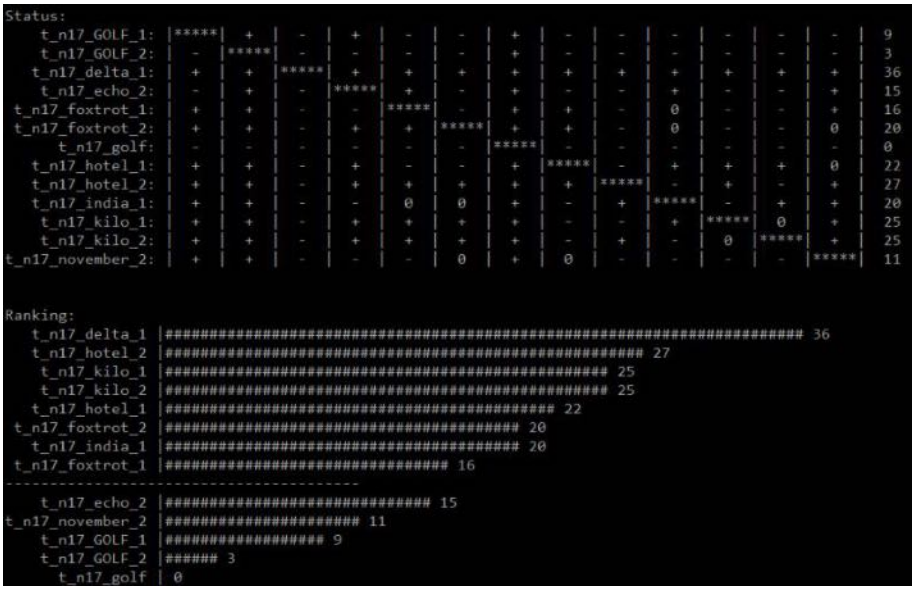


图 2 第二次热身赛情况及结果

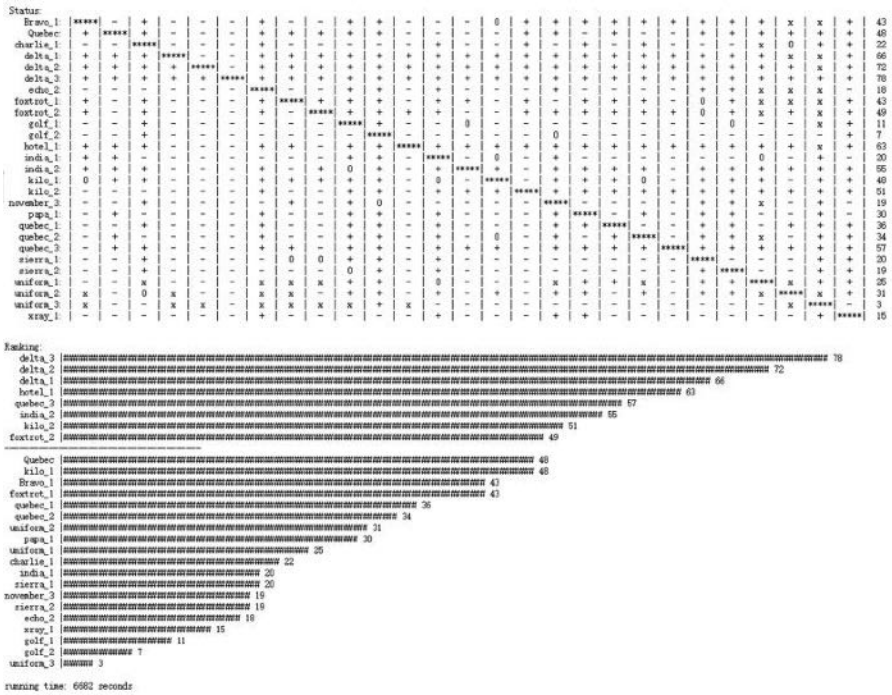


图 3 第三次热身赛情况及结果

3.2 结果分析

本组参加了三次热身赛，在其中互相切磋交流的同时，也加强了对自己策略的完善。

主要增添了领地内行为：

（1）增添”double_turn”行为，即回到领地时连续转两次弯出领地；

（2）考虑要抵抗敌方对自己领地的入侵，如果敌方在己方领地，则靠近敌方；

（3）优先考虑双方是否进入近距离纠缠，如果距离小于 5 则不出领地；

（1）、（2）两个增补增加了出领地方向的随机性，避免 AI 在一小片区域重复地圈地，在初代代码运行时回到领地后只会执行直走操作，导致有大量步数耗费在领地内无意义的行进上，改进后效率明显提升。

（3）是在观察了在两方都已经圈出较大面积之后在领地交接处缠斗后加入的规则。在对抗进攻型为主的 AI 中，起到自己在领地边界加强防守的效果（图 4、图 5）。在正式比赛中所有实力强劲的队伍无一例外地均采取了此种策略，注意到边界处会出现两方都不敢轻易出领地的僵局状态，直到步数耗尽以陷入僵局之前占有领地大小决出胜负。

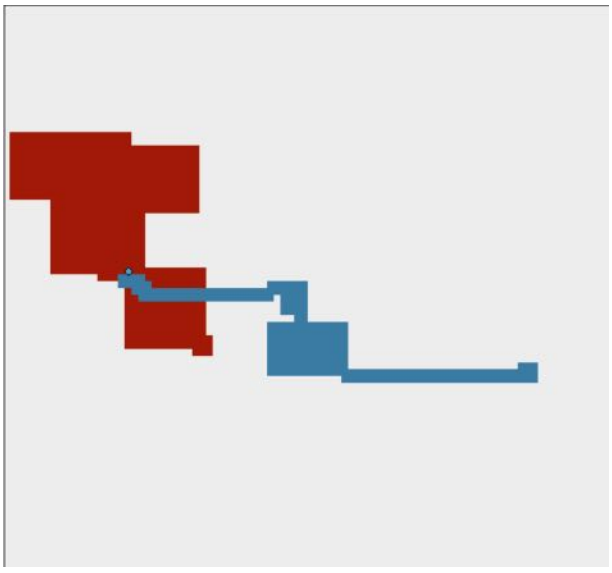


图 4 第一次热身赛中与对手 india_1 交战

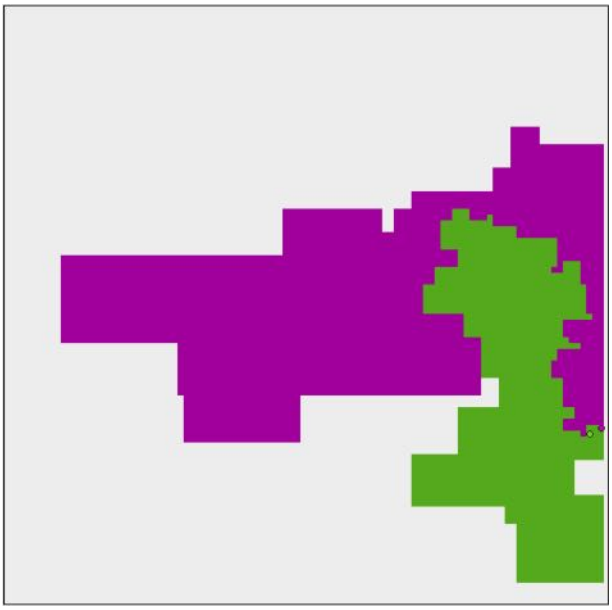


图 5 第二次热身赛与对手 delta_3 进行交界处缠斗

就圈地效率方面，本组采用了在前进途中观察对方行径，来扩大自己圈地范围的功能。由于没有突破正方形圈地模式限制，这一功能的收效甚微。之后又采用比较左转或右转两个方向圈地面积大小来优化决策方案。虽然能有效避免 AI 在边界打转，但对整体的圈地效率提高并没有太大帮助，而在遇到边界的地方走出狭长形的矩形也拖低了效率。

此外，就“领地外最大安全前进距离”组内也进行过多次探讨，经讨论后的长度能够保证自己的路线不会主动送给对方去击杀。

6 月 12 号的实战对弈中，本组一共进行了 95 场比赛，其中获胜 64 场，失败 31 场。获胜的场次中主动撞击对手 50 场，领地内被对手撞击 5 场，围地面积较大 2 场，对手自杀 6 场，对手撞墙 1 场；失败的场次中，被对手撞击 7 场，围地面积较小 14 场，自杀 10 场（均为纸带脱离领地导致自杀，示例见图 6）。

	主动撞击	被动撞击	围地面积	撞自己纸带	撞墙、出错
获胜	50	5	2	6	1
失败	0	7	14	10	0

通过表 1 中的数据，可以看出本组获胜的主要途径为主动撞击。说明本组的 Attack 部分可以达到令人满意的效果。而且 Attack 部分主要用于检测对方是否犯错，除非判断为可

以击杀，否则不会对圈地效率产生任何影响。

失败的原因中主要来自于打满 4000 回合后，围地面积小于对手。说明 Farm 部分的圈地效率还有较大的改进空间。有一点值得一提的是，除了围地面积小以外，被对手撞击均发生在对方领地内；其中所有的撞击自己纸带与大部分被对手撞击都是在己方纸带与领地脱离的情况下发生的；不会出现在地方领地主动撞击对手或撞墙报错的情况。这说明本组的 Farm 部分调控的行为过于保守，虽然能够保证自己不出错，但显得有些太过中规中矩。当对方具有完善的风险评估函数时，就难以取胜。

图 6 第二次热身赛中被 delta_3 切断与己方领地联系的局部图

这次实战中，我们共与 8 支队伍进行较量：Delta、Foxtrot、Juliet、Mike、November、Papa、Quebec 和 Uniform。他们的 AI 风格各异，策略上互不相同。所以在用时判断上，可以他们分别单独进行计算统计，再与总平均用时相比较。如表 2 所示，算法在运行所花费的时间并不多。经加权统计后，总共的平均用时为 0.10 s，平均步数为 2248 步，平均每步用时 44 ms。可见，虽然对战的对手不同，但平均每步耗时都差不多。其中较为特殊的是与 Papa 组的对战过程。双方多局比赛都是以 4000 回合结束的，但围地面积都不大，基本上对峙到一定程度之后就开始各自在领地内画圈，直到耗尽回合。

	Delta	Foxtrot	Juliet	Mike	November	Papa	Quebec	Uniform
平均用时/s	0.10	0.12	0.10	0.04	0.08	0.10	0.24	0.11
平均步数	2419	2619	1809	847	1427	2901	4000	2736
平均每步用时/ms	0.041	0.046	0.055	0.05	0.056	0.034	0.060	0.040

这个用时是在我们组预期之内的。其中主要运行时间开销花在自己出领地时对路线的规划，与对方在领地外时试探能否击杀对方。这也就是为什么与 Papa 对阵时用时尤其的短。

3.3 经典战局

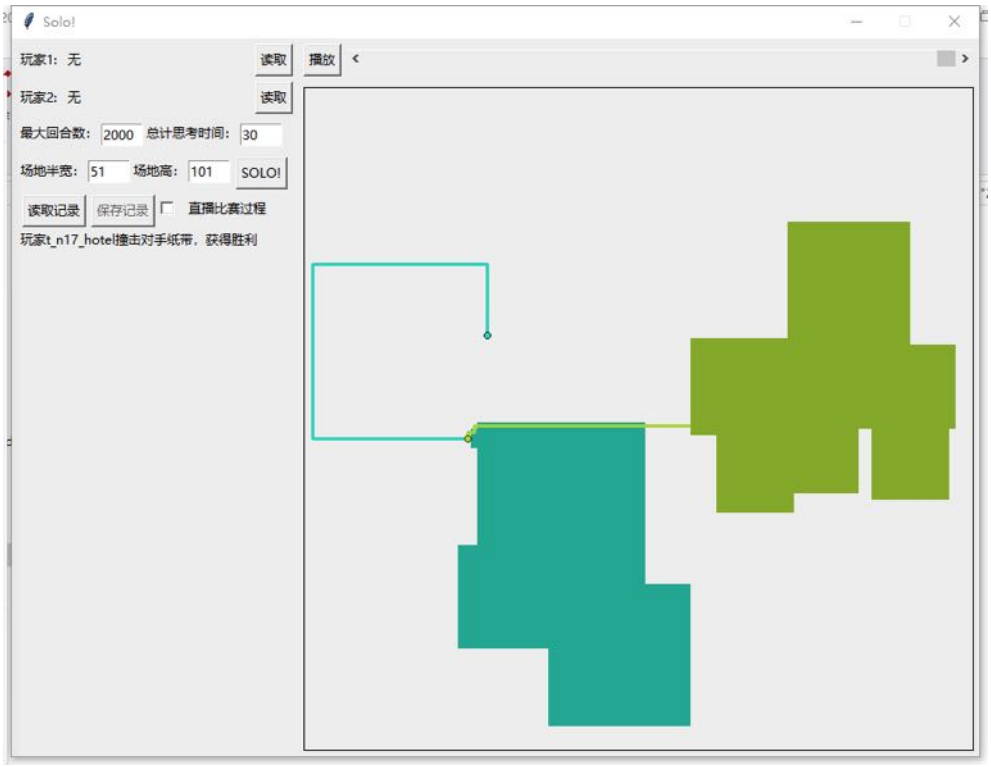


图 7 t_n17_Juliet-t_n17_hotel(2) 终局图

本算法的 Attack 部分在最后的实战中犹如严厉的戒尺。看见有人在面前皮，就要上去给他沉痛一击。图 7 是选自小组赛中本组 (Hotel) 作为后手与 Juliet 组对阵的第三局。Juliet 最开始就没有按照安全的长度来进行正方形围地，不过由于 Hotel 认为其也可以按

长方形提前回去，就没有采取进攻。而且 Hotel 与 Juliet 的纸卷的移动方向在大多数时候是垂直的，互相难以判断对方走向。而在 579 回合，双方纸卷进入同向，如果 Juliet 再犯错误就会难逃一劫。在 629 回合 Juliet 及时进入领地，化险为夷并将领地比分反超。在 664 回合，双方再次进入同向，而 Juliet 刚刚走出领地，并仍旧没有按照安全的长度来圈地。在 726 回合，Hotel 在自己领地里就发现对方的致命漏洞，直线冲刺，穿过对方领地，将对手击杀。

4 实习过程总结

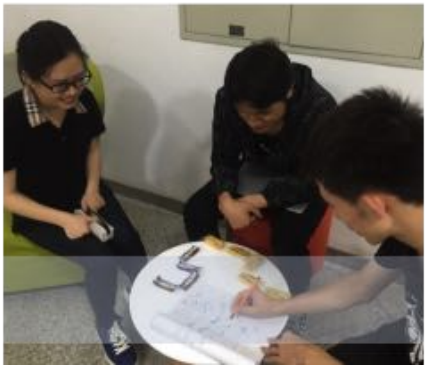
4.1 分工与合作

本小组由五位同学组成，分别担任自己部分的工作。程本夏同学担任会议的记录与实习报告的编写，李思伟同学担任 Farm 部分代码的编写，马艺铭同学担任 Attack 部分代码的编写，申景航同学担任 Farm 部分代码的编写、代码调试与整合以及实习报告的编写，周劲松同学担任比赛数据收集与分析。

会议内容：小组成员之间互相认识，推选申景航同学为组长；初步确定算法内容，并推演几种预想到的情况；初步 AI 设计，小组抢注，组员登记，投票选择“小猪佩奇”为小组头像。

会议内容：用两个 AI 参加热身赛之后，发现几个问题：撞墙角问题，领地内没能撞击对手，前进距离稍微大了一些，攻击性不足。经过进一步的讨论和图纸推演，确定解决方案后分工。

会议内容：针对热身赛中出现的圈地能力稍显不足和攻击力不足的问题进行讨论和推演，并交流解决办法



会议内容：针对测试过程中出现的在圈地过程中在后方被截断，Attack 函数出现撞墙问题以及在对战过程中如何识别对方的策略进行讨论和纸上推演。

4.2 经验与教训

组内同学分工明确，能够各自独立完成自己部分内容，而不需要耗费去等待别人进度。本组 AI 的 Attack 部分与 Farm 部分结构清晰，互不干扰，不存在接口上的混乱。而且组内同学代码编译时都有写注释的习惯，方便别人去阅读和理解，使工作效率大大提升。

对于本组 AI，我认为其采用 Attack 部分来击败对手，是在最终实战中主要的制胜手段。算法中将纸带线段化，再用纸卷位置与线段的矢量乘积来确定纸卷与纸带的距离，并找到距离最近的点作为进攻点。这种算法能够大大减少计算纸卷与纸带距离时所需遍历的数据量，提高时间效率。

相比之下，Farm 部分略有些平庸，在最终实战中略显保守。虽然不能让对方一举击杀，但面对一些冒险的圈地策略时，圈地效率上就会占据下风。而且如果对方有完善的风险评估，就很难找到漏洞去进攻。最终只能因圈地效率上的不足而输掉比赛。想要提高 Farm 效率就应该突破原有的圈地方式。我们在每局对战中所花费的时间大约只有 0.1 s 左右，说明还有很大的空间来进行尝试和探索。比如尝试一些复杂度更高的算法，来进行更大范围的规划和推演，从局部最优化圈地达到整体的最优化圈地。

4.3 建议与设想

本次实习作业中，采用回合制行动来进行比赛。其移动方式应是国际象棋中王的走法，而最好不要是中国象棋兵卒这样的走法。虽然简化了规则，使程序变得简单容易操作，但也同时限制了双方的行动策略。因为纸卷只能按照横竖走造成的结果是：在自己回合，双方纸卷的距离会始终保持为奇数或偶数（由开局的随机位置而定）。作为奇数的一方会有更大的选择空间，因为他在双方对峙时占有可以侧击对手的优势。而偶数一方则只能先进行圈地，争取圈地面积较大后再进攻。因为只要奇数一方应对正确，偶数一方进攻的结果只能是扑空或正撞。改为国际象棋中王的走法后，可以通过走三角来改变双方距离的奇偶性，来实现更多更复杂的策略。

5 致谢

非常感谢陈斌老师给我们这样一次知行合一的机会，在学习理论之余，完成游戏 AI 的编写是对我们学习的极大鼓舞，激发了我们的学习动力。非常感谢助教们不厌其烦的答疑，这让我们在学习过程中减少了很多阻力，让我们更好的掌握了学习的方法和技巧。同时也非常大作业中技术组同学，是他们的辛勤付出让我们在学习和大作业中才会有更坚实的基础和后盾，让我们的大作业更具有趣味性和实用性。还要感谢 F17 India 小组和我们课下一起进行友谊赛对战，给我们提供了更多的算法思路和反思。最后，我们非常感谢小组内所有成员的协助，一起在算法讨论、程序编写、算法测试、论文撰写中共同努力，尤其是算法编写测试的大佬们还要熬夜编程测试，但最后 N 区第四的成绩也让我们非常珍惜，珍惜这份来之不易的成果和回忆！

6 参考文献

1.Hcormen T, Eleiserson C, Lrivest R, et al. Introduction to Algorithms(Third Edition)[J]. Computer Education, 2013, 39(4): 1292.

2.Miller B W, Ranum D L. Problem Solving with Algorithms and Data Structures Using Python[J]. Braz.j.biol, 2006, 70(2012):483-490.

第九章　N17_India 报告

张楚珩 * 王依琛 徐枫 杨舒雅 叶一帆 张宇博

摘要：本小组 India 在小组循环赛中拿到并列第一的成绩。本文主要介绍我们攻击型 AI 的战略思维，我们使用的算法结构以及在完成实习过程中，总结的经验和教训。在代码中主要使用到了 python 的字典和列表等数据结构，遍历全局以及距离判断的算法。欠缺点在于忽视了圈地策略，而将所有运算资源放在攻击上，导致最终因为圈地面积过小而没有出线（虽然小组循环赛中成绩与出现的小组比分相同，仅负一场）

关键字：纸带圈地 遍历 攻击性策略

1 算法思想

1.1 总体思路

在离开领地时，状态为“远征”，即向敌人方向前行。若侦测到敌人纸带则向纸带运动进行进攻；超过安全距离后，将远征模式变为“回家”，返回领地，开始下一次远征。

主要利用 if 条件判断语句、for 循环语句进行单步内的遍历与条件判断。主要运用的数据结构为字典和列表。利用字典和 key-value 从游戏函数 stat 中读取相关数据，利用列表记录信息。

采用贪心策略，在“远征”模式下每一步都向对手接近并侦查对手纸带动向，如果能击杀则毫不犹豫；“回家”模式下，每一步判断是否到家，向着家的方向专心回家。

首先，我们搭建了最基本的向某个方向前进的函数，随后搭建了侦查敌人纸带方位的

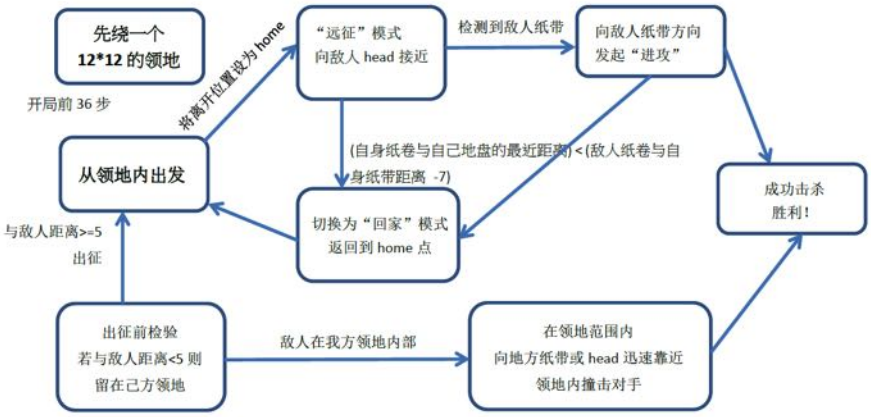
函数，以及防止自杀的模块。在与敌人距离大于 40 时，我们处于“远征”状态，即向着敌人方向移动。此时如果侦测到敌人纸带，则向敌人纸带运动进行“进攻”。若离敌人距离小于 40 且还没有进攻机会，则切换到“回家”模式，向着领地内的 home 点运动。

防守策略，在与敌人距离小于 20 时，执行“回家”策略避战，返回自己的领地；若此时我们已经身处自己的领地，则向着敌人的方向前进，实现“领地内撞击”实现胜利。

但与敌人距离小于 40 时，我们的纸带可能已经拖得很长了，非常危险。故改进中，我们加入了“远征”距离的限制，离开自己的领地超过 80 时，就将模式切换为“回家”模式。

而回家函数 home 的位置是在离开领地时将离开领地的点确定为 home。这样返回时虽然效率高，但是圈地面积效率低。

1.2 算法流程图



1.3 算法运行时间复杂度分析

主要的时间开销花在 scanning 函数上，将整个场地进行遍历。时间复杂度为 O(n*m)，n, m 分别为场地的长度和高度。

2 程序代码说明

2.1 数据结构说明

算法中主要采用了字典、列表等 Python 自带的数据结构。没有使用自定义类、树、图等结构。

2.2 函数说明

1. load 函数：每次比赛开始初始化 init 和 step，并将其存入 storage 中

2. distance 函数：测量两点之间的距离，在后面的函数中会用到

3. distance_with_enemy 函数：测量己方纸卷与对方纸卷之间的距离，用于进行策略的选择和规划

4. minDistance 函数：计算从从“一个点”到“一个列表里的所有点”的最小距离，比如己方纸卷到对方纸带的最小距离等

5. returnhome 函数：通过判断己方纸卷与己方领地的相对位置，选择一条最近的路回到己方领地中；若已在己方领地中，则将策略改为远征

6. attack_enemy_band 函数：计算己方纸卷与对方纸带的最近距离，选择最短路径攻击对方纸带

7. myBorder 函数：输入一个点的坐标，判断这个点是否为我方领地边界

8. enemyBorder 函数：输入一个点的坐标，判断这个点是否为对方领地边界

9. nextPos 函数：根据己方纸卷的当前方向，预判己方纸卷的下一步的位置

10. toward_enemy 函数：判断对方纸卷与我方纸卷的相对位置，选择最近路线靠近对方纸卷

11. headeast 函数：控制我方纸卷向屏幕东部移动

12. headwest 函数：控制我方纸卷向屏幕西部移动

13. headnorth 函数：控制我方纸卷向屏幕北部移动

14. headsouth 函数：控制我方纸卷向屏幕南部移动

15. scanning 函数：扫描整个场地，记录对方纸带和领地边界、我方纸带和领地边界的位置信息，储存在相应的列表中

2.3 程序限制

暂时没有发现程序报错情况。

3 实验结果

3.1 测试数据

实验环境说明：

- 硬件配置：（CPU/内存）

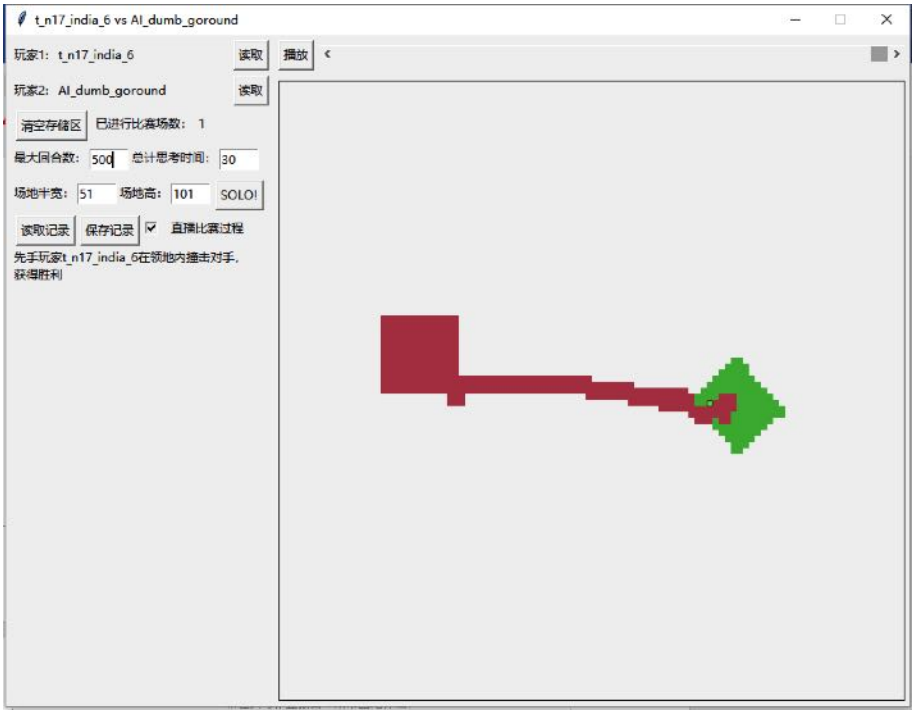
- 操作系统：（名称/版本）

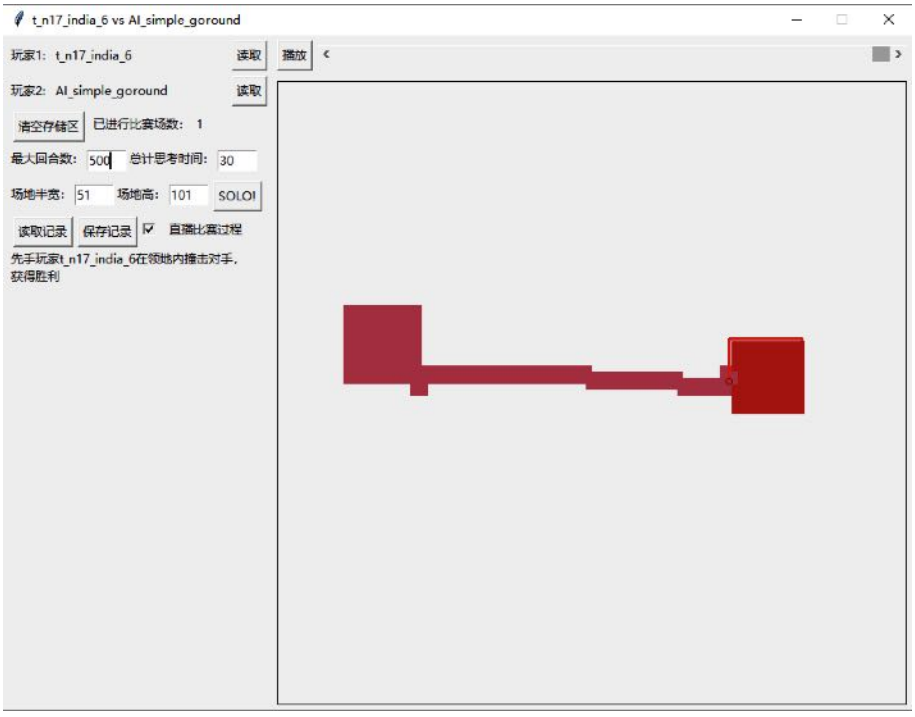
- Python 版本：（版本号）

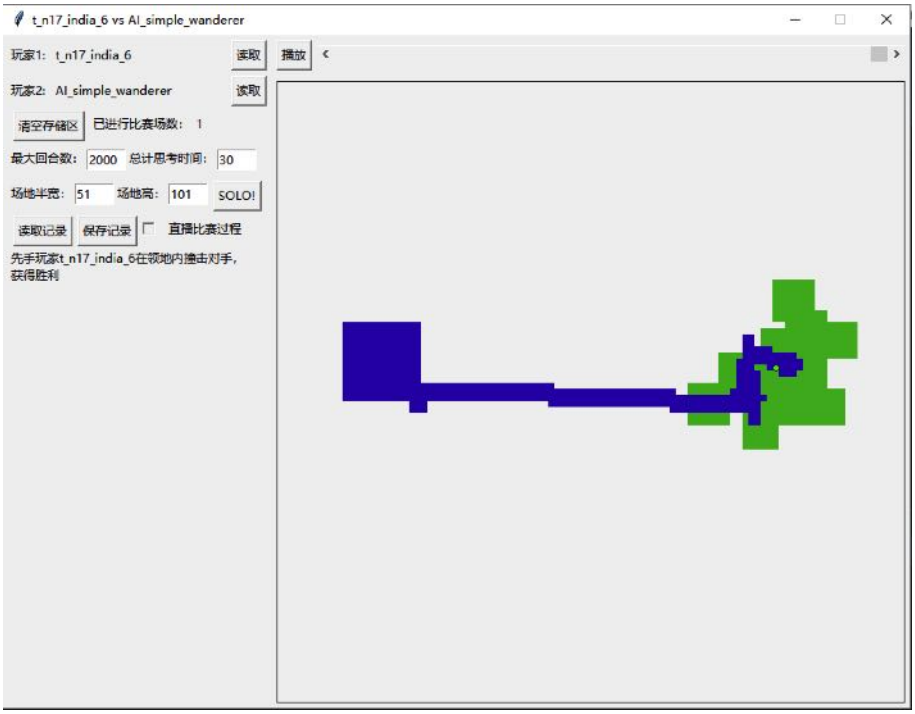
3.6

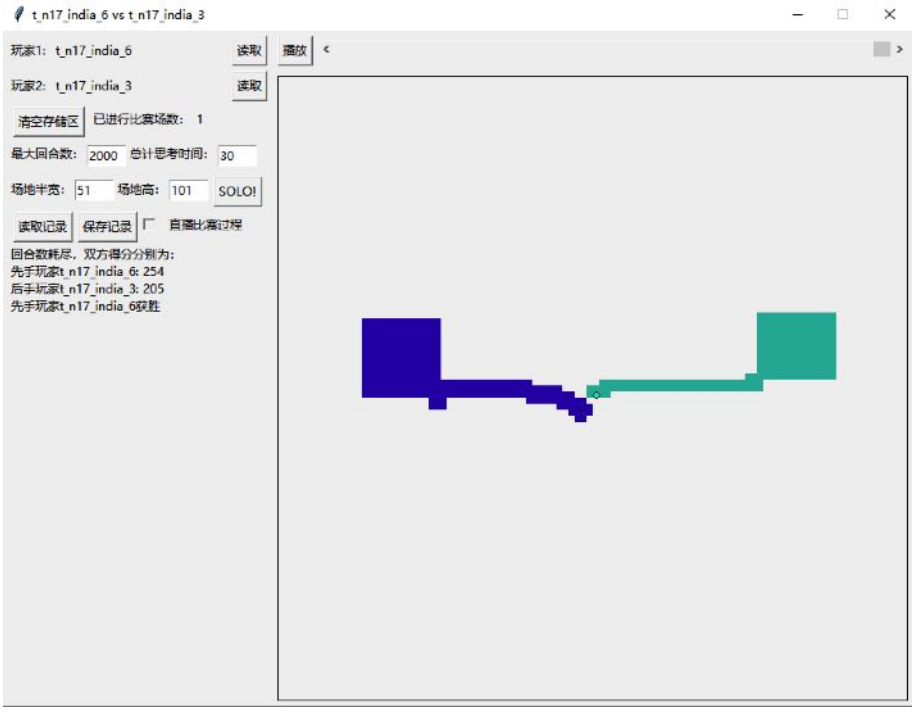
在前期，我们将自己开发的 India 系列 AI 与课程提供的各种测试 AI 进行比赛，来调试我们 AI 的性能，对其不断进行完善。到了后期，课程提供的简单 AI 不能满足我们的测试需求了，我们就将各个不同版本号的 India 系列 AI 进行比赛，进一步提高其表现。

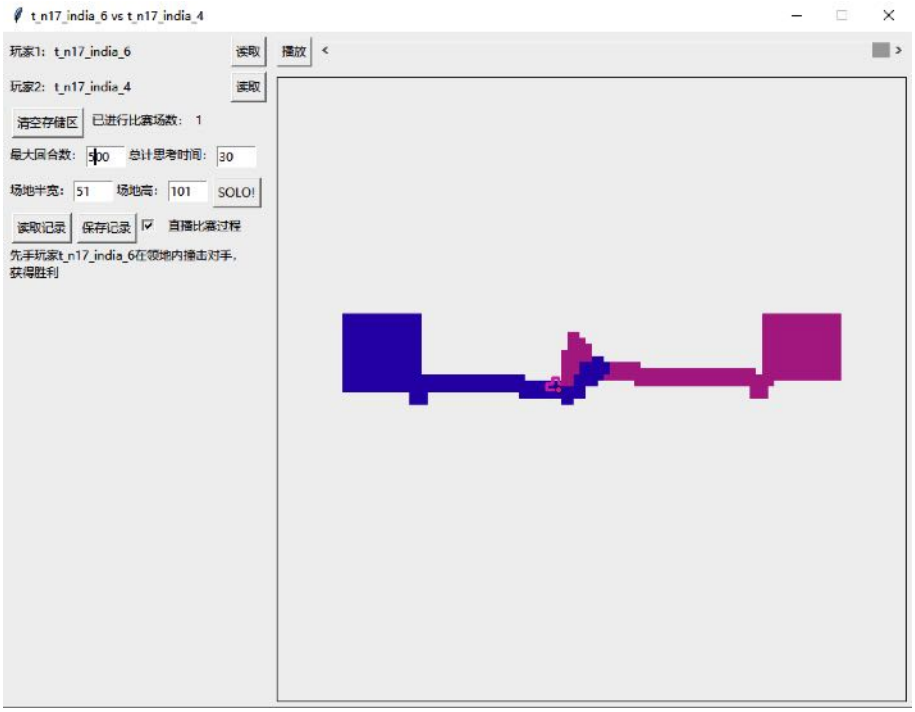
部分测试结果如下：


t_n17_india_6 vs AI_dumb_goround
玩家1：t_n17_india_6
读取
播放
玩家2：AI_dumb_goround
读取
清空存储区
已进行比赛场数：1
最大回合数：500
总计思考时间：30
场地半宽：51
场地高：101
SOLO!
读取记录
保存记录
直播比赛过程
先手玩家t_n17_india_6在领地内撞击对手，
获得胜利


t_n17_india_6 vs AI_simple_goround
玩家1：t_n17_india_6
读取
播放
玩家2：AI_simple_goround
读取
清空存储区
已进行比赛场数：1
最大回合数：500
总计思考时间：30
场地半宽：51
场地高：101
SOLO!
读取记录
保存记录
直播比赛过程
先手玩家t_n17_india_6在领地内撞击对手，
获得胜利

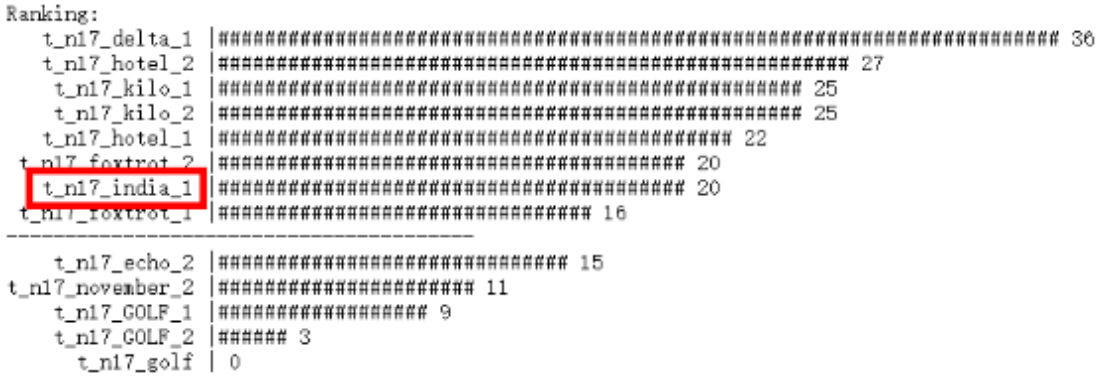

t_n17_india_6 vs AI_simple_wanderer
玩家1: t_n17_india_6
读取
播放
玩家2: AI_simple_wanderer
读取
清空存储区
已进行比赛场数: 1
最大回合数: 2000
总计思考时间: 30
场地半宽: 51
场地高: 101
SOLO!
读取记录
保存记录
直播比赛过程
先手玩家t_n17_india_6在领地内撞击对手，
获得胜利


t_n17_india_6 vs t_n17_india_3
玩家1: t_n17_india_6
读取
播放
玩家2: t_n17_india_3
读取
清空存储区
已进行比赛场数: 1
最大回合数: 2000
总计思考时间: 30
场地半宽: 51
场地高: 101
SOLO!
读取记录
保存记录
直播比赛过程
回合数耗尽，双方得分分别为:
先手玩家t_n17_india_6: 254
后手玩家t_n17_india_3: 205
先手玩家t_n17_india_6获胜

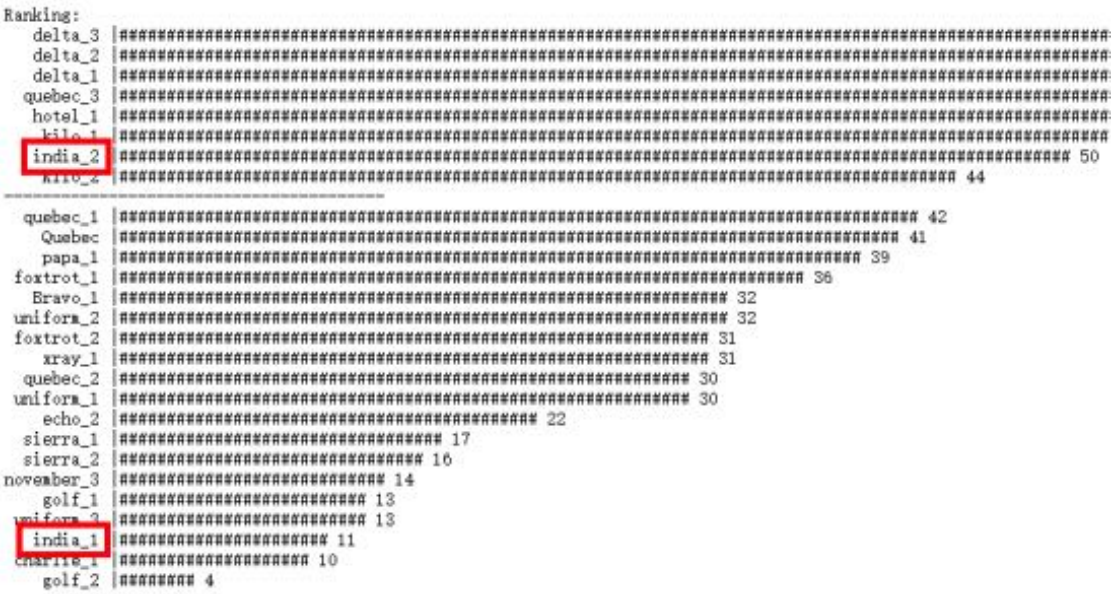


第一次热身赛结果：India_1 胜 21 场，在参赛的 12 个 AI 中排第六。

第二次热身赛结果：India_1 胜 20 场，在参赛的 13 个 AI 中排第七。



第三次热身赛结果：India_1 胜 11 场，在参赛的 27 个 AI 中排 25；India_2 胜 50 场，在参赛的 27 个 AI 中排第七。



可以看出，随着热身赛的进行，参赛 AI 不断增加，但是我们的 AI 一直能保持第七名的成绩，十分稳定。

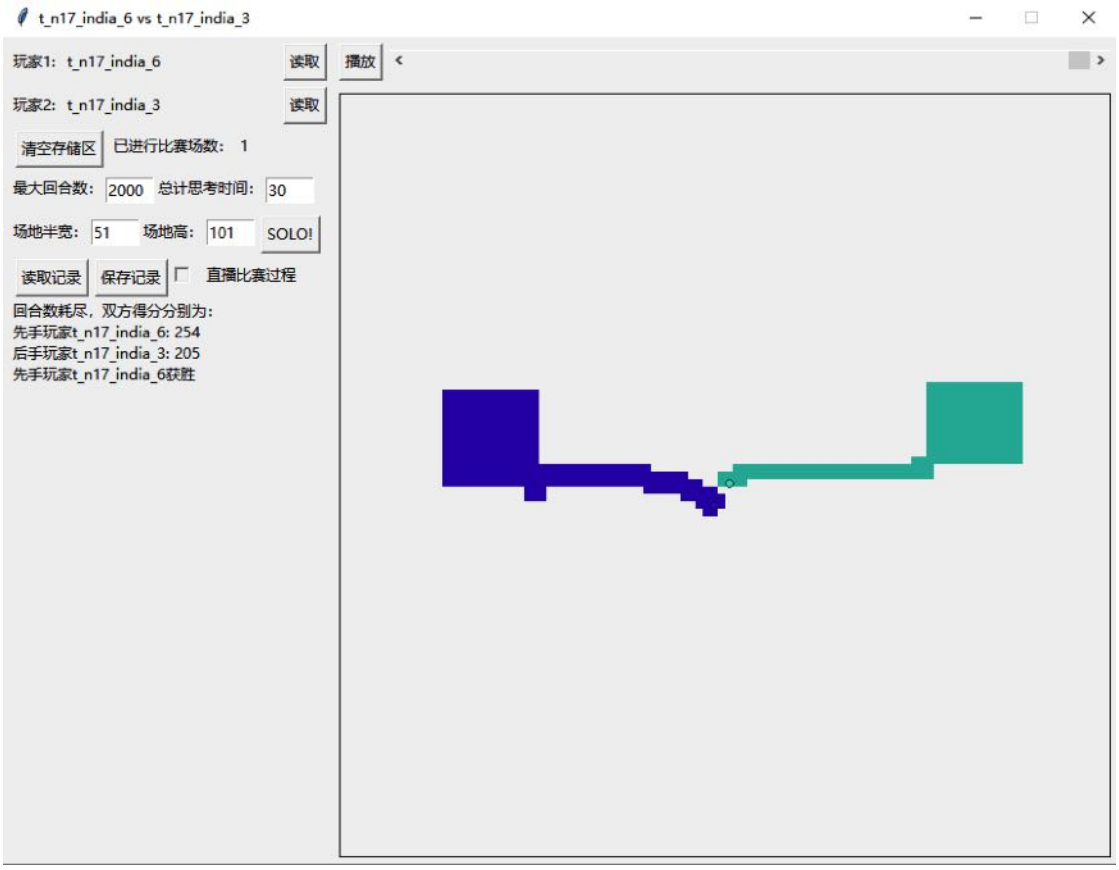
3.2 结果分析

本算法属于攻击型算法，主要策略是攻击对方纸带或在己方领地内与对方纸卷相碰。对于一般的圈地型算法，本算法能有效地对其进行攻击；但是面对和自己同类型的算法时，

可能会出现双方在原地对峙，陷入僵局的场面。在这种情况下，往往是耗尽了回合数，根据圈地面积判断胜负，这时就暴露了本算法的劣势。由于圈地不是我们的优先考虑，算法圈地面积很小。在最后的正式比赛中，也正是由于这个原因，我们组尽管和其他两个小组并列小组第一也没能出线。

在运行时间方面，本算法的运行时间在预期内。主要的时间开销发生在对整个场地的扫描上。

3.3 经典战局



在两个不同版本的 India 进行对战时，我们发现有时会出现上图所示的情况，即双方纸卷各自在一块 2*3 的领地内转圈，僵持不下，直到对战结束，圈地大者获胜。

4 实习过程总结

4.1 分工与合作

我们是生命科学学院 15 级老鲜肉组成的特殊 6 人团队。

本次纸带圈地大作业中，由组长张楚衡负责搭建代码的基础框架与基本功能，王依琛负责进攻、防守策略的实现，张宇博负责防自杀功能的完成，杨舒雅负责策略提供与代码优化，徐枫、叶一帆负责策略提供与论文撰写。

在完成这个比赛的过程中，我们进行了 3 次线下讨论，多次线上讨论，时刻在微信群内协调沟通。组员们在组长的组织下，积极参加讨论，每次讨论至少 5 人参加。

4.2 组会记录

第一次组会：

接到课题后初步理解、熟悉环境，讨论后续分工。是否采用深度学习，或是人脑逻辑。本次组会时间为 5/25

第二次组会：

否定了深度学习方案，对于初步编写的 AI debug，利用构建起的基础框架实现战胜傻瓜 AI 的目标。组会之后实现了一个适于热身赛的基础 AI。本次组回时间为 6/1

第三次组会：

正式比赛前，讨论算法改良方案。优化攻击代码，解决容易自杀的问题，增加了初始圈地函数。并且添加了扫描全局的函数。使得「攻击」与「回家」方案更为合理。本次组会时间为 6/8

多次线上交流：

11:26
（未来的）数...作业小组(7)
这是目前写出来的AI，有攻击对手、远征、回家、伺机而动等能力
可以打败原始AI以及绕培壁走的AI
你们可以玩玩看，顺便帮我增加一下防止自杀的功能
参数可以帮我调整一下
我还没仔细看你们的攻略，参数可以按照攻略调整
好 qvq
微信(5)
苜蓿
啊，我想到了，等我一下
可是想不出怎么回溯路线.....
我的想法比较简单，只能得出回家需要多少步
感觉3还不如直接修复了BUG的2效果好
t_n17_india_2.py
12K
修复完之后跟fox还能五五开...?
微信(5)
杨舒雅
and的左边是判断field在十字搜索内
右边是给你一个数值
ok我懂了
关键是左边那个是吧qvq
okok
我看看
我发现了一个bug
nearest_my_field(stat, storage)不能对西和北做出反应
我再改一下

4.3 经验与教训

大作业的讨论氛围增进了同学们的友谊，同时我们享受了新装后地学楼奢华先进的讨论环境。在思维碰撞中，我们的 India 共迭代了 7 个大版本。我们一同见证着它从不停报错，到 India 1,2,3 疯狂撞墙或自杀，再到 India 4,5 可以稳健得行动，最后到 India 6,7 的虎虎生威。

我们的 India 是一个极具攻击性的 AI，战斗风格属于绝食性，即不围地也要战斗！每一次出征都是为了杀敌，但是遇见危险又回迅速逃回家，显得非常可爱。但由于我们的圈地效率不高，赛制原因，在组内 3 名平局的情况下，India 遗憾没有出线。

圈地效率低是我们的主要问题。因为我们采用的策略是从领地内出发时，会将 home 设定为离开领地很近的一点，而后续圈地策略都围绕回到 home 点实行圈地。这样出现的问题就是每次围成的新领地为长条状，走相同的距离，正方形围地的效率远高于我们。同时，由于 home 点与纸袋离开领地的点相距过近，我们会出现如下的（蛇形）bug 状态：

即直接返回 home 时会撞向自己的纸袋，这时候防自杀模块会让 head 转弯，但是进程上又需要及时返回 home 处，故回家函数与防自杀函数冲突产生了蛇形状态。但这也与我们的回家函数考虑较少有关系。我们只是简单得条件判断，当 head 坐标与 home 坐标的 x 或 y 相同时，就向该方向直线返回。并没有考虑到 home 位置特殊时需要绕路、转向的情况。

产生上述 bug 的原因还有 India 的智商问题。我们只涉及了一步之内需要做什么判断，怎样处理，没有长远的规划。所以 India 自身并没有做错，每一步都在依照环境努力判断着自己的行动，但是每一步都是正确的不代表结果是好的。如果能多步考虑，路径的选取、回家规划，我们就不会因为路径的随机性而让自己走出不合理的蛇形路径。

同时，我们的 home 函数太过低效，不能结合领地形状给出最高效的纸带返回点，细长的领地付出了大量步数与危险系数，得到的回报却较低。一旦我们不能杀死对手，我们

的领地面积就会迅速落入下风。

代码中所用到的数据结构较为简单，没有高级搜索算法，也没有动态规划的过程。所以不能实现返回位置的变化。Home 点在离开领地是就确定并且不会再改变，这点不利于灵活的进攻与防守。但由于我们的进攻非常直接，且程序优化很好，即使每步前需要遍历全地图，我们的 India 也不会超时。

4.4 建议与设想

虽然明年的大作业会采取不同的游戏，但在此对本次大作业提出改进建议。

1. 大作业的题目选定最好能提前一些。这样搭建平台的同学不会太仓促，平台的更新、代码框架的调整对参赛同学与开发同学都是不小的负担。开发同学十分辛苦，却还要被一些同学提出质疑。以及有些同学认为的便利工具在平台上线初期还未开发出来，且代码框架的更新又使同学们的代码产生了新的问题等等。如果题目尽早选定，平台充分搭建完成后，同学们再开始大作业的编写，这样会避免许多问题的发生。

2. 赛制的评定规则上，纸带圈地有圈地面积、战胜回合数两个指标。平局时，仅以圈地面积作为评定标准可能不太全面。希望明年有考虑更加周全、更综合的规则。

3. 赛制的分组上，如果不巧与冠亚军分在同一个组里，可能非常绝望。或许未来的季军就被淘汰而无法出现。建议明年可以增加一个“复活赛”的大混战环境，为那些竞争过于激烈的强组成员提供复活机会。小组前 2 出线，第 3 获得复活赛资格，复活赛内又可以有 2 位复活参加决赛资格。这样没有增加太多工作量，但是给予厉害的小组第二次机会，总体上使得赛制更加合理，减少同学们的遗憾。

希望学弟学妹们能在明年的数算课画出更美的二叉树作品，同时在期末大作业中享受自我，写出独步武林的最强 AI。

关于明年的大作业设想，是否可以采取拳皇对打类的 AI 对战。战斗一定非常激烈精彩。可以简化成火柴人的模式，有连招、跳跃、格挡、移动等基本操作。地图可以是简单的打斗环境，也可以是有纵轴的大地形环境。限定时间内 KO 对手或者血量高者获胜。故策略上可以选择血量占优后就避战拖到战斗结束。

5 致谢

感谢陈斌老师上课时精彩的讲解，让我们入门了 Python 语言。感谢易超、陈旭、赵宇、冀锐、缪舜、滕沅建、张颢丹六位助教，不辞辛劳为我们解答疑惑。感谢技术组的陈天翔同学和张赖和同学，给我们搭建了这么好用的对战平台。感谢班上其他同学优秀的 AI，让我们认识到了与大佬的差距。

6 参考文献

[1]https://www.reddit.com/r/paperio/comments/5ip6ct/how_to_get_100_in_paperio_strategy_and_walkthrough/

[2]https://www.gamezebo.com/2017/01/17/paper-io-tips-cheats-strategies/

[3]https://github.com/AppliedDataSciencePartners/DeepReinforcementLearning/blob/master/funcs.py

[4]https://blog.csdn.net/cs123951/article/details/77560074

第十章　N17_Juliet 报告

章昊、童乐、邹运佳、官焕钦、刁一飞 *

摘要：本算法的总体思路分为以下四个部分，包括领地内游走，领地外圈地和发动进攻和状态转换模块。面对比较保守的对手，本算法靠圈地进行获胜；面对比较贪心的对手，本算法靠直接进攻获胜。本算法在测试环节表现良好，击败了所有的 AI 算法，在分组竞赛环节获得了小组第四。

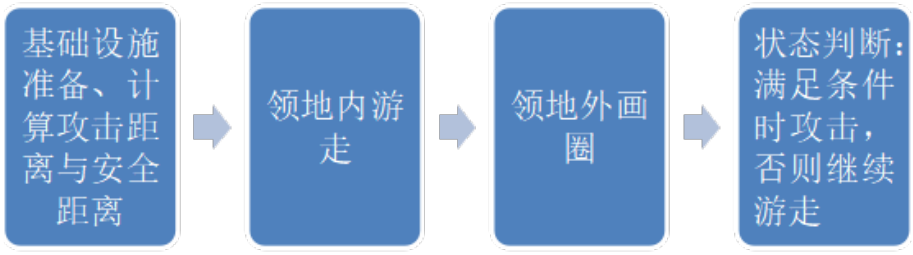
关键字：字典 一击必杀

1 算法思想

1.1 总体思路

本算法的总体思路分为以下四个部分，包括领地内游走，领地外圈地和发动进攻和状态转换模块。面对比较保守的对手，本算法靠圈地进行获胜；面对比较贪心的对手，本算法靠直接进攻获胜。本算法主要采用了字典和列表等数据结构。最开始我们想要设计一个攻击性较强，扫描范围较大的算法，但是发现时间过长，收益较低，难度较大，因此我们简化了算法，对扫描范围进行缩小，加强了圈地功能。

1.2 算法流程图



1.3 算法运行时间复杂度分析

本算法运行时间主要由 20 X 20 的扫描步骤消耗，复杂度为 O（1），在实际运行中时间开销仅为 0.01s。

2 程序代码说明

2.1 数据结构说明

算法采用的主要数据结构为字典和列表，未使用自定义类。

2.2 函数说明

本算法主要使用了 play，load，wander，square，goback，attack 等函数，play 函数用来实现储存功能，load 函数用来实现调用功能，wander 函数实现领地内游走，square 函数实现领地外圈地，goback 函数实现返回领地中心，attack 函数实现攻击功能。面对比较保守的对手，本算法靠圈地进行获胜；面对比较贪心的对手，本算法靠直接进攻获胜。在距离较近时，进行 20X20 范围的扫描，如果发现我们离纸带的距离小于敌人里基地的距离，则发动直线进攻，一击必胜；如果发现并不满足条件，则继续游走。

2.3 程序限制

在刚出基地时，纸带会进行一步随机行走，如果碰到边界，则会出错；在概率极小的情况下，如果没能对敌人实现一击必杀，则会直线撞墙。

3 实验结果

3.1 测试数据

实验环境说明：

硬件配置：（CPU/内存）Intel（R）Core（TM）i5-4200U @ 1.60GHz 2.30 GHz/4.00 GB

操作系统：（名称/版本）WIN7 旗舰版 64 位操作系统

Python 版本：（版本号）3.4

采用了技术组提供的 AI 算法进行测试，将部分算法直接击败，另外的算法在步数耗尽之后，以圈地面积较大而获胜。

3.2 结果分析

本算法的总体思路是分为以下四个部分，包括领地内游走，领地外圈地和发动进攻和状态转换模块。面对比较保守的对手，本算法靠圈地进行获胜；面对比较贪心的对手，本算法靠直接进攻获胜。在距离较近时，进行 20X20 范围的扫描，如果发现我们离纸带的距离小于敌人里基地的距离，则发动直线进攻，一击必胜；如果发现并不满足条件，则继续游走。

算法运行时间很短，在预期之中，仅为 0.01s，主要的运行时间开销发生在扫描过程。

3.3 经典战局

一、与 AI_dumb_random 算法比赛中，对手直接自杀。

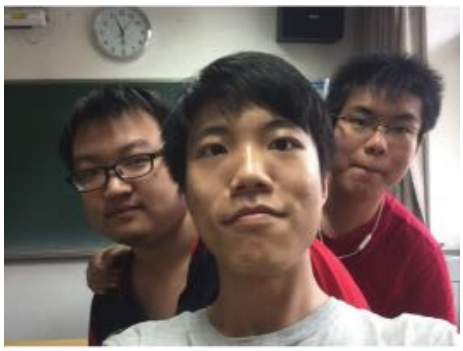
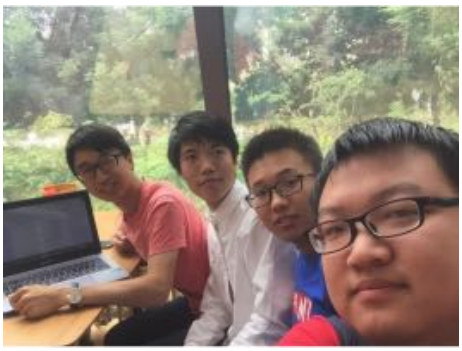
二、与 AI_normal_wanderer 算法比赛中，两个算法执着于圈地，最终本方以圈地面积 5195:4333 获胜。

4 实习过程总结

4.1 分工与合作

章昊、童乐、刁一飞负责写算法；邹运佳、官焕钦负责写大报告。

小组主要通过微信群聊和线下开会的方式进行合作交流。

4.2 经验与教训

在本次实习中，小组时间安排得不太合理，导致许多设想的算法功能没有完成。我们算法的攻击模块设计精妙，思路直接，耗时较短，扫描范围可调，并且可以实现一击必杀。但是我们的算法对于那种集稳定圈地与强大攻击的模块较为无力，后续可以在这方面进行改进。同时没有考虑到攻击失败的后续操作，因此有可能会导致直线撞墙，可以加入攻击失败的后续程序。

4.3 建议与设想

开发组可以考虑提供更为强大，全面的 AI 代码，以免发生自杀现象。同时竞赛可以采取全循环或者设立分档次抽签分组的方式。希望明年学弟学妹们合理安排实习时间，不要好高骛远，要脚踏实地合理设计代码。我们认为我们的代码在实现后续修改后可以变得更

为强大。

5 致谢

感谢技术开发组提供的基础代码，感谢助教们的耐心指导，感谢陈斌老师对于我们的鼓励与支持，感谢全体组员的齐心协力。

6 参考文献

1. http://gis4g.pku.edu.cn/course/pythonds/

2. https://github.com/chbpku/paper.io.sessdsa

第十一章　N17_Kilo 报告

欧阳萌淞 *、耿宗泽、刘书颖、张欣然

摘要：本算法利用列表、字典、栈和图四类数据结构，实现对当前对战局势信息的记录、对各类动作触发条件中所用参数的计算以及根据参数进行下一步动作的决策。在记录对战局势信息时，本算法在每回合开始后，通过调用上一回合的领地和纸带归属情况，根据条件判断是否需要更新领地和纸带归属情况，从而降低算法复杂度。在判断各类操作触发条件时，本算法计算了我方到敌方纸带距离、我方到敌方潜在回家路径的距离、敌方到我方纸带距离、敌方到我方潜在回家路径距离、我方回家距离、敌方回家距离等条件。根据判断条件，本算法实现包括追击敌人、跑回自己领地、圈地在内的三种操作行为。

关键字：评估函数；经验预判；历史数据分析；列表；字典；栈；图；A* 算法

1 算法思想

1.1 总体思路

1. 算法的总体思路

本算法总体来说由三大部分组成，分别为记录当前对战局势信息、计算各类动作触发条件中所用的参数以及根据参数决策下一步动作。

第一部分是每回合开始时记录当前场地信息。具体来讲是调用 storage 中的上一回合的领地和纸带归属情况，根据一些条件判断是否需要更新，如果需要则进行更新，然后将新的领地和纸带归属情况再存回 storage 中。

第二部分是调用函数来计算判断各类动作触发条件时所用的参数。具体来讲需要判断我方到敌方纸带距离、我方到敌方潜在回家路径的距离、敌方到我方纸带距离、敌方到我方潜在回家路径距离、我方回家距离、敌方回家距离等条件。

第三部分是用以上关键参数来看是否满足各类判断条件，最终决定是否要进行包括追击敌人、跑回自己领地以及圈地三种行为。

2. 采用的主要数据结构与算法

本算法主要采取了列表、字典、栈和图四类数据结构。算法中用到的较为复杂的算法包括启发式寻找有障碍下最短路径的 A* 算法，但算法主体还是各类条件语句和循环语句的灵活应用。

3. 算法实现过程

我们的算法主要经历了三个阶段的设计和主要调整。

第一阶段是算法整体思路设计阶段，是从开始进行算法准备到第一次热身赛之前（约 5 月 26 日）。我们从最开始即确定了算法的基本逻辑为上述的三大部分，即记录局势信息、计算关键参数和根据参数触发各类执行条件。在这一阶段，我们完成了以下重点环节的设计：

（1）按每个点遍历记录纸带和领地归属

（2）遍历上述归属信息，求最小值计算以上各类距离参数的函数，同时确定最短回家或击杀路径

（3）按照简单版的动作触发逻辑选择待执行动作

（4）圈地函数、击杀函数和回家函数

（5）动作执行时的基本防撞墙和防撞纸带操作

基本此时我们的版本即属于可以进行实战的版本了。

第二阶段是对上述判断条件和算法复杂度的优化阶段，重点在于减少判断时的逻辑漏洞，以保证做到回家、击杀和圈地的动态决策调整，同时在此基础上改进了数据结构以减少遍历。该阶段是在两次热身赛之间（5 月 28 日到 6 月 3 日），主要改进了以下部分：

（1）重新梳理之前的判断逻辑，目标是减少执行中可能出现的各类逻辑漏洞，做到动

作内流畅运行、动作外能按合理条件切换

（2）防撞墙和防撞纸带条件的进一步改进，具体来说之前版本中单一动作执行中防撞墙和防撞纸带运行效果好，但在动作切换过程中容易出现决策失误，该版在这些步骤上做了进一步改进

（3）用栈的数据结构存储纸带信息，通过调用上局存储记录，看当前位置是否是己方/敌方纸带，如果是则将该位置入栈，否则清空栈，由此避免了遍历整个场地，降低算法复杂度

（4）通过判断己方和敌方纸带栈是否被清空来判断是否需要重新遍历场地记录领地信息归属（即如果两者纸带都在延长则说明未发生领地变化），由此大幅降低遍历次数；该点和上一点结合解决了每次要遍历地图的操作，将运行时间缩短了约 1/2 到 2/3

该阶段结束后大大减少了可能出现的超时问题，同时出现非预期决策导致死亡的情况也有了明显的减少。

第三阶段是针对上一轮热身赛暴露的问题进一步改进击杀和回家函数，同时增加了在双方边界处和对方领域内的执行操作，该阶段是在第二次热身赛直到最终比赛（6 月 4 日到 6 月 12 日）。该阶段主要改进了以下部分：

（1）增加了预测自己和对方回家路线（即根据最短回家路径进行预测），并在回家路线上击杀对方或者避免对方在回家路线上击杀我方的函数，从而在避免失误的同时增大了击杀可能

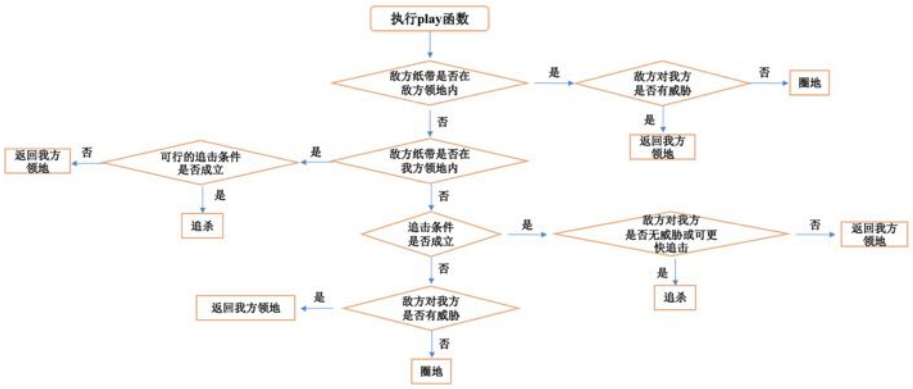
（2）针对在两方边界处和在对方领地内的条件做了改进，减少为了击杀而非常接近对方领地的行为，并避免深入敌方领地的行为

（3）调整圈地函数中所用到的各类关键参数，加大圈地面积

（4）提出了用图的 A* 算法来计算有障碍下最短距离的函数，即在确定最短回家路径或击杀路径时考虑敌方领地，选择绕过其领地的方式回家或击杀；但该方法最终由于耗时较大且在多个回合中随领地变化容易反复修改决策的问题而被放弃

1.2 算法流程图

下面是 play(stat, storage) 函数的流程图：



1.3 算法运行时间复杂度分析

本算法运行时间复杂度为 O(n2)。复杂度最高的地方出现在记录赛场领地与纸带归属时，在 x 方向和 y 方向进行二维遍历。为降低算法复杂度，本算法在更新领地与纸带归属前，进行了是否出现重大变化的判断，如若未出现明显改变领地面积与纸带长度的情况，则无需进行遍历，仅需根据操作进行补充即可。这种方式大大节约了时间。

2 程序代码说明

2.1 数据结构说明

本算法主要采取了列表、字典、栈和图四类数据结构。其主体采用的是列表和字典的数据结构进行信息保存、判断和处理。

在记录纸带存储信息时使用了栈的数据结构。这是因为纸带本身即是一个栈的结构，即在进行完每一步操作后要么会使纸带延伸一步，即让新的一格入栈而原先其他格子依然在栈中；要么发生纸带变为领地，则整个清空栈即可。这样不用遍历整个场地判断纸带归属，只用判断当前格是否的归属即可。相对遍历而言，使得算法复杂度从 O(n2) 缩小到了 O(1)，其中 n 为场地边长。当然需要指出，我们最终用 Python 的列表实现了栈，没有调用 Stack 类。这是因为执行角度讲两个耗时几乎没有差异，而列表实现栈（即只用 append 和 pop 功能）则相对简便。

在进行计算最短击杀和回家距离与路径时我们用到了图的数据结构。具体而言主要是

引入了 Node 类。通过该类，我们实现了 A* 算法来计算有障碍下最短距离的函数，即计算从 A 点到 B 点最短路径时从 A 点开始沿其周围非边界、非敌方领地的格子进行路径搜索，而搜索中使用启发式规则最小化 f(n) = g(n) + h(n)，其中 g(n) 为起点到当前点的最短已知路径距离，h(n) 为当前点到终点曼哈顿距离。相对一般的 A* 算法（或比较类似的 Dijkstra 算法），在具体实现中，由于场地本身即是一个完备的图，故我们没有再定义 Graph 类，而是直接在 Node 类上的基础上判断与之连接的、且不为对方领地的点，从而进行路径搜索。最后需要说明，由于其复杂度极高，且在场地局势发生变化时实际操作效果并不显著优于直接遍历求最短路径，故我们未在比赛中正式调用之，而还是选取了遍历的方案。

2.2 函数说明

1.play(stat, storage) 函数首先，从 stat 中提取场地高度、宽度的信息，及我方和敌方当前时刻的领地、纸带信息。

```
def play(stat, storage):
    height = stat['size'][1]  # 场地高
    width = stat['size'][0]  # 场地宽
    me, enemy, field, band = stat['now']['me'], stat['now']['enemy'], stat['now']['fields'], stat['now']['bands']
```

其次，记录全图的纸带归属信息。具体而言，我们用栈的数据结构记录纸带信息。通过调用上局存储记录，看当前位置是否是我方（敌方）纸带，如果是则将该位置入栈，否则意味着经过这步操作，已经完成一次圈地，原有的纸带已经属于领地，因此清空记录纸带信息的栈，只保留当前位置。

```
# 记录全图的纸带归属信息
try:
    mybands = storage['mybands']
except:
    mybands = [[me['x'], me['y']]]

try:
    enemybands = storage['enemybands']
except:
    enemybands = [[enemy['x'], enemy['y']]]

try:
    # 判定思路：现在的这步圈完一块地，之前的纸带部分现在已经是领地了
    # 判定方式：如果现在所处位置的纸带属性不属于me，则清空我的纸带，只保留现在的位置
    if band[me['x']][me['y']] != me['id']:
        mybands = [[me['x'], me['y']]]
    # 否则，在我的纸带中加入现在位置
    else:
        mybands.append([me['x'], me['y']])
except:
    mybands = [[me['x'], me['y']]]
```

```
# 以下是用来完整保证该方法和之前的遍历法的点完全相同
if band[mybands[0][0]][mybands[0][1]] != me['id']:
    mybands.pop(0)
mybands.append([me['x'], me['y']])

try:
    if band[enemy['x']][enemy['y']] != enemy['id']:
        enemybands = [[enemy['x'], enemy['y']]]
    else:
        enemybands.append([enemy['x'], enemy['y']])
except:
    enemybands = [[enemy['x'], enemy['y']]]

if band[enemybands[0][0]][enemybands[0][1]] != enemy['id']:
    enemybands.pop(0)
enemybands.append([enemy['x'], enemy['y']])

storage['mybands'] = mybands
storage['enemybands'] = enemybands
```

再次，记录全图的领地归属信息。为节省计算量，先判断领地是否发生变化。具体的方式是，如果领地发生变化，意味着这步操作实现了一次圈地，之前的纸带头回到了大本营，所以新状态下纸带长度不超过 1。因此，采用纸带长度是否超过 1 判断领地是否发生变化。若我方和敌方任意一方纸带长度不超过 1，则双方至少有一方领地发生变化，于是采用遍历的方式重新记录领地归属。值得注意的是，只要有一方纸带长度不超过 1，双方的领地归属都可能发生变化，而非仅有纸袋长度不超过 1 的一方领地发生变化。若双方纸带长度均大于 1，则双方的领地未发生变化，各自的纸带都在延伸而尚未回到大本营，因此直接调用上局存储的领地归属记录。

```
# 记录全图的领地归属信息
try:
    # 如果我方或敌方纸带不超过一格则要重新遍历
    if len(mybands)<=1 or len(enemybands)<=1:
        myhome = []
        enemyhome = []
        for i in range(0, width):
            for j in range(0, height):
                if field[i][j] == me['id']:
                    myhome.append([i, j])
                elif field[i][j] == enemy['id']:
                    enemyhome.append([i, j])

    # 当前所走的这步，没有对领地产生影响，则双方领地不变，故直接提取存储的双方领地信息
    else:
        myhome = storage['myhome']
        enemyhome = storage['enemyhome']

except:  # try不合理时，比如没有前一期时
    myhome = []
    enemyhome = []
    for i in range(0, width):
        for j in range(0, height):
            if field[i][j] == me['id']:
                myhome.append([i, j])
            elif field[i][j] == enemy['id']:
                enemyhome.append([i, j])

storage['myhome'] = myhome
storage['enemyhome'] = enemyhome
```

之后，在进入 play 函数核心操作部分之前，为方便调用，我们做了一些基础准备。首先，我们利用 storage 模块调用 load 函数里面的一些游戏具体行为的函数。例如，我们在 load 函数中定义了敌我双方距离的函数（dist）、最快回家函数（distToHome）、圈地函数（square）等。同时，我们还封装了各类计算距离的函数结果，形成列表，以供后续使用。例如，enemyToHome 用来记录敌方回到他们的领地的最快回家方式，meKillEnemy 记录我方攻击敌方的最快方式。

```
# play中可供调用的函数及其它基础设施准备
distToHome = storage['distToHome']  # 调用storage模块，方便下面使用最短回家距离函数
dist = storage['dist']  # 敌我双方距离函数
killDist = storage['killDist']  # 一方攻击另一方纸带的最短距离函数
reach = storage['reach']  # 一方到另一个指定点的移动函数
square = storage['square']  # 向领地外圈地的函数
directions = [[1, 0], [0, 1], [-1, 0], [0, -1]]
expectedband = storage['expectedband']  # 未来几步中可能出现的纸带
gohomebands = expectedband(distToHome(me, myhome), me)  # 定义我回家拉出来的纸带
killhimbands = expectedband(killDist(me, enemybands), me)  # 定义我干掉他时将要拉出来的纸带
killmebands = expectedband(killDist(enemy, mybands), enemy)  # 定义他杀我时可能拉出来的纸带
escapebands = expectedband(distToHome(enemy, enemyhome), enemy)  # 定义他跑回家时可能拉出来的纸带

# 封装各类计算距离的函数结果，形成列表，以供后续使用
enemyToHome = distToHome(enemy, enemyhome)
meToHome = distToHome(me, myhome)
meKillEnemy = killDist(me, enemybands)
enemyKillMe = killDist(enemy, mybands)
expectedEnemyKillMe = killDist(enemy, gohomebands)
expectedMeKillEnemy = killDist(me, escapebands)
```

下面进入 play 函数操作判断的核心部分。我们分别讨论了两种可能：（1）敌方纸带不在敌方的领地内；（2）敌方纸带在敌方的领地内。

（1）敌方纸带不在敌方的领地内

当敌方纸带不在敌方的领地内时，我们分别讨论敌方纸带不在我方领地和地方纸带在我方领地的两种情况。

i. 敌方纸带不在敌方领地且不在我方领地

当敌方纸带不在我方领地内时，我们利用敌人回家的距离是否大于我们去击杀敌人的距离来判断追击条件是否成立。若敌人回家距离大于我方去击杀敌人的距离，则追击条件成立，反之则不成立。然而，追击判断成立并不等于选择击杀，因为在去击杀的路上我方可能会被敌方击杀。因此，除了追击条件这个维度外，我们还考虑了敌方是否对我方有威胁这个维度。

下面是地方纸带在我方领地内时的具体情况：（a）当追击判断成立且敌方对我方回家没有威胁时，选择击杀操作。（b）当追击判断成立且虽然敌方对我方回家有威胁，但我方可以更快追杀到对方时，选择击杀操作。（c）当追击判断成立但是敌方对我方回家有威胁且我方不能更快追杀到对方时，选择快速爬回家。（d）当追击判断不成立时，如果敌方对我方造成威胁就快速爬回家。（e）当追击判断不成立时，如果敌方对我方不造成威胁，就继续圈地。

```
##################################
# 以下进入play函数操作判断的核心部分 #
##################################

# 敌人纸带不在敌人的领地内
if enemybands != [[enemy['x'], enemy['y']]]:
    i = 0
    while i <= len(enemybands) - 1:
        if field[enemybands[i][0]][enemybands[i][1]] == me['id']:  # 敌人的纸带在我方领地
            break
        i = i + 1
    # 敌人纸带不在敌人领地且不在我方领地
    if i == len(enemybands):
        # 追击判断成立并且对方对我方可以爬回家没有威胁，选择击杀操作
        if enemyToHome[0] > meKillEnemy[0] and \
                meToHome[0] <= enemyKillMe[0] - 1:
            # 返回快速击杀
            if band[me['x'] + directions[me['direction']][0]][me['y'] + directions[me['direction']][1]] == me['id']:
                # 防撞纸带
                return reach(meToHome, me, band, width, height, myhome)
            else:
                return reach(meKillEnemy, me, band, width, height, myhome)
        # 追击判断成立并且虽然对方可能对我方爬回家有威胁，但我方可更快追击对方，选择击杀操作
        elif enemyToHome[0] > meKillEnemy[0] and \
                    enemyKillMe[0] >= meKillEnemy[0] + 1:
            if band[me['x'] + directions[me['direction']][0]][me['y'] + directions[me['direction']][1]] == me['id']:
                # 防撞纸带
                return reach(meToHome, me, band, width, height, myhome)
            else:
                return reach(meKillEnemy, me, band, width, height, myhome)

        # 追击判断成立但是对方对我方有威胁且不可能更快追击到，此时爬回家
        elif enemyToHome[0] > meKillEnemy[0]:
            return reach(meToHome, me, band, width, height, myhome)
```

```
        # 追击判断不成立
        if enemyToHome[0] <= meKillEnemy[0]:
            # 圈地中如果对方造成威胁则回家
            if meToHome[0] >= enemyKillMe[0] - 5 or expectedEnemyKillMe[0] <= meToHome[0] + 5:
                return reach(meToHome, me, band, width, height, myhome)
            # 圈地中如果对方不造成威胁则继续圈地
            else:
                if me['x'] + directions[me['direction']][0] > width - 1 or me['x'] + directions[me['direction']][0] \
                    < 0 or me['y'] + directions[me['direction']][1] > height - 1 or me['y'] + \
                        directions[me['direction']][1] < 0:
                    return square(field, me, storage, band)  # 使用square的防撞墙
                else:
                    if band[me['x'] + directions[me['direction']][0]][me['y'] + directions[me['direction']][1]] == \
                            me['id']:  # 防止撞上自己的纸带
                        return reach(meToHome, me, band, width, height, myhome)
                    else:
                        return square(field, me, storage, band)
```

（a）当追击判断成立且敌方对我方回家没有威胁时，选择击杀操作

上文已经描述了判断追击条件是否成立的方式。我们通过我方纸带头回我方领地的距离是否小于敌方击杀我们的距离来判断敌方对我方是否有威胁。击杀操作我们使用了在 load 函数中会介绍的 reach 函数，这是一方到另一个指定点的移动函数。同时，我们增加了一个防撞纸带的判断条件。

（b）当追击判断成立且虽然敌方对我方回家有威胁，但我方可以更快追杀到对方时，选择击杀操作

此时，对方击杀我们的距离大于我们击杀对方的距离，所以我们可以选择击杀对方。具体击杀操作同（a）。

（c）当追击判断成立但是敌方对我方回家有威胁且我方不能更快追杀到对方时，选择快速爬回家

此时的判断条件是敌方回到敌方领地的距离大于我方击杀对方的距离，此时选择快速跑回家。回家的操作也是使用 reach 函数，当然此时的参数与击杀操作时 reach 函数的参数是不同的。

（d）当追击判断不成立时，如果敌方对我方造成威胁就快速爬回家

需要注意的是，这里我们判断敌方是否对我方造成威胁的方式相对简易。具体而言，如果我方回家所需距离大于敌方击杀我方所需距离减 5，或者如果预期敌方击杀我方距离小于我方回家距离加 5，则判断为敌方对我方造成威胁。此处的参数 5 是我们经过一些试验与比对选择的。从这个方法可以看出，这里我们的判断不是很完全，是基于一定的分析加上经验得到的判断方式，这也是我们的程序限制所在，和潜在的导致比赛中失败的原因。

（e）当追击判断不成立时，如果敌方对我方不造成威胁，就继续圈地

圈地操作我们使用了 load 函数中会介绍的 square 函数。同时，我们增加了防止撞墙与撞自身纸带的判断条件。

ii. 敌人纸带在我方领地

当敌方纸带在我方领地内时，如果没有威胁（即我们在回家的过程中不会被撞），且击杀时机成立，便选择击杀。如果击杀时机不成立但没有威胁，便选择继续圈地。如果受到威胁，便选择快速跑回家。

首先我们判断了是否会撞墙，并在可能会撞墙的情况下，利用 square 函数中的防撞墙机制改变路线。

```
156     # 敌人在我方阵营，如果没有威胁（gohomebands也不会被撞）而且击杀时机成立，再击杀；如果击杀时机不成立但没有威胁就接着圈地，如果有威胁，就回家
157     else:
158         # 防撞墙
159         if me['x'] + directions[me['direction']][0] > width - 1 or me['x'] + directions[me['direction']][0] \
160                 < 0 or me['y'] + directions[me['direction']][1] > height - 1 or me['y'] + \
161                 directions[me['direction']][1] < 0:
162             return square(field, me, storage, band)  # 使用square的防撞墙
```

在不会撞墙的情况下，我们讨论了下面几种情形：（a）敌人只差一步就回到敌人的领地，我方追杀不及即将进入敌方领地。（b）我方完全不受到威胁且满足追击条件，选择追杀。（c）我方受到威胁但可以更快地进行追杀，选择追杀。（d）我方完全不受威胁，虽然不满足追击条件，但预测未来纸带行为满足追击条件，则选择追杀。（e）其余情况，为防止敌方圈走我方地后，影响我方顺利回家，选择先回家。

```
# 特殊情况的判定，敌人只差一步回到自己领地，我方追杀不及将要进入对方领地。
elif field[me['x'] + directions[me['direction']][0]][me['y'] + directions[me['direction']][1]] == enemy['id'] \
        and field[me['x']][me['y']] == me['id'] and enemyToHome[0] <= 1:
    # 判断往哪个方向躲避以避免在对方领地和对方碰撞
    # 当我方向东边行时
    if me['direction'] == 0 and field[me['x']][me['y'] + 1] == me['id']:
        return 'r'
    elif me['direction'] == 0 and field[me['x']][me['y'] - 1] == me['id']:
        return 'l'
    elif me['direction'] == 0:  # 左右转都会出界时
        if enemy['y'] < me['y']:
            return 'r'
        elif enemy['y'] >= me['y']:
            return 'l'
    # 当我方向南边行时
    elif me['direction'] == 1 and field[me['x'] - 1][me['y']] == me['id']:
        return 'r'
    elif me['direction'] == 1 and field[me['x'] + 1][me['y']] == me['id']:
        return 'l'
    elif me['direction'] == 1:  # 左右转都会出界时
        if enemy['x'] < me['x']:
            return 'l'
        elif enemy['x'] >= me['x']:
            return 'r'
    # 当我方向西边行时
    elif me['direction'] == 2 and field[me['x']][me['y'] - 1] == me['id']:
        return 'r'
    elif me['direction'] == 2 and field[me['x']][me['y'] + 1] == me['id']:
        return 'l'
    elif me['direction'] == 2:  # 左右转都会出界时
        if enemy['y'] < me['y']:
            return 'l'
        elif enemy['y'] >= me['y']:
            return 'r'
    # 当我方向北边行时
    elif me['direction'] == 3 and field[me['x'] + 1][me['y']] == me['id']:
        return 'r'
    elif me['direction'] == 3 and field[me['x'] - 1][me['y']] == me['id']:
        return 'l'
    elif me['direction'] == 3:  # 左右转都会出界时
        if enemy['x'] < me['x']:
            return 'r'
        elif enemy['x'] >= me['x']:
            return 'l'
```

（a）敌人只差一步就回到敌人的领地，我方追杀不及即将进入敌方领地

这种情况下，我们根据自己当时前行方向，判断下一步方向，以避免在对方领地和对方碰撞。具体而言，我们分别对当前方向为东、西、南、北进行讨论。

（b）我方完全不受到威胁且满足追击条件，选择追杀

判断我方完全不受威胁的方式是：我方回家的距离，既小于敌方现在击杀我方的距离，也小于敌方在预期路线下选择击杀我方的距离。追击条件的判断方式是：我方击杀敌方的距离小于敌方回到敌方家里的距离。同时，我们增加了一个防撞纸带的判断条件（事实上，在之后的每种情况，我们都增加了防撞纸带的判断条件，后文不再赘述）。

（c）我方受到威胁但可以更快地进行追杀，选择追杀

我方收到威胁但可以更快地进行追杀的判断条件是：我方追杀敌方的距离，既小于敌方追杀我方的距离，也小于敌方回到敌方家里的距离。

（d）我方完全不受威胁，虽然不满足追击条件，但预测未来纸带行为满足追击条件，则选择追杀

我方完全不受威胁的判断条件与追击判断条件同（b）中所述。此时不满足追击条件，但是我们预测双方未来纸带的行为可能满足我方追杀对方的条件，因此依旧选择追杀。

（e）其余情况，为防止敌方圈走我方地后，影响我方顺利回家，选择先回家

只要我方回家的距离小于敌方击杀我方的距离，也小于敌方预期纸带击杀我方的距离，同时我回家的距离小于敌方回家的距离，为防止敌方圈走我方地后，影响我方顺利回家，我们选择先回家。

```
        # 完全没有威胁且满足追击，选择击杀
        elif meToHome[0] < enemyKillMe[0] and expectedEnemyKillMe[0] \
            > meToHome[0] and meKillEnemy[0] < enemyToHome[0]:
            if band[me['x'] + directions[me['direction']][0]][me['y'] + directions[me['direction']][1]] == \
                    me['id']:  # 防止撞上自己的纸带
                return reach(meToHome, me, band, width, height, myhome)
            else:
                return reach(meKillEnemy, me, band, width, height, myhome)
        # 有威胁但满足更快追击，选择击杀
        elif meKillEnemy[0] < enemyKillMe[0] and meKillEnemy[0] < enemyToHome[0]:
            if band[me['x'] + directions[me['direction']][0]][me['y'] + directions[me['direction']][1]] == \
                    me['id']:  # 防止撞上自己的纸带
                return reach(meToHome, me, band, width, height, myhome)
            else:
                return reach(meKillEnemy, me, band, width, height, myhome)
        # 完全没有威胁但是不满足追击，但是预测纸带可能满足追击，选择击杀
        elif meToHome[0] < enemyKillMe[0] and expectedEnemyKillMe[0] \
            > meToHome[0] and expectedMeKillEnemy[0] < enemyToHome[0]:
            if band[me['x'] + directions[me['direction']][0]][me['y'] + directions[me['direction']][1]] == \
                    me['id']:  # 防止撞上自己的纸带
                return reach(meToHome, me, band, width, height, myhome)
            else:
                return reach(expectedMeKillEnemy, me, band, width, height, myhome)
        # 如果对方踏入了我方阵营但是不满足追击条件，为了防止它圈走后回不了家的情况，还是先回家
        elif meToHome[0] < enemyKillMe[0] and expectedEnemyKillMe[0] \
            > meToHome[0] and enemyToHome[0] > meToHome[0]:
            if band[me['x'] + directions[me['direction']][0]][me['y'] + directions[me['direction']][1]] == \
                    me['id']:  # 防止撞上自己的纸带
                return reach(meToHome, me, band, width, height, myhome)
            else:
                return reach(expectedMeKillEnemy, me, band, width, height, myhome)
        else:
            return reach(meToHome, me, band, width, height, myhome)
```

（2）敌方纸带在敌方的领地内

当敌方纸带在敌方的领地内时，一定有追击条件判断不成立。此时，我们的思路是先判断敌方与我方的距离。如果距离较远，则敌方来不及过来追杀我们，于是我们先圈走敌方的一部分地。而当敌方距离我们较近时，我们要选择跑掉。

与敌人纸带在我方领地时类似，这里我们首先判断了是否会撞墙，并在可能会撞墙的情况下，利用 square 函数中的防撞墙机制改变路线。

```
    # 敌人纸带在敌人的领地时
    else:
        # 敌人在敌人的领地时，一定有追击判断不成立
        if enemyToHome[0] <= meKillEnemy[0]:
            # 圈地中如果下一步是出去到了对方的领地，需要判断对方离我的距离，小于多少时再出去
            # 圈地中如果对方造成威胁则回家
            # 如果可能撞墙，则利用square函数的防撞墙机制
            if me['x'] + directions[me['direction']][0] > width - 1 or me['x'] + directions[me['direction']][0] < 0 or \
                    me['y'] + directions[me['direction']][1] > height - 1 or me['y'] + directions[me['direction']][1] < 0:
                return square(field, me, storage, band)
```

不会撞墙时，首先，当双方距离小于 10 时，我们开始考虑判断往哪个方向跑以避免在对方领地与对方碰撞。事实上，这里的 10 也是根据经验得到的，不是最优的方式，也是我们的程序限制所在。具体躲避的方式，与（1） 敌方纸带在我方领地时（a）情况类似，在此不再赘述。

```
        # 不会撞墙时
        elif field[me['x'] + directions[me['direction']][0]][me['y'] + directions[me['direction']][1]] \
                == enemy['id'] and dist(me, enemy) < 10 and meToHome[0] == 0:
            # 判断往哪个方向躲避以避免在对方领地和对方碰撞
            # 当我方向东边行时
            if me['direction'] == 0 and field[me['x']][me['y'] + 1] == me['id']:
                return 'r'
            elif me['direction'] == 0 and field[me['x']][me['y'] - 1] == me['id']:
                return 'l'
            elif me['direction'] == 0:  # 左右转都会出界时
                if enemy['y'] < me['y']:
                    return 'r'
                elif enemy['y'] >= me['y']:
                    return 'l'
            # 当我方向南边行时
            elif me['direction'] == 1 and field[me['x'] - 1][me['y']] == me['id']:
                return 'r'
            elif me['direction'] == 1 and field[me['x'] + 1][me['y']] == me['id']:
                return 'l'
            elif me['direction'] == 1:  # 左右转都会出界时
                if enemy['x'] < me['x']:
                    return 'l'
                elif enemy['x'] >= me['x']:
                    return 'r'
```

```
            # 当我方向西边行时
            elif me['direction'] == 2 and field[me['x']][me['y'] - 1] == me['id']:
                return 'r'
            elif me['direction'] == 2 and field[me['x']][me['y'] + 1] == me['id']:
                return 'l'
            elif me['direction'] == 2:  # 左右转都会出界时
                if enemy['y'] < me['y']:
                    return 'l'
                elif enemy['y'] >= me['y']:
                    return 'r'
            # 当我方向北边行时
            elif me['direction'] == 3 and field[me['x'] + 1][me['y']] == me['id']:
                return 'r'
            elif me['direction'] == 3 and field[me['x'] - 1][me['y']] == me['id']:
                return 'l'
            elif me['direction'] == 3:  # 左右转都会出界时
                if enemy['x'] < me['x']:
                    return 'r'
                elif enemy['x'] >= me['x']:
                    return 'l'
```

当敌方击杀我方的距离小于我方回家距离加 5 时，我们选择快速回家。当预期双方未来的纸带行为可能会出现敌方追杀我方的情况时，如果这时他们追杀我们的距离小于我们回家的距离加 5，则我们也选择快速回家。这里，5 依旧是根据尝试与经验得到的，是我们的程序限制所在。

```
        elif expectedEnemyKillMe[0] <= meToHome[0] + 5:
            return reach(meToHome, me, band, width, height, myhome)

        elif meToHome[0] >= enemyKillMe[0] - 5:
            return reach(meToHome, me, band, width, height, myhome)
```

其余情况，我们认为对方不存在威胁，于是选择圈地。同时，我们考虑了防止撞墙和撞纸带的情况。

```
        elif expectedEnemyKillMe[0] <= meToHome[0] + 5:
            return reach(meToHome, me, band, width, height, myhome)

        elif meToHome[0] >= enemyKillMe[0] - 5:
            return reach(meToHome, me, band, width, height, myhome)
```

2.load(stat, storage) 函数

（1）dist(me, enemy) 函数

dist(me, enemy) 用来计算我方当前位置和敌方当前位置的距离。因为游戏只能水平方向或者竖直方向行走，因此对距离最好的刻画就是横纵坐标差的绝对值之和。

（2）distToHome(player0, home) 函数

distToHome(player0, home) 用来计算 player0 躲回领地 home 的最短距离，返回最短距离、x 和 y 的方向和最近到家的位置。

具体的实现方式是遍历 home 里的每一个点，计算与 player0 当前位置的距离，选取距离最小的 home 里的点，作为最近到家位置，同时记录最短距离、x 和 y 的方向。

（3）killDist(play1, bands2) 函数

killDist(play1, bands2) 用来计算一方到另一方的最短击杀距离，即一方到对方纸带的最短距离，返回最短距离、x 和 y 的方向和击杀位置。

采用的根本方式是遍历 bands2 里的所有点，计算 play1 到 bands2 各个点的距离，选择最短距离。

（4）reach(alist, play0, band, width, height, myhome) 函数

reach(alist, play0, band, width, height, myhome) 是定义最快跑到一个指定地方的函数，多用于最快跑回家或者最快去击杀的情形。其中，alist 是用来传入最短距离列表的参数。

首先，判断目标方向。然后根据目标方向和现在方向，判断转向哪个方向。具体而言，我们分了不用转向、只在 y 方向转、只在 x 方向转、x 和 y 两个方向都要转的四种情况。

i. 不用转向的情况

当目标方向的 x 和 y 维度都是 None 时，不用转向，继续直行。

ii. 只需在 y 方向转或只需在 x 方向转

当 xDirection 为 None 时，只需要在 y 方向转（当 yDirection 为 None，只需要在 x 方向转时与这种情况类似，为避免重复，以只在 y 方向转为例）。首先，我们进行了防止撞击自己的纸带的设置。根据当前方向和目标 y 方向，分 4 种情况。每种情况判断若下一步采用目标方向，是否会撞击自己的纸带，如果会撞击则保持直行不变方向，如果不会撞击则按照目标方向改变方向。

当现在的前进方向正是目标 y 方向时，分向南和向北两种情况讨论。每种情况具体内容类似，为避免重复，以向南为例。当沿着向南的方向走一步后，play0 纸带头的位置是在 play0 的领地时，判断现在的纸带是否已经包围住领地。如果纸带没有包裹住领地，遍历可

能的 x 方向的点，找到击杀位置。通过判断击杀位置 x 坐标和现在位置 x 坐标的大小，判断向左还是向右转向。如果纸带包裹住领地，用类似的方式判断转动方向，唯一的区别是向左向右的判断条件和纸带没有包裹住领地的情形相反。其余情况，选择直行。

当目标 y 方向与现在前行的方向不同时，分从向北转到向南和从向南转到向北两种情况。在掉头的过程中，如果已经距离一边的墙太近，就从另一边掉头，如果不存在此问题，则一律向右转来实现掉头。

只需在 x 方向转与只需在 y 方向转类似，在此省略。

iii. 两个方向均需转向

我们根据当前前进方向分 4 种情况讨论。为避免重复，在此以当前前进方向向东，即 play0[‘direction’] == 0 为例。与只需在 y 方向转向类似，我们判断现在的纸带是否已经包裹住领地，然后据此确定向左还是向右。

之后，与只需在 y 方向转向类似，需要添加机制防止撞击到自己的纸带。定义数组 directionChangeY 和 directionChangeX，记录当前方向并分别记录目标 y 方向和目标 x 方向。然后分当前方向是横向（x 向）还是纵向（y 向）进行讨论。如果改变方向会导致碰到自己的纸带，则保持直行。为避免重复，此处截图以当前方向时 x 向为例。

以上是 reach(alist, play0, band, width, height, myhome) 函数的全部内容。

（5）square(field, me, storage, band) 函数

square(field, me, storage, band) 函数用来向外圈地。

首先，添设防止出界机制。根据当前行进方向，分别讨论保证 y 轴不出界和保证 x 轴不出界两种情况，并且区分多种靠角落的情形。

之后，进入圈地函数的核心操作，即圈地操作。我们采取正方形圈地方式，每回到领地一次就扩大一次圈地范围。具体而言，每次我们会扩大 10 格，这个参数具体数值也是根据尝试与经验得到的。因此这也是我们的程序所限之处。

（6）expectedband(distlist, play0) 函数

expectedband(distlist, play0) 用来预测未来几步可能出现的纸带。设计这个函数的主要目的是预测我方和敌方回家的路线，从而有助于更好地判断进行追杀、回家、圈地中的哪项操作。我们分为三种情况进行讨论：预计到达的点和现在选手（play0）的横坐标相

同、预计到达的点和现在选手的纵坐标相同、预计到达的点和现在选手的横纵坐标均不同。由于前两种逻辑类似，为避免重复，仅以第一种为例。

当预计到达的点和现在选手（play0）的横坐标相同时，分别讨论现在选手的纵坐标小于、大于、等于预计到达的点的纵坐标三种情况。若等于，则直接在 eband 中添加当前的点。若现在选手的纵坐标小于预计到达的点的纵坐标，则根据当前方向是否向北，决定是否提前右转。若向北则提前右转，反之直接在 eband 中添加相应的点。大于则将向北变为向南即可。

当预计到达的点和现在选手的横纵坐标均不同时，分 4 种情况讨论。第一个维度是 x 方向和 y 方向，第二个维度是预计到达的点的坐标大于还是小于现在选手的坐标。不妨以当前选手 x 方向坐标大于预计到达的点的横坐标、当前选手 y 方向坐标小于预计到达的点的纵坐标为例。将当前方向向东或向南分为一种情况，向西或者向北分为另一种情况，将会经过的点的位置加入到 eband 中。

（7）存储至 storage 模块

最后，我们将需要存储的内容存在 storage 中。

（8）A* 算法

虽然我们在最终比赛时未使用 A* 算法，但 A* 算法的思路对于纸带圈地问题的探索是有启发意义的。但因为这不是我们采用的主要方法，在此仅做简单介绍。更多关于算法思想的阐述请见 2.1 节数据结果说明部分。

首先，我们定义节点类。定义的节点参数包括父节点和自身位置。如果添加了会出界或到达敌方边界的节点，则触发异常，用于之后在 addAdjacentIntoOpen(node) 中捕捉。

其次，我们定义了计算曼哈顿距离的函数和求 open_list 中 F 值最小的节点的函数。

之后，定义了对 open_list 中节点的操作函数。

最后，定义了寻找最终路径的函数。

2.3 程序限制

从第一次热身赛开始，我们的程序便未在热身赛和正式比赛中报错，说明我们的程序在与比赛环境接口的调用上和基本的算法完整性上是表现得尽如人意的。但是，我们的算

法在竞技中还是暴露了问题。

我们的部分进行操作选择前的判断条件是基于尝试和经验的，而非经过计算得到确切的判断条件，这是相对不严谨的地方。例如，在 play 函数中，敌方纸带不在敌方领地且不在我方领地，当追击判断不成立时，如果敌方对我方造成威胁就快速爬回家的这一情形下，是否造成威胁的判断就是基于尝试与经验的。其余存在限制的判断在函数说明部分有作说明。

此外，我们的圈地函数相对简单。我们采用的是简单的画正方形圈地。事实上，为了在经济中表现更出色，可以考虑走最贴近四周的路线等方式，更好地实现圈地面积最大化。

3 实验结果

3.1 测试数据

实验环境说明：

硬件配置：CPU 2.2 GHz Intel Core i7, 内存 16 GB 1600 MHz DDR3

操作系统：macOS Sierra 10.12

Python 版本：3.6.5

在第一次热身赛中，我们上传了名为 kilo-1 的纸带 AI，它在第一次热身赛中表现的非常优秀，所有对战都获得了胜利。第一次热身赛结果如下：

	quebec_2	november_1	india_1	Whiskey_1	kilo_1	quebec_1	delta_2	delta_1	hotel_1	xray	uniform_1	hotel_2	Score
quebec_2		-	+	+	-	+	+	+	-	+	+	-	21
november_1	+		-	+	-	+	+	+	+	+	-	+	24
india_1	-	+		+	-	+	+	+	-	+	-	+	21
Whiskey_1	-	-	-		-	-	-	-	-	-	-	-	0
kilo_1	+	+	+	+		+	+	+	+	+	+	+	33
quebec_1	-	-	-	+	-		+	-	-	-	-	-	6
delta_2	-	-	-	+	-	-		-	-	-	-	-	3
delta_1	-	-	-	+	-	+	+		-	-	-	-	9
hotel_1	+	-	+	+	-	+	+	+		+	+	+	27
xray	-	-	-	+	-	+	+	+	-		-	-	12
uniform_1	-	+	+	+	-	+	+	+	-	+		0	22
hotel_2	+	-	-	+	-	+	+	+	-	+	0		19

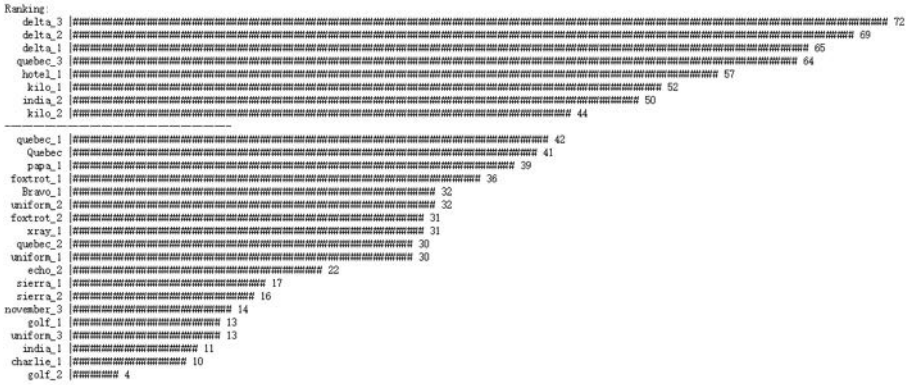
```
   kilo_1 33 ------------------------------------------------------------------
  hotel_1 27 ------------------------------------------------------
november_1 24 ------------------------------------------------
uniform_1 22 --------------------------------------------
 quebec_2 21 ------------------------------------------
  india_1 21 ------------------------------------------
  hotel_2 19 --------------------------------------
     xray 12 ------------------------
  delta_1  9 ------------------
 quebec_1  6 ------------
  delta_2  3 ------
Whiskey_1  0
```

第二次热身赛中，我们上传了两个不同版本的纸带 AI，分别是 kilo-1 和 kilo-2。这一次我们的对手实力都有所增强，特别是 hotel 和经过深度学习训练的 delta 实力大涨，我们的 kilo-1、kilo-2 被 delta 多次击败。位于小组第三、第四名。第二次热身赛结果如下：

```
Status:
      t_n17_GOLF_1: |*****|  +  |  -  |  +  |  -  |  -  |  +  |  -  |  -  |  -  |  -  |  -  |  -  |  9
      t_n17_GOLF_2: |  -  |*****|  -  |  -  |  -  |  -  |  +  |  -  |  -  |  -  |  -  |  -  |  -  |  3
     t_n17_delta_1: |  +  |  +  |*****|  +  |  +  |  +  |  +  |  +  |  +  |  +  |  +  |  +  |  +  |  36
      t_n17_echo_2: |  -  |  +  |  -  |*****|  +  |  -  |  +  |  -  |  -  |  +  |  -  |  -  |  +  |  15
   t_n17_foxtrot_1: |  +  |  +  |  -  |  -  |*****|  -  |  +  |  +  |  -  |  0  |  -  |  -  |  +  |  16
   t_n17_foxtrot_2: |  +  |  +  |  -  |  +  |  +  |*****|  +  |  +  |  -  |  0  |  -  |  -  |  0  |  20
        t_n17_golf: |  -  |  -  |  -  |  -  |  -  |  -  |*****|  -  |  -  |  -  |  -  |  -  |  -  |  0
     t_n17_hotel_1: |  +  |  +  |  -  |  +  |  -  |  -  |  +  |*****|  -  |  +  |  +  |  +  |  0  |  22
     t_n17_hotel_2: |  +  |  +  |  -  |  +  |  +  |  +  |  +  |  +  |*****|  -  |  +  |  -  |  +  |  27
     t_n17_india_1: |  +  |  +  |  -  |  -  |  0  |  0  |  +  |  -  |  +  |*****|  -  |  +  |  +  |  20
      t_n17_kilo_1: |  +  |  +  |  -  |  +  |  +  |  +  |  +  |  -  |  -  |  +  |*****|  0  |  +  |  25
      t_n17_kilo_2: |  +  |  +  |  -  |  +  |  +  |  +  |  +  |  -  |  +  |  -  |  0  |*****|  +  |  25
t_n17_november_2: |  +  |  +  |  -  |  -  |  -  |  0  |  +  |  0  |  -  |  -  |  -  |  -  |*****|  11


Ranking:
     t_n17_delta_1 |######################################################################## 36
     t_n17_hotel_2 |###################################################### 27
      t_n17_kilo_1 |################################################## 25
      t_n17_kilo_2 |################################################## 25
     t_n17_hotel_1 |############################################ 22
   t_n17_foxtrot_2 |######################################## 20
     t_n17_india_1 |######################################## 20
   t_n17_foxtrot_1 |################################ 16
---------------------------------------
      t_n17_echo_2 |############################## 15
  t_n17_november_2 |###################### 11
      t_n17_GOLF_1 |################## 9
      t_n17_GOLF_2 |###### 3
        t_n17_golf | 0
```

第三次热身赛中，又出现了另一名强劲对手 quebec-3，我们的 kilo-1 只位列第六。结果如下。



3.2 结果分析

针对第一次的测试结果和我们对代码的分析，我们针对下列内容进行了改进。 由于自己对打的死因经常是超时，我们尝试了优化存储信息的数据结构，看看能否在最短回家/击杀函数中不每次都遍历一遍。 在不影响性能的情况下简化转向代码。 解决 play 里面的个别判断语句未穷尽的情况。 对方在我方领地时，需要判断是否对我方纸带有威胁，如果有威胁，且威胁比击杀要快，则首先返回家里，如果没有威胁或者我方会击杀，则选择击杀模式。 在击杀判断部分，除了对手在我方领地的情况外，增加对手的纸带在我方领地的情况判断。 增加考虑我方到对方的纸带的时间大于对方回家让纸带小时的时间的情况，此时不选择击杀。 增加函数判断最短回家路径是否会被对手攻击。 增加函数判断在我方领地、对方领地内是否会被撞击。

针对第二次的测试结果，我们进行了如下分析。 对方在我领地内时我方追击时机不妥，总是直接冲出去，而对方已经跑回领地。 对方在领地接壤处时，我方会自己走出来撞上去，这个主要是由于 square 圈地函数设计有误。 追击模式和回家模式切换时易撞纸带自杀，需要微调这两个函数判断思路。 有时对方圈走我方一块地之后会导致我方来不及回到领地，需要对这种情况加以判断。

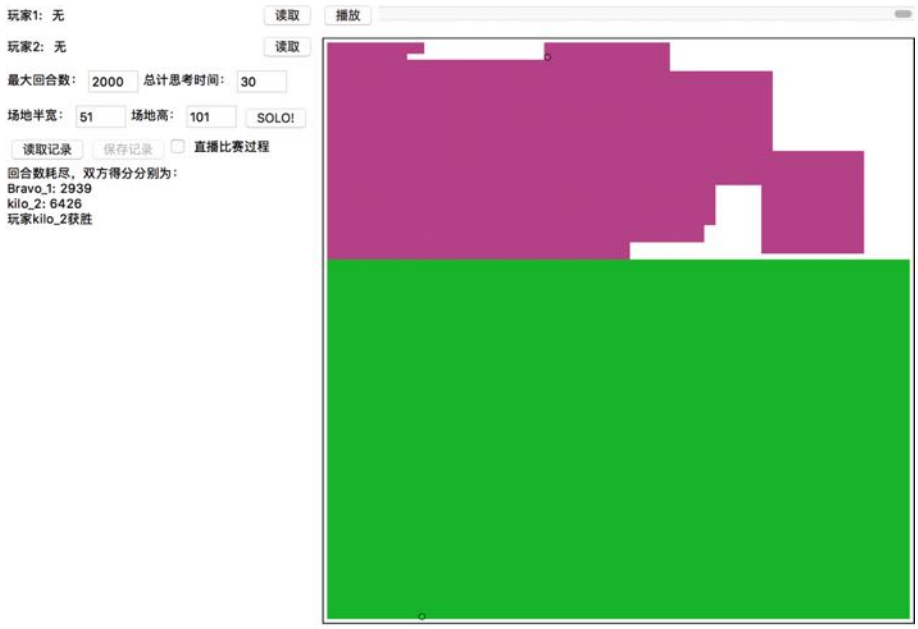
针对第三次测试结果，我们进行了如下分析。 在追击过程中对方反追击就会很容易自杀，需要继续改进纸带防撞部分。 依靠躲避和圈地的效果不错，反而追击有时会容易把自己搞乱，积极躲避后多圈地可能会有不错的效果。 会出现时间耗尽的情况，需要优化数据结构和算法。 在追击对方的途中需要注意安全距离，有时对方会由逃跑变为反追击，我方来不及回家纸带就被撞断。

除了参与热身赛测试外，我们还私下里与其他组的 AI 进行了单独对战测试。我们曾与某 AI 对战 20 场，先手我方胜 7 场，后手我方胜 6 场，纸带被自己撞断三次，失去领地后纸带被对手撞断三次，有一次压迫情况下撞墙自杀。由于对方 AI 比较激进，我们圈地能力相比之下明显要差一些。针对这些对战结果，我们首先主要改进的就是撞纸带问题。

3.3 经典战局

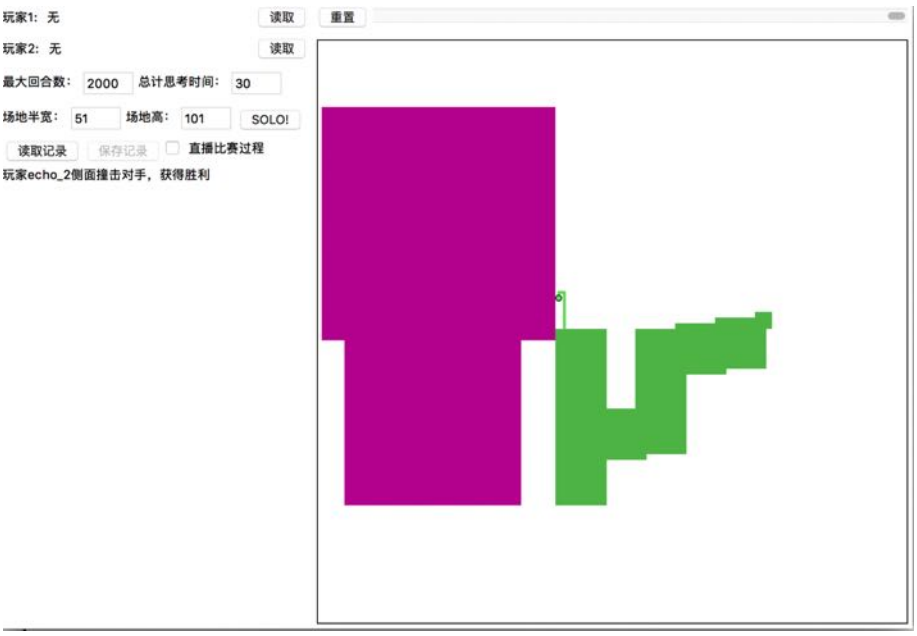
1. kilo（我方）VS Bravo

在这一局中，我方主要采用了圈地和躲避技术，最后超过了回合数，双方比较圈地面积，我方获胜。在其他大部分战局中我方都以进攻为主，这局是为数不多的表现为以躲避圈地为主，看起来也取到了不错的效果。



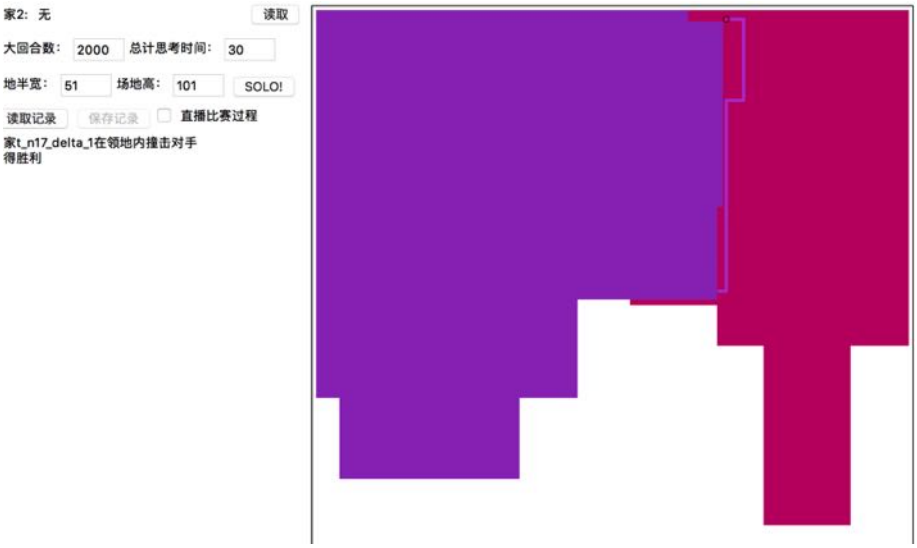
2. kilo（我方）VS echo

在大部分对战中，我方都优于 echo，唯独在这一战局中，echo 向北运动，我们想中途去撞断 E 的纸带，进攻途中没有注意保持安全距离，被 E 反过来从侧面撞断纸带。为此我们开始在进攻中注意保护自己的安全，注意我方和对手纸带最前端的安全距离。



3. kilo VS delta

在这次对战中，原本我方在进攻状态，正在追击对方，中途对手突然圈地成功，我方领地突然变成对方领地，导致我方 AI 反应不过来，在对方刚圈出的领地内被反击撞死。这一战局使我们开始考虑为 AI 增加一些对手行为预测和风险预判功能。



4 实习过程总结

4.1 分工与合作

合作交流方式：

1. 在数算课后组织讨论

2. 在教学楼讨论区开小组会议集中讨论分析

3. 在微信群中随时沟通讨论，上传程序

分工：

1. 开发算法和编程任务：张欣然同学主要负责代码的编写和修改，耿宗泽同学负责优化算法和数据结构，欧阳萌淞和刘书颖同学主要负责写报告并对每次热身赛结果进行复盘总结。

2. 小组报告：耿宗泽同学负责算法思想阐述，欧阳萌淞同学负责程序代码说明部分，刘书颖同学负责实验结果分析和实习过程总结部分。

小组初建

选课之初，为了大作业便早早联系组队，形成数算课程小群。5 月布置任务，一片和谐之声中选出了可爱的组长，迅速展开讨论。同时依靠本组可爱的特性，选出来小组名称 kilo 和头像。

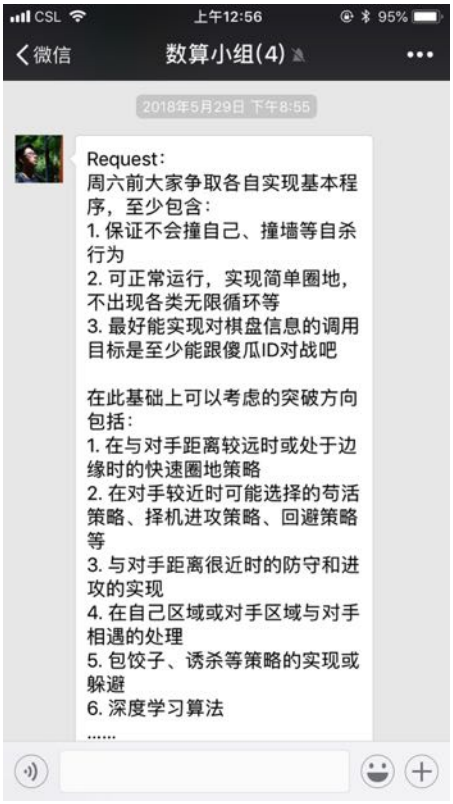
CSL
上午1:00
94%
微信
数算小组(4)
吼滴!
PAPER·IO
SHARE
PLAY AGAIN
不好好学习的我，又去玩游戏了
我们要选一个，大家想选啥吗?
2018年6月1日 下午2:07
要不来个K开头的?(随便选的)
可以啊
那就Kilo好了

CSL
上午12:58
94%
微信
数算小组(4)
2018年6月1日 上午8:17
大家吼！这个作为小组头像可以吗?
哈哈哈哈哈哈哈可以
2018年6月1日 下午1:36
大家觉得如果ok的话，就用这个啦~
我觉得没问题!
2018年6月1日 下午1:47
可以的

第一次组会

时间：5 月 29 日

初步确立对战想法，讨论了基本功能，分配任务，采用各自编写然后相互对战的方式选出组内最优作为基础，最终学妹的代码遥遥领先。



初步任务分配

第二次组会

时间：6 月 5 日

经历了第一次热身赛，几乎保持了全胜的记录（当然主要可能是大家都还没准备好），针对 uniform、hotel 等在热身赛中和我么异常焦灼的队伍，再次进行会议。主要提出修改圈地函数和击杀函数两部分，同时也开始准备做一下数据结构的优化。

1. 对手在我方领地时，需要判断是否对我方纸带有威胁，如果有威胁，且威胁比击杀要快，首先返回家里；如果没有威胁或者我方会先击杀，就选择击杀模式。
2. 优化圈地函数，可以让一开始圈的地无限直到要到边界或者有被击杀的威胁再跑回来？或者先圈一圈，然后反过来再圈一次（uniform组就是），在离对方更远的方向走长边。（往后放）
3. 有可能，回家的地盘被覆上了对手的纸带，还没跑回去就已经被圈走了
4. 击杀方面，不只是对手在我们领地的时候这样判断，对手的纸带在我们领地的时候也是要击杀的。
5. 追击函数有bug，ydirection和我方direction相同都为北时，不应该转向，已修改。
6. 加一个如果我方到它的纸带的时间大于它回家让纸带消失的时间，就不选择追杀，而是使用gohome判断。
7. 加一个判断回家拉出来的纸带会不会被撞断的函数，就是实时更新最短回家路径会不会被它攻击
8. 加一个防领地内被撞的函数
9. 纸带和领地的遍历改成一次
10. 判断双方走完之后，是否在对方领地，从而可以判断是否需要遍历。
11. 数据结构和算法优化。

针对第一次热身赛的修改任务

讨论区自拍

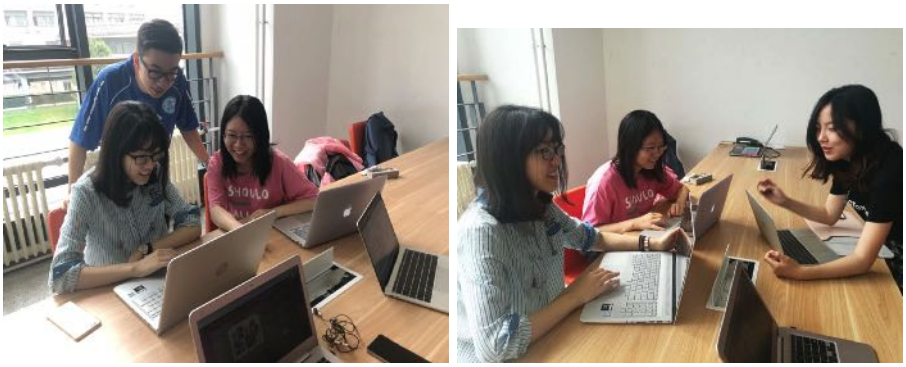
4.1.3 第三次组会

时间：6 月 11 日

决赛在即，异常紧张。和其他组小伙伴们进行了一次友谊赛，发现了圈地能力上还是

有一些缺憾，但在整体上撞纸带自杀等问题都基本解决。于是召开了这次组会，希望在赛前能够不断优化功能。

友好的组间比赛



老地方继续讨论

4.1.4 随时随地的微信讨论

时间：清晨、中午、午夜 anytime

充分利用微信工具，有新的想法或者新的修改都会立马在微信中联系组员，大家一起开脑洞，一起复盘，一起总结，更新了 10 多版代码，不断优化不断成长。

老地方继续讨论

4.2 经验与教训

1. 算法的选择

我们本次实习作业的算法部分完全由人工设计，非常依靠人类的先验知识，算法中包含了大量程序设计者任意或者依据经验指定的阈值和条件判断。这样设计的算法显然无法考虑到所有可能的情况并为其设计出最优解决方案，相比之下采用深度学习方法则一定程度上避免了这些人为因素，有可能学习到更优的策略，鲁棒性也更强，而且从对战结果来看基于深度学习的 AI 性能明显更为优秀。由于小组成立之初各位成员都没有相关的背景知识，所以选择了更为保守和简单的方向，没有机会尝试深度学习解决方案也是遗憾之一。

2. 代码风格

在完成代码的过程中，由于不同组员都需要对代码进行阅读和改写，因此增加必要且

准确地注释是非常必要的。后期我们为代码添加了必要的注释后，代码的可读性大大增加，提高了小组共同开发的效率。

3. 版本管理

这次实习作业我们前后一共编写了许多版本的 AI，但由于之前没有大规模编程的经验，小组没有使用版本管理平台，各个版本的代码分享都是在微信群中进行的，每个版本修改的内容在时间长了后也都需要查阅相关的聊天记录才能回忆起来。在后期这样的版本管理就变得有些混乱，如果能够从一开始就使用 GitHub 等版本管理平台，想必可以有效解决这一问题。

4.3 建议与设想

总体而言，感觉本次大作业还是让我们有了很多收获的，特别是我们四个人作为社科学生，之前都没有体验过这种程序/项目设计的经验。

主要的收获是体会到 Python 的强大，几十行代码即可以做出一个 AI，几百行可以做出一个很强的 AI，这些都是我之前没想到的。同时和同学们一起从想策略、实现策略、复盘、debug 的过程也是非常宝贵的。

建议方面我认为本次游戏的选取可以相对简单，可操作空间不大，从而实现起来实际主要还是应用的是基本线性结构和判断、循环语句，负责数据结构和算法的用武之地不是很大。或者也可能是仅仅靠课内所学和课外不多的积累，很难用复杂算法实现更多内容。所以或许选择步骤较多的（如棋牌类）可以更能引导我们应用学习的复杂的树和图的算法。同时在赛制上，采用小组出线的方式，一次性决胜负，没有轮流对战。着或许可以有效节省时间，但是存在很多队伍一轮游的现象，同时分组的影响也会比较大。如果在正式比赛前先进行相互对战，或许效果很更好，最终的结果也更为公平。

5 致谢

首先感谢陈斌老师和助教们的创意和辛勤工作，为我们提供了一个非常优秀的平台来施展锻炼自己的编程能力和算法设计能力，让我们可以对本学期学过的大部分数据结构与算法知识进行一次综合利用，既享受着游戏对战的乐趣，也能在实战过程中对所学知识融

会贯通，加深理解。尤其感谢技术组在热身赛中对我们的帮助，使得我们能一遍遍地优化程序。其次感谢和我们进行友谊赛的小组，感谢他们帮助我们不断发现自身问题并进行改正。最后也要感谢组内的每个人，认真的完成自己的任务，最终使我们圆满完成课程实习！

6 参考文献

堪称最好的 A* 算法 https://blog.csdn.net/b2b160/article/details/4057781

A*（A 星）算法 python 实现 https://blog.csdn.net/nobugtodebug/article/details/43986597

第十二章　N17_Lima 报告

周忠鹏 *、陈鑫燚、丁锐奇、侯松林、曹柱

摘要：本算法的主要思路为螺旋圈地，只圈地，不进攻。圈地模式比较固定，面对进攻性比较强的对手胜率较低，但面对无思考能力的 AI 时有时可取得胜利。本算法在测试环节无明显 BUG，能正常运行，但在比赛中由于代码强度有限，一场未胜，未能出线。

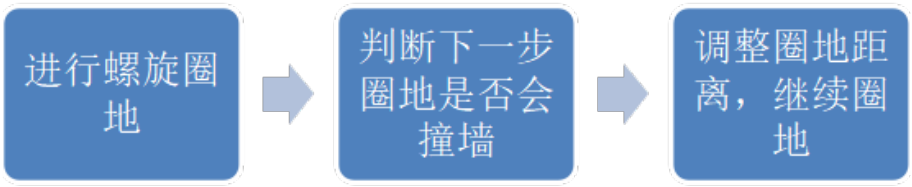
关键字：螺旋圈地

1 算法思想

1.1 总体思路

本算法的总体思路是采用螺旋式方法进行圈地，只圈地，不进攻。由于纸带随时离本方领地较近，对于一些攻击性较强的算法可以有效实施防御。但由于圈地效率太低，很难在圈地方面取得优势。

1.2 算法流程图



1.3 算法运行时间复杂度分析

本算法运行时间主要由螺旋圈地的循环过程消耗，复杂度为 O（1），在实际运行中时间开销小于为 0.01s。

2 程序代码说明

2.1 数据结构说明

算法采用的主要数据结构为顺序结构和循环结构等基本结构，未使用自定义类。

2.2 函数说明

本算法主要使用了 play，load 函数，play 函数用来实现储存功能，load 函数用来实现调用功能，其中 load 是主要函数，它能判断纸带运动是否撞墙，控制纸带的行进和转向，从而达到螺旋圈地的目的。

2.3 程序限制

由于本算法内的 load 函数不具备判断场地大小的功能，所以在场地较小时可能会出现圈地过程中撞墙的情况，这也是本算法的最大限制。

3 实验结果

3.1 测试数据

实验环境说明：

硬件配置：（CPU/内存）Intel（R）Core（TM）i5-4200U @ 1.60GHz 2.30 GHz/4.00 GB

操作系统：（名称/版本）WIN7 旗舰版 64 位操作系统

Python 版本：（版本号）3.5

采用了技术组提供的 AI 算法进行测试，面对技术组提供的 AI 算法时能达到约 40% 以上的胜利，其中大部分情况都是在步数耗尽之后，以圈地面积较大而获胜。

3.2 结果分析

本算法的总体思路是采用螺旋式方法进行圈地，只圈地，不进攻。由于纸带随时离本方领地较近，对于一些攻击性较强的算法可以有效实施防御。但由于圈地效率太低，很难在圈地方面取得优势。而且随场地大小的变化，本算法无法实现灵活转变，很有可能最后因撞墙导致比赛失败，这也是本算法需要改进的地方。

4 实习过程总结

4.1 分工与合作

小组中五名同学均在本次大作业中有所贡献，对代码的完成和报告的完成均能提出自己的意见和建议，共同完成了本次大作业。由于时间比较紧迫，所以暂时没有照片。

4.2 经验与教训

（1）整个大作业完成过程中，较好的应用到了我们这学期学到的知识和方法，具有很强的实践性。

（2）由于大作业和大四同学的本科答辩时间严重冲突，再加上本小组为最后没有分到组的同学组建起来的小组，导致出现了代码强度不够、考虑不周全等一系列问题，这也是本次大作业需要改进的地方。

4.3 建议与设想

在本次实习中，小组时间安排得不太合理，导致许多设想的算法功能没有完成。在实习作业的设置方面，希望能提供不同难度的实习作业供不同知识水平的同学们进行选择，

并根据其难度在分数上也设置分层；对于基础知识相对薄弱的同学，可以对作业中可以用到的数据结构进行提示，不然这种开放性的作业对于没有时间的大四同学和没有编程基础的同学来说还是有一定难度的。

5 致谢

感谢技术开发组提供的基础代码和供大家参考的 AI，感谢助教们在平时的学习过程中对我们的帮助，感谢陈斌老师本学期带给我们这样精彩的课程，感谢全体组员的付出。

6 参考文献

https://github.com/chbpku/paper.io.sessdsa

第十三章　N17_Mike 报告

贺海军 * 明波 黄智强

摘要：依托于 paper-io 基础设施，通过字典、列表和栈等基本数据结构设计出的人工智能算法能够保证不碰撞边界和己方纸带的情况下以矩形扩展己方面积，通过与示例算法对战，能够取得约 90% 胜率。

关键字：stack dict list offense

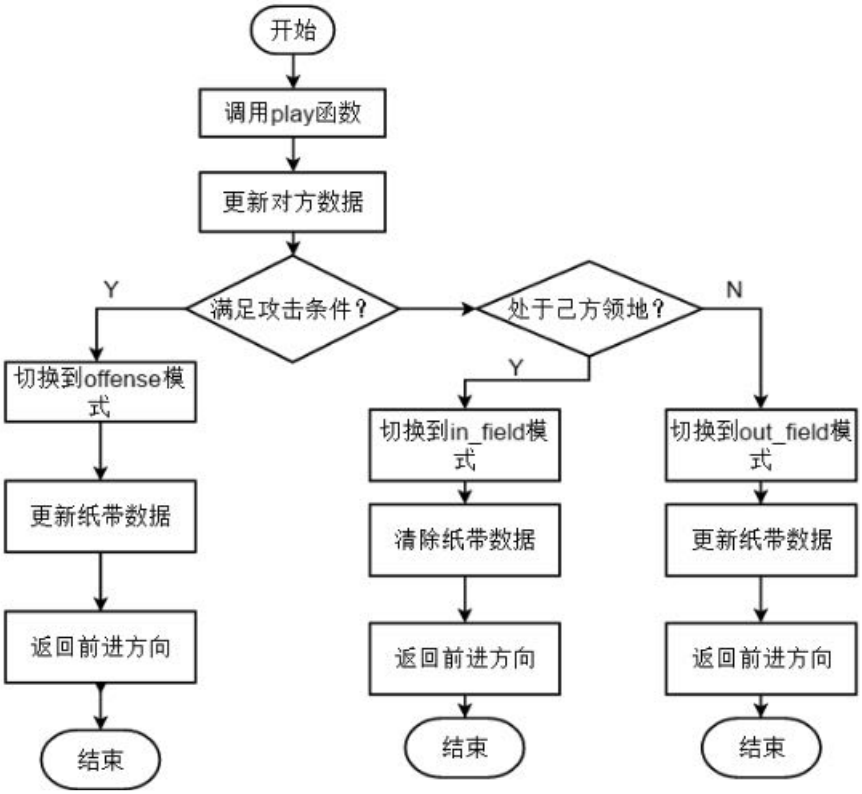
1 算法思想

1.1 总体思路

首先要满足 3 个基本条件。一是不能撞墙，检测头与围墙的距离，当距离为 1 时，转向。二是自己不能撞自己的纸带，检测头与纸带的距离，使头不能撞到纸带。三是头的行动规则只能是前进、左转和右转。

在满足上述的 3 个基本条件下，进行攻击和防御。当对方头在己方领地时，朝对方纸带前进，选择最短路进。当双方处于在空白区交战状态时，分情况考虑。设己方的最短回防距离为 d1，对方的最短回防距离为 d2；己方的最短攻击距离为 a1，对方的最短攻击距离为 a2。当 a1<=d2 时，放弃圈地，朝对方纸带进攻。当 a2-d1=3 时，放弃圈地，选择最短路劲回防。当 a2<d1 时，此时己方必然无法安全回防，选择朝对方头前进，而非纸带，迫使对方回防，以求最后一搏。

1.2 算法流程图



1.3 算法运行时间复杂度分析

算法在 in_field 模式和 out_field 模式时间复杂度较低，只需要进行少量的计算，返回前进方向；在 offense 模式进行攻击时，需要进行大量判断，时间开销较大；

2 程序代码说明

2.1 数据结构说明

算法中主要采用了

1. 字典和列表、字符串的混合使用，即以列表和字符串构建的字典，主要用于 storage 对战数据的存储；

2. 采用栈用于记录纸带信息，并在纸带状态发生改变时进行压栈或者出栈操作；

2.2 函数说明

函数 play，读取当前地图的全部状态 stat、并更新存储数据 storage，调用相应模式下的计算函数，得到正确的方向，函数接口为（stat，storage）

函数 distance，用于计算两个点之间的距离

函数 enemy_update，更新 storage 中有关的对方数据

函数 load，初始化 storage 字典，定义三种模式用于 paly 函数调用，得到不同情况下的前进方向，并进行己方数据更新.

函数 distance，用于计算两个点之间的距离

函数 update_min_dist、update_band、update_infield，用于在各种模式下的 storage 己方数据更新，包括最短攻击距离、纸带信息更新和进入己方领地后的数据清除

函数 towards、towards_copy，用于计算到朝向一个点的方向，两个函数分别以 x 方向和 y 方向作为基准

三种模式之间进行切换，调用参数保持一致：field, band, me , enemy, storage

函数 in_field，当纸卷处于己方领地时调用，将纸带信息清空，保证不出边界，向着对方纸卷所在方向扩展，如遇对方纸卷在己方领地，切换到 offense 模式

函数 out_field，当纸卷处于非己方领地时调用，在每一步的操作下更新纸带数据，在不满攻击条件下，画矩形扩展领地面积；如满足攻击条件，放弃拓展面积，进入攻击模式

函数 offense，攻击模式，由 in_field 或者 out_field 模式切换而来，向着对方纸带的最近点攻击、或者攻击对方纸卷、造成对方自杀或撞墙

2.3 程序限制

（1）在 offense 模式和 out_field 模式之间进行多次复杂切换的时候，由于受到对方的引诱，可能会陷入多次转向后不得已自杀的局面；

（2）采取矩形圈地是较为保守的圈地方法，并不能最大程度地扩展面积，在和局时不

占优

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) i7-8550U CPU @ 1.80GHz 2.00GHz

操作系统：Windows 10 专业版

Python 版本：Python 3.5(64-bit)

3.2 结果分析

程序正常运行，没有明显 Bug。可以保证纸头不会出现撞墙和自杀等行为。由于没有参加热身赛，故没有其他说明。

4 实习过程总结

4.1 分工与合作

明波：撰写报告 + 少量编写代码。

贺海军：编写代码 + 思路提供。

黄智强：编写代码 + 思路提供。

合作与交流的方式：线下讨论与线上交流。

组会照片：

4.2 经验与教训

（1）整个大作业完成过程中，较好的应用到了我们这学期学到的知识和方法，具有很强的实践性。

（2）大作业时间与本科论文写作及提交时间重了，对于大四学生来说时间紧，不能很

好的完成大作业。

4.3 建议与设想

大作业的时间希望能在开学时就启动，这样对于大四学生来说能有更充裕的时间。同时，比赛时能进行视觉化的比赛更好，只是统计数据让观者无聊；就像世界杯，只有看直播才有激情。

5 致谢

感谢老师和助教这一学期以来的教学和指导。老师上课风趣幽默，给我们学习这门课程提供了很好的氛围，传达学习内容的方式也很有意思，包括这次实习大作业。助教们也辛苦啦，每次批改作业等都很及时有效。

最后，必须要感谢这次一起完成实习作业的小伙伴们，以此次实习大作业纪念我们在本科期间的最后一次作业。

第十四章　N17_November 报告

杨少栋 赵钰浓 徐双余 熊韦 何昊 哈尔西哈·库尔曼江

摘要：本组算法主要采用『近攻远圈地』策略，整体 AI 由多种策略组合协同作用。『近攻』指敌人或敌方纸带较近时，算法自动切换为攻击模式，尝试击杀对手。『远圈地』指敌人较远或敌人在敌人自己领地内时，算法自动切换为圈地模式，尝试获得尽可能多的土地。前述两种模式由多种策略配合实现，并辅助以其他模式（如撤退模式和领地内防守模式）。主要采用的算法为贪心算法、宽度优先搜索和最短路径算法，同时采用 Python 基础数据结构元组，列表，字典等，并构造使用了栈和队列等数据结构。经过小组成员的努力，最终算法在比赛中取得了小组第三的好成绩。

关键字：栈、队列、贪心算法、宽度优先搜索、最短路径算法

1 算法思想

1.1 总体思路

本组算法主要采用『近攻远圈地』策略，整体 AI 由多种策略组合协同作用。『近攻』指敌人或敌方纸带较近时，算法自动切换为攻击模式，尝试击杀对手。『远圈地』指敌人较远或敌人在敌人自己领地内时，算法自动切换为圈地模式，尝试获得尽可能多的土地。前述两种模式由多种策略配合实现，并辅助以其他模式（如撤退模式和领地内防守模式）。主要采用的算法为贪心算法、宽度优先搜索和最短路径算法，同时采用 Python 基础数据结构元组，列表，字典等，并构造使用了栈和队列等数据结构。

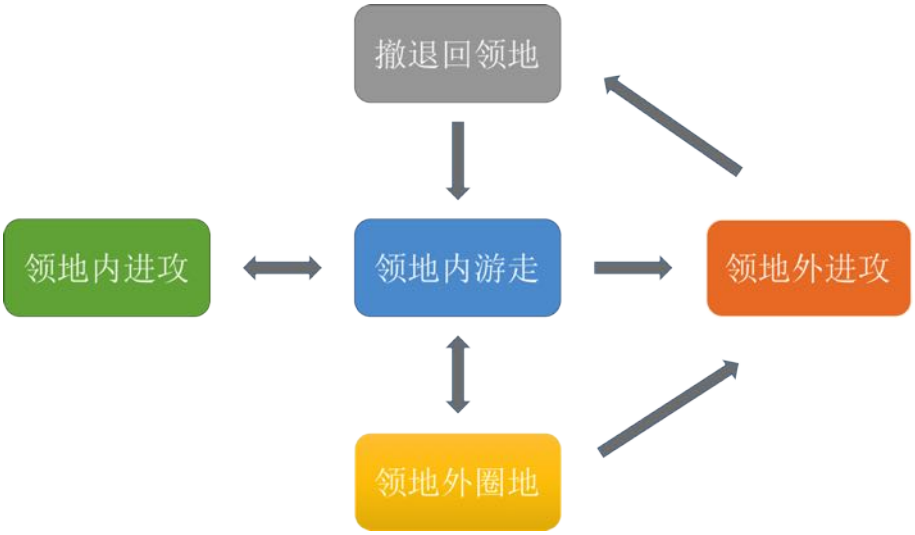


图 1 算法整体框架图

算法整体框架如上图，以下我们具体介绍每一模式使用算法。

1.2 细分策略

1.2.1 领地内游走

『领地内游走』为算法初始模式，此时自己纸卷位于自己领地内。如图 1 所示，可以由该模式主动切换到除『撤回领地』模式外的其余三个模式。该模块为算法的主控模块，扮演中枢的角色，针对不同的场面局势，承担调度功能。

场面局势 1，敌人较『远』。

敌我之前距离计算为简单计算敌我纸带的水平距离加垂直距离，代码如下所示：

```
# 计算距离
def dist(me, enemy):
    return abs(enemy['x'] - me['x']) + abs(enemy['y'] - me['y'])
```

敌人『远』也分为两种情形：

1. 敌人在入侵自己领地：

此时判断敌我距离，如果较近则切换到『领地内攻击』模式，如图 2 红色箭头所示。算法控制纸卷在领地内撞击对手或其纸带，击杀对手。

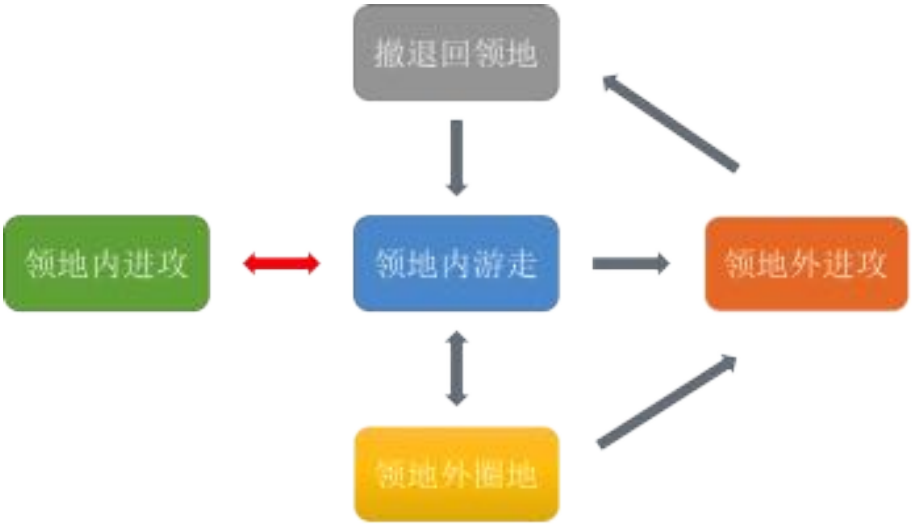


图 2 领地内攻击对手模式切换

2. 敌人不在己方领地内：

此时，以贪心策略选择最近的边界让纸卷出去圈地，从而转换为『领地外圈地』模式，如图 3 红色箭头所示。

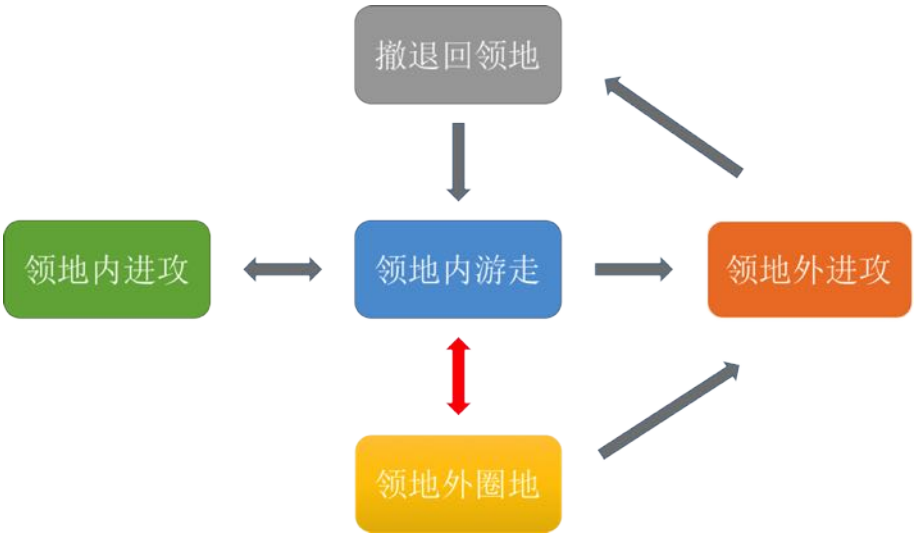


图 3 领地内攻击对手模式切换

上述两模式切换对应代码如下：

```
# 状态转换
if field[me['x']][me['y']] != me['id']:
    storage['mode'] = 'square'
    storage['count'] = randrange(1, 3)
    storage['turn'] = choice('rl')
    storage['earlyturn'] = True
    storage['medis'] = dist(me, enemy)
    storage['maxl'] = max(4, dist(me, enemy) // 3)
    return ''
else:
    if field[enemy['x']][enemy['y']] == me['id']:
        # 敌人入侵 且 距离较近
        return attack(field, me, enemy, storage, bands)
    return gready(field, me, enemy, storage)
```

这里，切换到圈地模式时，根据当前敌我距离动态选择圈地步长，并配置好其他所需参数信息。具体领地内进攻策略见下一节 1.1.2 领地内进攻，下面介绍一下贪心策略尽快走出自己领地去圈地的策略。

如上图代码所示，返回 greay() 函数方向，在 gready() 函数中，通过计算当前纸带到东南西北四个方向自己领地边界距离，贪心选择距离最近的方向走出领地去圈地。由于每一步都选择最近的方向，故保证能从最近的边界走出自己的领地前去圈地。核心代码如下：

1. 找一个可行方向备用，防止所选方向越界

```
# 找到一个可行的下一步，备用
offset = [0, 1, 3]
op = []
for ofs in range(3):
    nextx = me['x'] + directions[(me['direction'] + offset[ofs]) % 4][0]
    nexty = me['y'] + directions[(me['direction'] + offset[ofs]) % 4][1]
    if check_xy(nextx, nexty, len(field), len(field[0])):
        op.append('xrl'[ofs])
```

2. 以东方向为列，其余方向代码类似

```
dislist = []
# 东
dis = 0
while ((me['x'] + dis) < (len(field) - 4) and
field[me['x'] + dis][me['y']] == me['id']):
    dis += 1
# 边界贴近地图边缘，则不倾向此方向
if (me['x'] + dis) == (len(field) - 4):
    dis = INF
dislist.append(dis)
```

3. 贪心选择最终方向

```
argvmindis = dislist.index(min(dislist))
if me['direction'] == 0 and argvmindis == 3:
    res = 'l'
if me['direction'] == 3 and argvmindis == 0:
    res = 'r'
if me['direction'] < argvmindis:
    res = 'r'
elif me['direction'] > argvmindis:
    res = 'l'
else:
    res = 'x'

res = res if res in op else op[0]
return res
```

算法在执行过程中，通过该策略可以最大程度的减少在领地内行走的步数，从而能有更多的步数用来扩张领地，如图 4 可以看出先手绿色玩家选择最近的方向走出领地。

图 4 贪心策略演示

1.2.2 领地内进攻

算法控制纸卷在领地内撞击对手或其纸带，击杀对手。这里考虑加入此时敌我领地面积的因素，如果差距太大，则有可能放弃进攻而转而回到『领地内游走』模式，如图 2 中红色双向箭头所示。考虑到领地内击杀对手收益更大，所以面积因素只起较小的影响。代码如下：

```python
# attack enemy
def attack(field, me, enemy, storage, bands):
    '''
    前提: me 在自己领地内
    基本思路:
        敌人侵犯自己领地内时:
            if 敌我领地面积比 * 距离 < 阈值 (可调参, 瞎选1/4场地长) :
                attack
            else:
                (gready) 继续扩大面积
    '''
    dis = dist(me, enemy)
    myLand, enemyLand = getLandS(me, enemy, field)
    thres = max(len(field), len(field[0])) // 4
    if (enemyLand / myLand) * dis < thres:
        aheadPoint = get_nearest_attack_point(me, enemy, field, bands)
        return ahead_enemy(me, aheadPoint)
    else:
        return gready(field, me, enemy, storage)
```

其中，进攻返回 ahead_enemy() 函数的返回的方向。因为是在自己领地内进攻，所以只要撞击敌人纸卷或纸带即可。该函数通过判断敌人相对于自己方向，而选择前进方向。这里需要注意的是，传入的 enemy 参数并不一定是敌人纸卷的位置，而是可以最近攻击到敌人的位置，由 get_nearest_attack_point() 函数返回。该函数通过遍历敌人纸卷和纸带对应坐标，从而选出最近的攻击点。代码如下：

```python
# 获得最近的可攻击点
def get_nearest_attack_point(me, enemy, field, bands):
    point = ''
    res = {}
    minDis = INF
    for i in range(len(field)):
        for j in range(len(field[0])):
            if field[i][j] == me['id'] and bands[i][j] == enemy['id']:
                tmpDis = abs(i - me['x']) + abs(j - me['y'])
                if tmpDis < minDis:
                    point = (i, j)
    res['x'] = point[0]
    res['y'] = point[1]
    return res
```

返回向地方进攻方向的代码如下：

```
# 控制 me 向 enemy 进攻
def ahead_enemy(me, enemy):
    x_ = me['x'] - enemy['x']
    y_ = me['y'] - enemy['y']
    d = me['direction']
    res = ''
    if d == 0:
        if y_ == 0:
            res = choice('rl') if x_ > 0 else 'x'
        elif y_ > 0:
            res = 'l'
        else:
            res = 'r'
    elif d == 1:
        if x_ == 0:
            res = choice('rl') if y_ > 0 else 'x'
        elif x_ > 0:
            res = 'r'
        else:
            res = 'l'
    elif d == 2:
        if y_ == 0:
            res = 'x' if x_ > 0 else choice('rl')
        elif y_ > 0:
            res = 'r'
        else:
            res = 'l'
    else:
        if x_ == 0:
            res = 'x' if y_ > 0 else choice('rl')
        elif x_ > 0:
            res = 'l'
        else:
            res = 'r'

    return res
```

算法在执行过程中，通过该策略可以通过在领地内直接击杀敌人得分，如图 5 可以看出先手绿色玩家在自己领地内击杀对手得分。

图 5 领地内击杀对手演示

1.2.3 领地外圈地

如 1.1.1 节所述，该模式由『领地内游走』变换过来，该模式也承担着部分控制的任

务。如果敌人较远时，则圈地；而如果附近有可以攻击的敌人，则切换到『领地外进攻』模式，尝试击杀敌人。

该模式也同时包含几个策略，总体采用画矩形的方式圈地。首先，判断当前前进方向是否有自己领地，有则在一定安全的条件下直走，减少画圈的步数。

如果敌人较近，敌我距离小于参数 d 时，BFS 分别计算进攻敌人的步数、敌人进攻自己的最小步数和自己返回自己领地的最小步数。这里使用经验参数 $d = 30$ 步衡量敌人的远近，因为 BFS 比较耗时，所以没有每一步都计算前述这三个值，而是折中在敌我距离小于一定范围时考虑是否进攻。这样同时避免了算法全局一直追着对手进攻而忽略圈地的情况。

当攻击步数小于被攻击步数，且敌人不在敌方领地内，则攻击。否则，当到达前方领地或者返回领地的步数大于危险距离，则返回。否则，如果前方有自己的领地则前进，而不倾向于转弯。

其中，计算进攻方向以及最小进攻步数等见下一节 1.1.4 领地外进攻。这里详述判断钱进方向是否有自己领地，并减少转弯的贪心策略。代码如下：

```python
# 检查当前方向上是否有自己领地
def has_land_ahead(field, me, enemy, storage):
    hasland = False
    nextx = me['x'] + directions[me['direction']][0]
    nexty = me['y'] + directions[me['direction']][1]
    cnt = 0
    while (nextx >= 0 and nextx < len(field)) and (nexty >= 0 and nexty < len(field[0])):
        if field[nextx][nexty] == me['id']:
            hasland = True
            break
        # if field[nextx][nexty] == enemy['id']:
        #     hasland = False
        #     break
        cnt += 1
        nextx += directions[me['direction']][0]
        nexty += directions[me['direction']][1]
    dis = cnt if hasland else INF
    return hasland, dis
```

整体领地外圈地代码如下：

```
# 判断前方是否有自己领地
hasLand, landDis = has_land_ahead(field, me, enemy, storage)

if dist(me, enemy) < 30:
    # 计算敌人攻击自己的最小步数
    (_, dangerDis) = get_attack_dis(field, enemy, me, storage, bands)
    # 计算自己攻击敌人的最小步数和方向
    (attackWay, attackDis) = get_attack_dis(field, me, enemy, storage, bands)
    # 计算返回领地的最小步数和方向
    (homeWay, homeDis) = get_home_dis(field, me, enemy, storage, bands)
    # print ('attDis: %d ; danDis: %d' % (attackDis, dangerDis))
    if attackDis < dangerDis and field[enemy['x']][enemy['y']] != enemy['id']:
        # 攻击步数小于被攻击步数，且敌人不在敌方领地内，则攻击
        storage['mode'] = 'kill'
        return attackWay

    if attackDis > dangerDis or landDis > dangerDis or homeDis > dangerDis:
        # 到达前方领地或者返回领地的步数大于危险距离，则返回
        storage['mode'] = 'gohome'
        return ''

# 前方有自己的领地，且敌人较远时，倾向于前进
if hasLand:
    return 'x'

storage['count'] += 1
if storage['count'] >= storage['maxl']:
    storage['count'] = 0
    return storage['turn']
```

算法在执行过程中，通过该策略可以在领地外决策调控，或选择画矩形圈地，或选择贪心前进减少圈地步数，或选择去进攻。如图 6 所示，先手绿色玩家通过直行减少了圈地步数，本来到达转向的步数该沿着红线前进，然而执行的步数更短且能圈得更大面积，于是纸卷继续直行。进攻演示见下一节 1.1.4 领地外进攻。

图 6 贪心圈地演示

1.2.4 领地外进攻

如图 7 所示，当纸卷圈地过程中，发现可以进攻则切换到『领地外进攻』模式。

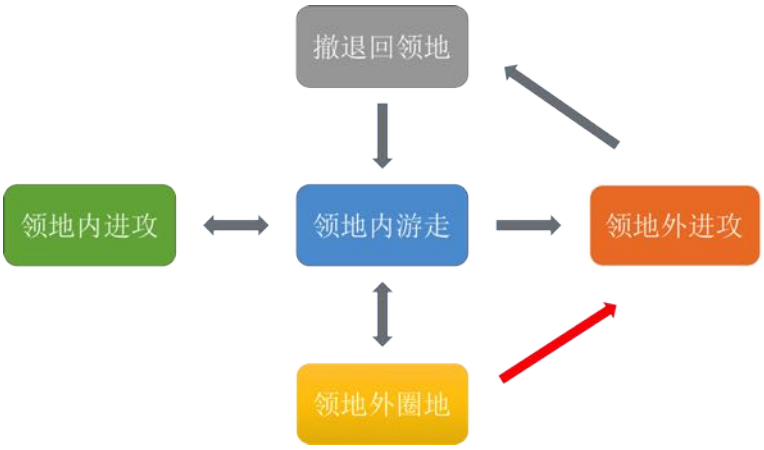


图 7 领地外进攻模式切换

主要算法，BFS 计算进攻的最短路和方向。分前进、左转和右转三个方向，bfs 找到进攻的最短路径，同时返回第一步操作方向。

```python
# 计算最短攻击距离
def get_attack_dis(field, plr1, plr2, storage, bands):
    # 返回 plr1 攻击 plr2 的最短步数，和起始方向
    dis = []
    res = 'xlr'
    # 'x' 方向
    x_dis = bfs_attack_path(field, plr1, plr2, storage, 'x', bands)
    dis.append(x_dis)
    # 'l' 方向
    l_dis = bfs_attack_path(field, plr1, plr2, storage, 'l', bands)
    dis.append(l_dis)
    # 'r' 方向
    r_dis = bfs_attack_path(field, plr1, plr2, storage, 'r', bands)
    dis.append(r_dis)

    return res[dis.index(min(dis))], min(dis)
```

其中，BFS 核心算法为 bfs_attack_path() 函数。核心算法部分如下：

```
# 越界
if not check_xy(nextx, nexty, len(field), len(field[0])):
    return INF
if field[nextx][nexty] == me['id']:
    return 1
visit[nextx][nexty] = 1
node = []
node.append([nextx, nexty, 1, curDirection])
while (len(node) > 0):
    cnode = node.pop(0)
    step = cnode[2] + 1
    curDirection = cnode[3]
    # 尝试 x, r, l 方向
    for ofs in [0, 1, 3]:
        newDirection = (curDirection + ofs) % 4
        nextx = cnode[0] + directions[newDirection][0]
        nexty = cnode[1] + directions[newDirection][1]
        if (check_xy(nextx, nexty, len(field), len(field[0])) and (bands[nextx][nexty] != me['id'])):
            if (field[nextx][nexty] == me['id']):
                # 成功返回领地
                return step
            else:
                if visit[nextx][nexty] == 0:
                    visit[nextx][nexty] = 1
                    node.append([nextx, nexty, step, newDirection])
return INF
```

在 1.1.3 节中提到的计算敌人攻击自己和自己攻击的人的最短路径都复用上述代码，bfs 找返回自己领地的路径算法框架同上，只是结束条件略有区别，这里不在赘述。

在进攻过程中，如果敌人返回了敌人的领地，则放弃进攻，如图 8 红色箭头所示切换到『撤退回领地』模式。

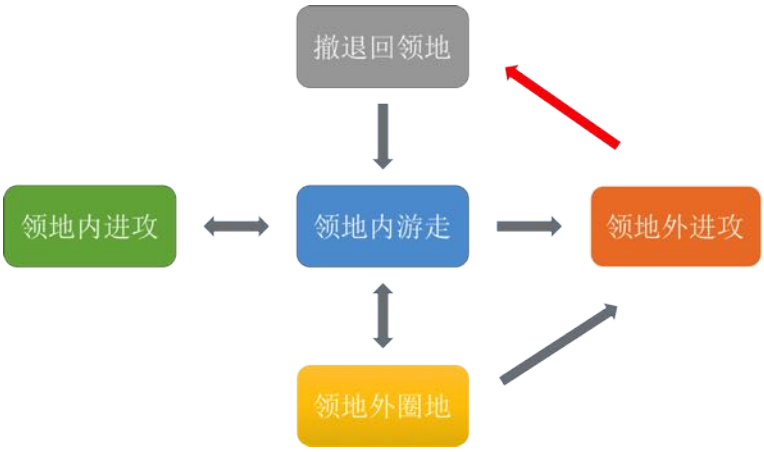


图 8 测退回领地模式切换

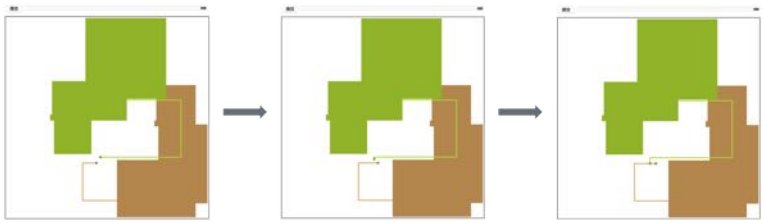


图 9 领地外进攻演示

1.2.5 撤退回领地

考虑到在进攻过程中，敌人可能率先返回敌方领地，于是失去攻击目标，所以增加撤退回领地模式。算法思路大体同 1.1.4 节，BFS 找到最短回家的第一步方向，执行该方向。核心算法见函数 bfs_home_path()，核心代码如下：

```
# 越界
if not check_xy(nextx, nexty, len(field), len(field[0])):
    return INF
if field[nextx][nexty] == me['id']:
    return 1
visit[nextx][nexty] = 1
node = []
node.append([nextx, nexty, 1, curDirection])
while (len(node) > 0):
    cnode = node.pop(0)
    step = cnode[2] + 1
    curDirection = cnode[3]
    # 尝试 x, r, l 方向
    for ofs in [0, 1, 3]:
        newDirection = (curDirection + ofs) % 4
        nextx = cnode[0] + directions[newDirection][0]
        nexty = cnode[1] + directions[newDirection][1]
        if (check_xy(nextx, nexty, len(field), len(field[0])) and (bands[nextx][nexty] != me['id'])):
            if (field[nextx][nexty] == me['id']):
                # 成功返回领地
                return step
            else:
                if visit[nextx][nexty] == 0:
                    visit[nextx][nexty] = 1
                    node.append([nextx, nexty, step, newDirection])
return INF
```

算法在执行过程中，通过该策略可以在进攻不成功时成功撤退。如图 10 演示了先手玩家在最初选择进攻，不成功后，果断返回自己领地的防守策略。

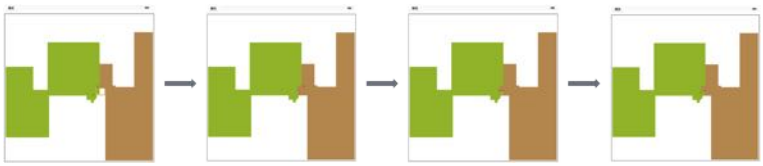


图 10 进攻失败返回领地演示

2 实验结果

2.1 测试数据

实验环境说明：

硬件配置：CPU：2.6 GHz Intel Core i5 内存：8 GB 1600 MHz DDR3

操作系统：macOS Sierra

Python 版本：3.6.1（运行平台：Anaconda）

测试过程说明：

1. 首先我们初步写了一个贪心版本 AI，与技术组提供的 AI（AI_simple_wander.py）进行了多组比赛，均取得了胜利。

2. 改进第一版本 AI 后，参加第一次热身赛，排名第三。我们进一步分析对局，发现领先我们的策略在圈地过程中更加鲁棒。同时有一些进攻的策略。于是我们开始设计自己的进攻策略。

3. 我们增加了领地内的进攻策略，然后恰好赶上第三次热身赛。这一次我们的结果不是很理想，排名较后。我们认真复盘总结了此次比赛，发现参加热身赛的队伍都有较大幅度的提升，相比于他们，我们 AI 提升幅度就要小多了。特别是圈地过程中的自我保护和进攻这一方面，我们尤其欠缺。

4. 通过上一次热身赛的总结，我们新增了圈地过程中自我保护和进攻的策略，同时加强了圈地策略。这次改动较大，于是在第三次热身赛时，仓促参赛，进攻策略存在 bug 未解决。于是这一次热身赛的排名没有什么起伏，我们再次认真总结复盘。问题同 3 中描述大体相同，我们抓紧时间修复 bug，并优化代码，使其更加鲁棒。最后一版代码完成，但已经没有热身赛让我们再次测试代码或者调参。

5. 最后比赛小组中我们获得第三，遗憾未能进入八强出线。我们有些许遗憾，本来应该能做的更好。

2.2 结果分析

我们提出了很多优秀的算法或策略，也基本实现了所有的策略算法，但由于中途进度问题，使我们仓促参加了一次热身赛，没有充分利用那次热身赛进行调参和策略选择。总的来说，我们算法较健壮，策略较新颖，也一定程度上达到我们的期望，同时也留有一些遗憾。

优点：贪心策略的多处使用，使算法尽可能处于一种优势局面。BFS 算法找最短进攻

路径，高效且高收益的攻击敌人。

缺点：未实现一开始设计的多回合数深搜和全局局势博弈，圈地可以通过实时分析对手行为而做到更加『智能』。

3 小组分工

本小组的同学都为大五的同学，许多同学都面临找工作的压力。因此，为了减轻小组同学的负担，我们共同进行了两次线下讨论，并分成三组，每组自行组织线上和线下讨论。其中杨少栋、赵钰浓为一组，负责主要策略的构思和代码实现；徐双余和熊韦为一组，负责反馈 Github 平台的更新以及汇总组长群中的“敌方策略”；哈尔西哈和何昊主要负责最终报告的撰写。同时，组内全部成员均共同参与分策略代码的分析和调试。

同时，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，通过视频讨论的方式，他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。

4 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。

5 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

第十五章　N17_Oscar 报告

丁力 *、范启东、吕博文、贺雨岑、李沐航、李承

摘要：数据结构与算法：本组采用的主要数据结构和算法有列表、字典、条件控制语句、循环语句、递归算法等。总体思路：在做好防守的前提下，在不同阶段采取不同的圈地/攻击策略。实验结果：我方一共胜利 38 局、失败 39 局。其中失败的最大原因是撞墙而死，胜利的原因最常见的是撞击对手纸带。

关键字：列表；字典；条件控制语句；循环语句；

1 算法思想

1.1 总体思路

此次大作业，本组纸带圈地算法的总体思路是：在做好防守的前提下，在不同阶段采取不同的圈地/攻击策略。

第一阶段：圈地阶段。比赛开始时先向我方场地方向行进并圈一个尽量大的长方形。第二阶段：挺进阶段。向对方的方向行进，借助领地为 1-3 条边圈多个方形，尽量接近对方。时刻警惕被攻击/攻击可能性，采取相应行动。

第三阶段：僵持阶段。当两方接壤时要吞吃敌方的领地，加大我方领地面积。此时双方距离较近，要时刻警惕被攻击/攻击可能性。时刻准备好攻击，但要以防守为主。

我们组采用的主要数据结构和算法有列表、字典、条件控制语句、循环语句、递归算法等。其中最为典型的则是对条件控制语句的应用——通过对当前各种条件的判断选择之

后一步或几步的行动策略，比如在圈地阶段中需要判断敌我距离、敌人回自家或我方回自家的距离，来确定之后是继续圈地还是向对方展开攻击或者苟回自方领地。所有的判断条件与策略都是事先确定好的，因此运算上来看应该是相对简单的，但也导致面对复杂情形难以变通，同时由于难以将条件划分至完备，这也是导致我们算法在面临某些特定状况时出现不合适的行为，比如在后文中我们将提到的撞墙自杀的行为等。

我们的算法基本符合最初的设想，在如下部分有些许调整：

1 圈地阶段，由于编程能力限制，一开始时写的是圈一个正方形：计算比赛开始时敌我双方纸卷的曼哈顿距离 n，算 1/4 n，向墙的方向走出第一条边，每走够 1/4 n 即转弯，直到走出一个正方形。

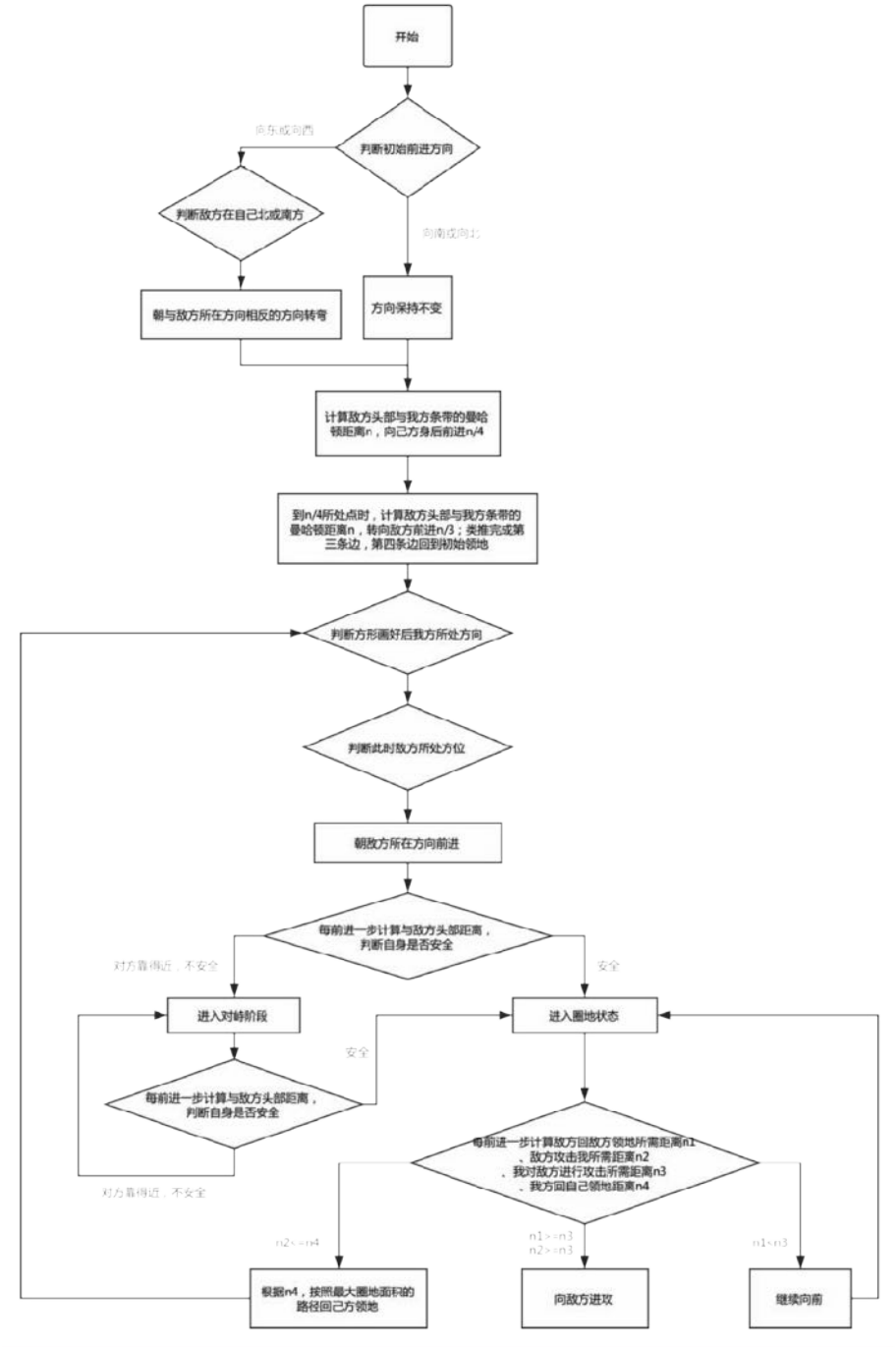
然而这种方式是最保守的圈地方式，基本不能体现 AI 临场应变抓住机会圈大地的能力。于是在最终版把圈地改为圈动态调整的长方形：计算比赛开始时敌我双方纸卷的曼哈顿距离 n1，算 1/4 n1，向墙的方向走 1/4 n1 长的第一条边。然后重新计算敌我双方纸卷的曼哈顿距离，算新的 n2，向敌方方向转向，走 1/3 n2 长的第二条边。类推第三条边。第四条边回到开始点，圈出一块长方形领地。

2 攻击算法。一开始我们想通过先/手优势来设计一击制胜的攻击算法。

攻击算法包括必杀判定和必杀路径输出两个部分。如果判断我方攻击对方的最短距离不大于敌方回到领地的最短距离（对方不能逃脱）和敌方攻击到我方的最短距离（对方不能攻击到我方）。这个是初步判定攻击条件。在实际操作过程中发现，在尝试进攻期间敌我双方的动态移动可能导致上述条件不满足的情况。我们将进攻路径进行模拟，防止进攻路径上的意外发生。

我们发现，在侧撞对手纸带时，如果我方为先手，则在开始攻击的一刻双方纸卷曼哈顿距离为奇数我方必赢。如果我方为后手，则在开始攻击的一刻双方纸卷曼哈顿距离为偶数我方必赢。一旦开局，奇偶优势永远不变，理论上没有优势的人不能直接攻击。然而在后期写代码过程中受时间、能力所限我们并未将此优势应用到实战中。

1.2 算法流程图



该部分我按照之前给出的代码名称给出我部分的时间复杂度。仅供参考。

下面给出主要功能函数的在一个回合时间复杂度分析。不妨假设棋盘为边长为 n 的正方形。

函数名	功能	实现方式	时间复杂度
renew FieldList	在判断任意一方纸卷回到自己领地时，更新双方的领地点的集合	遍历领地状态的矩阵，寻找各自 id 对应的点，加入到领地集合之中。	O(n2)
renewBondlist	有纸带生成或者消失的时候，更新双方纸带的点的集合	遍历纸带状态的矩阵，寻找各自 id 对应的点，加入到纸带集合之中。	O(n2)
myAttackDistance	计算我方进攻敌方的最短距离	遍历敌方纸带点的集合，计算我方纸卷和对方纸带的曼哈顿距离。	O(1) O(n).
取决于敌方的纸带长度。一般情况下更加接近于 O(1)，因为纸带是稀疏的 enemyAttackDistance	和上面类似	和上面类似	O(1) O(n)
enemyWithdrawDistance	计算敌方撤退距离	遍历敌方领地点的集合，计算敌方纸卷和地方领地点的距离	O(1) O(n2).
取决于对方的领地大小 attackCondition1	判断攻击的一个必要条件。	判断我方攻击距离和敌方撤退距离，敌方攻击距离的关系。时间复杂度在于调用上面三个函数	O(1) O(n2).

主要取决于领地大小（纸带是稀疏的）findAttackWay	寻找一个攻击路径	采用应当全面搜索的策略，用来搜索我方纸卷到攻击目标点一个的可能的最短路径。并且加入到攻击路径的列表中。搜索可能立即找到，也可能搜索所有可能的结果遍历。	取决于我方纸卷和攻击点之间的相对位置关系。如果横坐标差和纵坐标差接近，将更大一些。理论上对于曼哈顿距离为 n 的两个点，最糟糕的情况下（横坐标差和纵坐标差相等），时间复杂度为 O(2n), 但是路径几乎都符合要求（除非自己的纸带挡住了去路），我们可以认为加入路径列表这个时间复杂度是 O(d) 的（其中 d 是两点之间的曼哈顿距离）
attackCondition2	检索攻击路径是否安全	遍历攻击路径的点，计算敌方攻击到我方攻击路径的点的最小距离。判断该距离和我方攻击距离的大小关系。之前的我方攻击距离是计算储存过的。	O(1) O(n); 和攻击路径长短有关。但是由于攻击路径较长的情况极少出现，一般接近于 O(1)
chooseTurning	选择攻击路径的下一个点，根据现在位置和方向计算下一个返回值。	利用 if 结构分类讨论	O(1)

2 程序代码说明

2.1 数据结构说明

我们组采用的主要数据结构和算法有列表、字典、条件控制语句、循环语句、递归算法等。

2.2 函数说明

mybond 列表，包括我方所有的纸带的点。

enemybond 列表，包括敌方所有纸带的点。

myfields 列表，包括我方领地所有点。

enemyfields 列表，包括敌方领地所有点。

储存形式：storage[' mybond']=[[11,12],[12,12],[12,13]]

moveStep int 类型，记录攻击步数

attackState 布尔值，判断能否攻击

returnList 判断可以攻击后，获得返回值的列表

play 函数里面的函数：

Manha tta nDistance(position1,positi on2) 计算曼哈顿距离

输入：坐标 1，坐标 2（列表）

输出：int，曼哈顿距离

myid(stat) 获得我方 id

输入：stat

输出：int，我方 id

enemyid(stat) 获得敌方 id

输入：stat

输出：int，敌方 id

myposition(sta t) 获得我方坐标

输入：stat

输出：列表，我方坐标

enemyposition(stat) 获得敌方坐标

输入：stat

输出：列表，敌方坐标

distanceOfTwoScrolls(stat) 计算敌我双方纸卷的曼哈顿距离

输入：stat

输出：int，敌我双方纸卷的曼哈顿距离

renew FieldList(stat) 更新领地矩阵信息：

输入：stat

输出：无

功能：更新领地信息，储存在 storage[' myfields'] 和 storage['enemyfields'] 中

实现方式：如果判断任意一方纸卷位于自己的领地（双方领地的属性可能发生变化），开始更新：清空领地列表，遍历领地信息，将我方领地的点储存在我方的领地列表中，敌方领地的点储存在敌方领地列表中。

每回合调用

renewBondList(stat) 更新领地矩阵信息：

输入：stat

输出：无

功能：更新纸带信息，储存在 storage[' myfields'] 和 storage['enemyfields'] 中

实现方式：判断我方纸卷是否位于自己的纸带，如果是，添加点。如果不是（那么就是围地成功），清空。

每回合调用。

myAttackDistance(stat,storage): 我方攻击距离

输入：stat,storage

输出：如果对方有纸带，输出最小攻击距离。否则输出 False

功能：如果对方有纸带，输出最小攻击距离，并且将攻击位点（坐标）储存在 storage[' myAttackPosition']，最短的攻击距离（int）储存在 storage[' minAttackDistance']

实现方式：遍历敌方纸带列表，计算曼哈顿距离

enemyAttackDistance(stat,storage): 敌方攻击距离

（注意，和上面不完全相同）

输入：stat,storage

输出：如果我方有纸带，输出敌方最小攻击距离。否则输出 10000

功能：同输出，不储存

实现方式：遍历我方纸带列表，计算曼哈顿距离

enemyWithdrawDistance(stat,storage): 敌方撤退距离

输入：stat,storage

输出：如果敌方有领地，敌方撤退最小距离。否则输出 10000

功能：同输出，不储存

实现方式：遍历敌方领地列表，计算曼哈顿距离

attackCondition1(sta t,storage):

判断是否满足攻击条件 1：

我方攻击距离 <= 敌方撤退距离

敌方攻击距离 >= 我方攻击距离

双方纸卷距离 > 我方攻击距离（避免最近攻击距离是敌方纸卷的情况。如果对方在我方领地，这删除这个条件）

我方攻击距离 <= 我方剩余回合数

返回 True/False

findAttackWay(stat,storage):

寻找一条攻击路线，返回能否找到路径的布尔值。

攻击路线点的坐标储存在 storage['attackWayPosition']

每次沿着攻击路线行走时，坐标差（比如目标坐标 [74,75] 与当前坐标 [73,75] 的坐标差为 [1,0]）储存在 storage['attackWay']

寻找攻击路线要满足路径不撞到我方纸带，并且与当前方向不是反向的（不能回撤）

attackCondition2(stat,storage)

判断敌方在我方攻击路径能否进行攻击，计算最小攻击距离。判断能攻击的必要条件2：我方攻击敌方最短距离 < 敌方攻击我方路径最短距离

如果可以攻击，那么将攻击状态 storage['attackState'] 切换为 True

返回布尔值

chooseTurning(stat,storage)

在可以攻击的情况下，根据当前方向和攻击路径选择转弯方向。

返回这个方向值

2.3 程序限制

在比赛中我方有撞墙的可能。

在对方领地内撞击对方: 在比赛中我方会在对方领地内撞击对方而死。

超时: 在测试程序时我方有超时的情况。

3 实验结果

3.1 测试数据

实验环境说明：

硬件配置：（3.1Ghz/ 8GB）

操作系统：（Windows /10）

Python 版本：（Python 3.5）

分别用和技术组提供的 AI 对战、与自己作战、与课程中另一个组的 AI 作战的方式检验我方算法能否运行以及能否体现出良好的圈地/攻击性能。结果显示，我方 AI 可以流畅地与其他 AI 作战，胜率近 60%。我们也对失败原因进行了分类统计，如下所示

有 40% 的概率我方因撞墙而死掉

有 30% 的概率我方因在对方领地内被撞击而死掉

有 20% 的概率我方因超时而死掉

3.2 结果分析

小组赛成绩表:

	Charlie	Lima	Uniform	Victor	Zulu
Oscar	0	+	-	+	-

可见一些具体的技术数据信息。综合所有比赛数据,，我方平均圈地面积步数约为 3000 格。我方参与的比赛平均平均回合数约为 650。可见，从单位步数上的圈地面积来看，我方在所有队伍中都位居前列，说明我方圈地算法效率较高；但遗憾的是比赛难以延续中后期，通常都在很早就结束了，说明我方的算法有漏洞，会被对方攻击或撞墙而死，从自我保护上看，我方在各组中处于中下等水平。

我方一共胜利 38 局、失败 39 局。其中失败的最大原因是撞墙而死（20 次）和被对手撞击而死（17 次），从图 3 我们也可以看到常见的撞墙场景都出现在“我方纸带进入敌方领地，且三面均被敌领地包围”的情况下，说明我们对此场景的重视程度不够，过于自信的认为这种场景出现的概率低，没有做好相应条件的判断与处理，另外一些死亡因素也和设计算法时未能充分考虑所有情况有关。胜利的原因比较多样，最常见的是撞击对手纸带（19 次）、在我方领地内撞击对手（5 次）等，只有 2 次是回合数耗尽以面积取胜。由此可见，圈地面积在这 77 局战斗中意义不大，进行完备的防守和有力的攻击设计才是胜利的关键。我方算法的优点主要在于攻击性强，圈地效率高，缺点主要在于撞墙率太高，说明算法有缺陷。不过我方未出现撞击自己纸带自杀的情况。

4 实习过程总结

4.1 分工与合作

小组交流的方式有两种：借助北大空间教室开会，会议共三次（5.27、6.1、6.8），每次时长 2 小时。；微信群交流。

会议记录：

2018.6.1

此次会议为三个阶段的策略设计讨论了对此问题的战略安排、各阶段划分、各阶

段进入条件等内容。。

关于攻击的方式侧撞：

开始距离为奇数：先手的人有优势；

开始距离为偶数：后手的人有优势。

攻击的安全条件：我方回去的优势小于他方攻击的优势

如果有奇数优势，可以侧撞。

奇偶优势永远不变。理论上没有优势的人不能直接攻击。

劣势方要考虑反杀问题

战略：我们将整个游戏过程分为三个阶段：

第一阶段圈地划出基本盘：

先算开始时敌我距离 N1，按照 N1/4 走一个 L ，走完后再计算敌我距离 N2，后视情况算 N2/3 转弯新的 N2 前进，转弯数次画出方格后画方格，直到画回到终点，第一阶段结束。

在转折的过程中要计算你的我方最近的边和他的纸卷的距离。

时刻准备好攻击。

时刻注意和墙的距离。

第二阶段向对方那边圈地：

向对方的方向走（上北或南下）。

每一个转折都要判断敌我双方对方和你最近纸带最靠近处之间的曼哈顿距离，然后以原有领地的边界为方形的一边继续试图画方形圈地，当然这样最后可能因为曼哈顿距离的变化而出现“动态地狗苟”情况，最终形成的不是均一排列的方形，而是而成为折线。注意，折线不要超过自己原来的领地距离，必须有一个完整的边是自己原来的领地。我方领地与敌方领地接壤后，第二阶段结束。

时刻准备好攻击。

时刻注意和墙的距离。

第三阶段接壤：

吞吃对方的地，最后会形成稳态。

时刻准备好攻击。

时刻注意和墙的距离。

针对以上情况，按照是否接壤、敌我间距离远近情况，纸卷有四种策略：

接壤 & 近

不接壤近：小小地圈地

接壤近：对峙

接壤不近：先接壤的人有优势

判断时应考虑的参量：

1. 先后手

2. 奇偶

3. 第几阶段

4. 墙的距离

5. 对手离你最近的边的距离

代码分工：

第一阶段：贺雨岑、李沐航

第二阶段：吕博文

第三阶段：范启东、李承

攻击函数：丁力

拟 DDL 周日 6.3

2018.6.8

此次会议为每个部分代码的策略概述

1. 贺雨岑第一阶段

简化逻辑，使它比较简单。

走一个正方形，每走一步都要预想出四个点，然后走完。

2. 吕博文第二阶段

核心是计算当前场景对于我方是否安全，以保命为主，并在保命的基础上完成圈地任务。具体的说可以划分以下几种情况：

如果走出领地会被攻击，在领地里闲逛。

如果走出领地不会被攻击，先走 LINEFIRST。如果判断后对方攻击不到我，可以继续前进再走。如果能攻击到我，先判断能否通过改为走长方形的方式，不行就转回领地。

然后走 LINELAST，判断我的身后有没有领地，以及回去的路径。

3. 丁力：攻击函数

1. DISTANCE (X1, Y1, X2,Y2)

生成一个绝对值，输出一个曼哈顿距离。

D: 纸卷之间的曼哈顿距离

A1= 我方纸卷到对方纸带的最近距离，返回目标点

A2= 对方攻击我方的距离

D1 我回到自己领地的最小距离

D2 他回到领地的最小距离

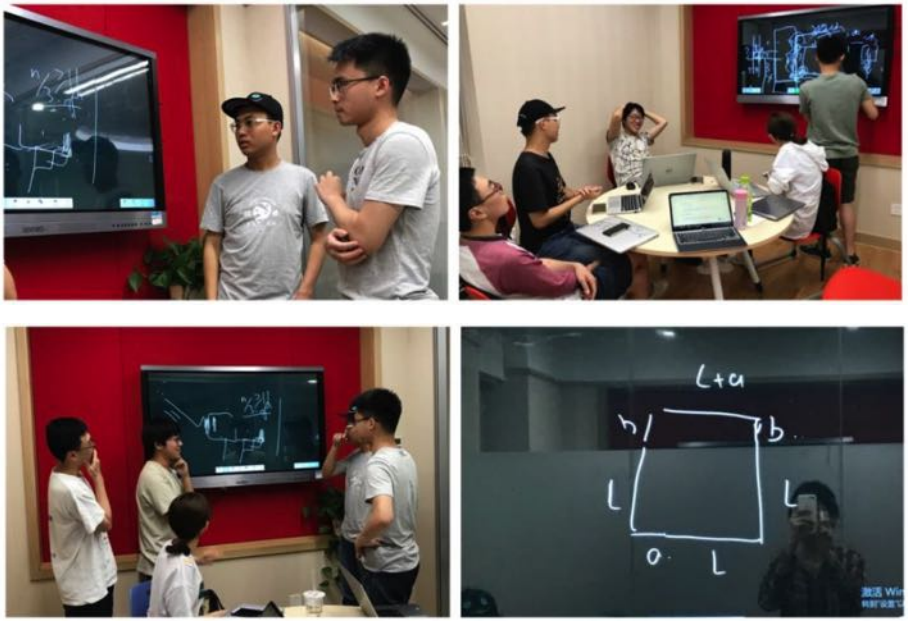
满足三个条件：

为防止在攻击时我方拉长的纸带被攻击，要先模拟出攻击路径，如果不会被攻击到，可以进行。

A1<D2 他不能先回去

如果 A2 ＜ A1 不能攻击

会议照片如下：



4.2 经验与教训

因为我们组成员基本都是生命科学学院的学生，参加这样的实习活动给我们都带来了全新的体验。虽然从过程上看，本实习与平时的科研活动类似，都有提出问题、抽象问题、控制条件、进行处理等过程，但本实习中用到的实际操作能力与技巧都是生命科学科研活动中难以用到的（最少我们几个做湿实验搬砖的是基本没有用过的）。通过此次实习，我们进一步加深了对计算问题、算法、程序设计的认识，锻炼了分析条件、写代码、检查代码的能力。

在这个过程中，笔者认为设定各层级的判断条件是最有意思的地方。人脑玩此游戏时，往往根据自己的感觉和情绪状态来决定下一步，但对于一个算法来说，它需要非常明确的行动规则，并且严格按照此规则运行。因此我们在设计算法时需要将各类可能出现的情形都尽可能地考虑到位，并对对应的情形应该做出怎样的反应也需要明确清楚地进行规定（部分用机器学习方法的小组可以忽略这一点）。从流程图上我们也能看出，在顶层战略上我们就设计了七个大的判断点，每一个具体的操作过程中也安排了不同类型的判断语句，最大程度上覆盖所有情形出现的可能。其中我们对是否启动攻击函数的判断是最令人得意的地方，因为我们的攻击有点像 xx 特攻一样，一旦启动就无法停止，也不能再进入或回到别的语句内容中的，出手即致命，要么敌方被杀，要么我方被杀。从我们的战果上看，出

击获胜的概率还是更高的，说明在判断是否进攻这一点上我们考虑的情形较为充分，不过出击后被杀的概率也很高，毕竟我们出手之后所有的路线与规划都与出手时刻的敌我状态有关，出击之后就只能看敌方是否给面子了。

我们的算法也有缺憾，笔者认为用设立规则的方式来思考这一问题其实是很难的，因为我们很难将所有可能的场景穷举出来并都写进代码中，所以出现了如撞墙等令人匪夷所思的场景。相反笔者个人更青睐于将纸带圈地看作是一个特殊规则的围棋游戏，只需要设定好最终的胜负规则，每一步下子时只能在当前下子处周围三个方向的格子上下子，通过机器学习，让机器自己去收集情景，并选出合适的应对方式。这样可以最大可能防止判断过程遗漏的情况，但能力所限，未能实施。

4.3 建议与设想

就组队上，高年级学生可能因为同年级小伙伴已经选过数算导致自己在本班里【人生地不熟】，组队更加困难，建议可以不按照年级进行分组。

本学期课程上技术组团队格外出色，基础设施代码运行非常良好，唯一建议就是如果可以提前准备好会更好，最好不要大家在写的时候再进行修订，避免因更新不到位错过信息。

竞赛可以用两节习题课 + 一节正课完成比赛，第一节习题课打完小组赛，淘汰赛前一部分可以在第二节习题课上解决，最后正课就只安排八强或者四强的比赛和后续的其他比赛。这样建议的原因有 1. 各组同学更多会关心自己本组或自己好友所在小组或公认强组的比赛状况，其他组可能会关注更少一些；2. 从最后两场决赛上可以看到，将比赛过程可视化对调动气氛、参与度、关注度来说都是极为重要的，因此在之后的课堂比赛中，建议用习题课教室分配不同区域播放各组比赛的可视化进程，这样大家的气氛都能给调动起来。

5 致谢

感谢北大空间为本组小组讨论提供场地。感谢 Papa 组提供陪练服务。

6 参考文献

[1] 微信小程序——围城大作战

[2]SESSDSA’18 课件

[3] 北京大学地空学院数据结构与算法 SESSDSA 内部教材

第十六章　N17_Papa 报告

鲍志成，程进前，刘书铭，杨晓宇，尹晨析 *，赵世俊

摘要：在本次大作业中，我们以纸带圈地游戏作为模型，思考了圈地、击杀、逃命、近战四方面的竞技内容。在圈地方面，我们考虑的是尽可能压迫对方领地，向对方领地处占领地盘，在击杀和逃跑方面，主要通过精准的距离判断做到只要攻击距离足够就进行击杀，在有危险是逃跑同时圈地。在近战方面，我们在实战中发现了 AI 近战的劣势并及时进行修正，保证在近战的过程中不会主动出领地“送命”。我们在热身赛及友谊赛中，通过失败不断 debug，最终由热身赛倒数第一逆袭成为正式赛第二名。

关键字：击杀和逃跑距离，路线规划，近战策略，深度优先算法

1 算法思想

1.1 总体思路

本算法主要分为以下几个优先级进行：（1）首先考虑是否可以在对方切断我方纸带之前切断对方的纸带；（2）其次考虑我方是否需要逃跑；（3）最后考虑围地。

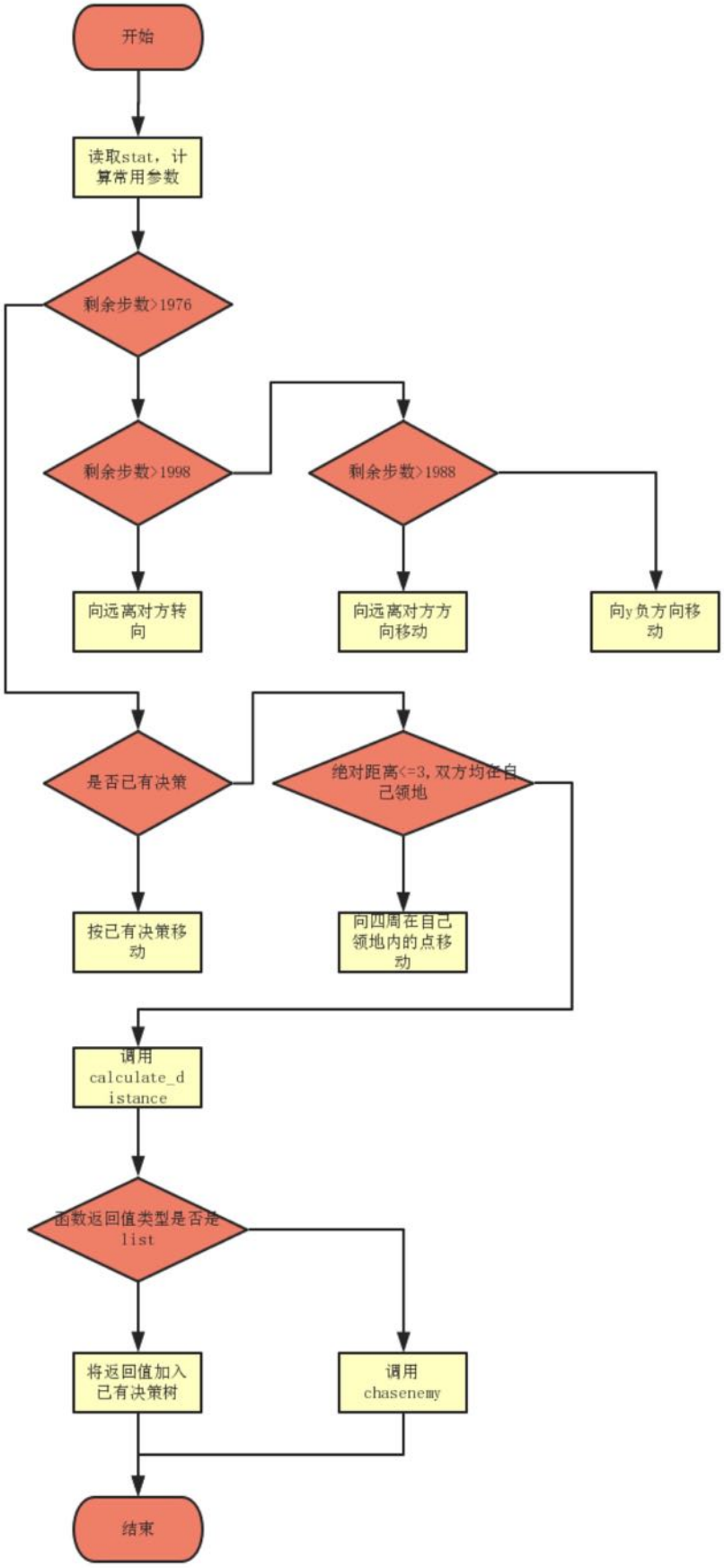
首先通过广度优先搜索计算出四个距离：L1 表示我方回到我方领地的最短距离；L2 表示我方切断对方纸带的最短距离；L3 表示对方回到对方领地的最短距离；L4 表示对方切断我方纸带的最短距离，并且计算出我方回到我方领地的最短距离对应的路线 R1 以及我方切断对方纸带的最短距离对应的路线 R2。如果 L2 <= L3 并且 L2 <= L4，则按照路线 R1 切断对方的纸带；否则，如果 L1 <= L4 - 3（此处实际上可以用一个更强的条件 L1 <= L4 - 2，实际中使用了 L1 <= L4 - 3，使得我方纸带可以与对方多保持一格安全距离）

并且我方的纸带长度小于等于 50（防止我方纸带拖太长，这样会增加风险），则考虑围地；否则，我方需要按照路线 R2 逃回我方领地。

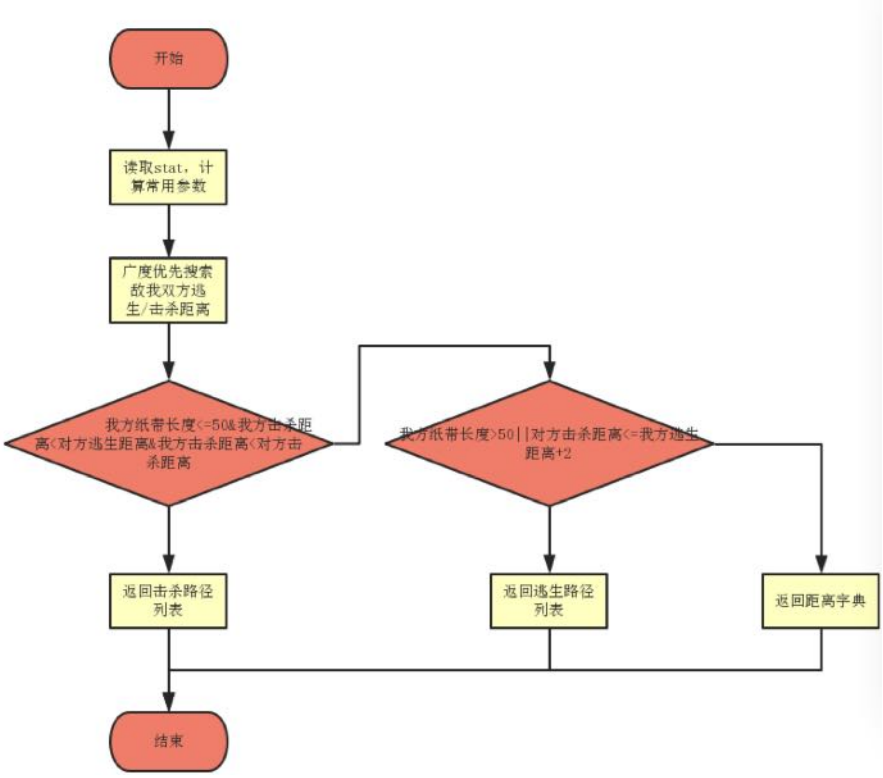
其次，关于围地策略，实际上我方的算法并没有任何主动围地的策略，只是在我方无法直接切断对方，并且我方不会被切断的前提下尽量靠近对方的纸卷，这样一方面我方的纸卷会尽量靠近对方纸卷，一旦对方出现破绽就可以快速切断对方的纸带；另一方面，虽然该方法的围地完全是基于一旦触发了我方纸卷逃跑的条件后就会快速逃回我方领地，该过程围到的领地面积往往并不大，一般都是长条形的领地，实际上围地效率较低，但是优势在于主动接近对方的纸卷可以有较大的概率进入对方的领地，从而将对方的领地直接转化为我方的领地，这样的收益效率实际上很高，而且可以主动威胁对方的纸卷，使对方的纸卷难以主动离开对方领地去围地。从最终比赛的结果来看，这个算法虽然简单，但是效果很好。

在整个算法构思的过程中，我们最初考虑的是通过蒙特卡洛方法对于每一步的决策进行随机搜索，并且通过打分的方式选出最佳决策；然而由于比赛的时间所限，该方法无法做到足够多的模拟回合数用于进行有效的决策。虽然这一尝试最终失败了，但是在写蒙特卡洛方法的时候，由于需要在底层进行模拟，所以需要在 play 函数内置一个函数用于计算每一回合某一方纸卷移动后对于场面的影响（比如我方纸卷切断了对方的纸卷，或者我方纸卷恰好回到领地，需要将围起来的领地转化为我方领地），在编写该函数的时候对于划分领地的操作用到了广度优先搜索，这也为后来通过广度优先搜索查找最短距离提供了思路。

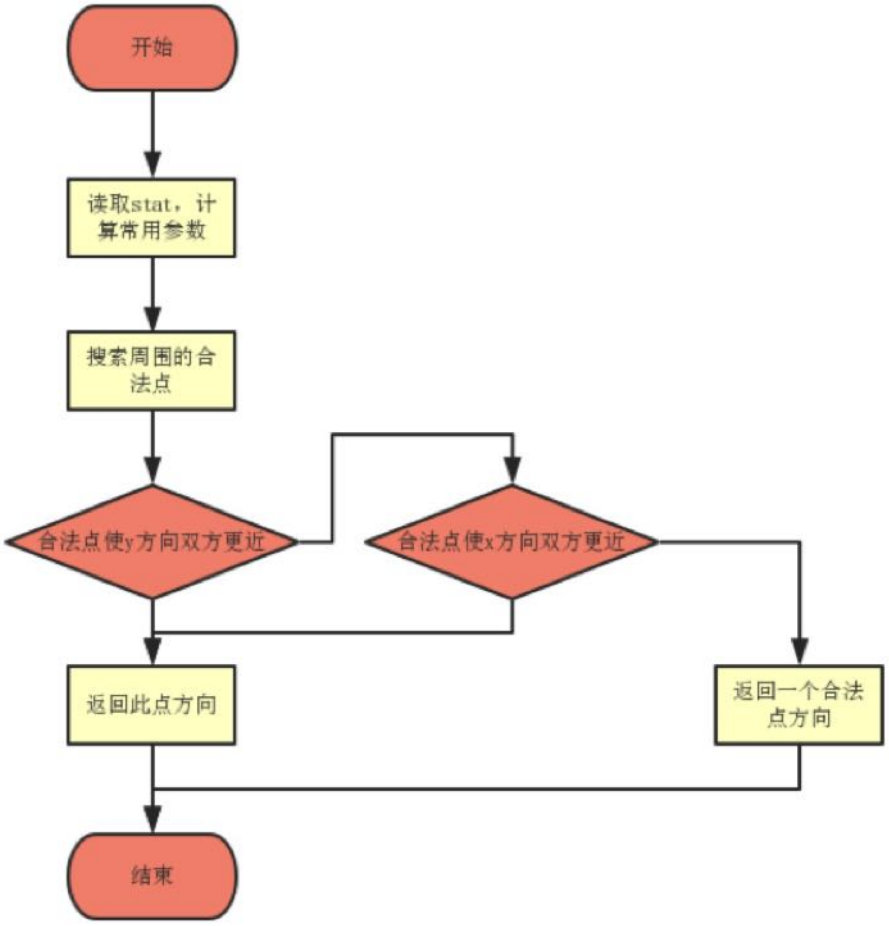
1.2 算法流程图


开始
读取stat，计算常用参数
剩余步数>1976
剩余步数>1998
剩余步数>1988
向远离对方转向
向远离对方方向移动
向y负方向移动
是否已有决策
绝对距离<=3，双方均在自己领地
按已有决策移动
向四周在自己领地内的点移动
调用calculate_distance
函数返回值类型是否是list
将返回值加入已有决策树
调用chasenemy
结束

总流程图



计算距离流程图



击杀流程图

1.3 算法运行时间复杂度分析

1.3.1 calculate_distance 函数

假设我方纸卷与对方纸卷的最短距离为 n，如果对方有纸带，那么必定在对方纸卷的后方有纸带，所以广搜的区域面积大约是一个对角线长度为 2n 的正方形，所以我方搜索对方纸带的时间复杂度为 O(n2)，同理，对方搜索我方纸带（假设我方有纸带）的时间复杂度也是 O(n2)。同样地，如果认为我方距离领地的最短距离也是 n，那么我方搜索到达我方领地的最短距离的时间复杂度就是 O(n2)，同理，搜索对方到达对方领地的最短距离的

时间复杂度也是 O(n2)。在实际中，由于加入了开关，当目标搜索到的情况下会关闭开关，并且结合 chasenemy 函数，我方纸卷会较为接近对方的纸卷，从而减少了搜索双方纸带到另一方纸卷最短距离的计算量。

1.3.2 chasenemy 函数

这个函数由于只需要查看当前纸卷周围的四个一步可到达的点，因此是常数次赋值语句，时间复杂度是 O(1) 的。

2 程序代码说明

2.1 数据结构说明

本程序主要用到了广度优先搜索，广度优先搜索需要引入列表，将需要搜索的点加入列表中，每次搜索完后从列表中弹出下一个需要搜索的点。

2.2 函数说明

2.2.1 定义 load 函数

从外而内地介绍 ai 中使用的函数。

最外层，load 函数。本函数非常简单，用于在每次比赛运行前为 storage 字典内增加一个新的 key，名字为‘track’，其功能是存储决策时得到的路径，用于在下一次调用 play 函数时不必再做一次多余决策。Play 函数将优先返回 track 中已有的方向。

2.2.2 定义 play 函数

同样最外层，play 函数。每一步都要被调用一次，用于返回下一步的行动方向。

首先记录下棋盘的大小，横纵坐标的最大值分别为 max_x_coordinate 和 max_y_coordinate，我方对应的序号为 me_id，对方对应的序号为 enemy_id.

为了决定下一步该走向何方，需要一套策略，我们在 play 的内部又定义了一组新的函数，calculate_distance，以及 chasenemy。这两个函数将在接下来的部分里主要介绍。

（1）定义 calculate_distance 函数

该函数计算我方回到我方领地的最短距离、我方切断对方纸带的最短距离、对方回到对方领地的最短距离和对方切断我方纸带的最短距离，并且根据这四个距离进行决策，如果可以直接切断对方则输出一个列表，列表包含切断对方的最短路径对应的一系列命令（比如 [‘left’,’right’,’None’]）；如果满足需要开始逃命的条件则输出逃跑的最短路径对应的一系列命令；如果以上两种条件都不满足则输出一个字典，字典包含上述四个距离，供之后的函数调用。

定义 calculate_distance 函数，该函数输入目前的棋盘状态 stat，最大横纵坐标 max_x_coordinate 和 max_y_coordinate 以及双方的序号 me_id 和 enemy_id。构建两个 $(\text{max\_x\_coordinate}+1)\times(\text{max\_y\_coordinate}+1)$ 的矩阵（二维数组）：me_distance_matrix 和 enemy_distance_matrix，二维数组的元素为字典，字典包含关键词’distance’,’fields_mark’和’bands_mark’，初始值均为 None。me_distance_matrix[i][j][‘distance’] 代表我方纸卷到达 (i,j) 点的最短距离，me_disntace_matrix[i][j][‘fields_mark’] 的取值为 me_id 或者 enemy_id 或者 None，代表 (i,j) 点是我方或者对方或者不属于任何一方的领地；me_distance_matrix[i][j][‘bands_mark’] 的取值为 me_id 或者 enemy_id 或者 None，代表 (i,j) 点有我方或者对方或者没有任何一方的纸带。同理，enemy_distance_matrix[i][j] 中的’distance’代表对方到达该点的最短距离,’fields_mark’和’bands_mark’分别代表该点的领地和纸带信息。

另外，用 me_bands_length 和 enemy_bands_length 分别记录我方和对方纸带的长度，用 me_fields_area 和 enemy_fields_area 分别记录我方和对方领地的面积。

遍历棋盘，将领地的位置和纸带的位置记录在 me_distance_matrix 和 enemy_distance_matrix 中，并且记录领地面积和纸带长度。

将双方纸卷目前所在的位置的距离设置为零，初始化字典 information_dict，该字典包含’me_to_me_fields_distance’（我方回到我方领地的最短距离),’me_to_enemy_bands_distnace’（我方切断对方纸带的最短距离),’enemy_to_enemy_fields_distance’（对方回到对方领地的最短距离)’enemy_to_me_bands_dsitance’（对方切断我方纸带的最短距离),’me_to_me_fields_min_distance_coordinate’（我方回到我方领地的最短距离对应的目的地的坐标）和’me_to_enemy_bands_min_distance_coordinate’（我方切断对方纸带的最短距离对应的目的地的坐标）这六个关键词，其中四个距离的初始值为很大的数（例如 10000),’me_to_me_fields_min_distance_coordinate’对应的初始值为 None,’me_to_enemy_bands_min_distance_coordinate’对应的初始值为 None。

建立四个作为开关的变量，代表是否需要继续在 me_distance_matrix 搜索我方领地、在 me_distance_matrix 搜索对方纸带、在 enemy_distance_matrix 搜索对方领地以及在 enemy_distance_matrix 搜索我方纸带，将四个开关的初始值分别设置为 True。如果我方纸带长度为零，则停止在 enemy_distance_matrix 中搜索我方的纸带；如果对方纸带长度为零，则停止在 me_distance_matrix 中搜索对方的纸带。

如果目前纸卷所在的位置就是所需要搜索的目标点，则将相应的作为开关的变量设置为 False，并且记录下相应的距离以及坐标。

定义函数 search_me_distance_matrix，输入当前纸卷的坐标 (i,j) 以及作为开关的变量。采用广度优先搜索，将 search_list 作为 BFS 队列，首先搜索对于纸卷当前所在位置的下一步能走的三个方向，避免撞上自己的纸带，如果该点就是所要找的目标点则记录该点的坐标和距离，并且关闭搜索开关，否则就在 me_distance_matrix 中该点的’distance’中记录到达该点的最短距离为 1，并且将该点加入 BFS 队列。

进一步进行广度优先搜索，当 BFS 队列中还有点并且还未找到所有需要找的点的时候，从 BFS 队列中弹出坐标 (x,y)，搜索 (x,y) 上下左右四个点（注意，虽然每一步只能走到三个点，但是此时可以保证这四个点中那个下一步不能走到的点在之前已经搜索过了，所以不会重复搜索），如果该点的最短距离没有被计算过，则该点的最短距离就是 (x,y) 的最短距离再加 1，并将该点加入 BFS 队列，如果该点就是所要搜索的点则记录下该点的距离和坐标，并且关闭相应的搜索开关。

同理，定义函数 search_enemy_distance_matrix，按照上述方法搜索我方纸带以及对方领地的距离。首先搜索对方纸带所在的位置下一步所能走到的三个点，并且将这三个点加入 BFS 队列。

同理，进一步进行广度优先搜索，记录下目标位置的最短距离。

运行定义好的函数 search_me_distance_matrix 和 search_enemy_distance_matrix，更新 information_dict，将更新后的距离和坐标存入相应的变量中

如果我方切断对方纸带的最短距离小于等于对方切断我方纸带的最短距离，并且我方切断对方纸带的距离小于等于对方回到领地的最短距离，此时的决策是主动出击切断对方的纸带。具体的做法是首先计算出从目的地到出发点的路径，计算的方法是根据目的地的距离逐渐倒推。

之后根据从目的地到出发点的路径坐标推出从出发点到目的地的一系列方向命令，并且将这些方向命令放在列表 route_order 中输出。

如果不能切断对方纸带，我方回到领地的最短距离小于对方切断我方纸带的最短距离减 3，并且我方纸带的长度小于等于 50 的时候，将之前算出的四个距离放在一个字典中输出。

在以上两个条件都不满足的情况下我方纸带有被切断的危险，需要逃命。按照之前搜索切断对方纸带的路径的方法搜索回到我方领地的最短距离对应的路径，进而根据路径计算出一系列方向命令，并且将这些方向命令放在列表 route_order 中输出。

小结：calculate_distance 函数通过广度优先的搜索方法搜索出了四个距离：我方回到我方领地的最短距离、我方切断对方纸带的最短距离、对方回到对方领地的最短距离、对方切断我方纸带的最短距离，并且根据这些距离进行决策：如果满足可以切断对方纸带的条件的时候直接计算出切断对方纸带的路径，将这条路径对应的一系列命令放在列表中输出；如果不能切断对方纸带，并且我方安全的情况下，输出四个距离；如果我方有危险，则计算出回到我方领地的最短路径对应的一系列命令，将这些命令凡在列表中输出。在具体调用 calculate_distance 函数的时候需要检查该函数的输出，如果输出的类型是列表，说明是一系列命令，则直接执行这一系列命令；否则，输出的类型是字典，则由 chasenemy 函数进行进一步决策。

（2）定义 chasenemy 函数

Chasenemy 的传入参数按照顺序分别为本方 id，本方 x 坐标，本方 y 坐标，对方 x 坐标，对方 y 坐标，本方与对方 y 方向的坐标差（对方 y 减去我方 y），本方与对方 x 方向坐标差（对方 x 减去我方 x）以及场上形势的 stat。首先，前 4 行是为了输出相对现在方向的前后左右所做的准备工作，与前述代码类似。列表 qq 用于存储最后的返回值。为了防止追击过程碰壁，提前算出 x，y 边界存储在 max_x_coordinate 和 max_y_coordinate。第一个 for 循环用于检查当前纸卷的四周所有一步可以到达的点，其中导致碰壁、反向、自杀的点被认为不合法，所有合法的点被存入 possible_step 中。第二个 for 循环用于检查 possible_step 中向对方追击的点。具体规则为，若此点在 y 方向逼近了对方则优先级最高，若没有 y 方向逼近 x 方向逼近优先级次之，若此二者都没有输出第一个合法移动。

总的来说，chasenemy 会使我方纸带先从 y 方向靠近对方，再从 x 方向靠近对方，并确保不会自杀。

接下来的是 play 函数的主体部分。上图是 play 函数中一些参数的计算，由上到下分别为，本方朝向，方向列表（东 0，南 1，西 2，北 3），相对方向字典，我方坐标，对方坐标，双方坐标差（对方 -我方）。

Play 函数的第一个判断语句的一个叉的一支。若我方剩余步数多于 1976 步时执行此支。第一部分为在前三步将方向调整为背离对方。第二部分为在 4-12 步向远离对方的 x 方向持续移动。第三部分为在 13-24 步向 y 轴负方向持续移动。

这一部分为 play 函数的第一叉的另一支。这一支为 25 步以后的策略决定部分。当 storage[‘track’] 中还有内容，说明之前做过决策，只需要维持原决策即可。若没有内容，则说明需要在这一步进行决策。首先，判断是否陷入了双方均在自己领地内的近身战，也就是双方均在自己领地，并且 x、y 方向绝对距离和少于等于 3（这个数字实际上就是所谓的‘安全距离’）。当已经陷入这种情况，我们就找到周围仍是自己领地的点，并走一步。若没有陷入这种近战，就调用 calculate_distance 函数，若其返回值是列表，这说明需要逃命或是可以绝杀，那么就将 storage[‘track’] 设为此列表。若其返回值不是列表说明没有可能绝杀也不需要逃命，这样就需要调用 chasenemy 继续追击。

2.3 程序限制

本程序主要有两种限制。

其一，在游戏过程中不能避免与对方头对头正面碰撞。这一问题表现得最明显的是在人机对战时。由于追击策略中，我们总是希望从 y 方向先靠近对方，然后再从 x 方向靠近对方，如此一来，当对方从 y 方向离开领地，我方恰好也追踪对方从 y 方向离开领地，若随后对方选择向我方这一侧逼近，就很有可能导致我方判断能击杀，从而与对方头对头行进，最后碰撞。实际上，正碰的胜负关系是依据双方领地大小，而我们根本没有将这一条件加入击杀判定。这里有两方面考虑，第一，是否能正面相碰很难在做出击杀判定的同时就预测到。第二就是运行时间限制，我们不能在击杀路径上加入更多判定条件了。最终，我们的改进方法是增加安全距离，在可能正碰时让纸带总是认为对方会更快地击杀自己，从而开始逃命，避免正碰。这样做导致的问题是，无论领地有多大，我方都不愿意与对方正碰。

其二，在近距离作战时，当对方与我方采用同样的保守策略时会出现无限循环。这一问题在经典战局中反复出现。由于本代码事实上根本没有圈地策略，而是模仿对方的圈地

策略，所以在开局初期经常会没有多少地，这时陷入无限循环就宣告了失败。事实上，当双方各自在自己领地内，同时又靠近 3 格以内时，谁先离开自己的领地谁就必输无疑。因此，从策略的角度来看，我们没法避免无限循环，除非我们能在此时从根本上改变策略。这显然是我们不愿意看到的，我们一直以来都是希望用最简单的策略和代码打出最好的战绩。所以这一点我们没有做修改。

3 实验结果

3.1 测试数据

实验环境说明：

硬件配置：（CPU/内存）QuadCore Intel Core i7-7700HQ, 3500 MHz (35 x 100)，8 GB DDR4-2400 DDR4 SDRAM (18-17-17-39 @ 1200 MHz) (17-17-17-39 @ 1200 MHz) (16-16-16-38 @ 1163 MHz) (15-15-15-35 @ 1090 MHz) (14-14-14-33 @ 1018 MHz) (13-13-13-31 @ 945 MHz) (12-12-12-28 @ 872 MHz) (11-11-11-26 @ 800 MHz) (10-10-10-24 @ 727 MHz)

操作系统：（名称/版本）Microsoft Windows 10 Home China

Python 版本：（版本号）3.6.2

3.2 结果分析

我们组的代码策略为在比赛开始时先远离对方，尽可能多地圈地。当双方接近到一定的距离时，我们的代码会变得极具攻击性，会开始追着对方的纸带，不断进行距离判断，若能够杀死对方，则发动进攻；若不能杀死对方，则在保证自身安全的情况下进行进攻。在保证自身安全这方面，我们考虑了许多种特殊情况，并在代码中对它们一一进行了单独编码。

在这种策略下，若是对方的纸带进攻性较弱，那就只能被我们追着赶，很可能被我们杀死或是自己撞墙、撞自己纸带自杀；若对方的进攻性能强，那么在双方正面交锋的情况下，由于我们的代码优先考虑自身的安全且考虑了绝大部分特殊情况，我们的取胜几率是比较大的。

假设对方的生存能力也很强，我们的代码在一开始采取的圈地策略就会取得优势。由于我们的代码采取不断进攻的策略，对方的代码在疲于应对的情况下也难以继续圈地，同时，我们的代码还会不断蚕食对方的领地，那么 4000 回合后，我们就会由于圈地面积的优势而取得胜利。

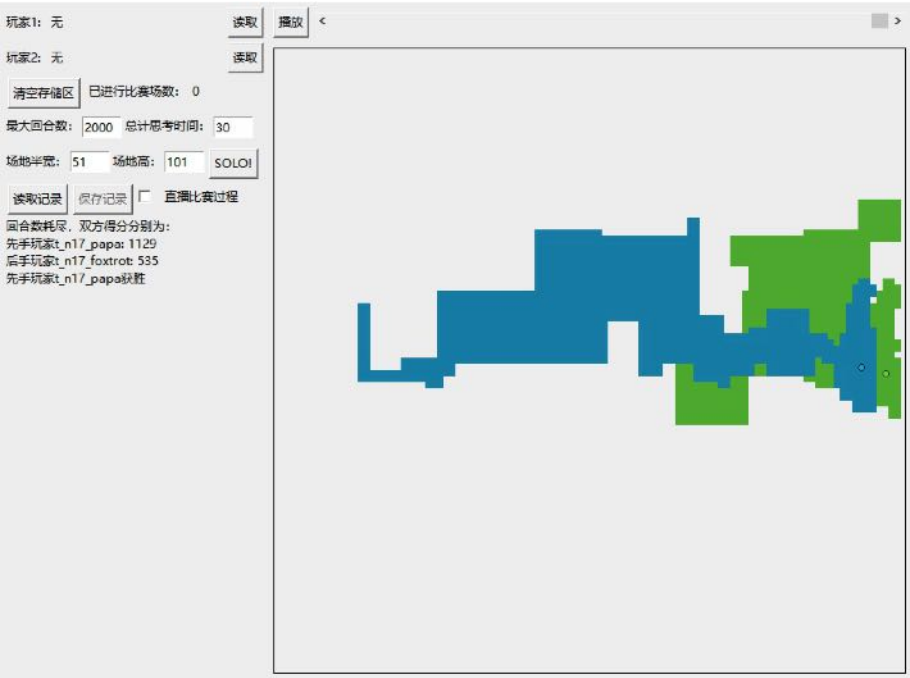
具体比赛情况

小组赛：

（1）对战 foxtrot：9-0

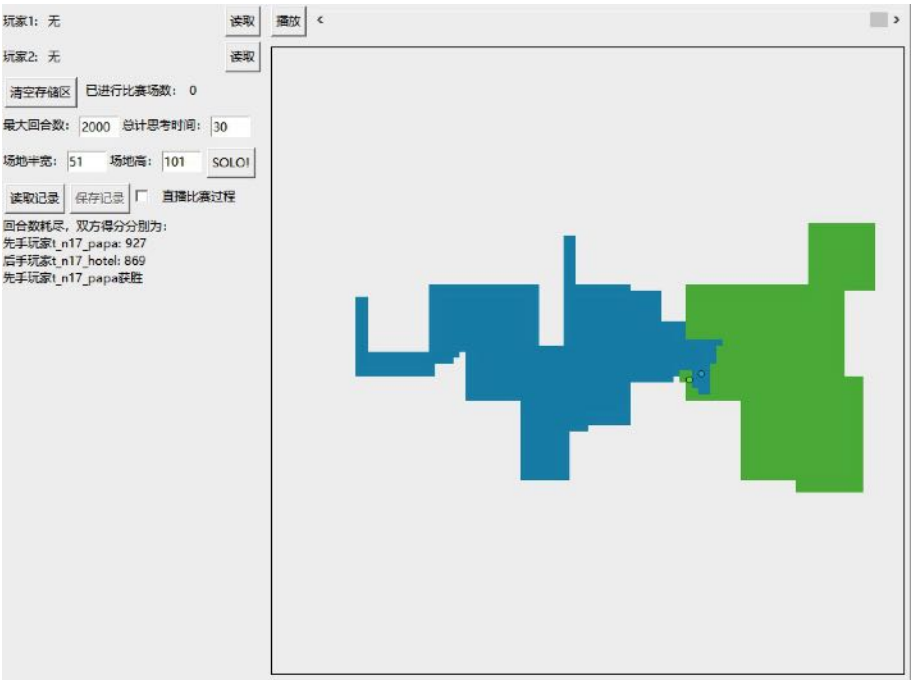
在 9 场比赛中，有 5 场走完了 4000 步，4 场提前结束。在这 4 场中，fox 队的代码在我方代码的不断追击下先后撞墙、撞自己纸带造成了自杀；而在 5 场 4000 回合比赛中，由于我们的代码的追击与蚕食，我们的圈地面积大于 fox 队，从而获得胜利。

值得一提的是，第（8）场比赛也体现了我们考虑到的一种特殊策略：在近身战斗中，当敌我双方都在自己的领地，且双方的 x、y 坐标差的绝对值之和小于 4，我们的代码就会远离对方一步，使敌我双方的距离保持在 4。



（2）对战 hotel

在对战 hotel 组时，我们的代码有三场在对方领地被对方撞死从而输掉了比赛。虽然无法找到原因，但这说明我们的代码还是存在一定的问题，还具有很大的改进空间。

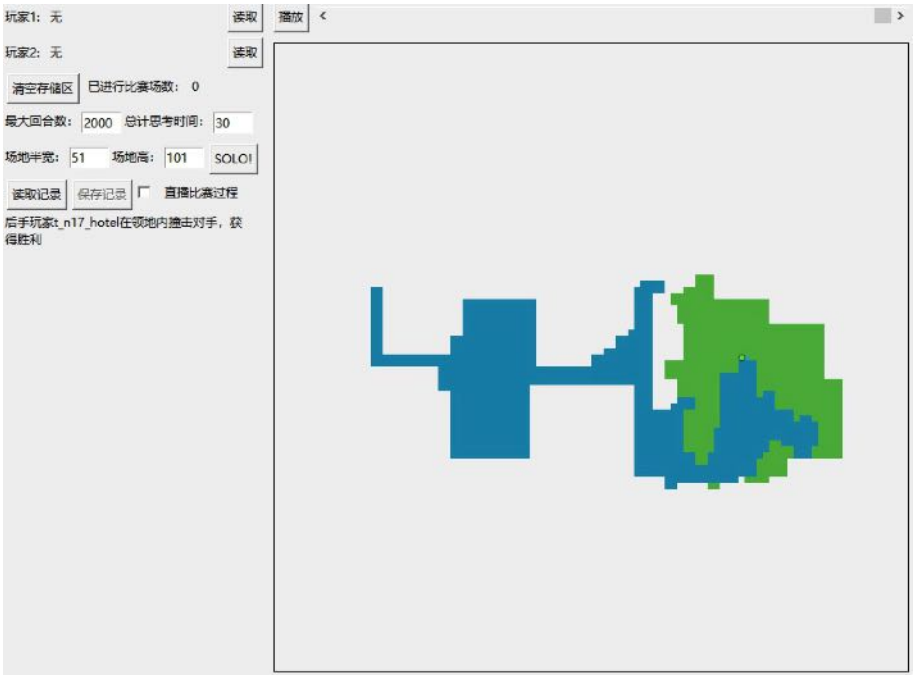


八强比赛：

对战 zulu：10-0

zulu 组的代码具有明显的圈地特征，它们的策略应该是在保证自身安全的情况下尽可能多地圈地，在 4000 回合后以圈地面积优势取胜。

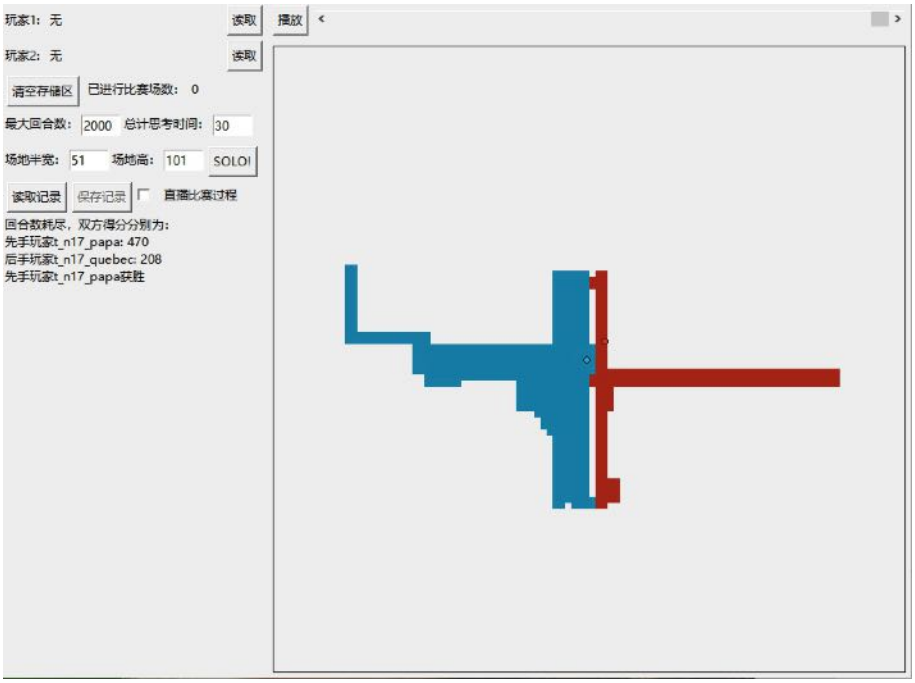
10 场比赛均在 2000 回合内结束，甚至许多对决没有超过 500 回合。在对决过程中，zulu 组代码的圈地面积往往大于我们的代码，然而在我们的进攻性策略下，对方代码往往会未到中局便被我方撞死。



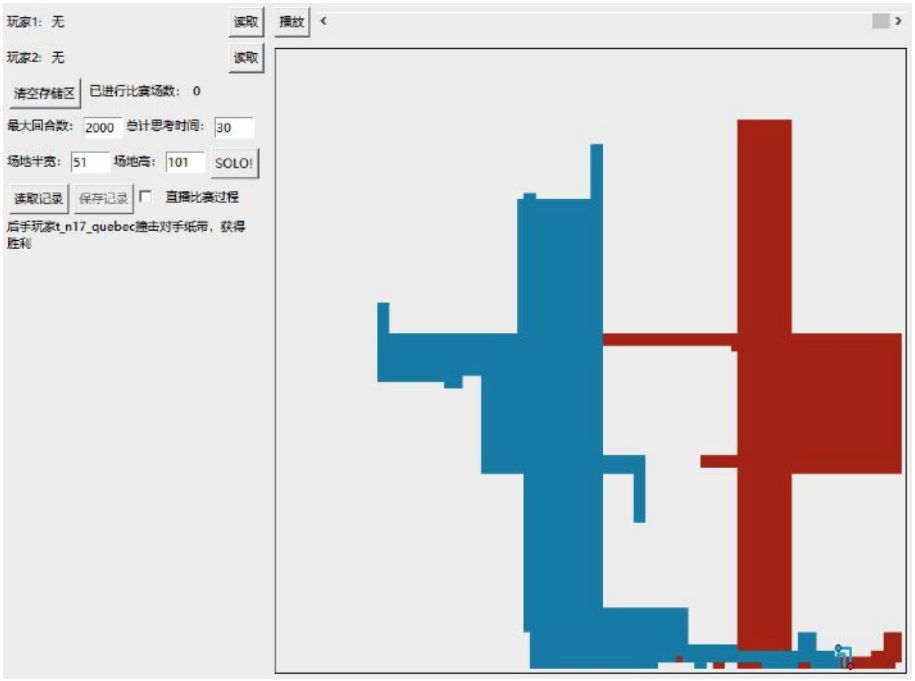
四强比赛：

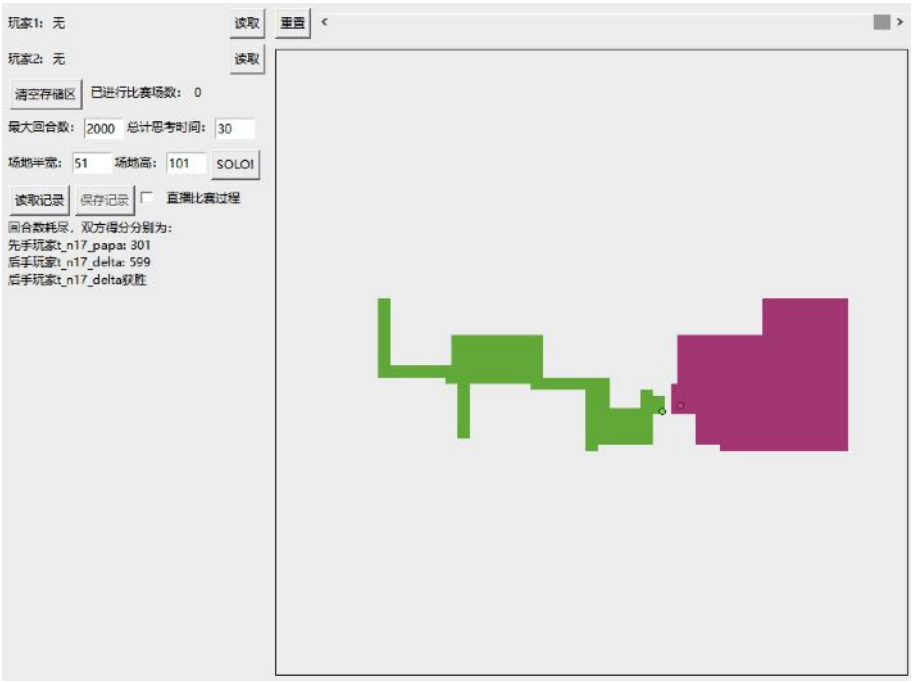
对战 quebec：10-4

在 14 次对决中，有 13 次以双方在近身情况下，在竖方向做死循环往返运动而结束，这说明 quebec 组的代码策略与我们组的非常相似，然而我们的优势在于更高的圈地效率。一方面我们的代码在比赛开局便选择圈出一块尽可能大的地；另一方面，我们在进攻的过程中通过蚕食对方领地来扩大自己的圈地面积。因此，我们在 10 次对决中最终以圈地面积优势获得胜利。



在 14 次对决中，有一场我方代码被对方撞死，这又一次说明我方代码存在一定的生存问题。





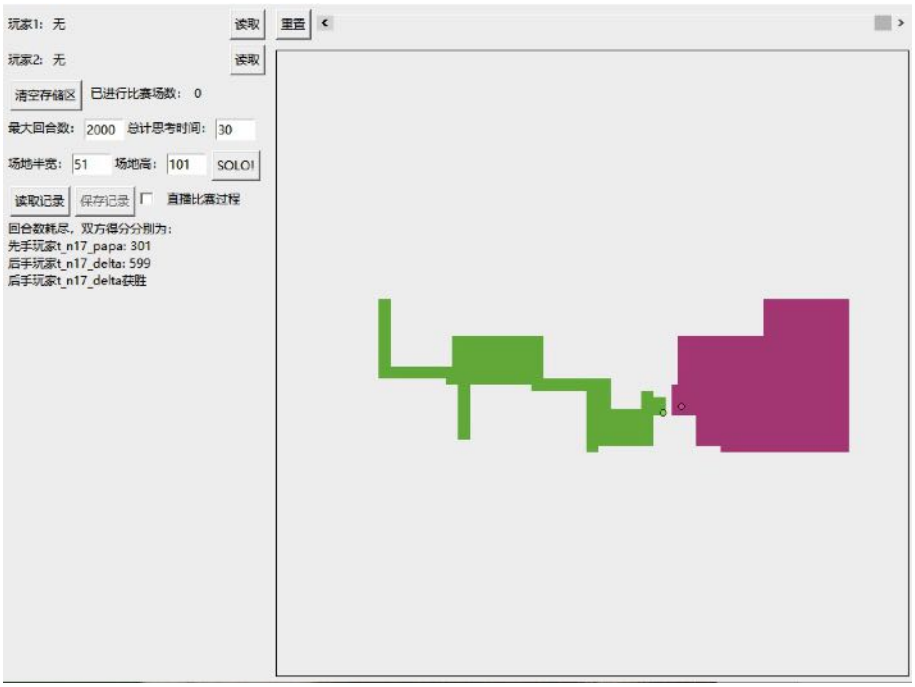
3.3 经典战局

决赛：

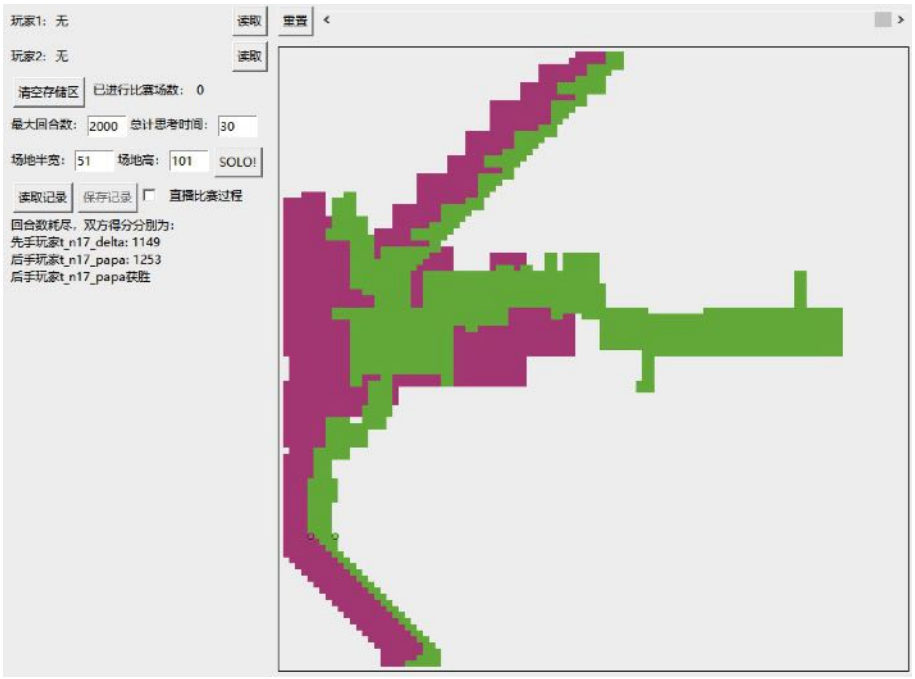
对战 delta：1-2

delta 的代码攻守兼备，是非常强劲的对手。

第一场比赛中，我方先手，我方代码在对方领地撞击对方，从而输掉了比赛，这也是之前一再提到的，我方的代码存在一定的生存问题，然而遗憾的是，我们直到现在也无法找出问题所在，只是可以确定这些生存问题一般出现在先手情况下。



第二场比赛，我方后手。由于对方的代码生存能力极强，我方的进攻难以杀死对方，但由于在进攻中不断蚕食对手领地，我方最终以圈地面积优势获得了胜利。



第三场比赛，我方先手。这里要说明一点，这个游戏由于双方的起始位置不变，双方每一次都只能移动一格，所以双方的 x、y 坐标绝对值之和奇偶性永远不变，故后手方在这场游戏中占据优势。

最终双方很快形成了死循环，而我方的初始圈地面积不及对方，输掉了比赛。

4 实习过程总结

4.1 分工与合作

4.1.1 分工

尹晨析：负责组织安排每次讨论，并且负责提供算法方面的建议；刘书铭、赵世俊：负责构建程序主体以及完善程序细节；鲍志成、程进前、杨晓宇：负责每次安排讨论的记录以及写报告。

4.1.2 照片

凯原楼讨论

图为综合科研楼讨论

凌晨 1 点的校园

正在给大家画图讲解深度优先算法的 dalao

综合科研楼最后一夜爆肝

会议时间	参加人数	参加人员	会议内容	分工安排	会议结论	用时
2018.5.27	6	尹晨析刘书铭鲍志成程进前赵世俊杨晓宇	1. 讨论初步算法设计并讨论每种方法的优劣及可行性；2. 进行分工安排	刘书铭与赵世俊尝试使用蒙特卡洛算法遍历两步内的所有可能性，并找出最优解，剩余四人从策略出发考虑另一个算法。	能用蒙卡在规定时间内算出下一步最理想，否则就用逻辑算法。	3 小时
2018.5.30	3	尹晨析刘书铭程进前	讨论了蒙特卡洛算法的实现可能性与具体方法，并考虑了开局时的最优策略	三人共同讨论算法的可能性与实现方法	开局时先使用提前设定好的最优策略，局中时再使用蒙特卡洛算法	3 小时
2018.6.5	6	尹晨析刘书铭鲍志成程进前赵世俊杨晓宇	完成算法的基本框架并进行测试 debug	两人负责设计路线，其他人负责用人机对战框架进行测试	算法在击杀方面非常优秀，但是时间和连续需要进一步改善	5 小时

2018.6.8	6	尹晨析刘书铭鲍志成程进前赵世俊杨晓宇	优化算法的用时	通过在找到路径后停止搜索节约计算时间，并进行人机测试	暂时没有其他问题，在圈地效率上稍显逊色	4 小时
2018.6.11	6	尹晨析刘书铭鲍志成程进前赵世俊杨晓宇	完善算法的近战策略，写报告的流程图，分析热身赛友谊赛的结果	两人负责写近战策略，四人负责进行分析、测试和写报告	在近战时时候，根据特定的结构维持在领地的状态	6 小时

4.2 经验与教训

4.2.1 经验

（1）如何统筹规划每个人的分工

在本次大作业的进行过程中，我们组在统筹规划每个人的分工方面做得较好，尤其是在写代码方面，通过将每个子程序进行封装以及约定好接口，使得不同的组员写的子程序能够较好地拼接，并且有效地组成一个整体。并且在分工的安排上充分发挥了每一位同学的特长，使得小组的工作效率最大化。

（2）通过与技术组同学交流获得算法灵感

本次大作业中，蒙特卡洛方法的“染色”（即判定领地的划分）以及最终采用的方案中的距离搜索都用到了广度优先搜索的方法。该方法的灵感主要来源于与技术组的同学交流，了解技术组如何实现游戏规则中的“染色”，进而设计出了利用广度优先搜索的方法解决了蒙特卡洛方法的“染色”，并且之后进一步将该方法应用到了距离搜索中。

4.2.2 教训

（1）算法方案的选择

在本次大作业开始初期，本组花了较多的时间写了通过蒙特卡洛方法模拟进行每一步决策的程序。然而该方法耗时较多，由于游戏的时间限制，该方法无法进行多步模拟，即搜索深度不够，并且无法模拟足够多的可能性。实际上，在选择一个算法之前应当考虑算法的可行性，可以在写出代码前估算算法的耗时以及时间复杂度，或者先写一个较为简单的程序并且计时，从而推测算法所需要的时间。事实上，由于蒙特卡洛方法是一种类似于深度优先的搜索方式，而每一步都有三种选择，所以随着模拟步数的增加，可能性是指数增加的，因此在规定的时间内模拟的回合数以及模拟的次数都比较有限。可以预料，如果有充足的时间，并且能够写出较好的评估函数，那么该方法的上限应该非常地高。

4.3 建议与设想

4.3.1 对于本次实习作业以及比赛的建议

希望在比赛之前可以确定好正式比赛的服务器的性能，因为如果只确定比赛时间而不确定服务器性能，在编写代码的时候可能难以决策是否需要用计算量较大的算法

4.3.2 寄语

往往在最开始，大家会希望靠机器学习的方法，因为一开始大家往往没有头绪，而机器学习的方法往往不需要有明确的思路，只要有足够的数据就行。但事实上，机器学习的方法效果未必会好，这主要是由于该方法的周期较长，而且需要大量的数据。实际上在推进大作业的过程中，最开始的阶段往往大家都没有明确的思路，此时可以先尝试写一些最简单的、能运行的算法，并且自己尝试玩一下游戏；如果有了一些想法也可以去尝试实现，虽然这些尝试的结果未必成功，但是可以在这些尝试的过程中获得新的灵感。最后还是希望大家在写大作业的时候可以想一想课上学的东西，例如这次我们的算法中搜索最短路径的方法用到了广度优先搜索，这就是课上讲的内容，在我们的代码中广度优先搜索起到了巨大的作用。

4.3.3 后续工作的设想

在有足够的时间以及数据的情况下机器学习很可能是最好的方法之一，可以将 1 对 1 的纸带圈地上传到网上，并且欢迎全世界的玩家来挑战，在这个过程中让 ai 自己学习，逐渐提高性能。

5 致谢

感谢陈斌老师的指导，感谢助教以及技术组的帮助与建议，感谢组内每一位同学的努力，感谢每一位为我们提供过建议的同学。特别感谢技术组的油库里同学，在我们遇到 Mac 人机操作键 bug 的时候进行了调试。同时也特别感谢 f17-Lima 组，进行了友谊赛，在友谊赛中我们发现了自己的不足并及时进行调整。

第十七章　N17_Quebec 报告

孟舜英 杨晓卿 张劲哲 欧一 * 暴嘉伟 许博东

摘要：我们利用启发式规则，通过自定义评估函数对场上两步移动以后的局势进行评估，由此得到一个价值分数。为了价值最大化，我们选取分数最高位置所对应的下一步移动作为我们下一步的策略。我们使用了树这样的逻辑结构与 python 自带的无序表、字典等数据结构，结合了函数值缓存的技术来实现我们的 AI。在综合考虑了效率与精准的平衡，我们的 AI 最终在比赛中取得了不错的效果，在大部分时候能对对手 AI 的策略做出正确的反应。

关键字：函数值缓存、A* 算法、无序表（Unordered List）

1 算法思想

1.1 总体思路

我们的总体思路是使用启发式算法，设定一个评估函数来比较不同移动路径的好坏，从中选出能够使评估函数最大的那条路径。具体而言，我们的算法会考虑移动两步之后的 9 种情况，从中选择最优的路径，返回第一步的指令，并将第二步的指令存储在 storage 里，之后直接读取并返回。

整个算法中最关键的部分就是我们设计的评估函数，它能够使 AI 具有逃跑和击杀的两种功能。我们的评估函数包括三部分：

1. 判断是否发生撞击。如果我方纸卷撞到边界或者自己的纸带，则返回 -10000；如果我方纸卷撞到对方纸带，则返回 10000。这个基础判定可以避免我方纸卷撞击自己的纸带，

并且在离对方纸带很近时会主动攻击。

2. 判断距离。计算对方纸卷到我方纸带的距离，我方纸卷到我方纸片的距离。如果对方纸卷到我方纸带的距离小于我方纸卷到我方纸片的距离加 p1（p1 是一个参数，会进行动态调整），则在返回值中减掉这个距离差。这样可以使我方 AI 在危险时向自己的领地逃跑。相应的，如果我方纸卷到对方纸带的距离小于对方纸卷到对方纸片的距离加 p2，则在返回值中加上这个距离差。这样可以使我方 AI 在对方离“家”太远时主动出击。

3. 判断方向。如果我方纸卷在纸片内，且前方离空白区域的距离小于长和宽的三分之一，或者是四个方向中离空白区域的距离最短的那个方向运动的话，则在返回值中加 1。这可以避免我们的 AI 在纸片中移动太长时间，而是会尽快前往空白区域中继续圈地。

由于我们计算距离的方式比较简化（仅使用了简单的曼哈顿距离），实际上会出现很多复杂的情况，所以我们在进入启发式算法进行评估之前，会先考虑一些特殊情况，对此做出反应。我们单独考虑的特殊情况如下：

1. 最开始先向对手方向移动，画出一块长条状纸片。这样在之后的移动中，可以更方便地回到自己的领地，并且可以圈出很大的一块纸片。

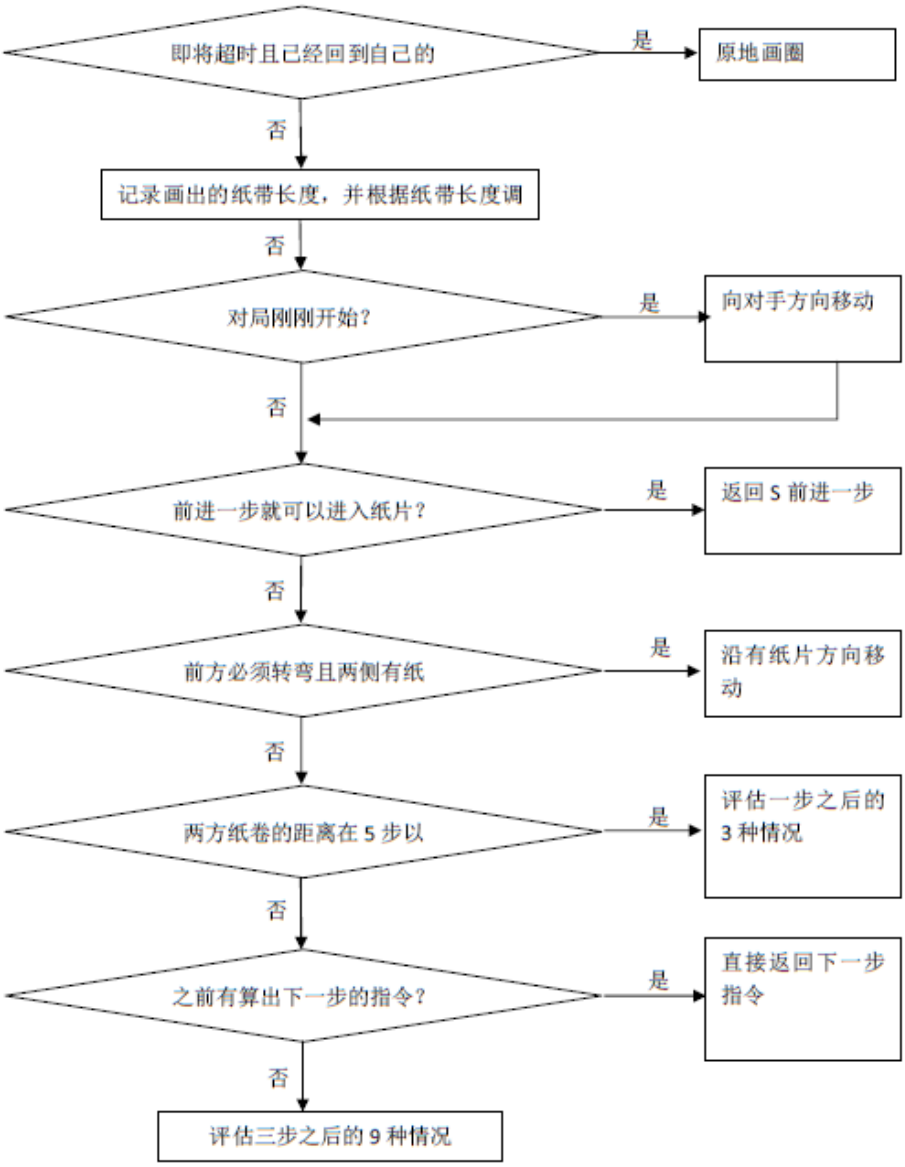
2. 如果只要向前一步就可以进入纸片，则一定进入纸片。这样在接近自己的纸片时可以实现圈地。

3. 如果前方必须转弯，则判断两侧是否有我方纸片，如果有的话一定回到纸片里。这样可以避免绕出“迷宫”状图案（后文会具体解释）。

4. 如果我方纸卷离对方纸卷的距离在 5 步以内，则只考察下一步的情况，并使用不同的评估函数。这样可以应对一些“短兵相接”时可能出现的特殊情况，更为精准地控制 AI 的行为。

在 project 开始之初，我们就确定了要使用启发式算法，让 AI 自行评估每步的情况并做出选择。最初，我们计划使用模拟许多步的 mini-max 算法，并且用 alpha-beta 剪枝算法来提高算法的速度。但是我们很快发现，4 层以上的算法就已经非常缓慢，难以达到时间要求，因而我们决定简化整个算法，只简单的考虑未来 2 步之后的 9 种情况。在整个项目的过程中，我们针对 AI 在对局中可能出现的奇怪行为（主要包括：画迷宫、无法及时逃回家、在纸片内长时间移动），不断完善了我们的评估函数，并设置了一些针对特殊情况的反应方式。

1.2 算法流程图



1.3 算法运行时间复杂度分析

我们的算法在每一步时都会重新运行一遍，因此消耗的时间与比赛的步数成正比。我们在计算距离时需要遍历场地中的每个点，因此消耗的时间与场地面积成正比。假设场地的长宽分别为 l 和 w，比赛的总步数为 n，则我们算法的时间复杂度是 O(lwn) 的。在实际

运行中，如果经常遇到不需要进行评估而直接返回指令的特殊情况，则实际花费的时间会有所减少。

我们的算法运行速度相对较慢，在正常的个人电脑上，在的场地中完成 4000 步的比赛大概需要 20 秒左右。我们采取了很多措施来提高我们算法的效率，比如存储第二步的指令并直接返回、对方纸卷到我方纸带和对方纸片的距离都只计算一次、先判断是否会撞到纸带再计算距离等等，具体的做法会在后面的章节中详细说明。

2 程序代码说明

2.1 数据结构说明

2.1.1. 主要数据结构

首先列表和字典的使用在整个程序中是无处不在的。例如在核心函数之一的评估函数中，使用二维列表用于存储场地和位置信息等，使用字典来存储决策指令等。

其次，我们使用启发式规则定义评估函数，使用了高度为 2 的树对接下来可能的步骤进行评估，由此得到评估分数，决定移动的位置。

2.1.2. 自定义类

在程序编写过程中，曾引入 Board 类，用于更新棋盘，提高搜索效率。除此以外没有使用其他的自定义类。

2.2 函数说明

2.2.1.dist(region,plr,x0,y0)：距离函数

该函数用于计算从地图上某一点 (x0, y0) 到指定玩家（自己/对方）的纸卷/领地的最小距离。

其中，参数 region 表示计算目标的类型（'fields' 或'bands'）；plr 表示计算的是到哪一方的距离（'me' 或'enemy'）；x0, y0 为外生参数。

我们用 ind 记录 plr 的 id，并初始化 res 为地图的长宽和。之后我们判断：

1. 若起点（x0, y0）恰好在 region 内，返回距离值为 0；

2. 若此时 plr 的纸卷头位置恰好在自己的领地里，则（x0, y0）到 plr 纸带的距离的返回值为 res 的初始值，即场地的长宽之和；

3. 若非以上两种情况，则遍历整个地图上的点，返回 res 为（x0, y0）到地图上符合距离计算要求的点的曼哈顿距离的最小值。

2.2.2.rightdirection(x0,y0,dire): 在领地内判断最优移动方向

当纸卷在领地内运动时，无法有效圈地，因此希望它能朝着离领地边界最近的路径方向运动，以尽快走出领地，开始“圈地”。该函数用于判断纸卷在领地内当前的运动方向是否为能使它最快走出领地的方向，如果是，返回 1，否则返回 0。

其中，参数 (x0,y0) 表示纸卷当前坐标位置，dire 代表纸卷当前运动方向。

我们利用 (x, y) 记录纸卷的坐标位置，将其初始化为 (x0,y0)。利用 while 循环，判断当纸卷在己方领地内时，就一直向东走 (x += 1)，直到走出领地为止。记录从初始位置到走出领地的步数 dir1 为 x-x0，则 dir1 为纸卷初始位置距离领地右边界的最短距离；如果纸卷移动出地图边界，则直接跳出循环。同理，可得到纸卷初始位置距离领地下边界、左边界、上边界的最短距离，分别记为 dir2,dir3,dir4。

若当前纸卷移动方向恰好为能使其最快走出领地的方向，或沿当前方向走出领地的距离小于场地边长的 1/3 时，返回 1，否则返回 0。

2.2.3.gohome():

判断在不碰撞到自己纸带的前提下，己方领地对于当前位置的相对左右位置。其中，参数 l2home 标记领地位置在当前位置的左边，r2home 标记领地位置位于当前位置右方。若向左转，在不碰自己纸带的前提下，发现了领地，标记 l2home 为 True。r2home 同理。若 l2home 和 r2home 均为 True，表示当前位置的左右两方都有纸片，函数返回 None；若只有一方有纸片，返回该方向的值（’L’或’R’）。

gohome() 函数可以避免纸卷一直在自己围成的纸带内部呈锯齿状移动，形成美丽“迷宫”，如下图所示。

2.2.4.deadend(x1, y1, dire):“保命”函数

遍历向前直走、左转、右转三种行走策略，若下一步移动不会使纸卷走出棋盘边界范

围，且能使纸卷回到己方领地，则返回 False; 否则返回 True。这个函数能保证下一步的行走策略是安全的。

2.2.5. eval_fn1(act1)：下一步精细评估函数

当己方纸卷到对方纸卷距离小于 5 时，需要精细地考虑每一步策略。eval_fn1(act1) 表示对于下一步行走策略的评估函数，之后我们会定义考虑之后两步的评估函数 eval_fn(act1, act2)，适用于两纸卷距离较大，而每一步对全局影响相对较小时。

参数值 act1 可选择向左一步、向右一步、直走一步三种方案。分别考虑以下情况：

若走一步后撞到棋盘边界，或撞到自己的纸带，返回 -10000

若走一步后处于对方领地内，且距离对方纸卷距离小于 2（或在对方领地内撞到对方纸卷），返回 -10000

若走一步后撞到对方纸带，返回 10000，此时 AI 会更加倾向于“主动出击”

若非以上三种确定游戏成功/失败的情况：

若己方纸卷逃回己方领地的距离（me2mefield）＋ p1（p1 根据纸带长度动态调整的参数）＞对方纸卷截杀我方纸带所需距离（enemy2meband），则在函数返回值中减掉这个距离差，使我方 AI 在对方有机会进攻时及时向自己的领地逃跑。

若对方纸卷返回对方领地的距离（enemy2enemyfield）＞我方纸卷到对方纸带的距离（me2enemyband），则在函数返回值中加上这个距离差，使我方 AI 在对方离对方领地较远时主动出击。

2.2.6.eval_fn(act1,act2): 下两步精细评估函数

逻辑类似 eval_fn1，考虑之后两步的策略。在己方纸卷距离对方纸卷较远时使用。

在第一步未直接返回 10000/-10000 时，即未能通过一步移动杀死对方/被杀死，此时考虑下一步策略：

若走一步后撞到棋盘边界，或撞到自己的纸带，返回 -10000

若走一步后处于对方领地内，且距离对方纸卷距离小于 2（或在对方领地内撞到对方纸卷），返回 -10000

若走一步后撞到对方纸带，返回 10000，此时 AI 会更加倾向于“主动出击”

若非以上三种确定游戏成功/失败的情况：

若纸卷处于自己领地内，调用 rightdirection 函数判断当前是否为最优方向移动，若为最优方向，返回 reightdir 为 1，否则返回 0。并在最终函数返回值中加上 rightdir。

若己方纸卷逃回己方领地的距离（me2mefield）＋ p1（p1 根据纸带长度动态调整的参数）＞对方纸卷截杀我方纸带所需距离（enemy2meband），则在函数返回值中减掉这个距离差，使我方 AI 在对方有机会进攻时及时向自己的领地逃跑。

若对方纸卷返回对方领地的距离（enemy2enemyfield）＋ p2（p2 为调整参数，P2 越大，越倾向于发起攻击）＞我方纸卷到对方纸带的距离（me2enemyband），则在函数返回值中加上这个距离差，使我方 AI 在对方离对方领地较远时主动出击。

2.3 程序限制

代码设计思路缜密，对纸带围城游戏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:

我们的策略通过人观测和计算得出，没有采用机器学习，故虽然面对对手的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。

同时，由于和很多较强的对手采取的策略类似，在测试和最后比赛时也都出现了双方在局部地区形成僵持，最终导致 4000 步耗尽的情况。

3 实验结果

3.1 测试数据

3.1.1. 实验环境说明：

硬件配置

处理器：Intel(R) Core(TM) i7-7500U CPU @ 2.70GHz / 8.00GB

操作系统：（名称/版本）Windows 10

Python 版本：（版本号）Python 3.5

3.1.2. 测试方法说明：

主要测试方法有：

1，编写一个和系统相似，但是规模更小的环境测试一些函数的性质；

我们重制了一个 20*20 的小型平台，并复制了里面的 memory，storage 变量，主要用于测试距离判断函数是否正确。然后对这个函数进行 debug，

2，使用技术组提供的简单方法 AI 进行测试；

我们主要使用随机游走的 AI 测试是否能够躲避潜在的危险。使用简单转圈 AI 测试我们的圈地速度。

3，将我们组旧版本的 AI 作为新版本陪练组；

我们组先后更新了四个版本的 AI，在更新时，我们使用原有的版本进行测试，在测试中发现问题。

4，参加热身赛进行实战测试；

5，和其他小组进行友谊赛。

我们先后和 N17 的 kilo 和 F17 的 Charlie 进行友谊赛。发现我们在前期很容易被 kilo 击杀，但是在比赛的后期，我们通过算法的改进，对 Kilo 和 Charlie 基本可以取胜。

3.1.3. 热身赛结果：

我们一共参加了 2 次热身赛，分别是第一个热身赛和第三次热身赛。

在第一次热身赛中，我们小组的两个版本的 AI 均被 Kilo 两个版本的 AI 和 india 击败，位列第四和第五。

在第三次热身赛中，我们改进了算法，但是遭遇了更为强劲的 delta，delta 的三个版本的 AI 包揽前三名，我们组 quebec_3 排列第四。但是相对原始的陪练代码表现并不理想。这些代码主要为了当作标准测试大家的平均水平。

AI 名称	得分	名次
Quebec_3	64	4
Quebec_2	41	9
Quebec_1	42	10

3.2 结果分析

在热身赛中，我们的策略分界比较明显，在前期，我们的的策略主要是圈地，在接近的时候我们才会开启自我保护和攻击状态。

在和比较简单的 AI 进行比赛时，我们通常很快能够找到机会，将对手击杀。

在和自我保护机制比较完善的对手进行对战的过程中，地域面积和计算时间，是决定能否取得胜利的重要因素。这些比赛通常会将我们的 4000 步耗尽，而在这个过程中，30 秒的时限经常耗尽。这时候，用于判断自己时候安全和是否有机会将对手击杀的 alpha-beta 搜索算法和其中的距离判断函数的调用花费了大量的时间。我们在改进中不再采用普适的距离判断函数，而基于我们总体的策略编写了计算更为简单的距离判断函数，这使得时间花销大大减少。

同样是由于时间的限制，我们的蒙特卡洛算法并没有能够实现。蒙特卡洛算法比我们的预期花费的时间要长很多。

在比赛进行的后期，我们的 AI 花费时间已经能够稳定在 30 秒之内了。在比赛前期的圈地过程中，花费的时间是极小的，通常不会超过 1 秒；但是在近身搏斗的过程中，情况变得很复杂，故而需要花更多的时间。

3.3 经典战局

我们组认真分析了在热身赛中和其他小组的对抗过程，我们小组的算法在于经常能够为圈一块很大的地方而伸出很多触角，并将他们联合在一起。这在我们最初是没有想到的，后来分析这和我们的参数设定有关。

让我们印象很深刻的战局有和 India 的，和 delta 的等等。虽然和 delta 的比赛中我们很少获得胜利，但是作为我们最强的对手，并且在一个小组，我们对 delta 的分析是最多的。Delta 的总体策略和我们很像，都是先圈地，之后在进行攻击。我们发现我们的问题主要在于很难预料到之前的领地被对手占据之后，回家难度大大增加。尤其是在局势非常复杂的时候，delta 的算法比我们的算法更加精致。

另外我们和一些比较强大的对手进行比赛时，双方的近身搏斗非常精彩。人和人的操作都很难如此精细。比如一开始很强势的 India 在后期由于精细化程度不够，很容易被我们的 quebec 直接击杀。

还有一点值得提及的是，在后期很多强势的 AI 的算法很相像，以至于经常出现比较镜像的策略形成某种意义上的“僵局”，而直接耗光 4000 步，这在某种也可以是“英雄所见略同”吧。

4 实习过程总结

4.1 分工与合作

本次分工与合作的交流形式为“群策群力，有的放矢”。接下来我们将按照时间顺序叙述本次活动的分工与合作：

首先，我们在 2018 年 5 月 29 日（周二）的数据结构与算法的课后，于理教 208 教室进行了第一次的集中讨论。参加成员为全体组员，在这次讨论中，我们明确了采取 3+3 的分工模式，即三个人为一个“组中组”进行代码撰写，并且在讨论中提及了两种思路——搜索算法与启发式规则，启发式规则即在讨论中进行了部分的猜想，例如画圈行走或攻击性走位。我们通过掷骰子的方式，分成了两组：欧一、暴嘉伟、许博东为一组，杨晓卿、孟舜英、张劲哲为一组，前者主要在启发式规则上做研究，而后者主要进行底层函数的封装和搜索算法的设计。

接着，我们在课后进行了网络资料的查询，包括孟舜英提出的 minimax 算法、欧一提出的 alpha-beta 算法和暴嘉伟提出的蒙特卡洛方法。

在 2018 年 5 月 31 日时，由孟舜英同学设计的初版搜索算法出炉了，并且在该日由张劲哲主导实现了语句标准化，使得组内代码复用的障碍得到了去除。并且，张劲哲提出对于安全距离的设计，并致力于实现安全距离和必杀距离的底层函数的实现。此外，欧一提出了如果采用搜索算法，为了达到全真模拟可以采用比赛环境的判定系统中的函数，但是耗时会是一个很大的问题。在如上讨论之后，孟舜英设计出了第二版搜索算法，具有了一定的攻击属性，然而，本组成员发现在耗时上，搜索算法的性能极差。

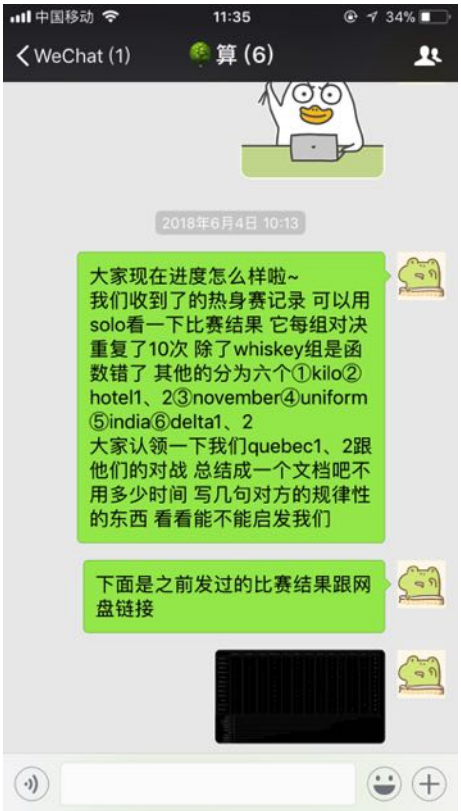
2018 年 6 月 1 日，欧一、杨晓卿和张劲哲在课后进行了简短的讨论，并且在孟舜英的

建议下更多的开始使用 storage 进行效率提升。在本日，张劲哲的底层函数封装有了阶段性的成果，并且欧一设计出了一个完全启发式的算法，能够以的复杂度与模板函数交锋 2000 回合，孟舜英也更新了新的版本，但是并未克服时间过长的问题。

2018 年 6 月 2 日，张劲哲的底层函数封装有了新的版本，此外杨晓卿也提出了对于遍历的优化的想法，并且提出了存路径的重要想法。

2018 年 6 月 3 日，杨晓卿加入了 refreshStorage() 函数和 Board 类，起到了更新棋盘等作用，提高了部分性能；另外，提出了关于 alpha-beta 搜索的改进想法。在该日的热身赛代码提交中，欧一组织提交了初版耗时长的孟舜英版和简单启发式规则的欧一版代码。暴嘉伟写了一个简单的用作 debug 的 testGen.py，进行实战测试找 bug。

2018 年 6 月 4 日，欧一线上组织了大家进行热身赛结果的观看，并且进行分任务观看，统计结果。分配任务截图如下：



2018 年 6 月 5 日，欧一、孟舜英、张劲哲、暴嘉伟集中进行了代码改进讨论活动，并留下了照片作为组会资料。

会中，集中决定了孟舜英负责优化模拟算法并由杨晓卿进行协助，张劲哲改进底层算法，欧一和其他人负责启发式规则的思考。在晚上，孟舜英成功推出了时间性能优异的模拟算法。

2018 年 6 月 6 日，在光华研讨室 419，欧一、张劲哲、孟舜英和杨晓卿集中进行了长达 5 个小时的组会讨论和代码书写，主要分工与前述相同，孟舜英和杨晓卿改进了 3 版代码，欧一提出许多启发式规则建议和函数提速的建议，张劲哲进行了底层函数的嵌入，实现了一定程度的改进。同时，四个人一起对于搜索算法的参数进行了经验结论上的调整。

同时，由于技术原因，我们组并没有参加本次的热身赛，因此暴嘉伟进行了重点对手的战术分析。

2018 年 6 月 9 日，为了解决上次的技术原因，我们组由暴嘉伟代替组长欧一完成代码提交的工作，并且提交了三个代码，最后取得了较为理想的结果。

2018 年 6 月 10 日，欧一再一次组织大家进行了比赛结果观看和总结。

2018 年 6 月 12 日凌晨，杨晓卿进行了代码的更新，进行了大幅度的速度提升。而其他人进行了比赛结果的观看后也有了一定的结论，同时张劲哲和孟舜英都私下与别的组进行了友谊赛，得到了许多有价值的收获，并且全组同学利用上述结论在比赛开始前形成了最后一版的代码。

综上所述，本组的分工如下：

孟舜英：主要负责撰写搜索算法，即最后的比赛函数，并且持续跟踪优化函数

杨晓卿：主要负责协助孟舜英进行搜索算法的性能优化和改进，创造性开发函数

张劲哲：前期负责底层函数的封装，后期负责算法性能分析和建议，全程技术指导

欧一：主要负责组织工作并参与讨论，并且在前期提供了简版启发式规则算法

暴嘉伟：主要负责 debug 函数的撰写和对战分析，提供许多改进思路

许博东：参与初期讨论

本组一共进行了 4 次组会，2 次小型碰面和 2 次有照片记录的大型组会，并且在 15 天的准备时间里在线上完成了 11 天的持续讨论，竞赛算法改进超过 20 次，并且拥有简单的备选算法，最后获得了 N17 第三名的好成绩，与组内的分工合作和组员的辛勤工作是密切相关的。

4.2 经验与教训

本次经验与教训中，我们诚然创造性的使用 alpha-beta 算法，以启发式规则和搜索算法的集合实现了一个较为优秀的结果，但在我们的开发过程中我们对于开发模式的选择走了很多的弯路。

我们最开始采取了类似于“瀑布模型”的开发流程。即对于这个大作业我们的理解是，可以将其分成两个部分：策略 + 底层接口。策略指的是关于我们具体采用宏观上的战术指导，比如是否采取镜像策略，是否更倾向于杀人等等。而底层接口是无论宏观策略如何都可能会用到的一些函数，比如计算回家的最短路径、计算纸片边界等等。那么我们类似于“瀑布模型”的开发模式具体则是先共同讨论确定底层接口有哪些需求，然后再由一部分同学编写这些接口，最后嵌入到已写好的策略中。

这样做的好处是可以省掉很多重复开发的工作，一些大家都需要的函数则不必要人人都来编写。这样看起来有诸多好处，但是在实际操作的时候我们却发现了一个很致命的问题即这样开发的工作量太大并且我们易忽略比赛规则对我们函数的需求的要求（时间限制等）。以我们组最开始编写的 safeDistance() 为例：

姓名：主要任务

```
1.def safeDist2point(who, t):
2.    '''''
3.返回玩家到任意一个点的安全距离（即自己移动过去不会跨过自己的纸带）
4.    :param who:给定的玩家 ‘’   me or 'enemy'
5.    :param t:给定的点 是一个,: tuple(x, y)
6.    :return返回距离如果没有安全距离，返回:,None
7.    '''
8.    #定义复合点类 ,下面算法中全部使用复合点类 [pos,f,g,h,parent] (是普通点类) parent
9.
10.    pos = (stat[who]['x'], stat[who]['y'])
11.    point = [pos, manDist(pos, t), 0, manDist(pos, t), None]
12.    unwalkable = getFB(who, 'bands')  #不能走的部分 注意这个不是复合点类）(
13.    if t in unwalkable: #如果终点不可抵达，返回 None
14.        return None
15.    #使用  A*算法来算越障碍距离
16.
17.    openlist = []  #用来存放探测点，起始点先加入
18.    openlist_normal = []  #为了之后计算方便把中普通点单独存放 openlist
19.    closelist = [point]  #用来存放考虑过的点
20.    closelist_normal = [point[0]]
21.
22.    #把第一个点先单独拿来处理
23.    check = point
24.    new = [(-1, 0), (0, -1), (0, 1), (1, 0)]
25.    dir = stat[who]['direction']
26.    if dir == 0:
27.        new.remove((0, -1))
28.    elif dir == 2:
29.        new.remove((0, 1))
30.    elif dir == 3:
31.        new.remove((1, 0))
32.    else:
33.        new.remove((-1, 0))
34.    for tu in new:
35.        i = tu[0]
36.        j = tu[1]
37.        x_new = check[0][0] + i
38.        y_new = check[0][1] + j
39.        if x_new < 0 or y_new < 0 or x_new > HEIGHT or y_new > WIDTH:  #点的合法性判断：不能
                                                出棋盘
40.            continue
41.        pos_new = (x_new, y_new)  #还不是复合点 pos_new
42.        if pos_new in unwalkable or pos_new in closelist_normal:  #去掉不能走的点
43.            continue
```

```
44.         if pos_new not in openlist_normal:
45.             g = manDist(pos_new, check[0]) + check[2]
46.             h = manDist(pos_new, t)
47.             openlist.append([pos_new, g + h, g, h, check[0]])
48.             openlist_normal.append(pos_new)
49.
50.     while t not in closelist_normal:  #算法的结束条件：终点加入 closelist
51.         #先找到中最小的点 openlistf
52.         minVal = min([p[1] for p in openlist])
53.         check = None  #选出来的最小点 f
54.         for pt in openlist:
55.             if pt[1] == minVal:
56.                 check = pt
57.                 openlist.remove(pt)
58.                 openlist_normal.remove(pt[0])
59.                 closelist.append(pt)
60.                 closelist_normal.append(pt[0])
61.                 break
62.         #更新中的值 openlistg
63.         for pt in openlist:
64.             if abs(pt[0][0] - check[0][0]) + abs(pt[0][1] - check[0][1]) != 1:  #只考虑和最
                                                新的点相邻的
65.                 continue
66.             dis = manDist(check[0], pt[0]) + check[2]
67.             if dis <= pt[2]:
68.                 pt[2] = dis
69.                 pt[1] = pt[2] + pt[3]
70.                 pt[4] = check[0]
71.
72.         #将其周围的点加入到 openlist
73.         #需要取出此时的方向，逆着该方向的不能走
74.         for tu in [(-1, 0), (0, -1), (0, 1), (1, 0)]:
75.             i = tu[0]
76.             j = tu[1]
77.             x_new = check[0][0] + i
78.             y_new = check[0][1] + j
79.             if x_new < 0 or y_new < 0 or x_new > HEIGHT or y_new > WIDTH: #点的合法性判断:
                                                不能出棋盘
80.                 continue
81.             pos_new = (x_new, y_new)  #还不是复合点 pos_new
82.             if pos_new in unwalkable or pos_new in closelist_normal:  #去掉不能走的点
83.                 continue
84.             if pos_new not in openlist_normal:
85.                 g = manDist(pos_new, check[0]) + check[2]
```

```
86.                 h = manDist(pos_new, t)
87.                 openlist.append([pos_new, g + h, g, h, check[0]])
88.                 openlist_normal.append(pos_new)
89.     #print('to:',closelist[-1][1])
90.     #print('man:',manDist(pos,(1,1)))
91.     return [closelist[-1][1], closelist[-1][4]]
```

在这一段代码中，采用了 A* 算法来进行有障碍的寻路计算安全的距离，意思是双方为了移动到某个点同时不能在移动的过程中踩到自己的纸带应走的最短距离。算法类似于用于解决词梯问题的算法，是一个无权图的 BFS，由此时间复杂度是 O(|V|+|E|)。这个算法是一个非常精确的距离，但是为此付出的效率特别高。我们在测试的时候发现，全程使用这个函数那么只能坚持几百回合左右，尤其当你和对手距离越远，这个函数耗时越多（因为实际意义上的图相对扩大了）。

但这个函数最大的问题是在于其实绝大部分用不上这个函数。这个函数我们最开始的初衷是为了避免 AI 走出弯弯绕绕的路径之后逃生途中踩到自己的纸带撞死，但这一点完全可以被预先设计好的宏观战略给避免掉：比如出去圈地可以走非常正规的正方形等，而两个 AI 短距离的时候则几乎不会出现圈圈绕绕的情况。所以这一部分有一大堆的工作都被浪费掉了。

后面我们讨论，觉得作为一个这样的大作业，相对来说工程量比较小，我们应该采取的是“敏捷开发”的策略。即快速整合出一个结果，并迅速测试然后再在已有的代码上进行增量式的开发与修改。意思是我们可以不必将底层接口和宏观策略拆开实现，而应该打包起来，宏观策略和底层一起实现。之前分开产生的问题也可以认为是需求定义不够清楚，因为我们分开实现时，在没有成型的策略以前，很难明确究竟哪些接口是需要的，会产生出一些看起来似乎很需要实则无用的需求（比如上面的安全距离函数）。而且这个比赛性质的 AI，能在实战中得到更多的反馈（我们从热身赛中获得了诸多启示），所以快速开发、多测试应该才是取胜的更好模式，而不是一上来就堆砌复杂的功能。用部分著名的敏捷宣言来说就是：“个体和交互胜过过程和工具，响应变更胜过遵循计划。”

除此之外，我们在进行小组合作的时候，还有一些有益的启示：比如同一个部分应该由两个同学结对完成，这样一方面提高效率避免一个人总是想不出来陷入困境而向其他组员求助时沟通成本又很高这样的问题，另一方面，两个人负责一块内容，在修改的时候就有更多的人能够维护，提升效率。

4.3 建议与设想

本次实习作业采取了一个规则简单但是实现起来难度较大的游戏，整体举办非常成功，但是也存在可以改进的方面。

技术组在举行热身赛的过程中付出了艰辛的努力，但是可能由于都是本科学生，活动组织经验不足，存在以下的可以改进的部分：

在第二次热身赛过程中，出现了 2 组的 MacBook 提交代码后文件夹为空的现象。然而，在技术组提供参赛代码截图让大家检查一下是否代码都有提交时，迅速开始了热身赛。这样，让第一次发现 MacBook 提交失败的 2 组的组员不具备重新提交的资格。在这个事件中，我们认为首先当技术组收到内容为空的压缩文件时，应当采取私戳或者群里告知的方式提醒提交组注意内容文件，尝试重新提交；并且不应该在给学生检查代码是否提交之后 5 分钟就开始热身赛，这样失去了提醒的意义。

简而言之，即需要一个更为系统化运作的技术组，并且能更好的与参赛学生进行沟通，这样带来的对群体的效益是最好的。我们很开心的看到，在这个事件结束后，技术组明显更好的执行了接下来的热身赛流程和比赛流程，在这里要给技术组的同学一个很大的感谢！

明年如果想要选修陈斌老师这门数据结构与算法，我们对学弟学妹们的期望就是：希望你们能全情投入这个充满乐趣又充实的课程，在结束后都有自己的收获，那就是老师辛勤组织这个在北大无出其右的丰富课堂的最大的回报了。

5 致谢

在此，我们全组成员感谢老师、助教与技术组提供一个优良的比赛环境和充分的热身赛机会，让我们本次实习作业能够圆满完成，并且收获了一个较为理想的好成绩。

接着，我们要感谢我们自己，是本组成员的努力和相互理解，让本次实习作业的小组合作取得了巨大的成功。

再者，我们要感谢 n17-bravo 和 f17-charlie 两组成员与我们组进行了友谊赛，通过友谊赛我们都收获了很多新的结论和想法。

接下来，我们要感谢贡茶提供组会讨论解困解乏工具（笑），感谢 Apple 公司对我们拍摄组会照片的大力支持（再笑）。

最后，感谢经历！我们都很感激能成为 sessdsa2018 的一份子！

6 参考文献

Alpha-Beta

https://ocw.mit.edu/courses/aeronautics-and-astronautics/16-410-principles-of-autonomy-and-
lecture-notes/MIT16_410F10_lec24.pdf

A* http://theory.stanford.edu/~amitp/GameProgramming/AStarComparison.html

https://www.cs.cmu.edu/~motionplanning/lecture/AppH-astar-dstar_howie.pdf

第十八章　N17_Sierra 报告

刘晟、史美琪、韩子程、张金霞、黄薇

摘要：本组的算法属于防守型算法，主要利用的数据结构有数组、列表和字典。基本的算法思路是在保障自己安全的前提下可以圈更多的地。每走下一步之前都会判断敌人到我即将走过的纸带的距离是否小于我回自己区域的距离，若检测到危险，会立刻将游戏模式切换为返回模式，若安全，则继续进行圈地的游戏模式，从而实现在保障自己安全的同时圈更多的地。

关键字：防守型；数组；字典；列表

1 算法思想

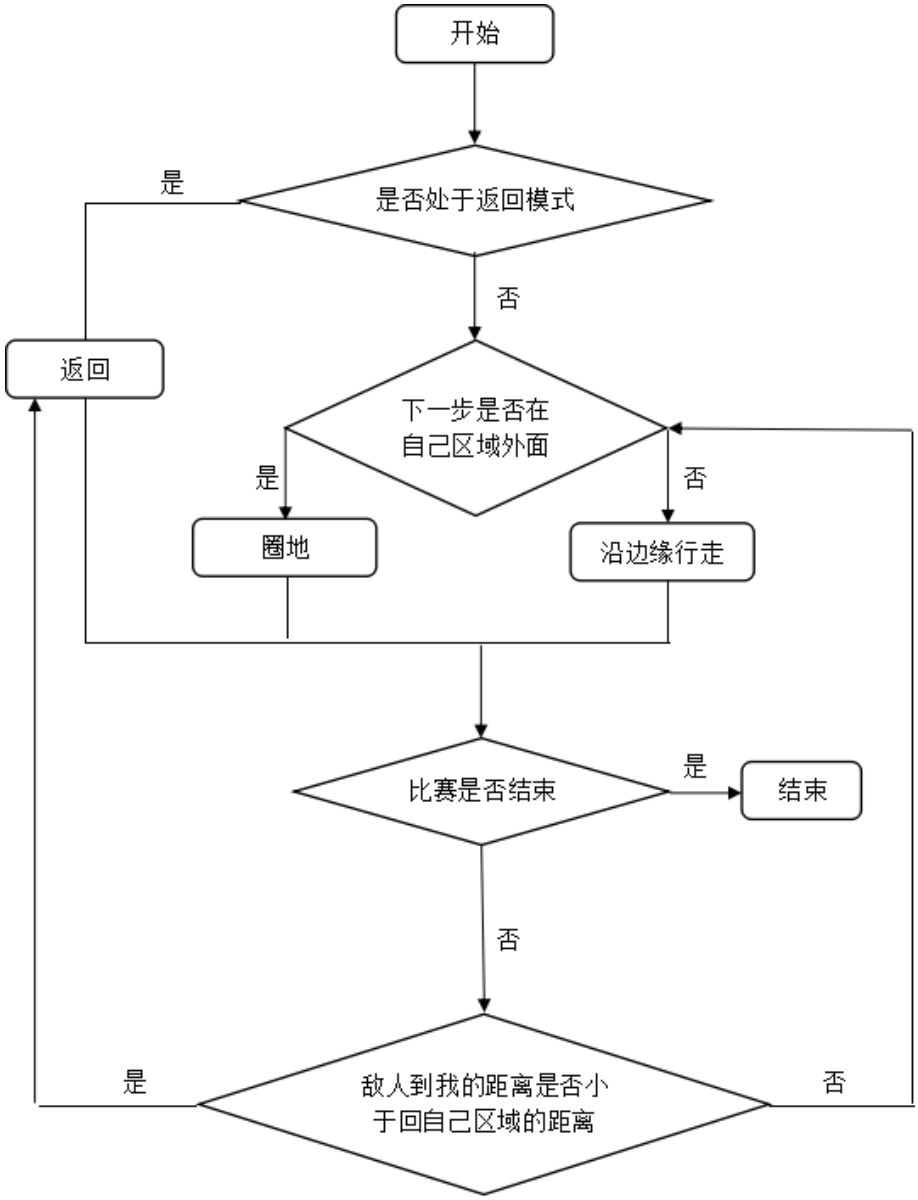
1.1 总体思路

算法的整体思路是在保证自己安全的前提下尽可能多圈地。每走下一步之前，算法都会进行一个是否会遇到危险的判断，以此来确保自己的安全。判断危险利用的是距离搜索函数，用于判断敌人到我纸带的距离和敌人到我下一步的纸带距离与我回自己区域的距离之间的关系。当判断出敌人到我纸带的距离或敌人到我下一步要走过的纸带的距离小于我回自己区域的距离时，即会出现危险，则启用“返回”模式，立即返回自己的区域，否则将会继续采用“圈地模式”或者“沿边缘行走模式”。

算法中会根据自己当前位置通过优先选择操作顺序函数决定自己的下一步位置，然后将下一步位置和目前位置对比，判断出游戏所处模式（“返回”“圈地”“沿边缘行走”），若游戏处于“返回”模式，不需要进行决策，返回即可。若处于“圈地”或“沿边缘行

走”，则需要通过最小距离函数，来判断敌人到我纸带的距离和敌人到我下一步要走过的纸带的距离与我回自己区域的距离之间的关系。如果判断结果是敌人到我纸带的距离或敌人到我下一步要走过的纸带的距离小于返回自己区域的距离时，则返回，否则就继续圈地。不管处于三种模式中的哪一种，走下一步之前都会进行危险判断，若不危险则模式切换为圈地，若危险，则游戏模式切换为返回。以此来实现在确保自己安全的前提下尽可能多的圈地。

1.2 算法流程图



1.3 算法运行时间复杂度分析

本组没有使用太多复杂的结构，大部分都是一些条件判断语句、循环语句以及距离计算语句等，但是在算法中，有一部分的条件判断需要把自己所在位置到对方区域里面每一个点的距离都需要计算出，然后再根据这个距离来判断要不要躲避，这个部分的复杂度较

高，而且决策较慢。这个部分的函数还应该再进行优化。

2 程序代码说明

2.1 数据结构说明

本组的算法中主要采用了数组、列表和字典的数据结构，数组作为算法的基础，用来定义行走的方向以及对下一步的位置进行判断，另外，算法中可以通过列表储存位置坐标等信息。由于字典可以很方便的储存和调用，所以我们使用字典存储前、左、右三个行走方向，从而可以判断下一步的移动方向。

2.2 函数说明

1. 判断障碍函数 nearPosition

首先要计算自己下一步要走的位置使用数组来表示位置，使用二维数组 [[1, 0], [0, 1], [-1, 0], [0, -1]] 来定义上下左右四个方向，使用数组相加来获得下一步可以走的位置。

1.1 判断是不是边界 isBorder

将自己下一步将要走的位置和边界位置比较，若是边界，返回 True。

1.2 判断是不是自己的纸带 isObstacle

将自己下一步将要走的位置和自己的纸带坐标对比，若下一步要走的位置是自己纸带的位置，返回 True。

2. 最小距离的计算 distance_xy

首先求出以 AB 为斜边的直角三角形的两条直角边的和，称为 AB 的折线距离

2.1 求一个点到一片区域的最短的折线距离 minimalDistance

利用 for 循环遍历区域中的每一个点，求出某点到该区域的最短的折线距离，同时获得最小距离的坐标

2.2 计算未来要经过路径和敌人之间的最小距离 imagineDistance

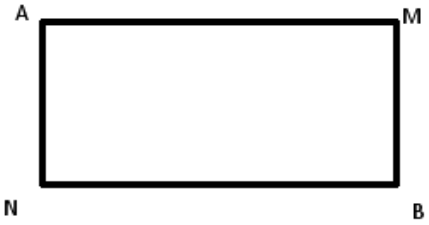
利用 for 循环遍历未来要经过的路径，求出到敌人位置的最小距离

3. 获得两点之间的行走路径 newRange

首先定义函数 newRange 将 A、B 两点间所有的 x 坐标和与坐标存储在数组中

3.1 获取两点之间的行走路径 pathAtoB

如图所示，AB 两点之间有两种行走路径 AMB 和 ANB，分别表示为 path_xy 和 path_yx，结合 newrange 函数分别求出这两条路径。如果一条路上存在障碍或者在自己区域就走另一条路径。



3.2 移动一步到达终点需要进行的操作

使用数组 [[1, 0], [0, 1], [-1, 0], [0, -1]] 表示前、右、后、左四个移动方向，使用 None、R、unable、L 表示

```
def nextPosition(begin_position, direction):
    directions = [[1, 0], [0, 1], [-1, 0], [0, -1]]
    return arrayAdd(begin_position, directions[direction])


def moveOnestep(begin_position, direction, end_position):
    directions = [[1, 0], [0, 1], [-1, 0], [0, -1]]
    if distance_xy(begin_position, end_position) != 1:
        return False

    target_direction = None
    for i in range(4):
        if arrayAdd(begin_position, directions[i]) == end_position:
            target_direction = i

    turn = None
    if target_direction < direction:
        turn = target_direction + 4 - direction
    else:
        turn = target_direction - direction

    out = {0:None, 1:'R', 2:'unable', 3:'L'}
    return out[turn], target_direction
```

4. 决策函数

使用字典存储前、左、右三个行走方向，用于判断下一步的移动方向

```
def selectDecision(decisions):
    out = {0:None, 1:'R', 2:'L'}
    for i in range(len(decisions)):
        if decisions[i]:
            return out[i]
    for i in range(len(decisions)):
        if decisions[i] == None:
            return out[i]

def newselectDecision(decisions):
    out = {0:None, 1:'R', 2:'L'}
    for i in range(1,3):
        if decisions[i]:
            return out[i]
    for i in range(1,3):
        if decisions[i] == None:
            return out[i]
```

2.3 程序限制

本组的算法在正常情况下一般不会出现报错的情况，但是考虑极端情况的话，也即如果当前坐标超出了格子的范围的时候，程序可能会出错。

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）Intel(R) Core(TM) i5-4200U CPU @ 1.60GHz 4.00 GB

操作系统：（名称/版本）Windows 7 32 位操作系统

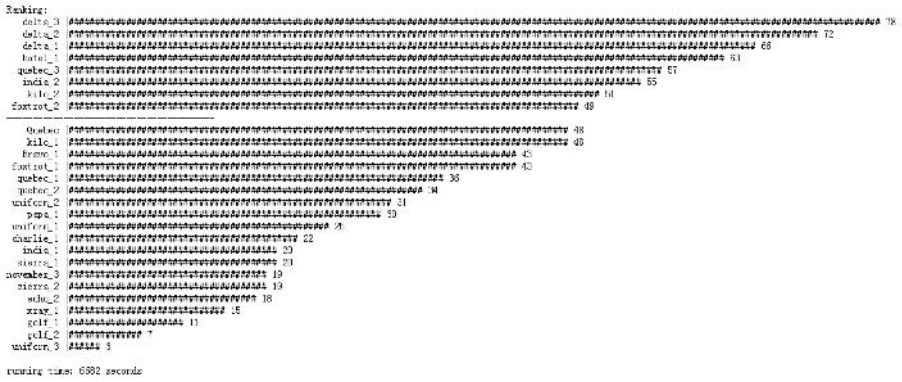
Python 版本：（版本号）python3.6

3.1.2 测试方法

本组采用的测试方法主要有两种：一种是组内对战，即大家先根据自己的思路编写一个算法，然后用这个算法进行对战，以此来发现自己算法中可能存在的问题以及需要进一步提高的部分；另一种方法是参加老师组织的热身赛，根据热身赛的情况来进一步对我们自己的算法进行改进和优化。

3.1.3 测试结果

在热身赛的过程中，我们组提交了两个代码，分别对这两种代码的性能进行测试，代码名称分别为 Sierra_1 和 Sierra_2，两个代码参加热身赛的情况如下图和下表所示：

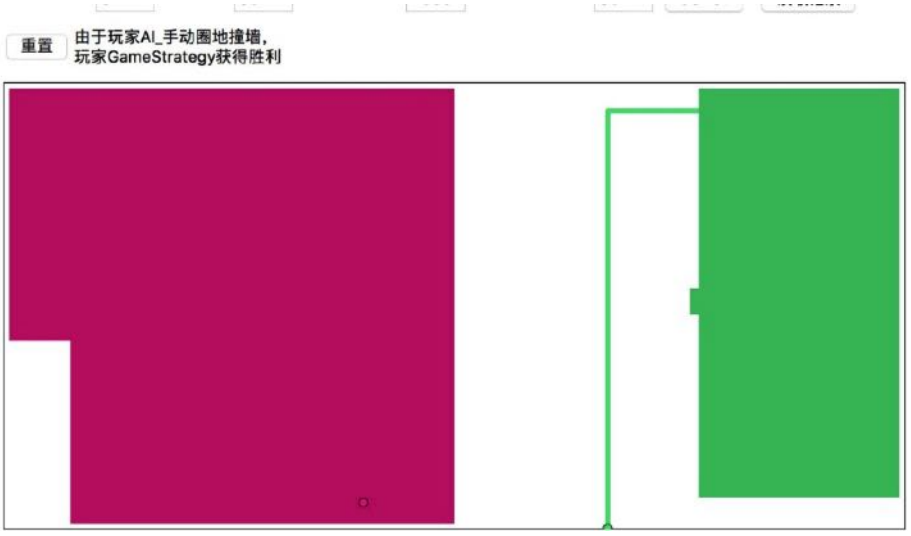


代码名称	胜局	败局	平局	得分
Sierra_1	5	19	2	17
Sierra_2	5	19	1	16

3.2 结果分析

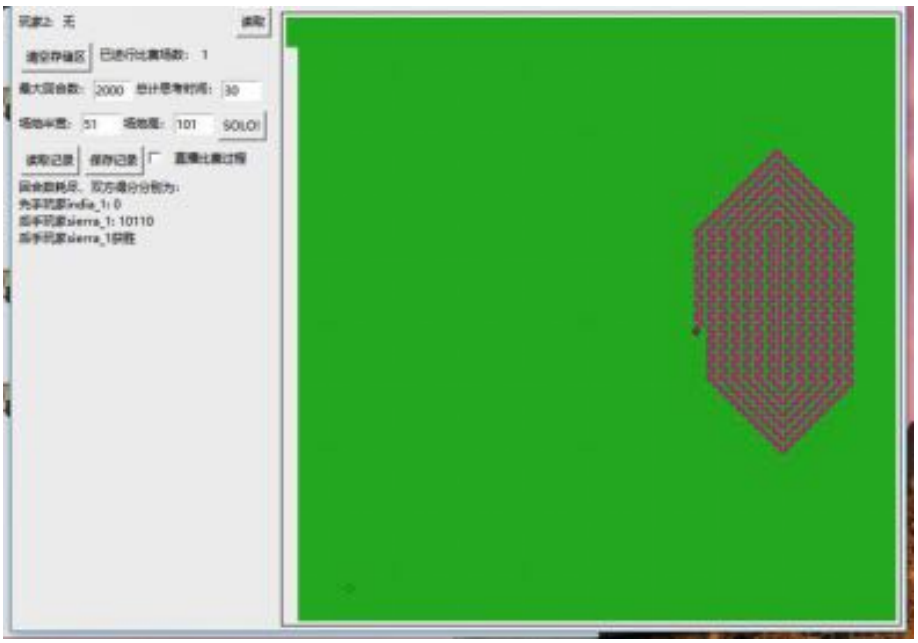
本组的算法中有很多函数是要根据计算出来的最小路径进行判断的，包括自己所处位置到敌人的位置的距离，以及自己所处位置到自己圈地范围的距离，但是在最小路径的搜索函数过于简单了，效率不是很高，在每次决策采用哪种模式的时候都需要把自己所在位置到对方区域里面每一个位置的距离都算了，再根据这个距离来判断要不要躲避，这个复杂度高，而且很慢，所以计算距离的这个函数需要进一步提高和优化。

最开始的第一版算法写出来以后，我们组内用两个同学写的第一版算法进行对战，发现其中一位同学的防护模块有较多漏洞，算法真的防护部分不是很强，所以从这一次的组内对战中得出我们需要写一个防撞模块。下图是对战的结果。



我们的策略就是要先严格保证对方进攻不到自己，也即不能撞到自己，在保证自己能存活下去的前提下再考虑进攻算法。

在热身赛中，我们用组内写出的两个算法分别和其他组的代码进行对战，测试结果是两个代码都没有报错或超时，但是算法的效率还是较低，需要进一步改进。从热身赛中我们和别人的对战记录中，可以看出大部分情况下我们都是因为被对方撞死而败下阵来，有小部分情况是因为最后的时间消耗完了。所以之后我们的代码的重点也就更多的放到了防撞这个方面，从下图我们的再一次测试可以看出，我们的圈地效率还是很高的，但是最后防止被撞死才是我们需要进一步改进算法的关键。



对算法进行优化后，我们自己又一次对代码进行了组内测试，多次测试结果显示优化吼的代码不容易出现被对手撞死的情况，但是在其中一次对战中偶然又出现了被对手撞死的情况，对战现场如下图所示。

从代码的角度具体来看，确实这个优化后的代码仍然可能会出现对方把自己撞死的情

况，我们讨论得出造成这种结果的原因是我们在代码中有一个部分的判断条件过于宽松了，也即是在判断对方能否撞到自己的最小距离的判据过松，导致在一些情况下，计算得到的自己到对方所在点的最小距离值偏大，从而使得最后在对战的一些情况下我们会被对手撞死的情况出现。所以我们队这一部分的判断条件进一步进行优化，调整过后的代码不容易出现被对手撞的情况，但是由于过分谨慎小心，所以也导致我们的圈地效率收到了一定的抑制影响。

4 实习过程总结

4.1 分工与合作

本组内分工明确，大家能够充分发挥自己的特长，共同协作，完成了本次实习。本组采取的代码编写方式是用几个人各自先按照自己的想法写几个可以对战的算法，然后彼此间进行对战，通过对战发现自己编写的代码中的问题，然后综合大家每个人算法中的优势，汇总成最终的代码。所以为了方便最后的函数汇总，我们将 play 函数中自定义的变量加在了 storage 里面，以便之后的调用。本小组分工：

刘晟: 组织讨论、代码编写、算法测试

史美琪: 代码编写、算法测试

韩子程: 代码编写、算法测试

张金霞: 算法测试、报告撰写

黄薇: 算法测试、报告撰写与排版

合作和交流的方式主要是通过微信群，为了方便大家每个人的信息同步，我们建立了一个小组的微信群，在群里大家会将自己的想法和思路提出，沟通比较及时方便。另外，对于我们在微信上没有解决的问题，组长会再召集大家开会讨论解决方案。以下是讨论的照片：

4.2 经验与教训

本次实习大家分工明确，从游戏策略的讨论、编程算法的实现、算法的改进、组内模拟对战、再到本次报告的撰写，大家既能发挥自己擅长的地方，也能相互合作提出建议，团队整体氛围非常好。

但是本次的比赛准备时间不是非常充足，而且临近毕业季组内成员都是大四的学生，需要有毕业论文、实习、毕业手续等相关事情的准备，而且从比赛通知到比赛分组再到最后的进行竞赛，时间不是特别的充足，大家经常要讨论到凌晨两三点。

所以算法中其实有很多可以再进行改进的地方，但是没有过多的时间可以去查找相关文献对算法进行改进。

希望之后可以由更加充足的时间，同时如果每个组能有一个咨询顾问之类的助教或老师，来专门可以作为技术方面的指导，最后的效果可能会好很多。

4.3 建议与设想

首先，在组队方面，比赛是由自由组队的方式，所以其实很多队伍会存在实力悬殊的问题。如果能够考虑根据大家编程的能力，然后较为均匀的分配，这样不仅可以促进大家认识彼此，而且可以有更多的技术上的交流，对于编程水平较低的同学也是一种更好的提高方式。

其次，在小组赛制上，随机分组会存在一定的公平性问题，以本次比赛的情况为例，两组很厉害的代码都在我们比赛小组，而每个小组只有一队可以出线，感觉这个赛制还有很大的改进空间。

从比赛展现的形式来看，在小组赛和八强赛的时候都是只有分数的展现，展现形式比较单一，而且缺少了可视化的氛围。如果能够在投影屏幕上分为多个区域，然后再在每个区域将现阶段所有正在进行的比赛对战进行直播的话，那么从观众的参与感来说会更强一些，而且可视化的效果也会更好。

对于选修明年课程学弟学妹的寄语：这是一堂实践和理论结合的课程，对编程的能力真的是有很大的提高，而且老师和助教们都特别的认真负责，也非常的平易近人，幽默风趣。这不仅仅是一门课程，更有机会将所学的理论性的知识变成有趣的好玩的体验，包括用 python 画图、设计游戏对战，似乎每一节课老师都会带来不一样的精彩，不一样的体验。这绝对会是你们大学阶段千万不能错过的一门课程！

5 致谢

非常感谢陈斌老师一学期的教学，这是一门真正理论与实战结合的课程，不仅仅学习理论的知识，更重要的是我们有机会自己真正去用算法实现一个游戏对战，对于一个刚刚学了四个月编程的人，就可以完成一项游戏对战，这在很多人看来是不太现实的，但是在老师和助教还有各位小伙伴的共同合作支持下，我们做到了。

另外，陈斌老师通过利用 python 编程的方式让我们对数据结构进行学习，这种在实战中体会数据结构的教学方式不仅让我们更深刻的理解了这些看似晦涩难懂的知识，同时也锻炼了我们利用 python 解决更多问题的能力，提高了 python 编程的运用能力。

同时，也非常感谢助教们的悉心解答和指导，包括每周的上机课，助教们都很辛苦的

耐心的帮大家解答我们在编程中的问题，以及这次的比赛助教们和老师们也都会在微信群中为大家随时解答对战过程可能会存在的问题，让我们在 python 的技术上有了很大的提高。

最后，也特别感谢本次小组作业中大家的共同协作，即便快到了期末季，大家都在为本次小组合作献出自己的一份力，包括算法的讨论、论文的撰写等。这也会是我们毕业季珍贵的一份回忆。

6 参考文献

1.https://blog.csdn.net/lanbing510/article/details/49912739

2. https://gym.openai.com/docs/

第十九章　N17_Tango 报告

韩豫哲 * 沈博文 刘怀亚 贺海贵 安贵沁 吴树楠

1 算法思想

1.1 总体思路

使用广度优先搜索在盘中寻找最合适的路径，并且每次根据胜负条件来判断选择前进、左转、右转三个方向是否安全以及有无机会进行进攻。进攻时按照计算得到的情况判断是进攻或是离开。

1.2 算法流程

使用广度优先搜索。先创建一个栈，之后调用 BFS_Find_Way 方法找到合适的位置，同时用 Map 记录下寻找的路径，用 Distance 记录寻到的路径的长度。因为寻路过程要将自己纸带的区域设定为禁走的区域，因此同时也就可以统计纸带长度 band_length。

使用 Move_to_it 方法前往，这个函数的逻辑基本是将寻路函数的过程倒过来，从 position 往回寻路，根据 Map 上记录的寻路过程，从 position 位置一步步倒回去，直到找到出发点并确定出发点周围哪个方向是有效的，如果该方向有效就往这个方向转。

再然后用 V_or_F 方法进行根据胜负条件、击杀条件、边界条件等进行三个方向的特判，从而知道胜负情况。

在领地外与领地内分别调用领地外/领地内模块进行决策。在领地内时，调用 BF-

S_Find_Way，对自己以及敌方情况进行分析，找最快出领地的路。如果距离足够远则用 Move_to_it 移动到那里，否则转向或直行，不离开领地。在领地外时，先判断对方是否在领地外，否则不进攻。如果自己有被对方攻击的可能性，则返回。如果离得远，且纸带不是很长，则调用 BFS_Find_Way 与 Move_to_it 进攻，否则调用 BFS_Find_Way 与 Move_to_it 回领地。

1.3 算法运行时间复杂度分析

设场地宽为 w，高为 h，路径长为 n，则 BFS_Find_Way 算法为 O(wh+n)，n 为搜索路径总长。Move_To_it 为 O(n)，V_or_F 在判断到空地头对头击杀时会调用，为 O(wh)，其他时候为 O(1)。Play 中的执行的函数实际上不存在有与上述变量相关的循环，故其为 O(wh+n)。

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与自定义类包括字典、嵌套列表、栈等，记录了游戏开始后场地以及玩家的相关信息，没有对课程上介绍的基本数据结构进行扩展和改进。

2.2 函数说明

主函数 play

参数：stat,storage

返回值：None ,Right ,Left

功能和基本策略：play 函数是游戏运行主函数，通过对传入参数（当前游戏数据和存储函数）以及内部函数的调用，根据当前局势对圈地、躲避、攻击等进行决策，然后返回游戏决策信息。采用常规的线性框架，即分为能直接击杀、在领地内、在领地外三种基本情况依次进行判断，并在相应的情况下进一步判断能否击杀、是否需要规避、目前的圈地风险等并做出相应决策

方向函数 Direction

参数：orientation

返回值：移动方向

功能和基本策略：根据数字返回对应的转向，这里 0 代表 None，1 代表 Right，2 代表 Left，基于我们代码中方向改变的方式：changed_direction = (current_direction + i + i//2) % 4，可以验证对于当前方向，i=0,1,2 分别表示了保持方向、右转、左转。

寻路函数 BFS_Find_Way

参数：who, target, B_or_F

返回值：Map, Distance, [next_x, next_y], band_length

功能和基本策略：

采用广度优先搜索（BFS），这里用栈 Stack 来实现，跟用队列 Queue 没有太大区别。参数 who 表示为谁寻路，可以是 me 和 enemy；参数 target 表示我寻路的目标是什么，这里不必给出具体的坐标，给出代表类型的数即可，比如我的 id，敌人的 id，空地 None 等，由此便找到最近的前往最近的这种类型区域的路径；B_or_F 表示是在 fields 上面寻路还是 bands 上寻路。

基本思路就是普通的广度优先搜索，同时用 Map 记录下寻找的路径，用 Distance 记录寻到的路径的长度。因为寻路过程要将自己纸带的区域设定为禁走的区域，因此同时也就可以统计纸带长度 band_length。

返回值 Map 即为寻路时用的二维数组，Distance 表示寻到的路径长度，[next_x, next_y] 表示寻到的目标坐标，band_length 为 who 的纸带长度。

比如 BFS_Find_Way（‘me’, [None, my_id], ‘fields’）就是从我的领地前往空地去圈地的寻路；BFS_Find_Way（‘enemy’, my_id, ‘bands’）就是敌方攻击我的寻路。

移动函数 Move_to_it

参数：Map, position

返回值：无返回值，直接 return 对应的转向值

功能和基本策略：

与寻路函数相对应，按照寻路函数找到的路为准，前往对应的地方。传入的参数 Map 就是寻路函数中返回的二维数组，position 是目标位置的坐标。这个函数的逻辑基本是将寻路函数的过程倒过来，从 position 往回寻路，根据 Map 上记录的寻路过程，从 position 位置一步步倒回去，直到找到出发点并确定出发点周围哪个方向是有效的，如果该方向有效就往这个方向转。

胜负条件判断函数: V_or_F

参数：无参数

返回值：Kill_or_Die

功能和基本策略：

这部分主要是根据胜负条件、击杀条件、边界条件等进行三个方向的特判。判定结果用 Kill_or_Die = [[], [], []] 来记录，正好对应转向的三个选择。其中每个内层 list 应当包含两个参数，一个 bool 型 True 或 False，False 表示如果按照某个方向走，那么下一步直接死亡；True 则表示下一步可以击杀敌人或者大家相安无事。第二个参数表示 True 或 False 时到底发生了什么，比如死亡时为’die’，击杀敌人为’KO’，相安无事为’nothing’。

该函数运行结束后，配合一个循环，如果检测到 Kill_or_Die 中每个 list 为 [True, ‘KO’] 则说明此时可直接击杀对方，也不必调用寻路函数了直接返回这个方向即可。如果没有可以直接击杀对方的选择，再进行之后的判断。

领地内决策模块

参数：无参数

返回值：无返回值

功能和基本策略：

在领地内是绝对安全的，所以这时候可以找个比较近的方向出去圈地。但同时如果跟敌人近距离缠斗时还需要判断安全性，主要判断敌我距离和地方撤回领地的距离，如果这两个值都比较小那么缠斗是有风险的，不妨怂一点。如果敌我离得比较远那么就近离开领地即可。

领地外决策模块

参数：无参数

返回值：无返回值

功能和基本策略：

在领地外是不一定是安全的，首先要判断敌我双方有没有拉出纸带，据此判断我能不能攻击对面。此时要计算敌我纸带长度，我方撤退的步数，敌方撤退的步数，我攻击对方的步数，敌方攻击我的步数等，根据这些距离判断。

比如我们采用的判断是，如果敌方攻击我方的距离比我撤回领地的距离大很多，那么如果我方的纸带拉的太长了，也要撤，在近距离缠斗时很不利；如果对方攻击我的步数比我攻击对方的步数大很多，那么此时尝试攻击对面。而如果敌方攻击我的距离以及跟我方撤回领地的距离差不多了，则直接选择回家。

2.3 程序限制

在算法的编写过程中，我们考虑了多种方案，前期主要考虑在比较保守的状态下如何最大化的进行圈地，后来逐渐意识到这个游戏的要义应当是缠斗，而单单依靠圈地面积获胜的情况其实是比较少的，所以后期转向攻守兼备的策略，按场地信息相应选择防御避战或者进攻状态，增强攻击性。通过对实验以及比赛记录的观察来看，程序在圈地效率上有的时候会明显不足，以及在一些极端情况下，会出现撞墙的现象。除此之外，由于整个算法思想相对来说简单，采用比较广泛普适的函数进行决策，对边界处缠斗没有做特殊处理，所以在这方面会稍弱一些。在领地交界处近距离缠斗时并不强，有时会因为交界处领地变化及交换太快而导致误入敌人领地较深处被击杀。

3 实验结果

3.1 测试数据

3.1.1 实验环境说明：

硬件配置：2.9GHz Intel Core i7/16GB 2133 MHz LPDDR3

操作系统：macOS 10.13.5

Python 版本：3.5.0

在测试阶段，我们采用的方式较为传统，即人工定性分析，检查代码遇到的问题，发现代码在对抗时存在的不足。我们首先编写了基础版的代码，然后在实现我们的算法思路，与之进行对抗，观察其实际表现，发现问题，进行优化迭代，再与旧版本进行对比，以确定其表现得到提升。此外，我们在线下与 F17 组的成员互相交换了代码进行对抗，以反思我们的算法思路等问题。

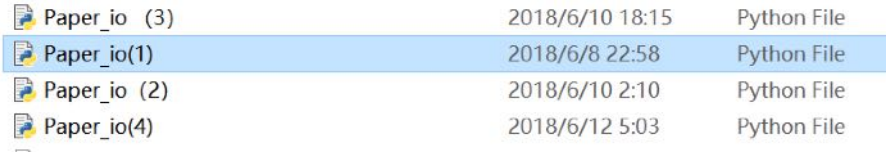
3.2 结果分析

在最初测试当中，我们主要在于发现代码 bug 问题，比如自杀现象，或在极少数情况下出现的撞墙问题，我们首先对这些 bug 问题进行了解决。

其次，我们在测试中发现，最初我们起步先圈地，占据一片较大区域的代码过于冒进，因为纸带离家太远容易被对面所截杀，但是当遇到同样保守型的代码，其优势便可以得到发挥，其圈地优势较为明显。但是没有进攻，一位防守圈地在比赛当中是处于下风的。随后，在和 F17 的同学交换代码进行对战的过程当中，我们不断发现本代码在缠斗进攻方面的弱势，并针对此进行改进。

3.3 代码迭代优化

基于在测试中发现的问题，我们进行了针对性的改进，前后共迭代了四个版本：



四个版本的基本思路是一致的，只是在细节上不断进行了优化。我们采用“回”字型策略——在遇到边界 / 遇到敌人 / 离开安全区等情况下左转，其余情况逆时针绕“回”字形路线圈地；在安全区的时候，寻找最近的离开点继续圈地。

要实现这一策略，必须给出的参量有：自己的位置（me_x, me_y），敌人的位置（enemy_x, enemy_y），自己的位置与自己领地的距离，自己位置与边界的距离，自己位置与敌人位置的距离。

在 Play 函数中，我们引进几个变量：

变量名	功能	说明
Me_in_my _ligeance	记录是否在自己领地	用一个存储布尔值的变量实现。当前在自己领地则为 True，否则 False；初始化为 False
back_path[mindis, min_x, min_y]	记录当前位置回到自己领地的最小回合	用一个列表实现。列表包含三个量，分别表示最短回合数、领地与当前位置最近点的横坐标，领地与当前位置最近点的纵坐标；初始化为 [1000, 0, 0]
dis[East, West, South, North]	记录与安全区的垂直距离	用一个列表实现。列表包含四个量，分别表示沿四个方向（东西南北）与安全区的垂直距离；初始化为 [0, 0, 0, 0]
danger	记录是否有危险	用一个存储布尔值的变量实现，当前有危险则为 True，否则为 False；初始化为 False
closest_distance	记录与敌人的最近距离	用一个存储整数值的变量实现。初始化为 1000.
depth[East, West, South, North]	记录在安全区时与边界的最小距离	用一个列表实现。列表包含四个量，分别表示四个方向（东西南北）与安全区边界的垂直距离；初始化为 [0, 0, 0, 0]

代码的思路可以被简单地概括为：

IF Me_in_my_ligeance THEN

IF danger THEN

Back(回家)

ELSE

Expand（圈地）

END IF

ELSE

Leave（离开安全区）

END IF

当然，实际写代码的时候，Back，Expand 和 Leave 也需要有一些判断过程，例如需要引入棋盘边界的判断、圈画的太大了需要转向避免危险等等。

第一版的代码就是以此为框架写出来的。但是在实际运行的时候发现几个不足。于是在第二版做了如下更动，增加了安全区预判。

```
dis_min = min(dis[0], dis[1], dis[2], dis[3])
dis[(2 + stat['now']['me']['direction'])%4] += 1
next_min = min(dis[0], dis[1], dis[2], dis[3]) # 如果按当前方向行走一步
```

1. 在计算安全区垂直距离的同时预判下一步如果走了，距离安全区的新垂直距离。

2. 增加了在安全区内最近表面方向的标记 min_dir，并令纸带从这个方向离开。

第二版的功能和实验结果比第一版有明显的改进，但仍然效果不理想。我们在第三版进行了以下改动：

1. 安全区内增加离开方向与边界远近的判断，如果距离边界太近则标记此方向为危险方向，继续从其他方向寻找离开的方向。

2. 上调了安全区判定下限（从 11 到 13），让纸带的运行更安全：

```
# 如果没有上述危险情况，正常运动
if ((next_min >= 13 and next_min < 200) and back_path[0] == dis_min) or

    # 安全地逆时针绕圈
    if next_min == dis_min and next_min >= 11 and back_path[0] < 15:
        return 'None'
    else:
        return 'Left'
else:
    return 'None'
    # 逆时针走
```

（上图：将安全区判定从 11 改为 13，采用更安全的策略）

然而，前三个版本在对战的时候无疑没有优势，遭遇敌人直接判定危险而选择安全退回。这一坚壁清野战术在游戏后期难以取胜。因此，第四版引入了缠斗部分的代码。

要进行缠斗，首先需要判断缠斗胜负的几种情况，亦即引入胜利条件（直接纸带触杀，领地内碰头击杀，空地侧面撞头击杀，空地冲撞判断领地大的击杀领地小的）和失败条件（出界，撞到自己纸带，在对方领地触碰到对方纸带）。为此，引入一个二维列表 Kill_or_Die 来记录三个方向的安全性及预判的进攻结果。

同时，为了做出上述判断，必须引入一个实时状态的地图 Map，Map 可以使用二维列表实现。

第四版与前三版相比，做了如下更动：

1. 引入记录棋盘实时状态的二维列表 Map；

2. 引入条件判断列表 Kill_or_Die；

3. 引入缠斗代码，在保证安全性的同时提高胜利的可能性；但保留了纸带过长则回到领地和原本的安全性判断系统——用以处理突发情况。

在增加了缠斗代码后，在对战当中表现的更加精彩好看，面对于一些规则较简单地代码也可以取得胜利，但是由于整体上代码风格还是趋于保守，只是增加了防守抵抗能力，在进攻能力上还是比较欠缺，所以在最终的小组比赛中也处于中游。

4 实习过程总结

4.1 分工与合作

秉承着自愿、自主、发挥个人所长、重在参与的精神，小组分工如下：

韩豫哲（组长）：负责项目管理、制定策略、进行代码测试、优化

沈博文：技术负责人，负责主要算法和基本代码实现

吴树楠：demo 开发和算法讨论

贺海贵：算法补充和部分代码实现

刘怀亚：负责代码优化、改进

安贵沁：测试工程师，负责代码测试、反馈和优化

部分活动照片如下：

第一次线下讨论，在理教自习教室

第二次线下讨论，二教三楼讨论空间

此外，由于成员年级较分散，所以也在线上进行了多次讨论。

4.2 经验与教训

在本次作业实习中，我们小组在设计游戏的过程中并不是一帆风顺的。在设计中遇到了一些问题。

4.2.1 战术方面

战术方面的问题是我们小组在设计中遇到的一个最大问题。首先我们对于这个作业设计一直持有保守的态度。在设计作业中，我们小组设计的纸带是这样走动的：1. 当没有危险的时候，是迭代圈地。2. 遇到危险时，就是遇到别的纸带时，我们的纸带就会规避危险，具体见下图所示：

```
# 下面这段相当于规避危险，往家里撤
# 如果敌人离我近到需要马上撤的地步（目前我不在领地内），判定危险
if closest_distance < back_path[0] + 5 and not Me_in_my_ligeance:
    danger = True
# 如果比较安全且画的圈不是太大就继续走(离边界还比较远)；画的圈大了

if not danger and not Me_in_my_ligeance:

    if (dis_min >= 6 and dis_min < 200) and back_path[0] >= 13:
        print(dis_min, end = ' ')
        print(back_path[0], end = ' ')
        print('fuck', end=' ')
        print(id, end=' ')
        print(4000 - stat['now']['turnleft'][0] - stat['now']['tur
        return 'Left'
    else:
        return 'None'
        # 逆时针走

# 往大致最近的领地上撤

# 不安全，回家
if danger == True:
    if me_x > back_path[1] and me_y > back_path[2]:
        # 在最近的领地的右下方
        if stat['now']['me']['direction'] == 3:
            return 'None'
```

这只是改进前代码的一部分，可以看出我们采用的时规避法来应对挑战。这个方案在我们与其他组进行课下比赛的时候，完败！其中最大的原因就是我们小组所采用的保守战术。这是我们战术上的失策，没有对敌人亮剑，是我们的刚开始时的失败的地方，不过后来我们对其代码进行了改善，增加了缠斗能力，其表现胜于消极避战。

4.2.2 代码方面

针对代码方面，我们小组进行了多次的更改调试。从一开始的冒进式打法到后来的迭代递归式圈地，这些改变也让我们组的成功率大大上升。我们组一共设计了四种方案，每一种方案都是前一种方案的改进版。见下图所示：

在第一种方案中，所遇到的问题是：纸带的打法过于冒进，就是纸带在圈地的过程中，一次所走的路程太远，导致很容易被敌人给攻击，失败率很高，所以我们舍弃了这种方案。第二种方案和第三种方案所遇到的问题差不多，纸带在行走的过程中，当进入别人的领地时，就会失去方向，然后“自杀”。这是一个很严重的问题，我们后来对其进行了改进，也就是第四种方案。第四种方案总的来说达到了我们的预期，迭代圈地，同时也会适当的规避危险，进入别人的领地后，依然会按照原来的计划进行，最后在测试时，取得成功的概率较大。

对于战术上的改进：我们在第四种方案中，没有再一味的躲避，我们采用的是判断位置，多种的进攻方法，具体见代码改进：

```
# 根据胜负条件来判断三个方向的安全性及进攻性
for i in range(3):
    next_direction = (current_stat['me']['direction'] + (i + i // 2)
    next_x = current_stat['me']['x'] + move_direction[next_direction
    next_y = current_stat['me']['y'] + move_direction[next_direction
    # 死亡条件
    # 出界死亡
    if next_x < 0 or next_y < 0 or next_x >= stat['size'][0] or next_
        Kill_or_Die[i] = [False, 'die']
    # 撞自己纸带死亡
    elif current_stat['bands'][next_x][next_y] == current_stat['me']
        Kill_or_Die[i] = [False, 'die']
    # 在对方领地碰到对方纸带头死亡
    elif current_stat['fields'][next_x][next_y] == current_stat['ener
            next_x == current_stat['enemy']['x'] and next_y == curre
        Kill_or_Die[i] = [False, 'die']

    # 胜利条件
    # 直接纸带触杀
    elif current_stat['bands'][next_x][next_y] == current_stat['enemy
        Kill_or_Die[i] = [True, 'KO']
    # 领地内碰头击杀
    elif current_stat['fields'][next_x][next_y] == current_stat['me']
            next_x == current_stat['enemy']['x'] and next_y == curre
        Kill_or_Die[i] = [True, 'KO']
    # 空地侧面撞头击杀
```

可以看出，我们在战术上，从原来的一味的防守，变成适度进攻，这样的话，我们所取得成功率就会更高。

在代码方面的改进，我们的第一种方案是不可取的，所以我们在第一种方案的基础上把每次圈地的范围给缩小一点，这样取得效果就会更好一点。但是改进之后，第二种，第三种方案都出现了“自杀”的现象，这显然不符合我们的要求，所以我们对第三种方案进行了改进。改进的方法大致有这两点：

1. 在方法上，我们采用了一种全新的方法。我们通过出入栈的方式进行广度优先搜索（BFS）。这和我们之前的方法完全不同，大大提高了我们的成功率。见下图：

```
                Map[i][j] = 40000
# 将起点入栈，开始BFS
next_step = Stack()
next_step.push([current_stat[who]['x'], current_stat[who]['y']])
while not next_step.isEmpty():
    Distance += 1
    temp = Stack()
    while not next_step.isEmpty():
        top = next_step.pop()
        # 搜索邻接点
        for i in range(4):
            next_x = top[0] + move_direction[i][0]
            next_y = top[1] + move_direction[i][1]
            if next_x >= 0 and next_y >= 0 and next_x < stat['si
                if Map[next_x][next_y] == 50000:  # 相当于visite
                    temp.push([next_x,next_y])
                    Map[next_x][next_y] = Distance  # 把地图直接
                    if current_stat[B_or_F][next_x][next_y] in t
                        # 四个返回值
                        # map, 一系列坐标，当前拉出的纸带长度，需
                        # print(4001 - current_stat['turnleft'][
                        # print(band_length, end=' ')
                        # print(Map[next_x][next_y])
                        return Map, Distance,[[next_x, next_y]],
    next_step = temp
# 寻找失败，返回一些失败的值
```

2. 在作业实习的过程中，我们把进入别人领地的这一部分代码进行了改进，当进入别人领地后，我们的纸带不再晕头转向，而是通过判断是否安全来选择自己的走动方式。第四种方案中，我们的纸带进入别人领地后会进行判断得到两种结果： 敌方纸带距离较远，地方纸带距离较近。通过这两种判断做出的判断也不一样，第一种判断所采用的应对技巧是继续走自己的路线；第二种判断采用的应对技巧是应用前面所提到的击杀代码，进行击杀。通过代码的改进，使得我们小组比赛成功的概率大大上升。

4.3 建议与设想

由于已经划分两个阵营，所以我们建议比赛是否可以首先进行一轮循环赛，划分档次，以此来确定成绩。此外，在设置趣味的大作业项目时，也可以群策群议，让同学们多提出想法，以保持和增加作业的趣味性，调动同学们积极性。

5 致谢

首先要感谢陈斌老师和各位助教以及技术组同学的精心策划和辛苦工作，给大家提供了一次充满乐趣和自我提高的实习机会。

其次，要感谢我们 N17_tango 组的成员们，我们的年级跨越大二到大四，各人或实习、或申请、或学习事务都比较繁忙，但在大作业上都表现的非常积极。由于我们组大多是文科生，且在编程方面受到的训练较少，所以整体实力上相对欠缺，但依然尽己所能的完善作品。首先要感谢韩豫哲同学，他在算法思路到最终的测试、修改的全过程中都付出了很大的精力。其次要感谢我们的技术总负责人沈博文同学，在他的指导下，各人都在最终的作品实现上发挥了自己的作用，且学习到了很多，在此祝愿他申请顺利！此外，我们在具体的实现过程中，也向身边的朋友请教，在此也一并向他们表示谢意！

第二十章　N17_Uniform 报告

席子涵、杨欣翰、沈铂涵、张俊峰、姜畅

摘要：纸带圈地游戏是双人竞技性游戏，双方玩家分别控制己方纸卷，在避免己方死亡的前提下，尽量在有限时间内圈地面积最大化。根据游戏的双方竞技的性质，本次纸带圈地算法中将代码实现整体框架结构成为进攻、防御和圈地三个状态，各部分之间基于当前双方纸卷、纸带和纸片的当前状况和对于未来行动的预期进行动态转换，达到随时适应双人竞技的场上局面，保证己方策略处于最优的情况。

三个部分中分别涉及到了广度优先搜索、决策树和最大化矩形圈地策略算法以及数、队列、散列等数据结构。在进攻部分的具体实现中，分成短距离博弈性进攻和长距离必杀性进攻。长距离必杀性进攻是根据双方纸带的距离情况进行广度优先搜索，计算出双方最短的回家距离以及己方进攻与被反击距离，根据距离的大小进行是否采取进攻的决策。而短距离博弈性进攻则是基于决策树的模型，采用经济学子博弈精炼均衡的思想，综合考虑双方的执行决策和未来的行动，经过理性分析后采取精准的前进、左转和右转的行动。在这里，必须指出的是，进攻状态处于最高优先级，即一旦进入进攻状态，后续状态将不再会进行判断或者执行。在第二部分防御部分中，则类似于上述的进攻算法采取广度优先搜索，确保己方处于不被对方可以必杀的状况中。对于防御算法，我们则必须要指出的是如果每一回合均采用这种广度有限搜索，己方便会将过多的时间用于分析这种在多数情形下均无效的决策中。因此，本方采用了矩形判断法，利用矩形的特殊性质和简化运算的方式作为是否启用防御算法的调用判断基础，从而大幅度减少了调用广度优先算法的时间消耗。最后一个部分是圈地，我方考虑采用圈地算法采用的是取点移动法，利用圈地面积最大化和风险最小化两重标准结合衡量，找寻目标点。接着根据目标点的方向进行行动。值得指出的是，在圈地算法处于最低的优先级状态，其中可以被攻击算法和防御算法打断。

就实验结果而言，我们小组主要关注了在比赛之前进行的三次热身赛的结果。从第一次热身赛中简单算法取得的良好结果中，我们意识到了一个稳定而可重复的圈地算法是整个算法的基础。而在结合了本组的其他两类算法方式后，我们的改进版算法可以进行有效的探测危险并回避和长距离攻击功能，但由于文件命名关系没有能参与第二次热身赛。在第三次热身赛中，我们意识到了我们的算法的严重超时问题，在总共的比赛中有接近一半的超时情况。在正式比赛之前的晚上，我们成功地解决了时间问题。同时，我们在经典战局中具体讲述了主动攻击性算法和相对保守的算法的区别以及在实战中的差异，并且讲述了我们研究攻克一个困难的 bug（走出诡异的曲线）的过程。

关键字：广度优先搜索、决策树、最大化矩形圈地策略、队列、散列

1 算法思想

1.1 总体思路

本次的 uniform 组的 paper.io 的算法设计思路主要包括以下五个部分： 获取必要的数据、 进攻模块、 防守模块、 圈地模块、 整合模块、

其中获取数据的逻辑思路如下：

本次程序设计中，小组发现获取数据的速度影响着对局中耗时的多少，特别会影响小组是否产生超时的情况。如果小组采用每回合全盘遍历的方式进行当前地盘、当前纸带和当前边界点的搜索和分析，会产生大幅度的时间消耗。本小组设计了一种方案，可以使本组程序仅仅在特殊的需要刷新的情况下，才进行全局的刷新。主要的代码逻辑如下：

判断是否为需要刷新的回合

领地区域面积的变化总是发生在纸带回归领地的回合中，因此我方认为如果纸卷上一回合所在的位置在上一回合还并非是我方的领地，但是在本回合变成了我方的领地；同理，对方纸卷上一回合所在的位置在上一回合还并非为对方的领地，但是在本回合变成了对方的领地。说明了双方的领地的面积发生了些许的变化，在这种情况下，我们需要对于全屏幕的 fields 数组进行搜索，更新领地情况和领地边界情况。

而对于纸带的更新，则是利用双向链表 list 自带的 append 函数功能，进行增加当前的纸带所在的位置。特别地，在己方纸带回归了领地的时候，list 清空，仅仅保留当前的纸

卷所在的位置。

通过上述两种方式定期刷新数据，大幅降低了程序的耗时情况。

	数据准备	short attack	long attack	danger detect	escape	normal	error
总耗时	23.81168	2.119654	0.38724	1.783929	0.013011	1.419556	0.088822
单次	0.081083	0.210049	0.011232	0.024229	0.013011	0.014861	0.023097

从上述的表格中，我们可以清晰看出如果采用遍历的方式进行搜索的话，会在 4000 回合内超时，并且其中将近 5/6 的部分都是由数据准备而造成的。

获取数据的方式

在进行领地刷新的时候，本组考虑需要额外获取两组数据，分别是所有边界点和能够到达的边界点。采用判断是否仅仅只有两个或者三个相邻点来判断领地的点是否为边界点。对于所有的边界点而言，还需要特殊考虑在边界上和与纸带相接壤的点的特殊情况。这两种类型的点对于领地外的纸卷而言并不算是真正的边界点，因为这些边界点对于纸卷而言是不可以到达的，无法实现回家和从家中离开的操作。

其中防守部分的逻辑思路如下：

本次 paper.io 游戏是一个生存类对战游戏，其中最重要的部分便是如何在圈地和对抗中生存下去，并且占据更大的领地范围。小组将防守算法列为代码设计的核心板块，并为之付出了大量的时间和心血，成型的防守算法主要包含以下的三个模块

搜索己方的回家路径的模块

离开领地后，己方的纸带便会暴露在对方的纸卷攻击威胁内，因此随时需要考虑自己是否可以回归自己的领地，躲避对方可能带来的威胁。搜索回家路径主要采用广度优先的搜索方法，避开己方已有的纸带和场地的边界，选择一条可行的最短路径。

在选择回家的路径的过程中有两点需要特别注意：

第一点是可行的回家路径的选取顺序，在使用广度优先的算法进行实现的过程中，由于广度优先对于前、右、左三个方向的搜索是采用一定顺序来实现的，从而带来的结果便

是，到达某一个地点的最佳路径总是会尽可能多地采用排列在前侧的方向实现。我方考虑的结果是，尽量保证走直线，而不是左转和右转。如果采用左转和右转作为第一选择的话，容易造成纸带沿着来时的路回家，从而无法实现圈地的效率的最大化。而如果采用直线作为第一选择的话，则可以尽可能地实现圈地的效率的最大化。

第二点是在圈地函数行走到路径的边缘的时候，由于圈地函数本身的特殊设定，会尽可能选择离边界较远的一边转弯从而有利于后来的圈地。然而，为了避免在转弯后突然发现无法返回到自己领地的情况，我方对于此种特殊现象进行了特殊的分析，即采用模拟（Simulated reality）进行的方式进行假设分析。假设我方采取左转的方向，利用此种情况进行分析回家的路径和是否选择使用本路径回家；同时对于右转的假设同样分析，尽可能在保证己方安全的情况下选择最佳的圈地方式。

判断是否采用回家路径的模块

搜索到己方回家的路径之后，需要考虑己方采用此种路径是否会被对方所击杀。即着重分析对方对于己方回家路径的威胁情况。实际上，部分小组的函数并非如此只能，并不会选择长距离攻击的方式去威胁对方，但作为一个风险厌恶的保守小组，经过了双方博弈的分析后，采取了保守的策略，即一旦对方即将可以有机会进攻自己的纸带时，便采用逃跑算法回家

其中攻击部分的逻辑思路如下：

经过协商讨论，小组将攻击部分的思路主要拆分成为两个部分，远距离进攻部分和近距离进攻部分。与此同时，由于小组成员都偏向于保守，对于风险较为厌恶，因此采用的进攻策略均为必杀策略，即一旦进攻，就不会为对手留下生还的机会。

短距离进攻的逻辑主要包括如下三个部分：

构建树形结构

短距离进攻，小组考虑采用博弈论的思想实现博弈算法，即完全考虑双方的决策可能和结果。在本局游戏中，双方的选择情况非常单一，仅仅只有前、左和右三种选择方向，同时根据是否撞墙、是否撞到自己进行 alpha-beta 剪枝操作，以减少搜索的复杂程度。

为了体现出算法的必杀情况，本树形结构在向下搭建的过程中如果己方撞上对方的纸带，或者其他的胜利情况的时候，则会对当前选择节点赋值为 1，如果对方撞上己方纸带或其他操作导致己方死亡时，则会徐娜则赋值为 -1.

同时，特别需要注意的是，在进行向下构造树形结构的过程中，为了帮助得到最佳的选择路径，以及进行完全充分分析，我们还将从第一层博弈开始到达当前层数的博弈过程和纸带上一步决策后的纸带占用的格点的情况赋值给当前节点，便于最后的节点选择。

序贯博弈（Sequential game）中的子博弈精炼均衡（Subgame refinement equilibrium）

本组成员均有着经济学背景，联系经济学中的序贯博弈的概念，借助上述搭建的树形结构进行了深入分析。小组研究发现，对于这种序贯博弈的模型，我们应当采用倒推法（backward induction）进行分析，轮到我方进行决策时，我方会选择使我方利益最大化，即选择树形节点分值最大的情况；轮到对方进行决策时，对方会选择使我方利益最小化，即选择树形节点分值最小化的情况。

基于上述倒推和 minmax 的分析考虑，小组采用了 score 评分函数递归实现这种后进先出（LIFO）的结果，并返回了当前选择的实现路径和能否必杀对方。

值得注意的是，这种方式是可以必然判断出短距离必杀情况和短距离被必杀情况的，因为在 minmax 的多层博弈中，任何一种可能必杀情况均会在对手 min 的过程中被对手所避开。当然，其中不可以避免会产生对方回家等场地发生变化的复杂现象，基于不冒风险的保守策略，短距离进攻算法预测到对方可以返回家时便会选择取消此次进攻避免被反杀。

方向决策

根据上述子博弈精炼均衡情况，得到最优的一组行走的路径后，利用 chooseBest 函数返回最终的最优当前行走路径。此时，仅仅对于返回的得分为 1 的情况进行判断，对于其余的情况均选择略过，将躲避的过程留给防御算法。

长距离进攻算法的逻辑主要包括以下两个部分

BFS 搜索必杀情况

长距离进攻算法无法采用完全的博弈决策，因为完全博弈决策中包含过多对于长距离进攻无意义的分支，例如不同方式到达同一点的情况。这种情况对于短距离博弈而言是非常重要的，影响到双方胜负的结果，但是对于长距离而言则是损耗过大，得不偿失。

对于长距离必杀而言，为了确保有效性，小组采用了广搜算法进行分析，即考量己方进攻对方纸带的距离，对方回家的最短距离和对方进攻我方纸带的距离，此时我方的纸带是包含我方进攻对方会拖出来的纸带的。只要我方进攻对方距离小于后两者，那么我方会采用远程攻击策略。

特别地，远程攻击策略中包含大范围的广度优先搜索，如果频繁调用效用会比较低下，因此小组设置了门槛，只有触发调用门槛的时候才会调用广度优先搜索。门槛是基于简化版的必杀思路，即仅仅考虑对方到我方已有纸带的折现距离，我方到对方的最短折线距离和对方回家的最短折线距离分析，忽略掉路径中可能存在的障碍物。如果满足条件，便调用深层次的搜索函数。

长距离进攻路径的转化

考虑到广度优先搜索的低效性，本小组采用一旦路径计算成功便直接使用的方式实现，即当某一回合判断我方存在远距离必杀的机会的时候，我方便会将本回合和之后回合会采用的路径存储到 storage 中，并逐回合采用 storage 中存储的路径进行移动。

由于我方的 BFS 判断必杀条件极为严苛，因此一旦调用长距离攻击函数，一定可以必杀对方；同时必杀对方的过程中也无需考虑被对方反杀的情况（已经在是否可以必杀的过程中分析过），在这种情况下，我方可以减少多次的重复计算。

上述两个进攻算法均是考虑到必杀对方的情况，在剩余很多存在击杀概率的情况中，保守的我方纸带选择不去进攻，继续执行保守的圈地算法。保证自己的领地范围最大化。

圈地算法采用的是取点移动法，find_target 函数和 find_backpoint 得到目标点和返回点。Move 函数输入点列，逐一得到按照点列行进的方向并输出。Move 函数输入的是起始点和终止点，用 get_right_direction 函数得到移动的正确方向，如果当前已经是正确方向就保持直走，不是正确方向则专向，这就确保了在无干扰情况下是折线围地的，我们将其命名为“最大化矩形圈地法”。

算法的圈地效率取决于找点方法，比如方向向东离开边界时，以当前离开方向看，在矩形边界（将所有有我方领地的点连成一个大矩形）的右上方或者右下方选择目标点，随机性基于与对手的距离，与边界的距离，为了破解贴边界一大圈圈完所有的算法，在选择到贴近场地边界的点时会直接改成场地边界的点，以切断敌人的场地边界的领地。

圈地过程每回合都要执行威胁判断和必杀判断，必杀算法优先级最高，一旦进入其他算法都会失效，其次时威胁判断，有威胁时就会触发回家算法，按照找到的路线直接回家，在无干扰的情况下就会持续的圈地。

在家中的部分是算法的中枢部分，如何离开边界，如何根据边界找目标点，如何在威胁接触后跳转到正常状态都是在家这一部分算法需要解决的问题，其主要思路是为可以离

开的点进行打分，评分系统考虑以下四个因素：

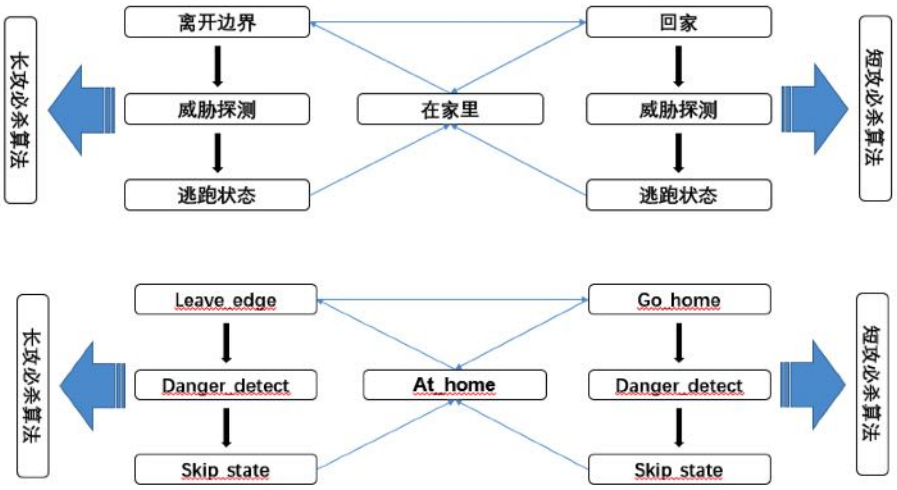
边界点与当前位置的距离，越近越好（避免走无效路径），但距当前位置 10 步以内的距离均视为 10

边界点距离边界的距离，由于每个边界点只有一个出门方向，离边界越远越好便于圈地，但距离边界超过场地宽/高的 1/3，视为距离仍为场地宽/高的 1/3

走出边界点一格后，垂直方向上与自己领地的距离，利用 search_around 函数实现垂直方向上离自己领地越远越好，避免沿着自己的领地外一格进行低效圈地

边界点与敌人的距离，进攻型策略认为离敌人越近越好，距离 20 步以内的距离均视为 20

1.2 算法流程图



1.3 算法运行时间复杂度分析

算法中采用将功能封装为函数的方法，主程序部分没有循环语句，只有条件语句和赋值语句，函数的复杂度决定了算法的复杂度。

涉及遍历全图的函数复杂度为 $O(n^2)$，比如寻找我方、敌方领地边界点，纸带点和纸卷位置，不过值得注意的是我们采用了维护的方式，只在领地改变时才重新刷新，大大减

少了函数的调用次数。

寻找最短回家路，长距离寻找必杀攻击路，采用的是 BFS 算法，算法复杂度是 O(|V|+|E|)。

圈地函数是 O(1) 复杂度，随机选取目标点和回家点，但是在靠近边界时直接选择边界点。

短距离攻击算法是采用搜索树的算法，每个点有三个分支，一共搜索 6 层，是指数级复杂度，但只会在我方纸卷和对方纸带小于一定阈值才会调用。

2 程序代码说明

2.1 数据结构说明

短攻击算法采用树的结构进行数据存储与搜索，定义了 TreeNode 类。

在具体应用中，因为本游戏中每一步有三种决策，故对于上课所讲的二叉树进行了拓展：

（1）加入 MiddleChild，即采用三叉树的形式为后续算法实现提供便利；

（2）在具体应用中也根据需要定义了是否有所有子树（hasAllChild）和是否有子树（hasAnyChild）；

（3）根据是否撞墙、是否撞到自己进行 alpha-beta 剪枝操作，既防止了自杀的发生，也减少搜索的复杂程度；

（4）通过不断实战测试所有时间调整树的层数，最终确定为 6 层。

长距离攻击算法广度优先搜索的方式进行判断，主要采用了队列（queue）的数据结构

在具体的应用中，主要利用了队列这种容器适配器的 FIFO 的特殊性质，对于其中存储的数据类型进行了部分修改：

（1）存储的数据中需要保留行走的路径，考虑到 python 的列表是传导引用，因此调用 python 自带的 copy 库中的 deepcopy 的方式实现路径的增删。

（2）使用广度优先搜索的过程中，逐层向外扩散的过程中，每次到达一个新的节点，

变会将本节点添加入队列中，根据队列的顺序依次对于点进行搜索。

（3）本小组在最开始考虑对于广度优先搜索进行剪枝，在研究过程中发现如果按照大方向进行剪枝容易产生因为纸带的阻拦而无法顺利找到回家的路的现象，因此小组放弃了采用大方向进行剪枝的方式。

2.2 函数说明

短距离攻击算法函数集合

短距离攻击算法基本策略是基于倒推和 minmax 的分析考虑，小组采用了 score 评分函数递归实现这种后进先出（LIFO）的结果，并返回了当前选择的实现路径和能否必杀对方。

共包括以下四个函数：

（1）generate(node, ID, maxdepth=6)：

用于给定当前情况生成返回一个六层的决策树，根据当前的前进方向分为东南西北四种情况，每一种情况中对于向左、前进和向右三种情况分别进行相似处理：如果发生该步有撞击，则我方获胜记为 1，对方获胜记为 -1；若不发生撞击，则分数记为 0，, 并继续使用 generate 递归，层数减一。

（2）score(node)：

用于在决策树构建好之后根据子树的分数更新该点的分数。基本算法是从下往上利用 min-max 博弈算法分析是否可以必杀对方：若是对方决策（奇数层）会选择最小分数，若是我方决策（偶数层）会选择最大分数。

（3）choosebest(node)：

用于判断更新后当前是否分数为 1（即是否有必杀策略）。

（4）shortAttack(distance)：

通过调用之前定义的三个函数，得到最后的移动情况。

获取方向函数——get_direction(startpoint, endpoint)

该函数目的在于给定起始点和终止点后判断下一步的正确方向。起始点和目标点相同

则返回空列表，起始点和目标点连成直线则返回一个方向；起始点和目标点连成折线返回两个方向。

移动判断函数——move(current_direction, startpoint, endpoint)

move 函数搭配 get_direction 使用，是生成信号的函数，需要给定起始点，终止点，当前的方向。如果当前的方向是正确的，就直接直走；如果逆时针旋转的方向是正确的，就左转；如果顺时针旋转的方向是正确的，就右转。同时，需要考虑背后有纸带的问题，这个时候正确的方向是向后，于是随机左右。

绝对距离计算函数——dist(a_point, b_point)

该函数可以求任意两点间的绝对行走距离，使用纵坐标的差的绝对值和横坐标差的绝对值之和。

BFS 算法——同时用于长距离攻击路径与逃跑路径寻找

——attack_and_home(stat, cur_place, my_num, enemy_num, attack_or_home)

该函数利用 BFS 搜索回到自己地盘/进攻敌人纸带的最小距离，两功能类似故合并为一个函数。该函数通过储存场地上每一个点的信息并编码场地所有点，用 0 到 (场地面积 -1) 的这些整数表示所有点。将所有点都初始化为一个列表，fields[i][0] 表示该点距离 BFS 起始点距离，fields[i][1] 储存搜索树上的父节点的编码。以向北搜索为例，每次将队首的点弹出记为当前点，进行搜索操作。如当前点不与北侧边界相邻时，向北搜索；如果找到符合目标点条件（我方领地/ 对方纸带），就记录距离、父节点，记录该点编码，break 跳出循环。而当未找到目标点时，将当前点加入队列末尾，记录父节点，记录 BFS 搜索距离。

最终，该函数会记录最小距离以及到达目标点的路径列表。

威胁距离估算函数——cal_threat_dist(stat, all_areas)

该函数判断当前对手的位置与我方纸带及潜在的回家的路之间的最小距离，称作威胁距离，返回值为该距离。主要用于防守算法之中。

可返回边界搜索函数——find_edges(stat, my_num, all_edges)

该函数传入我方全部边界点，筛选出当前可以返回的边界点。其中，当上下左右都是自己地盘/自己的纸带/地图边界时，则不是可以返回的边界点，其中的判断遵循：是否靠近边界，是否与我方地盘/我方纸带相邻的顺序。最后返回储存可返回边界点的列表。该函

数也是防守算法的重要部分。

边界与纸带搜索函数——

edges_bands(stat, storage, my_num, enemy_num,cur_place,enemy_place)

该函数是我们的关键函数，最终将会返回我方和敌方的边界和纸带。这对于防守和进攻都非常重要。以找寻己方的纸带为例，我方先手的第一回合/我方后手第一回合时，可以根据我方的当前位置/上回合位置直接判断边界和纸带，其余情况，每次我方/敌方从领地外回到领地刷新双方边界。这样就不会导致每一回合都在刷新，而是少部分的回合才会刷新。在寻找边界时，需要遍历地图所有的点，如果遍历到的是自己的地盘，而上下左右之中有不到 4 个方向都是自己地盘/地图边界时，则是边界点，而敌方边界亦然。该函数还考虑到与敌方地盘相邻以及靠近边界的情况，防止出现低级失误。总之，该函数的精髓在于如果当前回合双方没有回到自己地盘，且当前不是在纸片中时，边界点不变，储存纸带的列表加入当前双方的位置。这使得我们的整体运行时间大大缩短，解决了超时的问题。

矩形计算函数——rect(my_edges)

该函数获取我方地盘在东西南北四个方向的最边界位置，即我方领地能延伸出的最大矩形。这一函数在寻找纸片外目标点时有用。

圈地第一目标点寻找函数——find_target(current_direction, rectan, cur_place)

该函数是正常圈地时使用，寻找圈地时的第一个目标点。该函数仍然使用了 random 函数来在小范围内选取点，但有着很多的分情况讨论。比如，当方向向东，当前位置在整个场地的右 1/2 时，目标点取在右边界上；对手距离较近时，目标点横坐标在 (x+2,x+6) 范围内随机选取，其余情况目标点横坐标取为我方领地矩形的右边界。而这时的纵坐标将随机选取上方和下方的两点，选取概率取决于当前位置纵坐标，越靠上越倾向于取下方的点，越靠下越倾向于取上方的点。这样，我们可以保证一个可以控制但又不绝对的圈地模式，其圈地的效率仍可以达到相当高的水平。

圈地返回点寻找函数——find_backpoint(startpoint, current_direction, cur_edges)

该函数寻找正常圈地时的返回点，而实际上这一选取非常简单，即出发方向背后的所欲边界点都是可以选择的。比如当前方向向东时，西侧的所有可返回边界点都可作为 backpoint。值得注意的是，为了防止在自己领地已经占领多个边界点时找不到返回点，在这时会默认选取 cur_edges 列表中的第一个边界点返回。

垂直方向探测函数——search_around(stat, cur_place, cur_direction, width, my_num)

该函数用于搜索给定点的特定方向上与自己领地的距离。比如传入方向为东、西方向，将搜索该点南、北与自己领地的距离的最小值。

下回合方向计算函数——get_next_direction(current_direction,signal)

该函数是根据当前方向和转向指令 signal 判断下回合的方向，是对判断当前方向的补充，将在后面的“at_home”部分中使用。

2.3 程序限制

我们的终极版代码已经成功解决了无故撞墙、撞带自杀、计算躲避路程不精确而被杀的问题，以及更为困难的 bug——行走诡异的弯曲路线并自杀的情况（见第三部分）。目前，我们唯一可能出现的 bug 是在对方将我们的全部纸片区域圈走之后我们的算法将会报错。但这种情况相当于直接判负，所以运行中并不会有 bug 出现。

3 实验结果

3.1 测试数据

实验环境说明：

硬件配置：

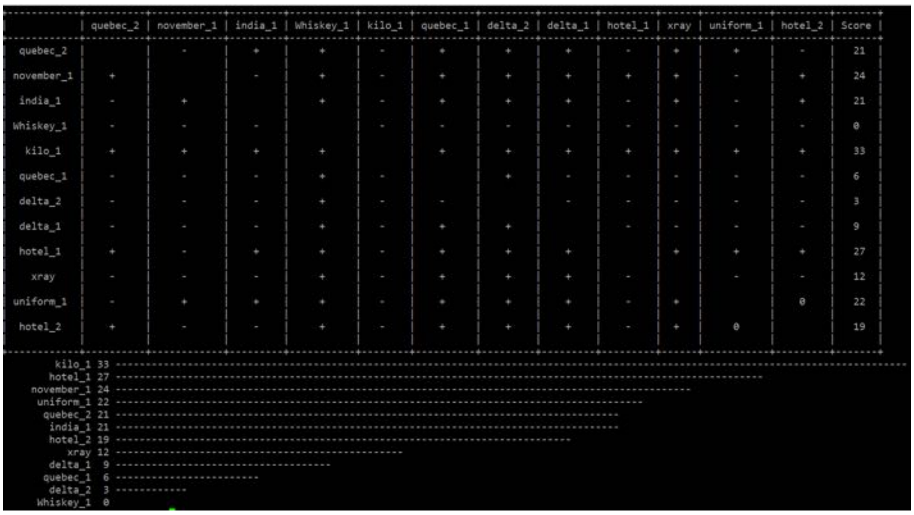
CPU：Intel® Core™ i5-5200U CPU @ 2.20GHz

内存：4.00 GB

操作系统：Windows 10，64 位操作系统

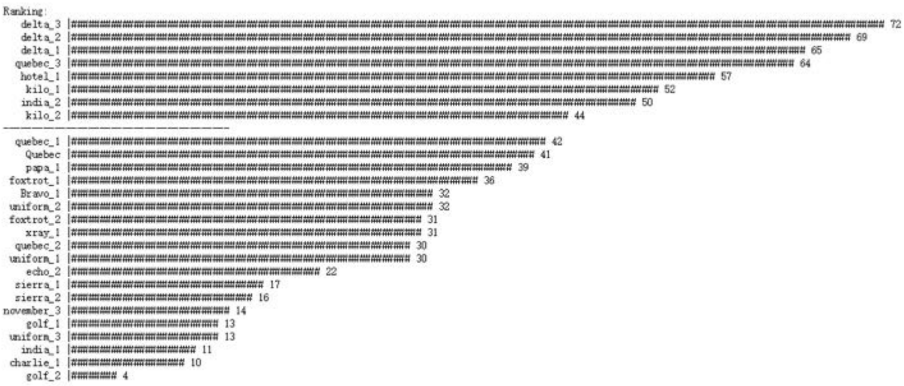
Python 版本：3.5

我们小组总共参加了三次热身赛，分别是 6 月 3 日，6 月 6 日和 6 月 9 日。其中，6 月 6 日的热身赛由于我们小组在命名文件格式上出现错误而未能得到结果反馈。下面是我们两次热身赛结果的截图：



	quebec_2	november_1	india_1	Whiskey_1	kilo_1	quebec_1	delta_2	delta_1	hotel_1	xray	uniform_1	hotel_2	Score
quebec_2		-	+	+	-	+	+	+	-	+	+	-	21
november_1	+		-	+	-	+	+	+	+	+	-	+	24
india_1	-	+		+	-	+	+	+	-	+	-	+	21
Whiskey_1	-	-	-		-	-	-	-	-	-	-	-	0
kilo_1	+	+	+	+		+	+	+	+	+	+	+	33
quebec_1	-	-	-	+	-		+	-	-	-	-	-	6
delta_2	-	-	-	+	-	-		-	-	-	-	-	3
delta_1	-	-	-	+	-	+	+		-	-	-	-	9
hotel_1	+	-	+	+	-	+	+	+		+	+	+	27
xray	-	-	-	+	-	+	+	+	-		-	-	12
uniform_1	-	+	+	+	-	+	+	+	-	+		0	22
hotel_2	+	-	-	+	-	+	+	+	-	+	0		19

6 月 3 日热身赛结果 (uniform_1)



在 6 月 3 日的第一次热身赛中，我们的代码为最原始的版本——基本圈地 + 简单逃跑策略。该版本代码量约为最终代码的一半，在没有受到威胁时会进行带有随机成分的圈地行为，但逻辑核心仍为纸片外选点 + 纸片边界选点的路径选择，在选择纸片外点时会选择距离该纸片上界的矩形边界到对应场地边界中点以内的点，然后选择一个可以返回纸片的边界点，接着使用 move 函数分别到达。整个环节由“at_home”、“leave_edge”和“go_home”三个状态切换构成。在热身赛开始前，我们终于将简单的探测敌人危险并返回的函数加上，并使用“danger_detect”的布尔值达到状态切换的功能。最终，该版本在参赛组中排名第四，成绩出人意料。

在 6 月 9 日的最后一次热身赛中，我们的代码已经加上了大部分的功能，包括优化后的危险探测返回函数、优化后的长距离攻击函数（防止对方进行绕场地的圈地）以及使用

二叉树的短距离博弈函数等。但最后的结果仅排名 14，和第一次热身赛相比有较大的退步，详细分析见后。

3.2 结果分析

1. 自我对战结果

1.1 初期的自我对战结果

在 6 月 2 日晚，为了选择第二天能够参加第一次热身赛的算法，我们小组内展开了三个算法之间的自我对战。结果，其中思路最为简单的、没有防守和攻击的圈地算法表现最佳，击败了带有简单防守算法的、有使用打分机制的代码和另一个主要依托机器学习模型的算法。因此，第一次的热身赛使用的代码是在该简单算法的基础上结合简单防守算法的部分构成的。

1.2 中期自我对战的结果

在 6 月 9 日我们初步完成了大部分的代码工作后，我们开始进行广泛的自我对战实验，当时小组内仍然分为了两组同学，一组在最初简单算法的基础上开始加上了可以远距离攻击的算法，另一组同学在原本的短距离博弈算法和另一版本的长距离的攻击算法的基础上加上了防守和圈地的算法。我们组内的代码进行了一系列的自我对战。结果发现带有短距离攻击的算法果然在近距离博弈时占据上风。同时我们也发现了一个非常明显的问题：我们在自我对战时常常出现超时的现象，很多时候如果不是其他方式结束了比赛我们的程序就会因超时而直接死亡。我们在当时并没有找到该问题的解决方法，于是只好在这样的结果上提交了最后一次热身赛的代码。

1.3 后期自我对战的结果

6 月 9 日的热身赛告诉了我们，超时问题是最为紧迫的问题。

我们首先对于程序中的各种 for 循环进行优化，减少了不必要的循环。同时，我们改进了广搜的方式。这两点使得我们的时间问题已经出现改进，在与他人进行友谊赛时，可以达到剩下 5 秒 -12 秒的状况。但仍然有组员反映，由于电脑配置不同，导致仍然会出现自我对战超时的现象。算法仍需要优化。

在 6 月 11 日晚，就在临近比赛的前夜，我们小组终于取得了惊人的突破。通过对于每个函数的耗时进行测量，我们发现了最为耗时的部分竟然就是我们每一回合都要进行的边

际寻找部分。与我们在最终代码中呈现的不同，当时我们的边界点寻找还是基于遍历整个场地的算法，这一算法看似合理，但却极度的耗时，也是导致我们的算法屡屡超时，以至于在 6 月 9 日热身赛中出现了大面积的超时死亡问题。在解决这一问题后，我们再次进行了多次自我对战的实验，结果用时被显著控制在了数秒之内，与之前至少都需要二十余秒的情况完全不同。这让我们有信心参加最终的比赛。

2. 热身赛结果分析

6 月 3 日热身赛分析

我们估计，在这一次的热身赛中我们的算法之所以能够取得较前的次序，主要还是因为当时参赛队伍少，且各算法都处于最初阶段，面临着许多报错、超时的情况。虽然我们的算法比较简单，但对手的算法可能还出于我们组内另外两种更“高级”算法的起步阶段，很有可能走出不经济的路线甚至撞带、撞墙自杀。因此，我们的算法已经表现的非常令人满意了；同时，我们也知道该结果对于最终对战的参考价值稍微。

6 月 9 日热身赛分析

在 6 月 9 日，我们的热身赛名次非常不尽如人意，而当我们仔细的观察结果时，会发现我们的 uniform_2 代码除了赢了 10 局之外还有接近 10 局是因为超时而判负。令人欣慰的是，正像之前所说，我们在正式比赛之前进行了全面的性能优化，热身赛之中的超时被彻底的解决。

3. 实战结果分析

在 6 月 12 日，我们的小组综合了组内两种算法，将基本圈地、危险探测返回、以及长距离攻击、使用二叉树的短距离博弈都完全的加入了一个代码中，该代码还有在前 5 回合进行积极策略（尽量靠近对手），接着 5 回合保守策略（尽量远离对手），然后根据前 10 局的结果选择一个更好的方式进行接下来的回合的性能。可以说，我们的最终代码结合了之前各个版本代码的最优功能。

在小组赛中，我们 5 战全胜进入 8 强。由于我们看到小组对手大多数并没有参加热身赛，所以有理由认为他们的代码可能还没有经过严密的调试，可能还有很大的改进空间，我们的代码不断在实战中磨练后确实显示了一定的优势。小组出现的结果并没有出乎我们的意料。

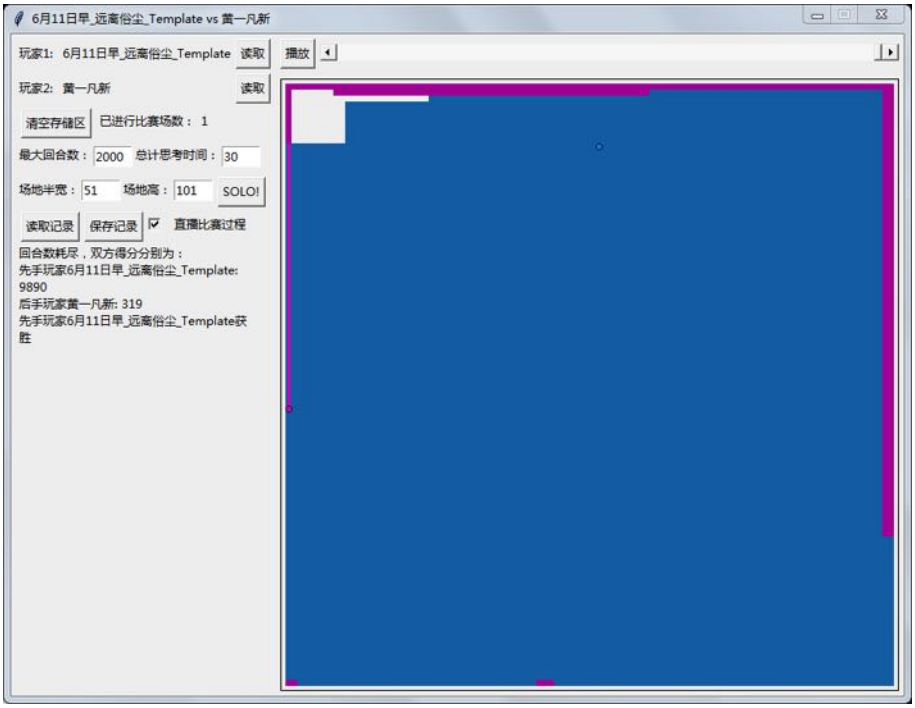
在四强赛中，我们对战 hotel 组，结果在开场时我们出现了 0：5 落后的情况。虽然

在 2：10 时我们的代码还非常争气的连续挽救 4 个赛点，但最终还是以 6：11 落败，无缘四强。我们分析，0：5 的局面主要是因为我最开始默认选择的积极性算法在与 hotel 的算法时有很大的劣势，而之后改用了更保守的圈地性质的算法后情况出现了改观，有能力与 hotel 的算法一搏，但由于之前劣势太大，难以挽回。最终八强的成绩虽然不够突出，整个过程还是体现了我们代码的特点的水平，也没有辜负我们的付出。

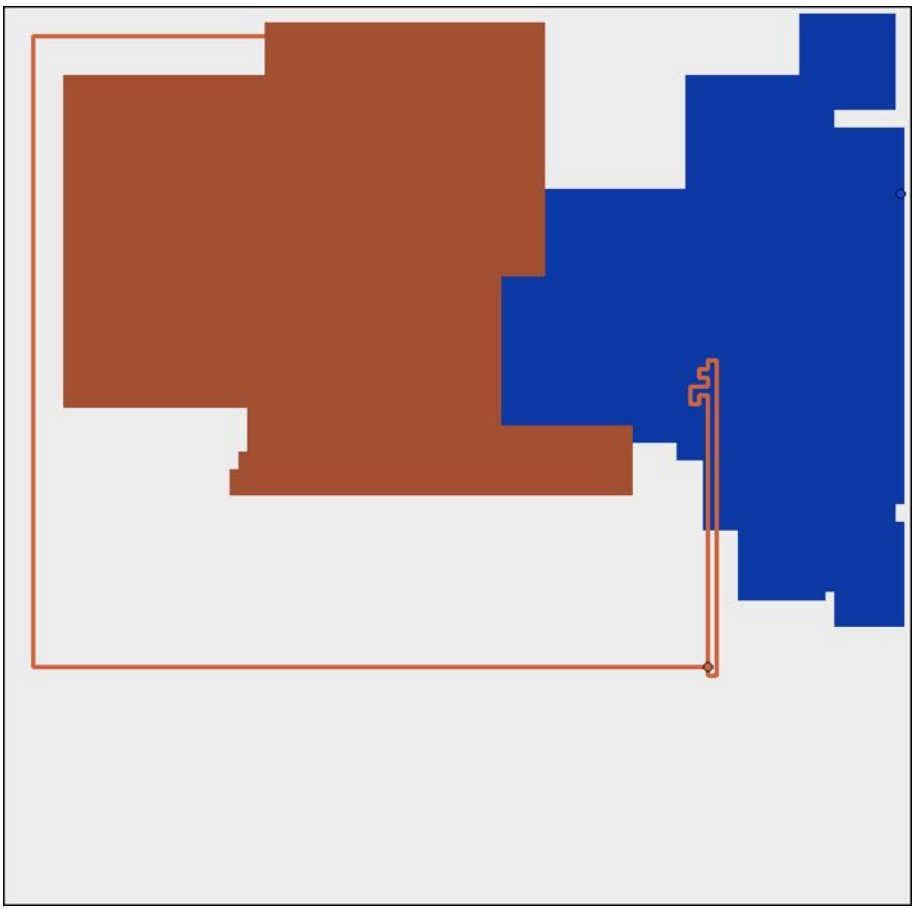
3.3 经典战局

本部分我们选取了两个对我们的最终代码产生了深远影响的战局，也是我们珍贵的回忆。

首先是我们在与环科的一组同学进行友谊赛时，发现了这样的现象：双方似乎都在紧随着对方的方向和路径行走，而我们的算法当时还没有很好的短距离攻击的方式，导致本来圈地效率很高的程序却出现了与敌人纠缠不休的情况。而这时，敌人又有一个恶毒的沿着外围圈地的算法，导致我们在开始连吃败仗。我们通过检查代码，发现原因出在我们在寻找纸片外的点进行正常圈地时，当前的算法中的打分会让纸带尽可能的往靠近敌人的方向前进。我们只是简单的更改了评分中对于敌人方向的加减号，结果就发生了很大的不同，圈地的效率变的非常之高，敌人往往无法近身，而且敌人的外沿圈地也大多被我们直接阻隔并且圈掉。最后基本都会出现了我们占据绝大多数区域的情况（如下图）。这一次的战局也启发了我们积极型算法和保守型算法的区别，导致我们最终的代码采用了前 5 局进攻后 5 局保守，之后两者中择优选择的设计。



另一个想呈现的战局反映了我们 debug 的过程。在最后一次热身赛前夜，我们的算法在对战中已经基本不会出现报错和撞墙的情况，但有一种非常“诡异”的路径却出现了数次，我们组员亲切地以“羊癫疯”的名字指代这样的自杀型行为。（见下图）



可以看到纸带在本来直线的道路上突然开始走出弯弯扭扭的形状，并在绕过一大圈之后最后撞带自杀。我们在之前认为这是一种小概率事件，而且形状确实过于奇异，感觉无法找到原因。

但是精益求精的数算作业要求我们不能放过任何一个 bug。于是通过在各个状态中设置不同的打印文字，我们在一格一格的定位路径后终于定位到了 bug 的位置——就在正常的“go_home”函数中。这一反常情况让我们突然意识到了一个重大的问题，我们的算法在得到可行的回到边界点后会一直前往该点，之后便切换到纸片内的状态。但我们没有考虑到敌人在我们的纸带回到纸片之前圈掉我们的可行边界点的情况。这导致纸带到达选定的边界后却发现没有回到纸片区域，只有一次又一次的尝试再次前往该点。而我们本身设置了最高层级的保护不撞带的规避方法，这便产生了各种奇怪的扭曲。我们很快进行了修订，而这样的自杀行为再也没有出现过。

4 实习过程总结

4.1 分工与合作

小组分工：

（1）函数设计

最初，我们小组兵分三路，基于不同的思路写出了三套可以独立运行的简单算法：

沈铂涵、姜畅：基于人工思维的圈地算法，圈地效率较高，保命能力与进攻能力较差

杨欣翰：基于人工思维及评分的算法，圈地效率不高，保命能力强，进攻能力较差

张俊锋、席子涵：基于机器学习的算法，但由于耗时过长及难以评估改进最后放弃了该思路

独立开发算法后，我们决定将三套算法中的优势集合在一起：

沈铂涵、姜畅：设计整体架构，构建状态切换逻辑，检查其他部分的问题，实战测试分析

杨欣翰：主攻探测危险及逃跑算法，优化修改圈地算法，撰写长距离进攻算法

张俊锋、席子涵：主攻短距离、长距离进攻算法

（2）对战分析

全组同学共同分析过往战局，提出了很多建设性的意见，一步步完成了代码的优化工作。

合作与交流：

主要采取线下共同讨论，分小组（个人）编写代码，在遇到瓶颈时再次共同讨论，相互质疑对方的算法是否可以优化，最终完成整个代码的设计优化工作。

会议记录：

（1）第一次会议

地点：小西门外 Zoocoffee

时间：6 月 1 日

参与人员：全体组员

主要事项：讨论整体思路，决定首先实现最简单的思路，再进行不断优化，会议分出了人工学习组与机器学习组。

（2）第二次会议

地点：光华楼四楼研讨室

时间：6 月 3 日

参与人员：全体组员

主要事项：人工学习组继续拆分成人工学习 1 组与人工学习 2 组，机器学习组调整思路改为完成专家系统，分小组完成代码，初步完成圈地算法与短距离攻击算法。

正在进行 debug 工作而陷入沉默的小组成员

（3）第三 -九次会议

地点：39 号楼宿舍

时间：6 月 4 日 -6 月 10 日每天傍晚至深夜

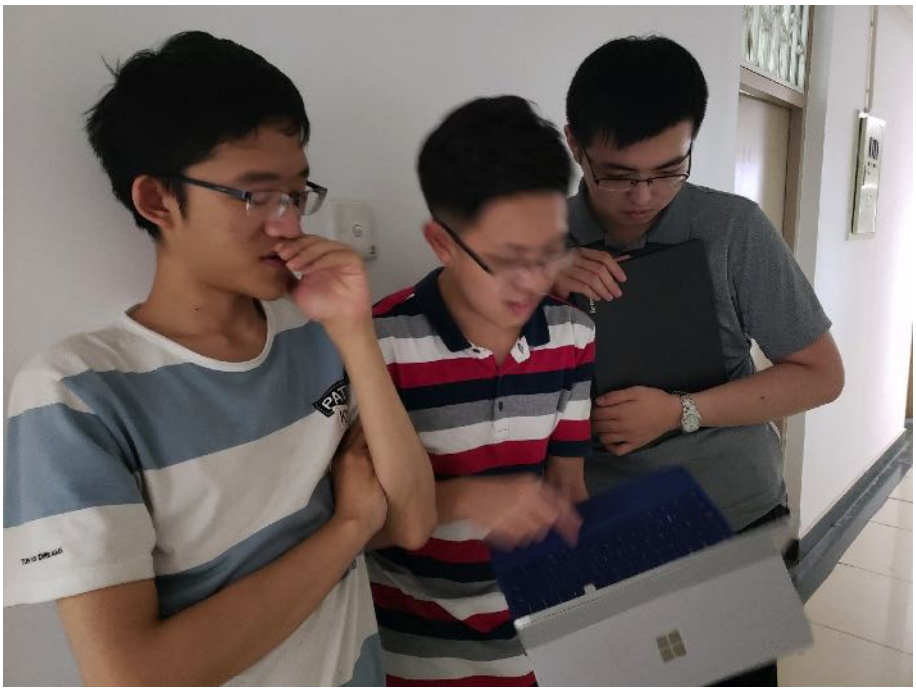
参与人员：全体组员

主要事项：前期主要是在写代码和 debug，后期主要是在进行三部分代码的合并与统一接口以及最最关键的时间优化问题。

从人工学习组中分离的杨欣翰同学在讲解他的单独完成的逃跑算法关键函数思路

深夜赶工

讨论算法改进方式



人工学习组讨论

机器学习组讨论

（4）第十次会议

地点：39 号楼宿舍

时间：6 月 11 日

参与人员：全体组员

主要事项：费尽周折找到了最耗时的函数并成功将其优化，完成了代码的最后一步。

4.2 经验与教训

总结下来整个过程中，小组做得比较好的有两方面：

（1）良好的团队分工：大家都比较靠谱，沟通方面没有花太多的成本，整体过程比较愉快；

（2）较早明确了方向并不断优化：贯彻“先写出来再说”的方针，将最开始的思路较好地实现了，并学以致用，改进了课上所学的二叉树结构与 BFS 算法，根据需要学习了 Alpha-Beta 剪枝，最终取得了不错的效果；

可以改进的方面：

可能因为没有与最后冠亚军队伍有过之前的热身，所以在圈地算法、切断对方回家路径等方面的考虑并没能做到最好。

4.3 建议与设想

建议：

（1）游戏选择：希望之后能够继续选择一些规则简单、趣味性强的游戏；

（2）竞赛：之后可以模范 dota2 等比赛的赛制，设立败者组，给予一定的复活名额，增强游戏的趣味性。

寄语：

（1）希望之后的同学能够享受不断改进算法的过程，并不简单将游戏设计好，而是有意识地注重将所学的数据结构与算法付诸实践应该会有更大的收获。

设想：

（1）整体感觉该游戏经过简化之后本质上是一个棋类游戏，并且得到最终才能有一个结果反馈（这一点比较像围棋），所以希望之后能够用强化学习等方式探索一下更加智能的AI。

5 致谢

首先感谢陈斌老师，技术组的陈天翔、张赖和同学，以及易超、陈旭、赵宇、冀锐、滕沅建、缪舜、张颢丹等助教大大们在指导我们选择思路、确定算法、编写代码、模拟测试、线上线下答疑等方方面面做出的辛苦工作！感谢油聚提醒我们的算法时间问题，这一问题的解决为我们最后的小组出现奠定了基础！感谢所有为比赛准备付出心血的工作人员！（包括奶死我们的主持人同学）感谢 39 楼 232、208、220、217 寝室的所有同学对于本小组常常讨论到凌晨 2 点的行为的包容和支持！感谢与我们小组进行友谊赛磨练代码的本班及环科班级小组！感谢摄影师翁昱昊！最后，感谢所有小组成员为我们亲爱的纸带所付出的辛勤工作！

6 参考文献

[1]Paper.io play online. http://paper-io.com/

[2]Paper.io - Need help? – Armedunity. https://armedunity.com/topic/14628-paperio/

[3]Paper.io – Wikipedia. https://en.wikipedia.org/wiki/Paper.io

[4]Breadth-first search – Wikipedia. https://en.wikipedia.org/wiki/Breadth-first_search

[5]Breadth First Search Tutorials & Notes | Algorithms | HackerEarth. https://www.hackerearth.com/zh/practice/algorithms/graphs/breadth-first-search/tutorial/

[6]Machine learning – Wikipedia. https://en.wikipedia.org/wiki/Machine_learning

[7]Machine Learning: Factor Selection – Hentsu. https://hentsu.com/machine-learning-factor-selectio

第二十一章　N17_Victor 报告

王馨 *、蒋锡辰、葛楠、刘佳明

摘要：本文反映了我组在本次数算大作业中的主要工作和经历，从问题分析出发，细致地阐述了我组算法的原理，即实验的过程中一开始采用了 S 形圈地的方法，这样可以迅速躲回家，但是后来发现速度较慢，并且有很大风险会被击杀，所以最终采用了按一定步幅向外拓展领地边缘的方法，在互动策略上主要是躲避、扩大自己的面积，如果自己的领地被侵入了，首先修复失去的领地，然后按照相同的策略向外拓展。我们采用队列数据结构储存即将行走的路径，在走路函数中利用了递归算法，最终实现的算法速度较快，能够高效地拓展己方面积。文章还从数据结构、函数等方面展示了实现该算法的程序，并附上了正式比赛中的结果。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感想与思考，并对本次实习进行了总结与反思，为未来的实习大作业与课程开展提出了展望。

关键字：S 形圈地、列表、队列、递归、学习、搜索。

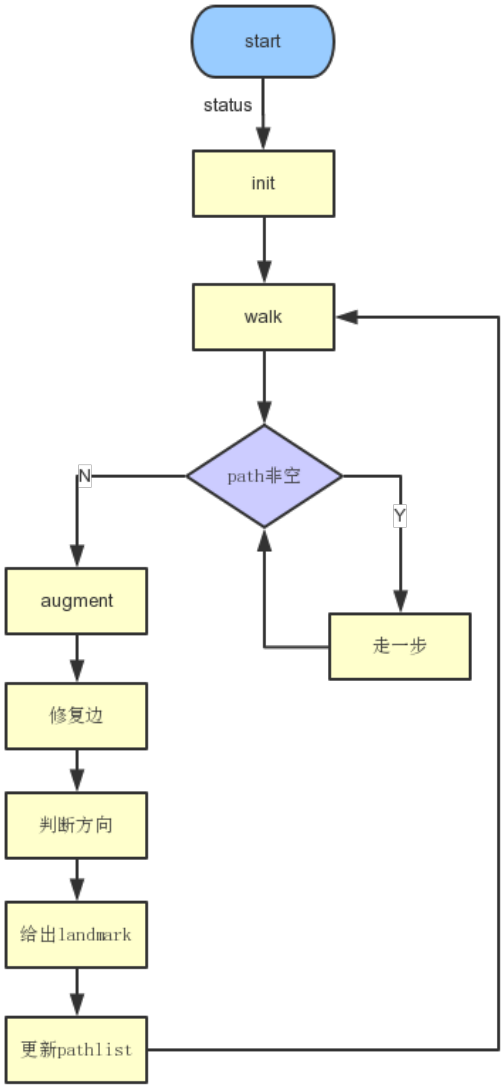
1 算法思想

1.1 总体思路

纸带圈地游戏有两个主要的策略：圈地策略和躲避/攻击策略，各组花了大量时间优化的也是回家/圈地的算法。我们在实验的过程中一开始采用了 S 形圈地的方法，这样可以迅速躲回家，但是后来发现速度较慢，并且有很大风险会被击杀，所以最终采用了按一定步幅向外拓展领地边缘的方法，在互动策略上主要是躲避、扩大自己的面积，如果自己的领

地被侵入了，首先修复失去的领地，然后按照相同的策略向外拓展。我们采用队列数据结构储存即将行走的路径，在走路函数中利用了递归算法，最终实现的算法速度较快，能够高效地拓展己方面积。

1.2 算法流程图



1.3 算法运行时间复杂度分析

时间复杂度为 O(d)，其中 d 为边的数量。运行时间构成主力是扩增边 augment 函数，其中调用了走路 walk 函数，走路的次数等于边的条数，每条边需要计算 path、记录 landmark。

其次，在 path 计算的时候采用了递归函数，输入当前位置、当前方向、终点位置，返回行走的路径和最终路径，但是递归层数比较浅，最多只需要调用三次。综合来看，时间复杂度就是常数级的 O(d)。

2 程序代码说明

2.1 数据结构说明

主要运用了线性数据结构。Path 是字符的一维列表，landmarks 记录了所有走过的点，是一个二维列表。

队列 edge_list 运用了队列 ADT Queue，记录了下面将要扩增的边，按照先进先出的方式取出一条边进行扩增。

2.2 函数说明

（1）play(stat, storage)

实现两个功能：读取数据，调用 move 函数。

（2）load(stat, storage)

这是规定的主函数，全部在 load 函数内完成函数声明和其它函数定义（来自于示例）。

Move(field, me, storage)

根据当前状态选择函数执行，一开始状态初始为 init，之后一直是 move。

Init(field, me, storage)

Init 是初始状态调用的函数，由四个部分组成。首先，以棋盘宽度的一半为标准，判

断起点位置偏左还是偏右。其次，画出一个正方形，并记录关键点，如果起始位置偏左，那么就画一个偏右的 9×9 的正方形，反之亦然；在走完四条边以后，还要把四条边加入队列，作为下一步扩增的对象。然后，init 函数的任务是根据 landmark 中记录的关键点算出路径，给出两点坐标和当前方向，调用 path 函数返回路径和最终方向。最后，调用 walk 函数（详情见下一模块）。

Walk(field, me storage)

判断当前路径是否走完，如果走完了就调用增广边函数 augment，如果没有走完就走一步。

Walk_landmarks(field, me, storage, landmark, directions)

输入一个关键点 list，walk_landmarks 函数会把状态设定为 walk，调用 path 函数给出路径，并且调用 walk 函数走一步，这个函数之所以单独写出来，是因为之后的环节会经常用到。

Repair_edge1(filed, me, storage, edge)

第一种修复边的函数对应于即将扩增的边的起点已经被敌人侵占，那么首先修复起点，然后再进行扩增。扩增的方法是从当前点出发，绕着被侵占的部分走一圈。分三种情况讨论：如果当前位置和侵占边的起点 x 坐标一样，那么以起点终点的连线为竖边，走一个宽度为 1 的矩形；如果当前位置和侵占边的起点的 y 坐标一样，那么以起点终点的连线为横边，走一个宽度为 1 的矩形；如果当前位置和侵占边起点的 x 坐标 y 坐标都不一样，那么以起点终点为对角线上的两个点，走一个矩形。路径判断完成后，把关键点加入 landmark 的列表，然后调用 walk_landmarks 函数计算相应的路径并且走路。

Repair_edge2(filed, me, storage, edge)

修复完起点以后，我们还需要修复待增广边的终点，从而完成对整条边的修复。具体的修复路径分了四种情况讨论，在此仅以第一种情况举例：首先根据修复边的起点终点位置判断这条边的方向，如果是与 x 轴平行，那么接下来判断敌人在这条边的哪一方，如果待修复边的北边被敌人占领，那么就往南边修复，修复矩形的宽度为 augment_width（我们设定为 10）。其它情况与之类似。判断修复路径后，把关键点加入 landmark 的列表，然后调用 walk_landmarks 函数计算相应的路径并且走路。

Augment(field, me, storage)

首先，从当前点走到扩增边的起点；判断是否安全，如果不安全，就把当前边丢回队列，返回初始位置；如果安全，就扩增当前边。

扩增的方法需要判断边的方向和扩增时前进的方向，也是分了四种情况讨论，判断方向后，把对应的两条边加入 edge_list 队列，把关键点加入 landmark 列表。还需要进行一步特殊判断，如果扩增边只需要新增两条边，那么就提前结束，如下图所示的这种情况，黄色是需要新增的两条边。

Path(x1, y1, x2, y2, direction)

Path 函数是最核心的函数之一，作用是给出两点坐标与坐标方向，返回路径与最终方向，我们分了很多情况讨论，详情请见代码，但是核心思想是判断起点到终点的向量方向和当前方向的单位向量方向，先确定转向，然后确定路径，递归调用自身，直到当前位置与终点重合。

2.3 程序限制

躲避算法仍然存在不足，现在的算法是在扩增边的时候计算自己与敌人的距离，如果敌人太近，就把当前要扩增的边放回队列，取队列里的下一条边进行扩展，但是如果扩增边的起点和敌人的距离在安全距离之外，在扩增过程中没有每一步都判断距离，因此可能会在扩增边的过程中被敌人击杀。

3 实验结果

3.1 测试数据

实验环境说明：

硬件配置：（CPU/内存）Intel64 Family 6 Model 142 Stepping 10 GenuineIntel 1500 Mhz 内存 8G

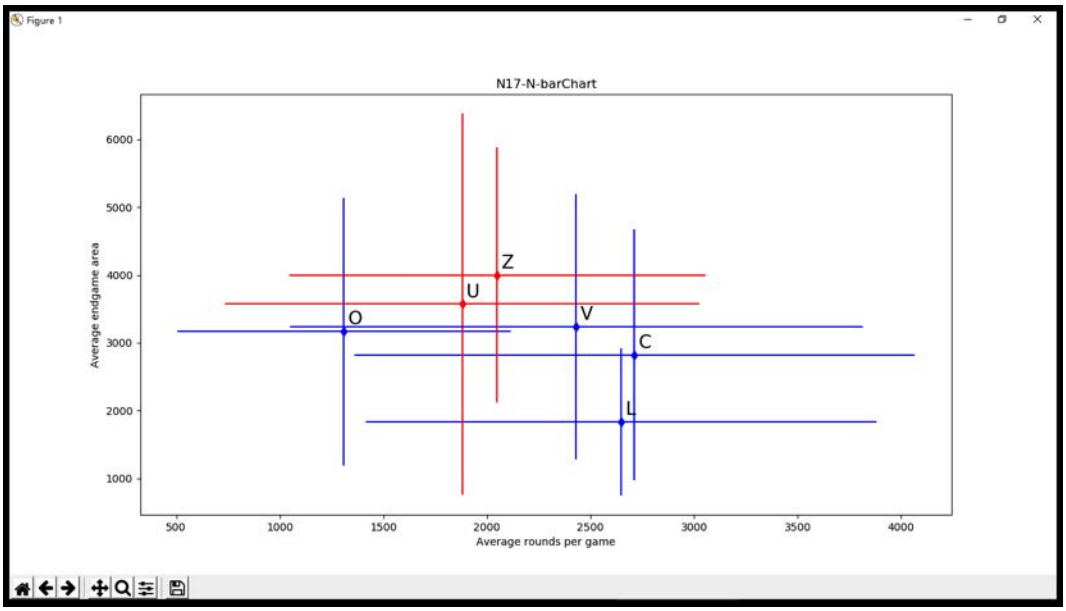
操作系统：（名称/版本）Windows10

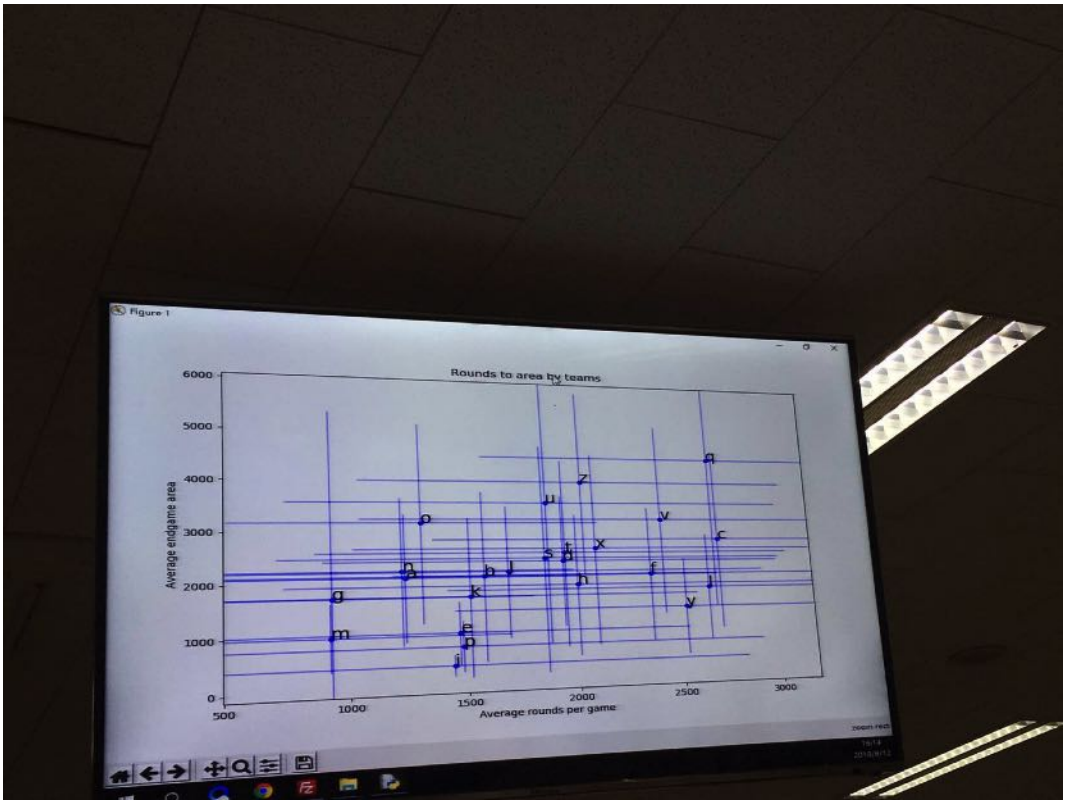
Python 版本：（版本号）Python 3.6.4

测试方法主要是和上一个版本的自己对打，从比赛回放录像上分析出线的纰漏，因为时间安排紧张和个人原因（拖延症?），我们没有来得及参加热身赛，最后很遗憾地在 N17-S 组差一名小组出线。

3.2 结果分析

在这一部分我们展示比赛的数据（上图：本区内所有小组情况，下图：N17 所有小组情况）。实际对战中，算法运行跟我们预想一样，还是比较快的，圈地面积略高于 O 组，离出线的两个小组还差的比较远，在赛后采访中，发现出线的两个小组都加入了主动攻击的策略，可能也是我们的一个致命缺陷。





我们还试验了一个比较奇葩的版本，一开始就首先圈一半领地的面积，在和自己对战的时候这个策略可以轻松取胜，但是担心其它组一开始就采用攻击策略，就没有提交对战。

4 实习过程总结

4.1 分工与合作

Victor 小组的全部成员是来自不同院系和专业的毕业班同学，在本次纸带圈地算法的实习大作业期间都面临着毕业论文和答辩的压力。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：所有四位同学（王馨、蒋锡辰、葛楠、刘佳明）负责完成代码，这其中刘佳明同学主要负责多次人力尝试游戏同时对算法提供思路，为算法提供模块化的功能与构架分解、王馨和

蒋锡辰则根据全组商议的算法的构架和流程各自写出一份代码，进行对照与参考，继而确定后续的代码组合与改良。葛楠同学则主要负责代码的整合与改写、在代码的优化与改良部分负责距离计算与战术调整，即能够合理对于对方的运动情况和与自己的距离进行更好的调整。在后续部分，刘佳明主要负责实验结果和现场比赛的记录，同时为进行了报告的构架，其余三位同学共同负责小组报告的完成，王馨负责算法思想的具体分析与归纳，同时探讨了算法的时间复杂度与效率，葛楠负责程序代码的各种说明解读与分析，结合实际情况梳理了代码编写思路和应战策略分析，蒋锡辰负责实习过程、经验教训、建议与设想的总结，同时完成本文最终的整合与修订工作。

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（6 月 2 日、6 月 5 日、6 月 10-11 日），并且由于实习小组人数较少，在代码编写和报告完成方面都是全员共同行动，同时划分了两个 2 人小组进行模块化的任务操作，两个 2 人小组内部也时常以约饭等形式进行过数次线下的讨论。

第一次线下组会是在二教三层的讨论区，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和 Github 上所提供给同学们的各项代码构架和资源道具，确定了小组自主的代码编写之外已经固定的条件，也指出了目前存在的未确定因素。其次，组会根据人为的玩家试验探讨了圈地与抵抗进攻的策略，确定下来小组算法的核心思想与基本流程，是尽可能做到走位都不能太远离领地的阶梯式纸带圈地方法，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，确定了具体的时间规划，明确了代码完成时间、测试时间以及参与竞赛的时间，同时安排了线上的讨论、每个人的任务与质量要求与后面组会商议的具体工作等。以下为第一次组会的照片。

图 4.1 第一次组会（2018 年 6 月 2 日晚，二教三层讨论区）

第二次线下组会是在理教四层的讨论区。此时王馨和蒋锡辰则根据全组商议的算法的构架和流程以及各自写出一份代码的初稿（粗稿，一部分还是伪代码），两人分别为组员进行了讲解，同时进行对照与参考，继而组员讨论，确定可以改进的方面（如距离计算与策略选取）和应该增加的功能（如绕路行为）。随后葛楠同学则主要在代码的优与战术调整方面提出了自己的设想，同时明确了后期的任务。同时组会明确了每个人最近都要利用时间查找一些机器学习的有效知识，放在群上供大家共同参考，有利于算法的提高与最终的成型。以下为第二次组会的照片。

图 4.2 第二次组会（2018 年 6 月 5 日晚，理教四层讨论区）

第三次线下组会是即将比赛的前两天（此时大家的毕业答辩刚刚完成），由王馨、蒋锡辰和葛楠主导的代码编写团队根据刘佳明同学进一步积累的经验与分析对于代码进行了最后的调整与整合。同时由组长领导，各组员再次梳理了了算法的整体思路和应战策略。同时本次组会安排了后续报告撰写的相关事宜，将报告所需的数据资料的记录、需要各组员协助解决的问题以及报告的分工进行了规划与安排，通过每个组员之间的交流以保证报告的顺利撰写。最后组会对于两天后的比赛进行了一些设想，以自拍形式振奋（jilu）了忙到凌晨极其疲劳的我们。以下为第三次组会的照片。

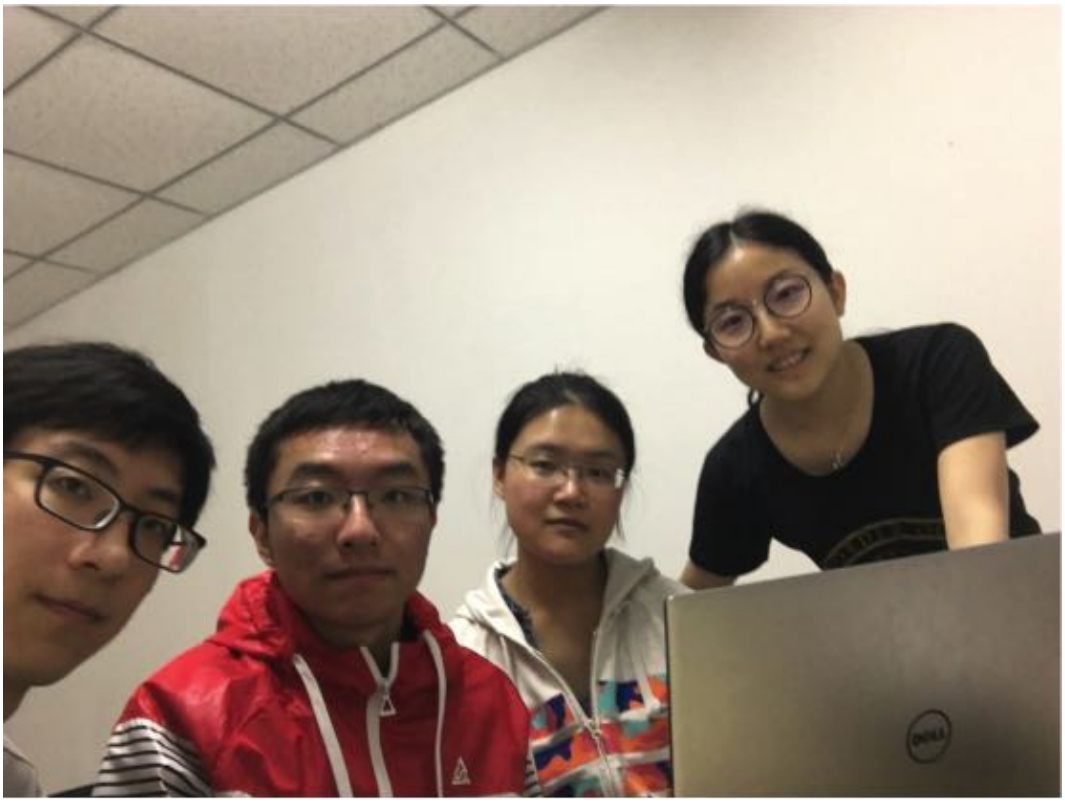


图 4.3 第三次组会（2018 年 6 月 10 日晚至 6 月 11 日凌晨，元培宿舍地下自习区）

4.2 经验与教训

在完成本次实习的过程中，我们首先根据我们“肉身”在微信小程序上的玩法确定了一些能够在圈地的同时避难的技能，从我们的纸袋圈地的逐步拓展模式就可以看出，这样能够在圈地的同时离自己的大本营不至于过远而被对方干掉。我们的纸袋圈地模式其实就是小组成员自己尝试游戏的经验写照，所以自己的亲身体验对于实习的完成是十分重要的。与微信小程序的好友对战几局，不仅可以发现自己的薄弱之处所在，同时也可以汲取其他人的一些思路。机器的智能从一开始是十分依赖人类自己的智能，而不能凭空只依靠距离与面积计算和算法来呈现出最后的对战策略。所以我们的尝试确实为后续的算法提供了很好的参考。包括在后续的代码调整中，我们也是通过大量组内的人类游戏对战方式，探讨不同的对方应答状况下的流程应对，同时看自己的走位会诱发人类怎样的思考与布局。我们可以猜想的是，其小组肯定也是参考了人类对战的模式进行算法上的开发与流程上的构建，于是通过人类的对战我们可以更好的揣摩不同的形势下的对方可能采取的应战举措，

来进一步规范与调整对战的策略，来保证本小组不会被 KO，同时尽量能够圈住更多的地。

但是由以上也可以发现，我们小组首要考虑的是不被 KO，其次是保证圈地面积，而没有过多思考如何去进攻其他人。所以可以说我们的思路比较稳重，但是过于稳重有太保守了。在现场比赛的后期也就是可视化的对战阶段，我们发现在一些局势下我们确实处于劣势，在行为上太保守反而难以应对激进型的算法，不仅无法截杀其他组，在这种情况下反而圈地面积也不占优势了。所以思路的过于稳重与保守也是本次实习算法的一个最大的问题，也是我们从中得到的一个教训。在最初的想法上一定要放的开一些，充分利用好规则，在肉身对战尝试时还应该换几种方式，不一定每次都要保命，多尝试一些总是有好处的，这样到了后续思维也会宽广许多。

具体到小组本身的任务组织方面，我们每次组会都会进行详细的商议与讨论，确保当场把预计要解决的问题化解，同时规划好未来的工作，这是一个很好的方面，不至于太拖沓。但是组会商议事情过多，时间过长，到了最后反而会导致效率低下，大家的思路受到了禁锢。而我们的线上交流则相对较为简单，其实很多的思路或者参考资料都可以在线上进行共享，不用带到组会上叙述太多，同时可以提升效率，本质上还节约了时间。所以者也是我们的一些教训。

还要说的一点是，由于时间问题我们组错过了热身赛，加之又是大四同学组成的小组，与班上其他小组几乎没有什么交流，对于他们采取的算法和思路也知之甚少。所以这就造成了我们丧失了最后关头调整代码的机会，终究没能发现和解决圈地算法太过于保守的问题，也没有特别合理的策略来应对一些特殊的局势，以至于在本区比赛中只差一名就出线了，真是一件颇为遗憾的事情。于是这件事情也告诉我们，一定不嫩闭门造车，不要害羞与其他组之间交流，能够博采众长，同时又能参与热身赛合理地分析自己的不足与优势，方能在最后取得不错的成绩。综上，多交流，多尝试，大胆去想去做，方能建立一支有特色，有实力的圈地纸带军团。

4.3 建议与设想

此次学期时间较为紧张，由于放假次数较多，相比往年的计划，其在比赛的准备与预热时间上都有压缩，因而在流程上不免有一些捉襟见肘的感觉。加上我们都是大四狗，杂事比较多，错过了热身赛的时间节点，对于在算法与技术调整上也没有一些参考的机会。对于在未来的比赛中，可以提早开始布局，使得各位同学有足够的时间安排组队和后续的

任务组织与实践工作。以上的安排可以不用拘泥于课程的计划，不一定要等到讲完图及算法以后才开始，可以为大家提供更为广阔的思维发散空间与自学提升的机会。事实上，放眼到比赛本身，其准备工作中很大很大一部分都是靠着组员的协作与配合，依靠不断的自主学习与查阅相关资料逐步完善，可见其中自主的成分是占有绝大多数的，很多的知识也并非课堂上学过后可以直接应用，还需要进行不断的调整与补充。所以我组成员均认为课堂计划并不是制约比赛时间进度安排的主要因素，可以适当提起开始。加之陈斌老师魅力确实很足很足，拥有强大的人格魅力，在课程设计和课程组织上也别出心裁，非常独到，吸引了北大很多学生的选课和参与。可以明显预见的是，未来的课程体量会更加庞大，参与的学生来自不同的院系，组成上也更加多元化，所以在组队与协调上可能需要花费的时间也会更多。从我们组的情况来看，小组成员的寻找本身就耗时较长，尤其是对于毕业班的同学，对于全班绝大多数的同学都是陌生的感觉，所以在成员的聚合上确实也不太容易。我们也是在课程的微信群上逐渐寻觅队友走到一起的。

其次，从我们组的讨论经历来看，前期的磨合期也需要占据大量的时间，这可能是由于成员之间一开始都不甚熟悉，在分工与思路上都有各自的见解，对于问题的理解上在最初也不是特别统一。当然，这也与成员之间较为陌生，一开始无法创造较为自由的讨论氛围也不无关系，所以大家最开始讨论的效率是比较低的，随着组队的时间进度往后发展，效率才有提升。到了临近比赛的时候，我们组成员都一致认为还有很大的调整与提升的空间，但是在时间上确实不允许了，所以不仅造成了较为匆忙的任务统筹，也让我们觉得还有缺憾导致的一些遗憾。还有一点想要补充的是，今年比赛的安排与准备的时间恰好与我们毕业论文最后的攻坚与成稿时间冲突了，所以那一段时间组内的每个成员都异常繁忙，感觉很多的事项都统筹不过来，也造成了较大的心理压力。所以比赛的时间安排对于毕业生来说还是一个非常关键的因素，如果早一些开始准备，可能在时间上的余地也会宽松很多。

放眼到比赛现场本身，今年的比赛对于往年的计划感觉场次要少一些，但是非常感谢助教和技术团队开发的平台让整个比赛过程较为高效。比赛现场穿插的很多彩蛋和一些有趣的缓解也很大地活跃了现场氛围，可以说是一门深受学生喜爱的大课开设中的教科书式的典范。以上诸多举措，不仅提升了所有选课学生对于课程本身的喜爱，也映射到使得学生认为编写代码去解决一个生活中有意思的问题是一件非常有意义且有趣的事情。此外，比赛的多元化与趣味化组织也带来了很强的边际效益，活跃了学生之间的交流，为大家创造一个非常良好的互动氛围，这对于新时代的阳光大学生是一件非常好的事情。但是为了

让日后的现场比赛更加顺利办的更加红火，我们组成员也汇集了一些个人的建议以供参考。首先是在正式的现场比赛开始前期，在组长群里发生了一些不愉快的争执，大家争执的重心其实是在于比赛的公平性与分组问题。事实上本次比赛已经在安排上较为随机，而且组队都是根据大家自主的意愿进行组合，所以感觉在组队的方案上改进的余地较少。感觉可以提升的方面在于在安排前一定要将各项情况说清楚，对于比赛的流程一定要通知各位成员仔细的阅读。

说到这里，需要额外补充的一句是，群上的水群现象确实较为严重了一些，导致了很多重要消息都难以及时看到，有时候一刷就过去了。而且对于时间安排较为紧张的同学和一些不常使用微信或者不喜欢水群的同学，就会在消息的接受上感到更加麻烦，甚至出现了一定的反感情绪。所以适当的水群是一件活跃关系的事情，这一点对于课堂氛围的构建也是一件不错的事情。但是大水使得大量的无关消息像洪水一样涌现，而有一定关键信息的消息在其中占比则太小太小，无疑为群上同学接受有效消息增添了巨大的困难，同时也会因此耽误了很多学习的时间。其实本学期有关课程和比赛的通知在群上都有说明，可是仍有一部分的同学反映自己不知道，还在群上额外询问，或者认为通知不清而有一些情绪。其实水群太过在其中扮演了很多的负面角色，因此未来在建立课程群的一开始一定要适当强调这是一个课程群，要在一定程度上对于水群的行为进行限制。同时在群聊的过程中，也要适当根据实际情况进行提醒与矫正，使得大家既能通过群聊解决各自的困惑，同时可以适当活跃氛围。

其次，我们注意到在现场比赛的时候，有一个区的比赛进程中出现了长时间卡顿的现象，这应该是参与比赛的组中有代码出现死循环所导致的。但是对战系统超时的宽限似乎过长，导致了该组比赛无法正常顺利进行，以至于用临时的比分当作最终比分，虽然似乎与最终的预期结果相差不大，但是还是有不妥当之初所在。此外，我们设想能否采取一个操作平台满足所有小组之间的对战，这样或许可以解决大家探讨较多的公平性的问题，防止出现各分区之间差异过大的问题。

对于未来参与此课程的实习小组，建议一定要从一早开始就尽量提高效率。根据“馒头定律”，人一天吃 100 个馒头是不可能完成的，但是 100 天吃 100 个却轻而易举。所以要学会将任务适当合理地分配，组会不用太多，可以在所建立的微信群上经常讨论或者共享一些有用的资料、个人的思路与设计方案。一定要为最后的比赛留有调整的时间，这样才能根据局势合理进行调整，强化优点，调整不足，不至于到最后留有遗憾。另外，这门课程真的非常值得，如果有所付出，细细品味，确实可以收获很多的学业和非学业的知识

与能力。

5 致谢

感谢陈斌老师的指导及在微信群上对于小组成员所提问题的解答。感谢助教团队和技术组团队对于本次实习和比赛的支持与付出。以及感谢现场比赛时所有小组的对于活跃氛围的配合，感谢主持人和解说者的付出，此外还需要感谢大大卷和某人抢红包的手速，以此让没有顺利入围的我们有了消遣时光的绝好伴侣。本次实习和本门课程都是一次非常愉快且神奇的体验，让小组的所有成员感受带了不一样的课堂内外，真的可以作为离开北大前非常难忘的一次体验，再次感谢本门课程，感谢博雅塔下未名湖畔的哪些日月流走与风中余香。

6 参考文献

http://gis4g.pku.edu.cn/course/pythonds

www.paper-io.com

https://gitee.com/chbpku/paper.io.sessdsa

https://blog.csdn.net/lihongxun945/article/details/50668253

https://handong1587.github.io/deep_learning/2015/10/09/nlp.html

Miller B N, Ranum D L. Problem Solving with Algorithms and Data Structures Using Python Second Edition [M]. Franklin, Beedle & Associates, 2011.

第二十二章　N17_X-ray 报告

范家豪 *　陈峻松　陈思如　潘登　王天贺　张毓昕

摘要：本组代码主要以人的思路策略为导向，通过马尔科夫链等一系列抽象算法结构将之实现。代码的核心结构是根据自己和敌人回城以及攻击的最短距离和路径参数做出决策，在普通圈地、逃跑、攻击、近战四个状态间进行转换。前者由 A* 等启发式算法得到，其他功能通过一些封装的函数实现。我们在测试阶段与 17 级多个小组进行了友谊赛并不断改进代码性能，正式比赛中不幸遭遇了 N17 组的冠军和季军，最终惜败，屈居小组第三。相信如果能够出线，我们定会走得更远。

在本报告中，我们将进一步分析算法思路、讨论经典战局、并给出小组分工和讨论等，配合我们的代码将之前工作完整呈现。我小队在接下来还会继续完善代码，获得更好的效果。

关键字：A* 算法 ADT graph 启发式算法马尔科夫链列表

1 算法思想

1.1 总体思路

我们队的比赛思路较为简明直接。首先，整个代码中，有四个参数对决策至关重要，它们应当首先被计算出来。分别是：1 我方回城的最短距离及路径；2 对方回城的最短距离；3 我方击杀对方的最短距离及路径；4 对方击杀我方的最短距离。

我们最早采用 A* 算法计算这四个参数，但由于耗时过长超出比赛限制时间，于是改用了启发式算法，牺牲了一定的精确度。但在大多数情况下，我们可以精确计算这四个参

数。

当然光有这四个参数是不够的，我们还需要几个函数去执行这些函数的功能，包括沿着最短路径回城，沿着最优路径进攻对方纸带。之后比较重要的是圈地函数部分。所谓圈地函数是指在完全忽略对方造成的可能危险的情况下，尽可能高效地圈地的一组函数。它们一方面要保证用有限的边长圈出较大的地，还要保证在领地内时能快速前往空地或对方领地较大的区域以保证圈地活动高效有序进行。对于圈地本身来说，我们尝试不同长宽比的矩形，以确定较为高效的参数。在领地内寻找空地采用的也是启发式算法，我们的纸带头部朝着东南西北四个方向都一步一步去探索，一旦发现某个方向是空地，就向那个方向前去。事实证明这种判断方式是较为有效的。

在热身赛的实战中，我们发现一个重要的问题：击杀总是在双方领地边缘发生，即不小心迈入对方领地而被直接击杀。因此，我们添加了一个模块专门处理当双方纸卷都在领地边缘且非常靠近的情况，该模块主要是根据人类经验，由各种可能的条件判断和情况判断组成。为了防止自杀，我们设计了用来检验是否会自杀的函数。自杀主要被分成两种情况：1 在领地外自杀，比如撞我方纸带或撞墙，这时会调用第一阶段计算出的回城路线，该路线根据其计算的原理是绝对安全的。特殊的情况即在对方领地撞击对方纸卷，这主要依靠我们的近战函数来排除。2 在自己领地内撞墙，我们在圈地函数内部添加了一些应对这种情况的机制。总之，在这些机制的保证下，自杀的可能基本被避免了。

之后就是将上述最主要的两个模块以及近战模块通过核心逻辑判断组织在一起。我们一共定义了四个模式，核心逻辑是：如果双方离得较远，完全没有威胁则进入一般状态（flag0）；如果我方有条件击杀对方，则进入进攻状态（flag1）走最短攻击路径去击杀；如果对方击杀我方的距离相较我方回城距离造成了威胁则进入逃跑状态（flag2），直接回城；如果双发都在领地边缘且离得很近，则启动专门的近战模块，针对具体情况进行应对；如果在进攻中突发威胁则调用最短路径回城；回城后危险结束重新开始圈地。最后为了增强后期和对方竞争圈地的能力，我们在圈地模块提添加了一项功能：相较空地，优先朝向对方的领地前进，即将吞食对方领地作为第一选择。实战结果证明这一判断是很有效的。

1.2 算法流程图

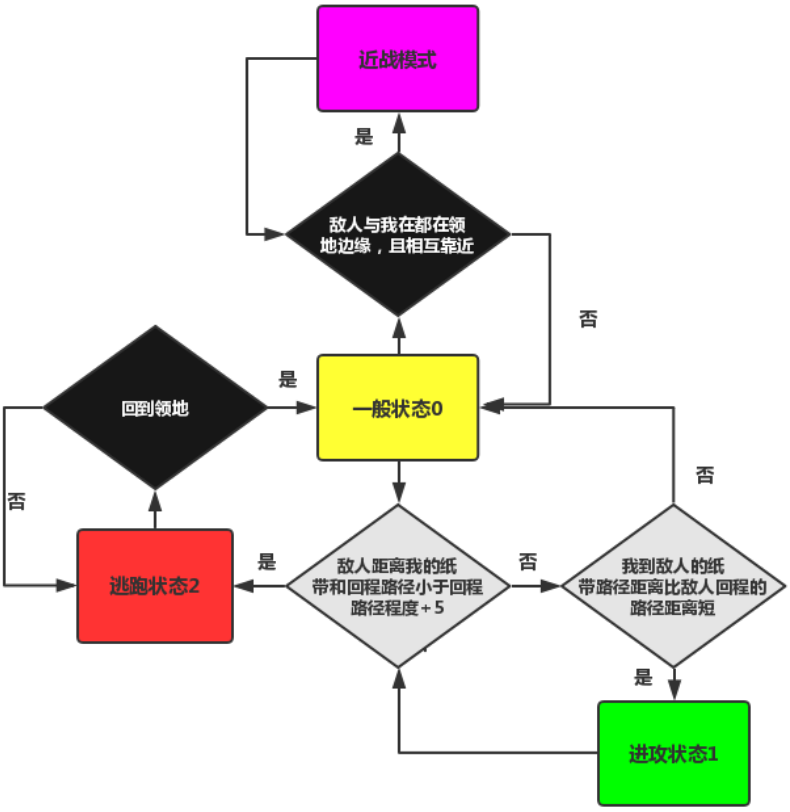


图 1 三个不同状态（状态 0,1,2）转换关系图

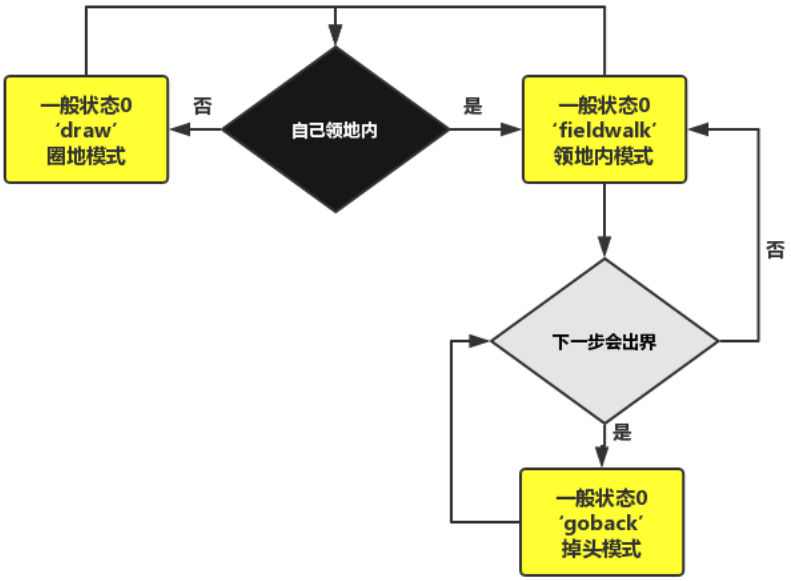


图 2 一般状态（flag0）下三个模式的转换

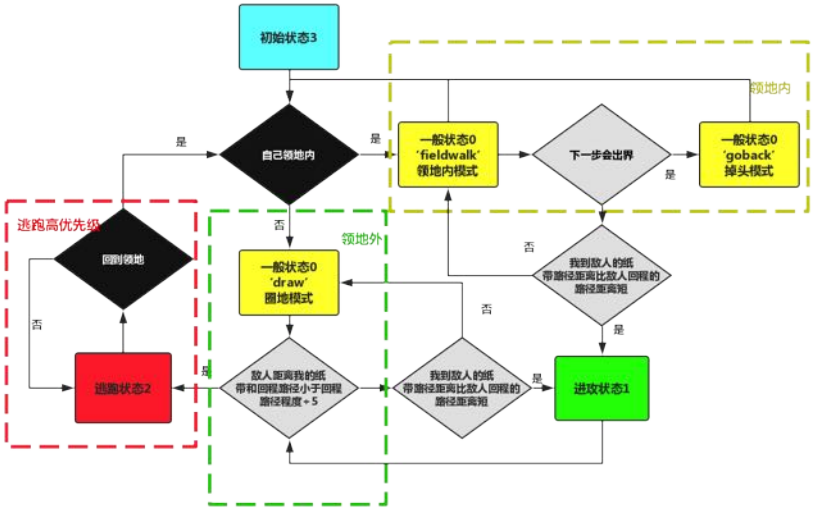


图 3 X-Ray 小组代码总运行流程图

1.3 算法运行时间复杂度分析

主要影响我们组代码运行速度的因素是双方纸带的长度及双方圈地的面积，根据计时实验的结果，对于最终代码来说，后者的影响更大。核心逻辑判断部分自然是 O(1) 的时间复杂度，且在绝对时间的消耗上也可以忽略，圈地部分有一个启发式的算方向的部分，该部分只在纸卷在领地内部时才会被调用，他的时间复杂度 O(n)，n 是纸卷到最近空地边的距离。对于实际情况来说，这一部分消耗的时间也可以忽略。

遍历一遍整个纸带图和领地图去提取纸带坐标和领地坐标是必须的，时间复杂度为 O(m), m 为总格子数。

时间复杂度分析的重点自然是整个算法的第一部分，即我方回城的最短距离及路径，对方回城即的最短距离，我方击杀对方的最短距离及路径和对方击杀我方的最短距离。

首先分析 A* 算法，就笔者认为，A* 是更加启发式的 Dijkstra 算法，且在我们当前的游戏中理论效果应当非常好。A* 算法的核心是 F = G + H，G 是起始点到当前点的距离，H 是估算的当前点到目标的距离，我们采用了曼哈顿距离，即直接将当前点和目标点的横纵坐标差加在一起。Dijkstra 其实就相当于忽略了 H 这一项，只保留 G。

因为在我们的游戏中较罕见纸带走的非常混乱，严重影响探索的情况，所以 A* 基本上能径直找到最优路线，如下图：

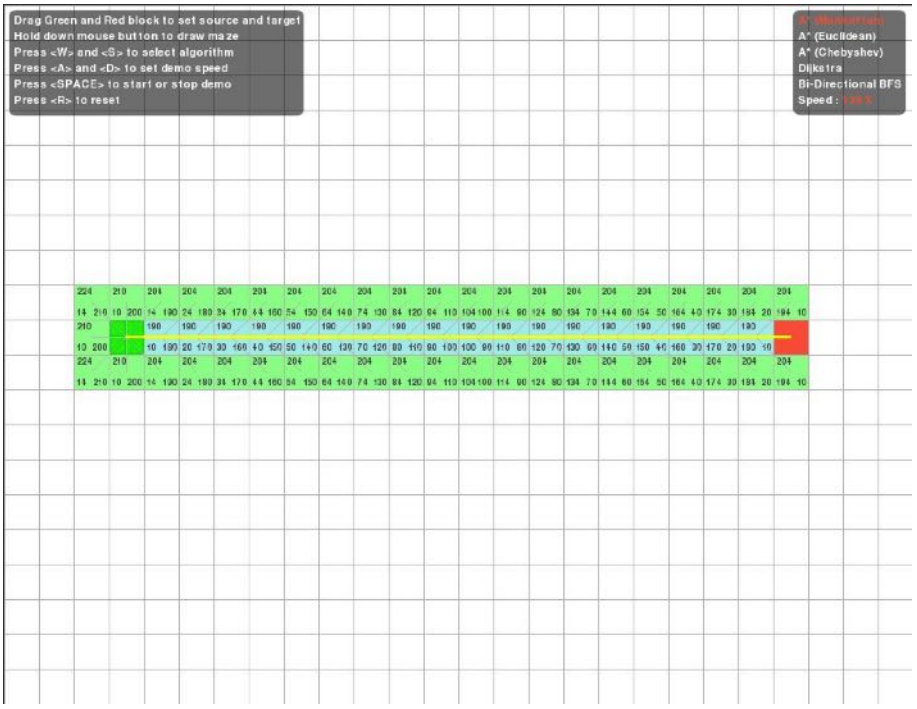


图 4 A* 算法（曼哈顿距离）

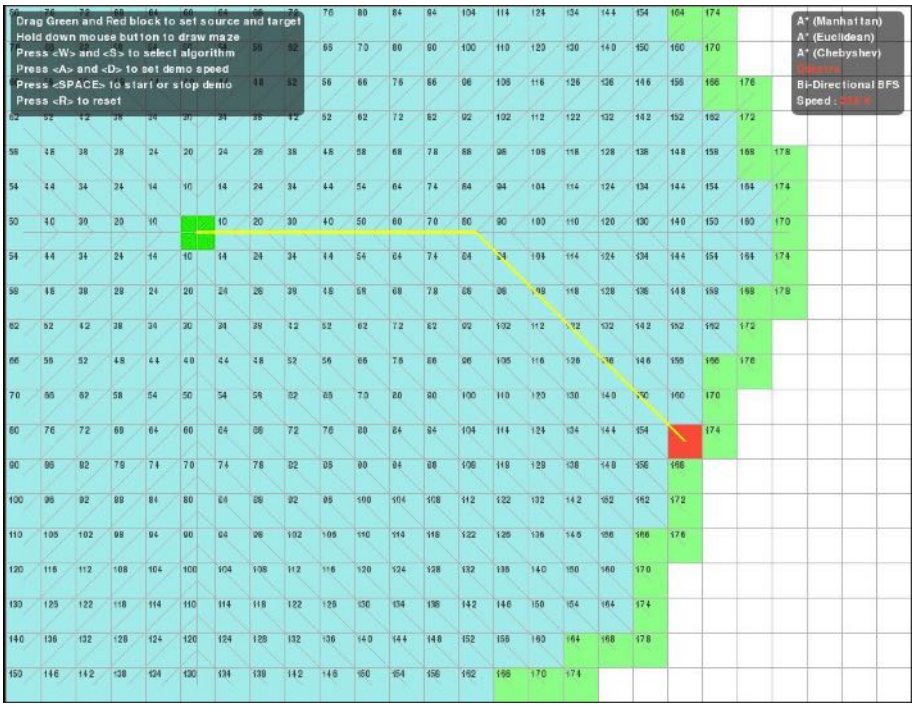


图 5 Dijkstra 算法

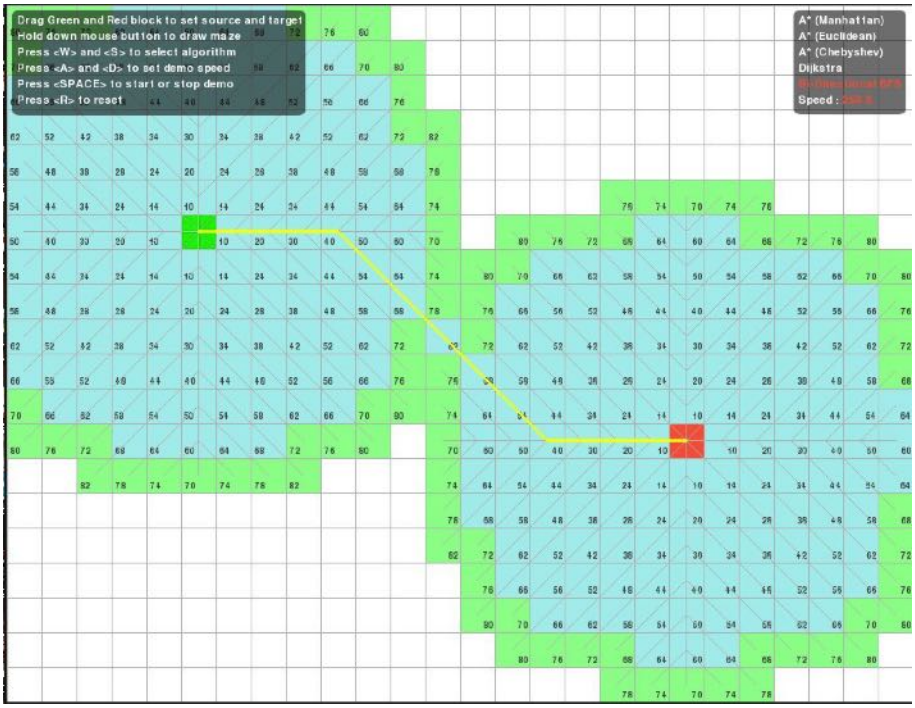


图 6 双向广度优先算法

可以看到在这种我们通常遇到的简单情况下，A* 算法比 Dijkstra 算法和双向 BFS 好太多，其操作数为 T(3n+6)，粗略的看时间复杂度为 O(n)，n 为目标和起始点间的距离。但 A* 算法每探索一个点就要更新一下 F 值，并维护一次优先队列，维护优先队列的时间复杂度是 O(logm)，m 是整个地图的格子数, 维护 n 次为 O(nlogm)。而且注意到，若要使用 A* 算法都需要将正题图上的每个格子都转成 ADT GRAPH 中的 vertex，也就是每次都要重建一次图，这是非常耗时间的，所以使用一次 A* 的时间复杂度为 O(n+nlogm+m)。

总之这种耗时程度在实际使用中还是不可以接受的，所以我们选用了更加启发的方式，即直接计算起始点和目标点间的曼哈顿距离，算一次的时间复杂度显然是 O(1)，比起 A* 算法来基本上没有时间消耗。

但这么简单当然是不可以的，为了保障安全，在计算对方攻击最短路径时还要计算对方离我们拟定的最短回城路径间的距离。拟定的回城路径其实有两条，一条是先走南北方向，再走东西方向；另一条路径反之。对方最近的攻击距离应当为对方离两条拟定回城路径中较远者和对方离我方纸带的最近距离二者的较小值。总之，计算对方离我方纸带的最短距离要遍历一遍这三条路径，所以操作数约为 T(3*n)，时间复杂度为 O(n)，n 为纸带长

度。这种算法在实际操纵中比 A* 快太多，而且就我的实战观察，极其罕见这种算法失效的情况，所以最终就采用了这种极其启发式的算法。

为了搜寻回城最短路径，应当得到双方领地全部的边界，然后计算卷轴到领地边缘的曼哈顿距离，然后生成上一段提到的两条路径，并观察这两条边是否会与当前纸带交叉，排除掉交叉致死的情况。因为回城距离与纸带长度成正比，判断是否交叉采用的是逐点比对（这里看起来确实不太好），所以判断路径是否交叉的时间复杂度是 O(n*n)，再算上领地的边界长度，时间复杂度为 O(k*n2)。在领地较大的时候 k 会很大，所以为了减少耗时，我在遍历领地图寻找边界的同时将里纸卷距离较近的边界点提前了。在制定返回路径时优先比较距离近的点，无法抵达的话在考虑较远的点，所以大多数情况下回城路径搜寻的时间复杂度都为 O(n2)。

计算我方进攻对方的算法不会去预判对方的回城路径，因为这基本不太可能，但需要计算攻击路径是否会与我方纸带交叉，时间复杂度为 O(n1*n2*d)，n1 是我方纸带长度，n2 是对方纸带长度，d 是双方距离。计算对方的攻击距离不会考虑路径，只算一个曼哈顿距离。

最后来总结一下，整个算法的时间最主要就是消耗在了计算双方攻击以及回城路径上，双方纸带拖得越长就会越慢。A* 算法虽然明显优于 Dijkstra 算法和双向 BFS 但因为实际耗时过长而被抛弃了，调整后的启发式算法性能基本理想。总的时间复杂度为 O(n12+ n1*n2*d)，所以双方纸带较长时耗时增加非常明显。

2 程序代码说明

2.1 数据结构说明

A* 算法中用到了图的结构。我们需要将整个地图生成一张 ADT graph 才可调用 A* 路径搜索算法，每个格子都是一个 vertex，东西南北四个方向各有一条 edge。此外，A* 中对于每个 vertex 的 F 值维持着一个优先队列结构。

但实际上我们之定义了 node 这个类，且它包含的属性极少：

class Node:

def ___init___(self, value, point):

self.value = value

self.point = point

self.parent = None

self.H = 0

self.G = 0

def move_cost(self, other):

return 1

value 记录这一点的属性，即如果其 value 为我方号码，则这个 node 是我方纸带，处理时需要被视为障碍物。

Point 记录了这一点的坐标，parent 记录 A* 算法探索时该节点的前驱，H，G 记录从出发点到这一点的距离，以及预期的从这一点到终点的距离。

实际上我们并没有定义一个图类，这是因为对于这张图来说，大多数的边是毫无用处的，建立他们完全是在浪费时间，而是将每个节点都储存在了原始的二维 list 里，直接靠坐标来索引并使用他们之间的连接会快得多。

2.1.2 列表

我们最主要使用的数据结构当然是 python list，其性能高效，使用便捷是自己构建的数据结构不能比的。因为使用自定义的数据结构每次需要先生成类里的元素，这是比较耗时的，所以在实际使用中我们尽量避免了自定义类。

比如接口中，储存纸带地图和领地的地图的都是二维的列表。我们转换的纸带坐标，领地边缘坐标等也都是列表。

2.1.3 其他

在之前的时间复杂度分析中也提到，我们在相同元素的比对方面做得不是很好，应当尝试使用 ADT map 去降低其时间复杂度。但无论是散列表还是二叉搜索树，创建 ADT map 都要消耗额外的时间，其效果需要实证才能下结论。

最后注意到我方程序在宏观上是在不同的状态下转换，当前状态和转移规则决定下一个状态，处于何种状态直接决定了当前代码的决策，这是一个马尔科夫链的过程，同时形

成了一个有向图的结构。

2.2 函数说明

2.2.1 Play 函数

2.2.1.1 基础函数

整个程序中一共定义了 8 个基础函数，分别用于计算敌我之间的距离，判断目前我方状态，避免自杀等情况。

（1）dt (a,b) 函数，路径距离计算：

输入 a、b 两个二维数组包含（x，y）的坐标，计算两个坐标的 x 与 y 坐标差的绝对值之和，称为“路径距离”。

（2）elic (pacman, food, band, stat) 函数，采用启发式方法计算距离和路径的普遍函数：

pacman 代表自己的坐标，food 代表目标位置的坐标，band 代表需要用来对路径进行判断的纸带坐标，stat 是输入的与场地长宽有关的变量，其中 stat 第一个值为宽，第二个值为高。具体计算方式如下：

第一步，判断自己的坐标与目标坐标的相对位置，根据位置制定两条相应的抵达路径，（例如自己在目标的左下方就可以先向右再向上、或者先向上在向右来抵达相应位置），所有需要经过的路径上的坐标存储在 path1 和 path2 中；第二步，判断路径是否可行（此处仅考虑是否会沿此路径是否会撞到输入的纸带所包含的坐标，以及如此行进是否会出界，不考虑其他诸如敌人等因素），根据判断，如果两条路径都不可行，返回 -1，如果其中一条路径可以则返回其总长度和路径的所有坐标，或者返回两条路径和其最小长度。

（3）next_step (positionlist, n=1) 函数，计算下一步状态：

用于计算下一步的三种可能状态（包括 x、y 坐标和方向）返回一个包含三个位置状态的列表。即根据当前的位置（x、y 和方向）确定下一步的 x、y 和方向。

（4）willSuicide (me, direction, field, band) 函数，判断自己是否会自杀：

输入自己的位置状态（x、y、方向），看由此走的下一步是否会撞到自己的纸带或这撞墙（出界）会返回 True，，发出警报并改变纸带行进方向，不会则返回 False。

（5）attack(me, bestpath) 函数，定义攻击：

me 包含现在位置的三个参数（x、y、方向），沿着最优路径 bestpath 的路径进行攻击，bestpath 包含路径的长度和依次经过的位置坐标。如果最优路径不存在则判断任意一个方向只要不会自杀就往这个方向行进；否则计算最优路径 bestpath 的从第一部到第二步的 x 和 y 坐标差距，并根据行进的方向进行判断下一步需要做的操作，例如计算的 x 坐标差为 1，y 坐标差为 0，（即向东走一步），此时的方向为向北，那么，接下来需要右转，则返回‘r’。

（6）escape (me, backpath) 函数，定义逃跑：

me 包含现在位置的三个参数（x、y、方向），沿着回城路径 backpath 的路径进行攻击。其判断方式与 attack 函数一致，不再赘述。

（7）init_parameter() 函数，初始化参数：

该函数用于根据自己是先手还是后手的状态，判断自己第一和第二步应该行进的方向，如果自己在对手的下方，则将自己的方向调整为向上，在对手的左方则调整为向右，总之就是将自己的方向迎向对手。这是我小组在多次对战后根据我方代码受限于同组的 delta 和 quebec 两个强组所采取的攻击型方案，以迎击对手的态度来主动谋求对战，占据先机。

（8）init_move(list) 函数，根据最初状态圈地：

其中调用了利用 load 函数中定义的 storage 中的状态‘step’，‘step’记录了行走过的步数，前两步调用 list 中的两个返回值，list 通常是初始化参数函数的返回值。如果走得步数已经大于 2 步，可以利用之后的判断，storage 中的‘count’用于临时记录行走的步数，当行走的步数等于我的位置与敌人的位置的距离时（即设定的参数）开始转向，圈地方式为圈一个正方形。

圈地采取正方形是我方代码的一个缺陷，我方最初设计时本计划在圈地过程中也能实时根据对手和自己的状态改变目前圈地的方式，例如走长方形，或是锯齿形等不规则图形来谋求圈地最大效率。但是由于后者需要进行精确的设计（例如保证改变圈地状态后纸带还能准确地回到我方领地）且需要多次更多地考虑双方接下里的若干步后的状态（确保不会走入对方的领地或者对方接下来圈地范围等），时间复杂度过高，在这种情况下，我方代码面临严重的超时问题，故只能舍弃掉更加完备的圈地方式，采取简单策略。在我们之后的改进中可以通过更加详细的方法改善这一问题（例如使用 if/else 判断而非复杂的算法来

确定圈地策略）。

2.2.1.2 变量准备

（1）确定自己和敌人的纸带坐标：

敌我纸带坐标的变量是函数中最为重要的变量，决定我们接下来的行动策略（进攻 or 逃跑等状态），并保证自己不被杀死。我们的函数每一步都将遍历 band，把其中我方纸带的坐标存储在 myband 中，地方纸带存储在 enband 中，

（2）计算自己离敌人纸带的最近距离：

为我方转换到进攻状态做准备，计算自己当前位置坐标到敌人纸带上每一个点坐标所需要经过的路径长度，将得到的最小路径长度值赋给 enbestpath，在这个值达到一定小的情况下（例如小于对方回领地距离并且小于对方攻击到自己路程上的距离），将触发进攻。

（3）计算当前位置与领地边缘的距离：

遍历 field 中的点，分别计算自己当前位置到领地边缘的路径距离，以及敌人当前位置到其领地边缘的距离，并将经过计算的点根据其路径距离的大小顺序依次存入一个新的专门存放边缘坐标的列表，自己的和敌人的分别存放，称为 msl 和 esl，方便之后调用。

（4）计算自己和敌人的回城路径：

将自己的回城路径 msl 列表中保存的边缘点的坐标依次取出，使用 elic 计算其抵达该点的路径，如果返回值为 -1，代表不存在到达该点的简单路径（elic 函数只计算包含至多一次转向的简单路径），则继续取下一个与当前位置路径距离最小的点计算，直到找到一个路径距离最近且可以通过简单路径到达的边缘点。接下来进行下一步判断，如果存在两条简单路径，分别计算两条路径的上所有点距离敌人当前位置的距离，并取其中的最大值，记为 en2me2，即选择其中距离敌人较远的路径；如果只存在一条路径，同样计算距路径上的点距离敌人的最小距离，并记为 en2me2。利用同样的办法计算敌人的回城路径。

（5）领地内近战部分（优先级最高）：

当我们位于自己的领地内，而且敌人距离我们只有 8 以内的路径距离，进入近战模式。调用 next_step 函数计算当前位置下一步的三种情况，并判断是否会出界（撞墙甚至撞墙角），如果会的话就迅速执行唯一可以选择方向行进，如果不会则判断下一步怎么走及自己是否还在领地内，优先选择在自己领地内行走，避免被对方在领地内撞击。

2.2.2 Load 函数中的主要函数与 storage

（1）fieldwalk(band, field, me, storage) 用于我方纸带在领地内行走的情况：

它首先判断是纸带否会在 x 方向和 y 方向出界，如果会则迅速选择不会出界的方向进行转向，并暂时将状态转变为‘goback’；如果出了领地，那么立马变成圈地模式‘draw’，同时判断出领地之后前进的方向上空地的长度有多少的。我们选定圈地的边长就设定为空地的长度，避免出现不断来回圈地的低效率现象；在非边缘地带回到领地后，转向两次掉头，不在领地内部逗留；判断上、左、右三个方向哪里更靠近对方的领地，就朝那个方向走，从而侵蚀对方的领地，并扩大自己的领地以提高效率。

（2）draw(band, field, me, storage) 指挥我方纸带在领地外圈地：

首先判断是否会在 x 方向和 y 方向出界，如果会则迅速选择不会出界的方向进行转向。画方块圈地，按照之前设定的边长（即 fieldwalk 函数在出领地后计算得到的长度）进行正方形的圈地；如果回到领地内，转换为 fieldwalk 模式。

（3）goback(band, field, me, storage) 防撞墙模式：

在领地内要撞墙时使用 goback，并前进指定步数为 5，5 步后状态转换为 fieldwalk，这一步的目的是不让 AI 在边缘逗留太久。

（4）其他函数:

Load 中同时也使用了与 play 一样的 distance、next_step、next_nstep、willSuicide 等函数。

（5）Storage 的初始化写入：

storage[’mode’] = ’fieldwalk’ # 记录当前在一般状态下的模式，初始模式为 fieldwalk

storage[’fieldwalk’] = fieldwalk

storage[’draw’] = draw

storage[’goback’] = goback

storage[’turn’] = choice(’rl’) # 初始 turn 为随机选择的左转或者右转

storage[’count’] = 0 count # 参数用于临时记录走过的步数，设定为 0

storage[’step’] = 0 # 记录总步数

storage['loop'] = 0 # 临时记录转过的圈数

storage['flag'] = 3 # 初始状态设定为 flag 为 3，即为初始圈地状态

storage['maxl'] = 40 # 正方形的边长

2.2.3 状态转换

2.2.3.1 状态类型

我们一共定义了四种状态，分别为 flag0、1、3。其中 flag3 是初始状态（主要用来进行初始圈地操作），2 是逃跑状态（跑向自己的领地内），1 是进攻状态，最后的 0 是一般状态（努力进行圈地以及领地内行走）。

其中一般状态下又分为三种“模式”：draw 表示普通圈地模式，fieldwalk 表示我方穿过领地从走到另一边的过程，goback 表示在领地边缘返回。

2.2.3.2 状态转换

状态 3：初始状态

开始时处于在初始状态 3 下，通过判断进行状态装换。初始状态 3 结束，之后 0、1、2 三种状态交替变换。

状态 0：一般状态

如果自己在领地内，则为领地内行走模式‘fieldwalk’模式，根据 fieldwalk 函数来行动，如果可能出界则短暂转变为‘goback’模式。出了领地转变为圈地模式‘draw’模式：首先判断，敌人是否在领敌人的地内，如果在，就把敌人放到纸带里面，因为纸带是我们禁止走的区域，（防止被敌人撞击，这不意味着不往敌人的领地走），根据 draw 函数进行圈地。

状态 1：进攻状态

如果我们距离敌人纸带比敌人距离我们的纸带近，而且我们到敌人纸带的距离比他回城的距离近，就转换为进攻状态（flag1 是进攻状态）

状态 2：逃跑状态（最高优先级）

当敌人纸带距离我们的纸带或者我们回城的路径距离小于我们回城路径的距离加 5 时，状态转换为逃跑，根据 escape 函数来进行逃跑，返回的是左转，右转，或者直行。逃跑状

态有最高优先级，一旦进入逃跑模式，直到回到领地之前都不再转变为其他模式。

如果碰到的领地是我的领地，说明回到领地了，这时脱离初始圈地状态，进入正常状态（flag 0 是正常状态，进行圈地和领地内走）

2.3 程序限制

我小组最初的代码存在边界自杀现象，检查时发现是函数设定中边界情况未能良好考虑在内，之后修复了这一错误。在比赛阶段使用的代码不存在出错的问题，但是仍然有一些需要改进的部分：

1. 近战代码考虑的情况十分有限，在边界较为复杂的情况下还是会被对方击杀；

2. 启发式算法计算四个参数的方法在纸带路线超复杂的情况下会失效，如果两条垂直的路径都不能走的话将完全不能返还回城路径；

3. 我们的代码只能根据当前局势进行决策而缺乏先读的能力，比虽然计算了回城距离和路径，但很有可能边界会被对方吞并，导致回城有关参数突变，此时情况就十分危险了；同样，我们也不能有效利用对方回城参数的变化击杀对手；

4. 介于我们的圈地方法是在不断尝试中构建的，数学上将肯定不是最优的，圈地效率还有些随机，时而会比较低效；

5. 还有很多可能发生的复杂情况未被考虑在内，这些都需要进一步改进；

6. 算法上一些已有的部分还有明显的不足，比如比较两天序列的坐标有无重叠时，可以通过散列表等方式降低时间复杂度。

3 实验结果

3.1 测试数据

（一）实验环境：

本组一共五名同学参与了代码的测试，主要测试方式是与其他组（主要是和我们不在一个赛区的 F17 区的组）进行对战，各个测试者的实验环境如下：

实验者	张毓昕	陈峻松	王天贺	范家豪	潘登
CPU	Inter（R）Core（TM）i7-6700HQ @2.6GHz	Inter（R）Core（TM）M5-6Y54 @1.1GHz	Inter（R）Core（TM）i5-6200U @2.3GHz	Inter（R）Core（TM）i5 @2.7GHz	Inter（R）Core（TM）i5-6300HQ @2.3GHz
内存	16GB，DDR3 1600MHz	8GB	8GB	8GB DDR3 1867MHz	8GB
操作系统	Windows7 旗舰版	Windows8.1 专业版	Windows10 家庭中文版	MacOS 10.13.4	Windows10 家庭中文版
Python 版本	3.6.4	3.5.2	3.5.2	3.6.4	3.6.5
备注	单线程单核运行	单线程单核运行	单线程单核运行	单线程单核运行	单线程单核运行

注：实验者的实验环境不同主要在于 CPU 和内存方面，大家在实验过程中都是以电脑的默认设置进行测试，不存在超频和多线程的情况，前期主要测试代码是否出错以及对战的胜负情况，临近比赛前除上述测试外还主要考虑了是否超时的问题，所以需要更接近比赛的环境 Inter（R）Xeon（R）Silver 4114CPU@2.20GHz 10 核心超线程双 CPU，这点在后面详细叙述。

（二）测试方法

我们组采用的测试方法主要是与 F17 大区的其他代码进行比赛，主要的假想对手是 F17 区 Foxtrot 队的代码以及 Manhera 组的代码，其中 Manhera 队的代码与我们组最终冠军 Delta 和季军 Quebec 总体方针类似，以圈地为主的思路的，主要是用来测试如果走完 4000 步我们代码的圈地效率；而 Foxtrot 是在热身赛中以进攻为主要特点的代码，我们通过与他们的比赛，测试我队代码的防御能力。测试中完全模拟真正比赛时小组赛的情况，双反作为先手和后手方分别进行 10 场比赛，记录比赛结果，分析代码中存在的问题进行修改；如果实验中出现了报错的问题则直接终止实验，寻找问题所在并且修改代码，修改过

程中为了知道代码某些情况下是否按照我们的思路运行，我们设置了一些输出来判断是否正常运行，比如输出 0 表示这一步左转，这在我们除了最终的 3.7 比赛版本之前的版本代码中都有体现。

测试中除了保证代码不报错和不犯低级错误以外，后期主要测试的是超时的问题，测试中我们发现即使在 20 局的情况下，由于我们和上述两组的代码并没有巨大的算法上克制或者实力差距，测试比赛的结果与电脑相关性很大，大家测试的结果不尽相同。但是走完 4000 步时候的耗时是与处理器的性能负相关的。由于在比赛之前没有通知我们时间限制又放宽到 60s，我们非常担心超时问题紧急放弃了部分防御的计算为了缩短时间，最后在性能略优于比赛环境的 6700HQ 处理器的电脑上，走完 4000 步平均耗时降为 10s 左右，最多不超过 15s。但我们之前的代码平均耗时也在 27s 左右，如果提前获知小组赛中时间限制放宽，我们认为作为小组第三名的我们与第二名仅差一个胜场而惜败，我们若没有为了缩短时间放弃部分性能，那么出线的几率还是很大的。

（三）测试结果

我们在 3 代版本之前主要是在优化我们的传递参数的算法以及圈地的效率，在 3.3 版本之后的代码则是改为大量的与前文提到的两组的代码进行对战，在实战中不断改进我们的代码，下文给出的是我们的不报错的完整版代码与对手进行完整的 20 场对战的结果，测试结果如下：

1. 自主测试

（注：后期为避免超时我们选用了 6700 和 6200 的 CPU 进行实验）

版本	对手	先手比分	后手比分	总比分	概况说明
3.5	F17_quebec	6:4	8:2	14:6	大比分获胜主要采取撞杀的方式，但是失败很多次由于超时
3.5	F17_foxtrot	5:5	6:4	11:9	基本打平。撞墙失败几次，改进代码
3.7.1	F17_quebec	4:6	4:6	8:12	为不超时简化防御，胜率明显降低
3.7.2	F17_quebec	6:4	3:7	9:11	基本打平，双方多次走完 4000 步结束

3.7.2	F17_foxtrot	8:2	5:5	13:7	多次被撞纸带失败，胜利多为圈地
3.7.3	F17_foxtrot	5:5	4:6	9:11	基本打平，zyz 胜利多为装纸带，我方胜利多为圈地
3.7.3	F17_quebec	7:3	6:4	13:7	多数走完 4000 步以圈地面积取胜，接近最终版本，耗时也大大缩短

从自主测试的结果来看，我们的防御机制一直是值得骄傲的地方，但不足的是耗时较长，在 3.5 版本与 qqq 比赛时不时出现超时现象，进行改进之后的 3.7 版本开始出现了一些问题，到 3.7.3 版本之后对 qqq 的成绩较好，但对战 zyz 时表现随机性很强，但考虑组内的算法从热身赛来看多数都是圈地类型（有三个组并未参加热身赛），所以我们最后也选择了这个版本作为最后的比赛版本

2. 热身赛结果

	delta-1	delta-2	delta-3	quebec-1	quebec-2	quebec-3	Quebec
我方先手							
1	我方撞纸带自杀	被对手撞击	被对手在领地内撞击	我方决策时间耗尽	我方决策时间耗尽	被对手撞击	我方决策时间耗尽
2		我方决策时间耗尽	我方撞纸带自杀	我方撞纸带自杀	我方撞击对手纸带	我方决策时间耗尽	
3		我方决策时间耗尽		我方在领地内撞击对手	我方决策时间耗尽	对手撞纸带自杀	
4		我方撞纸带自杀		我方决策时间耗尽	我方撞击对手纸带	我方决策时间耗尽	

5		我方决策时间耗尽			我方撞纸带自杀	我方决策时间耗尽	
6		被对手撞击			我方决策时间耗尽	我方决策时间耗尽	
7		我方决策时间耗尽			我方决策时间耗尽	我方决策时间耗尽	
8		我方决策时间耗尽			我方决策时间耗尽	我方决策时间耗尽	
9		我方决策时间耗尽			我方决策时间耗尽	我方决策时间耗尽	
10		我方决策时间耗尽			我方被对手撞击纸带	我方决策时间耗尽	
胜负情况	1 负	10 负	2 负	3 负 1 胜	8 负 2 胜	9 负 1 胜	1 负
对方先手							
1	我方决策时间耗尽	我方决策时间耗尽	在对手领地内撞击对手	对手撞纸带自杀	我方决策时间耗尽	我方决策时间耗尽	被对手撞击
2	我方决策时间耗尽		被对手撞击	我方决策时间耗尽	我方撞纸带自杀	我方决策时间耗尽	我方决策时间耗尽
3	我方决策时间耗尽		在对手领地内撞击对手	对手撞纸带自杀	我方在领地内撞击对手		我方决策时间耗尽
4	我方决策时间耗尽		回合数耗尽我方面积小	我方决策时间耗尽	我方决策时间耗尽		被对手撞击
5	我方决策时间耗尽		被对手撞击	我方决策时间耗尽			被对手撞击

6	在对方领地内撞击对手		被对手撞击	我方决策时间耗尽			我方决策时间耗尽
7	在对方领地内撞击对手		在对手领地内撞击对手	我方决策时间耗尽			我方决策时间耗尽
8	我方决策时间耗尽		被对手撞击	我方决策时间耗尽			我方决策时间耗尽
9	我方决策时间耗尽		回合数耗尽我方面积大	我方决策时间耗尽			我方决策时间耗尽
10	我方决策时间耗尽		在对手领地内撞击对手	我方决策时间耗尽			我方决策时间耗尽
胜负情况	10 负	1 负	9 负 1 胜	8 负 2 胜	3 负 1 胜	2 负	10 负
总情况	11 负	11 负	11 负 1 胜	11 负 2 胜	11 负 3 胜	11 负 1 胜	11 负

总体上看我们组的热身赛结果还是不容乐观的，由于比赛机制有一方先达到 11 胜则直接结束比赛，所以上面热身赛结果中出现了很多空白。但是去看每一局的比赛过程和结果，我们发现我们对战过程中多数情况下是占优的，我们失败的七成以上的原因是决策时间耗尽，由于我们的算法本身就比较耗时，再加上热身赛的电脑的配置未知，导致我们的热身赛（包括与其他组的比赛）多数情况下都是因为超时直接被判负了，而在正常结束的比赛中我方也有几次撞击对手和圈地面积更大赢得比赛。

另一方面，热身赛我们组内只有包括我们和 Delta、Quebec 三个小组提交了代码，而最终另外两组分别是 N17 区的冠军和季军，虽然还是我们代码的事例不够，但这也是一方面的原因。

综上所述，由于开始写代码较晚，耗时问题以及分组等主管客观的原因，我们的热身赛结果并不乐观，在热身赛之后我们也进行了即使的总结和反思，并开始研究一些强的对手的思路，加紧升级自己的代码。

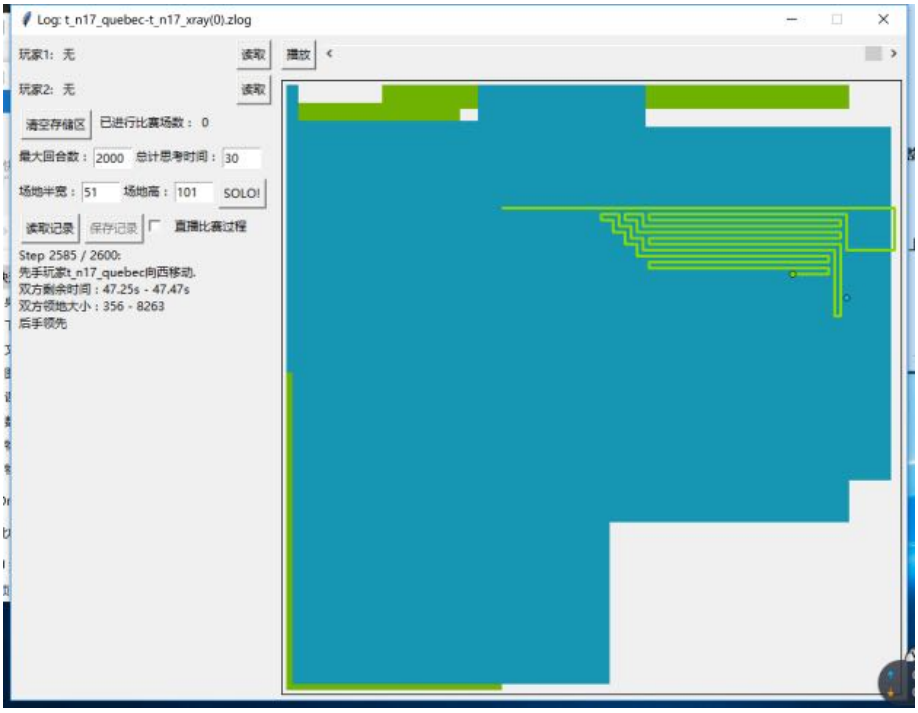
3.2 结果分析

在测试中我们发现我们的防御机制的性能还是比较优秀的，并且最后比赛中证明了优秀的算法在这方面的处理机制都是很相似的；我们定义的 elic 函数采用启发方法计算距离和路径，计算准确并保证能在击杀和回城中走最短的路径。初始的圈地模式保证我们前期安全的情况下获得较大的领地，从而使得我们在很早期就与攻击性代码在边界循环，直到走完 4000 步时能够依靠圈地面积获胜。而定义的 fieldwalk 函数保证了对战圈地型代码时，我们的代码又具有一定的进攻性，能蚕食对方的领地，不让对方获得很大连续的领地，在实验过程中我们的一名组员称我们与 qqq 的对战为“百团大战”，函数的这个部分保证了会切割对方的领地，所以在后期经常出现双方的领地面积忽然变化的情况，经常发生大反转。

实验中我们算法的运行时间基本符合预期，但是在调整时间限制 60s 到 30s 之后，就不时会出现走完 4000 步超时的情况，时间消耗主要是在遍历双方的纸带获得信息，进而计算我方和对方攻击和回城的最短距离上，所以我们的时间消耗与双方步数的关系是正相关的且均匀变化。为了处理这个问题，我们加入了 truecal 函数，计算我离对方知道的最短距离，在一定条件触发这个函数减少时间的消耗，效果很显著，之后我们走完 4000 步消耗的时间控制在了 15s 以内。

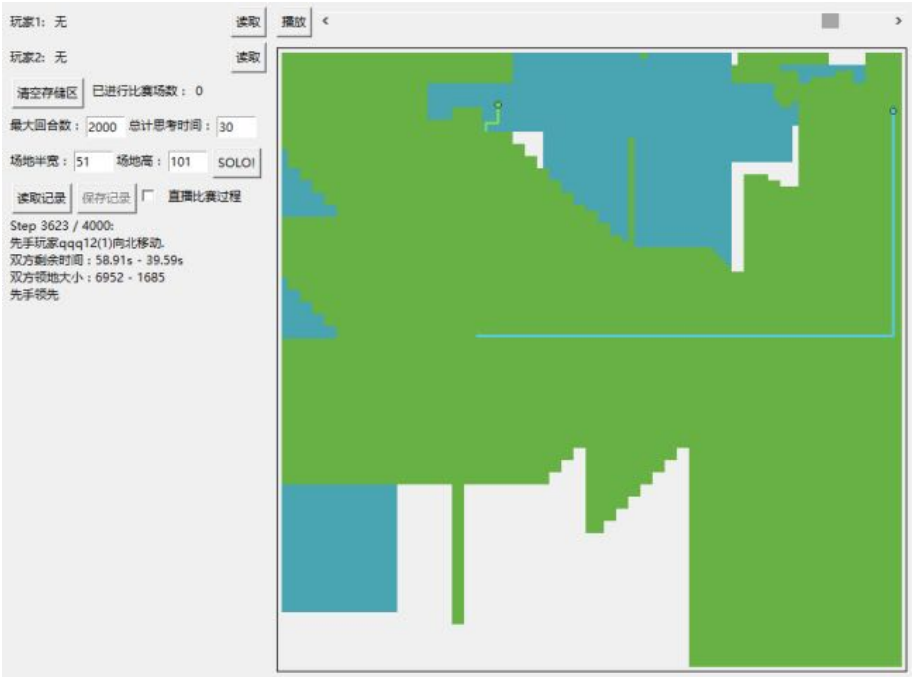
3.3 经典战局

（1）小组赛中对战季军 Quebec



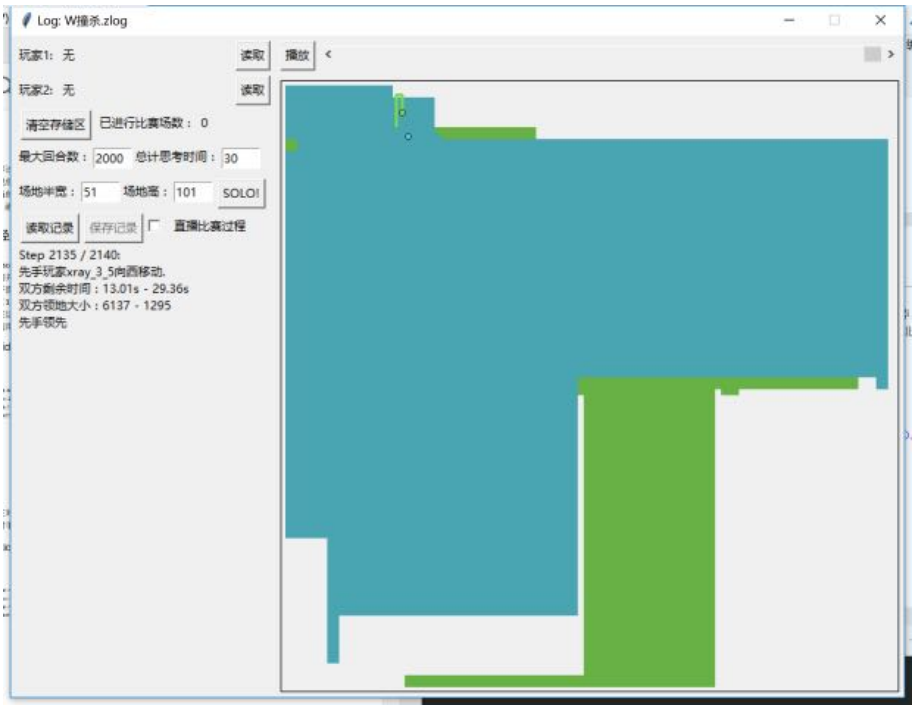
这场比赛中我们很精彩的后发制人，从 Quebec 的后方偷袭吃掉了它的领地，使它的纸带失去了源头也就无法计算，最终只能挣扎着被我方撞击纸带。

（2）自主实验中 3.5 vs. qqq



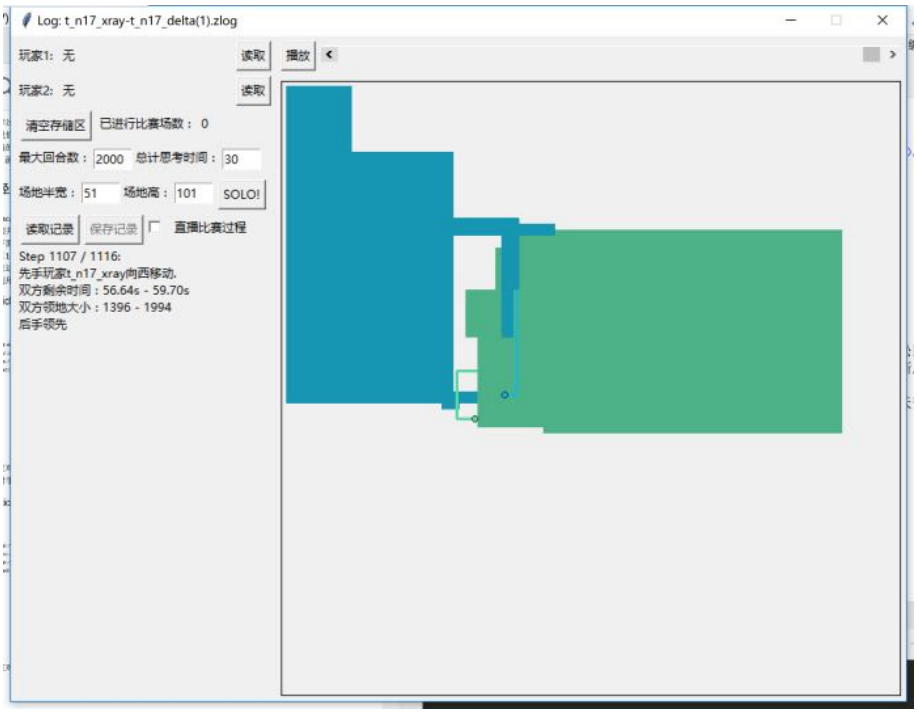
在这局中我们出现了和正式比赛中 Quebec 相同的情况，后方的基地被人全部侵蚀掉，但是我们及时的回到了上方的小块基地保证安全，但无奈此时圈地面积相差太多最终败北。

（3）自主实验中 3.7 vs. qqq



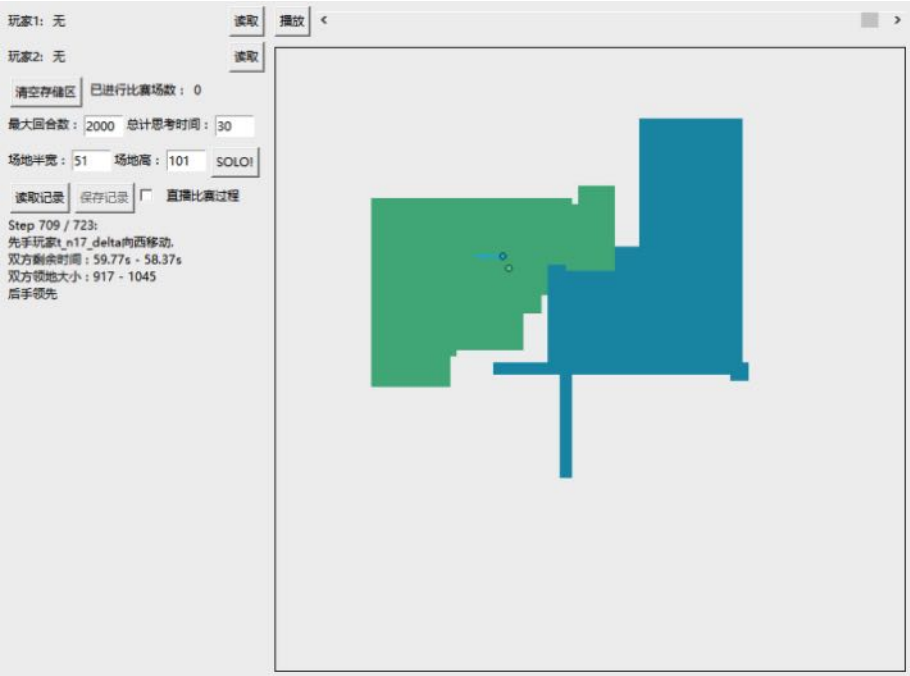
本局我们开始圈地落后但是因为函数中的侵入对方领地的算法，不断追击对方，最终将其逼入绝境轻松击杀

（4）正式比赛中小组赛对阵冠军 Delta



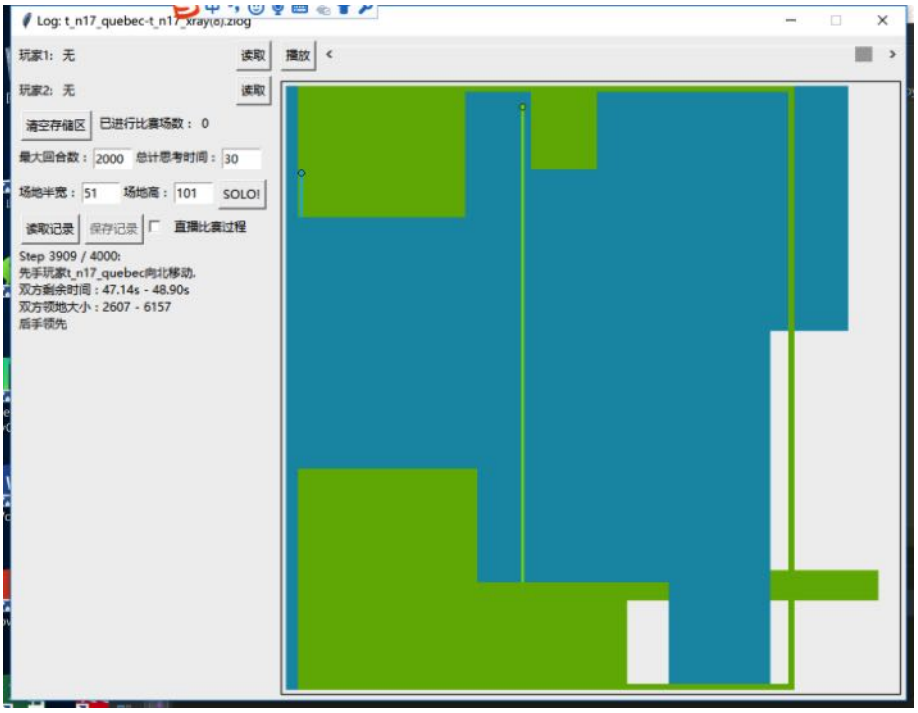
本局同样是侵入对方领地的方式，我方回城的步数本来是足够的，但是 Delta 的算法在此时忽然吃掉了我方边界的一块领地，回城距离突变，我方变成在其领地中深入，无奈被撞头击杀。

（5）正式比赛中小组赛对战冠军 Delta



这次在我们自主实验中多次发挥作用的侵入对方领地的函数，从后方包抄切断了 Delta 的全部后路，只剩一个纸带头在我方领地内，毫无悬念的拿下这一局。事实上我们在小组赛中先手对阵 Delta 比分 3:7，胜利的场次都比较精彩，而冠军组 Delta 在淘汰赛中两次横扫，所以还是比较遗憾与冠军和季军分在同组。

（6）正式比赛小组赛中对战 Quebec



这局比赛非常经典，我们与他们的代码都是圈地为主，在走到 3900 步时候我们依然巨大的圈地优势领先，但是 Quebec 忽然冒险横跨整个 y 轴一次性得到了将近全屏一半的领地，由于所剩步数太少我们最后圈地面积小于对方而败北。

4 实习过程总结

4.1 分工与合作

开发算法与程序编写：主要由王天贺、范家豪两位同学负责代码编写，其他同学负责为程序设计提供思路，并与其他组同学交流、吸收经验为我们所用。作为组长，范家豪参与全局，把控实习作业进度。小组报告：由小组所有成员共同编写。

合作交流方式：

1. 在 39 号宿舍楼 320 寝室讨论
2. 在微信群讨论沟通，并上传代码
3. 在农园咖啡厅、理科教学楼开展讨论

历次组会记录：

5.22 确定小组成员：王天贺、潘登、范家豪、陈思如、陈峻松和张毓昕，队长为范家豪；小组成员互相熟悉，同时确定在该天进行了游戏体验，了解了游戏的基础玩法；

5.27 小组召开第一次正式组会，大家在提前了解了 solo 等基础代码模板后明确了我们的代码应该如何编写。小组进行了任务分工……同时小组成员讨论了我们的游戏策略以及可能采用的高级算法。

6.2 小组进行线上讨论，将几个同学分别写的部分代码合并。同时我们检测了自己的代码部分，发现存在一些 bug，程序中考虑不完整。并定下了下一次组会的议程。

6.5 小组在热身赛后组会，大家讨论了小组之前热身赛出现的问题，陈峻松负责浏览了之前全部的对战记录，为大家介绍了其它各个组的宏观策略以及对战中各组出现的错误。王天贺同学介绍了他撰写的几个重要参数的代码，范家豪同学为大家介绍了我们组最新的策略以及圈地方案。同时我们检测了自己代码的时间，确保我们能够有效地将时间降到足够低。另一方面，陈峻松联系了大一的同学，通过双方交换代码来了解代码的情况。

6.12 正式比赛前夕，大家在农园咖啡厅做最后的冲刺，感谢峻松带来的葡萄。我们分析了主要对手 delta 和 quebec 的战术特点，并集中修复了本组几个重要的问题：在领地内乱转，L 形低效选地模式，偶尔撞墙等。峻松和家豪讨论战术的同时，潘登和思如开始动笔写实验报告。当晚，两位代码编写主力人员家豪和天贺一直挑灯夜战，见到了凌晨四点的北京。

6.12 正式比赛日。Delta 毫无悬念地以赛区第一名出线，本来与 Quebec 组并列第二的 X-Ray 小组还怀有一丝希望。最后还是“不出所料”地败给对方。秉着“知足常乐”的原则，依然很开心。享受努力和探索的过程，合影一张以做纪念。

4.2 经验与教训

4.2.1 经验

AI 的编写对我们组成员来说算是全新的领域，在黑暗中摸索最终见到晨曦的感觉是如此幸福。无论战绩如何，我们已经拼尽全力，无愧于心。数算课带给我们的不仅仅是课堂上的知识，还是一次探索未知领域和创造自己作品的体验。

在编写代码的过程中，保持交流，可以利用 GitHub 平台，来保持代码的同步性。另外，代码注释是十分必要的，一来可以增强可读性，更重要的是帮助其他成员理解思路，为以后的改善提供方便，可以大大提高合作开发的效率。此次我们的对手实力强劲，我们必须要不断的改进算法。多多分析热身赛，对改进算法十分有帮助，正所谓“知己知彼，百战不殆”。

同时，小组作业锻炼了我们与人合作的能力，能力有高有低，但大家相互帮助，做好力所能及的工作，共同将大作业完成得更好。代码主要由两名同学来写，其他人负责提供思路想法及后勤保障工作。各司其职，分工明确，将每个人的作用最大化，将团队的效益最大化。

4.2.2 寄语

每学期都会有全新的游戏，选课同学都将经历全新的旅程。这门课提供了一个开放的平台，你尽可以大开脑洞，出奇制胜。将自己的想法用数据结构与算法的形式表达出来，也是一件很酷的事。还有一点就是不要一味依赖“大腿”，亲自动手耕耘才能获得真正的成长。挖掘自己的潜能比一个冰冷的名次和分数来得更重要，“尽吾力而不能至者，可以无悔矣”。希望学弟学妹们享受学习、探索和携手并进的过程，这将是一段难忘而珍贵的回忆！

4.3 建议与设想

4.3.1 一点小小的建议

本次大作业完成过程中，因为版本频繁更新，而且组内成员版本不一致，也给交流带来了一定的麻烦。希望在今后的课堂上，技术组可以尽早组织并开始平台代码的编写工作，并在发布之前尽可能完善，发布之后减少改动次数，从而使得大家在编写自己的代码时更加方便。

目前比赛的赛制还存在强弱分极的可能性，赛区与赛区的竞争强度和出线难度都存在显著差异。希望以后可以加以改进，变成大循环赛制或者按热身赛重新分区，例如按照热身赛的 1、5、9、13、17、21 名一组（其它以此类推），既可以错开表现强弱不同的队伍，也能避免有大佬故意“卖弱”。

目前约战只能自行联络一些熟悉的同学，以后可否建立一个约战平台，各组可以在上面发起和接受挑战？大家自愿交流自己的代码情况，在更大的范围内测试自己的水平，发现问题，互相促进，在相互比赛和沟通中更好地进步。

4.3.2 一些设想

纸带圈地是一项十分有趣的游戏，但一般是多人对战，数算作业可能是出于简化任务的考虑改成了双人模式，未来也许可以尝试多人对战的算法，应该会更加精彩。而且那个代码需要考虑的情况需要更多一些，但是本质并没有改变，所以是我们在自己代码基础上进行进步的一个思考方向。

另外，本学期我们和环科数算进行了友好交流，以后可以考虑举办联合算法友谊赛等。

5 致谢

首先要感谢陈斌老师一学期的教导，以及技术组成员搭建和维护平台的辛苦工作。群里优秀同学和助教的实时答疑也给我们组的工作提供了十分及时和巨大的帮助。另外，还要在此感谢与 X-Ray 约战的小组，双方在实战中发现不足，相互促进，从而将战斗变得愈发精彩！

在组长的带领下，每个人都积极贡献自己的力量。综合利用课堂所学和丰富的网络资源，完成了本次的大作业，从中获益良多，能力得到了很大的锻炼和提升。本次大作业的顺利完成，离不开组员的共同努力。

感谢 carry 全组，亲自操刀上阵又聚又帅的组长家豪，感谢编程一流不眠不休奋战到底同样很帅的天贺，感谢乐观开朗认真负责多方沟通的潘登，感谢多方联络组织约战分析赛况的峻松，感谢超级幽默口才一流的名嘴毓昕，还有负责改善男女比例以及听组长讲代码的思如（思如同学开朗活泼的性格为我小组提供了很多欢乐）。

最后，感谢 X-Ray 小组的全体成员，祝大家在今后的学习与生活中，更好地利用这堂课所学的知识，为自己的进步增添动力！

6 参考文献

A* 算法与 BFS、Dijkstra 算法比较：`https://blog.csdn.net/qq_34446253/article/details/51427423`

第二十三章　N17_Yankee 报告

拉毛切忠 *，李顺吾，刘昕玥，王子璇

摘要：本次纸带圈地作业中，本组成员的算法思路为重圈地，轻攻击，主要目的是在回合数耗尽之前尽量圈大面积的地，同时在邻近对方正在运行的纸带时进行小范围攻击。与基础设施代码网站上的示例进行对决基本会以圈地大而取胜，偶尔因为攻击取胜。进行自我对决时基本因为被攻击而结束。从结果看来我们在圈地时对敌方位置的判断以及攻击范围后还可以进一步完善。

关键字：圈地；边缘判断；敌方判断

1 算法思想

1.1 总体思路

我们的算法大致分为两个部分：游走和攻击。这两个部分是相对独立的组成部分；在游走部分里面有着防出界、画方块等次级模块；在防出界模块中有着不同边界条件的处理方式作为再次一级的模块。在我们的算法设计中，这种逻辑的处理起了很重要的作用。

因为我们认为圈地的重要性在比赛中更为重要，所以设计思路主要放在了如何安全地圈地并在适当的时候进行攻击。在圈地模块中，我们也分为了领地内和领地外的圈地模式，并在每种模式中都有着对边界和敌方位置的判定，根据判定结果进入下一个次级结构。由于两个模块相对独立并且进攻成功的成本较高，我们的进攻算法并不是主动出击，而是不断进行对方位置的判定，当对手处于可攻击的坐标范围内再发起攻击。总体上来说，我们的程序侧重与于圈地和防守，主要运行的模块是圈地模块。当对方的位置出现在可进攻的

范围内时，算法进入进攻模块。

1.2 算法流程图

我们的算法流程图可以分为领地内（图 1）和领地外（图 2）分别展示。

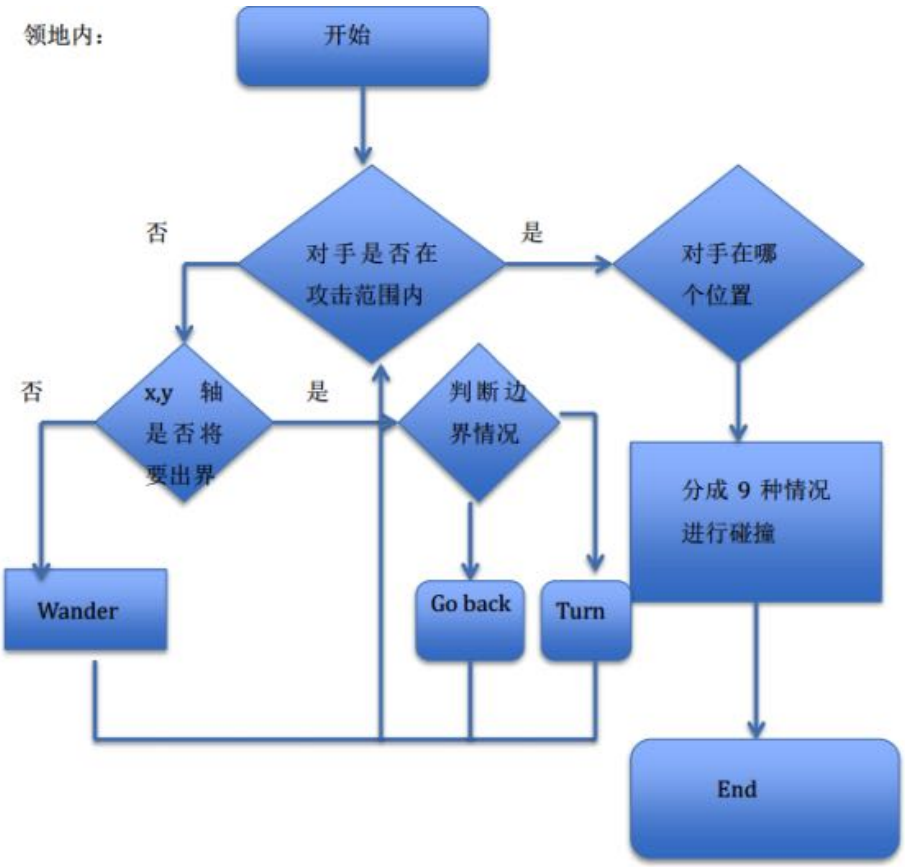


图 1. 领地内算法流程图

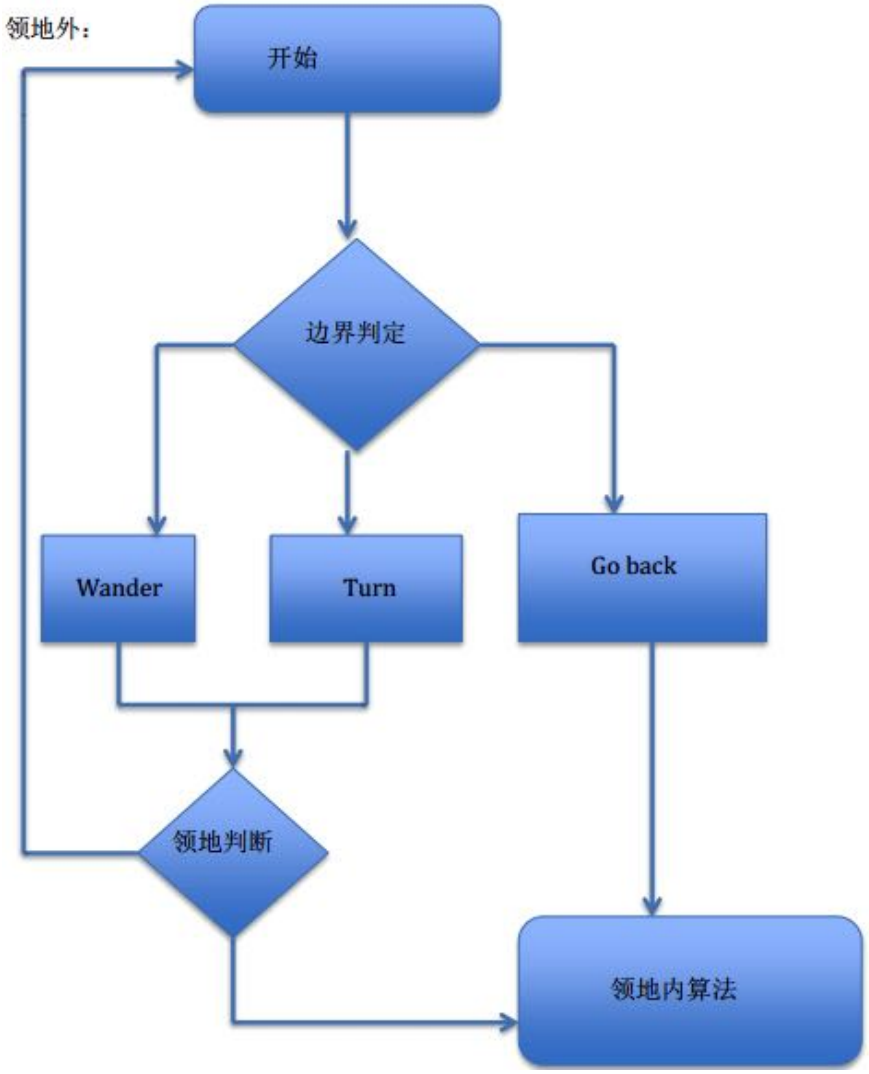


图 2. 领地外算法流程图

1.3 算法运行时间复杂度分析

根据流程图分模块来计算时间复杂度：

1.attack 模块的时间复杂度。主要逻辑是进行判断加一步操作。这部分的时间复杂度是 O（1）。

2. 领域内游走函数：从流程图中我们看到其包括 attack、wander 两个模块。其中

wander 包括了 goback 和 turn，它们都是时间复杂度为 O（1）的函数。因为领地内函数有循环，所以根据流程图判断，流域内游走函数的时间复杂度为 0（n）。

3. 领地外函数。和领地内类似，其时间复杂度为 O（n）（将领地外算法进入领地外算法的部分算作其出循环）。

所以，综合下来整个算法的时间复杂度为 O（n）。在计算时间复杂度的时候思路和之前的步骤都是一样的，即先抽象出大的模块，通过合适的逻辑层次和数据类型来实现这些模块与模块和模块之间的关系。这种思维方式的形成是这门数据结构与算法课给我们的最大启发之一。这次的大作业是我们利用这种思路所做的尝试，相信这样的收获会对我们日后的学习起着关键的作用。

2 程序代码说明

2.1 数据结构说明

本程序最大的特点是以参数 mode 的切换作为执行力来源，并未采取任何特殊的数据结构。

2.2 函数说明

Load 函数中，directions 首先对四个方向的引用进行准备；dist 判断敌我之间的安全距离。另外添加了一些具有主要功能的函数

1.attack 函数：所处位置小范围攻击函数，判断所处位置上下左右的局势，如果恰好附近有正在移动的敌方纸带，则选择拐弯进行攻击。

2.wander 函数：领地内游走函数，判断圈地起点，且保证纸卷在 x 轴和 y 轴方向上不出界。当沿原方向继续行走一步的位置与赛场边缘的距离不大于 1，引用 goback 函数，并随机返回左或右，防止触碰边界。当与敌人位置过近时，引用 attack 函数。进入非我方领地时，引用 square 函数进行圈地。

3.square 函数：领地外圈地函数，使用与 wander 相似的原理同样保证纸卷不会出界。进入我方领地时，引用 wander 函数。当模式从 wander 切换至 square 时，count 初始值

设为 3。maxl 实时检测敌我之间的距离。每次引用 square 函数时 count 会增加 1，当 count 小于 maxl（由 dist 函数实现其功能）保持直行，而当 count 递增到大于 maxl 时，将 count 归零并随机进行一次左转或右转；从我方领地出发后第一次的转向方向是随机的，并且被 turn 记录；随后在非我方领地的每一次转向都与第一次相同，以实现方形的圈地。

4.goback 函数：wander 模式中即将触碰赛场边界时，返回我方领地中心。首先随机转向，然后判断是否进入非我方领地，若是，引用 square 函数，count 取 1,2,3 中随机一个，可以使圈地范围大小有所变化。若下一步仍然在我方领域，则在前进几步（依赖于此时的 count）后，继续引用 wander。

5.dist 函数：采用 max 函数，在 2 和（敌我 x 间距与敌我 y 间距之和的一半）之间取更大值，在 square 函数中与圈地直径有关。当距离过近（横坐标或纵坐标相同，最大距离为 2；横坐标纵坐标皆不同，最大距离为根号 2），会触发一次随机转向。避免碰撞。

2.3 程序限制

多次测试表明，一般情况下程序不会出错，也不会有自杀情况出现。但自杀确实可能发生，在赛场边界的强制转向可能导致 square 失败，在方形圈地未完成前产生错误的转向，从而产生撞向自己纸带的可能。当对手占据我方大量领土导致我方纸卷丧失与本有的片状领土区域的连接时，非常有可能发生自己撞向自己（首尾相接）的自杀。

3 实验结果

3.1 测试数据

实验环境说明：

1. 硬件配置：（CPU/内存）Intel(R)Core™ i5-4210U CPU@ 1.70GHz / RAM=4.00GB

2. 操作系统：（名称/版本）Windows 10 家庭中文版

3.Python 版本：（版本号）Python3.6

测试时，为充分辨认本代码是否存在 bug 或自我毁灭倾向，在给出的 AI 样例中选择 simple goround 作为对手（绝对跑够 2000 局，且基本不会对我方进行过多干扰）；此

外，也尝试了自我对决，以更好地探索算法存在的缺陷。在 solo 文件开头写入“from match_core import match”，运行进行比拼。

3.2 结果分析

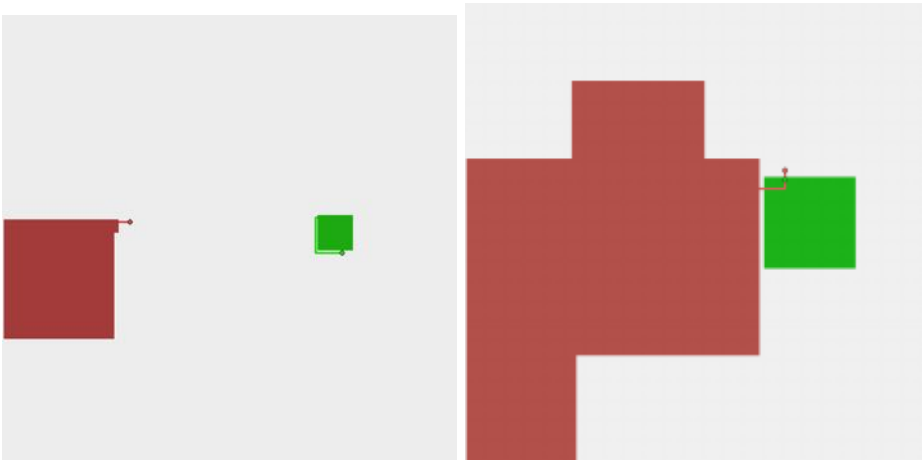


图 3. 与 simple goround 对决结果

首先尝试与 simple goround 对决。图 3A 为比赛进行约 30s 的局势，图 3B 是双方 4000 局回合结束时的局势。可以看出，在比赛开始初期，本程序的圈地能力还是比较有竞争力的，主要归功于我们的圈地策略，圈地的直径相当大，而且集中在我方出发的半场，安全系数也比较高（除非遇到强烈攻击型的对手）。但我们发现，不考虑对方侵占的情况下，随着回合数增多，纸卷明显有很多步都浪费在我方领地之中；虽然这样可以避免地方对我方领地的侵占，但还是有降低圈地效率的可能。由于本算法复杂度相对较低，在运行时间上基本上满足要求；主要的运行时间可能发生在 count 的累加和判断过程中。

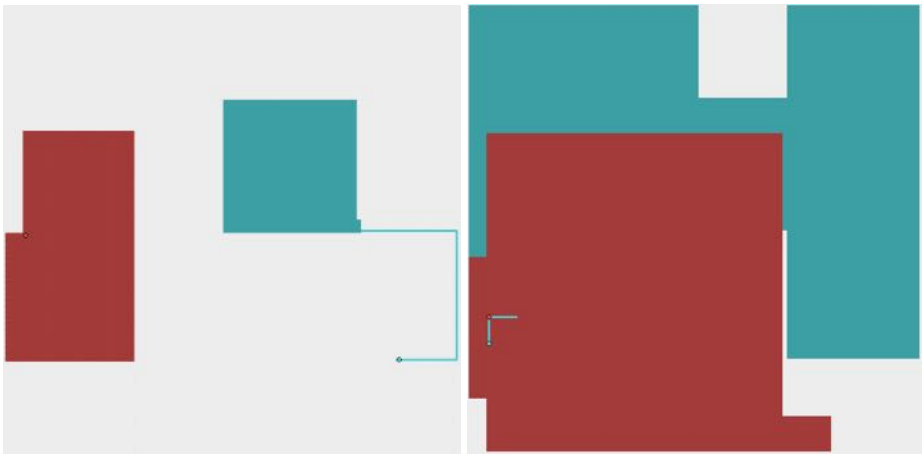


图 3. 自我对决结果

然后尝试了自我对决（图 3）。大多数情况下，二者不到 2000 回合就可决出胜负（图 3B），最少可以在约第 80 回合即结束比赛（图 3A）。输的原因大多是被对方撞击纸带，这暴露了本算法最大的缺陷——不考虑敌方纸卷位置，进行模式太过于单一和固定的方形圈地。

4 实习过程总结

4.1 分工与合作

期末分组大作业发布之后，成员通过自愿报名以及招募的方式初步形成了当前的小组，小组成员推选拉毛切忠同学担任组长。考虑到成员阅读理解作业要求以及代码样例需要不同的时间，以及方便成员们随时交流自己的新想法，在小组讨论的初期，小组讨论的形式是以微信线上讨论为主，由组长确定讨论的话题，成员们随时添加补充意见。在对作业的要求有了整体的了解以及对于实现的方法有了初步的思路之后，组长决定于 6 月 3 日进行一次线下讨论，地点定在理教一楼讨论区，参加的成员包括组长拉毛切忠、组员刘昕玥、组员李顺吾，组员王子璇同学因故于 6 月 2 日晚与组长单独进行了讨论。在讨论的过程中，每位组员都积极与其他成员分享自己的想法和意见，通过讨论，小组成员们明确了作业的任务、代码的设计思路，同时经过交流，在小组的努力下，每位成员在自己研究代码时遇到的困难和困惑也基本得到了解决。本次讨论的结果是，小组明确了为了实现游戏规则设定的要求，游戏代码需要考虑场地中的不同情况，根据场地中敌我双方的布局以及运动方

向，设计不同的方案加以应对，例如进攻、撤退、圈地等。组长将这些情况加以分类，分配给每位组员考虑实现方法。

在第一次线下讨论之后，小组成员按照组长安排的工作各自落，其间的讨论主要是微信线上讨论为主。6 月 9 日，组长拉毛切忠组员刘昕玥进行了一次线下讨论，同时与李顺吾同学进行线上讨论，在这次讨论中，两位同学汇总了从各个渠道收集的代码资料和思路，对于即将编写的代码建立了初步框架。6 月 10 日，组长拉毛切忠和组员王子璇对代码作业进行了最终的修改与完善，形成了最终版本的作业代码，希望参考样例给予的思路，在局部范围内实现小规模的攻击。由于除了李顺吾同学外，其他三位同学寝室邻近，因此在第一次小组碰面确定主要的思路后，大多数时间都是采取三人线下讨论同时与李顺吾同学线上讨论的方式。图 4 是拉毛切忠、刘昕玥、王子璇线下讨论，同时与李顺吾同学线上讨论的照片。

图 4. 小组讨论记录

小组的分工为：组长拉毛切忠负责组织讨论、分配工作、草拟部分代码、总结代码、撰写实验报告中部分程序说明、汇总并修改完善实验报告；此外全体成员负责阅读理解代码要求、构思代码并参与讨论：组员李顺吾负责撰写实验报告的算法思路部分；组员王子璇负责撰写实验报告的部分程序、实验结果；组员刘昕玥负责撰写实验报告的实验过程总

结部分。

4.2 经验与教训

由于小组四位成员都不算是精通程序设计的大牛级成员，所以在设计、编写代码的过程中稍显吃力，但是通过这次分组实习大作业，小组将每位成员的想法汇合起来，完成了一位同学很难完成的任务。在分工以及讨论的过程中，增进了成员们对于课上所学习知识的理解程度以及应用能力，小组成员之间互相的答疑解惑，也大大提高了小组的工作效率。此外，通过本次实习作业，小组成员认识到，想要编写出强大的代码，需要更深的算法与数据结构的基础知识，这部分是需要小组成员今后加强的部分。

4.3 建议与设想

本次实习作业将课堂上学习到的有关算法与数据结构的基础知识应用到游戏开发中，通过这样的实习作业，加深了小组成员对于课堂内容的理解和掌握，将不同章节的知识加以融合，达到融会贯通的效果。在实习作业的设置方面，希望能提供不同难度的实习作业供不同知识水平的同学们进行选择，并根据其难度在分数上也设置分层；对于基础知识相对薄弱的同学，可以对作业中可以用到的数据结构进行提示，像本次作业这样完全开放式的题目，如果对于数据结构的掌握不能做到足够熟练，在程序设计中可以用到哪些数据结构是不容易想到的，这样有可能降低同学实际的参与程度。

总之，通过这次大作业以及整个学期的课程，同学们都能感受到老师在课堂内容设计上花费了很多心血，相信这样的授课模式在以后也会让更多的学弟学妹们收益。

5 致谢

感谢陈斌老师一学期以来有趣的授课以及微信群中积极解答我们的问题，谢谢各位助教平时在微信群以及 mooc 上对我们问题的解答，谢谢各位老师、油聚以及其余各位技术组大佬同学为我们这次大作业付出的心血，感谢技术组同学创建并不断更新完全的基础设施代码，以及提供到网站上详细的基础设施描述和一些示例。感谢我们组中每一位邻近毕业而且非常忙碌的四位同学互相的理解和积极的合作。拥有了这一切，才让我们体会到让

想法可视化的有趣之处，有机会窥得些许算法的魅力，让我们对另一个领域有了全新的认识和兴趣。

6 参考文献

https://github.com/chbpku/paper.io.sessdsa

第二十四章　N17_Zulu 报告

周竞宇 * 燕国智 宋生升 赵子溢 曾涛涛 卢明皓

摘要：本组主要采用的数据结构有列表、字典、集合、元组等，利用条件判断、循环等控制流语句，使用了类似于广度优先搜索的递归算法。总体思路是保守圈地，保证自身安全，在可以保证杀死对手的情况下主动出击。该策略取得了不错的效果，模拟测试时全胜，比赛时进入八强。

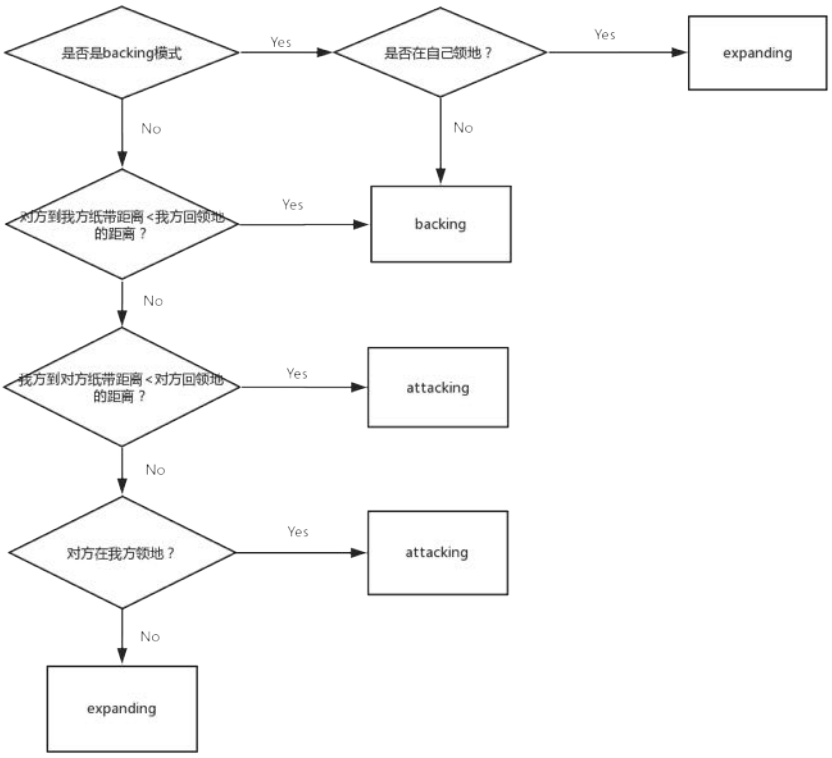
关键字：列表；字典；集合；元组；条件判断；循环语句；递归

1 算法思想

1.1 总体思路

总体的策略：在保证自身安全的情况下圈地，如果发现进攻机会，则发动进攻。关键是各种距离条件的判断，纸带头之间的距离可以利用坐标简单计算得到，卷头到领地的距离由递归算法得到。具体运行可分为三个板块：扩张 expanding、返回 backing 和进攻 attacking，使用的数据结构为列表、集合、元组、字典。

1.2 算法流程图



1.3 算法运行时间复杂度分析

计算纸卷头到领地/纸带的最小距离时，采取了递归算法，因此这一步骤是时间消耗的大头。在每次递归中，有两个 for 循环，每个 for 循环里包含两个赋值语句和两个条件判断语句。经过 dist 次递归，总的时间复杂度为 O(n2)。

2 程序代码说明

2.1 数据结构说明

主要用到的数据结构有列表、字典、集合、元组等，也用到 while、for、if 等控制流语句，在判断最小距离时使用了类似于广度优先搜索的算法，使用递归实现。由于整体思路

较为简单，并没有自定义类以及对课程上介绍的线性、树、图等基本数据结构进行扩展和改进，使用最基本的数据结构即可实现我们的想法。

2.2 函数说明

distant_field: 计算纸卷头到领地/纸带的最小距离的函数，以递归方式实现，从距离中心 0 开始算起。

center = me 或 enemy，表示以哪一方纸卷头为中心

typeOfField = fields 或 bands，表示计算距离的目标是领地还是纸带

owner = me 或 enemy，表示目标领地/纸带的归属

dist: int，表示本次递归计算距离纸卷头 dist 远处是否有目标领地/纸带

positionSet：集合，用来储存距离中心 dist 远的坐标值

在某 dist 下找到目标，则返回当前距离。寻找 bands 时最大 dist 为到敌方纸带头的距离。超出最大可能距离（场地长 + 宽）时，返回 None。该距离下无目标领地/纸带，将 dist+1 后进行下一轮搜索。

distant_player：计算双方玩家纸卷头的距离，直接读取双方坐标，此距离等于双方玩家纸卷头横坐标之差的绝对值加上纵坐标之差的绝对值。

expanding：扩张模式。如果当前纸卷头在我方领地内，则寻找最近的空白领地进行扩张；领地外扩张时，一般会一直前进，当前进距离较大（多于场地宽高 1/3）时转弯。第二次转弯和第一次转弯方向相同，之后切换为 backing 模式。

backing：返回模式。从当前位置回到我方领地，不管其他情况，即返回模式的优先级是最高的。返回领地后切换回 expanding 模式。

attacking：攻击模式，和返回模式基本一样，区别只是目标方位（target）不同。backing 的目标为最近的我方领地，attacking 的目标为距离自己最近的地方纸带的坐标。

navigation：导航函数，输入目标方位 target，使得纸卷头向该方位前进。该函数作为辅助函数，以方便各种模式的实现。首先计算不转弯时下一步的坐标。若下一步 y 坐标在两点的 y 坐标范围外，则转弯，转弯方向视两点 x 坐标相对位置而定，当 x 和目标点的 x 一样时，向对方纸带方位的反方向转弯。下一步 x 坐标在范围外，转弯方向视两点 y 坐标

相对位置而定，当 y 和目标点的 y 一样时，向对方纸带方位的反方向转弯。若下一步 x、y 坐标在目前坐标和目标坐标之间，则不转弯。

secure_safety：安全确保函数，防止撞墙或撞纸带自杀。

2.3 程序限制

由于计算距离时采用递归算法，且从距离为 1 开始递归，因此当棋盘大小太大时会导致计算量剧增，耗时很长，甚至可能卡死。因此该程序在棋盘不太大的情况下表现相对较好。

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)_m3-6Y30_CPU_@_0.90GHz，RAM 4.00GB

操作系统：Windows 10

Python 版本：Python 3.7

测试方法：利用技术组提供的 visualize.py 工具与若干个 AI 进行对战，以及用 glory_of_minkind.py 让该程序和自己进行对战。测试结果如下标所示：

表 1. n17_zulu vs AI 结果统计

AI	获胜次数/测试次数
AI_dumb_goround	10/10
AI_dumb_random	10/10
AI_normal_wanderer	10/10
AI_random_2	10/10
AI_random_3	10/10
AI_simple_goround	10/10

AI_simple_wanderer	10/10

表 2. n17_zulu vs 人类玩家

对战局数	胜利方	胜利原因
1	zulu	撞击对手纸带
2	zulu	撞击对手纸带
3	zulu	人类撞纸带自杀
4	zulu	撞击对手纸带
5	zulu	撞击对手纸带
6	zulu	撞击对手纸带
7	zulu	撞击对手纸带
8	zulu	撞击对手纸带
9	zulu	撞击对手纸带
10	zulu	撞击对手纸带

测试结果显示出该程序还是有很强的强度的。尤其是对战人类玩家时，显示出其明显的优势：当它计算出人类玩家有进攻意图或者自身安全受到威胁时，会马上进行防守回到自己领地，这使得玩家对其发动进攻基本都是无效的；而它又能利用住人类玩家的失误，一旦发现有进攻机会便立刻进攻，绝不失手。而自身程序更是保证了自己不会犯人类可能犯的如撞到自身纸带的这类低级错误，因此在于人类玩家对战时展示出了超高胜率。

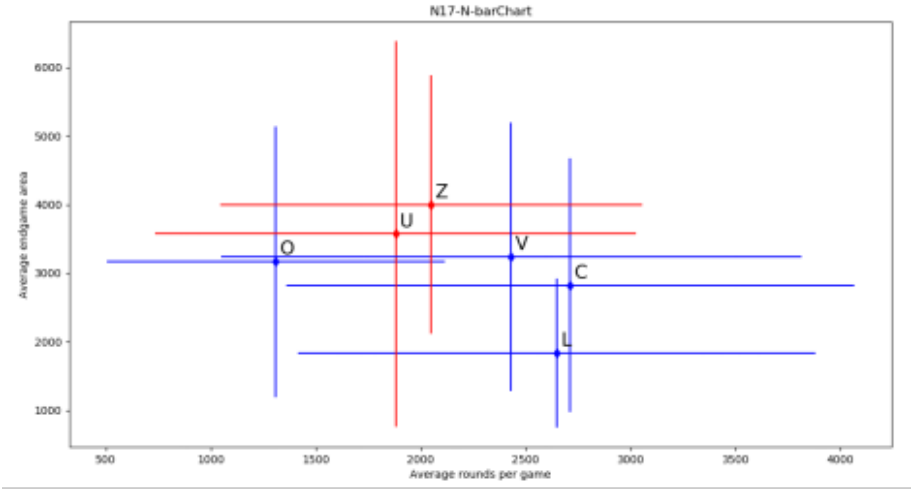
3.2 结果分析

由于一直对代码进行完善和改进，我们小组错过了热身赛。在正式比赛中，我方被分配到 N17-N 小组。在小组赛中，我方 Zulu 仅败给 Uniform 组，最终获得了 12 分，位列小组第 2，涉险出线。详细对战结果如下：

表 3. 小组赛对战结果

Status:							

C	-C-	+	0	-	-	-	4
L	-	-L-	-	-	-	-	0
O	0	+	-O-	-	+	-	7
U	+	+	+	-U-	+	+	15
V	+	+	-	-	-V-	-	6
Z	+	+	+	-	+	-Z-	12
Ranking:							
uniform	15						
zulu	12						
oscar	7						
victor	6						
charlie	4						
lina	0						



小组赛结果

从该坐标图中可以看出，我方平均圈地面积是最高的。由于策略相对保守，尽管圈地效率不高，但只要能活下来就能累积出较大的面积。尽管如此，我们的步数在小组中居中，这也是因为我们有终结比赛的手段：一旦发现对方防守空缺，立刻发起进攻，结束比赛。从小组赛结果来看，我方策略取得了不错的效果。

在淘汰赛中不敌 Papa，完败出局。Papa 最终获得亚军，实力强劲，事后分析他们代码确实比我们完善许多。

表 4. 淘汰赛结果（vs Papa）

No.	对战结果	原因
1	lose	对手撞击纸带
2	lose	对手撞击纸带
3	lose	对手撞击纸带
4	lose	对手撞击纸带
5	lose	对手撞击纸带
6	lose	对手撞击纸带
7	lose	对手撞击纸带
8	lose	对手撞击纸带
9	lose	对手撞击纸带
10	lose	对手撞击纸带
11	lose	对手撞击纸带

4 实习过程总结

4.1 分工与合作

小组分工：

周竞宇（组长）：负责组织例会和线上讨论、会议记录；负责代码的测试环节；负责报告的撰写；负责比赛名牌制作。

曾涛涛：负责 navigation 函数的编写。

赵子溢、宋生升：负责三个模块的程序，以及整体策略的实现。

卢明皓：负责距离判断函数的编写、play 函数和 load 函数编写、secure_safety 函数编写、代码的汇总、校正和优化。

燕国智：提供思路。

小组讨论的方式分为线上和线下。线上讨论是由组长牵头，在微信群中进行交流，主要解决细节问题和具体操作中的磨合和交接。线下讨论是由组长组织例会，集中进行面对面讨论，主要解决策略、代码框架、分工等大问题。

一共组织了 3 次例会，地点均为二教麦隆咖啡旁：

第一次例会时间：2018.5.30 参会人员：周竞宇、赵子溢、曾涛涛

会议内容：

讨论拟采用策略：首先对对手上一步操作进行判断，对手是倾向于选择圈地还是进攻；如果判断对手会圈地，则判断能不能干掉他。如果可以，选择进攻策略；如果不能，选择圈地策略。如果判断对手会进攻，则选择防守策略。

第二次例会时间：2018.6.1 参会人员：周竞宇、赵子溢、曾涛涛、宋生升、卢明皓

会议内容：

再次讨论策略，最终决定的策略为保守圈地 + 适时进攻。当发现对方有可能击杀我们时，立刻返回领地保证安全；如果发现可以击杀对手，则主动撞击对方纸带。

对工作进行拆解和分工。

第三次例会时间：2018.6.10 参会人员：曾涛涛、宋生升、赵子溢

会议内容：

各个模块的具体实现以及整体策略的实现，在什么条件下采取和种策略。

讨论场景

4.2 经验与教训

本次实习过程中，我们小组分工合理、配合默契，最后取得了还算令人满意的成果。在大家非常忙碌的学业生活之余，还要抽出时间进行团队合作，实属一个很大的挑战。我们组在讨论时进行了充分的可行性论证：因为我们组都是一些编程小白，大家都是因为数算课才开始编程，没有大佬加持的我们放弃了很多看上去很美好的思路。而且有些情况我们尽管有考虑到，但由于其涉及到很复杂的情形，可能会带来很大的工作量，我们也选择了战略性放弃。事实证明，这种牺牲强度换取效率的方法是非常重要的，因为尽管我们组员都很努力，但囿于能力有限，在最后关头才把完整代码搞出来。

但这也成了我们组的一个遗憾。有很多美好的想法没有得以实现。最大的教训是组队时一定要拉一个大佬。

4.3 建议与设想

在组队上，由于大家背景不同，实力也是参差不齐，大佬集群之后小白们很难取得进展，因此建议把大佬们拆散开。竞赛方面需要改进的是时间把控，尽管最后的比赛非常有趣，但由于很多不必要的环节导致两节课内比赛没有进行完。

对选修明年课程的学弟学妹们，我们想说：请好好对待这个数算实习，这可能是你为数不多的可以学以致用的机会，更何况这个实习非常有趣味性，请一定要好好享受这个过程！

实习作业后续设想：将游戏做得更精美，保存优秀小组的代码，让以后选修此课的同学可以来挑战。

5 致谢

感谢技术组为大家搭建平台、提供信息；感谢陈斌老师在微信群里的答疑。

6 参考文献

[1]SESSDSA’18 课件

[2] 北京大学地空学院数据结构与算法 SESSDSA 内部教材

[3] `https://github.com/chbpku/paper.io.sessdsa`