SESSDSA 2018 实习作业
paper.io.sesseda 报告合集
第二册

陈斌、陈天翔、张赖和

目录

第三部分　F17 联盟各参赛队报告 **5**

第一章　F17_007 报告 **7**

第二章　F17_Alpha 报告 **15**

第三章　F17_Bravo 组报告 **37**

第四章　F17_Charlie 报告 **73**

第五章　F17_Delta 报告 **91**

第六章　F17_Echo 报告 **103**

第七章　F17_Foxtrot 报告 **123**

第八章　F17_Golf 报告 **147**

第九章　F17_Hotel 报告 **177**

第十章　F17_India 报告 **185**

第十一章　F17_Juliet 报告 **209**

第十二章　F17_KizunaAI 报告 **243**

第十三章 F17_Lima 报告 **259**

第十四章 F17_Menhera 报告 **271**

第十五章 F17_November 报告 **293**

第十六章 F17_Oscar 报告 **303**

第十七章 F17_Papa 报告 **309**

第十八章 F17_Quebec 报告 **327**

第十九章 F17_Romeo 报告 **349**

第二十章 F17_Tango 报告 **365**

第二十一章 F17_Uniform 报告 **381**

第二十二章 F17_Victor 报告 **411**

第二十三章 F17_Whiskey 报告 **433**

第二十四章 F17_X-ray 报告 **443**

第二十五章 F17_Yankee 报告 **481**

第二十六章 F17_Zulu 报告 **491**

第三部分

F17 联盟各参赛队报告

第一章　F17_007 报告

刘一鸣 *、朱玉源、郭新年、揭宇

摘要：这次我们采用了近乎全保守性的战略，只有在与敌人非常接近并且有必胜把握的情况下才主动出击。总的来看，该算法比较稳定，极少出现自杀或报错等问题，但过于保守的策略加上并不完美的圈地计划让最后的结果并不尽如人意。

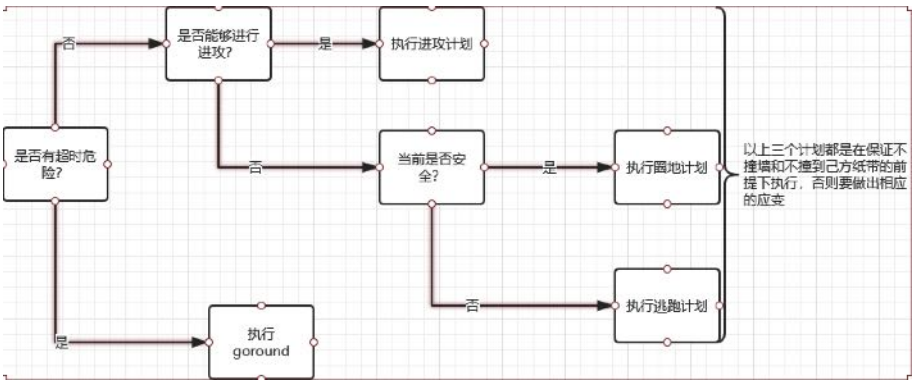
关键词：保守性、攻击函数、最小距离

1 算法思想

1.1 总体思路

这次我们小组的 paper.io 主要采取了近乎纯保守圈地的策略。主要是见到对方如果不是我们所考虑到的特殊类型，就一定要迅速跑回己方领地。所以我们设计了四个参量：圈地计划、逃跑计划、安全距离（己方下一步位置能够逃回己方领地的最小距离）、最小距离（己方下一步离对手可能的下一步的最短距离）。总体的算法思路是这样的：在保证不撞墙以及纸带的情况下，先判断能否进行攻击，如果不能的话：如果安全（安全距离 < 最小距离），执行圈地计划，否则执行逃跑计划。另外，为了防止可能出现的超时错误，我们小组设计了对时间的判断，如果时间所剩不多（小于五秒）开始执行四处游走的简单 AI。（事后证明这一担心完全是多余的）

1.2 算法流程图



1.3 算法运行时间

在开始的程序中，我们担心由于遍历导致的超时，在实际运行中也确实出现了这样的情况。但是后来组长发现这实际上并不是遍历导致的超时，而是由于在 load 函数中定义了列表 myband 和 myspace，从而每一轮之后这个列表都会扩大，这样便浪费了很多无谓的时间，再加上我们组的策略是保守性的，基本是靠消耗完回合数比较圈地面积去取胜，这样的时间浪费累加起来便导致了糟糕的超时。在 load 函数中删除了这两个列表之后程序的速度得到了明显的增加，再也不会出现之前的几乎每次都要超时启用备用方案的情况。

2 程序代码说明

2.1 数据结构说明

我们组在算法中并没有运用很多的数据结构。我认为我们组主要运用的工具就是列表和元组。利用列表来存储各种信息，利用元组来存储坐标。在圈地计划和逃跑计划的设计中可以说利用了栈的思想。就是每次的操作相当于栈顶元素的出栈，如果安全的话就继续出栈，如果不安全的话就将这个栈整个清空，执行逃跑栈中的操作，直到回到自己的领地。之后将新的操作依次加入圈地栈中。但是我们组只是利用了这样的思想，并没有实际上去运用 ADT STACK 数据类型完成我们的这些操作。

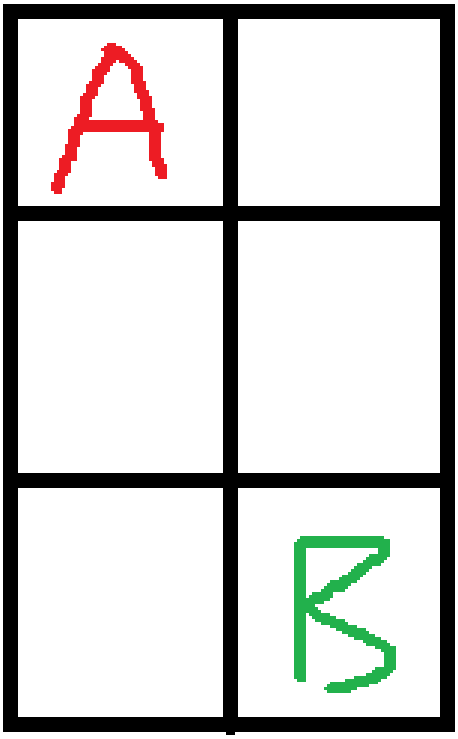
2.2 函数说明

我们小组的 load 函数非常简单，只有一个列表来规定各种方向的坐标加减关系。

Attack1 和 attack2 以及 gameover 函数是揭宇同学的功劳，这几个函数只有一个简单的接受参数 stat（=stat[‘now’]），目的是完成那必须出击击杀对方的致命一击。是一套组合拳。下面用简单的示意图来表示一下（注：在下面的图中 A 代表我方的纸带头（纸卷），B 代表敌方的。）

函数 attack1 接收初始游戏信息 stat，主要功能为判断是否进行攻击，并返回进行攻击的第一步的前进方向。该算法先判断双方玩家位置是否构成“日”字，若构成“日”字且敌方玩家无法迅速跑回敌方领地，则根据初始游戏信息判断前进方向，使双方玩家位置是否构成“田”字。

Attack1 函数是攻击的第一步，它实际上基于如下图所示的简单模型：

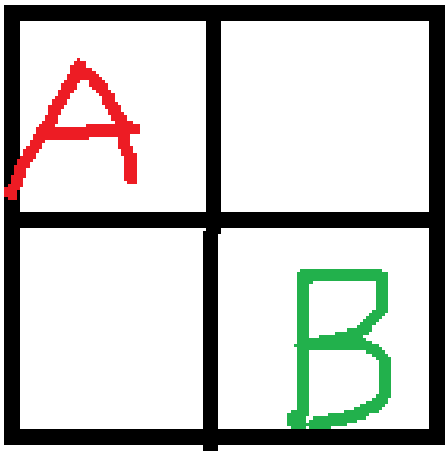


在这种情况下，分析得到只要敌人离“老巢”不是太近，就应该进逼。

函数 attack2 接收初始游戏信息 stat，主要功能为返回进攻第二步的前进方向。函数

attack2 是继函数 attack1 之后的第二步，该算法先判断双方玩家位置是否位于边缘，若位于边缘且敌方玩家无法迅速跑回敌方领地，则根据初始游戏信息判断前进方向使双方玩家位置构成“田”字；若不是位于边缘，则直行接近敌方纸带。

Attack2 函数是攻击的第二步，它是基于如下图所示的模型：



在这种情况下我们应该继续进逼。

函数 gameover 接收初始游戏信息 stat，主要功能为判断前进方向并撞死对方。函数 gameover 是继函数 attack1 和 attack2 之后的第三步，该算法主要是根据初始游戏信息判断敌方玩家或敌方纸带的位置，并据此返回我方玩家的前进方向，做到撞死对方。

Gameover 函数是击杀的一步，当然可以推断出是下面的这种情况：

函数 mindistance 接收我方玩家信息，函数存储 storage 以及我方玩家前进方向，主要功能为返回沿此方向前进后我方玩家距我方领地的最小距离，辅助我方玩家快速逃回我方领地。该算法先计算我方玩家下一步所处的位置，然后遍历我方领地，计算该位置到我方领地的最小距离。

2.3 程序限制

我们小组的程序基本上避免了撞纸带和报错的情况。但是在进入对方领地后由于慌不择路，如果之前已经拐过弯，那么在回去的时候可能会撞击自己的纸带，不过在这种情况在我们的多次测试中未曾发生过，只是有理论上的可能性。

我们的程序从理论上讲应该是不会被对手撞击纸带的，但是从实际上讲这种情况仍然有所出现。原因在于在圈地和逃跑计划中敌人的突然变化可能会截断我们的退路。原因在于逃跑过程中想避开自己的纸带，不得不多次“绕远”，这就给了对手“截击”的机会。

3 实验结果

3.1 测试数据 & 结果分析

实验中采用实战的方法来对算法进行测试。

该算法与简单的 AI 算法对战时，基本全胜。在对战中，攻击策略基本不发挥作用。相较之下，防撞墙策略，防撞纸带策略以及圈地策略发挥了主要作用。这些对战大部分以敌方自杀或双方步数用完我方领地较多结束。

该算法与遍历算法对战时，其中遍历算法是我们组编写的早期 AI 算法，对战结果 6：1，新算法胜利。其中新算法的防撞墙策略和防撞纸带策略发挥了主要作用，这两个策略是算法避免了墙自杀失败或撞纸带自杀失败的情况，而遍历算法则有两次撞到自己的纸带而导致死亡的情况，相较之下可以看出改进后的防撞纸带策略效果很好。除此之外，逃跑策略和圈地策略也发挥了作用，避免了相撞的情况发生。

该算法与算法本身进行对战时，对战结果为平局。因为双方算法都是保守型算法，故对局过程中起主要作用的是圈地策略，大多数的对战结果是双方步数用尽，圈地多的一方获胜。算法的逃跑策略也发挥了作用，避免了与对方的攻击。在少数对局中，攻击策略发挥作用，造成一方被撞失败的情况。在所有对局中，防撞墙策略和防撞纸带策略始终发挥了良好的作用，无撞墙自杀失败或撞纸带自杀失败的情况发生。

从以上对局情况来看，算法在运行时间上在预期之中，主要的运行时间开销发生判断我方是否安全以及逃跑计划中判断前进方向的环节。

4 实习过程总结

4.1 分工与合作

在本次实习过程中，我们组可谓经历了众多困难。首先，没有大佬的腿可抱（自嘲为弱鸡一群），因而对于算法设计，程序编写来说，全部需要从基础做起。其次，之前从未有过这种经历，面对一个陌生的课题还是有点难以下手。针对这些问题，我们还是有计可循，精细的分工和团结协作发挥了重要的作用。

首先，我们分析了组员的特点特长，针对这些特长然后具体分配任务。对于算法的设计，群员一起讨论，大家共同探讨可行的算法。而对于程序的编写，则能者居重位，下面是具体的任务安排：

刘一鸣（组长）：负责核心代码的编写，主体函数的设计（具体为逃跑计划等），代码的整合；

郭新年：最小距离和圈地计划部分代码的编写；

朱玉源：函数的调试，实习报告的编写，以及拍照等杂事；

揭宇：进攻函数的编写；

由于组员为一个班的同学组成，这为讨论带来了方便。就不需要像有些组线上交流一样麻烦，我们基本面对面地直接讨论。

4.2 经验与教训

三个臭皮匠顶个诸葛亮：完成此大作业不必把筹码押在大佬身上，不能一味地求抱巨腿。虽然我们个人能力有限，但还是可以完成这项任务。从中我们收获的是合作协作的精神，同时也锻炼了自己与他人交流的本领，磨砺了清晰表达自己的观点的能力。

不必注重结果，偏重过程中的收获：从模拟赛中我们就清楚自己的代码与其他组还是存在差距的，尽管我们进行了一些改进，但还是输多赢少。但我们并没有灰心，希望决赛可以改进得更好吧。

4.3 反思

我们在思想上过于保守，由于畏惧对手和怀疑自我能力。在算法上采取保守的做法，并没有采取主动攻击敌人的方法，而是防卫为主。后来的事实证明，这种做法是比较不成功的。不要追求所谓的高级算法，自己摸索的更适用，当初我们去搜寻一些奇怪的算法，不仅没能驾驭，而且最后导致程序无法运行。后来还是采用自己熟悉的函数算法，才得以继续。

4.4 感想

虽然自己没有在这项任务中发挥核心作用（自己确实很弱），但还是努力配合队友们完成任务，查找资料，编写报告等，但经历这一过程，从队友那学会了很多东西。

——朱玉源

进攻代码一开始是没有思路的，但在组员们的帮助下，一点一滴的弄明白，一丝一缕的码出来，学到了很多，也明白了很多，可就是说不出来，就那么感觉到了。

——揭宇

在编写代码的过程中，我学到了很多，也遇到过困难，但在组员的帮助下，我最终完成了我的代码。在这个过程中，我不仅学到数据结构与算法的相关知识，还加深了我与组员的友谊。我要感谢老师和助教的付出，感谢 CCTV，感谢党和国家，感谢。

——郭新年

我这次有幸成为小组的组长。从开始的算法设计到最后的代码调试，我的组员们和我都一起完成了许多在原来看来非常困难的事情。我从这次的大作业中收获了许多，也明白了讨论交流，团队合作的重要性。虽然成绩并不是十分的尽如人意，但我认为我们小组的每一个人都尽了最大的努力，也都从中获益良多。感谢老师和这门课程提供的这次机会！

——刘一鸣

5 致谢

- 首先感谢陈斌老师这学期的辛苦付出；
- 其次感谢一直陪伴我们的助教学长和助教学姐；
- 然后感谢选此课的每一位同学的陪伴；
- 最后感谢队友们的辛苦努力；

第二章　F17_Alpha 报告

周子楠 *、任和、林小靖、丁泽宇、李博、蔡翔远

摘要：算法以保守圈地的策略为蓝本，添加 BFS 机制对于抽象出的“最短路径”问题进行搜索。代码深刻地体现了贪心算法、图算法等的实际应用，对于同为保守策略的代码和偏激进的代码都有很好的应对。

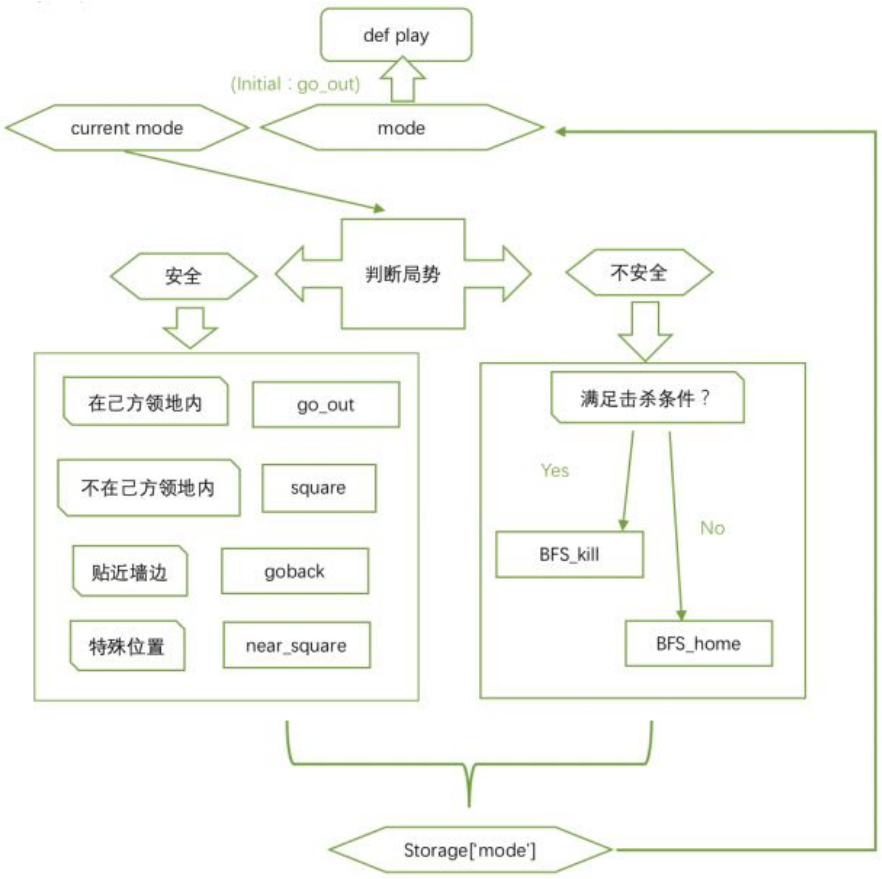
关键字：贪心算法，图算法，广度优先搜索，BFS

1 算法思想

1.1 总体思路

对场上战局进行分析，根据不同情况选择调用不同模式 (mode)。所有 mode 储存在 load 函数里，通过修改 storage 里的值来切换 mode。play 函数返回当前 mode 返回值。

1.2 算法流程图



1.3 算法复杂度

主要在于启动 BFS 算法以后耗时增加。主要部分的时间复杂度和 BFS 一致，为 O(V+E)。其中图中节点数大致等于比赛区域格点总数，边数大致等于节点数的 4 倍。BFS 本身性能较好，但由于此情况下节点数多，边数多（10^4），此算法成为代码中的主要时间开销部分。

2 程序代码说明

2.1 BFS 算法说明

BFS 算法是本程序的核心算法。其所采用的 Graph,Vertex 类，接口及实现都和教材提供的 pythonds.graphs.adjGraph 一致。（删去了未使用的其他接口）

我们注意到，在设计 AI 的过程中会频繁遇到“己方攻击对方纸带的最短距离“、“己方返回己方领地的最短距离”以及“对方攻击己方纸带的最短距离”一类的距离问题。以己方攻击对方纸带的最短距离为例，需要考虑走的路径不能越过己方纸带。上述问题可以普遍地抽象为“从指定位置绕过指定障碍物区域，到达指定目标区域的最短距离”问题。

而我们敏锐地注意到这一问题和课程教材中 Word Ladder 问题的相似性，决定采用广度优先搜索算法来解决问题。目标是返回这些最短距离，同时返回对应的路径。下对这部分代码进行分析。

buildGraph 函数

```
def buildGraph(obstacle_area, obstacle_num, current_pos):

    """buildGraph建立起所有可能的节点和可行的边"""

    loc = {}  # 存储节点在地图上位置的dict
    path_graph = Graph()  # 存储节点本身的图
    for i in range(stat['size'][0]):
        for j in range(stat['size'][1]):
            if obstacle_area[i][j] != obstacle_num:
                #  在可进入区域（除开自己纸带）建立节点
                path_graph.addVertex((i, j))  # 节点加入图中
    path_graph.addVertex(current_pos)  # 当前位置需要手动加入
    for v in path_graph:
        loc[v.id] = v  # 建立节点所在位置（tuple）到节点的映射
    for location, v in loc.items():
        c1 = (location[0] + 1, location[1])
        c2 = (location[0], location[1] + 1)
        c3 = (location[0] - 1, location[1])
        c4 = (location[0], location[1] - 1)
        for c in [c1, c2, c3, c4]:
            if c in loc:
                path_graph.addEdge(location, c)  # 在图中且位置相邻的两点建立边
    # 节点查看测试工具，勿删
    # for vertex in path_graph:
    #     print(vertex)
    return path_graph, loc[current_pos]  # 返回图和起始位置的节点
```

接收障碍物区域、障碍物标号（己方、对方）和当前位置，返回路径图和当前位置在图中对应的节点。

首先，将比赛场地除去障碍物区域的每一个格点都封装为 Vertex 并添加到图 path_graph 中，以这些格点的位置（和游戏系统的坐标一致）为节点的 id 属性。由于障碍物区域根本不存在节点，在搜索过程中得到的自然都是避开障碍物之后的路径，这就解决了“绕过障碍”的问题。

之后为节点添加边，显而易见，只有格点上相邻的位置才可以相互通达，因此将相邻格点对应的节点之间添加边。有了将实际的场地情况抽象得到的路径图 path_graph，下一部就可以编写搜索算法进行搜索了。

bfs 函数

```
def bfs(obstacle_area, obstacle_num, current_pos, target_area, target_num, saved_path, saved_distance, helper):
    """广度优先搜索算法，从图中找到从起始位置到最近的目标位置的距离，并存储下要到达该位置的路径
    helper 辅助参数：用于对方无或己方无纸带，增加现在位置，以避免搜索不出结果！"""

    path_graph, start = buildGraph(obstacle_area, obstacle_num, current_pos)  # 接收buildGraph的成果
    start.setDistance(0)
    start.setPred(None)
    vertQueue = []
    vertQueue.append(start)
    while(len(vertQueue) > 0):
        currentVert = vertQueue.pop(0)
        for nbr in currentVert.getConnections():
            if nbr.getColor() == 'white':
                nbr.setColor('gray')
                nbr.setDistance(currentVert.getDistance() + 1)
                nbr.setPred(currentVert)
                vertQueue.append(nbr)
        currentVert.setColor('black')
        if target_area[currentVert.id[0]][currentVert.id[1]] == target_num \
                or (currentVert.id[0] == helper[0] and currentVert.id[1] == helper[1]):  # 找到最近己方领地
            # 打印棋盘测试工具，勿删
            # print('My territory has been found.')
            # print(currentVert.id[0], currentVert.id[1])
            # for i in range(stat['size'][1]):
            #     for j in range(stat['size'][0]):
            #         print(stat['now']['fields'][j][i], end=' ')
            #     print('')
            min_path = []  # 存储路径到storage
            while currentVert:
                min_path.append(currentVert.getId())
                currentVert = currentVert.getPred()
            min_path.reverse()
            storage[saved_path] = min_path
            storage[saved_distance] = len(min_path)
            return storage[saved_distance]  # 短路算法，一旦发现停止搜索，返回最短距离
```

接收参数较为复杂，分别为障碍区域、障碍区域编号（己方或对方，下同）、当前位置、目标区域、目标区域编号、存储路径的位置、存储距离的位置、辅助器。之所以参数较为复杂，是后续有进一步封装的结果，最后三个参数的详情将在后面陈述。

bfs 搜索本身和课程上的例子大同小异，核心就是通过队列操作保证探索顺序，通过节点 color 属性的设置避免重复探测。但这里和课程中 Word Ladder 的 bfs 算法略有不同：Word Ladder 问题中，不光有了一个明确的起点，也有一个明确的终点，之后才来搜索其间最短的路径，而在 paper-io 的问题背景下，终点是在对最短路径的寻求过程中来发现的。因此这里设置了一个短路机制，在搜索过程中一旦发现第一个“目标区域”的节点，就退出循环，并且采用 getPred 方法回溯得到路径，存储路径和最短距离到 storage 中通过参

数 saved_path 和 saved_distance 指定的位置，并为了后续的方便同时返回最短距离。

bfs_tool_box 函数

```
def bfs_tool_box(mode, other=None, saved_path='min_path', saved_distance='min_distance'):
    """param:mode
    mode = 'back_home' 己方回到己方领地的最短路径
    mode = 'kill' 己方到达对方纸带的最短路径
    mode = 'retreat' 对方回到对方领地的最短路径
    mode = 'check' 对方到达己方纸带的最短路径
    mode = 'other' 自行给出other参数，为列表，
    给出障碍区域、障碍编号、现行位置、目标区域、目标编号和助手变量。
    other 参见上文
    saved_path 最短路径存储位置，字符串，为storage中的一项
    saved_distance 最短距离存储位置，为storage中的一项"""
    if mode == 'back_home':
        o_a = stat['now']['bands']
        o_n = my_id
        c_p = (x1, y1)
        t_a = stat['now']['fields']
        t_n = my_id
        hh = (-1, -1)  # 此时helper不起作用
    elif mode == 'kill':
        o_a = stat['now']['bands']
        o_n = my_id
        c_p = (x1, y1)
        t_a = stat['now']['bands']
        t_n = enemy_id
        hh = (x2, y2)  # helper会把对方位置添加到目标
    elif mode == 'retreat':
        o_a = stat['now']['bands']
        o_n = enemy_id
        c_p = (x2, y2)
        t_a = stat['now']['fields']
        t_n = enemy_id
        hh = (-1, -1)
    elif mode == 'check':
        o_a = stat['now']['bands']
        o_n = enemy_id
        c_p = (x2, y2)
        t_a = stat['now']['bands']
        t_n = my_id
        hh = (x1, y1)  # helper会把己方位置添加到目标
    else:
        o_a = other[0]
        o_n = other[1]
        c_p = other[2]
        t_a = other[3]
        t_n = other[4]
        hh = other[5]

    md = bfs(o_a, o_n, c_p, t_a, t_n, saved_path, saved_distance, hh)
    movelist(saved_path)
    # print(md)
    return md
```

bfs 函数的参数较为复杂，为了进一步体现逻辑层和物理层分离的“抽象”思想，方便调用者（即本组其他组员）使用，对 bfs 模块进行封装，构建了 bfs_tool_box 函数。在标准情况下使用 bfs_tool_box，只需要给出参数 mode 就可以了。bfs_tool_box 一共设置了四种预制模式：

1.back_home（归家）：己方回到己方领地

2.kill（击杀）：己方攻击对方纸带

3.retreat（败退）：对方回到对方领地

4.check（检查）：对方攻击己方纸带

只要输入相应模式，bfs_tool_box 便会为 bfs 填入相应参数。如 bfs_tool_box(‘back_home’)，则会调用起始位置为己方位置、障碍物区域为己方纸带、目标区域为己方领地的 bfs 搜索算法。如果要使用其他自定义情况下的搜索，调用者也可以通过填写‘other’模式和相应的 other 参数列表来启动 bfs。同时，在测试代码的过程中，编写者也发现了一个较隐蔽的 bug：如果对方正在对方领地中，按照游戏的设置对方没有纸带，如果这个时候使用 kill 模式进行搜索，那么 path_graph 中将不存在终点，为无效搜索，会

导致己方 AI 无规行走而撞墙。为了解决这一 bug，对 bfs 引入 helper 参数，这显然对于调用者来说是难以理解的，于是也通过 bfs_tool_box 进行设置。helper 会在 kill 模式下，将对方当前位置设置为和对方纸带区域等同的地位，在 check 模式下将自己位置设置为和自己纸带区域等同的地位。这样，在任何情况下，目标区域不为空，可以用 bfs 找到正确路径。总之，调用者使用 bfs 算法，只需要使用 bfs_tool_box，不需要对 bfs 中复杂的参数有更多的了解。

2.2 函数说明

play 函数

storage[‘mode’] 里储存当前 mode, 通过更改 storage[‘mode’] 的值触发不同函数。play 返回当前 mode 所代表的函数的返回值。

```
def play(stat, storage):
    curr_mode = storage[storage['mode']]
    field, me = stat['now']['fields'], stat['now']['me']
    storage['enemy'] = stat['now']['enemy']

    return curr_mode(field, me, storage)
```

```
    storage['go_out'] = go_out
    storage['square'] = square
    storage['goback'] = goback
    storage['near_square'] = near_square
    storage['BFS_home'] = BFS_home
    storage['BFS_kill'] = BFS_kill

    storage['mode'] = 'go_out'
```

公共工具

1. 函数 dst：计算对方纸卷与我方纸卷的最短折线距离

```
# 计算距离
def dst(me, enemy):
    return abs(enemy['x'] - me['x']) + abs(enemy['y'] - me['y'])
```

函数 dst

2. relative_dirc：判断东南西北四个绝对方向相对于我方目前前进方向的相对位置

   基本实现思路：根据 me[‘direction’] 分出四种情况，每种情况返回一个 key 为东南西北、value 为前后左右的字典。

```
#判断相对方向
def relative_dirc(me):
    if me['direction'] == 0:#east
        return {'north':'l','south':'r','west':'bk','east':' '}
    if me['direction'] == 1:#south
        return {'north':'bk','south':' ','west':'r','east':'l'}
    if me['direction'] == 2:#west
        return {'north':'r','south':'l','west':' ','east':'bk'}
    if me['direction'] == 3:#north
        return {'north':' ','south':'bk','west':'l','east':'r'}
```

函数 relative_dirc

3. away：判断我方纸卷往哪个方向拐能最大限度远离对方纸卷

基本思路：以我方纸卷为参考点建立直角坐标系，敌方纸卷可能位于四个象限。又根据我方纸卷目前的行进方向，判断往左还是往右能远离对方。分情况治之。

```
#判断远离方向
def away(field, me, storage):
    x1 = me['x']
    y1 = me['y']
    x2 = storage['enemy']['x']
    y2 = storage['enemy']['y']
    enemy_dirc=storage['enemy']['direction']
    my_dirc=me['direction']
    '''''''''''''''''
    '  2    |   1  '
    '-------------'
    '  3    |   4  '
    '       |      '
    '''''''''''''''''
    '''

    if (x1-x2)<0 and (y1-y2)>=0:#enemy section :1
        if my_dirc == 2 or my_dirc == 3:
            return 'l'
        else:
            return 'r'

    if (x1-x2)>0 and (y1-y2)>=0:#enemy section :2
        if my_dirc == 1 or my_dirc == 2:
            return 'l'
        else:
            return 'r'

    if (x1-x2)>0 and (y1-y2)<=0:#enemy section :3
        if my_dirc == 0 or my_dirc == 1:
            return 'l'
        else:
            return 'r'

    if (x1-x2)<0 and (y1-y2)<=0:#enemy section :4
        if my_dirc == 0 or my_dirc == 3:
            return 'l'
        else:
            return 'r'
```

函数 away

4. avoid_wall：根据当前局势判断纸卷转向，尤其避免撞墙情况的发生。

   基本思路：根据纸卷所处位置的不同，分为两种情况，通过修改 switch 的值触发不同机制。

   (a) 不在己方领地内 (switch=0)：不近墙时，往远离对方纸卷的方向转向圈地（调用 away 函数）；近墙时，纸卷会触发 goback 模式，可能出现掉头，在这种情况下，要求纸卷往原方向（即掉头方向）转向，避免反向转头撞墙。

   (b) 在己方领地内 (switch=1)：不近墙时，随机转向（配合 mode go_out 找到出领地的路径）；近墙时，根据所近墙面和行进方向的不同，特殊处理——面向墙直行时及时往左右转向，否则往远离墙的方向转。

```
# 领地内碰墙掉头返回
def goback(field, me, storage):#毒瘤
    # 第一步掉头
    if storage['turn']:
        storage['last_turn'] = storage['turn']
        res, storage['turn'] = storage['turn'], None
        return res
```

mode goback 中对 storage[‘turn’] 和 storage[‘last_turn’] 的处理

```
#避免靠墙的时候撞墙
def avoid_wall(field, me, storage, switch=0):
    if switch==0:#贴墙延伸
        if  me['y'] <= 2 \
            or   me['y'] >= len(field[0])-2 \
            or   me['x'] <= 2 \
            or   me['x'] >= len(field)-2 :
            return storage['last_turn']
        else:
            return away(field, me, storage)
    else:#墙边找路
        if  me['y'] <= 2:
            if  me['direction'] == 3 :
                return choice('rl')
            else:
                return relative_dirc(me)['south']
        elif  me['y'] >= len(field[0])-2:
            if me['direction'] == 1:
                return choice('rl')
            else:
                return relative_dirc(me)['north']
        if  me['x'] <= 2 :
            if me['direction'] == 2:
                return choice('rl')
            else:
                return relative_dirc(me)['east']
        if  me['x'] >= len(field)-2 :
            if me['direction'] == 0:
                return choice('rl')
            else:
                return relative_dirc(me)['west']

        return choice('rl1234')
```

函数 avoid_wall

5. 函数 minfield：当纸卷在己方领地时，配合 go_out 函数找出最近出领地路线

基本思路：将己方位置距离走出领地的四个方向进行比较，选出最快走出领地的方向

(a) 分别对所处位置的四个方向进行遍历，并储存在四个变量中代表向各个方向里领地外的距离。若检测到墙壁即设为最大值（1000）。

(b) 比较各个存储值并返回最小距离所代表的方向

```
#判断哪边离外界最近
def minfield(field, me, storage):
    res = -1
    count0 = count1 = count2 = count3 = 0
    x = me['x']
    y = me['y']
    while x < len(field) and field[x][me['y']] == me['id']:      #east
        count0 += 1
        x += 1
        if x == len(field) - 3:
            count0 = 1000
            break
    x = me['x']
    y = me['y']
    while y < len(field[0]) and field[me['x']][y] == me['id']:      #south
        count1 += 1
        y += 1
        if y == len(field[0]) - 3:
            count1 = 1000
            break
    x = me['x']
    y = me['y']
    while x > 0 and field[x][me['y']] == me['id']:      #west
        count2 += 1
        x -= 1
        if x == 3:
            count2 = 1000
            break
    x = me['x']
    y = me['y']
    while y > 0 and field[me['x']][y] == me['id']:      #north
        count3 += 1
        y -= 1
        if y == 3:
            count3 = 1000
            break

    dic = {count0:0, count1:1, count2:2, count3:3}
    if min(count0, count1, count2, count3) < 2:
        return -1
    if (count0-count1)*(count0-count2)*(count0-count3)*(count1-count2)*(count1-count3)*(count2-count3) == 0\
        and max(count0, count1, count2, count3) != 1000:
        return -1
    return dic[min(count0, count1, count2, count3)]
```

mode

所有 mode 函数至少包含两个部分：1. 针对目前局势给出走法 2. 针对目前局势判断下一步走法，切换 mode

1. go_out

使用条件：当纸卷进入己方区域且判断为安全时

目标：找出走出己方区域的路径，尽快走出己方阵营

基本思路：

(a) 常规走法：即为在一个小范围内进行转向同时判断是否朝向最快离开方向，若是则返回直走离开领地。

(b) 防撞墙机制：当检测到 x 轴或 y 轴即将碰墙时，转换状态为掉头（goback）。同时在小范围游走时利用转向避免碰墙。

(c) 状态切换判断：当走出领地时，转换状态为圈地（square）或者特殊圈地（near_square）；当满足和敌方的一定条件时，切换状态为击杀（BFS_kill）。

```
# 向尽快走出领地的方向前进
if me['direction'] != minfield(field, me, storage) and minfield(field, me, storage) != -1:
    if me['y'] - 2 <= 0 and me['direction']!=3:   #避免返回bk以后直走撞墙
        return relative_dirc(me)['south']
    elif me['y'] + 2 >= len(field[0]) and me['direction']!=1:
        return relative_dirc(me)['north']
    if me['x'] - 2 <= 0 and me['direction']!=2:
        return relative_dirc(me)['east']
    elif me['x'] + 2 >= len(field) and me['direction']!=0:
        return relative_dirc(me)['west']

    return choice('rl')
```

go_out 的常规走法

```
    # 防止撞墙
    # x轴不撞墙
    nextx = me['x'] + directions[me['direction']][0]
    if nextx <= 1 and me['direction'] != 0 or nextx >= len(
            field) - 2 and me['direction'] != 2:
        storage['mode'] = 'goback'
        storage['count'] = 0
        if me['direction'] % 2 == 0:  # 掉头
            if me['y'] - 2 <= 0:
                next_turn = relative_dirc(me)['south']
                storage['turn'] = next_turn
                return next_turn
            elif me['y'] + 2 >= len(field[0]):
                next_turn = relative_dirc(me)['north']
                storage['turn'] = next_turn
                return next_turn
            next_turn = away(field, me , storage)
            storage['turn'] = next_turn
            return next_turn
        else:
            return 'lr' [(nextx <= 1) ^ (me['direction'] == 1)]

    # y轴不撞墙
    nexty = me['y'] + directions[me['direction']][1]
    if nexty <= 1 and me['direction'] != 1 or nexty >= len(
            field[0]) - 2 and me['direction'] != 3:
        storage['mode'] = 'goback'
        storage['count'] = 0
        if me['direction'] % 2:  # 掉头
            if me['x'] - 2 <= 0:
                next_turn = relative_dirc(me)['east']
                storage['turn'] = next_turn
                return next_turn
            elif me['x'] + 2 >= len(field):
                next_turn = relative_dirc(me)['west']
                storage['turn'] = next_turn
                return next_turn
            next_turn = away(field, me , storage)
            storage['turn'] = next_turn
            return next_turn
        else:
            return 'lr' [(nexty <= 1) ^ (me['direction'] == 2)]
```

go_out 的防撞墙机制

```
# 状态转换
if field[me['x']][me['y']] != me['id']:
    storage['mode'] = 'square'
    storage['count'] = randrange(1, 3)
    storage['turn'] = away(field, me , storage)
    storage['maxl'] = max(
        randrange(5, 10),
        dst(me, storage['enemy']) // 3)
    return ''
if dst(me,storage['enemy']) <= 10:
    storage['mode'] = 'BFS_kill'
    return
if dst(me,storage['enemy']) <= 25 and field[storage['enemy']['x']][storage['enemy']['y']] == me['id']:
    storage['mode'] = 'BFS_kill'
    return
```

go_out 的状态切换判断

2. square

使用条件：当在空白地区且判断为安全时

目标：在安全条件下以最大效率圈地

基本思路：

(a) 常规走法：以到敌方的折线距离的 1/3 为边长画正方形（因一般是画三边，如此做在较高圈地效率的情况下保证了一定安全）

(b) 防撞墙机制：当检测到靠墙时将 count 清零并转向，重新累计 count。相当于沿墙行走一个边长并返回.

(c) 状态切换判断：当回到自己的领地时，切换状态到走出领地（go_out）；当处于敌方领地时，若距离敌方过近产生威胁，便放弃圈地改为返回领地（BFS_home）；当处于外部时，在一定情况下触发击杀模式（BFS_kill）。

```
# 画方块
storage['count'] += 1
if storage['count'] >= storage['maxl']:
    storage['count'] = 0
    return storage['turn']
```

square 的常规走法

```
# 防止撞墙
if me['direction'] % 2 :  # y轴不撞墙
    nexty = me['y'] + directions[me['direction']][1]
    if nexty < 0 or nexty >= len(field[0]):
        storage['count'] = 0
        return storage['turn']
else:  # x轴不撞墙
    nextx = me['x'] + directions[me['direction']][0]
    if nextx < 0 or nextx >= len(field):
        storage['count'] = 0
        return storage['turn']
```

square 的防撞墙机制

```
# 状态转换
if field[me['x']][me['y']] == me['id']:
    storage['mode'] = 'go_out'
    storage['count'] = 2
    return
elif field[me['x']][me['y']] == storage['enemy']['id']:
    if dst(me,storage['enemy']) < storage['maxl']*2.5:
        storage['mode'] = 'BFS_home'
        return
else:
    if field[storage['enemy']['x']][storage['enemy']['y']] == me['id'] and dst(me,storage['enemy']) < storage['maxl']*1.5:
        storage['mode'] = 'BFS_kill'
        return
    elif dst(me,storage['enemy']) <= storage['maxl']:
        storage['mode'] = 'BFS_kill'
        return
    '''elif dst(me,storage['enemy']) < storage['maxl']*2:
        storage['mode'] = 'BFS_home'
        return'''
```

square 的状态切换判断

3. goback

使用条件：近墙时的特殊处理

目标：使纸卷近墙时掉头

基本思路：

(a) 掉头走法：转向掉头并将 storage[‘turn’] 清空，之后向前走制定步数

(b) 状态切换：正常情况下切换为走出领地模式（go_out）；满足条件时转换为击杀模式（BFS_kill）；若某些情况下掉头离开了领地即转换为特殊正方形模式（near_square）。

```
# 第一步掉头
if storage['turn']:
    storage['last_turn'] = storage['turn']
    res, storage['turn'] = storage['turn'], None
    return res
```

```
# 前进指定步数
storage['count'] += 1
if storage['count'] > 1:
    storage['mode'] = 'go_out'
    storage['count'] = 1
    return avoid_wall(field, me, storage,1)
```

```
# 状态转换
if dst(me,storage['enemy']) < 20:
    storage['mode'] = 'BFS_kill'

elif field[me['x']][me['y']] != me['id'] :
    storage['mode'] = 'near_square'
    storage['count'] = 2
    storage['maxl'] = max(
        randrange(5, 10),
        dst(me, storage['enemy']) // 3)
    storage['turn'] = avoid_wall(field, me, storage)
    return ''
```

4. near_square 使用条件：在 goback 和 go_out 后，纸卷以某种特殊姿势伸出领地继续圈地。

目标：保证纸卷在这种姿势下不自杀且仍以最大效率圈地

组成单元：

(a) 常规画方块：增加 storage[‘side’] 计画方块的边数。在画最后一条边时，增大边长，使得纸卷能回到自己领地

(b) 防撞墙机制：继承自 square。

(c) 防撞墙机制：继承自 square。

```
# 画方块
storage['count'] += 1
if storage['count'] >= storage['maxl']:
    storage['count'] = 0
    storage['side']+=1
    if storage['side'] == 2:
        storage['maxl']+=1
    return storage['turn']
```

5. BFS_kill

使用条件：当和对方并不处于安全距离内，且判断能够出手击杀

目标：以最短路径碰撞对方纸带

组成单元：

(a) BFS 算法

(b) kill 条件判断和状态切换：以 storage[‘leave’] 计算开启 kill 模式后的步数

具体情况分析如下：

i. 走步较多时依然没能攻击到对方时：若此时我方纸卷不在己方地内且距离敌方纸卷较远，及时切换到 BFS_home 模式，以最短路径返回自己领地；否则继续攻击

ii. 如果对方处于己方领地中：切换到 BFS_home 模式，以最短路径返回自己领地。（之后再由 BFS_home 切换至 BFS_kill 模式）

iii. 如果我方纸卷处于对方领地中且距离对方纸卷较近：切换到 BFS_home 模式，以最短路径返回自己领地。

iv. 否则开启攻击模式，攻击时注意避免撞墙（调用 avoid_wall）【若攻击模式异常，则选择一键回家】

```
bfs_tool_box('kill')
if storage['leave']>storage['maxl']*1.5:
    if field[me['x']][me['y']] != me['id'] and dst(me,storage['enemy']) > storage['maxl']:
        #print('tao')
        storage['mode'] = 'BFS_home'
        storage['leave'] = 0
        return
if field[storage['enemy']['x']][storage['enemy']['y']] == storage['enemy']['id']:
    storage['mode'] = 'BFS_home'
    storage['leave'] = 0
    return
if field[me['x']][me['y']] == storage['enemy']['id'] and dst(me,storage['enemy']) < storage['maxl']*2.5:
    #print('pao')
    bfs_tool_box('back_home')
    storage['mode'] = 'BFS_home'
    storage['leave'] = 0
    return
elif len(storage['move_list'])!=0:
    if me['y'] <= 2 or me['y'] >= len(field[0])-2 or  me['x'] <= 2 or me['x'] >= len(field)-2:
        return avoid_wall(field, me, storage, 1)
    #print('woc')
    return storage['move_list'].pop(0)
else:
    #print('What?')
    storage['mode']='BFS_home'
    return
```

6. BFS_home:

使用条件：和对方不在安全距离内且判断此时不是击杀时机，选择逃之夭夭

目标：以最短路径返回自己领地

组成单元：

(a) BFS 算法

(b) back_home 条件判断和状态切换：

i. 如果对方不在对方的领地内且双方距离很近：切换至 BFS_kill 进攻

ii. 不符合以上条件后，如果在自己领地内，切换至 go_out，先往领地外走

iii. 否则（在领地外，且不适合击杀），开启回家模式，逃之夭夭。

3 实验结果

3.1 测试过程

实验环境说明：

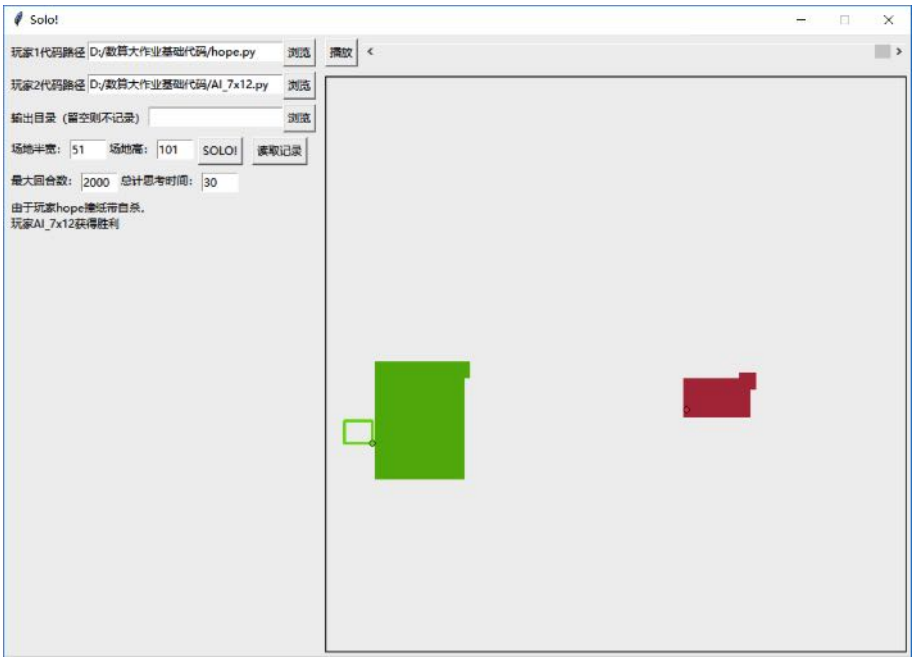
- 硬件配置：1.3 GHz Intel Core m7 / 8 GB 1867 MHz LPDDR3
- 操作系统：MacOS HighSierra 10.13.4
- Python 版本：3.6.2

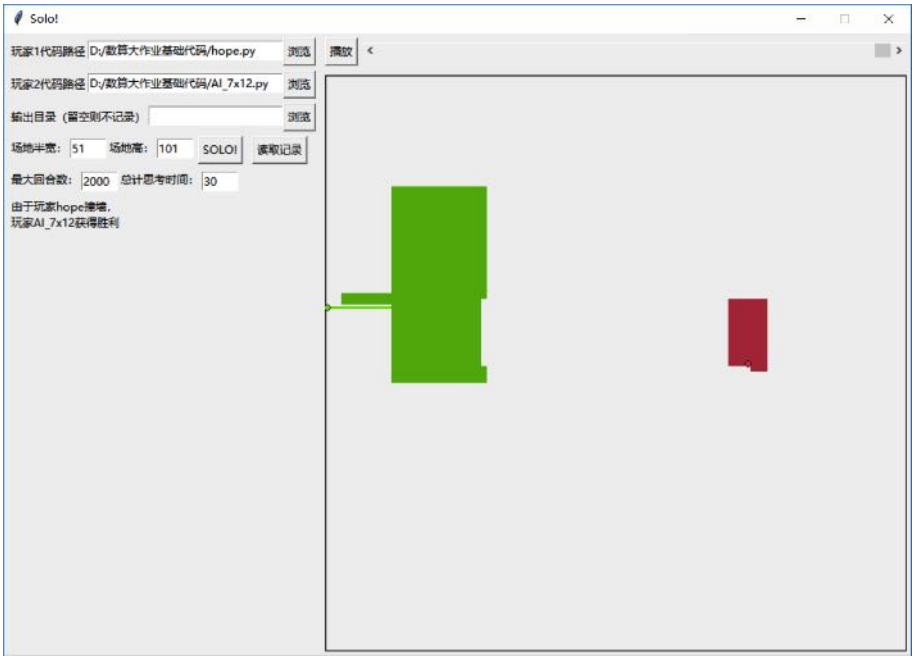
本组的代码调试是不断 debug 和进行组内“自然选择”的过程。首先是两个思路的提出：

1、选择一个我方领地的端点作为目标点，然后将这个过程中经过点选取出，形成纸带的路径，并执行。

2、每次围地前计算出最长的安全路径，并执行之。

两个思路的代码在开始时都存在不少 bug。比较常见的 bug 就是纸带的撞墙和撞击己方纸带。如下图所示：



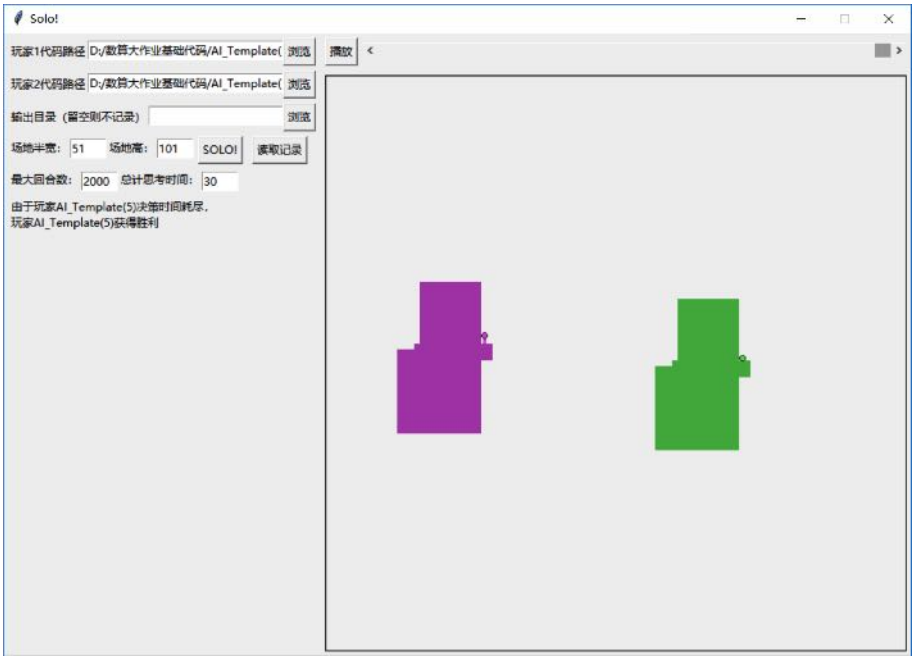


后来经过一番调整，改变了算法的一些判断机制和执行步骤的思考逻辑，纸带的自我撞毁概率有了极大的下降，但还是偶有发生。经过小组成员在一起讨论，修复 bug，最终纸带能够保证在不自我撞毁的情况下进行圈地。

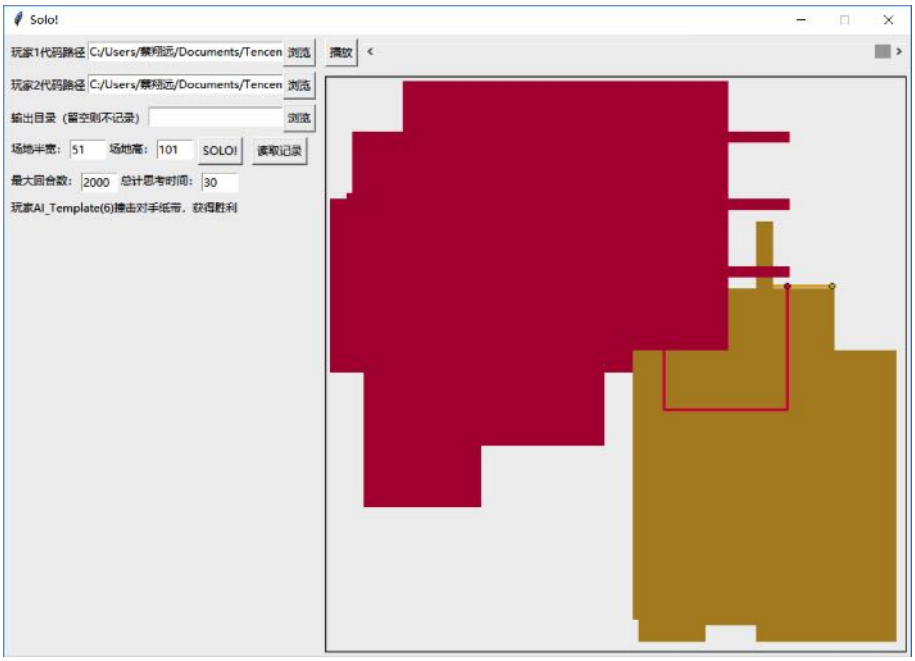
在第一次的组内较量中，发现思路 1 的算法有更高的灵活性的生存能力，于是经过讨论我们决定以该思路为主要想法，并保留思路 1 作为测试。

这时我们也发现了该算法在攻击方面显示出的不足，想到不久前在课堂上学到的图算法，我们小组成员开发出 bfs 的模块，对之前的算法进行改进。

刚进行合并算法后很快发现了问题：如果大量使用该模块，在有限的思考时间限制下，往往会出现决策时间耗尽的情况，如图：



这种情况是十分不利的，于是在讨论后，我们将 bfs 运用的条件修改为一定条件下才会触发，从而减少了决策时间的浪费。修改后的代码在思考时间的进攻性方面都有很大的性能提升。如下图（红色方为新的代码）：



在经过一些细节上的微调后，我们的参战代码最终产生。

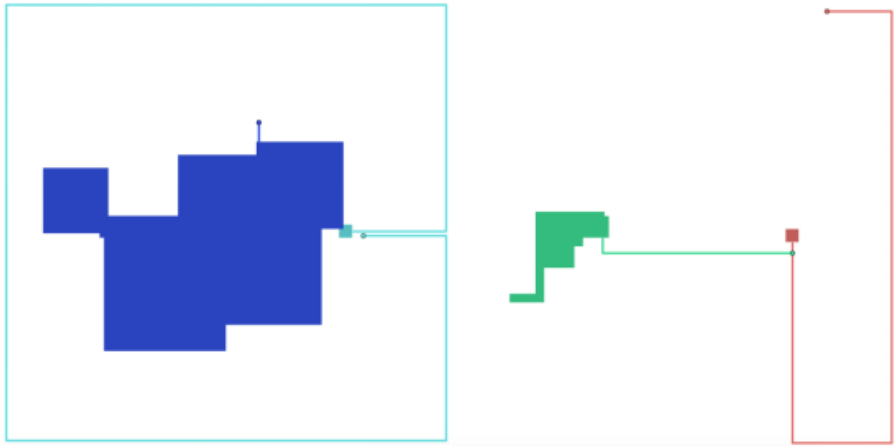
3.2 结果分析

在对战的初期，我们代码中一开始的贪心策略起到了迅速扩展根据地的作用，对战过程中，BFS 算法也能够对能够绝杀的情况进行精准狙击。但是，我们的代码并没有做到把战局向能够绝杀的方向演进的作用，因此在面对强大的智能算法是会惨遭消灭。

本算法主要的时间都花费在调用 BFS 进行搜索上。经过测试，一局比赛中能调用 BFS 的次数在 600-700 左右，这样保证能够不超过时间上限 30s。

3.3 经典战局

在还未开发出 BFS 版本的初版代码中，我们的代码面对“围大圈”的进攻策略，溃不成军。BFS 算法首次应用之后，便能够抓住时机，击杀围大圈代码。



溃败的初版代码　　击杀“围大圈”算法的整合 BFS 之后的代码

4 实习过程总结

4.1 分工与合作

六人两两一组分为三组：（1）李博、蔡翔远（2）林小靖、丁泽宇（3）周子楠、任和。分工合作分为如下几个阶段：

第一阶段。（1）（2）组各自负责编写一个无 bug、不会自杀、在无进攻型假想敌的前提下高效圈地的代码；（3）组负责编写功能函数，通过 bfs 算法计算从当前位置到达目标位置（如回领地或必杀出击）的最快可行路径，作为下一阶段进攻性部分的准备。

第二阶段。（1）（2）组圈地代码竞赛，积累经验，将两种算法各自的优势整合，并解决一些双方实际抢围领地过程中出现的一些新 bug，解决“被逼撞墙”、“陷入死循环”等情况；（3）组确定出击策略，利用之前准备好的 bfs 算法，通过比较敌我双方的出击路程和归程，确保在可以一击必杀时有效出击，在将要产生被必杀威胁时立即回城，在没有威胁时大片圈地，完成与地方交战、博弈部分的代码。

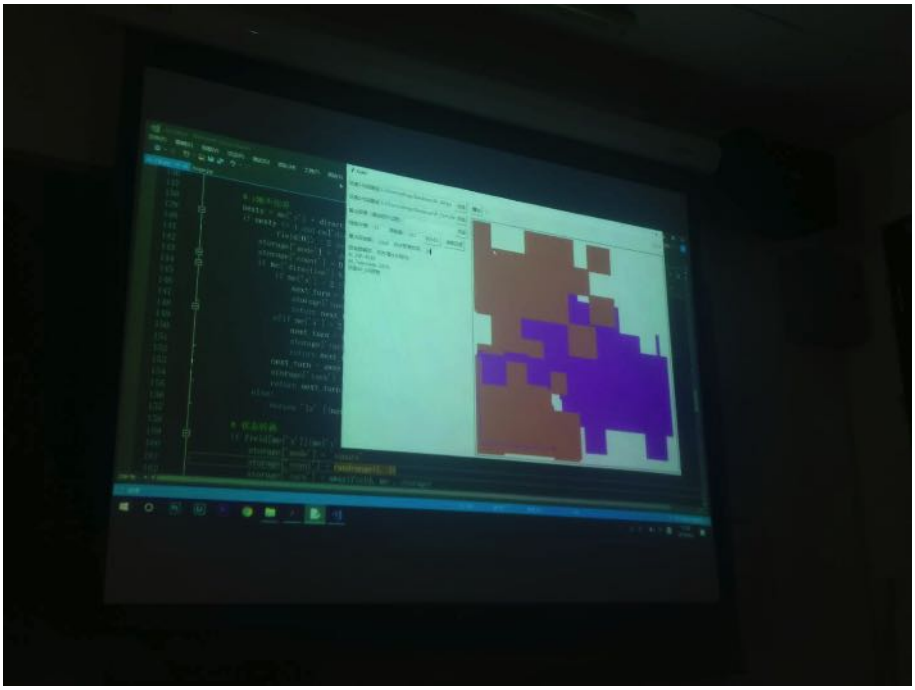
第三阶段。（1）（2）（3）组合作，探讨并编写在每回合中圈地还是出击的判断机制和逻辑层级关系，在组长的带领下整合、对接代码，排除 bug，并考虑到比赛时间限制，将代码不必要的部分进行删减，完成一个功能基本完整、可用于热身赛的代码。

第四阶段。继续优化代码结构，消除 bug，并通过在热身赛中人脑积累的学习经验，将先前代码中存在漏洞、不能很好地处理的情况（如遭遇“围大圈”的算法）单列出来，单独为其编写对策，以此提高与各种不同类型代码博弈中获胜的概率。

第五阶段。参加最终比赛，并总结这次小组学习的收获。（1）组负责总结算法思想，提供程序代码说明；（2）组负责记录并总结代码测试过程；（3）组负责 bfs 部分的代码说明以及小组分工与总结。

大作业完成期间我们进行了频繁的线上交流，同时进行了 2 次正式、准备充分、高效的组会来解决问题。竞赛期间小组成员积极观看比赛并进行总结。

第一次组会上，积极讨论，构建基本框架



第二次组会，现场展示，寻找问题

认真观摩正式比赛之后，小组合影

4.2 经验与教训

本组组织工作较好的地方是讨论方式安排合理。在恰当的时机安排线下组会，集中解决关键性问题。而对于细节问题采取线上讨论的方式，不占用大家太多时间，效率较高。但本组的分工稍微有些分散，而且没有事先对接口部分达成一个清晰的共识，导致代码最终在对接上遇到困难，不得不牺牲一部分功能来解决。

4.3 建议与设想

希望今后的同学在比赛之前充分利用时间阅读文档，进行小组分工时事先商量清楚接口的相关共识，较少不必要的麻烦。

5 致谢

感谢清华大学陈厚睿学长对本小组工作提供的疑难解答。

感谢北京大学宋煦学长为本小组提供的详尽咨询服务。

6 参考文献

[1] 纸带圈地游戏原始网页, paper-io.com

[2] 课程教材原始网页，https://runestone.academy/runestone/static/pythonds/index.html

第三章　F17_Bravo 组报告

李想 *，宁湘宇，高宇航，宋筱泊

摘要：我们的大致策略很简单，就是以防御为主的围地策略。在领地内行走时尽量以最快的方式走出来，出领地时就围矩形，若有被攻击的危险就立即返回领地。上述功能主要用列表来实现。我们的算法在应对防御围地型 AI 时优势还是比较大的，但在应对攻击型 AI 时很吃亏。

关键词：列表、围矩形、最短逃跑路径、领地内部走法

1 算法思想

1.1 总体思路

我们组开始时定下了圈地为主、攻击为辅的策略，在编写程序之前先通过 paper.io 游戏研究应采取什么样圈地的方法。最初我们提出了如下几种方法：

1)“围蚊香”式圈地法：在领地外圈地时靠近己方领地边缘行走，出现危险时可以迅速回到领地内部。

2)“围海造田”式圈地法：通过围一些细长条的形状使领地变成“凹”字形，之后对“凹”字形进行封口，从而用较短的路径围出较大的区域。

3)“边路偷袭”式圈地法：绕场地围一圈，最后一举把全部场地变为自己的领地。

4) 围矩形式圈地法：从自己的领地边缘出发向外围矩形，根据敌人的位置调整矩形的长与宽。

我们组对这四种方法的效率、可行性等进行了讨论与分析，最后认为：“围蚊香”法围地效率太低，虽然能保证存活，但在领地面积上无法与敌人抗衡；“围海造田”法不易操作，围长条形区域时较为危险且效率低下，“凹”字形区域也容易遭到敌人切断导致计划失败；“边路偷袭”法需要很长时间才能完成目的，中间领地被敌人切断的可能性很大。因此最终我们倾向于选择矩形围地法，其优点主要有三：一是比较容易完成，且比较灵活；二是圈地效率与成功率相对较高，矩形相对其他图形周长面积比最小，即用同样的圈地长度可以围出更大的面积；三是相对安全，可以根据敌人情况调整矩形的长与宽，保护自己不被击杀，也可以避免撞到自己的情况。

定下这一方法后我们尝试编写了第一个程序，主要用了两个规则：到领地边缘时生成围矩形的走法并用列表存储，之后调用列表进行圈地；在领地内走直线快速出领地。之后的测试中，我们根据它的一些缺陷又做了如下改进：

1) 确定比较合适的确定矩形长宽的方法；

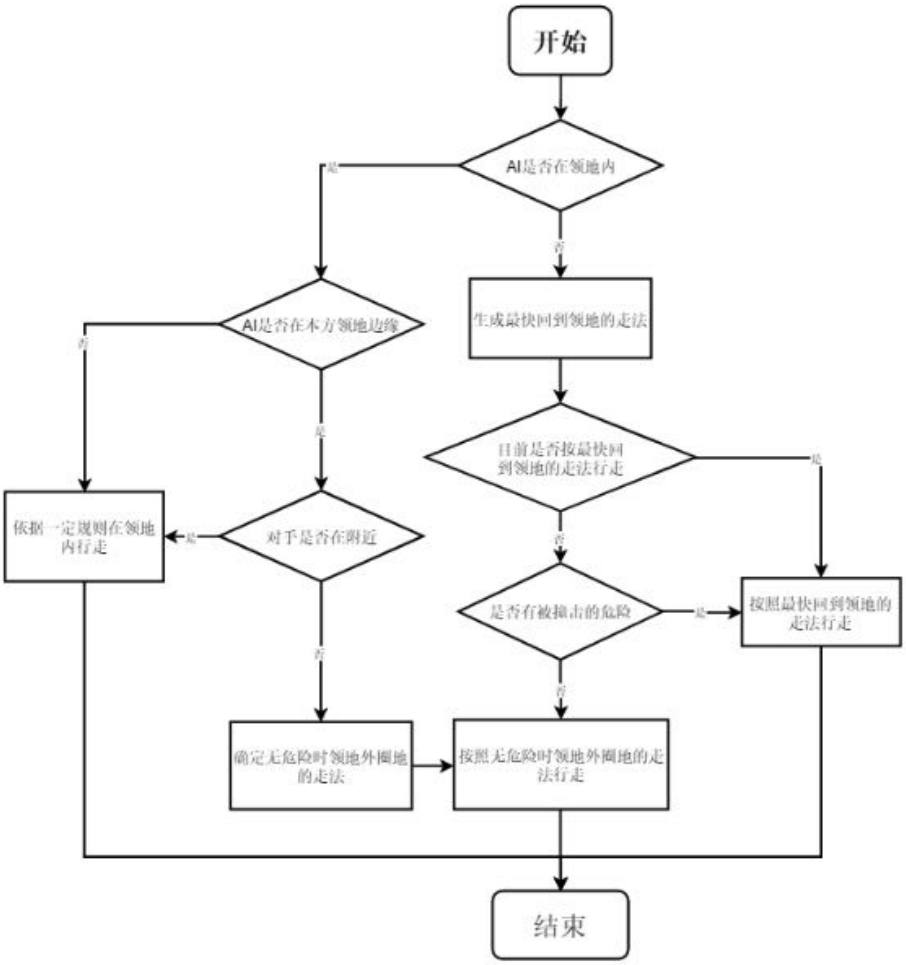
2) 增加防御机制，在领地外圈地遇到危险时可以选择最快回领地的路径进行逃脱；

3) 优化在领地内的走法，原走法（直走）可能导致 AI 在领地内的时间较长，产生很多垃圾时间，改进后能让 AI 尽快出领地进行圈地；

4) 在圈地时若后路被截断（如图所示），则改变走法，避免自杀，增加回到领地的可能性。

因为时间原因，我们没能再编写进行攻击的方法，因此我们的最后成果就是根据以上思路编程的以圈地防御为主的 AI。

1.2 算法流程图



1.3 算法复杂度

我们的算法复杂度主要体现在两个操作过程中，同时复杂度也与 AI 所处的位置有关，不同的位置对应不同的复杂度。

第一个操作过程是在领地内时判断前进的方向，此时需要进行循环操作，循环的规模相当于场地的边长 K，算法复杂度为 O(K)。

第二个操作过程是在领地外时需要计算围地时行走的路径或者遇到危险时回领地的最短路径。当刚出领地时计算的围地路径，该路径的长度与场地规模相当，算法复杂度仍为

O(K)。当在领地外行走时，计算敌人离己方纸带的最小距离和计算回领地的最短路径都与已经形成的纸带长度 L 有关，算法复杂度为 O(L)。

2 程序代码说明

2.1 数据结构说明

我们的算法主要使用的是列表，列表主要用于以下两个部分：

1、将围地的走法提前计算出来存入列表中，再依次出列执行操作。

2、利用列表记录纸带以及将要走的路径。

2.2 函数说明

（1）foursquare 函数参数：矩形的长和宽，拐角处的的转向先创建一个存储行走法则的空列表 routine，之后根据长和宽，输入相应数量的‘S’，同时根据在拐角处的转向，输入‘L’或‘R’

```
def foursquare(a, b, c):  # 定义画四条边矩形的函数：为长，为宽，为转向。走法存储到列表中abcroutine
    routine = []
    for i in range(a - 1):
        routine.append('S')
    routine.append(c)
    for i in range(b - 2):
        routine.append('S')
    routine.append(c)
    for i in range(a - 1):
        routine.append('S')
    routine.append(c)
    for i in range(b - 2):
        routine.append('S')
    return routine
```

（2）minthreat 函数参数：己方纸带列表，对手横纵坐标 alist 是己方纸带的列表。调用对方头部的坐标，求出其到己方纸带上所有点的距离，取其中的最小值返回。这样得到的距离其实不一定就是对方头部撞到己方纸带的最小距离，因为受对方纸带分布情况等的

限制，实际最小距离可能比该函数返回的距离大，但考虑到精确算出最小距离考虑的情况太多，算法很复杂，故这里我们只考虑最坏的情况。

```
def minthreat(alist, x, y):
    distance = abs(alist[0][0] - x) + abs(alist[0][1] - y)
    for i in alist:  # 遍历整个列表，求出对手与列表中所有坐标的距离alist
        if abs(i[0] - x) + abs(i[1] - y) < distance:  # 一直取较小的数distance
            distance = abs(i[0] - x) + abs(i[1] - y)
    return distance  # 最终返回的即为所有距离中最小的distance
```

（3）mingohome 函数参数：己方纸带列表，己方横纵坐标，己方 id，棋盘 stat 这个函数用于己方在己方领地外行走时确定返回领地的最短路径，以防备对手可能的攻击。大致策略是：己方在领地外时，确定一条路径，使得该路径与己方现有纸带刚好围成一个矩形，这样路径长度与纸带长度基本相同。这样走是为了保证在确保能安全回领地的前提下围尽量多的地。

```
# 定义返回领地的最短路径的函数，返回领地走的还是一个矩形，其中为己方纸带列表，、为己方坐标，为己方，为棋盘mingohomealistxymeidstat
def mingohome(alist, x, y, me, stat):
    if stat['now']['fields'][x][y] == me:  # 若此时己方就在自己的领地内，不用这个函数
        return None
    else:  # 若此时己方不在自己领地内，计算最短路径
        d = stat['now']['me']['direction']  # 表示此时己方的方向d
        gohome = []  # 定义存储沿最短路径行走时经过的坐标的列表gohome
        gohomeroute = []  # 定义存储沿最短路径行走的走法的列表（含有gohomeroute'S、''L、''R）'
```

当己方还没有围完矩形的第一条边时，最短路径即为：直接掉头返回领地，转弯方向视情况而定。

```
        if alist[0][0] == alist[-1][0] or alist[0][1] == alist[-1][1]:  # 若纸带仅有一条边（这时纸带首位的纵坐标或横坐标是相等的）
            if alist[0][0] == alist[-1][0] and alist[0][1] != alist[-1][1]:  # 若纸带初始横坐标等于最后的横坐标
                yy = alist[-1][1]  # 令为最后的纵坐标yy
                if alist[0][0] < 50:  # 若横坐标小于，令加上，表示第一步向横坐标增大的方向走，用存储的值50xx1x11xx
                    xx = alist[0][0] + 1
                    x11 = alist[0][0] + 1
                else:  # 否则，令减去，表示第一步向横坐标减小的方向走，用存储的值xx1x11xx
                    xx = alist[0][0] - 1
                    x11 = alist[0][0] - 1
            elif alist[0][1] == alist[len(alist) - 1][1] and alist[0][0] != alist[len(alist) - 1][0]:  # 若纸带初始纵坐标等于最后的纵坐标
```

```
    xx = alist[len(alist) - 1][0]  # 令为最后的横坐标xx
    if alist[0][1] < 50:  # 若纵坐标小于，令加上，表示第一步向纵坐标增大的方向走，
                          用存储的值50yy1y11yy
        yy = alist[0][1] + 1
        y11 = alist[0][1] + 1
    else:  # 否则令,减去，表示第一步向纵坐标减小的方向走，用存储的值yy1y11yy
        yy = alist[0][1] - 1
        y11 = alist[0][1] - 1
a = 0
for i in range(len(alist) - 1):  # 掉头后走的步数恰等于之前纸带的长度
    gohome = gohome + [[xx, yy]]  # 记录沿最短路径回领地时要走过的坐标
    while a < 2:  # 在最初始的步视情况在里存储左转或右转2gohomeroute
        if (d == 0 and yy == alist[0][1] + 1) or (d == 1 and xx == alist[0
                          ][0] - 1) or (
                d == 2 and yy == alist[0][1] - 1) or (d == 3 and xx ==
                          alist[0][0] + 1):
            for i in range(2):  # 在以上的四种情况下，最短路径走法的前两步都是右
                          转，在里加两个gohomeroute'R'
                gohomeroute.append('R')
                a = a + 1
        else:  # 其他情况下，最短路径走法的前两步都是左转，在里加两个gohomeroute'L'
            for i in range(2):
                gohomeroute.append('L')
                a = a + 1
        # 接下来视初始方向，判断里存储的第二个坐标是什么gohome
        if d == 0:
            xx, yy = xx - 1, yy
            gohome = gohome + [[xx, yy]]
        elif d == 1:
            xx, yy = xx, yy - 1
            gohome = gohome + [[xx, yy]]
        elif d == 2:
            xx, yy = xx + 1, yy
            gohome = gohome + [[xx, yy]]
        else:
            xx, yy = xx, yy + 1
            gohome = gohome + [[xx, yy]]
        d = (d + 2) % 4  # 完成前两步后，己方方向刚好转了，以此计算出此时的方向180°
    gohomeroute.append('S')  # 接下来的走法就是一直直走
    xx = xx + storage['directions'][d][0]  # 判断下一步的坐标x
    yy = yy + storage['directions'][d][1]  # 判断下一步的坐标y
    a = a + 1
gohome = gohome + [[xx, yy]]
# 最后一步是一个转向，视之前的转向情况而定
```

```
            if 'R' in gohomeroute:
                gohomeroute.append('R')
            elif 'L' in gohomeroute:
                gohomeroute.append('L')
            a = a + 2
            return [len(gohome), gohome, gohomeroute]  # 最后返回一个含有三个元素的列表：第一个是最短路径长度，第二个是最短路径中经过的坐标，第三个是最短路径走法
```

当己方正在围矩形的第二条边时，最短路径即为：与第一次转弯的转向相同，先转向，然后向前走 a 个长度单位（a 为矩形第一条边的长度），之后再跟之前一样转向，然后一直走，直到回到初始位置。

```
        elif alist[0][0] != alist[-1][0] and alist[0][1] != alist[-1][1]:  # 若纸带有两条边（这时纸带首尾两点的横纵坐标都不相等，且不可能出现纸带有三、四条边的情况，具体原因后面会提到）
            # 先确定已围两边中第一条边的长度和第二条边的长度ab
            if alist[0][0] == alist[1][0]:
                a = 0
                while alist[a][0] == alist[0][0]:
                    a = a + 1
                b = 0
                while a + b < len(alist):
                    b = b + 1
            else:
                a = 0
                while alist[a][1] == alist[0][1]:
                    a = a + 1
                b = 0
                while a + b < len(alist):
                    b = b + 1
            # xx(yy)，xxx(yyy)，x3(y3)，x4(y4)分别为转弯点的下一个点的坐标、转弯点的上一个点的坐标、转弯点的坐标、纸带最后一个点的坐标
            xx, xxx, x3, x4 = alist[a][0], alist[a - 2][0], alist[a - 1][0], alist[len(alist) - 1][0]
            yy, yyy, y3, y4 = alist[a][1], alist[a - 2][1], alist[a - 1][1], alist[len(alist) - 1][1]
            # 下面根据、、、、、的相对关系，判断己方在转弯点是左转还是右转，以此作为最短路径中如何转弯的依据xxxxxx3yyyyyy3
            if xx == xxx + 1:
                if yy == yyy + 1:
                    if xxx == x3:  # 当xx == xxx + 、1yy == yyy + 、1xxx == 时，说明己方在转弯处是左转x3
                        gohomeroute.append('L')
                        for i in range(a - 1):
                            gohome = gohome + [[x4, y4 - 1 - i]]
```

```
                    gohomeroute.append('S')
                gohomeroute.append('L')
                for i in range(b - 1):
                    gohome = gohome + [[x4 - 1 - i, y4 - a + 1]]
                    gohomeroute.append('S')
            else:  # 当xx == xxx + 、1yy == yyy + 、1xxx != 时，说明己方在转弯处是右转x3
                gohomeroute.append('R')
                for i in range(a - 1):
                    gohome = gohome + [[x4 - 1 - i, y4]]
                    gohomeroute.append('S')
                gohomeroute.append('R')
                for i in range(b - 1):
                    gohome = gohome + [[x4 - a + 1, y4 - 1 - i]]
                    gohomeroute.append('S')
        else:
            if xxx == x3:  # 当xx == xxx + 、1yy == yyy - 、1xxx == 时，说明己方在转弯处是右转x3
                gohomeroute.append('R')
                for i in range(a - 1):
                    gohome = gohome + [[x4, y4 + 1 + i]]
                    gohomeroute.append('S')
                gohomeroute.append('R')
                for i in range(b - 1):
                    gohome = gohome + [[x4 - 1 - i, y4 + a - 1]]
                    gohomeroute.append('S')
            else:  # 当xx == xxx + 、1yy == yyy - 、1xxx != 时，说明己方在转弯处是左转x3
                gohomeroute.append('L')
                for i in range(a - 1):
                    gohome = gohome + [[x4 - 1 - i, y4]]
                    gohomeroute.append('S')
                gohomeroute.append('L')
                for i in range(b - 1):
                    gohome = gohome + [[x4 - a + 1, y4 + 1 + i]]
                    gohomeroute.append('S')
    else:
        if yy == yyy + 1:
            if xxx == x3:  # 当xx == xxx - 、1yy == yyy + 、1xxx == 时，说明己方在转弯处是右转x3
                gohomeroute.append('R')
                for i in range(a - 1):
                    gohome = gohome + [[x4, y4 - 1 - i]]
                    gohomeroute.append('S')
                gohomeroute.append('R')
```

```
                for i in range(b - 1):
                    gohome = gohome + [[x4 + 1 + i, y4 - a + 1]]
                    gohomeroute.append('S')
            else:  # 当xx == xxx - 、1yy == yyy + 、1xxx != 时，说明己方在转弯处是左转x3
                gohomeroute.append('L')
                for i in range(a - 1):
                    gohome = gohome + [[x4 + 1 + i, y4]]
                    gohomeroute.append('S')
                gohomeroute.append('L')
                for i in range(b - 1):
                    gohome = gohome + [[x4 + a - 1, y4 - 1 - i]]
                    gohomeroute.append('S')
        else:
            if xxx == x3:  # 当xx == xxx - 、1yy == yyy - 、1xxx == 时，说明己方在转弯处是左转x3
                gohomeroute.append('L')
                for i in range(a - 1):
                    gohome = gohome + [[x4, y4 + 1 + i]]
                    gohomeroute.append('S')
                gohomeroute.append('L')
                for i in range(b - 1):
                    gohome = gohome + [[x4 + 1 + i, y4 + a - 1]]
                    gohomeroute.append('S')
            else:  # 当xx == xxx - 、1yy == yyy - 、1xxx != 时，说明己方在转弯处是右转x3
                gohomeroute.append('R')
                for i in range(a - 1):
                    gohome = gohome + [[x4 + 1 + i, y4]]
                    gohomeroute.append('S')
                gohomeroute.append('R')
                for i in range(b - 1):
                    gohome = gohome + [[x4 + a - 1, y4 + 1 + i]]
                    gohomeroute.append('S')
    return [len(gohome), gohome, gohomeroute]  # 最后返回一个含有三个元素的列表：第一个是最短路径长度，第二个是最短路径中经过的坐标，第三个是最短路径走法
```

当己方已经围完矩形的两条边时，按照最短路径的生成法则，这时的最短路径与最初规定好的围矩形的走法一样。故不再讨论。

（4）wanderer 函数参数：方向，纸卷的横纵坐标，stat，本方 AI 的 id 创建一个列表包括左、前、右三个方向，如果某个方向上会撞到墙壁或自己纸袋，则从列表中删除该方向。最后函数的返回值为该列表，其中包含了可选的方向。

```
def wanderer(direction, x, y, stat, id):  # 定义在领地内行走法则的函数wanderer
    turnList = ['L', 'R', 'S']

    if direction == 0:
        if x + 1 == stat['size'][0] or stat['now']['bands'][x + 1][y] == id:
            turnList.remove('S')
        if y == 0 or stat['now']['bands'][x][y - 1] == id:
            turnList.remove('L')
        if y + 1 == stat['size'][1] or stat['now']['bands'][x][y + 1] == id:
            turnList.remove('R')
    if direction == 2:
        if x == 0 or stat['now']['bands'][x - 1][y] == id:
            turnList.remove('S')
        if y == 0 or stat['now']['bands'][x][y - 1] == id:
            turnList.remove('R')
        if y + 1 == stat['size'][1] or stat['now']['bands'][x][y + 1] == id:
            turnList.remove('L')
    if direction == 1:
        if y + 1 == stat['size'][1] or stat['now']['bands'][x][y + 1] == id:
            turnList.remove('S')
        if x + 1 == stat['size'][0] or stat['now']['bands'][x + 1][y] == id:
            turnList.remove('L')
        if x == 0 or stat['now']['bands'][x - 1][y] == id:
            turnList.remove('R')
    if direction == 3:
        if y == 0 or stat['now']['bands'][x][y - 1] == id:
            turnList.remove('S')
        if x == 0 or stat['now']['bands'][x - 1][y] == id:
            turnList.remove('L')
        if x + 1 == stat['size'][0] or stat['now']['bands'][x + 1][y] == id:
            turnList.remove('R')
    return turnList
```

（5）play 函数里在领地内走法的进一步制定上面四个函数都是在 load 函数里设定的基本函数。在实际操作中，还需要考虑很多情况。下面对其进行说明：

当 AI 在领地内时，光有 wanderer 函数是不够的，需要为 AI 挑一个方向，让他能走出领地去围地。我们选择了这样一种方法：沿 AI 可走的三个方向的三条线段遍历，哪个方向上不属于自己的坐标数最多，就让 AI 走哪个方向。

```
            if stat['fields'][nextx][nexty] == me + 1:  # 若沿当前方向走下一步后仍在领地内
                nextTurn = 's'
```

```
nextD = 0
turnList = storage['wanderer'](stat['me']['direction'], x1, y1,
                               newstat, me + 1)

# 选择一个方向，沿该方向上不是自己领地的格子最多，目的是尽快出自己领地围地，提高围地
    效率
if d == 0:
    if len(turnList) == 1:
        nextTurn = 's'
    else:
        white1 = 0
        white2 = 0
        white3 = 0
        i = x1 + 1
        while i < size1:
            if stat['fields'][i][y1] != me + 1:
                white1 += 1
            i += 1
        if 'L' in turnList:
            i = y1 - 1
            while i >= 0:
                if stat['fields'][x1][i] != me + 1:
                    white2 += 1
                i -= 1
        if 'R' in turnList:
            i = y1 + 1
            while i < size2:
                if stat['fields'][x1][i] != me + 1:
                    white3 += 1
                i += 1
        if white1 == max(white1, white2, white3):
            nextTurn = 's'
            nextD = 0
        elif white2 == max(white1, white2, white3):
            nextTurn = 'l'
            nextD = 3
        else:
            nextTurn = 'r'
            nextD = 1
if d == 2:
    if len(turnList) == 1:
        nextTurn = 's'
    else:
        white1 = 0
```

```
                white2 = 0
                white3 = 0
                i = x1 - 1
                while i >= 0:
                    if stat['fields'][i][y1] != me + 1:
                        white1 += 1
                    i -= 1
                if 'L' in turnList:
                    i = y1 + 1
                    while i < size2:
                        if stat['fields'][x1][i] != me + 1:
                            white2 += 1
                        i += 1
                if 'R' in turnList:
                    i = y1 - 1
                    while i >= 0:
                        if stat['fields'][x1][i] != me + 1:
                            white3 += 1
                        i -= 1
                if white1 == max(white1, white2, white3):
                    nextTurn = 's'
                    nextD = 2
                elif white2 == max(white1, white2, white3):
                    nextTurn = 'l'
                    nextD = 1
                else:
                    nextTurn = 'r'
                    nextD = 3
        if d == 1:
            if len(turnList) == 1:
                nextTurn = 's'
            else:
                white1 = 0
                white2 = 0
                white3 = 0
                i = y1 + 1
                while i < size2:
                    if stat['fields'][x1][i] != me + 1:
                        white1 += 1
                    i += 1
                if 'L' in turnList:
                    i = x1 + 1
                    while i < size1:
```

```
                    if stat['fields'][i][y1] != me + 1:
                        white2 += 1
                    i += 1
            if 'R' in turnList:
                i = x1 - 1
                while i >= 0:
                    if stat['fields'][i][y1] != me + 1:
                        white3 += 1
                    i -= 1
            if white1 == max(white1, white2, white3):
                nextTurn = 's'
                nextD = 1
            elif white2 == max(white1, white2, white3):
                nextTurn = 'l'
                nextD = 0
            else:
                nextTurn = 'r'
                nextD = 2
    if d == 3:
        if len(turnList) == 1:
            nextTurn = 's'
        else:
            white1 = 0
            white2 = 0
            white3 = 0
            i = y1 - 1
            while i >= 0:
                if stat['fields'][x1][i] != me + 1:
                    white1 += 1
                i -= 1
            if 'R' in turnList:
                i = x1 + 1
                while i < size1:
                    if stat['fields'][i][y1] != me + 1:
                        white2 += 1
                    i += 1
            if 'L' in turnList:
                i = x1 - 1
                while i >= 0:
                    if stat['fields'][i][y1] != me + 1:
                        white3 += 1
                    i -= 1
            if white1 == max(white1, white2, white3):
```

```
                nextTurn = 's'
                nextD = 3
            elif white2 == max(white1, white2, white3):
                nextTurn = 'r'
                nextD = 0
            else:
                nextTurn = 'l'
                nextD = 2

        nextx1 = x1 + storage['directions'][nextD][0]
        nexty1 = y1 + storage['directions'][nextD][1]
        if stat['fields'][nextx1][nexty1] != me + 1 and distance <= 8:  # 如果
                                    按以上操作得到的方向走一步就出自己领地，且离对手非常近
                                    直走，不出领地，
            return 's'
        # 下一步没有出领地或者出领地但离敌人不近，可以放心地按该方向走
        else:
            return nextTurn
    # 沿当前方向下一步出领地了
```

当 AI 即将出领地时，发现敌人离自己非常近，只要出去就有被撞的可能，这时不能让 AI 沿当前方向走。必须选择一个下一格不会出领地的方向。如果遇到最极端的情况，三个方向都必须出领地，那么选择这样一个方向：从这个方向出领地后离敌人距离最远，降低被撞的可能性。并且要为出去的 AI 规划好回领地的路径，将走法存入一个名为”VIP”的列表中，该列表专门用于存储特殊情况的走法，只要该列表中有元素，那么会优先将该列表中的元素出列作为返回值。

```
        if distance <= 8:  # 如果离对手非常近，尽量不沿该方向出领地，另谋出路，以防被撞
            # 分方向讨论
            if d == 0:
                if y1 == 0:
                    return 'r'
                elif y1 == size2 - 1:
                    return 'l'
                else:
                    if stat['fields'][x1][y1 - 1] == me + 1:
                        return 'l'
                    elif stat['fields'][x1][y1 + 1] == me + 1:
                        return 'r'
                    else:
                        distance1 = abs(nextx - x2) + abs(nexty - y2)
                        distance2 = abs(x1 - x2) + abs(y1 - 1 - y2)
```

```
                    distance3 = abs(x1 - x2) + abs(y1 + 1 - y2)

                    if distance2 == max(distance1, distance2, distance3):
                        storage['VIP'].append('l')
                        storage['VIP'].append('l')
                        return 'l'
                    elif distance3 == max(distance1, distance2, distance3)
                            :
                        storage['VIP'].append('r')
                        storage['VIP'].append('r')
                        return 'r'
                    else:
                        storage['VIP'].append('r')
                        storage['VIP'].append('r')
                        storage['VIP'].append('r')
                        return 's'
    if d == 2:
        if y1 == 0:
            return 'l'
        elif y1 == size2 - 1:
            return 'r'
        else:
            if stat['fields'][x1][y1 - 1] == me + 1:
                return 'r'
            elif stat['fields'][x1][y1 + 1] == me + 1:
                return 'l'
            else:
                distance1 = abs(nextx - x2) + abs(nexty - y2)
                distance2 = abs(x1 - x2) + abs(y1 - 1 - y2)
                distance3 = abs(x1 - x2) + abs(y1 + 1 - y2)

                if distance2 == max(distance1, distance2, distance3):
                    storage['VIP'].append('r')
                    storage['VIP'].append('r')
                    return 'r'
                elif distance3 == max(distance1, distance2, distance3)
                        :
                    storage['VIP'].append('l')
                    storage['VIP'].append('l')
                    return 'l'
                else:
                    storage['VIP'].append('l')
                    storage['VIP'].append('l')
```

```
                storage['VIP'].append('l')
                return 's'
    if d == 1:
        if x1 == 0:
            return 'l'
        elif x1 == size1 - 1:
            return 'r'
        else:
            if stat['fields'][x1 - 1][y1] == me + 1:
                return 'r'
            elif stat['fields'][x1 + 1][y1] == me + 1:
                return 'l'
            else:
                distance1 = abs(nextx - x2) + abs(nexty - y2)
                distance2 = abs(x1 - 1 - x2) + abs(y1 - y2)
                distance3 = abs(x1 + 1 - x2) + abs(y1 - y2)

                if distance2 == max(distance1, distance2, distance3):
                    storage['VIP'].append('r')
                    storage['VIP'].append('r')
                    return 'r'
                elif distance3 == max(distance1, distance2, distance3)
                        :
                    storage['VIP'].append('l')
                    storage['VIP'].append('l')
                    return 'l'
                else:
                    storage['VIP'].append('l')
                    storage['VIP'].append('l')
                    storage['VIP'].append('l')
                    return 's'
    if d == 3:
        if x1 == 0:
            return 'r'
        elif x1 == size1 - 1:
            return 'l'
        else:
            if stat['fields'][x1 - 1][y1] == me + 1:
                return 'l'
            elif stat['fields'][x1 + 1][y1] == me + 1:
                return 'r'
            else:
                distance1 = abs(nextx - x2) + abs(nexty - y2)
```

```
                distance2 = abs(x1 - 1 - x2) + abs(y1 - y2)
                distance3 = abs(x1 + 1 - x2) + abs(y1 - y2)

                if distance2 == max(distance1, distance2, distance3):
                    storage['VIP'].append('l')
                    storage['VIP'].append('l')
                    return 'l'
                elif distance3 == max(distance1, distance2, distance3)
                        :
                    storage['VIP'].append('r')
                    storage['VIP'].append('r')
                    return 'r'
                else:
                    storage['VIP'].append('l')
                    storage['VIP'].append('l')
                    storage['VIP'].append('l')
                    return 's'
```

当 AI 在领地外时，每走一步就要判断最短路径走法，并且还要判断是否会受到攻击威胁。如果没有威胁则还按原来的围矩形方案走，若有威胁则立即按照返回领地最短路径走。

```
            if 1 < len(storage['meband']) < (len(storage['routine2']) // 2 + 1):  # 只有当
                                        己方刚开始围至少两步或还未围完矩形的1/时，才判断是否需
                                        要调用最短路径2
                mgh = storage['mingohome'](storage['meband'], x1, y1, me + 1, newstat)
                storage['gohomeroute'] = storage['meband'] + mgh[1]  # 为自己沿最短路径走时所
                                        围整个矩形的坐标列表gohomeroute
                minattack = storage['minthreat'](storage['gohomeroute'], x2, y2)
                if minattack <= mgh[0] + 4:  # 当不大于“逃跑最短路径长度minattack+”时4
                    storage['fangyu'] = 1  # 己方进入防御状态
                    storage['routine'].clear()  # 原计划路线取消
                    storage['gohomeroutine'] = mgh[2]  # 改用新计划路线，快速返回己方领地
                    return storage['gohomeroutine'].pop(0)
                else:
                    return storage['routine'].pop(0)
```

2.3 程序限制

当纸带与领地分离时，处理方法考虑不全面，不能有效应对这种情况。

3 实验结果

3.1 测试数据

实验环境说明：硬件配置为 i5-6200U/8G；操作系统为 Win10；Python 版本为 3.6。

程序的测试主要采用了两种方法：一种是利用 visualize.py（可视化比赛工具）进行 AI 之间的比赛，另一种是利用 glory_of_mankind.py(人机对战工具) 通过人工操作方式对 AI 进行测试。

AI 之间的比赛测试主要进行了三种比赛：

1) 本组 AI 与示例 AI 进行比赛，这是对 AI 性能与效率最基本的测试。

测试结果：最初编写的 AI 与示例 AI（AI_normal_wanderer.py）的比赛在不出现 bug 的时候基本可以取胜，进行有针对性的修改后基本上可以完全取胜，之后便不再进行此项测试。

2) 每次改进后的 AI 与改进前的 AI 进行比赛，目的是观察程序的改进对 AI 的实力有多大提升。

测试结果：我们组的 AI 一共有四代，后几代在此项测试中基本可以完成击败前代的任务，胜率可以达到 90% 以上；同时测试中也偶尔会发现 bug 与值得改进的地方，一般在下一代加以修改。

3)AI 相互之间的比赛，是为了发现 AI 还有那些值得改进的地方。测试结果：我们对各代的 AI 都进行了相互的比赛，得到的结果基本上都是不出所料地打成平手，但是也发现了一些有待改进之处。例如，第三代 AI 在测试过程中出现了两个问题：一是在边界如果遇到较为特殊的情况，用于防止撞墙和防止被对手撞击的机制同时发生作用，会导致撞到自己而失败，如图 1。二是 AI 如果与领地之间的联系被对手切断，那么也会无法回到领地进而死亡，如图 2。根据这两个问题，我们在第四代 AI 的程序中进行了改进，但可惜的是由于时间原因，第二个问题解决得不太充分，这也在比赛中成为了我们未能小组出线的重要原因之一。

图 1

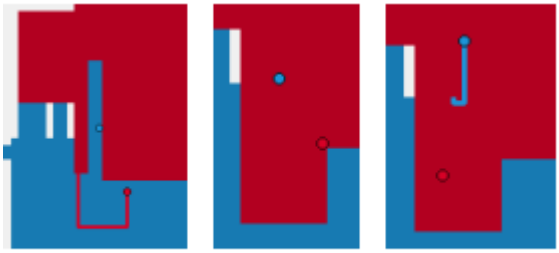


图 2

人工操作对 AI 的测试一般是为了更加精确地确认某种特殊情况下 AI 的反应，用于发现 bug 与程序不完善的地方。一般来说每当对程序进行改进之后，我们都会用人工操作的方法进行测验，来观察改进后的效果，这主要是由于改进针对的情况在 AI 相互对战中不常出现，因此用人机能够更加有效地进行测试。

测试结果：我们对圈地时被截断问题进行改进后进行了测试，结果是在对手不直接攻击时，AI 仍能回到领地，不过还是缺少对对手攻击的应对方法。因此我们做了进一步的改进，尽量降低了被对手攻击的可能性。

我们参加了三次热身赛，图 3、图 4、图 5 为三次比赛的结果：

图 3 第一次热身赛比赛结果

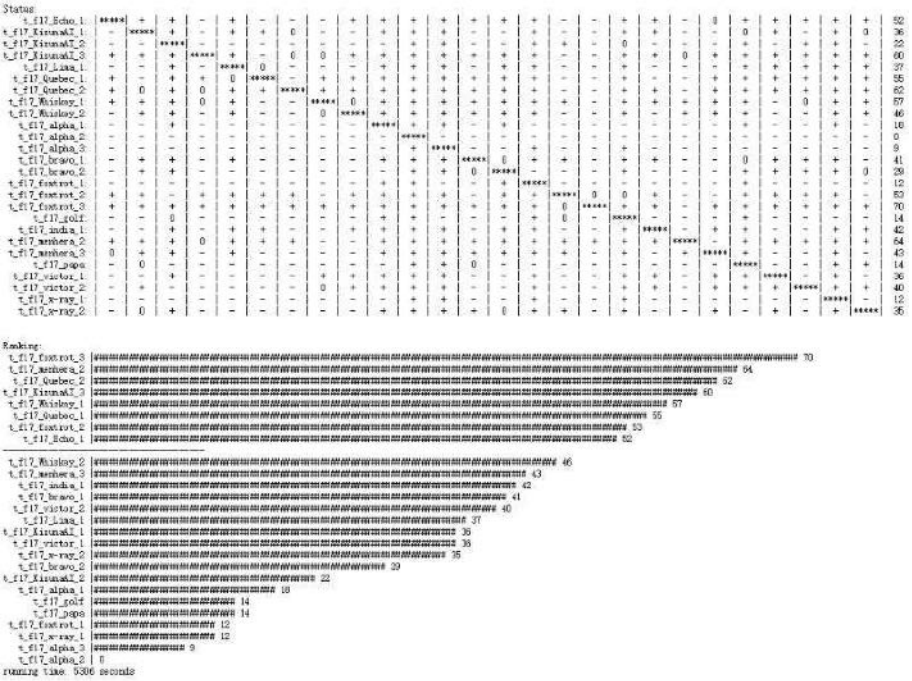


图 4 第二次热身赛比赛结果

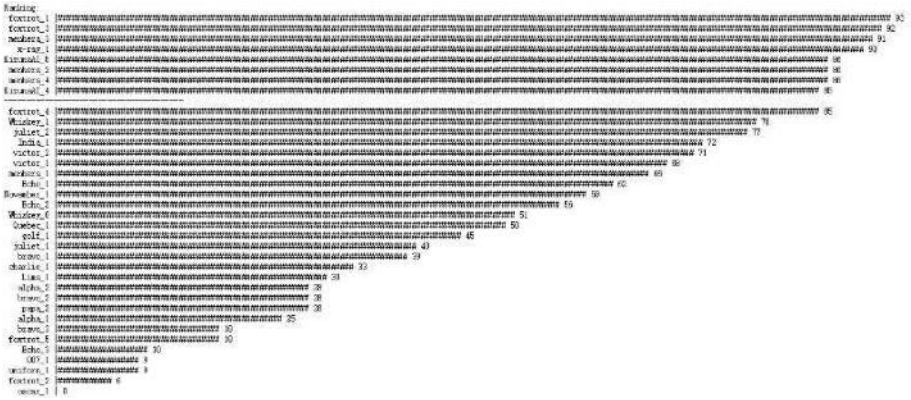


图 5 第三次热身赛比赛结果

图 3、图 4 中上面的表格是相互对局记录，胜利记为“+”，失败记为“-”，平局记为“0”（胜负数相同）；下面的统计表（包括图 5）是对最终积分的统计，胜利积 3 分，平局积 1 分，失败不积分。

第三次热身赛中结果较差是有因为提交时改进后的程序（第三代 AI）完成度不高，存在较多 bug，提交时把它作为 bravo_3 与第二次热身赛的代码一同提交了上去，最终导致成绩并不理想。但是，第三次热身赛后，我们对之又进行了修复和升级，使其在比赛中拿出了不错的表现。

3.2 结果分析

由于热身赛提交的程序代码很不成熟，或者还没有实现我们的策略，这里准备略去对热身赛结果的详细分析，而重点关注最终比赛时的结果。

3.2.1 比赛结果

正式比赛于 6 月 12 日进行，我们与 007、Alpha、India、Juliet、Quebec 分在同一组（W 组），循环赛的结果如图 6 所示，我们组五战三胜排名第三遗憾出局。

```
   +-----+-----+-----+-----+-----+-----+
0: | -0- |  -  |  -  |  -  |  -  |  -  |  0
   +-----+-----+-----+-----+-----+-----+
A: |  +  | -A- |  -  |  -  |  -  |  -  |  3
   +-----+-----+-----+-----+-----+-----+
B: |  +  |  +  | -B- |  +  |  -  |  -  |  9
   +-----+-----+-----+-----+-----+-----+
I: |  +  |  +  |  -  | -I- |  +  |  +  |  12
   +-----+-----+-----+-----+-----+-----+
J: |  +  |  +  |  +  |  -  | -J- |  +  |  12
   +-----+-----+-----+-----+-----+-----+
Q: |  +  |  +  |  +  |  -  |  -  | -Q- |  9
   +-----+-----+-----+-----+-----+-----+


Ranking:
t_f17_india  |######################## 12
t_f17_juliet |######################## 12
---------------------------------------
t_f17_bravo  |################## 9
t_f17_quebec |################## 9
t_f17_alpha  |###### 3
t_f17_007    | 0
```

图 6 W 组循环赛比赛结果

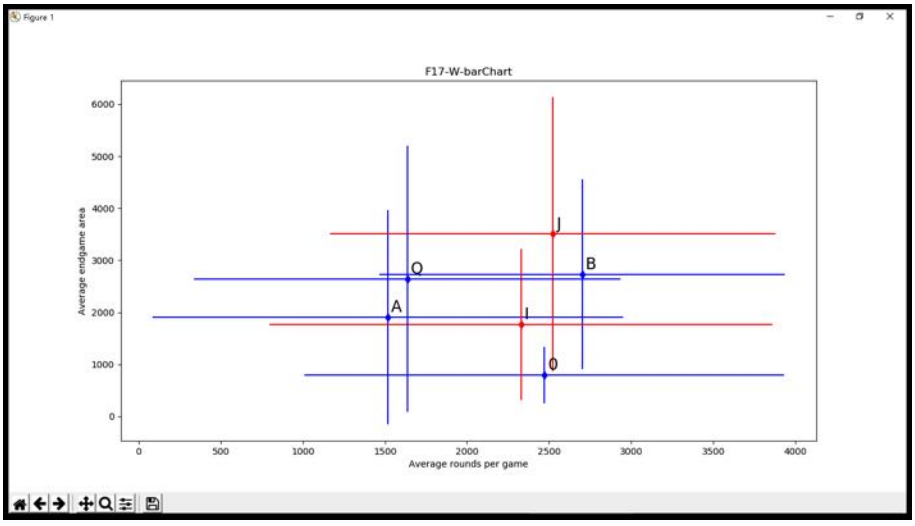


图 7 比赛平均圈地面积和平均回合数的散点图

图 7 为比赛中各组平均圈地面积和平均回合数的散点图。从中可以看出，我们的平均圈地面积仅次于 Juliet 排名第二，而平均回合数则高居小组第一，这反映出我们增强圈地能力、防御为主的战术思想得到了较好的贯彻。

3.2.2 采取的策略及作用

我们的主要策略主要有两个：一个是在领地内部时选择走能取得最佳围地效果的方向，即判断走那个方向可以圈更多的地；另一个则是在领地外圈地时如遇到危险则按生成的最短路径迅速返回。在实战中，这两个策略都发挥了不小的作用。

第一个策略的效果可以在测试的结果中看出。我们开始时的程序只规定在领地内尽量直走，而没有对哪个方向较好进行比较选择；改进之后的 AI 与之前的进行了测试比赛，由于采用此策略后大大减少了在领地内做无用行走的时间，因而明显提高了圈地效率，往往可以取得较大的胜利。一些结果如图所示，其中绿色和紫色分别为改进前和改进后的。

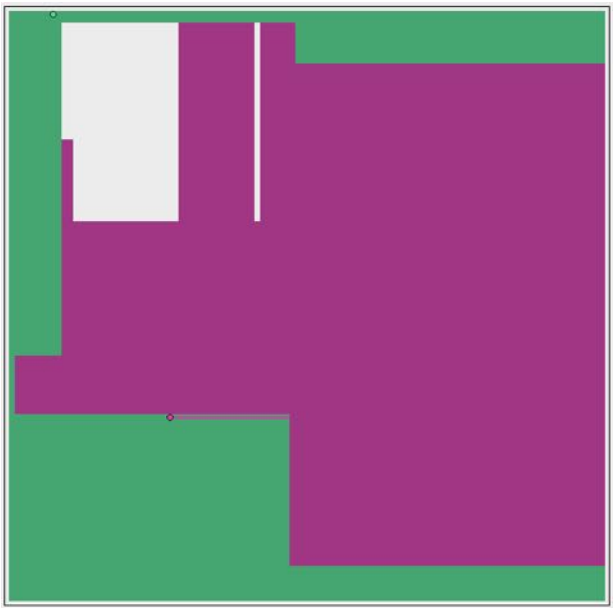


图 8 改进前后 AI 的对战结果之一

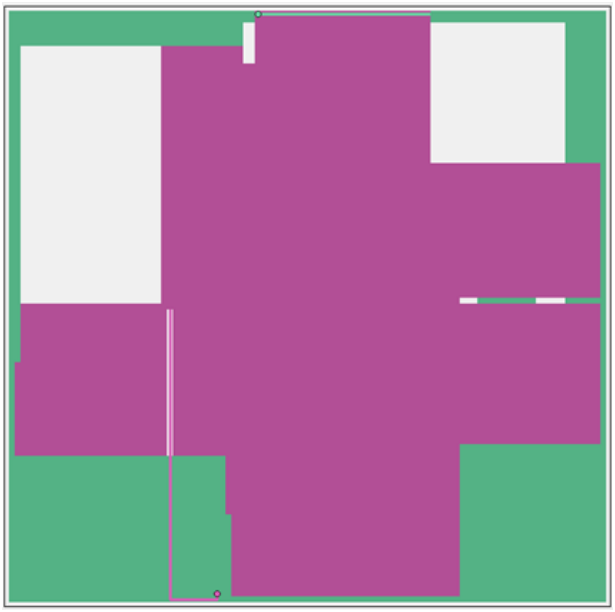


图 9 改进前后 AI 的对战结果之二

第二个策略体现作为我们最重要的防御策略，在比赛中有着很多出色的表现。下面几张图是在后手对战 Alpha 的一场较量中出现的，能比较明显地反映出这一策略的效果。

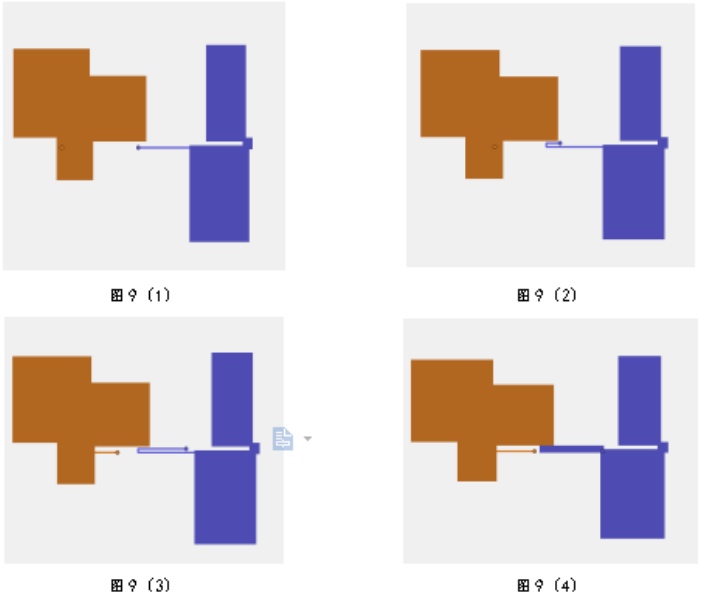


图 9 t_f17_alpha-t_f17_bravo(2) 的对局记录截图

从图中可以看出，我方 AI 通过预判在对方即将撞到我方纸带时刚好回到了领地，化解了这次危险。

3.2.3 运行时间

本算法在运行时间上耗时并不多，基本上可以把时间控制在 0.1s 内，完全无需担心超时问题。主要的运行时间消耗在圈地时最短路径的寻找与领地内行进方向的选择上，这也是我们最为关键的两个函数操作。

3.3 经典战局

3.3.1 厚积薄发，逆转取胜——后手击败小组第二 India

正式比赛中，我们虽然未能出线，但还是成功击败了后来位列小组第二强敌 India。在与 India 的对战中，第一场最为激烈精彩。我们作为后手，在中场阶段陷入大比分落后，但最后靠着稳扎稳打的积累成功逆转取胜。

如图 10，我们和对手 India 分别对应蓝色和棕色。到 1400 步时，双方互不相让，所圈

面积也差距不大（India：1044；Bravo：816），但是总体来看，还是 India 更胜一筹。

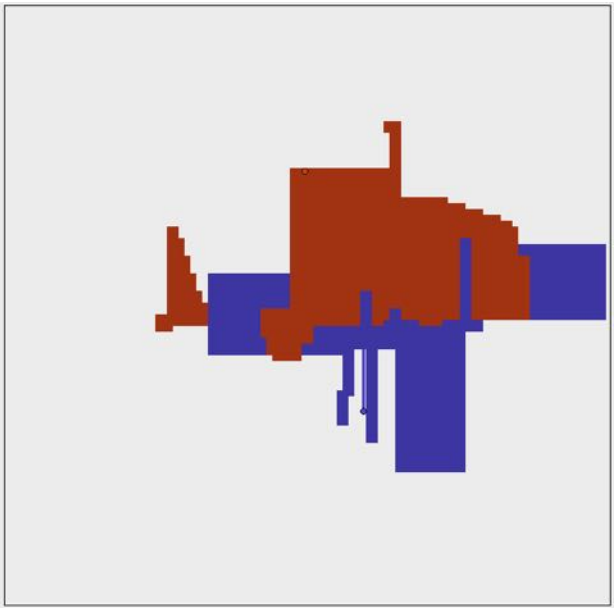


图 10 t_f17_india-t_f17_bravo(0) Step 1400/4000 India：1044 Bravo：816

但是随即，我们的形势急转直下。双方都进行了一次比较大的圈地操作，但是我们没能达到预期效果，面积开始大比分落后。（如图 11）

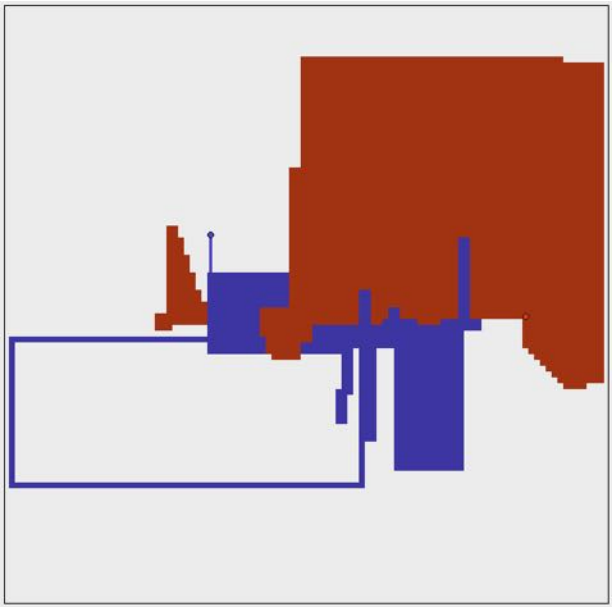


图 11 t_f17_india-t_f17_bravo(0) Step 1701/4000 India：2643 Bravo：785

不过之后，我们没有放弃，奋起直追，接连围出了数块很大的有效面积；而 India 却放慢了一些圈地的脚步。不久，我们完成了反超。（如图 12）

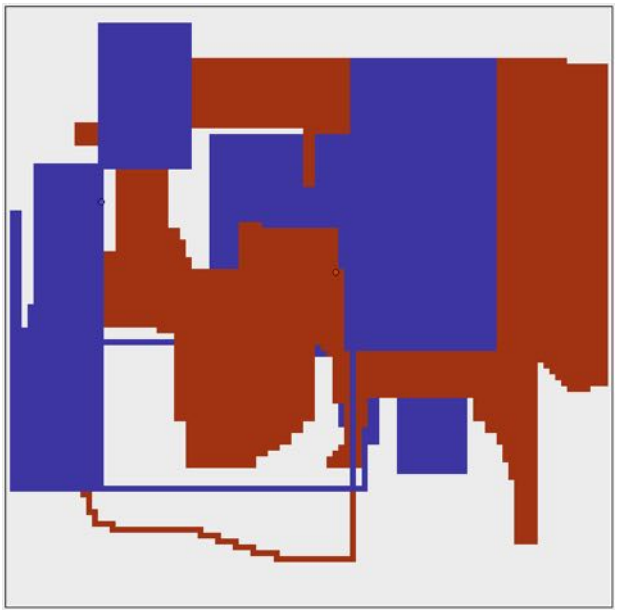


图 12 t_f17_india-t_f17_bravo(0) Step 3335/4000 India：3159 Bravo：3171

最后几百步中，我们越战越勇，利用之前的基础，大比分超过了对手，取得了最后的胜利。（如图 13）

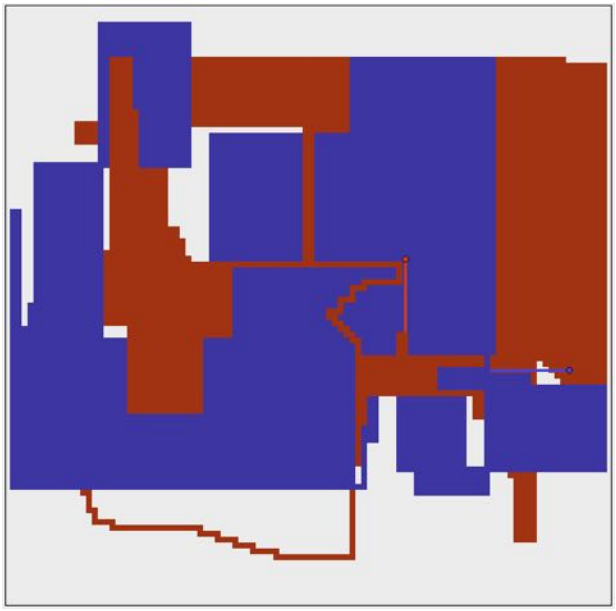


图 13 t_f17_india-t_f17_bravo(0) Step 4000/4000 India：2476 Bravo：4642

3.3.2 固若金汤，以守为攻——与 Alpha 和 Foxtrot 的两场对局

我们的程序以防御和圈地为主，没有设计进攻的手段，但是强大的防守能力往往可以使我们坚持到最后取胜，甚至以守为攻，在近距离缠斗中击败对手。图 14 是正式比赛中对阵 Alpha 的一场较量，重要关头双方缠斗在了一起，相互蚕食，最后还是我们棋高一着，在领地内撞击对手取胜。

图 14 t_f17_bravo-t_f17_alpha(5) 绿色为 Bravo，蓝色为 India

图 15 则是比赛结束后我们与 f17 亚军 Foxtrot 的一场“友谊赛”，绿色和蓝色分别代表我们和 Foxtrot。Foxtrot 有强大的攻击力，一般会从开始就冲向对方，逐步蚕食对方领地或撞击对方取胜。但这场比赛中，我们虽然被不断追杀，但最后还是保证了自己的安全，最后和 Foxtrot 纠缠在一起陷入循环直到比赛结束，依靠前期圈出的面积险胜对手。

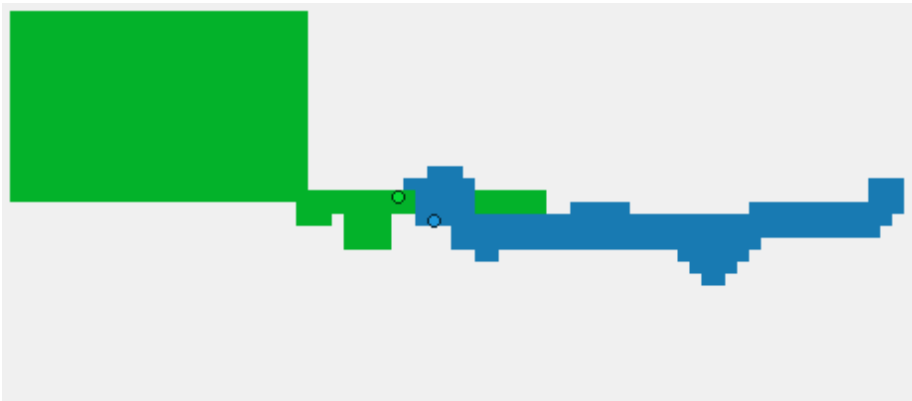


图 15（1）Foxtrot 冲向我们的领地

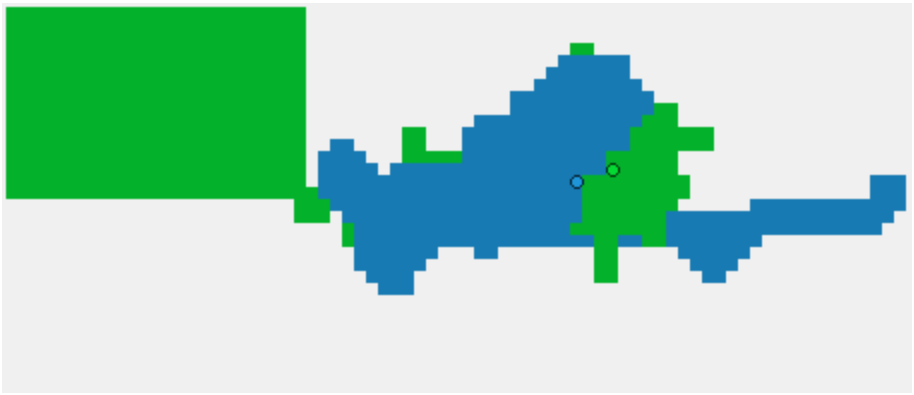


图 15（2）Foxtrot 对我们进行蚕食和紧逼

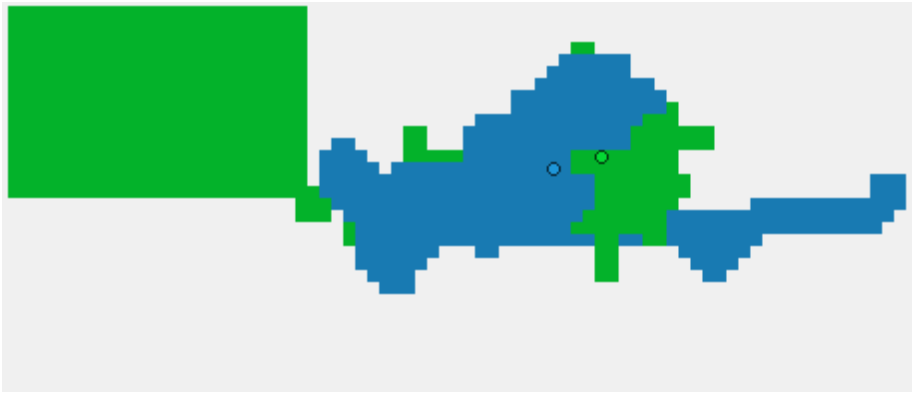


图 15（3）我们抵挡住了攻击，与 Foxtrot 僵持到结束，最后 504：352 取胜

图 15 友谊赛 t_f17_bravo-t_f17_foxtrot

4 实习过程总结

4.1 分工与合作

4.1.1 小组分工

算法策略：全员参与

代码实现：宁湘宇、李想

报告撰写：全员参与：高宇航（第 1、3 节）、宁湘宇（第 2、3 节部分）、李想（第 2

节部分，第 5、6 节）、宋筱泊（第 4 节）

4.1.2 合作过程

第一次讨论会

时间：5 月 28 日 8：40-10：30

地点：四教 404

讨论内容：

第一次主要是小组同学一起熟悉平台与代码。

讨论研究返回对方位置，自己位置，如何确认圈地及方向变换等一系列问题。对于记录的储存和调取做了初步的构想。对于提供的 AI 代码，做了简单的改动，研究行进圈地的策略。

第二次讨论会

时间：5 月 31 日 8：40-10：30

地点：四教 411

讨论内容：

根据提供的新的 AI 代码，进行了改编，开始编写大致的框架，如比赛所需要的基本属性。宋筱泊、高宇航主要讨论战术。初步定下以防御为主的策略。讨论会上编写的程序后手时可以轻松击败 AI，先手时占下风，定为课余时间思考问题，下次讨论会分享想法。

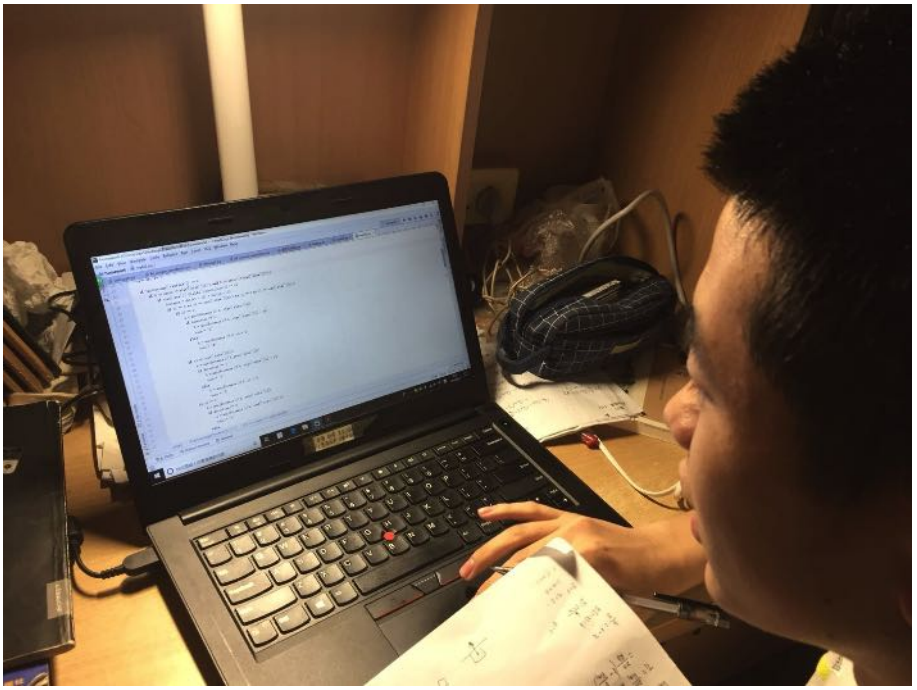
第三次讨论会

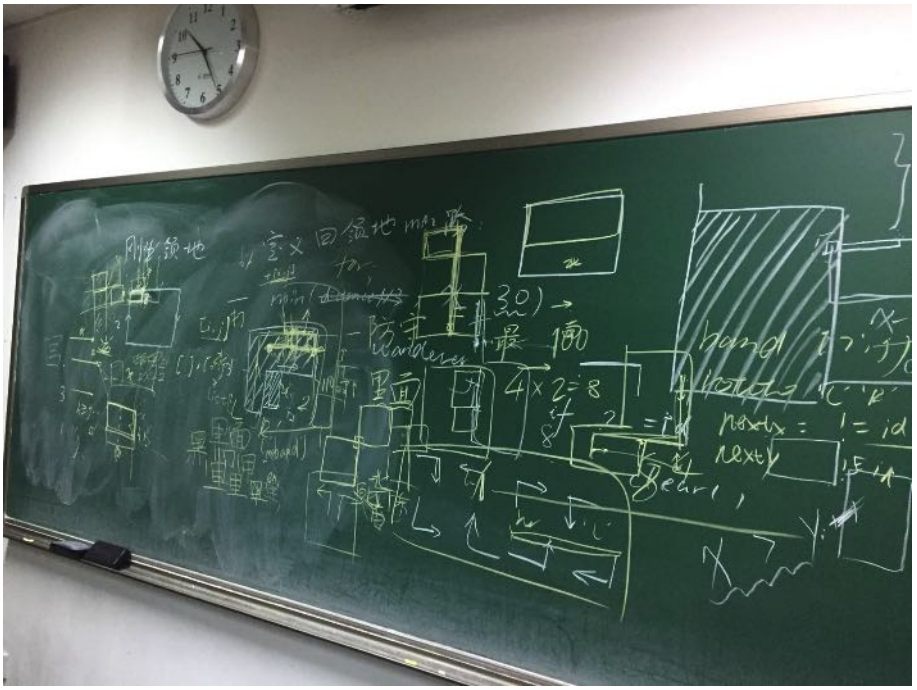
时间：6 月 1 日 9：00-10：40

地点：38 楼 239 宿舍

讨论内容：

对于战术策略有了更为详尽的规划。李想和宋筱泊确定了先划长条，再根据与对手距离画矩形的方案，整体策略仍是以防御为主，缺乏必要的攻击战术，后续需要完善。





第四次讨论会

时间：6 月 4 日 8：40-10：30

地点：四教 407

讨论内容：

特别感谢宁湘宇同学熬夜实现算法。观看了热身赛的记录，总结了自身战术的缺陷，只有防御没有进攻的话，方法太过于消极，如果遇上强攻型对手，非常的吃亏。李想编写了最短的判断与规划方法。组内暂定领地内攻击的策略，来完善总体算法。

第五次讨论会

时间：6 月 11 日 8：40-10：30

地点：二教 524

讨论内容：

赛前最后一次讨论，观看对抗记录，宁湘宇同学对代码在一些情况下会撞墙的问题进行了改善，领地内识别攻击策略，由于转向会产生冲突而放弃，只进行领地内直向攻击。

4.2 经验与教训

1、关于程序与算法：

（1）圈地策略方面，战术过于保守。对于情况分析不够全面。最初阶段的代码策略是避离对手进行圈地，希望以圈地面积最大而获胜。但是当位出一定面积后，就不得不与对手近距离接触，从而导致围地效率变得很低。

（2）主要以防御为主，进攻策略还是简单的被动攻击，只有当对方“送死”时才会进行攻击。所以就导致对方大面积圈地，并且距自己很近时也没有进行攻击，吃亏很大。

（3）一开始是采取领地内尽量直走策略来进行变向，导致有时会出现长时间在领地内不出来而浪费回合数的情况。

（4）考虑情况是忽略了被截断或是背对方领地所围上的情况，从而导致最终比赛时，出现被对方领地截断或围住时而无法正常重新圈地的情况。

2、小组讨论方面：

（1）小组讨论效率过低，进行的组会讨论次数多，但每次实质性进展较小。从而浪费了很多时间。

（2）经验是要多利用课余时间一起熟悉平台与规划策略，组会时集中讨论自己的想法与策略。

（3）没有划分时间来一起讨论所会出现的各种情况，从而导致比赛时有很多情况漏掉，而没有相应的应对策略。

4.3 建议与设想

1、希望老师能讲授一些关于可视化界面的知识，在比赛中可视化创意也占一定比例的分数，例如改变头部的形状变成图片，走过的路径形状进行改变。从而在强调比赛技术性的同时还能增加比赛趣味性。

2、希望今后的数算大作业的提出时间能够提前些，至少技术组开始开发的时间要提前。这一次的数算大作业直到最终比赛前一个星期还有好几次更新，有些更新还涉及到几乎全部程序的修改，这一定程度上会打乱各组的计划，因此希望下一次大作业能够尽早提出，未雨绸缪。

3、赛制方面，将世界杯赛制改为足球联赛积分赛制或 NBA 的常规赛 + 季后赛赛制可能更加合理，因为这样能够使得每两个组之间都能交手，最终的成绩更能反映一个组的代码在整个联盟中的优劣程度。如果这样做会使得电脑运行时间过长，可以考虑将比赛时间延长至两次课，或者一部分比赛线下比。

5 致谢

首先，感谢陈斌老师提供给我们的这次宝贵的数算实习机会。数算大作业不仅极大地提高了计算机课程的趣味性，还给了我们综合应用各种 python 功能语言写程序的机会，也让我们第一次切身接触到了 AI，这都让我们受益匪浅。

其次，感谢老师和助教以及技术组日夜操劳为搭建好平台作出的努力，以及你们对我们在开发算法过程中遇到的问题的及时解答。

最后，感谢小组里每一位成员的付出。全组成员集思广益，每个人都在策略制定、报告撰写等方面贡献了自己的力量，没有大家的积极投入就没有最后取得的成绩。

6 参考文献

[1] 大作业代码仓库 https://github.com/chbpku/paper.io.sessdsa

[2] 数算课 ppt

第四章　F17_Charlie 报告

张昀昊 *、刘杨洛融、张居安、李知霖、唐甘宇、李云鹤

摘要：本文是关于我们小组（F17 Charlie）编写的纸袋圈地游戏 AI 的介绍，我们小组采取的是防御为主，保证自身不出现失误被击杀，兼顾自身的圈地效率，同时当对方采取主动进攻策略时，与之对攻的策略。程序主要使用列表，字典等内置数据结构。采用启发式规则对部分搜索算法进行了优化。本程序在防御方面功能强大，不足之处在于碰到采用类似策略同时圈地效率更佳的 AI 时，会由于圈地少而失败。

关键字：纸袋圈地 AI 防御策略启发式规则

1 算法思想

1.1 总体思路

本学期的期末大作业——纸带圈地改编自游戏 paper.io，但由于游戏形式有了较大的变化，纸袋圈地又与 paper.io 颇为不同，所以我们针对于回合制特点和游戏的新规则进行了研究和讨论，最终确定了我们选择的算法。根据制胜的规则：1. 攻击对手制胜；2. 在触发一定条件时 (野外对碰和回合数耗尽)，比较地盘大小，我们得出了以下三点算法的基本原则：1. 确保自己安全 (有一个好的防御函数)。2. 使自己的圈地效率较高 (有一个好的圈地函数)。3. 在能确保击杀对手的情况下，进行击杀 (有一个好的进攻函数)。在算法具体设计时，我们使用了最朴素的列表数据结构，而并未涉及数，图等，但重剑无锋，大巧不工，我们认为算法思路才是灵魂，具体实现选用最基础的数据结构搭配控制流语句也未尝不可。

算法的整体策略如下：算法主要分为三个阶段：1. 建立大后方 (对应的函数为

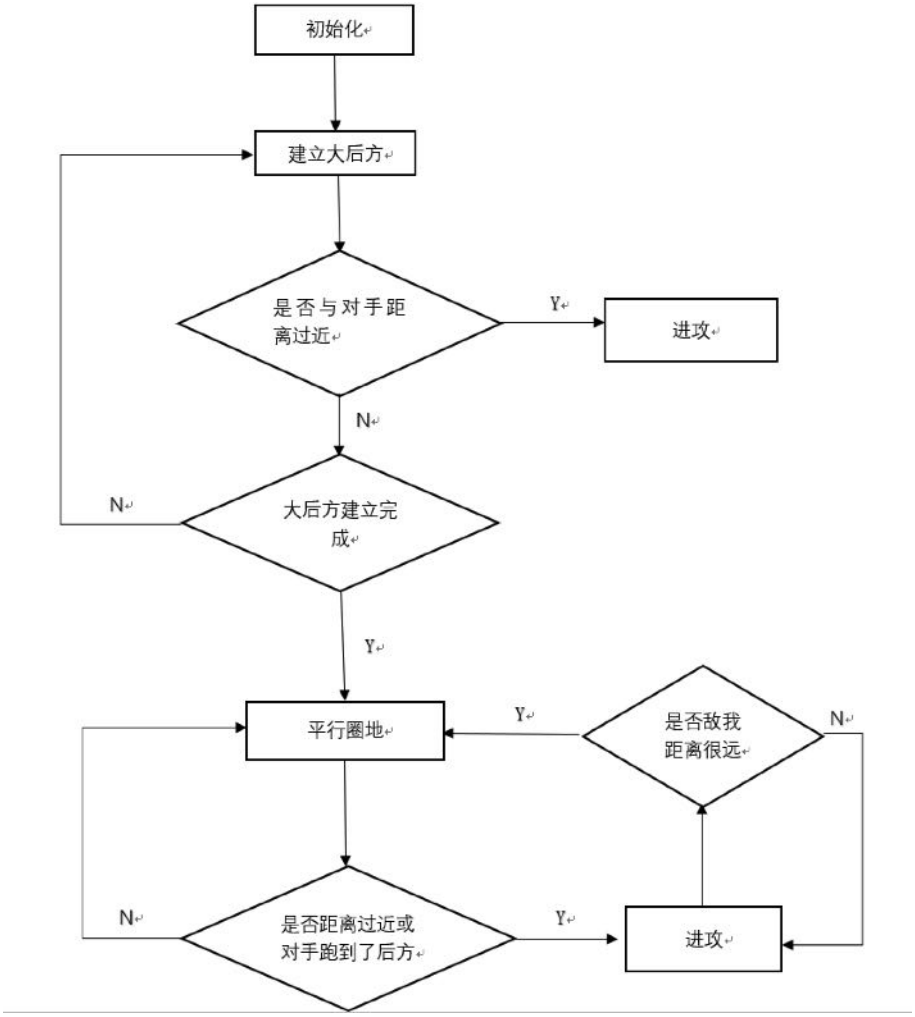
enclosure1)。由于在程序开始早期，一般来说双方的距离相对较远，因此便于进行大块圈地的操作，而在向前和向后两个选择中，我们认为不宜过早地与对手距离较近，这样会使自己陷于不安全的境地，因此我们选择向后进行圈地，尽可能地将初始位置靠里的地方基本圈满，为自己建立一个稳固的大后方。当然，如果在行动时碰到不安全的情况，我们也会及时启动防御工事。2. 平行圈地 (对应的函数为 enclosure2)。在进行完建立大后方步骤之后，我们已经有了一定的圈地面积，后方也相对来说比较稳固。在此基础上，同时考虑到安全性和圈地效率，我们决定进行平行圈地，也就是向前前进一定距离后，在转弯进行平行移动，到达边境之后再转弯回家。与第一步一样，在平行圈地中遇到了不安全因素，我们也会采取相应的防御措施。3. 进攻 (对应的函数为 enclosure3)。在进行到一定的阶段之后，基本进入阵地战状态，安全系数大大降低，平行圈地的策略已经不那么有效，另一方面此时的圈地面积一般来说也比较大了，所以我们决定在平行圈地达到一定的程度就进行进攻。当然进攻的同时也要兼顾防御。进攻的思路主要基于这样两个理念：纸圈在自己地盘无敌；防守就是进攻。毫无疑问，牵着一条纸带进攻无疑远逊于发展一条狭长的地盘去进攻。而我们只需要遵循两条基本的规则，就可以实现复杂的进攻：（1）在自己地盘时，靠近对方纸带移动；（2）不在自己地盘时，迅速返回自己地盘。这样就可以形成一条狭长的攻击带，即确保了自己的安全，又对对手产生了威胁。在之后的近距离搏斗当中，只需要事先设置好情况就可以取得胜利，至少可以保证自己不被击杀另外，安全是在每一个阶段都必须考虑到的问题，因而安全策略至关重要。为了保险起见，我们确立了一个比较强的安全判断条件，也就是针对于原计划将要到达的下一个位置，找到一条回家的路径，如果这条回家路径以及自己的纸带不会受到对方的攻击，那么我们就是安全的，我们就可以照原计划执行，但如果判断结果是会遭受到攻击而败北，那么我们就立即启动防御步骤。

当然，理想和现实之间毕竟存在着距离。在算法的实现的过程中，我们碰到了各种各样的问题，有算法复杂度过高超时，有想实现的功能在实际操作上很难编写程序，当然最多的还是一路伴随着我们的各种各样的报错。但我们还是克服困难，基本完成预计的功能，同时，针对于一些在初期讨论中没有预料到的问题，我们也进行了相应的处理。具体而言就是，在 e1 函数中，针对对方在开局就进攻过来的情况进行了特殊应对，在 e2 函数中，由于 e2 函数的特殊性，在进行安全判断的时候，没有使用安全函数的回家路径，而是另外设计函数找出 e2 函数自身最合适的回家路径。

但是我们仍然有一些到最后也没有实现的功能。最致命的就是缺乏一个通用的圈地策略来尽最大的可能提高我们算法圈地的效率。（这就影响到后来我们在八进四淘汰赛中因为回合数耗尽时圈地较少而被后来的冠军 November 淘汰），此外，在 e1 中未处理对方吃掉

我们底线的地盘 (虽然这种情况下不会报错)，在 e2 中本来预计针对实际情况对前进格数进行调整的功能也未能实现 (虽然对最终算法的效用影响不大)。

1.2 算法流程图



1.3 算法运行时间复杂度分析

由于本程序并未采用太多搜索算法，控制流也以 ifelse 语句为主，所以容易看出时间复杂度主要来源于循环结构。即 enclosure1 中的建立回家的 waylist，对己方，敌方边界的更新，safe 函数中的 getway，以及 distance 函数对于路径的遍历，e2 中的 waytohome 函

数。可以发现这几个函数都是对于一条路径而非整个棋盘的遍历，所以时间复杂度可以近似估计为 O(n)，n 为棋盘边长。

实际对战中也发现由于我们组并未过多地进行搜索，所以对战用时基本不超过一秒，这也反映了我们组对于时间复杂度的优化很好。

2 程序代码说明

2.1 数据结构说明

本算法没有采取复杂的数据结构，也没有使用树和图等数据结构。本算法的核心就是 if-else 判断语句和通过状态参数的切换方法，故未使用特殊的数据结构以及自定义类。主要的线性数据结构是列表。总体上将程序分为圈地 1、圈地 2、圈地 3（即进攻与防守）三大步骤，并根据遍历所得的数据进行条件判断，再决定状态参数值，采取相应状态。

2.2 函数说明

本算法主要使用了三个状态函数，及其附属的功能函数。下面对其进行介绍。

2.2.1 安全函数

安全函数是本程序的核心结构之一。接口为 position,go_home_way。Position 参数是输入的下一个点的坐标。go_home_way 是一个列表，其中记录了一条返回自己领地的路径，0 位是起点，-1 位是终点。默认为 None。安全函数将返回一个 bool 值来判断走了下一步之后是否安全。所谓安全，就是指对方无论如何也不存在必定可以击杀我方的方法。

采用的算法则是一个多条件按优先级排序的判断函数。先进行了 go_home_way 的判断：如果是 None，则自己利用之后会介绍的 getway（）函数来生成一条路径；如果已经提供了路径，则使用提供的路径。之后存在以下判断：（1）是否超出边界。这一点毋庸置疑，超出边界判负是规则级别的失败，故放在第一位。（2）是否在自己领地。这一点也是规则层面的判断，在自己地盘无论是撞击对手还是被对手撞击都能取得胜利。同时，由于第一步已经判断了处于整个棋盘内，那么利用 fields 判断地盘的时候也不会出现超出检索范围的错误。所以将这一点放在第二。（3）判断己方纸卷撞击己方纸带。规则层面的判断

之三。（4）判断敌方纸圈与我方纸带最短距离短于我方回家距离。倘若这种情况实现，则对手存在必定击杀我方的策略，只要对手能够找到这条路径。所以此种情况是不允许出现的。（5）判断敌方纸带与我方回家路径的最短距离小于回家距离。这种情况下对方的击杀难度上较之上一种情况对对方程序的要求更高。但是这种情况下，对方仍有必胜的策略，因此需要避免这种情况的出现。

安全函数保证了状态函数的正常运行，确保了自己不会在一定程度上败北。当安全函数返回 False 时，往往需要改变既有战略，迅速返回己方地盘来，以逃离被击杀的命运。

2.2.2 路径模拟函数

路径模拟函数也是本程序的核心之一。由于时间仓促，未能进行较好的整合，因此代码中出现了两个版本的路径模拟函数，也就是 getway() 和 getway2()。后者是更为先进的版本，因此此处的讨论以 getway2() 为主，简称为 getway（）。

对于 getway2() 函数来说，存在如下接口：here，there，attack，stoppoint。其中，here 是起始点的坐标，there 是目标点的坐标。attack 是攻击倾向，默认为 True。攻击倾向是指，这条路径是靠近敌方还是远离敌方；若 attack=True，则表示生成的路径靠近对手。stoppoint 是停止点，用来控制生成路径的程度。例如 stoppoint=2，则生成路径的长度为 2。这个参数可以一定程度上减少无意义的计算。通过 getway（）函数，我们可以得到这两条路之间的模拟路径。

算法的基本思想是，先枚举出下一步的位置，也就是三个点，再依次删除不合格的点。获得下一步的位置列表后，先判断到终点的距离，形成一个新的包含位置和距离的元组。删除撞己方纸带的点，并按离终点的距离排序，获得最短距离的列表切片。之后如果 attack 为 True，那么按照同样的方法按照离对手近的距离排序，获得最短的切片。如果是 False，则获得离对手远的切片。最后优先添加在自己领地的路径。重复直到达到目的地。

2.2.3 圈地函数 1

圈地函数 1 的功能是，实现对靠近自己初始位置的两个角落的圈地，为了保证 enclosure2 的正常运行，必须保证 y=0，y=100 两侧都顶墙。最终效果如下图所示。

y=0，y=100 两侧都顶墙的例子

整体思路：将一次完整的圈地分为四个阶段，也就是走一个矩形的四条边来完成圈地，但由于对方的干扰与进攻，实际路线会偏离矩形，此时采用贪心算法，给每一个阶段都设置一个优先方向，每一步先判断沿优先方向走是否安全（第 i 阶段的优先方向设为变量 diri），不安全再考虑其他方向。然后在判断安全时，需要调用回家的路径，此处为使圈地最大，直接将矩形的另两条边作为回家路径。

然后考虑到受到对方的干扰威胁，并不是每一次圈地都能够走到尽头，而 enclosure2 能正常工作的条件就是在 enclosure1 中上下两侧的圈地都要顶到墙，所以这就需要重复进行 enclosure1，直到顶到墙。设置变量 halfOK 代表是否有一侧完成了顶墙操作，并返回是哪一侧，变量 whetherOK 代表是否两侧都完成了顶墙操作。

最后，由于完全人为设定了 AI 应当如何走，需要面临的一个问题就是会出现大量的 if-else 语句，但其中很多都是完全重复的，比如向上与向下应当是完全对称的，为了避免这种大量重复功能的代码，我们考虑利用数学运算来减少 ifelse 的判断。具体实现思路如下：游戏中一共有东南西北四个方向（即 x 轴正向，y 轴正向，x 轴负向，y 轴负向），我们将他们分别对应为 0,1,2,3，假设这一步我们要走的方向为 x1，而上一步的方向是 x2，那么由 x0=x1-x2 就可以得到转向，为 0,4，-4 就直走，为 1，-3 就右转，为 3，-1 就左转。设置字典将 x0 与转向对应起来，就实现了对转向判断的统一。

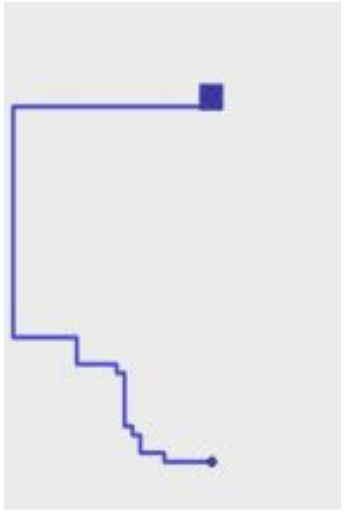
阶段一：这一阶段直接向靠近初始位置的墙边走，首先判断初始位置在左还是在右，在左就往左走，并储存此阶段优先方向。之后就是一旦未撞墙就直行，即将撞墙时，根据对手位置判断转向，对手在上就往下走，反之亦然，然后跳入阶段二。

阶段二：这一阶段是核心过程。

优先方向由之前的第一阶段给出，思路是之前介绍过的贪心算法，回家路径在这种情况下就是左拐向右然后左拐向上，得到一个列表，其中储存元素为这条路径上的所有点。

一旦向下走到即将碰到墙，就准备转向，并且跳入第三阶段，同时修改 halfok 与 whetherOK，而一旦向右走到头了（x 坐标与初始边界 x 坐标相同），那么就代表必须往家的方向走了，而这次圈地也并未完成顶墙操作，此时跳入第四阶段。

阶段一示例



阶段二示例

阶段三：此阶段比较简单，只需要直走，当走到边界处转向，并跳入第四阶段即可。

阶段四：回家，此过程也比较简单，直行一直到自己纸带处在自己的领地之内。

阶段五：这一阶段可以说是阶段一的后续，因为阶段一只在游戏开始时的第一次圈地中调用，之后的圈地过程并不适用，用阶段五代替。此阶段的圈地逻辑较简单，由于始终在自己领地内，所以一直沿优先方向走就行。核心在于与其他阶段的衔接，如果已经完成了上下两侧的顶墙，那么就可以进入 enclosure2，如果完成了本侧的顶墙操作，就向反方向圈地，如果没有完成，就继续在本侧圈地。

2.2.4 圈地函数 2

圈地函数 2 的接口仍然是 storage, stat，具体实现的功能如前所述，即在进行完建立大后方之后，向前进行平行圈地。本函数实际上的核心思想就是分阶段执行相应的指令。首先需要说明的一点是圈地函数 2 在距离对手很近的时候不会被执行，所以不用担心出门就会被对手撞击的情况发生。另外再说明一点的就是本函数体内定义了 way_to_home 函数，它接受下一步预计前进的方向，返回下一步的点以及下一步的一个合适回家路径，从而作为参数提供给安全函数进行判断。它分为两种情况，第一，在向外前进的过程中，回家路径为向 to_explore 的方向拐弯然后掉头，第二，在平行移动的过程中，回家路径为直接转个弯向家走。这时需要的前提是向家走一定要有地，enclosure1 已经尽可能地保证了这一条件，并且如果这一条件没有得到保证，我们就会转入 enlosure3 状态。

圈地函数 2 的起始点就是家，在家时是圈地函数 2 的第 1 阶段，如果在家，就会让纸卷从家里向外出发。这个时候分为两种情况：第一，如果是在边界回家，为了同样能够在边界出门（这样能够提高算法的圈地效率），需要转弯掉头之后再转一次转到边界，这样就设置一个 c2_process 的变量来判断是否应该进入这一过程以及写出进入这一过程之后相应的配套指令。第二，如果是在一个平凡的点回家，那么就朝着 to_explore 的方向 (to_explore 的方向为在平行移动的过程中进行探索的方向) 进行转弯掉头。在出门之后，圈地函数 2 进入第 2 阶段，如果危险，则启动 in_danger1 工事，马上跑回家，而如果安全，则前进，并使 has_forward 变量增加 1，再根据累积变量累计的值判断是否应该进行转弯 (并且是转入 to_explore 的方向) 来平行移动。转入平行移动后，圈地函数 2 进入第 3 阶段，同样地，如果此时发生了危险，就进入 in_danger2 状态，立即转弯并且跑回家，而如果安全，就平行移动直到碰到墙壁。在碰到墙壁后进入第 4 阶段，这个时候进行转弯然后跑回家。这个时候由于已经到达了边界，所以 to_explore 变量需要改变，另外在回到家的前一步，对于一些状态参量也需要进行改变。

2.2.5 圈地函数 3

圈地函数 3 实质上是一个进攻/防守函数。这个函数会在以下情况触发：执行圈地函数 1 时，敌方与我方的距离接近（设定为 <8）；执行圈地函数 2 时，敌方与我方的距离接近（设定为 <15）；本要执行圈地函数 2，但是敌人处于我方地盘，且我方处于自己地盘；正在执行圈地函数 2，但是敌方进入我方领地且我方不在我方地盘。这样的处理方式不一定是最优的，但是进过实践检验不会出错，故本代码采用了这样的判断顺序及判断条件。

圈地函数 3 的核心功能是进攻和防守相结合。最为基础的算法就是：在自己地盘，则向对手方向移动；不在己方地盘，则迅速返回自己地盘。如果能保证自己走出地盘后可以安全的返回自己地盘，那么这就是一个绝对安全的策略；同时正在向对手移动，那么如果对手出错，则将获得游戏胜利。而圈地函数的核心，就是判断什么时候走出自己地盘，以及如何返回自己的地盘。

圈地函数 3 开始有较多的准备工作。首先定义了简易的转向函数，用来将决定好的路径转换为规则中说明的指令。其次定义了一个游走函数，当处于自己地盘时触发会在自己地盘游走。若不在自己地盘则会迅速返回。定义完函数后，首先建立了当前位置下的周围八个坐标（简称九宫格）中属于自己领地列表。之后，在九宫格和我方领地边界中选择一个作为之后搜索所要使用的列表。九宫格不为空时，选择九宫格，并当上一个回合的位置不是唯一的待选位置时，将其删除。这样就保证了可以在和距离相等的步数上返回自己的领地。

如何判断是否走出地盘：（1）判断是否可以追尾。所谓追尾，即走到敌方的上一步。如果对方在对方地盘边界上移动的话，这样是绝对安全且至少可以获得两块地盘的。但是，九宫格内只有上一步是自己地盘时，取消追尾行动，因为这时候无法在两步之内返回自己的领地，存在被击杀的风险。如果不能追尾的话，判断下一步是否在自己的地盘：如果是的话果断走，不是的话继续判断。下一步不在自己地盘的话，如果和敌人的距离为 1，毋庸置疑会被击杀。此处有必要进行说明：当时采取的策略是避免正碰。所以采取游走函数。如果下一步和敌人直线距离为 0，即撞击对手，判断对手是否在其地盘，根据此判断是游走还是击杀。同时，和敌人处于一定距离时，考虑到情况的复杂性和时间因素，采取了强安全的模糊处理，在距离 2-4 的场合都采取了游走函数。

如果不在自己的地盘的话，立马返回自己的地盘。但是如果追尾仍然可以进行，那么继续追尾。

2.3 程序限制

本程序出错的概率极低，至少就目前而言，我们未发现任何的出错情况。最终版的测试中，不曾出现过一次报错；且在和对手的比较中，所有的失败都是由于回合数耗尽，圈地面积小于对方而失败。

3 实验结果

3.1 测试数据

实验环境说明：

硬件配置：i7 操作系统：win10Python 版本：3.6

测试方法：在自己的电脑上让己方程序与 Al 进行对战，多次重复实验，收集数据及失败原因，并对结果进行统计。

	Charlie	AI	回合数	己方胜负	原因
一	6152	2271	3405	胜	K.O.
二	4540	2576	3112	胜	K.O.
三	5555	2260	2407	胜	K.O.
四	4734	1632	2523	胜	K.O.
五	5537	2736	2678	胜	K.O.
六	6949	2277	4000	胜	占地多
七	3635	1760	1085	胜	K.O.
八	4650	2289	1475	胜	K.O.
九	6600	2289	2400	胜	K.O.
十	5893	2446	3219	胜	K.O.
十一	6963	2763	3801	胜	K.O.
十二	5551	2928	2404	胜	K.O.
十三	2827	1284	1005	胜	K.O.
十四	4168	2252	1693	胜	K.O.
十五	4747	1972	1411	胜	K.O.
十六	5584	1944	2521	胜	K.O.
十七	6228	2517	2159	胜	K.O.
十八	5235	2189	1726	负	被 K.O.
十九	5229	1457	2072	胜	K.O.
二十	6052	1959	2451	胜	K.O.
二十一	6526	1957	2284	胜	K.O.

二十二	6454	2358	2202	胜	K.O.
二十三	4759	2670	3319	胜	K.O.
二十四	6558	2649	2957	胜	K.O.
二十五	5569	2697	1845	胜	K.O.

表一　测试数据记录（节选）

实验进行了上百次，但是由于部分时候是边跑边进行程序的修改，在运行的过程中程序是不断在升级的，故只截取了两次修改之间的部分数据，以控制单一变量。但是依旧统计了总的数据。胜率 94.51%，其中 K.O. 取胜 90.24%，圈地取胜 4.27%，失败的场次中被 K.O. 的比例为 4.88%，撞墙失败 0.61%。

3.2 结果分析

总的来看，本算法的胜率还算是很高的，从两者之间的圈地面积来看，本算法是一个圈地效率比较高的算法。从取胜原因的概率分布来看，绝大多数情况下都是 K.O. 取胜，说明本算法在进攻方面的时机判断以及进攻的效率也是很高的。这表明这个算法是一种攻守均衡的算法，在保证大的圈地面积的前提下，只要抓到了正确的时机，也会冲出去一击必杀而取得胜利，相对来说是比较完备的一种算法。

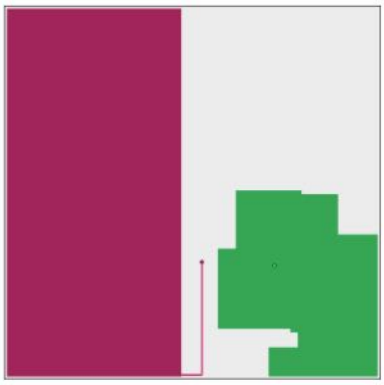
失败的场次中，没有一次是圈地面积小失败的，却有几次情况是被 K.O. 的，这说明我们在防御的步骤中是出现了问题的，在对于己方是否安全的条件判断以及回家路径的选择上可能出现了些许问题。经过分析，是由于原来属于己方的地块被敌方圈走，导致原先是绝对安全的路径突然变得不安全了，即使这个时候立马选择最短路径回家也是会失败的，这给我们提供了一个警告，并指明了修正代码的方向。

在失败的场次中，出现了一次非常奇怪的现象。某一次，我们的纸头居然撞到了墙上，而且是唯一的一次。经过过程的重演，我们发现是在运行到了整个游戏区域的右上角的那个点时，一头向上撞死了，经过分析，是由于每一步在判定前进方向时是需要有一个输入的，由于位于角上，传回来的是一个空列表，导致无法进行判断。不过由于能够跑到右上角的机会过于微小，而且只需要加一组特殊情况即可，紧迫程度并不高。不过这种 Bug 还是挺让人出乎意料的。

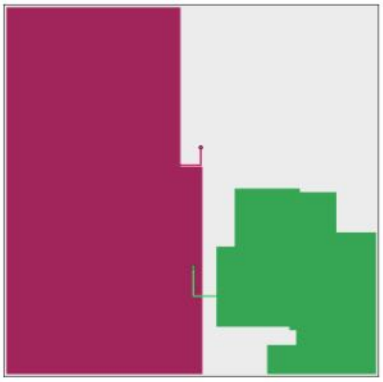
算法在运行时间上是远低于课件所要求的，是我们预料之中的情况。我们的算法没有运用到线性，树，图之类的东西，主要的时间花费都是在遍历以及 if-else 语句的判断上。

3.3 经典战局

本次战局是我们第一次出现回家路径被圈而失败的情况。



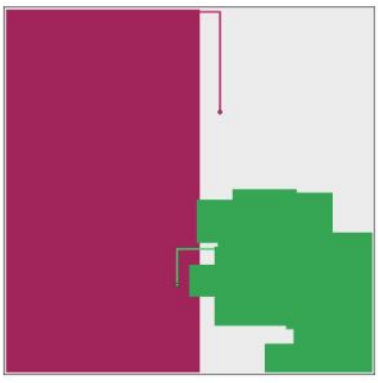
图五



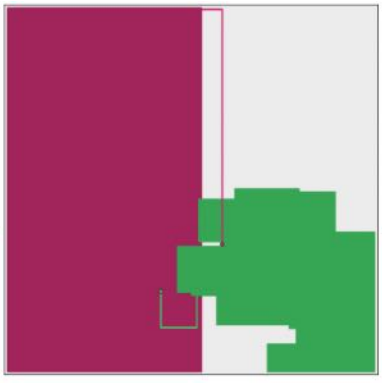
图六

在图五中，我们的程序执行的是简单的平行圈地程序，一切都有条不紊地进行着。到了图六，我们发现敌方已经进入到我们的领地内了，此时可以看见，由于过程中敌方的靠近，我们是执行了一次成功的防御措施进入到己方领地的。

图七中，我方正在向下移动，而敌方开始在我方的地块中圈地了。图四，此时可以看见我方的纸头即将进入到被圈走的我方地块的上方，而敌方正在向上向我方逼近，此时我方还是安全的，故继续向前前进。

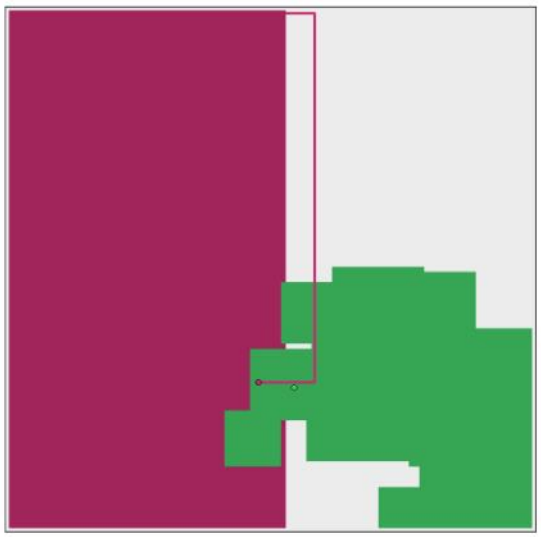


图七



图八

图九是失败前的最后一回合，敌方圈好了地之后，开始进逼，我方察觉到危险开始回到己方领地，但是由于己方领地被圈走，回去的路径大大延长，最后导致仅差一步就回到家的敌方被敌方切断了纸带。如果敌方没有在我方进行圈地，那么回去的路径是安全的，圈了地之后，这个问题就几乎无解了。在人机对战中，人类的胜利正是利用了这个判断条件的突然改变而获得胜利的。



图九

最终的解决方法是一个强条件方案，就是在判断到敌方进入到我方领地之后，立马回到己方领地，来避免回家路径上的地块被圈走而失败的情况。在加了这个强条件之后，经过了两百次的实验，没有再出现此类失败的情况。

4 实习过程总结

4.1 分工与合作

各组员的具体分工：李知霖：整体框架，边界确定函数；张昀昊：圈地 1；张居安：圈地 2；刘杨洛融：防御，圈地 3，代码的整合；唐甘宇，李云鹤：部分调试工作。

由于我们小组六名组员来自三个不同院系，所以平时的面对面交流几乎没有，我们组采取的措施是提高组会的频率，并延长组会时间，每一次组会时长基本都在三小时以上，两周的时间一共是开展了五次组会。平常基本依靠微信群联系，不过我们组大家都是高中

同学，所以在平常联系交流方面比较方便。

在具体的分工方面，我们考虑到多个人实现同一部分代码时，交流并不方便，而如果实现多个版本的话，会降低工作效率，所以我们在第二次组会时就确定了整个的思路，将需要实现的防御，圈地，进攻抽象成函数，并确定了各个函数的接口，然后一个人负责一个函数，以及一个人专门负责整体的框架和结构。这种方法带来同时也带来了一些问题，最主要的是当大家的代码对接时，每一部分都尚未调试过，而且也只有具体的负责人清楚代码的具体实现，所以将会有很长的时间用来 debug，这个部分需要大家的集中讨论，我们组会的大部分时间都是在集中 debug，这个缺陷在某种程度上降低了组会的效率。

4.1.1 历次组会

1. 前两次组会在艺园二楼展开，边吃夜宵边讨论。

前两次组会主要是确定整体思路，由于是初次做这种 AI 类型的大作业，大家都没有一个清晰的思路，就都谈了下自己关于这个游戏策略的看法，首先讨论得出了再具体游戏中容易发生的对碰在 AI 对战中并不经常发生，而且在对局中，存在一方拥有主动进攻的优势，可以将对碰转化为自己一方对对方的侧碰。尽管这一结论在最后并没有用到。然后大家又谈了一些关于防守进攻的判断，最后得出，想要得到一个十分精准的关于自身是否安全以及能否进攻的判断需要用到对整个局面的遍历，这无疑会使时间复杂度超标，所以我

们最后采取的是一种“强安全”的策略，即“如果能指定一条回家的路线，且在回家过程中对方无法撞击我的纸带，那么我必定是安全的”，其他情况可能我方安全，但我们不予以考虑，只要不满足强安全策略，就不安全。接着明确了我们代码的整体思路，首先保证圈地效率，使用分步圈地，第一步把自己附近的两个角落圈好，之后使用之字形圈地，圈地过程保证自身的绝对安全，先实现这两个基本功能，进攻之后再添加。最后进行了具体分工。

2. 第三次组会在肯德基进行，方便刷夜。（并没有图）

第三次组会的主要任务是将之前大家分工的部分整合在一起，并进行调试 debug，保证能够打过白痴 AI。

这次组会并没有想象中成功，主要是还是低估了 debug 的难度，之前大家的代码都只能保证语句没有错误，而逻辑上的问题必须要调试后才能发现。这次组会最终进行了五个小时，而且还未完成任务。

3. 第四次组会在物院进行，同样是为了刷夜。（也没有照片）

这次组会的主要任务是将之前代码中的一些细节问题解决掉，最后结果还算满意，除了几个尚未处理的问题，比如进攻，可以保证在与 AI 的对战中不报错，不出现低级错误。

4. 最后一次组会在二教 513 展开

这次组会则是讨论进攻策略，以及如何优化对于领地边界的确定，如果直接用搜索算法确定边界的话，时间复杂度将会很高，这次组会最后利用减少查找次数，筛选查找对象将算法优化了很多。

4.2 经验与教训

我们小组做的比较好的一点组员之间的交流很密切，尽管大家负责的部分不相同，但几乎所有算法都是大家集体讨论的结论，而且在编写代码时，就会经常在微信群里讨论代码的对接，比如函数的接口与调用，这极大地提高了工作效率。

其次一个就是要把握住整个问题的重点，比如我们第一次组会上讨论了很久的进攻必胜条件最后根本没有用上，本质上就是这种情况很难出现，而且实现起来过于复杂，到后面就发现这个游戏还是要立足于自己的安全，然后才是圈地与进攻，所以纠结于一些局势复杂的情况是无用的，对战双方对安全性的要求导致很多复杂情况根本不会出现。对于一个问题，第一步并不是想着把它完整地解决掉，而是从整体出发，考虑整体的策略比讨论一些细节的战术有用的多。

不足之处也有很多，比如对纸带圈地问题的理解。在最后的竞赛过程中，我们发现我们对游戏进攻的理解存在很大的问题，我们在之前的讨论中忽略了对手直接蚕食我方领地的打法，而之后其他组的代码与复盘证明了这个策略是极佳的。这其实也导致了我们组止步于八强。

其他方面比如分工也存在很多可以改进的地方，一个是缺少一个专业的调试人员，导致经常出现三四个人在一起调试的情况。还有一个问题就是对于热身赛不够重视，所以在设置 ddl 时就没有考虑到准备热身赛，最后一次热身赛只是交流一份半完成版的代码上去，也就导致大家的思维被局限了，没能参考对战其他组的策略。

4.3 建议与设想

我们组对于分组以及竞赛赛制都很满意，现在的赛制既保证了公平性，同时也添加了一定的抽签运气因素，更加贴近于一项比赛。关于基础设施代码方面想提的一个意见就是在发布了基础设施代码后，希望能够减少对于函数接口的修改，这样经常会出现更新了平台代码后就莫名其妙报错。

其他的一些设想，大作业发布时间能否提早一些，因为大作业和课程进度没有直接的关联，所以提早发布可以让大家准备更加充分。另外除了现在采取的杯赛制度，可以考虑添加联赛或是排位打法，两者综合得到最终竞赛成绩，这样可以减少游戏策略的克制导致的出局。

5 致谢

感谢陈斌老师以及技术组同学提供的对战平台，以及所有与我们组进行过友谊赛的同学，在同你们的比赛中，我们发现了许多漏洞，此处特别感谢。

同时需要特别感谢的是孟舜英学姐和她所在的 N17_Quebec 组。非常非常幸运的是，在赛前的两个小时里，我们组的刘杨洛融偶然认识了 N17Quebec 组的孟舜英学姐。孟学姐组使用的算法相当的惊艳，是和我们完全不同的智能算法；我们两组进行了相当多的友谊赛，这些友谊赛提供了相当宝贵的数据，并借此修复了我们组一直以来面临的一些根深蒂固的问题。正是由于孟学姐的帮助，我们最终才能做到不被 KO 一次的地步。可以说，在遇见孟学姐之后，我们的代码产生了质的飞跃。在这里再一次感谢孟舜英学姐和 N17 的 Quebec 组。

6 参考文献

[1] Brad, M.; David, R. Problem Solving with Algorithms and Data Structures using Python, Luther College

[2] https://docs.python.org/3/

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

第五章　F17_Delta 报告

蔡家骥 *、富云齐、蔡紫葳、蔡紫菁、隋绍丹

摘要：本组纸带圈地算法的总体策略分为自己在领地内、领地外两部分。如果自己和敌人同在己方领地内且可以完成击杀，则发起攻击，否则找到最近的出口；在领地外通过离家距离和敌我距离的大小关系判断是否安全，安全则扩张地盘，否则立即回家。其中在计算距离的时候用到了图和优先队列的结构，应用了 BFS 算法和 Dijkstra 算法。程序能够处理大部分常见情况以及一些复杂场景，局部摩擦时足够精细，很难击杀。

关键字：图、优先队列、BFS 算法、Dijkstra 算法

1 算法思想

1.1 总体思路

1.1.1 算法策略

限于开发效率，程序采取纯逐步分析的策略，即并无预测以后数步的路径并存储的过程。程序分为我方在领地外的情况、我方在领地内的情况两部分，两种情况完全分开进行决策。

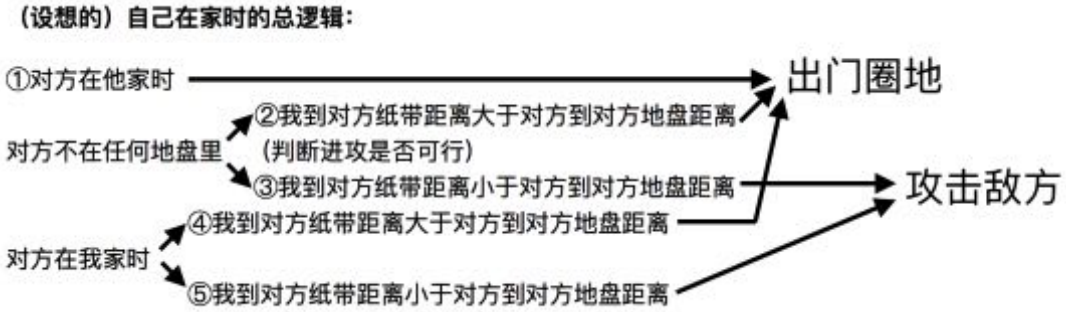
领地外情况总体上采用“敌不犯我，我不犯人”的策略。每一回合都判断敌我距离与回家距离的相对大小，如果安全，按照恒定距离的轨道扩张，否则立即回家。

在领地内时，若对方也在己方领地内，则判断对方回家距离与己方攻击距离的大小关系，以此判断能否发起攻击，可以则进攻；不能发起攻击，或对方不在己方领地内时，就找到离自己最近、且离对方有一段距离（没道理地设为 10）的出口离开领地。

另外，开局时先绕一个 12*12 的圈（这是绝对安全的）。

1.1.2 开发过程

一、领地内



最后实现：属于 (4) 则进攻，否则出门

进攻部分：

一开始，求自己到对方纸带最短距离使用的算法是遍历整个地图。

后来考虑到时间开销太大，觉得可以遍历 border 包围的矩形（border 是 match_core 里面的东西，但是可以自己写一个一样的放在 storage 里面），或者直接遍历对方纸带点集。

再后来学了图的算法，改用 bfs（知识就是力量!）。

出门部分：

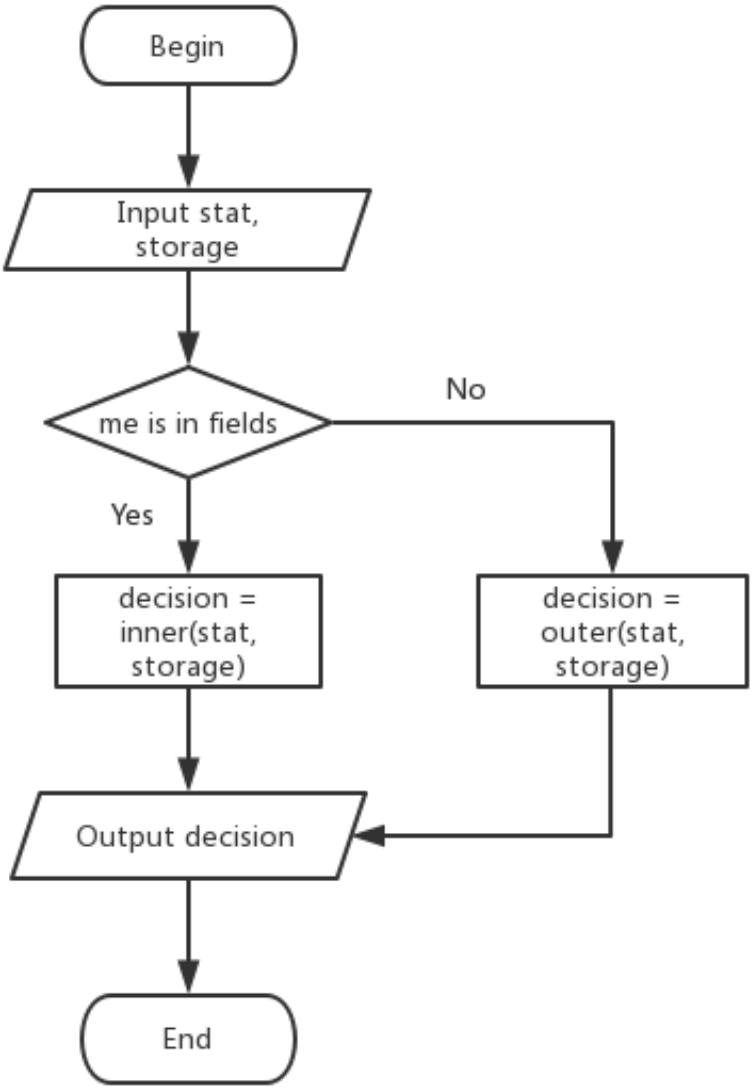
最初设定是从南/北面中比较近的一边出门。界定“比较近”的方式是求平均距离。

后来觉得这个方法有失合理性，改成从离自己最近的边界点出门。但是“先走到边界上，再出门”的模式需要考虑非常多特殊情况，打了很多补丁之后程序的逻辑变得非常不简洁，最后还是选择改用 bfs。

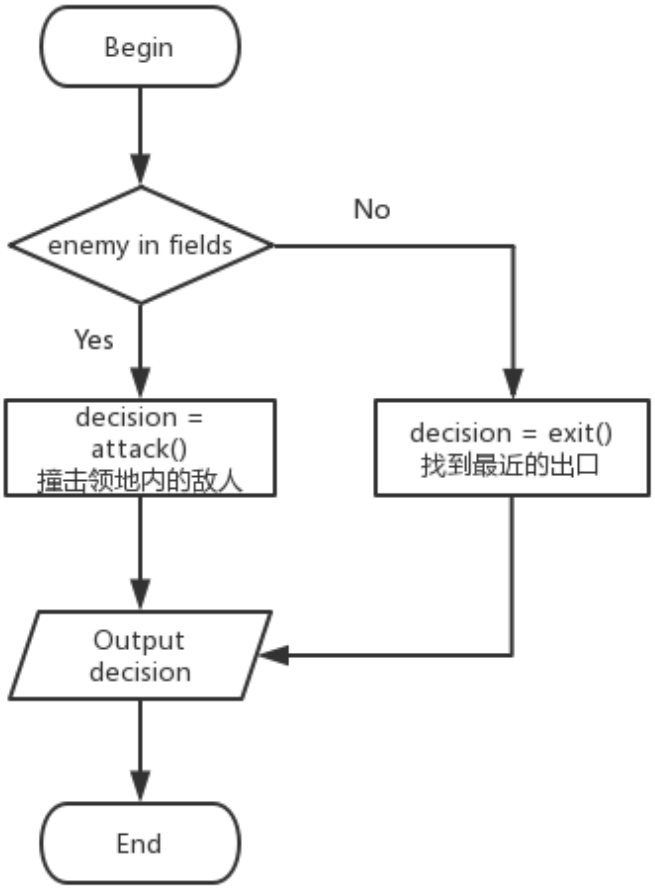
二、领地外

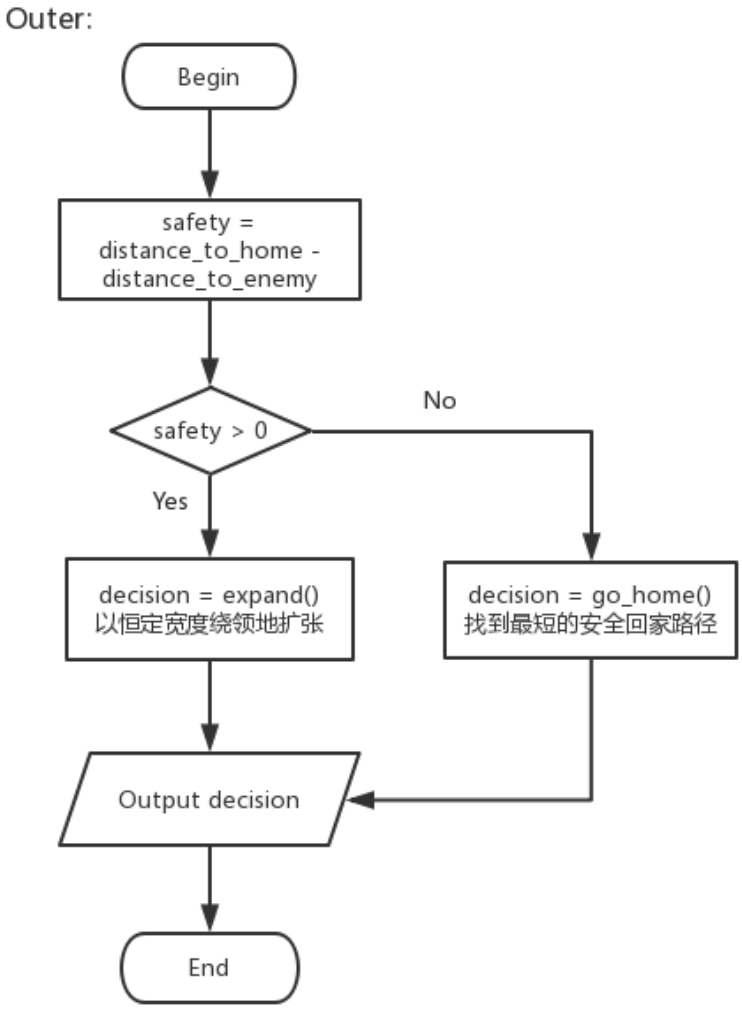
开发过程丢失

1.2 算法流程图


Begin
Input stat, storage
me is in fields
No
Yes
decision = inner(stat, storage)
decision = outer(stat, storage)
Output decision
End

Inner:


Begin
enemy in fields
No
Yes
decision =
attack()
撞击领地内的敌人
decision = exit()
找到最近的出口
Output
decision
End



1.3 算法复杂度

维护 storage：

维护纸带节点只使用 if 结构，时间复杂度为 O(1)；

维护领地边界节点调用了 findborder() 函数，时间复杂度为 O(领地周长)。

记录一些距离：

调用 mehome()，如可直达，时间复杂度为 O(领地边界拐点数 * 纸带拐点数)，如不可，复杂度为 O(领地边界拐点数 * 纸带拐点数 + 纸带拐点数!*log(纸带拐点数))；

调用 enemyhome()，时间复杂度为 O(对方领地边界拐点数)；

调用 dis2band('enemy', 'me')，如可直达，时间复杂度为 O(己方纸带拐点数 * 对方纸带拐点数), 如不可，复杂度为 O(己方纸带拐点数 * 对方纸带拐点数 + 对方纸带拐点数!*log(对方纸带拐点数))

决策：

先进行一些维护：都是 O(1)；

在领地内：调用 play_inner()，时间复杂度大约为 O(最近攻击距离 ^{2})

在领地外：调用 play_outer()，其中：

- safety(), O(1)
- glide_wall(), O(1) / O(己方领地边界拐点数)
- routine(), O(己方领地边界拐点数)
- domoi(), O(己方领地边界拐点数)

综上，时间复杂度为 O(己方领地边界拐点数)

扩张和回家均需要大幅使用求距离的函数，尤其路径上有障碍的情况，耗时较大。

2 程序代码说明

2.1 数据结构说明

使用了 Vertex、Graph、PriorityQueue 类和一些 Python 的基本类。

计算最短距离时，使用的是 dijkstra 算法，为减少节点数量、避免超时，只把纸带拐点（计算敌我距离时）或领地边界拐点（计算回家距离时）加入图中，能直接互通的两点建立边，权为距离。

2.2 函数说明

findborder(): 传入字符串’me’或’enemy’, 返回相应的领地边界与节点，在游戏开始时和有一方头部进入自己领地（前一回合不在领地）时调用。先确定搜寻的行 line：如头部在家则从此行，否则考虑纸带第一个节点，如出领地时方向为左右则从该行开始，为上下则相应 ±1。从 line 行自左至右搜索到第一个属于领地的点（不好的方法，可能被飞地挡住）。之后顺时针绕一圈，大致为让自己左侧为领地外，得到边界，在转弯时录此点为拐点。

strdis2band(hd,tarc,avo,is_circle=False): 传入头 hd、目标的拐点列表 tarc、避免撞到的部分的拐点列表 avo、目标为线形还是环形 is_circle, 返回 hd 到目标的距离 d0 和目标点坐标 p0。方法：遍历一遍 tarc，对每个点，得到该点与头部距离（横坐标之差和纵坐标之差的绝对值之和）d1。如从 hd 到该点与下一点连线所在直线的垂足在连线段上，则求该点与下一点连线与 hd 距离（| 两点相同的坐标 -头部该坐标 |）d2。将两者与 d0 比较如此距离小于 d0 则看是否可行，如 True 则替换之，p0 则替换为该点坐标（前一种情况）或两点连线上的垂足坐标 p（如两点横坐标相同，则横坐标不变，纵坐标为 hd 的纵坐标）。可行性判断通过对 avo 逐段使用 intercept 检查是否拦截 p 与 hd。最后返回 d0,p0。

intercept(k1,k2,h1,h2): 传入 k1,k2,h1,h2 均为点坐标，返回 k1,k2 连线是否拦截 h1,h2（布尔值）。其中 k1,k2 有一坐标相同。即判断是否 k1k2 相同坐标在 h1h2 中间，而且 k1k2 相异坐标又包住 h1h2 的该坐标。

mehome(): 返回我回家的距离，较为准确但复杂的部分其实并没有被应用到。立一 flag, 先用类似 strdis2band 的方法估测要直接走到目的地，如果其中遇到不可行则 flag 为 false。如最后 flag 为 true 则直接返回 d0,p0 否则对每个’顶头’用 strdis2band 求到家距离，并对顶头和我方头部建图，相互不被拦截的连上边，权为直接距离，使用 dijkstra 算出到头部的距离，再与各自到家距离相加求最小值。

routine(param)：在距离领地 param 的轨道上巡航，扩张领地，每次返回维持在轨道上所应采取的转向。具体做法为遍历前、左、右三种选择，调用 mehome() 函数分别求得回家距离，返回维持距离在 param（偏离值允许为 +1）的选择。

enemyhome(): 返回敌人回家的距离，向小估计忽略阻挡。返回（avo 为空列表）strdis2band((stat[’enemy’][’x’],stat[’enemy’][’y’]),sto[’fieldcn’][’enemy’],[],1) 或 0（敌人在其领地内）

dis2band(host,targ): 传入 host 和 targ 均为’me’或’enemy’字符串，返回 host 的头到 targ 纸带的距离，后仅被用于求敌人的头到我方纸带的距离。与 mehome 类似，在有 flag 的 strdis2band 之后如果 flag==False 则 d0 与先回家再无阻碍地出击距离作比较，再进行建图和 dijkstra。

bfs_nearest_enemy()：返回距自己最近的对方（纸带或纸卷）的坐标元组。采用了广度优先搜索算法的思路。

go_out()：返回距自己最近的、与敌人距离在 10 以上的（防止一出门就被撞）、非自己领地的坐标元组。采用了广度优先搜索算法的思路。

find_route(point)：传入一个坐标元组 point，返回从 me 到 point 可走的方向所组成的列表（0, 1, 2, 3 分别表示东、南、西、北）若 me 与它正对，则返回的列表中，第一个方向为相向，第二、第三个方向为横向的。如 point 在 me 的正南方，则返回 [1, 0, 2]。把相向的方向放在第一个，这样在后面的逻辑中，如果能直接走过去就不会拐弯。若不是正对，则返回接近的两个方向。如 point 在 me 的西南方，则返回 [1, 2]

avoid_wall()：返回当前可以走的（不撞墙、不掉头）的方向集合。

2.3 程序限制

1. 直至最终比赛时，一方头部被包入敌方领地、纸带不再与自己领地相连的情况仍处理不善。此种情况下 findborder 由于搜寻行不正确，有可能失灵使得报错。如果我方头部被包入会立即报错（原因尚未明确）、敌方被包入则报错概率较低。

2. 由于 findborder 无法处理飞地情况，甚至可能将飞地错当做自己所在的领地，在有飞地时可能做出非最佳甚至危险的决策。（有部分解决方法但未得实施）

3.dijkstra 算法要求顶头点数目较少，但固定回家距离不变的方法使纸带拐点数增大导致复杂度暴增。在路线弯曲的情况下，通过拐点获得各种信息其实是一种低效的方法。

4. 己方所处领地极小时，可能找不到符合要求的出口（如下图），造成程序报错。

3 实验结果

3.1 测试数据

实验环境说明：

硬件配置：2.8 GHz Intel Core i5；8 GB 1600 MHz DDR3

操作系统：macOS High Sierra v10.13.4

Python 版本：3.6

在 visualize.py 文件中多次运行 play.py 和另一个对照 AI，观察比赛结果。经过多轮修改优化，大部分情况下能够正常运行并获得胜利，但仍有一些特殊情况下程序会报错，最终也没能全部解决。

3.2 结果分析

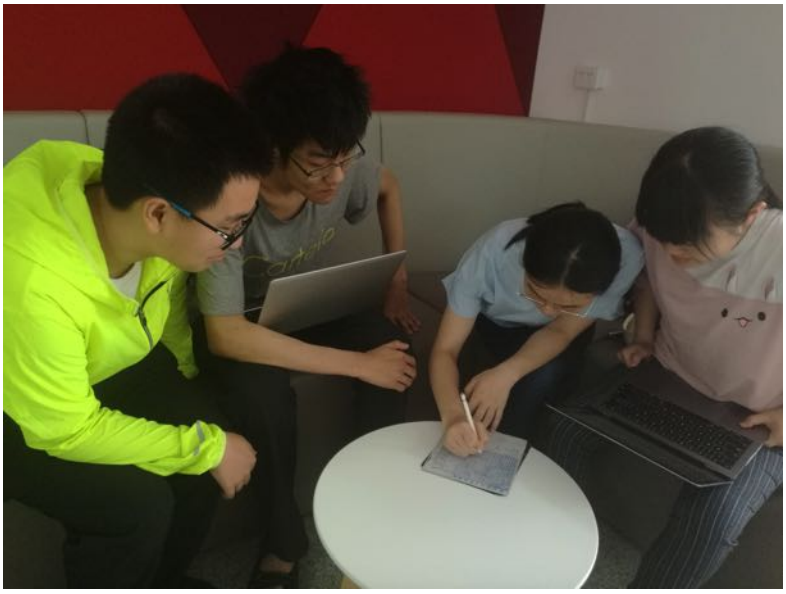
程序对危险的察觉灵敏，很难被敌方击杀；扩张速度和抓住击杀机会的能力均尚可；在一些特殊情况下程序会报错，说明 debug 工作还有漏洞。用图算法计算距离时间开销较大，但总体来讲在可控范围之内。

4 实习过程总结

4.1 分工与合作

我们把程序分成两大模块：自己在领地内和自己在领地外。富云齐、隋绍丹和蔡家骥负责外部，其中富云齐和隋绍丹负责计算距离函数以及返回领地函数的编写，蔡家骥负责圈地扩张函数；蔡紫菁和蔡紫葳负责内部，蔡紫菁负责攻击己方领地内敌人函数的编写，蔡紫葳负责找到最近出口出领地的函数。

除了经常性的线上讨论以外，我们进行了若干次线下组会讨论，主要包括汇报成果进展、探讨遇到的问题的解决方案、规划之后的任务等工作。



4.2 经验与教训

我们组成功之处主要在于进行了科学的分工，积极交流保证所有人前进的方向一致。

我们组失误之处在于工作开展较为缓慢，以至于最后并没有完成所有的 debug 工作，也没能参加热身赛，以至于程序的一些运行上的问题一直没能发现。（我们的程序在小组赛中报错 40+ 次，查看复盘记录发现是水土不服，在最终的代码中已经改正。）此外，各部分负责人员没有统一函数接口和变量命名，给整合工作产生了不便。在这些方面我们还有一定的提升空间。

4.3 建议与设想

我们对于本次实习作业在组队、基础设施代码、竞赛等方面均非常满意。希望对战平台更新等较为重要的信息和通知的发布不要只依赖于微信群，而能够在课程网站上发布信息，以避免重要信息与不重要的信息混杂，容易错过重要通知的情况。

对选修明年课程的学弟学妹，我们最想说的是，不要太在意输赢，一定要珍惜这次完成一个完整的编程项目的机会。这是一个宝贵的锻炼编程与合作能力的机会，而对于非本专业人士这种机会以后可能不多了。

5 致谢

感谢陈斌老师和所有助教的耐心指导，感谢技术组提供的极有帮助的资源，以及维护对战平台付出的辛苦努力。如果没有他们的支持和帮助，我们所有的想法都只能是空中楼阁。

6 参考文献

《数据结构与算法》课程讲义

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

http://interactivepython.org/runestone/static/pythonds/index.html

第六章　F17_Echo 报告

莫文韬 *、周江诚、常鹤翔、宋肖汉、魏秋实

摘要：我们组采用了强化学习和传统的基于规则的算法编写了纸袋圈地的 AI 程序, 前者主要利用了强化学习中的 Q-Learning 方法与深度网络结合的 DQN 进行训练与实际对战, 同时利用了完全二叉树实现了对数时间的在大量样本中按优先级随机取样的方算法, 后者主要利用的是基于人工寻找规则的方法进行游戏对战. 前者由于准备不够充分等问题并不能较后者更优, 基于规则或简单搜索 (如 $\alpha - \beta$ 剪枝) 的方法在这个问题上仍然具有较大优势.

关键字：强化学习,DQN, 完全二叉树, 随机取样, 基于规则的 AI,Q-Learning

1 算法思想

1.1 总体思路

经初步分析，纸带圈地这一比赛赛制使得这一游戏策略俱有很强的变化性，加上之前对于强化学习人工智能在棋牌类游戏中卓越表现的了解，虽然没有编写强化学习的经验，但我组仍决定将强化学习作为编写纸带圈地 AI 的重点。另一方面，也考虑到由于没有强化学习的编写经验，在主攻强化学习 AI 的同时我们也兼顾尝试从非学习类算法入手编写具有固定决策能力的 AI。对于非学习类纸带圈地 AI，我们的初步讨论是先使得其在“不撞墙”、“不自杀”的前提下围出最大面积（正方形），并进而加入回避攻击和进攻性函数, 以下讨论强化学习算法的思路.

强化学习算法

强化学习关注的是”环境”和”agent(智能体)”(这里即为算法所控制的纸带) 的互动. 如果这个互动具有 Markov 性, 即, 每一个状态的分布概率只与其上一个状态决定, 状态的变化构成了一个 Markov 链 (纸袋圈地的对弈显然符合这一性质: 每一步的最优策略必然只由上一步的状态决定), 则根据 Bellman 方程, 如果已知环境的各状态的准确条件分布概率, 则可以求得 agent 的”最优”策略 (通过值迭代和策略迭代). 然而最常见的情况是这些概率并不能事先获得, 所以在强化学习中人们引入了后来被策略价值估计 (本质上是 Monte-Carlo 方法在值迭代的应用) 的 Q 值 (Q 在被发明的时候并被没有特殊的意义), 它是一个二元函数:$Q : S \times A \to \Re$, 它接收两个参数, 当前状态 $s_i \in S$ 和采取的操作 $a_i \in A$, 得到的是该动作的”价值”Q, 它是一个实数, 其中 S 是状态空间，A_i 为在 i 种状态下所有可以使用的动作组成的空间. 如果得到了一个”正确”的价值估值函数 Q, 那么我们就可以根据 Q 值得到最优的策略.

在传统的机器学习中,Q 值学习有三种算法, 对应的是不同的对环境状态概率分布的近似”取样”方法, 它们都是迭代过程, 每一步迭代都会适当的”改善”Q 值函数, 使之更加正确, 当使用 Q-table 形式将 Q 值一一列出的情况下 (即对每个 (s,a) 对分别储存它们的 Q 值), 我们可以得到以下三种 Q 值的更新公式:

1. MCMC(Monte-Carlo Markov Chain):

$$q(s,a) = \frac{q(s,a) + n(s,a) + g}{n(s,a) + 1}; \qquad n(s,a) = n(s,a) + 1$$

其中 q 即为需要训练的 Q 值函数,n 为遇到 (s,a) 状态对的次数,g 值为奖励值, 需要通过向下模拟 N 步来实现:

$$g = \sum_{i=0}^{n} \gamma^i r_{t+i}$$

其中 r 为接下来某一个状态 (通过模拟游戏进行得到) 的奖励值, 对总奖励值 g 的贡献随着步数增加而衰减 γ, 是一个小于一的常数.

2. SARSA

MCMC 需要将接下来的 N 步的过程全部计算, 进而我们需要很大量的计算,SARSA 和加下来的 Q-Learning 都避免这一点, 而只需要取这一步或是下一步的计算即可:

$$q(s,a) = (1-\alpha)q(s,a) + \alpha(r + q(s',a'))$$

其中 s',a' 为下一步的状态与下一步的动作, 是学习率常数.

3.Q-Learning:

这一算法最早由 Watkins 在 1989 年推导出来, 它是一个模型无关 (model-off) 的学习算法 (和上述二者一样).

$$q(s,a) = (1-\alpha)q(s,a) + \alpha(r + argmax_{a'}q(s',a'))$$

可以证明, 在无限运行的情况下,Q-Learning 一定可以收敛到最优策略价值 -动作估计上 (Watkins,1989). 需要声明的是, 在训练的过程中 s 状态对应的动作一般由 $\epsilon - greedy$ 策略决定, 即以一个概率随机采取一个动作 (即随机的探索), 除此之外的情况会选择使得 q 最大的动作 (即价值最高的策略).

策略价值估计逼近与 DQN

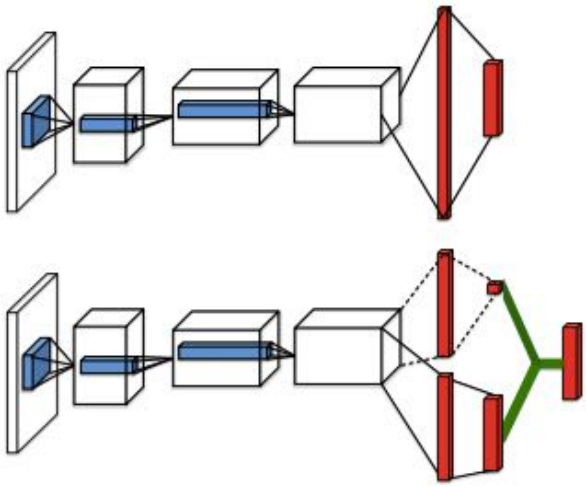
显然, 在状态空间过大的问题上, 比如纸袋圈地这个游戏上状态空间的大小在 20^{4700} 的量级上, 所以不可能使用一个表将它们完全记录下来, 这个状态数目远远超过了星星的数目, 也超过了宇宙中所有粒子的数量级. 对于这些状态空间超级大的问题, 算法科学家们也提出了解决办法: 策略价值逼近, 引入一个由数个参数控制的函数来拟合实际的策略价值 -状态 -动作关系. 可以抽象的记为 $q(s,a;w)$, 最简单的策略价值函数包括用一个线性函数或者是核函数的线性组合进行拟合, 在深度学习被广泛应用的现在, 卷积网络也用于拟合 q 函数, 这里我们可以用监督学习的方法来训练一卷积网络, 其期望输出值即为 QL 法中更新公式中的第二项:

$$q_{expected} = r + argmax_{a'}q(s',a')$$

在这一次的 project 中, 我们采用的是 Dueling Deep Q Net, 在普通 CNN 最后添加了一层 Dueling 层, 将价值函数分解为价值 (Value) 和优势 (Advantage):

$$q(s,a) = v(s) + a(s,a)$$

具体应用到网络结构上可以下图说明, 上面的网络结构是普通的 DQN, 下面的网络结构是 Dueling DQN.



我们采用的最终网络结构如下:

$$s, 3\ channels$$

$$\xrightarrow{Convolution+BatchNormlization,\ ReLU\ activation} s', 32\ channels$$

$$\xrightarrow{Convolution+BatchNormlization,\ ReLU\ activation} s'', 128\ channels$$

$$\xrightarrow{Convolution+BatchNormlization,\ ReLU\ activation} s''', 64 channels$$

$$\xrightarrow{2\ DenseConnectedLinearLayer} A(3\times 1\ vector), V(scalar)$$

$$\xrightarrow{LinearLayer} q(3\times 1\ vector)$$

网络的损失函数是 Huber-Loss, 是平滑的 L1 范数, 不含正则化项 (可能是以后的改进方向):

$$J(s,a) = d^2(if\, d \le 1)\ or\ |d|(otherwise); \quad d = q(s,a) - (r + argmax_{a'} q(s',a'))$$

网络的训练采用的是 Adam 梯度递降算法, 是包含了自适应学习率 (对每个不同参数) 调整的小批量梯度递降 (Mini-batch Gradient Descent) 方法, 更新公式如下:

$$\Delta w_i = -\eta_i(t) \times \frac{\partial J}{\partial w_i}$$

其中 η 是由离散时间步 (即迭代步数) 与梯度向量的一阶二阶矩确定的的学习率常数 (learning rate). 具体的训练算法在将会在算法流程图一节中描述.

Prioritized Experience Replay

我们在设计网络结构的同时, 由于网络性能不太理想, 从最一般的 DQN 转换到了 Dueling DQN, 也采用了 Prioritized Experience Replay(PER) 技术加快网络的收敛速度, 这是取样训练所需要的样本时优先提取优先级更高的样本的训练方法. 它赋予了每一个 transition(这里是 (s,a,s',r) 的状态 -动作 -下个状态 -奖励的四元对) 一个优先级, 这个有优先级与 TD-error 有关:

$$TD - error = |q - q_{expected}|$$

而具体的优先级计算可以使用以下公式:

$$pr = (\epsilon + TD - error)^a$$

其中 $\epsilon > 0$ 和 $a > 1$ 都是参数, 通过记录所有已经记录下来的 p 值, 可以根据这些 p 值采样出 n 个样本, 且服从概率分布:

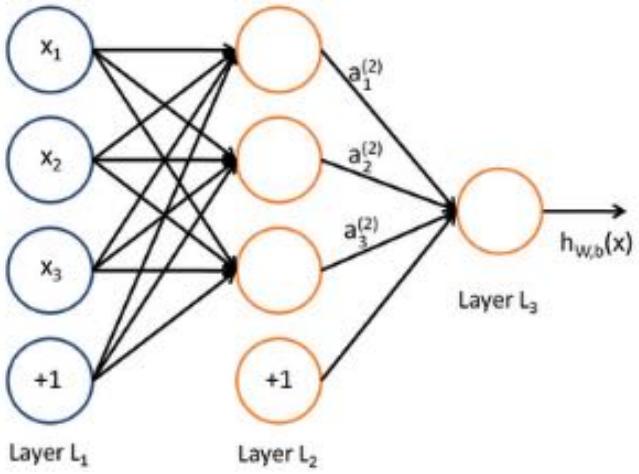
$$p_i = \frac{pr_i}{\sum_1^N pr_i}$$

如果采用普通的数组存储这些 transition 和它们的优先级, 采样的时间复杂度会变为, 这在需要极大量的采样的训练算法中是不合适的 (大约需要亿这一数量级的采样) 我们采用了 Sum Tree 这一数据结构对优先级进行存储: 这是一个完全二叉树, 采用数组线性储存, 每个节点 (如果它不是树叶节点) 的值是它的所有子节点的值的和. 这样我们可以在 O(logN) 的时间内完成采样, 大大加快了采样速度 (典型的 N 值在 5000-100000 之间).

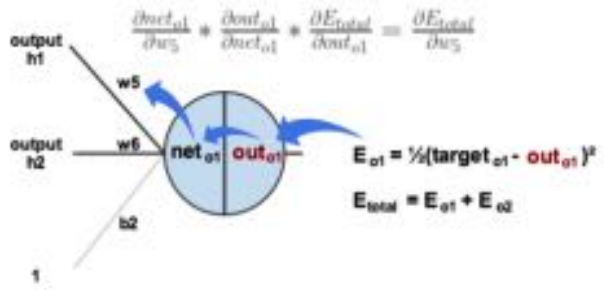
1.2 算法流程图

DQN

1. 训练网络 (back propagation):

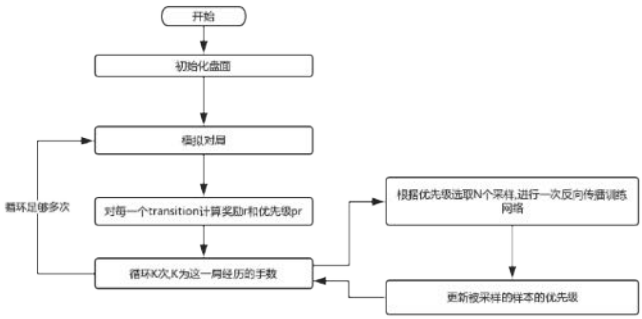


正向传播过程

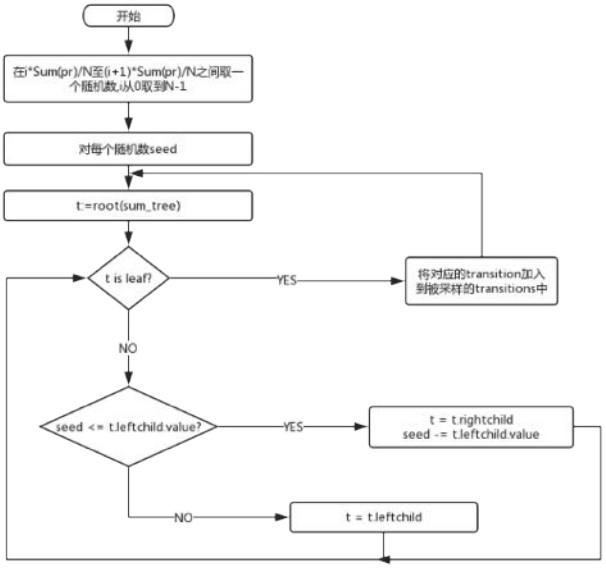


反向传播过程

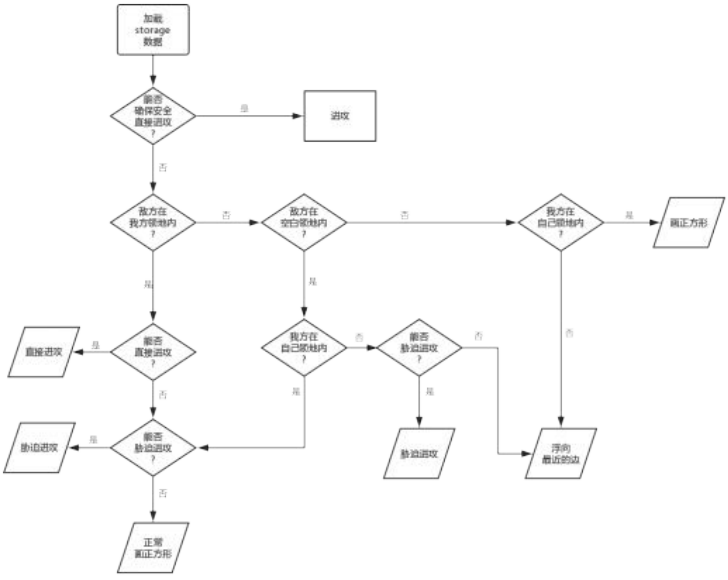
2. 进行对局模拟与具体训练:



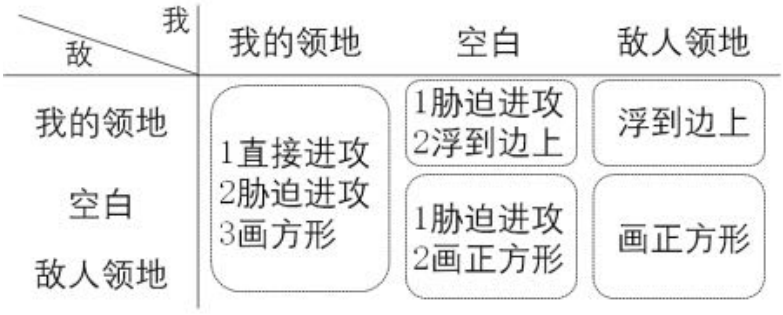
3. 选取样本:



传统算法



以上算法流程可简化为攻防决策表如下，其中数字表示同一情形下的决策优先顺序。（关于“胁迫进攻”概念请参考 2.1.7）



1.3 算法运行时间复杂度分析

传统算法

决策函数的公式计算时间都在 O(1)，因此算法运行时主要时间消耗在需遍历整个地图的辅助函数上。由于每次循环都要遍历一次地图，故总时间复杂度为 O(n2).

DQN

1. 模拟对局

每一手的前向传播时间是一个常数, 它和网络的参数总数成正比, 每一帧 (即每一手) 的时间复杂度是 O(W),W 为网络的参数总数, 但是由于使用 GPU 进行前/反向传播过程, 这一步时间复杂度的常数值很小, 使网络的计算与训练不再是最主要的时间消耗.

2. 训练

每一次的训练 (每一帧) 需要取得 N 个样本 (在程序中取的是 128 或者 256), 然后对 J 函数的均值平均后进行反向传播, 需要的时间也是 O(W) 量级的, 由于取均值的时间与网络训练相比可以忽略, 故不出现在最后的时间复杂度中.

3. 采样

每一帧需要采样 N 个样本, 每取得一个样本需要 O(logM) 次基本操作 (从一个完全二叉树的顶部访问到底部, 需要的时间与其深度成正比,M 为所有数据的个数) 故此处总时间复杂度为 O(logM).

4. 总时间复杂度

总时间复杂度是以上操作复杂度的叠加 (在不知道各操作常数之前不进行忽略), 每一帧的时间复杂度为 O(W+NlogM), 平均到每个采样的复杂度是 O(W/N+logM)

5. 经过实际测量, 实际训练中模拟对局与训练的时间取决于网络大小, 在网络规模比较小的时候模拟对局的速度是性能瓶颈, 一般是时间花费是后者的两倍, 而在网络规模较大和储存的 transitions 数量比较大的时候, 训练和采样速度从模拟速度的两倍变为不到 1/2, 称为性能瓶颈. 在网络规模较大和储存的 transitions 数量比较大的情况下, 模拟的速度约为 0.017s/frame, 训练的速度和采样的速度在 0.03s/frame 和 0.01s/frame 左右, 虽然网络的参数数量在 10^8 数量级, 但是 NVIDIA GTX 1080 的强大性能使得训练过程变得很快 (是 1050Ti 的两倍速度, 是 i5-5250U 单核的 60 倍速度). 总训练效率在 16frame/s 左右, 一小时约可以进行 60000 帧的训练. 约 17 个小时可以完成较为标准的一百万的帧训练.

2 程序代码说明

2.1 数据结构说明

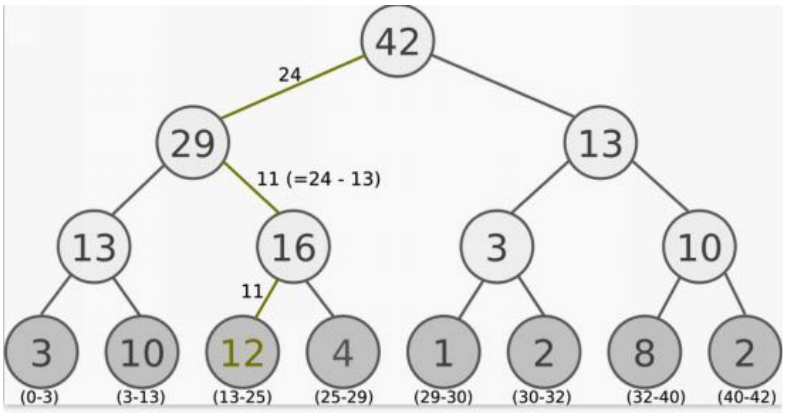
传统算法

1. 算法中采用的基本数据结构：
   (a) 线性数据结构
   (b) 树型数据结构
2. 所用到的类有：
   (a) math
   (b) Python 内置基本类（列表、元组等）
   (c) random

DQN

学习算法中使用的最主要的数据结构是多维数组。通过调用 pytorch 模块，我们可以建立一个 N 维数组 (称为张量,Tensor)，其在矩阵处理方面的性能相较于 Python 自身的嵌套列表要高，可以用于完成多用 C++，Matlab 等软件完成的工作。具体地，我们建立了一个三维数组——用于记录对战场地各点的状态、一个二维核（Kernel）——参与卷积神经网络的计算，处理上述三维数组。

另外，在实现 Prioritized Replay 以改进学习算法的 AI 的性能时，我们使用了 SumTree 来优化训练记录样本的抽取。我们给每一个样本赋一个值 p 作为抽样优先级，作为树的叶，树的其他节点对应的 p 值为两子节点 p 值的和。相应地，根节点对应所有叶节点 p 值的总和。将 [0，p 值总和] 的区间平均分为小区间，数量为样本数。在每个区间随机抽取一个数，从根节点开始，按照“若此数小于当前节点的左子节点，则向左下行，反之向右下行，并减去左子节点对应的数值”的方法在 SumTree 中选出一条由根到叶的路径，路径最终到达的叶节点即为抽样结果。这样，p 值较大的叶节点更容易被选到。具体实现上，我们定义了一个 SumTree 类，是一颗二叉树，其中包括 add 方法——用于插入新叶和节点、update 方法——用于在 add 新叶时更新各节点 p 值，get_leaf 方法——用于取样。实际处理中，要取的样本是训练时 AI 所走的各步（即下述 Transition），p 值与此步前一个状态的 state_action_value 和 expected_state_action_value 的差（参见 2.2 optimize_model() 函数）成正比，即训练记录中二者相差大的步优先被抽样、优化。



卷积神经网络通过 DDQN 类实现。DDQN 是 nn.Module 的继承类。DDQN 类的类属性是一些处理场地参数矩阵的函数，使用的是为 torch.nn 模块中提供的函数，包括同 Kernel 核计算，正则化、求和降维等。这些过程最终输出三个数，分别是纸带头向三个方向前进的行动评估值。DDQN 类中的 forward 函数用于串接这些过程，并利用 relu 函数在三次卷积之间添加三次非线性操作，以打破算法的线性性质。

Transition 类用于存储训练过程中每一步的当前状态、行动、下一个状态及奖励值，是训练过程每一步的记录。每一步的 Transition 都会被存储在训练的 memory 中，待下一步取样优化使用。

2.2 函数说明

传统算法

1. distance(x1, y1, x2, y2) → distance

   辅助函数，返回两点间路程。

   实现方式：公式计算

2. goto(x0, y0, direction, x, y) → direction

   辅助函数，返回从点 (x0, y0) 点走到点 (x, y) 的转向方法。

   若目标点在我前方，则先直走再转弯；反之，若该目标在我后方，则右转，意在先通过右转使目标点处于我的前方。

   实现方式：采用 if - else 分支结构，通过对不同情况的分类讨论得以实现

3. do_action(x, y, direction, action) → [x1, y1, direction]

   辅助函数，以 list 的形式返回进行某转向操作后新的坐标和方向，用于辅助复杂情况的判断。

   实现方式：采用 if - else 分支结构，通过对不同情况的分类讨论获得转向后位置和方向。

4. p_border(player1, player2) → [distance, x0, x0]

   辅助函数，返回一名玩家 player1 到一名玩家 player2 领地的墙的最小距离，和这一处于墙的点的坐标。

   遍历地图的所有点，计算每一个 player2 领地点距 player1 的距离，记录并返回其中最短的距离和该墙点的坐标。若有多个这样的点，只返回检索的第一个。

   此处 player1, player2 可以不同，也可以相同，用于判断是否逃跑（me 距 me 的领地距离大于约等于 enemy 距 me 的距离）或击杀（enemy 距 enemy 的领地距离大于约等于 me 距 enemy 的距离）。

5. p_fields(player1,player2) → [distance, x0, x0] 辅助函数，返回一名玩家 player1 到一名玩家 player2 领地的最小距离，和这一最近的点的坐标。

   同样采用遍历地图取极值的方法。与上一个函数不同的是，这里扩展了函数的接口，使得对于 player1 对于玩家对象（‘me’，‘enemy’）和坐标对象兼容。

6. p_bands(player1, player2) → [distance, x0, x0] 辅助函数，返回一名玩家 player1 到一名玩家 player2 纸带的最小距离，和这一纸带上的点的坐标。

   同样采用遍历地图取极值的方法，时间复杂度 O(n^2)

7. aggressive_turn(x,y,direction,)

   根据自己是否有末步行动者 [1] 优势决定转弯。如果判定为自己有末步优势 [1]，则转向靠近敌人的方向尽力击杀；反之转向远离敌人的方向尽力圈地。

   [定义] 缠斗：双方都在做不以圈地为目的的行动的一种状态，包括试图击杀、拦截对方，逃跑等。

   [定义] 末步行动者：当双方非常接近，离自家领地非常遥远，且纸带不在由两人位位置所确定的矩形内时，击杀不可避免。而击杀时，假设是双方相撞（这种假设是基于对方采取理智决策，即最低死亡可能性的决策而做出的推论），走出最后一步的一方具极大主动权，称之为“末步行动者”。末步行动者具有极大优势，在缠斗 [2] 中大多数况下能杀死对方，只有少数几种情况例外（这几种情况下，对方可以胁迫末步行动者以正碰结局）。而末步行动者的身份，是在开局时即决定的。

末步行动者	两人距离为奇	两人距离为偶
我方先手	我	敌
我方后手	敌	我

   当我方先手距离为奇、或我方后手距离为偶时，我方是末步行动者，处于优势，宜采取进攻型战略；反之当我方先手距离为偶、我方后手距离为奇时，宜采取保守战略。

   当我方为末步行动者，目双发即将发生击杀时，只要情况能够递归到’ 田’ 字，我方必胜。这一点大多情况可以满足，唯一例外是双方处于一条直线上，且对手永远沿着这条直线逼近末步行动者，这样末步行动者会被胁迫与对手正碰。

   [定义] 胁迫进攻：末步行动者在对方无法逃回，只能迎战的情况下，接近对手，试图利用自己的末步优势击杀对手。

   胁迫进攻的发动需要满足三个条件：

   1. 己方具有末步优势

   2. 双方距离自己的领地较远，面对进攻无法回避

   3. 防御方与进攻方不处于同一条水平线或垂直线上

8. in_judge(player1, player2) → True/False

   辅助函数，判断 player1 是否在 player2 的区域里。同样 player1, player2 可以不同也可以相同。

   用于构建攻防决策表，在不同状态下采取含不同功防属性的决策。

9. being_surrounded_judge() → float()

   警戒函数，判断领地及纸带对自己的包围程度，以免被对手合围。但由于我们的水平和时间有限，实际上此函数并未被调用。

10. side_judge(x, y, direction)

    警戒函数，判断前方是否是墙，防止撞墙

11. touch_me_judge(x, y, direction)

    警戒函数，判断自己是否会在领地外撞上自己的纸带。

12. die_judge(x, y, direction, act)

    警戒函数. 判断若是做了 act 动作是否会撞到自己或撞到墙

13. judge_again_change(x, y, direction)

    决策函数，若发现自己回到领地之前还会再次撞到自己纸带或墙，转向相反方向。

14. gostraight_judge(x, y, direction)

    警戒函数，判断直行到达边界或回到己方前是否会撞到自己。

15. normal(x, y, direction)

    决策函数，在一般状况下的行动，包含圈地、躲避和逃跑。

    以双方距离的 1/3 为边长，从边界出发画正方形。双方距离越远，正方形的边长越大。在以下几种情况下，结束画正方形的过程，且转向使回逃路径最短的方向：

    - 撞墙
    - 撞自己的纸带
    - 我方回逃距离大于约等于敌方进攻距离

16. aggressive(x0, y0, x1, y1, direction)

    决策函数，当我方有末步优势（参考 7 中定义），且对手无法回逃时的进攻

    play 函数已经保证了绝对击杀且我方绝对安全的时刻会进攻，因此本函数只需缩短两人距离，同时使两人接近正方形对角线，尽量使双方向’田’字状态靠拢，然后由 play 函数下的机制击杀对手。

DQN

下面对学习算法中主要用于实现学习与训练的几个函数作简单说明。

simple_reward() 函数 () 是平凡状态下的奖励函数，设置不出现相互攻击等情况时的奖励值。奖励值由每一步时 AI 所围面积与伸出的纸带长度决定。在最终版代码中，我们选择圈地面积平方的一半作为奖励值。更多细节情况下——例如击杀、撞墙、正碰等——的奖励方法在主要运行过程的代码中给出。

e_select_action() 函数设置了 AI 的训练的“保守”程度。这里的“保守”指的是训练时 AI 是否按照已获得参数计算出的较优解采取行动。我们设置了一个单调下降的 eps_threshold 参数，每次进行选择时，用一个随机数与其比较，若随机数大于 eps_threshold，AI 选择较优解，若随机数小于 eps_threshold，AI 随机采取行动。这样使初期的训练“保守”程度较低，末期“保守”程度较高。

optimize_model() 函数用于对卷积神经网络中参与卷积的 Kernel 进行优化。optimize_model() 会分别算出两种行动评估值函数。一方面，它利用上述卷积神经网络，按照目前已获得的参数，计算出 AI 在当前状态下的行动评估值（state_action_value）。然后又利用我们设置的各步的评估值之间的递推式，将此步后 AI 所处下一个状态的最优行动评估值向前传递，推出当前状态下的评估值（expected_state_action_values）。这两个评估值函数之间有差距，我们用梯度递降法寻找两者差的最小值，并通过循环递降将 Kernel 中各参数调至其差函数最小值点所对应的值。

2.3 程序限制

传统算法

1. 在判断胁迫进攻时，若双方恰好处于同一水平线或垂直线上，会与对手发生正碰，不能通过侧碰击杀对手

2. 无法预判由于对手的侵占使原本计划回逃的领地成为对方所有，这导致了回逃距离的增长，可能长过对手的击杀距离，从而会被对手拦截。

DQN

由于学习算法的 AI 只能通过训练的“经验”来获得强化，不能依靠人们的事先设定的策略，AI 的成长不得不从零开始。这样就使得 AI 的训练像是盲人走路，参数的收敛需要很漫长的过程和无数次的反复训练，尤其是相互进攻和防守的细节处。另外，因为没有策略保证，学习算法的 AI 在训练不充分的情况下，容易出现撞墙、自杀的状况，而充分的训练又不易完成，学习算法在这个游戏中就难以以有限准备时间达到较高实战水平。

3 实验结果

3.1 测试数据

实验环境说明：

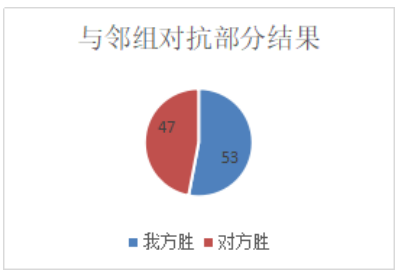
硬件配置: Mac Book Air: CPU: Intel Core i5-5250U(1.6GHz), Memory: 4GB, Server: CPU: Xeon W Series(2.8GHz), GPU: GTX 1080 11GB, Memory: 32GB

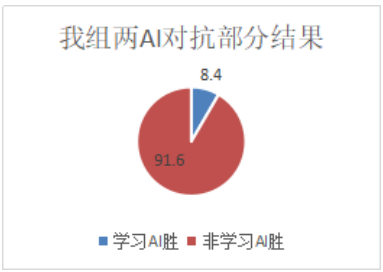
操作系统: macOS High Sierra/Ubuntu 16.12/Windows 10.1

Python 版本: Python 3.5/3.6

测试方法：相同 AI 互搏、与邻组 AI 对抗、学习与非学习 AI 对抗、新旧版本 AI 对抗、热身赛。

测试结果：与邻组 AI 对抗中我组胜负参半，多为圈地胜出。学习与非学习 AI 对抗时，大部分为非学习 AI 胜出。新版本 AI 一般强于旧版本 AI。热身赛最好成绩是第八名





3.2 结果分析

实际上, 使用了 ML 的 AI 并没有体现出 DQN 的强大威力, 常常陷入自杀和在自己的领地里四处绕圈的结果, 这有可能是因为在原点附近的位置网络被过度地训练了, 而在较偏远的区域网络几乎没有被训练到, 这导致了网络训练效率低下, 除此之外, 由于数个 bug, 网络规模和网络结构的调整和 PER 的加入, 我们浪费了太多时间, 以至于没有很长的训练时间为每一种略有差别网络跑完整个训练过程 (15-20 个小时), 而且网络很有可能陷入了平坦的鞍点附近, 可以通过训练过程的 reward-time 曲线看出来这个趋势.

实战博弈中，我组主要使用非学习算法 AI，此处针对非学习算法 AI 的实战测试结果作出分析。

我组算法有如下优点：

第一，我组非学习算法 AI 圈正方形的圈地策略能够节省步数，提高圈地效率。在与邻组对抗测试时，我组的圈地速度要略快于邻组，使得我组可以凭借此与其平分秋色。

第二，我组算法防守型很强。在圈地过程中若遇到可能被进攻的态势便果断回防，在不是保证击杀的情况下不会贸然出击，这使得测试中击杀的情况不占主要。

第三，若存在双方均无法返回圈地内部的情况（返回会被击杀），我组的 AI 能够保证半数的对决必胜。

第四，我组算法不存在超时现象。

然而，缺点也很明显：

第一，我组 AI 对进攻倾向差，不能与对方进行激烈博弈，不能对对手进行高强度压制。我组后来分析了两 AI 回旋时的情况，补充了进攻的算法，但是仍无法达到高强度与高集中度的进攻与回旋。

第二，我组 AI 防守时会破坏正方形圈地的策略，一旦对手对自身有威胁，我方的圈地效率大幅下降。若碰到压制性较强的对手，我组的圈地面积就会较小，且较不规则。这一点我们最后并没有很好改善。

第三，在回防时，可能会出现向着自己的纸带转向最终绕入“死胡同”以至于自杀的情况。在后来的测试中，这一缺陷被很好地改善了。

第四，我组算法细节处理不够周全，以至于在遭遇具有十分完备策略的对手时容易被

对方抓住漏洞击杀。

4 实习过程总结

4.1 分工与合作

第一次组会：

时间：2018/5/30。

地点：地学楼 311


第一次组会主要简要讨论了纸带圈地 AI 可能的非学习制胜策略可以突破的方向，也指出了一些算法的优点与不足。之后组会讨论将重点放在了强化学习算法的原理和实现步骤上，主要由对计算机编程已有一定功底的组长莫文韬同学向组员介绍强化学习算法，包括 QLearning、QFunction、Replay Memory 等。之后文韬同学继续向组员演示 PyTorch 的使用，强化学习算法所需的矩阵、梯度、自动求导等如何在 PyTorch 上实现等。最后文韬同学向组员普及了强化学习中非常重要的求卷积算法。

本次组会亦确立了组员间的分工情况：

组长莫文韬：机器学习 AI 的算法设计和代码编写。

周江诚：非机器学习 AI 的算法初步设计与编写。

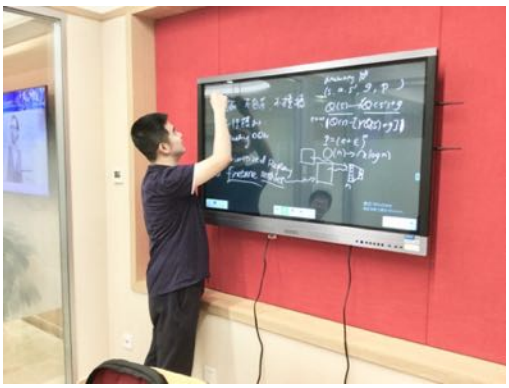
魏秋实、宋肖汉、常鹤翔：主要学习强化学习算法并协助莫文韬测试改进强化学习 AI，后期报告的整理。

：在周江诚的基础上改进算法、测试与完善非学习 AI，后期报告的整理。

第二次组会：

时间：2018/6/8

地点：地学楼 309



经过一个星期着重强化学习的尝试，我们的强化学期 AI 虽已具备一定的圈地和击杀能力，但是效果并不理想，而且技术提升速度较慢。于此我们开了第二次组会。组会主要介绍了周江诚同学非学习算法 AI 的初步设计和现有功能，并指出最终用非学习算法出战的可能性在升高这一现状。同时，我们也并没有放弃机器学习算法，并希望能够通过机器学习的算法改进提高它的能力。在对非学习 AI 的讨论中，我们指出现有的非学习 AI 最大的缺陷在于圈地方法单一，回避敌方击杀能力欠缺，几乎没有击杀对手的能力。对此接下来的重心应该放在如何提高 AI 的生存力和攻击性上面。同时文韬同学也利用剩下的时间介绍了提高机器学习 AI 性能的可行方向，并希望在剩下几天的时间里改进机器学习 AI 增强其实战能力。

此后组内分工也进行了调整：

组长莫文韬、组员周江诚的分工依然没有改变，魏秋实仍继续协助莫文韬改进机器学习 AI 算法。

宋肖汉、常鹤翔转而与周江诚合作编写非学习类算法的程序，主攻回避算法的完善和攻击算法的加强。

4.2 经验与教训

非常遗憾的是，我们组虽然从计划的初步建立到第二次组会前都将主要精力放在了机器学习算法的尝试中，但是机器学习算法的 AI 最终没能参加最终比赛。虽然所幸的是第二次组会过后，通过改进，机器学习算法的实战能力得到了显著提升，但是终究由于不能击败非学习算法，最后我组决定用非学习算法参赛。非学习算法的设计由于本身的时间不足以及很多考虑欠缺的因素，在比赛中也未能获得小组出线。

对此我们有如下反思：

1. 这次对于所有组员来说都是编写强化 AI 的第一次尝试，经验不足，在计划确立之初将大量精力放在强化学习的设计与编写上具有一定风险。同时在训练与改进强化学习算法的过程中我组也忽视了将非学习算法与学习算法进行对战比较性能的步骤，导致长时间未能发现机器学习算法的自我进展缓慢的问题，导致直到比赛前才将重心放在非学习算法上。

2. 非学习算法的初步构架由周江诚一人完成，缺乏初步的利弊讨论与方向的确定。后期的算法编写中大多是在原有函数的基础上构思回避函数与攻击函数，使得算法本身初期架构上的缺陷、漏洞等不能得到有效的修复与完善，编写函数时也缺乏对其它小组可能出现算法的分析，没有针对性的策略。圈地效率不能提高又难以通过有效的方法击杀对手，导致最终本算法没能出线。

4.3 建议与设想

在代码的改进上，我们认为如果是建立在两个星期内需要完成一个具有实战能力的 AI 的话，应该着重设计并改进非学习类算法，并从数学角度严谨地分析制胜策略，以期获得一些制胜规律与判断公式，将得到的算法由编程实现以提升实战能力。但是若不考虑有限时间，我们仍希望能坚持最初用机器学习编写 AI 的设想，通过改进机器学习的算法，尝试调参等提高机器学习类算法的实战性能。毕竟机器学习的能力虽然在短时间类难以体现，但其拥有超越人类有限经验分析能力的潜力，在长期的训练后有望自我完善和自我加强到更高水平。同时，机器学习也是现代人工智能的重要算法，其自身也在不断发展与完善中。运用机器学习编写纸带圈地这样的竞技类游戏 AI，也是当代大学生在学习中扩展知识、接触计算机前沿发展的一种很好的途径。

5 致谢

感谢陈斌老师提供的图形工作站和显卡.

感谢技术组的菊苣无偿劳动, 为我们写好对战的框架.

感谢小组成员的无私奉献与辛勤代码工作.

感谢所有为强化学习铺平道路的算法科学家与数学家们, 为我们提供这么多强有力的工具.

感谢开发了 CUDA 和 PyTorch 的开发者们.

6 参考文献

[1]. Simon Haykins, Neural Networks and Learning Machines 3ed.

[2]. Max Japan, Speeding up DQN in PyTorch: https://medium.com/mlreview/speeding-up-dqn-on-pytorch-solving-pong-in-30-minutes-81a1bd2dff55

[3]. PyTorch Documents. https://pytorch.org/docs/

[4]. Prioritized Replay Tutorial:

https://morvanzhou.github.io/tutorials/machine-learning/reinforcement-learning/4-6-prioritized-replay/

[5]. M. Hessel et al., Rainbow: Combing Improvements in Deep Reinforcement Learning, arXiv:1710.02298: https://arxiv.org/abs/1710.02298 (under review for AAAI 2018)

第七章　F17_Foxtrot 报告

张懿卓 * 王斌昊 章文博 刘威 谭术超

摘要：在本次实习作业——纸带圈地游戏的比赛中，本小组（foxtrot）获得亚军。本文主要介绍实习作业所用算法的原理及开发过程的经验教训。决定 AI 的策略以攻击为主，经过几次讨论，逐渐减弱攻击性、增强防御能力，平衡攻防性能，完善代码。在代码中主要使用了队列、栈等结构和列表等 python 的内置基础结构。在多次实验中，代码取得较好的效果，多次处于出线前列，同时也存在一些问题，比如由于考虑不周，代码的圈地效率并不高，导致在与策略有相似之处的组对战时常会吃亏。

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

我们开发的 AI 为一个基本纯进攻性 AI，所采用的开发方式主要为人工进化，即初始版本为一个只会进攻防守极差的 AI，之后根据各种死法进行归类，逐个解决合并，最终使得 AI 的防守性能有较大提高。

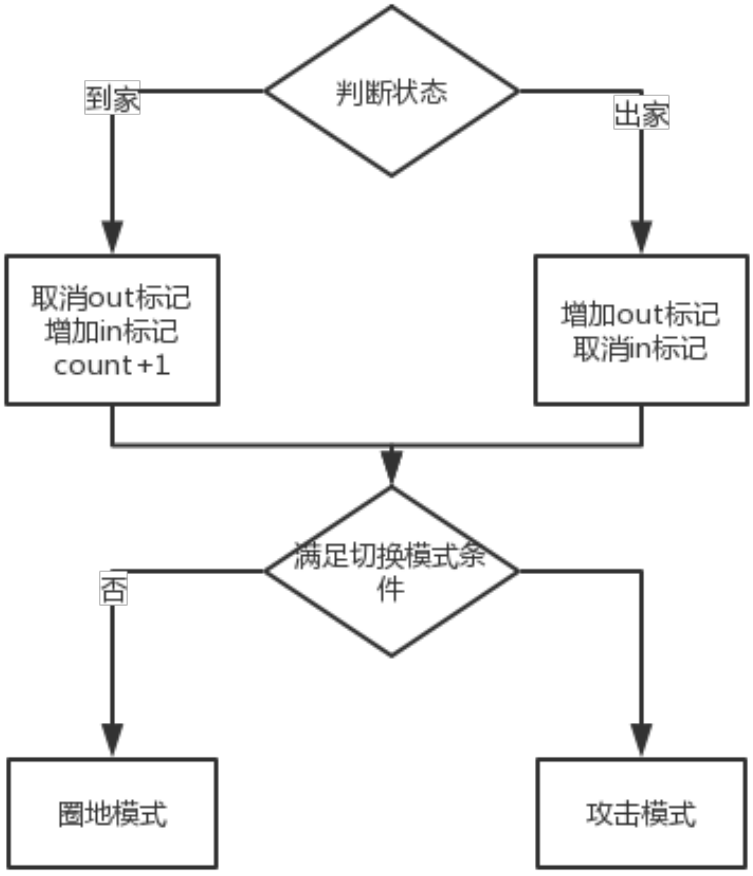
所采取的算法主要策略是一个预测机制：定义多个动作，每次对所有动作进行评估，得到最优的动作并执行。

在决赛时添加了一个圈地模块。算法的核心一个是圈地方式，一个是走出领地的方式。

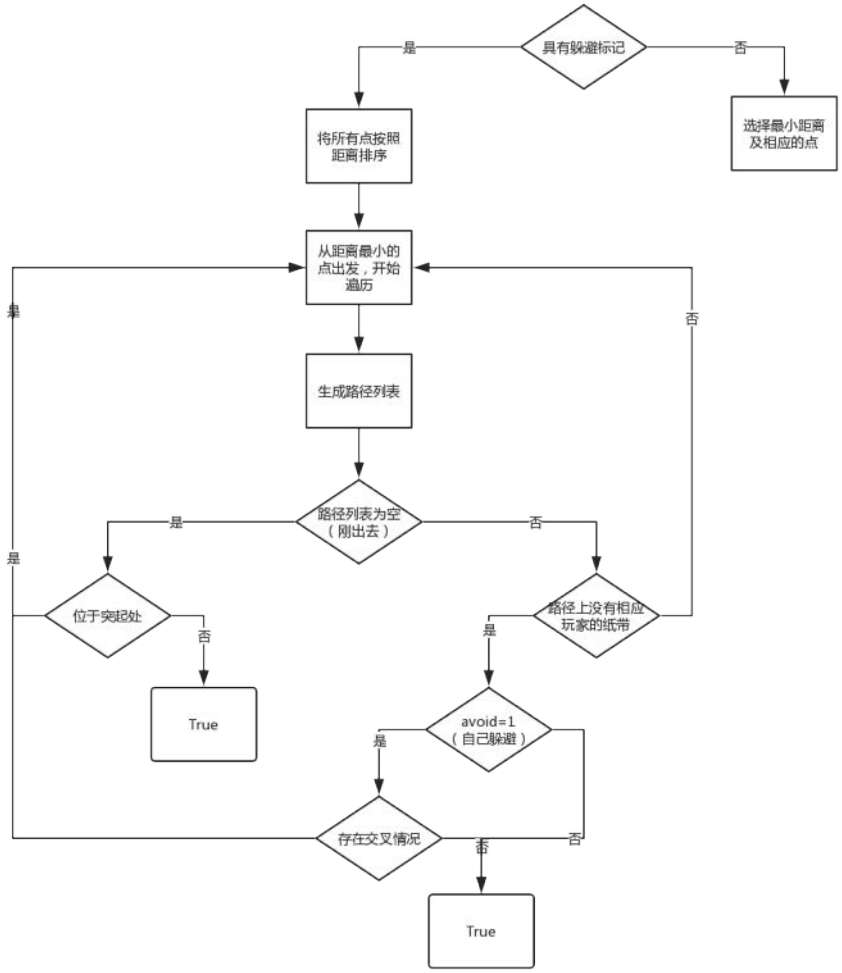
圈地方式：以规则正方形方式进行圈地，圈地动作由一个队列 spacePlan 存储，如果行进过程中没有受到敌方威胁则一直执行当前 spacePlan，否则优先进行躲避。

走出领地：基本使用与攻击模式下相同的方式走出领地，即尽可能以最快速度走出己方领地。具体地，如果当前处于己方领地边界且现在走出去没有危险，就最快走出去；如果当前处于己方领地但走出去是危险的，就在领地边界徘徊；如果当前处于领地内部，则向离自己最近的领地边界走去。

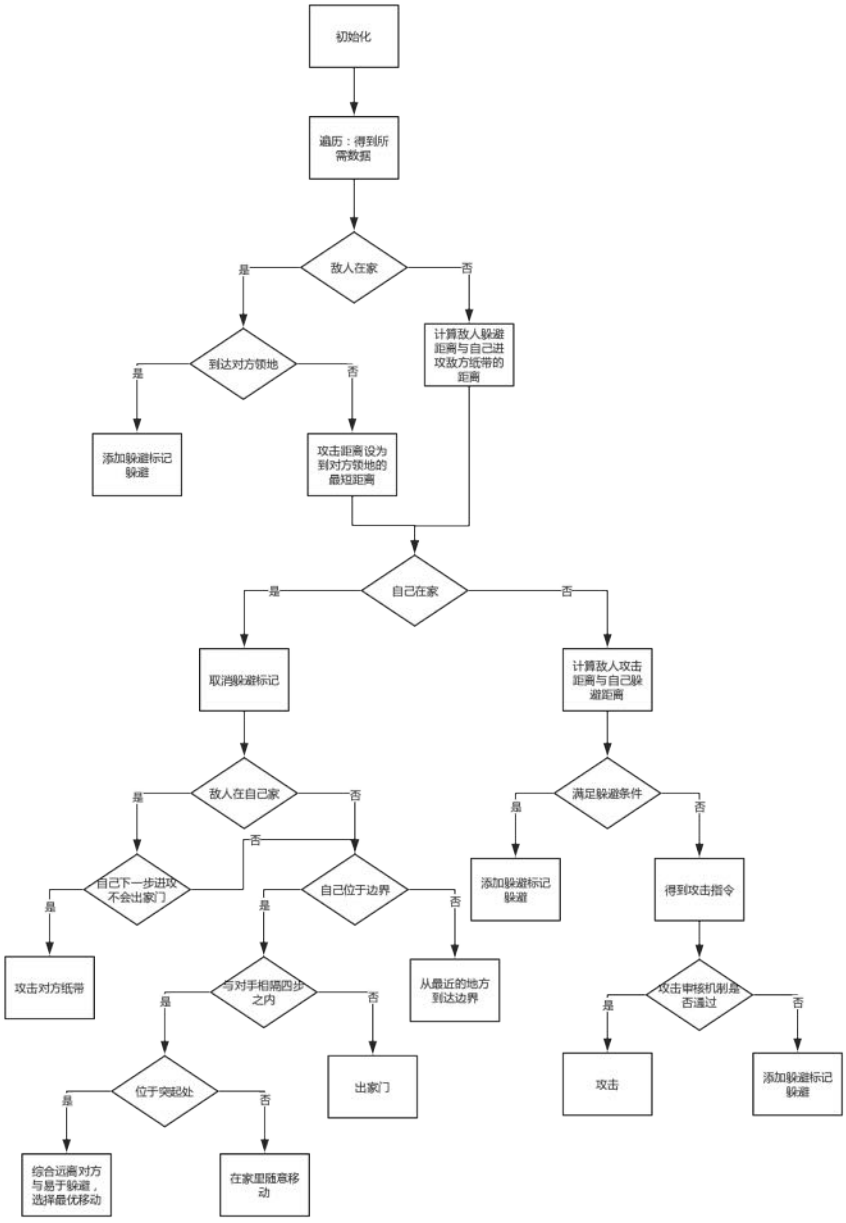
1.2 算法流程图



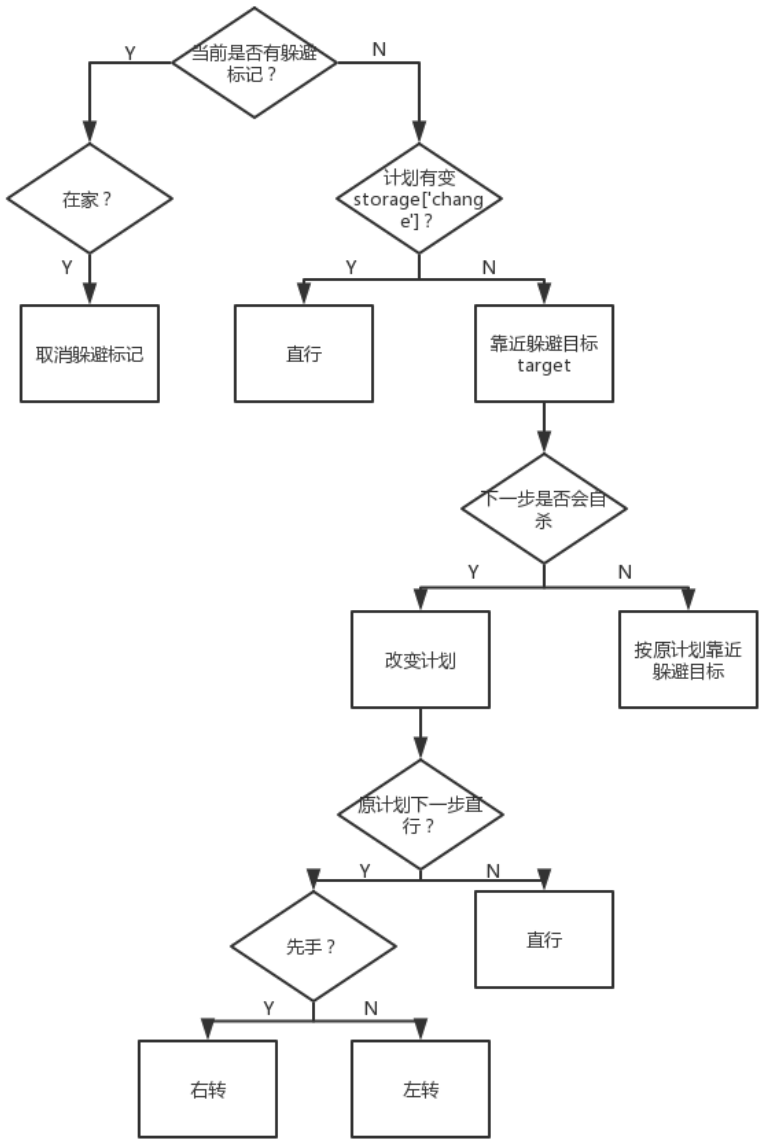
主函数


具有躲避标记
是
否
将所有点按照距离排序
选择最小距离及相应的点
从距离最小的点出发，开始遍历
是
否
生成路径列表
路径列表为空（刚出去）
是
否
是
位于突起处
否
True
路径上没有相应玩家的纸带
是
否
avoid=1（自己躲避）
是
存在交叉情况
否
否
True

计算最小距离


初始化
遍历：得到所需数据
敌人在家
是
否
到达对方领地
是
否
添加躲避标记躲避
攻击距离设为到对方领地的最短距离
计算敌人躲避距离与自己进攻敌方纸带的距离
自己在家
是
否
取消躲避标记
计算敌人攻击距离与自己躲避距离
敌人在自己家
是
否
满足躲避条件
是
否
自己下一步进攻不会出家门
是
否
自己位于边界
是
否
添加躲避标记躲避
得到攻击指令
攻击对方纸带
与对手相隔四步之内
是
否
从最近的地方到达边界
攻击申核机制是否通过
是
否
位于突起处
是
否
出家门
攻击
添加躲避标记躲避
综合远离对方与易于躲避，选择最优移动
在家里随意移动

攻击函数主逻辑



圈地函数

1.3 算法复杂度

遍历所耗时间较多，与比赛面积有关。其他耗时则依赖于具体状态。

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有列表、类。

1. 定义了 gesture 类，用来估计各种动作的最小距离及相应点等，如 me_avoid、enemy_avoid、me_attack、enemy_attack、willavoid 等。
2. 设计了一套相对方位体系，用来消除代码重复的情况，减少代码量。对方相对自己：正东‘0’、东南‘01’、正南‘1’、西南‘12’、正西‘2’、西北‘23’、正北‘3’、东北‘30’。利用这套相对方位体系，在代码中各种情况都可以用一个特定的公式概括，从而大大缩减代码量。
3. 定义 turnList=[‘s’，‘r’，‘l’]，它们的索引值为 0，1，-1，正好是转向公式所对应的值，因此可以合并许多代码重复的情况。
4. 加入了圈地模式的扩展 extend。

2.2 函数说明

2.2.1 攻击部分

```
1.# 基础函数包------------------------------
2.
3.distance(a, b)
4.#计算距离
5.#参数：两个点，列表形式 [x, y]
6.#返回值：距离， int
7.
8.nextPos(point, direction)
9.#下一步位置
10.#参数：当前所处位置，方向 pointdirection
11.#返回值：下一步位置， [int, int]
12.
13.willSuicide(point, direction)
14.#判断是否会自杀（撞纸带，撞墙）
15.#参数：当前所处位置，方向 pointdirection
16.#返回值：布尔值，表示会自杀，表示不会自杀 TrueFalse
```

```
17.
18.orient(target, pos)
19.#判断对方相对自己的方位
20.#参数：对方位置，自己位置 targetpos
21.#返回值：一套相对方位体系表示方法， str
22.
23.close(target, pos, direction)
24.#接近目标位置
25.#参数：对方位置，自己位置，方向 targetposdirection
26.#返回值：下一步动作的索引值， int
27.
28.isBorder(x, y, num)
29.#判断是否为相应玩家的边界
30.#参数：，坐标，相应玩家的值 xyid
31.#返回值：布尔值，为边界，不是边界 TrueFalse
32.
33.traverse()
34.#遍历
35.#得到对方纸带、对方区域边界，己方纸带，己方区域边界
36.
37.# 动作类-----------------------------
38.
39.#生成各种动作的数据，便于之后预测判断
40.class gesture():
41.    path(self, target, pos, direction, Id, avoid, attack)
42.    #路径判断
43.    #参数：目标位置，自己位置，方向，相应玩家号 targetposdirectionidId
44.    #参数：躲避标记，攻击距离（只有计算躲避距离时会用到） avoidattack
45.    #返回值：布尔值，表示该路径符合要求 True
46.    #要求：路径内部没有相应玩家的纸带（即回家时不会出现死胡同）
47.        _traverse(mode)
48.        #遍历函数
49.        #参数：遍历模式，表示先直走，再转弯；表示先转弯，再直走 mode01
50.        #返回值：布尔值，表示该种模式下的路径符合要求 True
51.            makePath(k, pointList)
52.            #制定路径函数
53.            #参数：遍历模式，点列表： kpathlist[pos, target]
54.            #返回值：无，但会改变 self.pathList
55.
56.    minDistance(self, point, lst, direction, Id, avoid=False, attack=None)
57.    #最小距离
58.    #必选参数：自己位置，目标列表，方向，相应号 pointlstdirectionidId
59.    #默认参数：躲避标记，攻击距离 avoidattack
```

2.2.2 圈地部分

```
1.getPosition(id)
2.#得到相应玩家的位置坐标
3.#参数：玩家 id
4.#返回值：玩家坐标 tuple (x,y)
5.
6.meIsDangerous()
7.#判断自己当前是否处于危险状态
8.#返回值： bool True or False
9.
10.strToInt(str_dirc)
11.#将相对方向（字符串型的方向 Left, Forward,）转为绝对方向（数字 Right0,1,2,）3
12.#参数：相对方向
13.#返回值：绝对方向
14.
15.goForward(planname, step)
16.#将前进步的计划动作加入对应的队列中 stepplan
17.#参数：计划名，步数
18.#无返回值
19.
20.turnLeft(planname)
21.#将左转的计划动作加入对应队列 plan
22.#参数：计划名
23.#无返回值
24.
25.turnRight(planname)
26.#将右转的计划动作加入对应队列 plan
27.#参数：计划名
28.#无返回值
29.
30.drawLeftSquare(a)
31.#方形逆时针走路圈地
32.#参数：为正方形边长 a
33.#无返回值
34.
35.drawRightSquare(a)
36.#方形顺时针走路圈地
37.#参数：为正方形边长 a
38.#无返回值
39.
40.goOut()
41.#走出领地的函数
42.#返回值：下一步转向 str 'Forward', 'Left', 'Right'
```

```
43.
44.AI_logic()
45.#主函数逻辑
46.#返回值：下一步转向 str 'Forward', 'Left', 'Right'
```

圈地代码分析说明

获取基本信息（如双方位置、双方领地边界、双方纸带列表等）基本采用与攻击模式相同的方式获取，在此不再赘述。在此主要说明 ADT Queue 在本算法中的应用以及一些值得说明的具体操作的原因。

ADT Queue

```
1.class Queue:
2.    def __init__(self):
3.        self.items = []
4.
5.    def isEmpty(self):
6.        return self.items == []
7.
8.    def enqueue(self, item):
9.        self.items.insert(0, item)
10.
11.    def dequeue(self):
12.        return self.items.pop()
13.
14.    def size(self):
15.        return len(self.items)
16.
17.    def clear(self):  #清空
18.        while not self.isEmpty():
19.            self.dequeue()
20.
21.    def pop(self):  #弹出至以转弯开头
22.        next = self.dequeue()
23.        while next == 'Forward':
24.            next = self.dequeue()
25.        self.items.append(next)
```

2.3 程序限制

2.3.1 对于主函数（除圈地函数）

基本不会出现报错 bug; 在一种极特殊的情况下可能会出现来不及躲避导致的死亡：对方在家，且对方的攻击路径与自己的躲避路径重合，由于为了保证具有一定的攻击性，我们在这里的估计值在极少数情况下可能不准确。另外，目前我们的 AI 最主要的死因是与一些碰巧采用了与我们相近的边界处理算法的 AI（f17 组的 November 和 Lima）对战时，会在一开局就陷入死循环，两者在边界处不断移动直到比赛结束（如 f17 组的决赛第一局），由于我们的算法最初定位就是纯攻击型 AI，因此没有设计圈地算法，所以在比赛结束比较地盘大小时我们常常会输。针对这种情况，我们在比赛前加紧设计了圈地接口和一些圈地函数，但由于时间太过紧张，这一部分处理的并不是太好。

2.3.2 对于圈地函数

设计圈地函数主要是在正式比赛的前一天几个小组自发组织的友谊赛中发现了之前纯进攻的漏洞：如果对方采取的出领地 goOut 函数与自己的判定条件相似，很容易陷入僵局耗到结束，如果前期没有一块看得过去的地盘的话结束时必定会输给对方。因此我们在一天之内赶出来了一个圈地模式 extend，期望在进攻前有一块地以在结算时与对方抗衡。圈地效果如下图所示 (先手)。

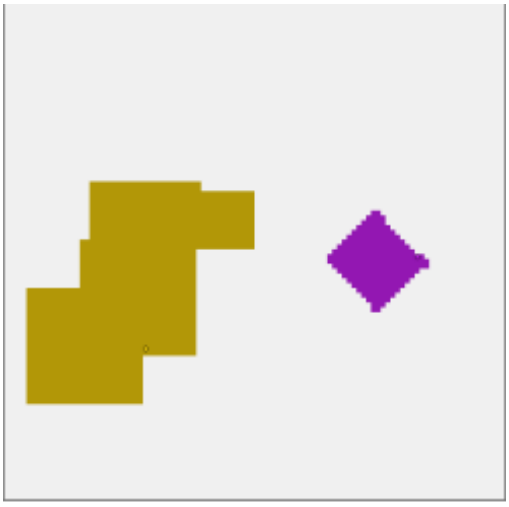
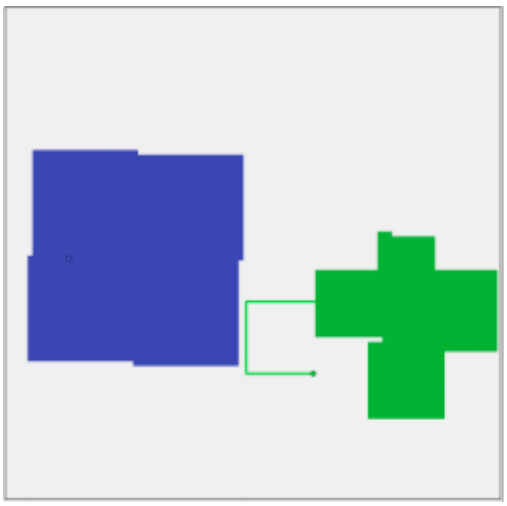


图 1 正方形边长 a = 20, limit = 4



图二 a = 10, limit = 10

圈地函数分析：想法比较简单，圈地效率其实也还行，调大正方形边长之后也能在短时间内圈得较大的领地。但是缺点也很显然：不能根据当前局势动态的规划圈地模式，很可能丧失圈地良机，从而输给更智能的圈地模式。而且圈地有冗余的步数，这也是跟不能动态根据当前情况圈地导致的。如果时间充裕，我们应该向其他优秀的小组学习，开发一个动态的圈地方式。

2.3.2 一些遗憾

赛前在自己电脑上跑的时候发现我们的算法耗时比较厉害，很多时候走不到 2000 步撑到比赛结束，因此圈地这一部分所有算法从简以节约时间，没有考虑到正式比赛时运行速度的提升，是我们的一大失算，导致圈地模式下的躲避过于简单以致在决赛第二场以一种莫名其妙的方式输给了 November 小组。

这种极端简单的躲避采用了粗糙的危险判定，直接拿自己离己方领地的最小距离与敌方离己方纸带的最小距离作差大于等于一个估算值作为判定条件，没有考虑实际路线所需的回家步数（因为可能回家的路上有自己的纸带）。对于回家路上有自己纸带的情形，这里的处理也是比较简单的，在撞上纸带之前就逆着之前画正方形的方向转弯（比如之前是逆时针画正方形的，那么在撞纸带之前应该右转以避免撞上自己的纸带）。但显然这样的处理时很粗糙的，它肯定不是最优路径，而且也不能保证躲避成功。因此，在测试时偶尔也会出现来不及躲避被杀的情形。不过，测试的时候大多数对局似乎表明这样的粗糙躲避在大多数时候还是有效的。

3 实验结果

3.1 测试数据

实验测试在各组员自己的电脑上进行。相关配置见下表。

	张懿卓	王斌昊	章文博	谭术超	刘威
CPU	intel® core™ i7-7500U CPU @ 2.70GHz	AMD A10-9600P RADEON R5,10 COMPUTE CORES4C+6G 2.40GHz	intel® Core™ i7-8550U 1.8GHz 4GHz	intel® Core™ i5-7500U CPU @2.50GHz 2.71GHz	intel® core™ i7-7500U CPU @ 2.70GHz
内存	8.00GB	8.00GB	16GB	4.00GB	8.00GB
操作系统	Windows10 家庭版	Windows 10 家庭版	Windows 10	Windows 10 家庭版	Windows 10
Python 版本	3.6.4	3.6.4	3.6.4	3.6.5	3.6.4

代码测试

刚开始我们是组员们分别独自开发代码。最先开发出来的就是初始的以攻击为主的一套代码。初代的攻击型代码击败了油聚上传的一些简单 AI。为了寻找代码的 bug，我们将后来逐渐开发的几套由弱鸡到成熟的圈地代码与主攻击的代码的对战，我们发现完善的主攻击代码都可以将他们击败。我们通过提交各种版本的攻击型代码与一些圈地代码到热身赛中测试，最终形成了这套以攻击为主的代码，并且通过对战和观察复盘数据不断修改 bug 并尝试减少时间损耗。我们还与非大一组的交流了代码，希望通过与他们对战找出更多 bug 并修复。在这些过程中我们逐步完善了代码的防御功能，也削弱了一定的攻击能力。在第二次热身赛结束后，我们又发现了有个别组开发出了可以将我们代码拖入无限循环的代码（比如在决赛中击败我们的 November 组）。我们又紧急制订了一个削弱了攻击性而具有简单的圈地功能函数的预备代码，用来对付这类对手。

3.2 结果分析

在测试、热身赛和最后的正式比赛中，我们采取的以攻击为主的策略取得了较好的效果，在很多情况下我们可以通过遍历获取敌人的信息在恰当的时候攻击敌人取得胜利。但是主要的缺陷也很明显。

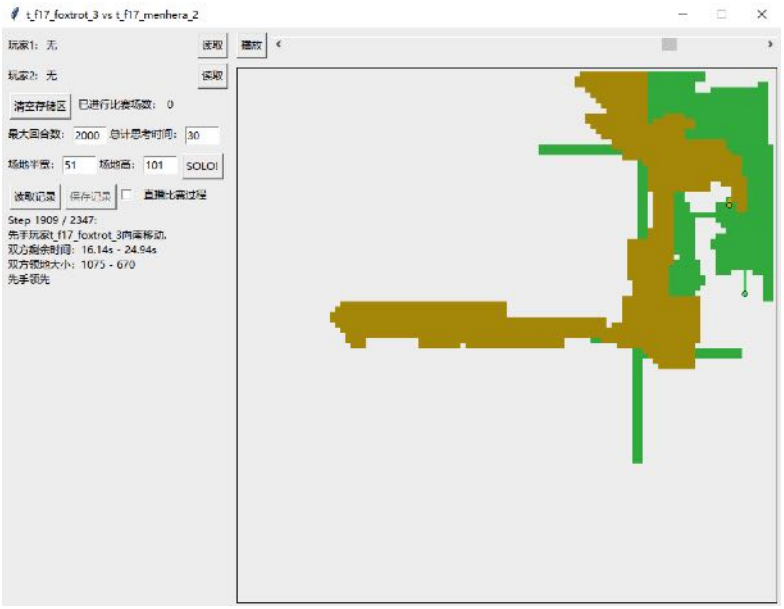
第一，防护措施不够，没有很好的躲避攻击的程序（在决赛中便被对手击杀了一次，在热身赛中也有不少的情况被对手击杀）。同时，代码中没有开发快速圈地的功能，当对手

是以圈地为主，并有很好的防护函数时，很容易在最后陷入了无限循环的走法之中，最终因为圈地面积太小或时间耗完，导致对战失败。在决赛中我们尝试用了一种具有简单圈地功能的代码与对手对战，还是因为圈地效率不如对方，陷入循环后告负。在耗时方面，由于遍历函数以及各种条件下控制纸卷移动的复杂度较高，导致代码的时间消耗极严重（在我们的电脑中正常情况下大概率无法完成 2000 回合）。这也就使得当对手的防护措施极高时，我们很可能会落败。在最终比赛中，时间扩充到 60s 解决了我们的超时问题（再此感谢扩时的陈斌老师与技术组）。

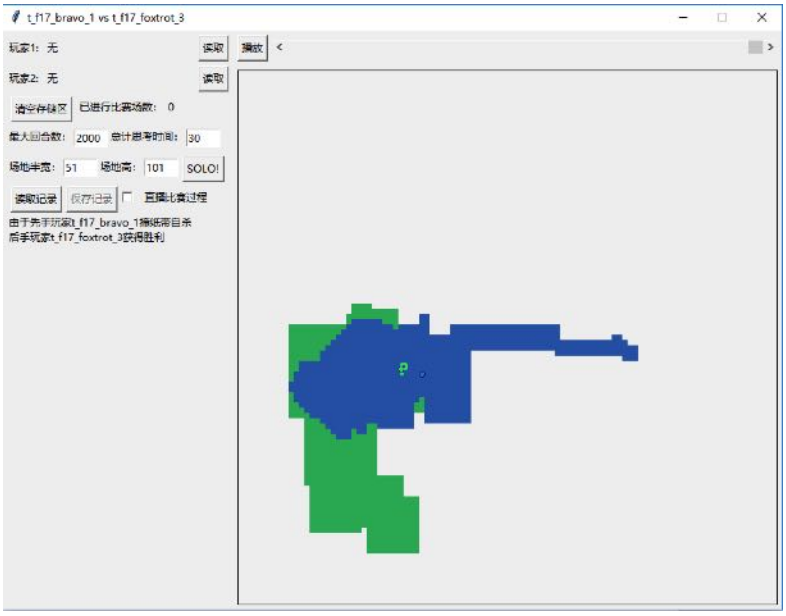
3.3 经典战局

(1) 第一次热身赛

Foxtrot VS Menhera: Foxtrot 向对手发起了猛烈的攻势，在 Menhera 的领地中对其穷追不舍，也暴露出了防守的不足，最后被对手抓住机会 gank 了。

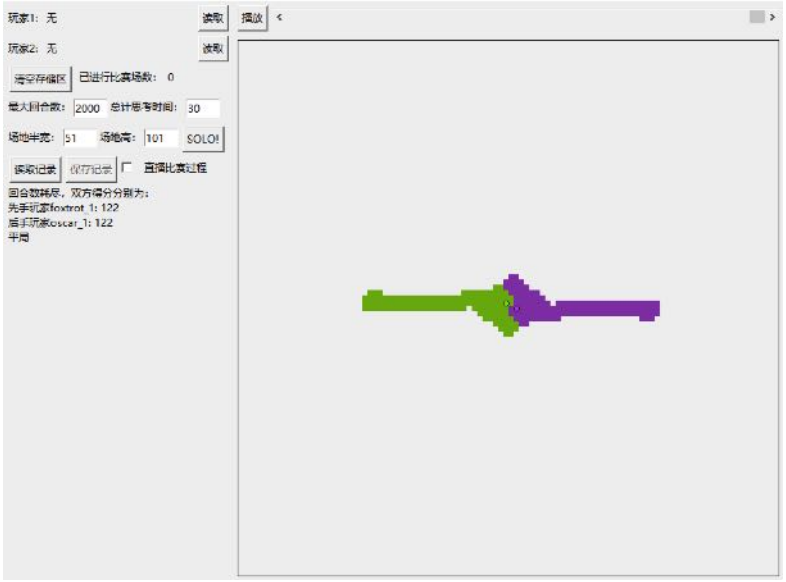


Bravo VS Foxtrot: Foxtrot 一如既往向对手发动猛攻，而 Bravo 也灵敏的一次次躲开了 Foxtrot 的攻击。但是人算不如天算，在躲避中，Bravo 渐渐迷失自我，最终被抄了后路，纸带成为孤家寡人，惨遭虐杀。



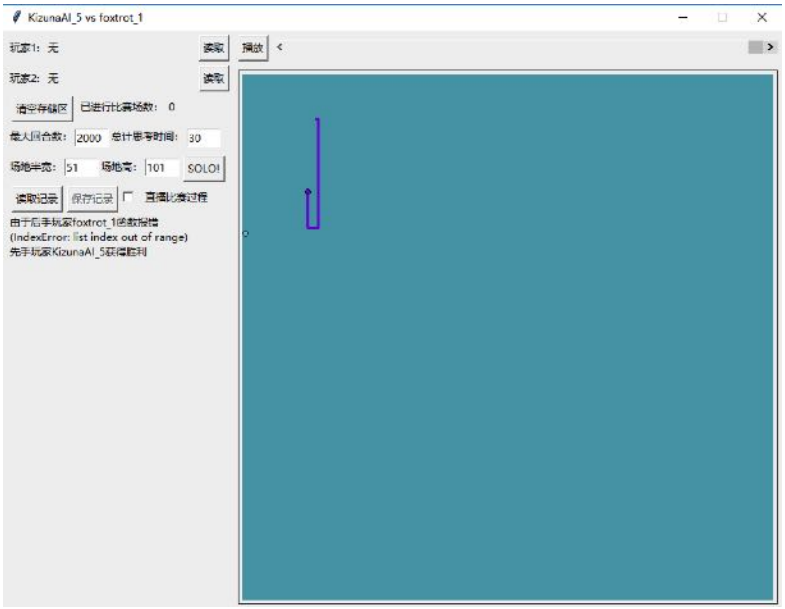
(2) 第二次热身赛

Foxtrot vs Oscar: 这局可谓是充满了笑点，oscar 老兄果然人如其名，浑身是戏，在后手中竟然能完全模仿 foxtrot 的走位，让凶猛的 foxtrot 无可奈何，出现了罕见的平局。

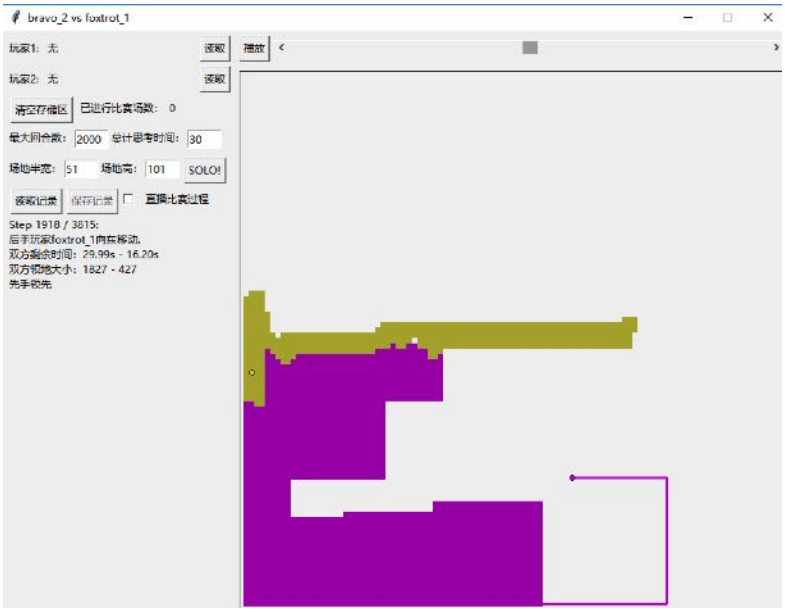


KizunaAI VS Foxtrot: 这局当凶猛的 foxtrot 去寻找对手决一死战之时，狡猾的 KizunAI 采取了大迂回策略，从中也可以看出 foxtrot 只有短期战术，对这种长远战略认识

不足，最终被对手给全图包饺子，惨遭虐杀。

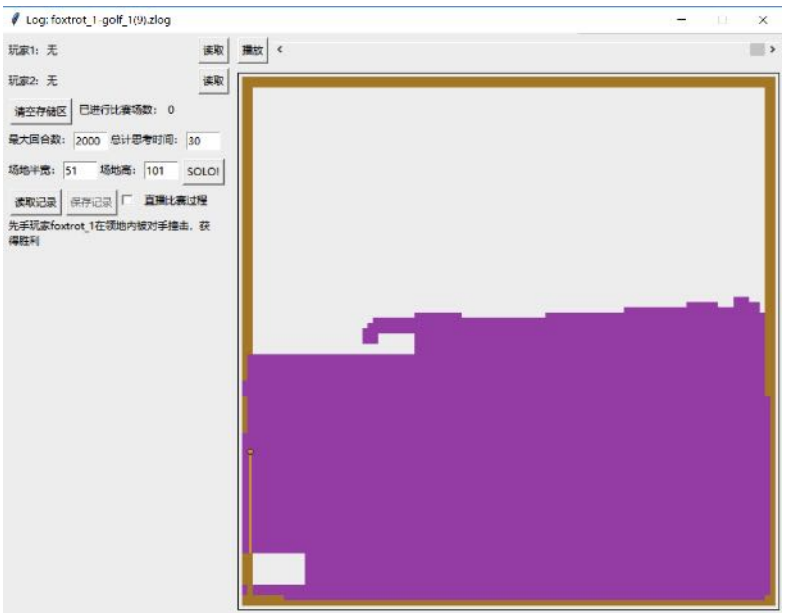


Bravo VS Foxtrot: 面对 Foxtrot 的凶狠攻势，bravo 选择了暂避锋芒。蠢萌的 Foxtrot 则选择守株待兔的策略，想要以逸待劳，殊不知，时不我待。最后因为超时 gg。

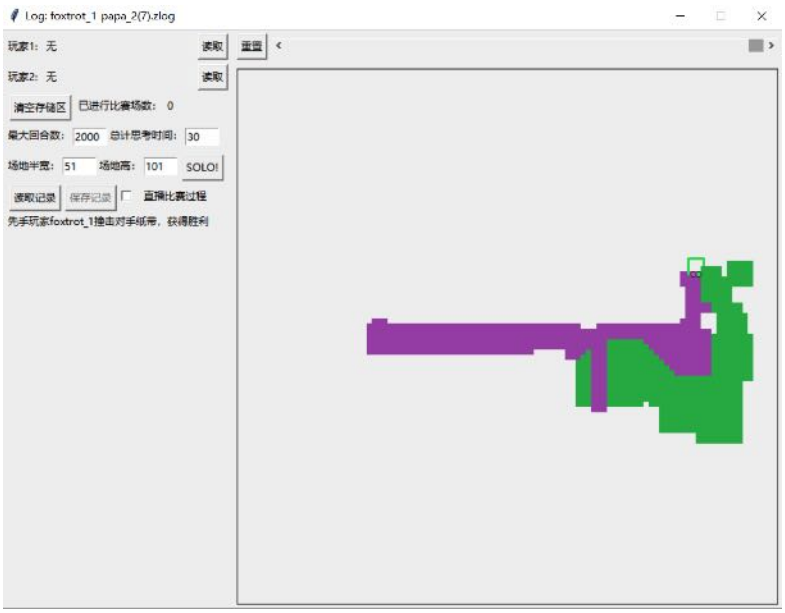


(3) 第三次热身赛

Foxtrot VS Golf: 这一局遭遇一个以圈满全场为目标的 AI。虽然在前期 foxtrot 也对 golf 进行了阻拦但还是几乎让 golf 完成圈地，好在接近尾声时 foxtrot 在半路拦截到 golf 取得胜利。



Foxtrot VS Papa: 这一局 foxtrot 对 papa 的 AI 的果断击杀，展现了 foxtrot 犀利的攻击能力。



4 实习过程总结

4.1 分工与合作

4.1.1 小组分工

算法：先由各个组员单独思考，之后集体交流、提出各自想法，最终确定进攻型算法策略。

代码：主要由张懿卓、王斌昊、章文博等人负责编写，其余组员（刘威、谭术超）辅助、提供意见。

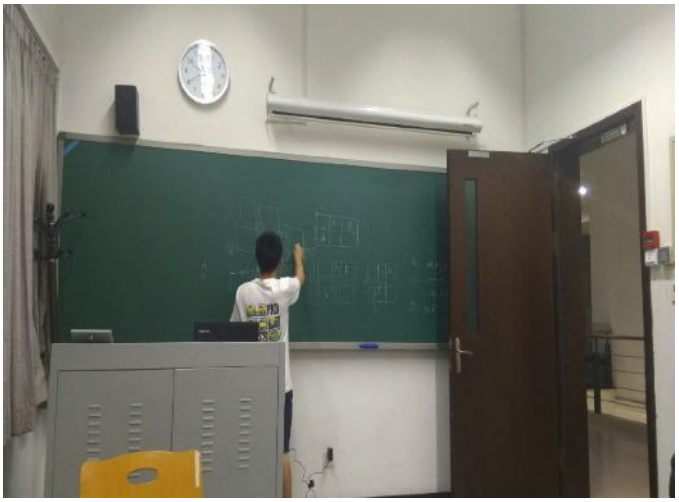
报告：代码组负责算法分析，张懿卓完善注释，谭术超负责总结、整理，由谭术超、刘威撰写报告，最终全组审核。

4.1.2 合作过程

合作交流方式: 线下组会交流和微信交流结合。组会交流确定主体策略，微信交流实时测试、提出修改意见。

各次组会交流记录

（1）第一次组会



刘威解释策略

时间：2018/06/02 地点：二教 526

大家讨论各自的想法，并将每个人初步编的代码进行对战，其中张懿卓提出攻击策略，王斌昊提出圈地策略，章文博提出模拟策略，刘威提出苟边界策略，最终决定先各自开发，然后在热身赛中根据结果选择主代码。

（2）第二次组会

分析对战结果

时间：2018/06/06 地点：宿舍

确定接下来四天的规划，完善代码，尝试开发出更强的代码，分析热身赛对战结果，着手实验报告。

1. 热身带之前完善当前主战代码，先由张懿卓主力。

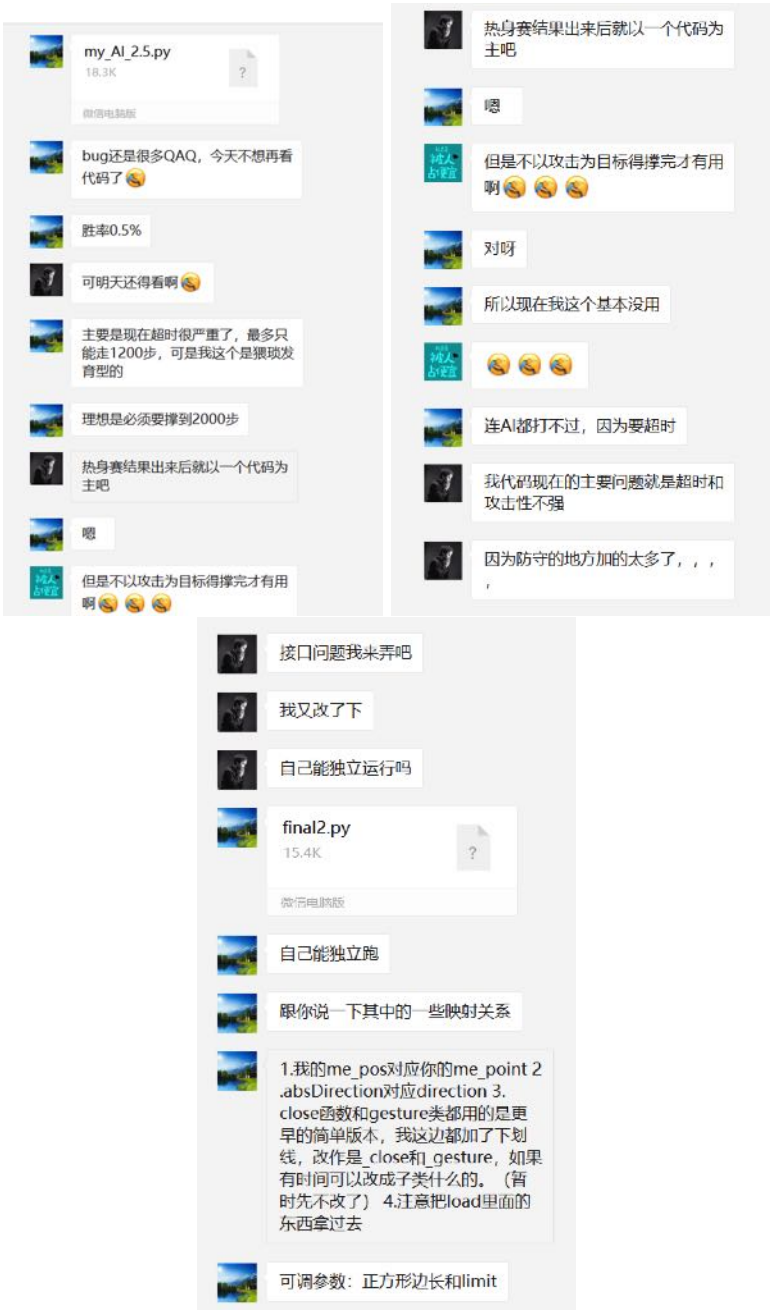
2. 尝试开发新代码。王斌昊和章文博继续完善自己的代码，周六热身赛提交所有代码，最终将名次最好的代码作为主战代码。

3. 热身赛之后完善主战代码。将确定下来的代码作为主体，其他两人将各自代码中感觉能兼容的部分加进去，之后继续完善。

4. 热身赛数据复盘。这次的先刘威一个人来做，周六热身赛后由刘威和谭术超分工做，需要找出三种情况: 死了，时间耗完了，明明能杀死对方却没杀。

5. 实验报告。代码分析部分: 编出主战代码的人写，其他两人补充。热身赛等实验部分: 刘威和谭术超写。其他部分和最终综合成完整的实验报告: 谭术超。

（3）多次线上交流


my_AI_2.5.py
18.3K
微信电脑版
bug还是很多QAQ，今天不想再看代码了
胜率0.5%
可明天还得看啊
主要是现在超时很严重了，最多只能走1200步，可是我这个是猥琐发育型的
理想是必须要撑到2000步
热身赛结果出来后就以一个代码为主吧
嗯
但是不以攻击为目标得撑完才有用啊
热身赛结果出来后就以一个代码为主吧
嗯
但是不以攻击为目标得撑完才有用啊
对呀
所以现在我这个基本没用
连AI都打不过，因为要超时
我代码现在的主要问题就是超时和攻击性不强
因为防守的地方加的太多了，，，
接口问题我来弄吧
我又改了下
自己能独立运行吗
final2.py
15.4K
微信电脑版
自己能独立跑
跟你说一下其中的一些映射关系
1.我的me_pos对应你的me_point 2.absDirection对应direction 3.close函数和gesture类都用的是更早的简单版本，我这边都加了下划线，改作是_close和_gesture，如果有时间可以改成子类什么的。（暂时先不改了） 4.注意把load里面的东西拿过去
可调参数：正方形边长和limit

4.2 经验与教训

这一次实习作业的结果对我们组的组员来说是比较满意的，它提供了一个应用数算课上所学的知识的机会，让我们体会到了码代码的乐趣。

在几次热身赛中，Foxtrot 组的代码在对战中表现不俗。第一热身赛中 foxtrot 处于中游，出线边缘。在组员的不懈努力下，对整体策略进行调整优化，改进代码，终于在第二、三次热身赛中获得第一。一直坚持地主攻策略有很好地优越性。

第三次热身赛时，由于更换服务器，进行了两轮，第二轮的结果和第一轮相比稍逊。经过分析，认为原因是更换服务器后硬件的处理能力有所降低，导致代码运行时容易超时，这也反映出一个问题——代码本身还存在一些冗余、处理问题的效率较低。于是迅速在前几次的基础上对代码进行优化，有效降低了超时的概率。

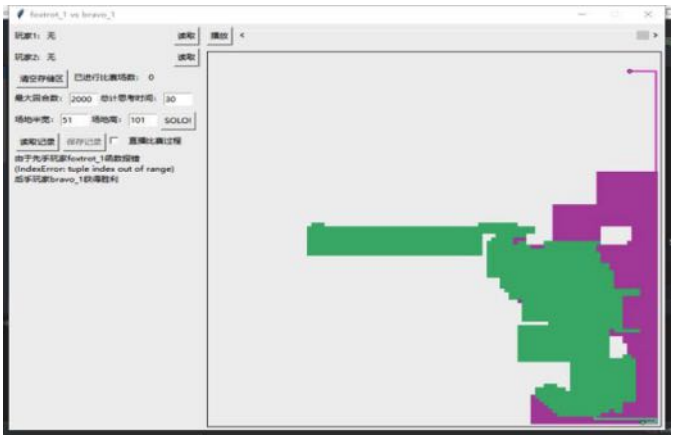
正式比赛前夕，在几个小组自发进行的友谊赛上发现了纯进攻策略的一些缺陷：当对方的攻击策略和自己相近时，双方容易陷入僵局，如果自己的圈地过小肯定会落败。意识到这个问题后小组内连夜修改代码，增加了圈地的扩展。由于在此前并没有这方面的想法，修改比较仓促，所以代码的劣势也很明显——对于圈地的考虑不周全，不能把握好圈地的时机，而且圈地函数里也会有一些冗余。这是策略上的失误，如果能早一些想到，在主策略刚提出时对这个问题进行讨论，或许情况会好很多。

另一个问题是，在之前的热身赛中已经显现出超时问题，为了缓解，将圈地算法从简处理，并没有考虑到正式比赛时会适当延长时间，而且运行速度也会有所提升，导致算法总体效率不高，进而导致与 November 比赛时莫名落败。

不足及改进

（1）报错 IndexError: tuple index out of range

记录：foxtrot_1 vs bravo_1



分析：我们的进攻模式中没有一处用到了元组 tuple，全部采用的是列表 list。因此，我们首先想到的是会不会是有函数有多个返回值？因为这样的有多个返回值的函数它返回的结果默认是 tuple 型的。经过查找，发现并没有 return 多个值的函数。那么问题只能是平台代码中的 tuple 下标越界了。

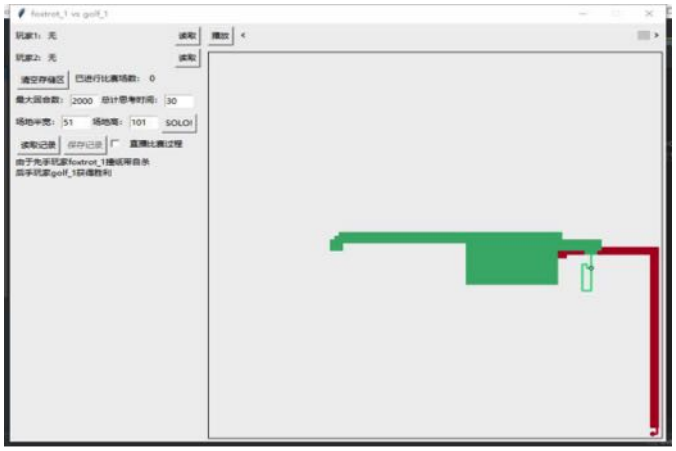
另一方面，注意到越界瞬间我们处于游戏场地边界。

综合以上考虑，需要排查的代码范围就大大缩小了，我们很快发现，报错的原因在于我们在本次热身赛加的一个新的判断，如下图所示。

解决方案：在这行代码前加一个自杀判断即可保证不越界。

（2）撞纸带自杀

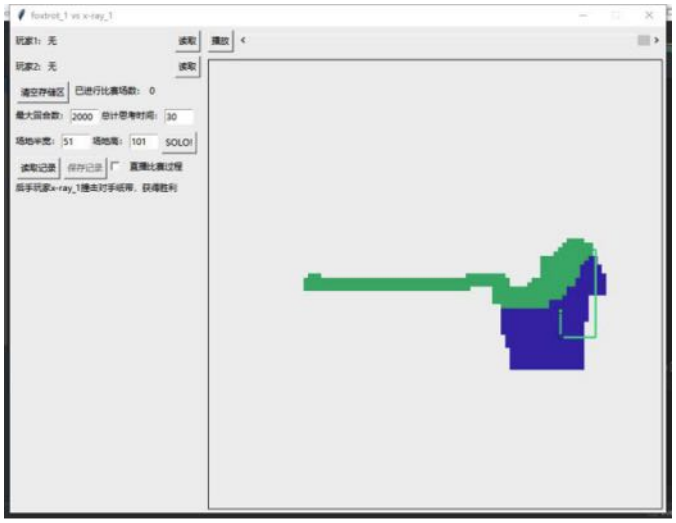
典型记录：foxtrot_1 vs golf_1



分析：照理说我们应该进行了防自杀判断，不应该出现这种情况。但是经过仔细研究复盘记录中的几次自杀问题，我们发现：这种情形只出现在对手在领地内外切换的时候。因此，我们猜测，我们之所以撞纸带自杀时因为目标发生变化导致的，就拿上图所示的复盘记录来说，本来对手在领地内，根据我们的算法，我们会去蚕食地方领地，即我们的目标是对方的地；而当对方走出领地后，根据我们的进攻算法，我们的目标变成了对方的纸带，在这种切换下我们的自杀判定失效了。解决方案：保持计划执行的连续性，即使用一个模式标记，尽量每次把一个计划执行完毕后再执行另一个计划，避免自杀问题。

（3）来不及回家

典型记录：foxtrot_1 vs x-ray_1



分析：从图中可以看出，我们的算法的路径规划似乎没有什么问题，确实是按最短路径回家的，因此问题应该是在于安全距离判定，我们的判定中使用的估计在少数情况下有问题。

解决方案：将安全距离判定的条件变得更保守一些。但是问题是变得保守之后会削弱算法的攻击性，因此我们最后只进行了微调。因为为偶尔一次无法逃回家而降低攻击的效率是不值得的。

4.3 建议与设想

4.3.1 关于本次实习作业的感想：

组队时不一定要抱大佬大腿。虽然和大佬一起编程确实能把实习作业完成得更好，但是可能少了一个能让自己得到锻炼的机会。毕竟自己投入其中获得的满足感和抱大腿是不一样的。另外，对于比赛结果也不要太在意，友谊第一，比赛第二；这个比赛只是检测实习作业的成果的一个方式而已，重要的是过程，不论输赢，有收获就是成功的。

4.3.2 关于代码后期的设想：

1. 为了得到更好的综合效果，在代码中没有采用广搜等算法来加快搜索速度，而是采用耗时最长但也是最全面的遍历来获取数据，因此常常会出现超时等问题。因此，代码之后还可以利用一些算法提高搜索速度。

2. 我们组基本没有开发圈地算法，因此在决赛中会因此失败。之后可以在设置一个判断，当所围面积大于敌方面积时再进攻，效果应该会更好一些。

4.3.3 小组成员感想

很开心为这次小组取得不错的成绩尽了一份绵薄之力。但是稍稍遗憾的是由于自己在纸带策略方面没有很好的想法，因而在代码开发方面没有能够给予小组很大的帮助。到了后期已经全然不参与了代码开发工作，而负责了复盘数据这样一些比较机械的工作，因而在编写代码方面的体验感略有不足。不过总体来说，这次大作业带给我的收获还是蛮多。希望下次在集体活动中我能为小组贡献更多的力量。——刘威

经过这两个多星期的码代码过程，我对代码产生了深深的热爱 (雾)，同时也获得了许多编写程序的经验，比如提前规划好大致结构可以为代码编写节省很多时间，好的算法与数据结构可以使代码的耗时大大减少。我也希望可以将这些知识运用到本专业的学习中。——张懿卓

从理论上的战略分析，到具体的代码实现，整个过程虽然耗时巨大，但是也是自己第一次体验完成一个 project 的感觉。倾听他人的想法，发现了一些有趣的策略；观看同学的代码，学到了许多很好的封装技巧。虽然自己的代码实现能力依然很差，但是本学期的课程使我摆脱了被编程支配的恐惧（情不自禁），而且感受到了数据结构与算法的强大应用，这是之前只是单纯编程所看不到的更高视角，希望以后能够得以灵活运用，继续提升自己在编程方面的兴趣和能力。——王斌昊

这次实习小组取得不错的成绩，但是比较遗憾自己因为个人安排的原因，在代码上没有提出太多意见，在开发上没有太多协助。能在最后负责报告，也是我对小组工作的贡献。在最开始试写代码时也发现自己编写起来还欠火候，在以后的学习中还需要多多练习。——谭术超

5 致谢

首先感谢陈斌老师这学期的奉献；

其次感谢助教哥哥、助教姐姐们的耐心辅导；

然后感谢各位组员的互相帮助；

之后感谢技术组的大佬们的默默付出；

最后感谢各位同学的陪伴。

特别感谢为我们组的代码做出贡献的各位大佬。

第八章　F17_Golf 报告

何为 * 甘天奕 郑煜衡 杨昌赫 谢思远

摘要：Golf 组的算法思路是，建造一个边界类用图来储存并读取历史数据，使用 a 星算法计算对方到己方轨迹的最短距离，在保证自己绝对安全的情况下进行最大圈地策略，并不断调用判断攻击函数，一旦发现有进攻的机会立即进入攻击状态，在攻击状态下，不断调用函数判断攻击路径和自身安全，一旦遇到危险便调用函数最快回到己方领地继续进行圈地策略。

关键字：图，a* 算法，最大圈地策略，判断攻击函数，攻击状态，攻击路径，最短回领地路径

1 算法思想

1.1 总体思路

我们最初的想法是兼顾围地和进攻，在自己不处于可被对方击杀的危险状态，且不确定能杀死对方时，就最大化地围地，通过计算自己纸片边界的形状来找到接下来最优的围地走法。然而，因为每一步都要进行状态判断，相应的围地走法也会改变，所以围地走法的不确定性太大，而且精确地最大化围地会耗费大量时间进行计算，这样围地也没有一个确定的可以逐步进行下去的算法来实现，因此这个想法中的围地策略被舍弃，需要调整。

我们后来的想法是，制定一个可以循环并且可以在每轮循环里能够微调的围地算法，在这个可循环的围地算法的每一轮循环中，依据判断自己和对方的安全状态对围地进行调整，如果确定能击杀对方则脱离围地算法进入进攻模式，而如果不能击杀并且自己处于危

险状态则立即返回，返回的走法不会破坏整个围地算法的循环，如果自己既不能击杀对方也处在安全状态，则继续执行最贪心的围地策略。

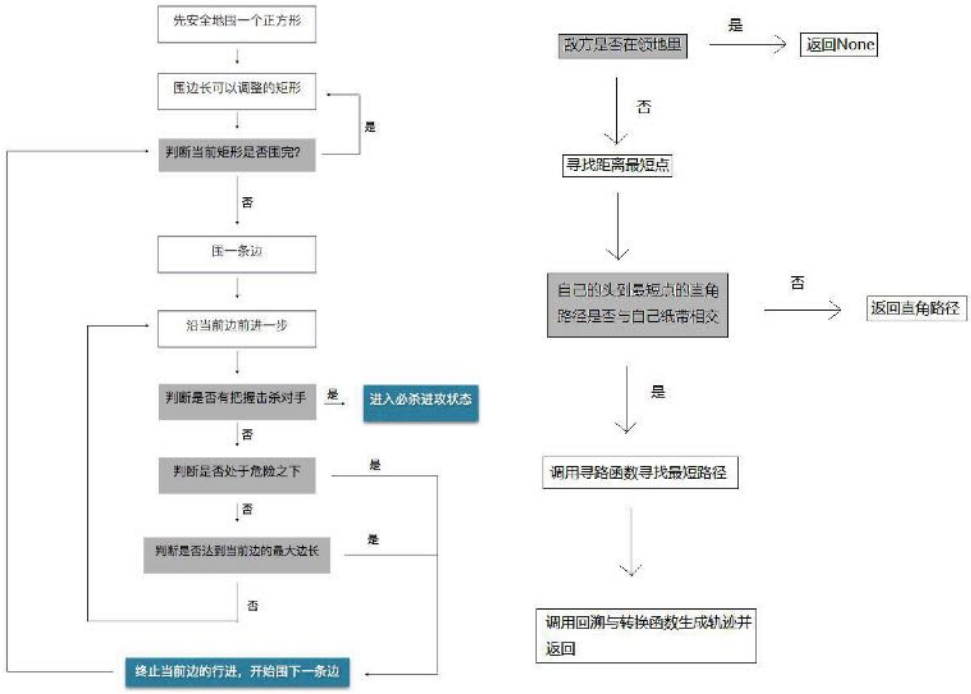
整个算法的难点部分也是占比最大的部分是如何判断自己和对方的安全状态，为了方便进行计算，自定义多个新的类，把游戏数据给的二维数组地图转译成多个向量，向量表示纸带或者纸片边界的各个折线段，采用 A* 算法寻路，找到击杀对方的线路，这个线路也是由多条向量拼接而成的。

Golf 组算法最终的大体框架是，不断地依附纸片的某一条边界围矩形（这是可循环进行的，围一个矩形后能依附该矩形的一条边继续围出下一个矩形，不断调整围矩形的顺逆时针可以保证纸卷不会停留在自己的纸片内），每两个相邻矩形间总有一条对齐的公共边，从而向外扩展自己的纸片，在向外围矩形的每一条边时，根据一个判断自身状态的函数调整自己下一步的策略，策略分为三种——击杀对方、快速返回领地以及继续扩大边长围矩形，实现方法与我们最初想法中的思路相同，快速返回领地通过立即转向结束矩形某条边的形成来实现，扩大领地通过增长当前所在边的长度（不撞墙）来实现，这样后两种策略都不会影响矩形的围成，调整边长参数就可以让围地算法继续循环。

按照这个围地算法，假如对方一直很谨慎，使我方没有进攻机会，则我们一直执行原来的围地策略，在地图总体上看，是沿着斜线不断作矩形延伸纸片，碰到地图边界时沿着斜线反弹（有点类似于打台球时的反弹线路），这样在矩形很宽我方纸带不易被切断的情况下，每经过一段时间后，可以使自己的整个反弹线路闭合，扩大围地面积。此外，除了这个模式外，我们在后来的调试过程中加入了另一个模式，以上述第一个模式为优先，当上述第一个模式对战失败并且对方没有向地图边界发展时，就切换为第二个模式，沿着边界包围对方纸片，接下来，当一种模式失败时就切换到另一个模式进行对战。

1.2 算法流程图

左图是整体流程图，右图是寻找进攻路径的算法流程图.



1.3 算法运行时间复杂度分析

Trace 类和 Border 类部分：

Trace 类的方法基本都是 O(1) 的，主要的时间开销来自 Border 的 extend 方法和 eaten 方法。eaten 函数中的 insert_vertex 是 O(n) 的，其余操作为 O(1)，故总的时间复杂度为 O(n)；而 extend 分为三部分，将 trace 加入是 O(|V1|) 的，|V1| 是 trace 的节点数；第二部分是深度优先遍历，由于过程中涉及 insert_vertex 和 find_neighbor 操作，时间复杂度为 O(n|V2|)，|V2| 是自己 border 的节点数；检查对方的领地的节点时 O(|V3|) 的，|V3| 是对方 Border 的节点数。

判断攻防状态部分：

在判断攻防时，需要计算攻防的点与进攻路径，并判断进攻路径是否与自己的已有轨迹相交，首先取进攻点的时要遍历对方轨迹，复杂度为 O(k),k 为对方轨迹便所包含的线段数 (找点到线段的距离为 O(1))，之后生成最简单的路径并判断是否相交，复杂度也为 O(k)，如最简路径与己方路径相交，调用寻路算法，考虑地图分割开销为 O(d^2)，d 为进攻点与自己的头之间的距离，寻路算法的开销与广度优先算法相同, 又有大约 d^2 个点，d^2 条

边，算法复杂度为 O(d^2)，回溯路径时一般要将 d 个点转化为轨迹，用栈的方法需遍历一遍，复杂度为 O(d), 综合来看，总复杂度约为 O(d^2)(实战中双方纸带所包含的线段数一般不超过 10 条，除寻路外所花的时间应远小于寻路)

围地函数：

围地函数内部是由多个情况分类的判断语句实现的，没有循环结构，通过在 storage 中存储表示围矩形时所处的状态的标志变量来进行状态的保存和判断，每次调用时都会进行 1 次 Trace、Border 类的转化和攻防状态的判别，时间复杂度为二者之和。

2 程序代码说明

2.1 数据结构说明

采用图的数据结构（Graph 类）用来存储纸带（Trace 类）和领地的边界类（Border 类）。Trace 类和 Border 类都是 Graph 类的子类。由于 Border 和 Graph 都是属于某一方的，因此增加属性 id。

对于图的节点（Vertex 类），用属性 coordinate 记录坐标，同时还把所有的 Vertex 类都存入一个字典（map_dict）中。对于课件上的 Vertex 类进行了改进，主要在于纸带圈地游戏中图的邻节点的方向只能是东南西北，因此用一个长度为 4 的列表存储邻节点，索引为 i 的邻节点在第 i 个方向。

Vertex 类还是 Graph 类及其子类都增加了很多方法，将在下面的函数说明中一一介绍。

由于储存时所用的类较为复杂，后续计算不方便，进行简化后重新定义 trace,border 等类，（用转换函数实现 2 个类之间的翻译），point 类包含 x，y 两个分量，并定义了 add 等方法，line 类构造为用 begin 和 end 所表示的两个点，并储存了 line 的绝对方位 (由 begin 指向 end)，并当两点不能构成 line 时将 wrongline 属性设为 True，便于后续检查，trace 类为用 list 保存的由一系列 line 表示的轨迹，并定义了 add 方法便于后续实现，border 类与 trace 类相近，二者都检查了所构成的 line 是否合法，并也有相应的类记录自身合法性。

寻找进攻线路时，所用数据类型有栈与图，栈的表示方法与课程相同，用于将由点表

示的轨迹转化为由 line 表示，方法是判断新的点与栈顶的点的连线方向是否与之前线段方向一致，如一致说明线段未到头，将新的点入栈，如不一致说明轨迹出现转折，将栈顶的点弹出并储存，将栈清空后将新的点入栈。依此直到遍历所有的点后，将储存的点依次连接为 line，转换完成。

图的改进是由于每个点都只与自己相邻的四个点相连，则不需要额外记录边，可以直接使用二维数组表示图，所采用的寻路算法是基于广度优先搜索，区别是加入了障碍物（自己之前的轨迹），当搜素到障碍物时，将其视为已经发现的点即可。同时由于边的特殊性，每个点只用储存一个方向即可进行回溯，数据结构相对于广度优先搜索大为简化。

2.2 函数说明

add 函数：对向量进行加法

```
def add(list1, list2):
  '''
  将两个向量相加，向量以列表或元组的形式传入返回元组，
  :param list1: 第一个向量，或listtuple
  :param list2: 第二个向量，或listtuple
  :return: tuple
  '''
  return (list1[0] + list2[0], list1[1] + list2[1])
```

judge_direction 函数：判断向量方向

```
def judge_direction(start, end):
'''对给定向量判断方向，传入向量的起点和终点，返回方向

:param start: 起点
:param end: 终点
:return: 一个整数，表示方向，对应方向的单位向量为idirections[i]
'''
    start, end = start.coordinate, end.coordinate
    for i in range(2):
        if start[i] == end[i]:
            if start[not i] > end[not i]:
                return ((not i) + 2)
            elif start[not i] < end[not i]:
            return not i
```

points 函数：给定两个点，返回一个包含两点之间所有点的列表

```
def points(start, end, direction):
'''给定向量，返回一个列表，列表中的元素为元组，包含起点到终点之间所有点的坐标，不包含起点和终点

:param start: 起点，类Vertex
:param end: 终点，类Vertex
:param direction: ，向量方向int
:return: 返回坐标的列表（），列表元素为元组list
'''
    point_list = []
    if start.coordinate == end.coordinate:
        return point_list
    current = add(start.coordinate, directions[direction])
    while current != end.coordinate:
        point_list.append(current)
        current = add(current, directions[direction])
    return point_list
```

Vertex：

属性：

coordinate：坐标，元组形式

connections：与这个点相连的点，列表中的元素为 Vertex 类，且第 0 个元素表示 0 方向上的点，以此类推。（0,1,2,3 分别表示东南西北，下同）

color：颜色，0,1,2 分别表示白灰黑，用于 dfs 方法：

add_neighbor：增加邻节点

```
def add_neighbor(self, neighbor, direction=None):
    '''传入和，可以不传，不传则自动判断方向，
    neighbordirectiondirectionself.中放入，connectionsneighbor
    neighbor.中放入。connectionsself
    :param neighbor: 类，邻节点Vertex
    :param direction: ，表示方向int
    :return: 不返回
    '''
    if direction is None:
        direction = judge_direction(self, neighbor)
    self.connections[direction] = neighbor
    neighbor.connections[(direction + 2) % 4] = self
```

del_neighbor：删除邻节点并返回删除的邻节点

```
  def del_neighbor(self, direction):
  '''
  传入，删除与此方向上的邻节点的联系int
  :param direction: ，表示方向int
  :return: 返回被删除的neighbor
  '''
  neighbor = self.connections[direction]
  if neighbor:
  self.connections[direction].connections[(direction + 2) % 4] = \
  self.connections[direction] = None
  return neighbor
```

str：返回表示类的字符串

```
def __str__(self):
  '''
  返回表示的字符串，形式为：selfVertex:[x, y] neighbor: x x x （的坐标加上的邻节点的坐标）xselfself
  :return: ，用来表示strself
  '''
  return 'Vertex:' + str(self.coordinate) + 'neighbor:' + ' '.join(
  (str(each.coordinate) if each else 'None') for each in self.connections) + '\n'
```

del_self：删除自己

```
'''删除，清出邻节点与相关的内容，以及中与相关的内容
selfselfmap_dictself
:return: None
'''
for i in range(4):
    if self.connections[i]:
        self.del_neighbor(i)
if self.coordinate in map_dict:
     map_dict.pop(self.coordinate)
```

erase：三点共线时删除中间的点

```
def erase(self, border):
    '''如果只有两个邻节点且三点共线，则清除，删除在中的存储，从的列表中删除。
    selfselfselfmap_dictborderself
    :param border: 类Border
    :return: None
    '''
```

```
    for i in range(2):
        if self.connections[i + 1] == self.connections[(i + 3) % 4] == None and \
            self.connections[i] and self.connections[i + 2]:
            map_dict.pop(self.coordinate)
            self.connections[i].connections[i + 2] = self.connections[i + 2]
            self.connections[i + 2].connections[i] = self.connections[i]
            self.connections[i] = self.connections[i + 2] = None
            border.vertlist.remove(self)
            break
```

neighbor_num：返回邻节点个数

```
def neighbor_num(self):
    '''计算此节点邻节点的个数

    :return: ，表示邻节点的个数int
    '''
    i = 0
    for each in self.connections:
        if each:
            i += 1
    return i
```

move：移动节点，进入自己领地触发 extend 方法，进入对方领地触发 eaten 方法。

```
def move(self, direction, storage=1):
    '''将点按传入的方向移动一个单位，如果点有超过一个邻节点会报错!!! 修改，只有在移动之前的点对
    map_dict应的的元素是时才会删除这个点，只有在之后的点对应的中的位置没有元素且
    map_dictselfmap_dictstorage=1时才会加入。

    :param direction: ，表示移动的方向int
    :param storage: ，或int01
    :return: None
    '''
    if self.coordinate in map_dict and map_dict[self.coordinate] == self:
        map_dict.pop(self.coordinate)
    self.coordinate = add(self.coordinate, directions[direction])
    if self.coordinate not in map_dict and storage:
        map_dict[self.coordinate] = self
```

find_neighbor 方法：

传入一个 Border 类和一个 prev_direction 表示方向，寻找 self 周围是否有其他节点，self 必须只有一个邻节点，且将邻节点相反的方向看作前方，先初始化一些变量。

```
    direction, find = None, 0
    for each_direction in range(4):
        if self.connections[each_direction]:
            direction = (each_direction + 2) % 4
            break
```

则 direction 表示前方，而 find 表示是否找到邻节点的 bool 变量; 然后寻找邻节点，分为两种情况：

1、prev_direction = None：直接寻找左右两边有没有节点，这里左右指和前方垂直的两个方向。代码如下：

```
for each_neighbor in [-1, 1]:
    new_point = add(self.coordinate, directions[(each_direction + each_neighbor) % 4])
    if border.belong_to_self(new_point) and new_point in map_dict:
        self.add_neighbor(map_dict[new_point])
        map_dict[new_point].erase(border)
        return None
```

2、prev_direction 为 int：

首先考虑这个点是否在领地内部，由于传进来的时候就能保证这个点最多只有前面三个点可能不是自己的领地，因此用下述代码即可判断：

```
list1 = [directions[prev_direction]]
list1.append(add(directions[prev_direction], directions[(prev_direction + 1) % 4]))
list1.append(add(directions[prev_direction], directions[(prev_direction - 1) % 4]))
# 中是方向的向量及其两侧的斜向量list1previous

for each in list1:
    each = add(each, self.coordinate)
    if not border.belong_to_self(each):
        break
else:
    return None
```

如果不在领地内部，则需要寻找邻节点，依次考虑以下三个点：

（1）朝 prev_direction 方向的点，代码如下：

```
        point1 = add(self.coordinate, directions[prev_direction])
        if border.belong_to_self(point1) and point1 in map_dict:
            self.add_neighbor(map_dict[point1])
            map_dict[point1].erase(border)
            return None
```

（2）前方的点，代码如下：

```
        point2 = add(self.coordinate, directions[direction])
        if point2 in map_dict:
            self.add_neighbor(map_dict[point2])
            self.erase(border)
            return None
```

（3）先向前移动一格，再找（1）方向的点，代码如下：

```
            self.move(direction)
            border.insert_vertex(self, prev_direction)
            return None
```

Graph 类：

属性：

id：玩家编号

vertlist：节点的列表，元素为 Vertex 类

方法：

clear：清除所有节点的 color

```
def clear(self):
    '''恢复所有点的为默认值，在之前调用
    colordfs
    :return: None
    '''
    for each in self.vertlist:
        each.color = 0
```

add_vertex：增加节点

```
def add_vertex(self, vertex):
    '''增加一个节点，在里存下这个点
    map_dict
```

```
    :param vertex: 增加的节点，类Vertex
    :return: None
    '''
    if vertex not in self.vertlist:
        self.vertlist.append(vertex)
    map_dict[vertex.coordinate] = vertex
```

del_vertex：删除节点

```
def del_vertex(self, vertex):
    '''删除一个节点

    :param vertex: 要删除的类Vertex
    :return: None
    '''
    if vertex in self.vertlist:
        self.vertlist.remove(vertex)
    vertex.del_self()
```

iter：返回一个迭代器，包含全部节点

```
def __iter__(self):
    return iter(self.vertlist)
```

str：返回一个表示自己的字符串，包含所有节点信息

```
def __str__(self):
    return ''.join(str(each) for each in self.vertlist)
```

Trace 类：Graph 的子类

属性：

Graph 的基本属性

current：终点的节点

head：起点的节点

direction：当前方向

方法：

go_straight：向前走一步

```
def go_straight(self):
    '''向前走一步，如果到自己领地执行，到对方领地执行
    extendeaten
    :return: None
    '''
    self.current.move(self.direction, 0)
    if stat['now']['fields'][self.current.coordinate[0]][self.current.coordinate[1]] == self
                                                        .id:
        borders[self.id - 1].extend(self)
    elif stat['now']['fields'][self.current.coordinate[0]][self.current.coordinate[1]] == 3
                                                        - self.id:
        borders[2 - self.id].eaten(self)
    elif self.current.coordinate not in map_dict:
        map_dict[self.current.coordinate] = self.current
```

turn：转弯

```
def turn(self, turn):
    '''转弯，表示方向，左转
    turn-，右转为，首先改变的方向，然后增加一个节点，最后前进一步11trace
    :param turn: ，表示方向int
    :return: None
    '''
    self.direction = (self.direction + turn) % 4
    new = Vertex(self.current.coordinate)
    self.vertlist.append(new)
    self.current.add_neighbor(new, self.direction)
    self.current = new
    self.go_straight()
```

str：返回一个字符串，用来表示自己，具体形式为箭头依次连接起点到终点的每一个点

```
def __str__(self):
    return '->'.join(str(each.coordinate) for each in self.vertlist)
```

Border 类：

属性：只有 Graph 的属性，没有新增其他属性

方法：

insert_vertex：在两个相邻节点之间插入一个节点

```
def insert_vertex(self, new, direction):
    '''给边界增加一个节点，将中该节点的坐标，对应的点加入，如果这个点不在的
    map_dictself内则加入其中。
    vertlist
    :param new: 类，表示增加的节点Vertex
    :param direction: 插入的边的方向
    :return: None
    '''
    direction_list = [direction, (direction + 2) % 4]
    curr_list = [new.coordinate] * 2
    map_dict[new.coordinate] = new
    while 1:
        for i in range(2):
            curr_list[i] = add(curr_list[i], directions[direction_list[i]])
            if self.belong_to_self(curr_list[i]) and curr_list[i] in map_dict:
                other = map_dict[curr_list[i]].connections[direction_list[not i]]
                if other:
                    other.add_neighbor(new, direction_list[i])
                new.add_neighbor(map_dict[curr_list[i]], direction_list[i])
                if new not in self.vertlist:
                    self.vertlist.append(new)
                return None
```

include：判断一个点两侧是否都为自己的领地

```
def include(self, point, direction_list):
    '''判断一个点的两侧是否全是的领地
    self
    :param point: 传入的点，或listtuple
    :param direction: ，表示方向，检查此方向两侧的点int
    :return: None
    '''
    for i in direction_list:
        if not self.belong_to_self(add(point, directions[i])):
            return 0
    return 1
```

put_side：将一个点加到另一个点旁边

```
def put_side(self, old, new, direction):
    '''将节点放在节点的某个方向（），如果的方向上原本就有节点，则将插入其中
    newolddirectionolddirectionnew
    :param old: 类，原本就有的节点vertex
```

```
    :param new: 类，新加的节点vertex
    :param direction: 相对于的方向newold
    :return: None
    '''
    if old.connections[direction]:
        new.add_neighbor(old.connections[direction], direction)
    old.add_neighbor(new, direction)
    if new not in self.vertlist:
        self.vertlist.append(new)
    map_dict[new.coordinate] = new: 判断一个点是否属于自己的领地
belong_to_self
def belong_to_self(self, p):
    '''判断一个点是否为自己领地

    :param p: 点，或listtuple
    :return: 或TrueFalse
    '''
    if p[0] < 0 or p[0] >= WIDTH or p[1] < 0 or p[1] >= HEIGHT:
        return False
    return stat['now']['fields'][p[0]][p[1]] == self.id \
           and stat['now']['bands'][p[0]][p[1]] != 3 - self.id
```

get_lines：找到边界上所有的边

```
def get_lines(self):
    '''找到边界上所有的线段并返回为一个
    list
    :return: ，元素为列表，列表的元素为起点和终点list
    '''
    line_list = []
    self.clear()
    self.vertlist[0].color = 1
    self.iter_dfs(self.vertlist[0], line_list, None)
    return line_list: 帮助实现功能的函数，寻找一个节点周围的边，并不断递归，实质上是深度优先遍历。
iter_dfsget_line
def iter_dfs(self, vertex, line_list, prev):
    '''深度优先遍历，将传入的点与周围的点形成的边加入一个列表，除了与上一个节点形成的边

    :param vertex: 传入一个节点
    :param line_list: 放边的列表
    :param prev: 上一个节点的方向
    :return: None
    '''
    for i in range(4):
```

```
        if i != prev and vertex.connections[i] and vertex.connections[i].color < 2:
            line_list.append([vertex, vertex.connections[i]])
            if not vertex.connections[i].color:
                vertex.connections[i].color = 1
                self.iter_dfs(vertex.connections[i], line_list, (i + 2) % 4)
    vertex.color = 2
```

extend：

自己的 trace 碰到自己的领地时调用，修改围成的区域的 border，传入参数为 Trace 类。

首先将 trace 连接到 border 类上，使之成为 border 的一部分

```
for i in range(4):
    if trace.head.connections[i]:
        head_direction = i
        break
# 找到的起点的方向trace

if trace.head.coordinate in map_dict:
    map_dict[trace.head.coordinate].add_neighbor(trace.vertlist[1], head_direction)
else:
    self.insert_vertex(trace.head, (head_direction + 1) % 4)
# 将的起点连入图中trace

if trace.current.coordinate in map_dict:
    map_dict[trace.current.coordinate].add_neighbor(trace.vertlist[-2])
else:
    self.insert_vertex(trace.current, (trace.direction + 1) % 4)
# 将的终点连入图中trace

for each in trace.vertlist[1: -1]:
    self.vertlist.append(each)
# 将的点全部加入的中trace border vertlist
```

然后对整个图进行深度优先遍历，深度优先遍历的操作间下面的 dfsvisit 函数。

```
self.clear()
for each in self:
    if not each.color:
        self.dfsvisit(each)
# 进行深度优先遍历，重新塑造自己的border
```

最后清理掉对方在自己领地内的 border 节点，这些部分是被自己吃掉的对方的领地。

```
del_list = []
for each in borders[2 - self.id]:
    if self.belong_to_self(each.coordinate):
        del_list.append(each)
for each in del_list:
    borders[2 - self.id].del_vertex(each)
# 删除自己领地中对方的节点，这些节点代表被吃掉的部分border
```

dfsvisit：

传入一个 Vertex 类，进行深度优先遍历，清除领地内部的边和点，同时修复边界上因此而断裂的位置。

首先将这个点颜色设为灰色

```
startVertex.color = 1 # 先将颜色设为灰色
```

之后依次寻找四个方向上的点，首先要初始化一些变量：

```
for i in range(4):
    if startVertex.connections[i] and startVertex.connections[i].color < 2:
        direction_list = [(i + 1) % 4, (i - 1) % 4]# 是左右两个方向direction
        status = self.include(startVertex.coordinate, direction_list)
        # 表示状态，表示两边都是领地，则相反status10
        point_list = points(startVertex, startVertex.connections[i], i)# 一条边上的所有点
        tail = startVertex.connections[i]# 边的终点
```

然后对这个方向上的边进行检查：

```
for j in range(len(point_list)):
    new_status = self.include(point_list[j], direction_list)# 检查新的点两侧是否都为领地
    if status != new_status:
        status = new_status
        #如果状态发生变化首先更新status
```

在 status 发生改变的地方插入节点，如果 status 从 0 变成 1 就直接在这里插入节点，从 1 变成 0 则在前一个点插入节点，如果这个节点恰好是起点则无需插入，因此最后的代码如下：

```
if j or new_status:
    self.insert_vertex(Vertex(point_list[j - 1 + new_status]), i)
    if not new_status:
```

```
        other = self.vertlist[-1].del_neighbor((i + 2) % 4)
        if self.vertlist[-1].neighbor_num() == 1:
            self.vertlist[-1].find_neighbor(None, self)
        if other.neighbor_num() == 1:
            other.find_neighbor((i + 2) % 4 if other == startVertex else None, self)
# 除了j=0 and 的特殊情况外，其余情况都在变化的地方新增一个节点，并删除内部的边new_statusstatus
```

这条边检查完后 j 的循环停止，最后还要检查终点：

```
if self.include(tail.coordinate, direction_list) and status:
    other = tail.del_neighbor((i + 2) % 4)
    if tail.neighbor_num() == 1:
        tail.find_neighbor(i, self)
    if other.neighbor_num() == 1:
        other.find_neighbor((i + 2) % 4 if other == startVertex else None, self)
# 检查终点相连的边是否在内部，若是则删除，并且让终点重新寻找邻节点

# 然后进行递归：
if not tail.color:
    self.dfsvisit(tail)
# 如果是白色继续遍历

# 的循环到此结束，最后将这个点设为黑色，并检查是否需要删除或擦除。i
if not startVertex.neighbor_num():
    self.del_vertex(startVertex)
else:
    startVertex.erase(self)
    startVertex.color = 2
# 如果这个节点没有邻节点则删除，否则先然后将颜色设为黑色erase
```

eaten：对方纸带进入自己领地的时候调用，将对方纸带碰到的部分看作被对方吃掉，然后重新修改自己的边界，首先将对方纸带的方向看作前方，然后在前、左、右、左前、右前五个点插入节点（若这些点是自己领地），最后删除多余的节点。

首先在对方当前位置加一个节点：

```
point = trace.current.coordinate
if point not in map_dict:
    self.insert_vertex(Vertex(point), (trace.direction + 1) % 4)
# 首先将对方纸带头目前的位置设为一个节点，这个节点只是用来帮助插入其他节点的，最后会删除

# 创建一个新的列表用来放点：
vlist = []  # 用来存放加入的节点vlist然后依次检查前、左、右，首先找相邻的点：
```

```
for i in [trace.direction, (trace.direction - 1) % 4, (trace.direction + 1) % 4]:
    # 以对方纸带的方向为前方，依次寻找前、左、右三个方向。
    new_point = add(point, directions[i])
    vlist.append(new_point)
    # 首先找相邻的点
```

如果发现这个点属于自己就插入一个节点：

```
    if self.belong_to_self(new_point):
        if new_point not in map_dict:
            if self.belong_to_self(add(new_point, directions[i])):
                self.put_side(map_dict[point], Vertex(new_point), i)
            else:
                self.insert_vertex(Vertex(new_point), (i + 1) % 4)
        map_dict[new_point].del_neighbor((i + 2) % 4)
```

如果是左或者右还需考虑左前方、右前方的点，用 another_point 表示这些点，如果属于自己则加入 border：

```
    if i != trace.direction:
        another_point = add(new_point, directions[trace.direction])
        if self.belong_to_self(another_point):
            if another_point in map_dict:
                if self.belong_to_self(vlist[0]) and vlist[0] in map_dict:
                    map_dict[vlist[0]].add_neighbor(map_dict[another_point], i)
                if self.belong_to_self(vlist[-1]) and vlist[-1] in map_dict:
                    map_dict[vlist[-1]].add_neighbor(map_dict[another_point], trace.
                                                                    direction)
            # 如果在中则将左或右的节点和前方的节点与此节点连接起来another_pointmap_dict

            else:
                new = Vertex(another_point)
                if self.belong_to_self(vlist[0]) and vlist[0] in map_dict:
                    self.put_side(map_dict[vlist[0]], new, i)
                if self.belong_to_self(vlist[-1]) and vlist[-1] in map_dict:
                    self.put_side(map_dict[vlist[-1]], new, trace.direction)
            # 否则插入一个节点然后将左或右的节点和前方的节点与此节点连接起来

# 并且加入：vlist
        vlist.append(another_point)
        # 如果是自己的领地则加入another_pointvlist

# 最后删除多余的点：
```

```
for each in vlist:
    if each in map_dict:
        map_dict[each].erase(self)

# 清除点: point
self.vertlist.remove(map_dict[point])
map_dict[trace.current.coordinate] = trace.current
# 清除掉点point
```

判断攻防的各个零件函数：转换函数 transpoint，transtrace，transborder 将记录时复杂的类简化，转换为较为简单的类便于后续处理，传入记录时的各种类，返回简化类

计算最短距离函数 dis，mindis1，mindis2，计算一个点到各种类型的距离，以点到线段的距离为基础，计算点到线段两个端点的距离，并判断点到线段的垂足是否在线段上，比较得到最小距离与目标点，trace 和 border 通过遍历其中的每一条线段比较得到最小值，此函数为后续攻防判断提供数据，函数传入一个点与一条 line 或 border 或 trace，返回最短距离与路径。

Linecrossline，判断两条线断是否相交，判断轨迹是否相交的基础，两个线段的 x，y 坐标差的乘积，均小于零则相交，传入两条线段，返回 True 或 False

cross 判断轨迹是否相交，检测路线与已有轨迹是否相交的基础，传入两个轨迹，逐一判断其中两条线段是否相交，返回 True 或 False

transtrace1，由点表示的轨迹转化为由 line 表示，方法是判断新的点与栈顶的点的连线方向是否与之前线段方向一致，如一致说明线段未到头，将新的点入栈，如不一致说明轨迹出现转折，将栈顶的点弹出并储存，将栈清空后将新的点入栈。依此直到遍历所有的点后，将储存的点依次连接为 line，转换完成。传入 list 储存的点列，返回一个 trace 类

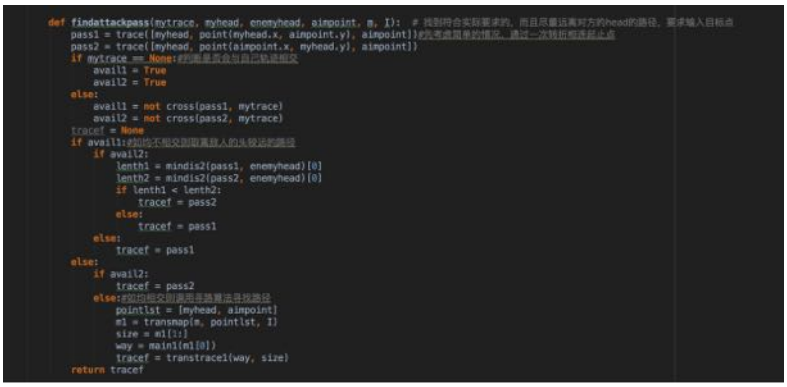
transmap 函数，将大地图分割成寻路所需的小地图并标记起止点，传入起止点，大地图，由起止点坐标向外扩大 5 格或碰到边界，切割地图并改变记号，记录修改后的起止点，返回切割后地图与最小点在原地图上的 x，y 坐标

dealCurrentNeibors，修改后的广度优先搜索，计算每一个未探索点周围的未发现点，并将未发现点加入队列，将障碍物视为已发现点，直到队列为空或找到了目标点时结束，同时记录每个点被发现时的方向，便于之后的回溯

printpath，由每个点的被发现时的方向进行回溯，直到找到起点或者已探索点中没有目标点，将经过的点计入 list 中，返回 list

main1 寻路算法的封装，调用 dealCurrentNeibors 与 printpath，输入为一个二维数组表示的图，利用广度优先搜索与回溯，得到一个用点表示的轨迹并返回，后续可调用 transtrace1 转换为简化的 trace 类

findattackpass 对以上函数的封装，输入攻方路径，被攻击方路径，二者的头，想攻击的点与大地图，首先构建攻防的头到被攻击点的最简路径，即一个直角，判断这两条路径是否与已有路径相交，如均不相交取距被攻击方头远的一条，如均相交则调用 transmap 分割地图后传入 main1 寻找路径，再调用 transtrace1 转化为标准路径，如均找不到路径，则返回 None

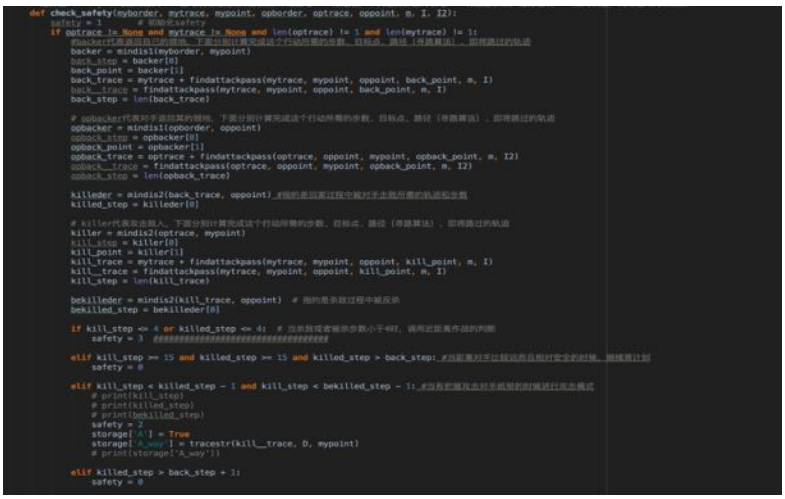


```
def findattackpass(mytrace, myhead, enemyhead, aimpoint, m, I):  # 找到符合实际要求的，而且尽量远离对方的head的路径，要求输入目标点
    pass1 = trace([myhead, point(myhead.x, aimpoint.y), aimpoint])#先考虑简单的情况，通过一次转折即连接止点
    pass2 = trace([myhead, point(aimpoint.x, myhead.y), aimpoint])
    if mytrace == None:#判断是否会与自己轨迹相交
        avail1 = True
        avail2 = True
    else:
        avail1 = not cross(pass1, mytrace)
        avail2 = not cross(pass2, mytrace)
    tracef = None
    if avail1:#如均不相交则取离敌人的头较远的路径
        if avail2:
            lenth1 = mindis2(pass1, enemyhead)[0]
            lenth2 = mindis2(pass2, enemyhead)[0]
            if lenth1 < lenth2:
                tracef = pass2
            else:
                tracef = pass1
        else:
            tracef = pass1
    else:
        if avail2:
            tracef = pass2
        else:#若均相交则调用寻路算法寻找路径
            pointlst = [myhead, aimpoint]
            m1 = transmap(m, pointlst, I)
            size = m1[1:]
            way = main1(m1[0])
            tracef = transtrace1(way, size)
    return tracef
```

围地函数：

def draw0(a1, a2, a3, a4, count, zhen): 围四条边的矩形，ai（i=1,2,3,4）表示依次围的四边长度，可以随时调整，count 表示在围矩形时已经走了 count 步，现在要去走第 count+1 步，zhen 为 0 表示顺时针，1 表示逆时针，返回值是一个指令。

def draw(a1, a2, a3, count, zhen): 围三条边的矩形（有一条边已经存在，依附于纸片边界上）ai（i=1,2,3）表示依次围的边长，当前处在第 i 条边时，转入的边长 ai 可以增大，count 表示在围矩形时已经走了 count 步，zhen 为 0 表示顺时针，1 表示逆时针，返回一个指令。



check_safety 用于判断目前状态是否安全的函数，其需要传入的参数应该包括:myborder, mytrace, mypoint, opborder, optrace, oppoint, m, I, I2，返回值 safety 和对应的含义：0-暂时没有危险继续按照原计划围地；1-有危险，尽快返回自己的领地；2-有把握杀敌，进行进攻；3-距离很近，调用近战判断函数。

此外还有 judge 函数：如果双方都离自己的领地很远，考虑双方横坐标差的绝对值与纵坐标差的绝对值之和，若为偶数则后手方必胜，否则先手方必胜，用这条规律可以判断双方距离较近的时候应该如何行动。

```
def judge():
    if storage['in'][stat['now']['enemy']['id'] - 1]:
        return None
    # 如果对方在领地内则停止进攻

    delta_x = abs(stat['now']['me']['x'] - stat['now']['enemy']['x'])
    delta_y = abs(stat['now']['me']['y'] - stat['now']['enemy']['y'])
    # 计算双方横纵坐标之差的绝对值

    if (delta_x + delta_y) % 2:
    # 如果绝对值之和为奇数则选择进攻
        if D == 0:
            if delta_x >= delta_y:
                return ' '
            else:
                if stat['now']['me']['y'] < stat['now']['enemy']['y']:
                    return 'r'
                elif stat['now']['me']['y'] > stat['now']['enemy']['y']:
```

```
                return 'l'
            else:
                return ' '
        # 总体思路是尽量让和的差变小，因此在delta_xdelta_ydelta_x < delta_y
        # 的情况下回改变方向，下面三种情况也是同理
        elif D == 1:
            if delta_x <= delta_y:
                return ' '
            else:
                if stat['now']['me']['x'] < stat['now']['enemy']['x']:
                    return 'l'
                elif stat['now']['me']['x'] > stat['now']['enemy']['x']:
                    return 'r'
                else:
                    return ' '
        elif D == 2:
            if delta_x >= delta_y:
                return ' '
            else:
                if stat['now']['me']['y'] < stat['now']['enemy']['y']:
                    return 'l'
                elif stat['now']['me']['y'] > stat['now']['enemy']['y']:
                    return 'r'
                else:
                    return ' '
        elif D == 3:
            if delta_x <= delta_y:
                return ' '
            else:
                if stat['now']['me']['x'] < stat['now']['enemy']['x']:
                    return 'r'
                elif stat['now']['me']['x'] > stat['now']['enemy']['x']:
                    return 'l'
                else:
                    return ' '

    else:
        return None
    # 否则放弃进攻
```

2.3 程序限制

Trace 类有一个局限性，即如果有一方纸带连接的领地被吃掉，在对方领地内游走，最后回到自己领地时，会发生报错。但自己遇到此种情况基本必败，无需考虑，对方遇到这种情况则调用攻击函数，争取做到一击必杀。

Border 类也有局限性，Border 如果被对方纸带覆盖一部分，则这部分看作是属于对方的，如果一条狭长的领地被对方的纸带穿过，则 Border 类会认为这片领地已经被切断。因此如果自己要围一块较大的领地，其中有一条边是自己狭长的领地，另外三条边是自己的纸带，如果围成的时候狭长的领地被对方的纸带切割，Border 类会认为这块领地没有围成，但实际上这片领地已经围成。

目前未发现寻路算法出错。在类之间进行转译的函数 transmap 在起始点与终点是同一点时会将其分成两个点导致传入地图出错。

3 实验结果

3.1 测试数据

实验环境说明：

- 硬件配置：CPU:INTEL 酷睿 I5-5200U 内存：4 GB
- 操作系统：Window10
- Python 版本：Python 3.6 64-bit

测试方法：

1. 采用人机对战方法，人为制造一些特殊局面来测试 AI 是否能及时地进行进攻和防守

2.AI 和自己对打，测试是否会超时

3. 让 AI 和我们编写的 02.py 对打，02.py 的围地策略和我们的 AI 是相同的，但它不判断攻防状态。

测试结果：

我们根据一些情况下 AI 反应的灵敏程度，对函数中的一些参数进行了调整，让 AI 能够更及时地做出战略变化。

比如，在判断攻防状态的函数中，为了保证进攻能够必杀对方，我们在比较自己进攻步数和对方回家步数时加入了调整因子，两个步数之差大于这个因子时才进入进攻状态，这个因子调小时 AI 进攻就越灵敏，但风险也更大，经过试验后我们把这个因子调为 5。

参加第二次热身赛：

当时还没有完善的 Golf 积 11 分排在 f17 组第 14 名。

3.2 结果分析

围地算法执行达到预期效果，每相邻矩形间有一条对齐重合，攻防状态的判断算法可以帮助围地算法进行相应的调整，但在自己的纸卷落入对方纸片时逃脱效果欠佳。

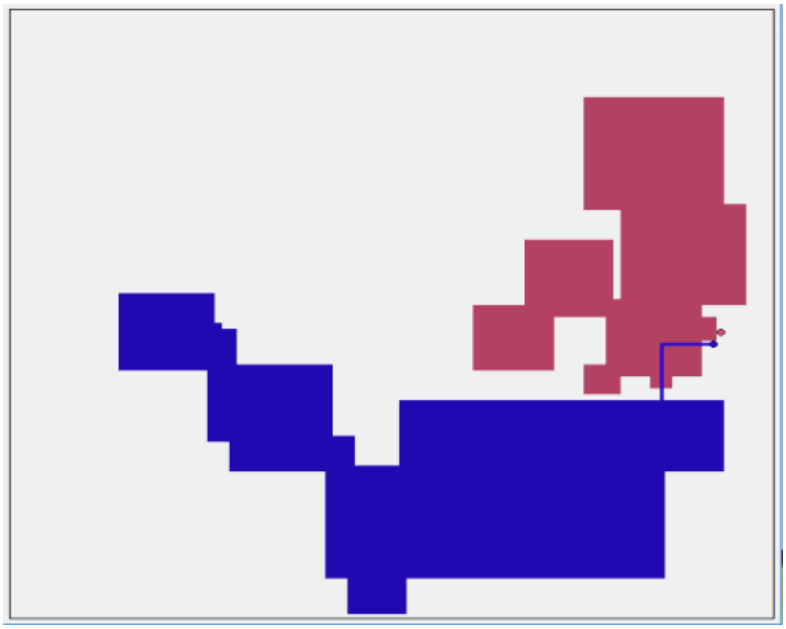
在对战结果中，我们发现，若围地时遇到危险立刻返回时，确实能很好地脱离危险，但是这样会让 AI 围出很细长的矩形，这样围地亏损很大，这是我们围地算法应对变化调整的不足之处，

模式一中算法的运行时间较长，虽然单纯围地算法是许多判断条件的叠加，不会占用太长时间，符合预期效果，但主要的耗时函数——依据二维数组地图求出 trace 和 border 类的函数以及使用 A* 算法寻找进攻线路的函数 findattackpass（）搜索过程很长，有时在回合数将近走满时会出现决策时间耗尽，于是我们对轨迹和边界的转译函数进行了一些限制。模式二的耗时很短，主要也是运用各种分支结构，（时间复杂度为 O（1）但在 storage 中开辟了很多新变量），当对方不能够多次触及地图某个边界时，我方攻击效果良好。

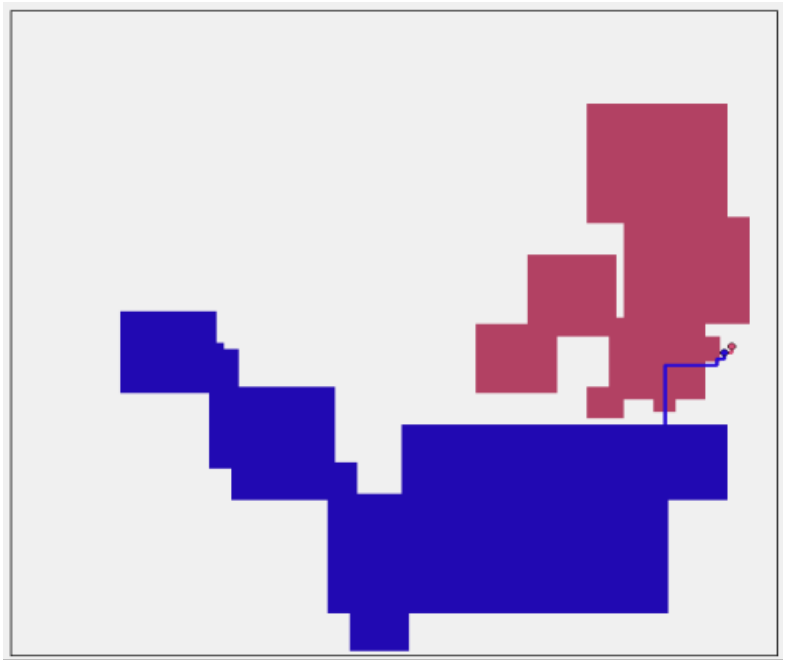
3.3 经典战局

（1）先手蓝色为我们的 AI，后手粉色为围地函数 02.py，这次战局中 AI 成功化险为夷，当自己的纸带陷入对方纸片领地时，此时由于双方距离很近，AI 调用近身作战函数 judge()，反杀敌方。

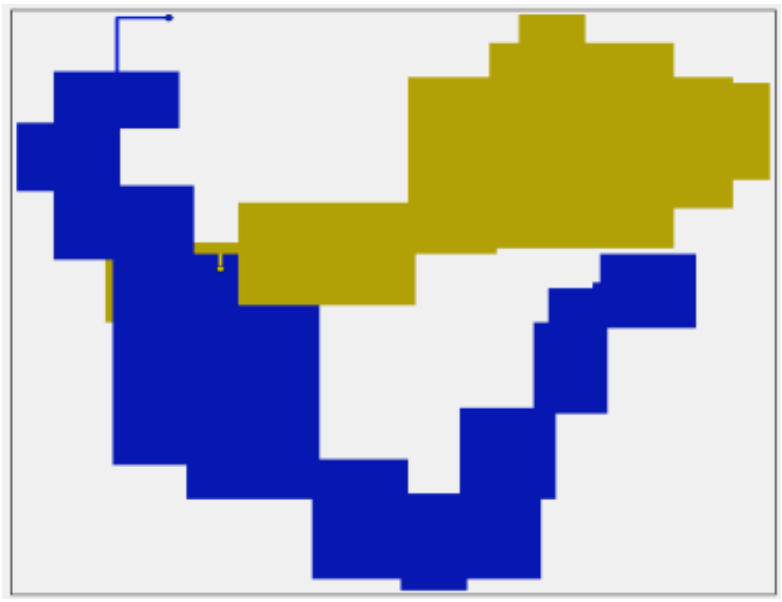
AI（蓝色）处于较危险的局面，但双方纸卷距离很近，有机会决斗反杀对方：



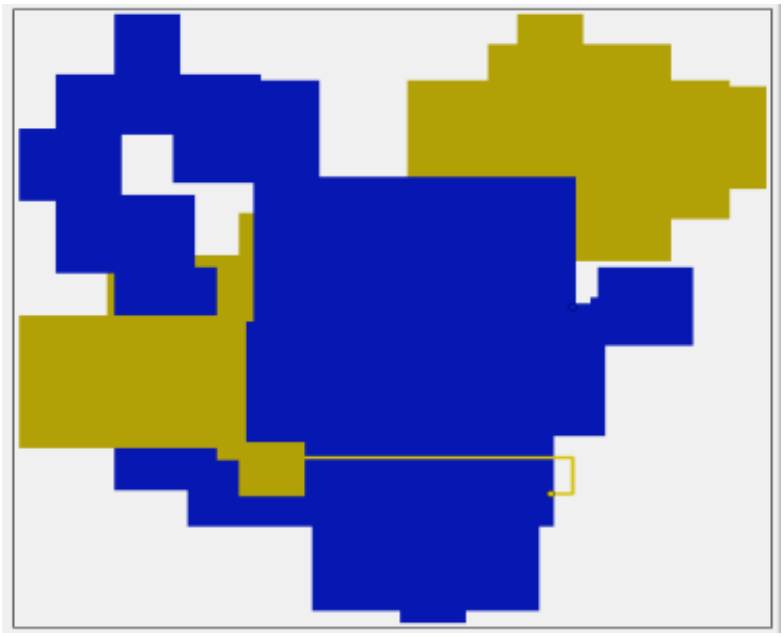
成功反杀：



（2）后手蓝色为我们的 AI，先手褐色为 02.py，我们发现，在逐个围矩形到达边界反弹时，如果反弹效果理想，则可以形成一个倾斜的包围圈，如下图，当 AI 在上边界围矩形发生反弹后，可以预见，它将借助之前在下边界和左边界的反弹路线，将 02.py 包围：



包围成功后，直接使得 02.py 的大部分纸片被吞掉：



因此，这种类似台球反弹的围地线路有机会达到类似于“包饺子”的效果，本身就带有潜在的攻击性.

4 实习过程总结

4.1 分工与合作

小组一共五人，杨昌赫负责地图类的编写，郑煜衡负责主程序和基础设施的编写，甘天奕负责各类函数的具体实现，何为负责整体进度的分配和部分基础设施的编写，谢思远负责部分基础设施的编写与部分函数的实现。

小组最开始在二教进行程序最初的整体思路和性能设计的讨论，五个人围坐在一起讨论着自己对这个问题的看法与见解，交流自己的思路和好的想法，尽量将整个问题抽象化，确定整个圈地策略是向前的，而且是以阶梯型向前圈地，同时确定了构造一个线段类来实现整个地图信息的存取与修改，同时考虑两个大的函数，分别为进攻函数和防御函数，具体功能为判断自己是否有进攻机会以及判断如何进攻，和判断自己是否危险以及如何保证自己的行动安全。组长何为将整个问题拆分成一些具体的部分，将各部分分别分配给各个组员进行编写。

第二周组会，最基本的地图类和一些必须的基础设施已经构造完毕，而攻防函数部分仍有着比较大的问题，攻防函数如何与具体的圈地策略进行结合成为首要问题，以及如何实现攻防函数的具体功能也成为一个拦路虎。在组会里面，各个组员依次发表自己对攻防函数的想法，并互相对对方的观点提出质疑和探讨，其中甘天奕同学，对函数功能和具体实现的想法得到了组员的认可。同时，地图类假如领地被切成几块，该如何表示？杨昌赫同学提出的这个问题也引发了组员的思考和讨论，基本确定了地图类接下来的修改方向和具体函数的实现方式。此次组会中，组长友情提供了饮料作为组员的福利，让组员们更团结一心，努力编程。

在第三周，经过甘天奕同学的查阅资料，攻防函数的具体实现方式已经被确定下来，基于 a 星算法对全地图距离信息的计算，让攻防函数的具体概念被确定下来，而地图类也进行了相应的修改，具体的主程序的基础设施也建设完毕，圈地策略上面却遇到了一定的问题。一是，有一种将边界圈起来从而将整个地图都圈起来的策略引起我们的注意，我们称之为包饺子策略，如何确保我们的程序能够打败包饺子策略引发了组员们的讨论；同时程序在向前圈地行进到无法继续前进的时候，我们的程序该如何应对？对于第一个问题，谢思远同学提出只要让我们程序一直向对方边界尽可能地行进即可应对。但同时第二个问题就显得更为重要了，对此组员们产生了不同的想法，有人认为该掉头回去继续圈地，但甘天奕同学认为应该贴着已有的边界继续圈地，直到要撞墙了才返回，经过激烈的探讨，

基于圈地的有效性和实现难度考量，最终采取了甘天奕同学的建议。同时，组长何为同学负责编写出一个包饺子策略的程序，以便考察我们的程序是否真的能打败这种策略。

在第四周，我们进入了紧张的调试阶段，各个部分的任务已经基本完成，开始各个细节方面的完善，诸如，如何防止撞墙，如何防止自杀行为，这些问题作为函数的功能来进行解决，同时也将各个部分开始整合，开始调试运行整个程序，不断调整代码以让他能够正常运行，这个过程是整个代码编写过程中最困难，也最枯燥的阶段，组员们不得不熬夜翻看代码，以修改代码中的 bug 使代码能够顺利运行而不报错。在这个枯燥烦恼的过程中，组长自掏腰包为我们点夜宵的行为为组员们起到了慰藉的作用，激励组员们继续与 bug 奋战下去。整个整合过程中出现了各种各样的问题，各种接口对不上和功能的错误使用屡见不鲜，经过组员们的熬夜奋战，大部分的 bug 都被消除，还有小部分的 bug 因为时间关系不得不放弃部分功能来解决。虽然存在着瑕疵，但程序总体上完成了，能够顺利运行且不会报错

4.2 经验与教训

首先，在编写代码的过程中要多思考，不要在没有考虑清楚一个问题的前提下便盲目的去进行工作。我们刚开始考虑的方案是优先向后圈地，并且编写了相应的代码，然而在后面进一步的讨论与思考的时候我们发现这并不是一个很好的解，所以之前一周的努力付诸东流，再重新对整个圈地策略进行审视后，才确定之后的整个程序运行的基调。

其次，在编写代码的过程要善于查找资料，不要总是闭门造车。在攻防函数的编写过程中，我们开始对攻防函数的一些具体设想因为各种原因一直没有具体的实现方法，而甘天奕同学在查找了相关资料之后，发现一种前人已经设计并且完善好了的算法，a 星算法，直接就解决了我们遇到的难题，而且 a 星算法极为完备，更方便了我们之后的程序的编写。

最后一点教训，要将任务分配更加明晰以及各个部分接口和功能一定要实现统一。在我们编写程序的过程中，一度出现过这个问题应该谁来解决的矛盾，互相以为是对方的任务，导致谁都没写，同时接口也没统一，导致后面进行整个程序的组装时，修改量几何倍数增长，而且很多功能不得不放弃，所以一定要在全局的统筹规划上面下好功夫。

4.3 建议与设想

本次实习作业感觉难度颇大，虽为竞赛制，但整个程序的设计的启发式规则并不好抽象化为函数语言，需要付出大量的时间进行思考和设计也不一定能取得期望的效果，而刚好大作业在期末季前不久才开始布置，刚好与许多其他的论文，考试时间相重叠，很大程度上没有充足的时间进行准备，简易以后大作业可以更加提前一点布置，给同学们更多时间进行准备和完善，同时大作业难度也要进行合理考虑。同时建议可以区分一下，直接交了参考程序代码的同学和至少自己认真写了代码的同学。以及建议学弟学妹们要多查阅资料，尽早开始代码的编写过程，不然时间紧迫就只能熬夜赶代码了。

5 致谢

本次实习作业在组长的带领下，全体组员共同积极参与，利用老师上课讲解的知识和查找的资料完成了本次实习作业。在这次大作业里面我们收获颇丰，不仅对老师上课讲授的知识进行了综合地运用，同时查找课外资料更进一步提升了我们对课程内容的了解。

首先感谢陈斌老师和助教们，是他们让我们有了这次难忘的实习作业的经验。同时也是陈老师上课传授的知识让我们有了编写程序的可能，而在整个编写程序的过程中，也是陈老师和助教们为我们答疑解惑，整个过程里面陈老师没有任何的抱怨，经常深夜了都还在回答同学们的疑问，感谢陈老师和助教们的不辞辛劳，也确实让我们获益良多。

其次要感谢技术组的同学们，是他们为这次实习作业搭建起了平台，给我们展示自己所学的成果的机会，技术组的同学们时间更为紧迫，要尽早提供一个完善的平台，同时还要回答同学们的各种困惑，整个过程里面也没有抱怨，只是在默默付出，为我们的本次实习作业默默的付出了许多，很感谢他们。

最后还要感谢一下我们组长和组员们，组长不仅负责我们全体的统筹规划，还自掏腰包负责发放组员福利，没有半点不满，实乃组长楷模，同时组员们也戮力同心，与子同袍，共同认真努力的完成了本次实习作业，没有组长的无私付出，没有组员们的团结配合，这次实习作业是决不可能完成的，很感谢我们是一个如此团结一心的团体。

再次感谢陈斌老师，助教们，技术组同学们，以及全体组员的团结一心。

6 参考文献

1. http://www.gamedev.net/reference/articles/article2003.asp

（A 星算法的参考资料）

2. 《图邻接矩阵的行列式的若干结果》黄玲玲硕士学位论文，2013.6.7

（为了写 Trace 和 Border 类，我们学习了一些图论的知识，这对于我们在处理邻节点的方法上有很大的启发）链接：

http://xueshu.baidu.com/s?wd=paperuri%3A%282a5ca6b91b3f0cdf1252eeabbc26f526%29&filter=sc_long_sign&tn=SE_xueshusource_2kduw22v&sc_vurl=http%3A%2F%2Fwww.doc88.com%2Fp-0012453057206.html&ie=utf-8&sc_us=5013919827289903050

第九章　F17_Hotel 报告

肖力哲 * 王睿哲 付天尧 翁纪伦

摘要：编写了两种偏好的 AI 模式，以随机的方式选择，进攻 AI 偏向直接包围对手，防守 AI 偏向保护自身，尽可能多地圈地。两种 AI 与 normal_wanderer 的对战胜率均较高，在正式比赛中全败出局。

关键字：队列、字典

1 算法思想

1.1 总体思路

在初始步数限制（50 步）的条件下，并无太多步数进行战术博弈，应以画地优先，最佳方式是画正方形，在确保自身安全的条件下（回避敌方纸带）尽可能占敌方地盘（围敌方地的效率高于自己围地），暂不考虑击杀敌方纸带的情况。

第二次热身赛得知步数限制提升（2000 步），有充分的步数进行攻防策略，那么在圈地的同时，也可以考虑击杀敌方纸带，本 AI 试图通过围杀对手实现进攻：

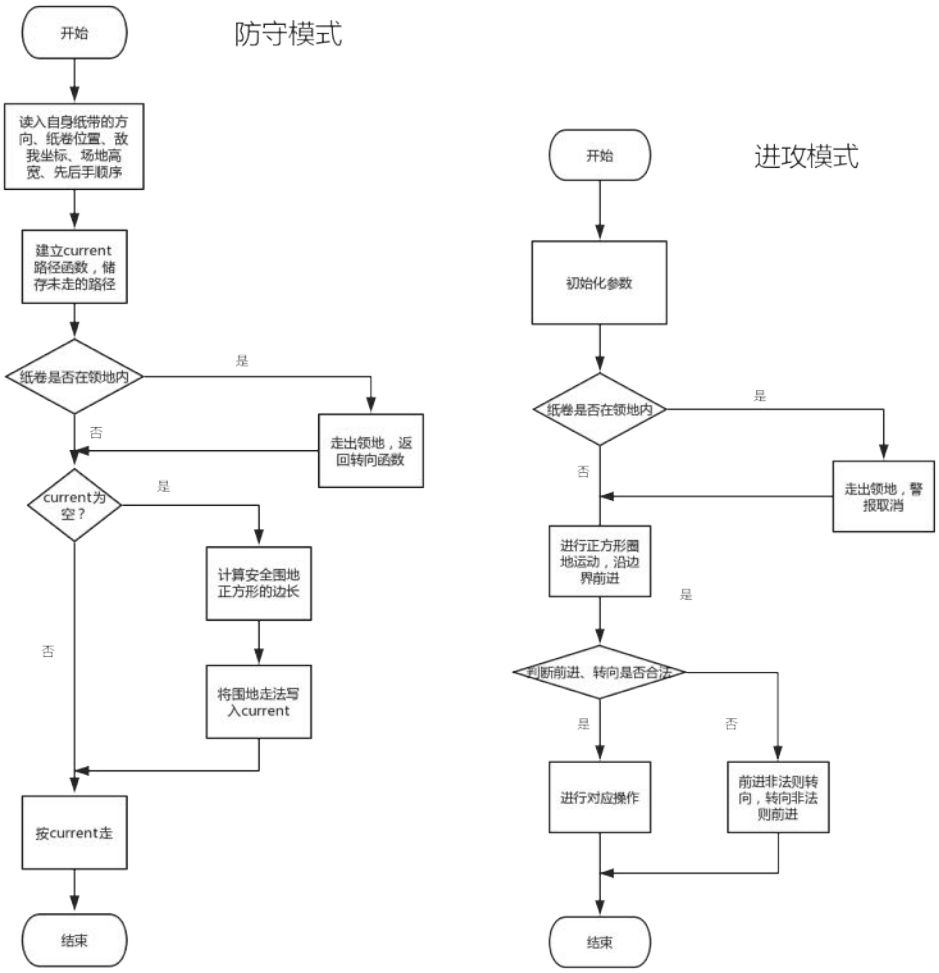
1. 计算敌方能否击毙己方: 基于自己和对方的领地、纸带的数值，考虑双方击毙对方的情况。如敌方纸卷到己方纸带的距离、敌方纸卷回敌方领地的距离（敌方纸卷在敌方领地内会击毙己方纸卷）、己方纸卷回己方地盘的距离（敌人可能回头来击杀己方）

2. 考虑如何围更大的地盘: 最经济的围法是少拐弯，最理想情况是拐两个弯就回己方领地，实际过程中要躲避敌方，路径不能预测

在编写过程中，由于某些不可抗拒因素，我们无法合理在进攻和防守中切换，于是将

两部分分开，做成了两个模式。

1.2 算法流程图



1.3 算法复杂度

初期算法运行时间主要用于遍历，后期尝试部分遍历数据以及只在圈地开始的时候进行遍历，但由于这种处理方式会导致易被对方击杀，放弃了这种算法。改变后运行时间主要花费在判断上。时间复杂度为 O(1)。在实际运行中的时间开销很少（<0.1s）。

2 程序代码说明

2.1 数据结构说明

防守模式的 current 函数运用了队列。此队列可以更多的判断场上情况，进行动态修改; 进攻模式沿用了 storage 和 stat 两个字典。

2.2 函数说明

（1）storage 是用于储存算法中要用到的自定义数据的字典。如 current 记录走法

（2）stat 是系统自定义的包含游戏开始以来的相关信息的字典

防守模式

（1）dis 求两点距离的直角距离（x 坐标差和 y 坐标差的绝对值的和）

（2）tellLorR 用于判定 b 在 a 左手边还是右手边，返回 a 迎着 b 时的转向 0，1，2，3 以及走法‘l’或‘r’

（3）isin 判断一个点是否在场地内

（4）disbound 判断一个点沿某个方向走到边的距离

进攻模式

（1）disPoints 求两点距离的直角距离

（2）DisNone 计算纸卷到边界的距离

（3）tryturn 转向函数，判断转向是否会撞到边界，如果是则返回 False

（4）tryforward 前进一步是否合法

（5）tryband 判断前方是否是自己的纸带，如果是则报警

2.3 程序限制

本 AI 的围地函数中有参数 l 决定围地半径，与 normal_wanderer 对战中围地半径取大值，以围地取胜，但是在实际对战中，由于对手进攻过于猛烈，围地半径即便设小值，

也被对手顺利击毙，毫无还手之力。

3 实验结果

3.1 测试数据

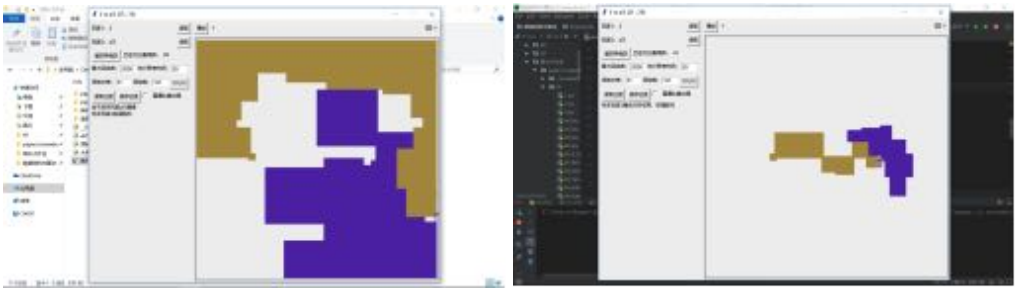
实验环境说明：

- 硬件配置：联想 80WB 笔记本电脑
- 处理器：Intel Core i5-7300HQ @ 2.50GHz 四核
- 内存：8GB 三星 DDR4 2400MHz
- 操作系统：windows10
- Python 版本：3.6

使用与 normal_wanderer 对战的方式测试算法。胜率维持在 70% 左右。

实验次数	比分
1	15-5
2	13-7
3	10-10
4	16-4
5	12-8
6	13-7
7	14-6
8	13-7
9	12-8

进攻和防守方式内部斗争



3 是进攻模式，ai5 是防守模式，两次比分分别为 21:20 和 55:35。

双方攻守不频繁，决定成败的主要因素是围地半径的取值。

3.2 结果分析

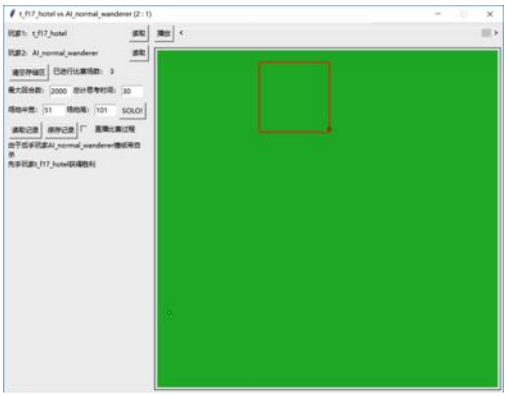
在对阵 normal_wanderer 的过程中，本算法的围地策略成效显著，normal_wanderer 围地上劣于本 AI。

在内战中互相暴露了 bug：防守模式当自身与敌人处在同一个边界时，容易撞墙。进攻模式在周边滚轮式前进时全部占据外圈后，敌方围去一块地盘，很大概率撞己方纸带死。

在实际对战中，保护机制并未起到应有的效果，防守模式经常被敌方击杀，进攻模式在与敌方缠斗时落于下风，被抢先击杀。

3.3 经典战局

进攻模式将 normal_wanderer 围杀



4 实习过程总结

4.1 分工与合作

肖力哲负责实验报告，王睿哲负责源代码的阅读和接口环境的技术处理，翁纪伦、付天尧负责进攻性、防守型 AI 的编写。

第一次组会确立了基本思想和具体分工，第二次组会时基本框架已经搭好，就实现攻防战略进行了讨论，第三次组会测试了攻防代码，进行修改和整合。由于在同一个寝室，交流十分方便，在编写时有问题马上沟通，一起解决。

4.2 经验与教训

交流十分频繁，大家编写的代码都通读过，讨论都能切中要点。一开始的步数是 50 步，在 50 步的前提下编写的算法以圈地为先，攻防策略考虑较少，之后增加到 2000 步时未及时接收到信息，没有更新算法，在 2000 步的前提下编写时间不够，有很多地方没有考虑周全。保护机制做的不是很成功，很大一部分失败的原因是纸带被撞死

4.3 建议与设想

希望接口可以更人性化一点，操作不太方便，我组在连接接口时花费了大量时间。

这是一门很有趣的课程，可以掌握 python 的基础用法，进行一些实用性较强的算法编写。最好在掌握一定数据结构与算法基础后再选此课程，可保证自己得分较高，这门课大佬太多。

希望在征得优胜者同意后可以分享优胜者的代码，让大家参考学习，获得更多进步空间。不然只是单纯被虐，没有特别大的提升。本次纸带圈地可以加入更多元素，如加速功能：在一定回合数内纸带速度加快；无敌功能：每次对局可使纸带无敌若干回合，限制一次

5 致谢

赵宇、冀锐助教处理了我关于提交代码的问题，十分感谢！

感谢油库里开发的接口环境，锻炼了阅读复杂源代码的能力，十分感谢！

6 参考文献

http://gis4g.pku.edu.cn/course/pythonds

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

《Python 基础教程》第二版修订版，Magnus Lie Hetland 著，司维曾军崴谭颖华译，钟读杭审校，人民邮电出版社

《零基础 Python 入门学习》，小甲鱼编著，清华大学出版社

第十章　F17_India 报告

黄荣 *、侯华丽、张君曼、祝佳琪、魏论研

摘要：我组代码分为三个阶段，赛前阶段、热身赛阶段、最终比赛阶段；主要是去根据游戏规则去寻找策略以及针对敌方策略的应对方法，总的来说是经验性质的战术。在多次热身赛后，我们不断增删一些处理方法，最终形成一个根据自己与对手和领地距离来决策的算法。最终，我组取得 17 级八强的成绩。

关键字：集合列表队列广度优先搜索计算几何动态规划

1 算法思想

1.1 总体思路

我们算法的总体思路是以保守的圈地策略为主，激进的进攻策略为辅。这种思路让我们遇到较强对手时被杀的几率减小，但会导致我们在耗尽步数之后圈地面积总是比对方的小。

前期的两个主要策略：圈地策略和进攻策略。

我们的圈地策略是简化圈地问题为圈一个矩形，枚举矩形的上下左右坐标，计算产生的矩形与自己的基地的交线，算出新产生部分的面积，对一个矩形给出一个估价，估价考虑以下因素：圈地面积（自己增加面积 + 对方减少面积），圈地所需步数，矩形离基地最长距离（便于快速回城），矩形离敌方距离等。我们选取一个估价较高的矩形进行圈地——一个比较保守的圈地策略。

我们的进攻策略主要有两个：保守策略和激进策略。首先二分进攻所需时间，这样可

以得到敌方攻击范围和己方攻击范围（两个菱形），计算己方攻击范围内是否有敌方纸带，敌方攻击范围内是否有己方纸带，己方攻击范围内是否有己方基地，敌方攻击范围内是否有敌方基地等。然后再采用这两个策略，保守策略：假设敌方会以最短路线攻击自己，敌方攻击范围内没有己方纸带，己方攻击范围内有敌方纸带，则发起攻击。激进策略：假设敌方会以最短路线回城，敌方攻击范围内没有敌方基地，己方攻击范围内有敌方纸带，则发起攻击。

在具体实施时，我们发现枚举会花费大量时间，于是不断改进算法。

改进得到的 t4 代码的思路如下。圈地策略：圈地前先将基地扩大 30 单位，再缩小 25 单位来抹平凹陷点，在场地边界选取一点作为边界起始点，圈地第一阶段沿距离基地最远方向到指定距离，然后按顺时针或逆时针同一方向进行圈地，通过计算几何计算叉积来保证沿同一方向进行圈地，圈过半圈后回城，结束圈地，用向量余弦 <-0.95 来判断圈过半圈；进攻策略：如果自己到敌方纸带距离小于敌方到己方纸带距离并且敌方回城距离小于己方进攻距离则直接进攻，如果 (自己到敌方纸带距离加两倍己方回城的距离) 小于敌方回城距离，则先回城再进攻，回城则找不撞纸带的距离最近的方式。

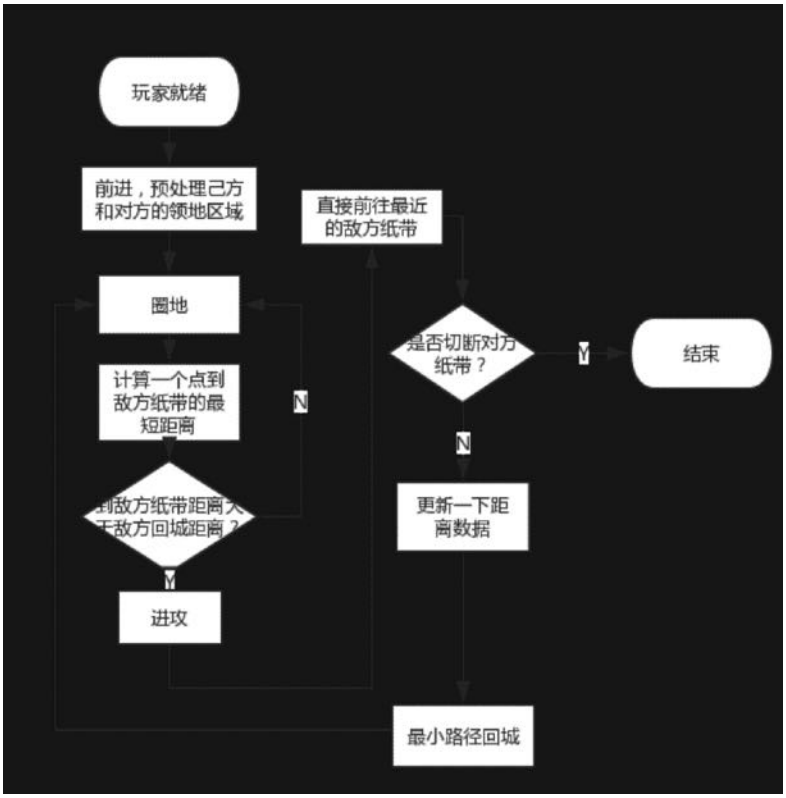
在 t8 中，我们做了如下改进：圈地起点选为在自己基地外、距离自己基地 70 单位的点，防止在凹陷处开始圈地；回城时，在广度优先搜索中考虑不撞纸带因素，防止因纸带干扰低估回程距离。

我们采用的数据结构有列表、队列等。在圈地策略中，我们采用队列、列表实现广度优先搜索的算法，之后采取贪心算法以处理到敌方纸带距离大于敌方回城距离、离对方纸带距离小于对方回城距离和对方攻击自己的距离、对方到达自己纸带距离大于 2* 自己回城距离 + 到对方纸带距离、敌方离己方纸带过近等多种情况。

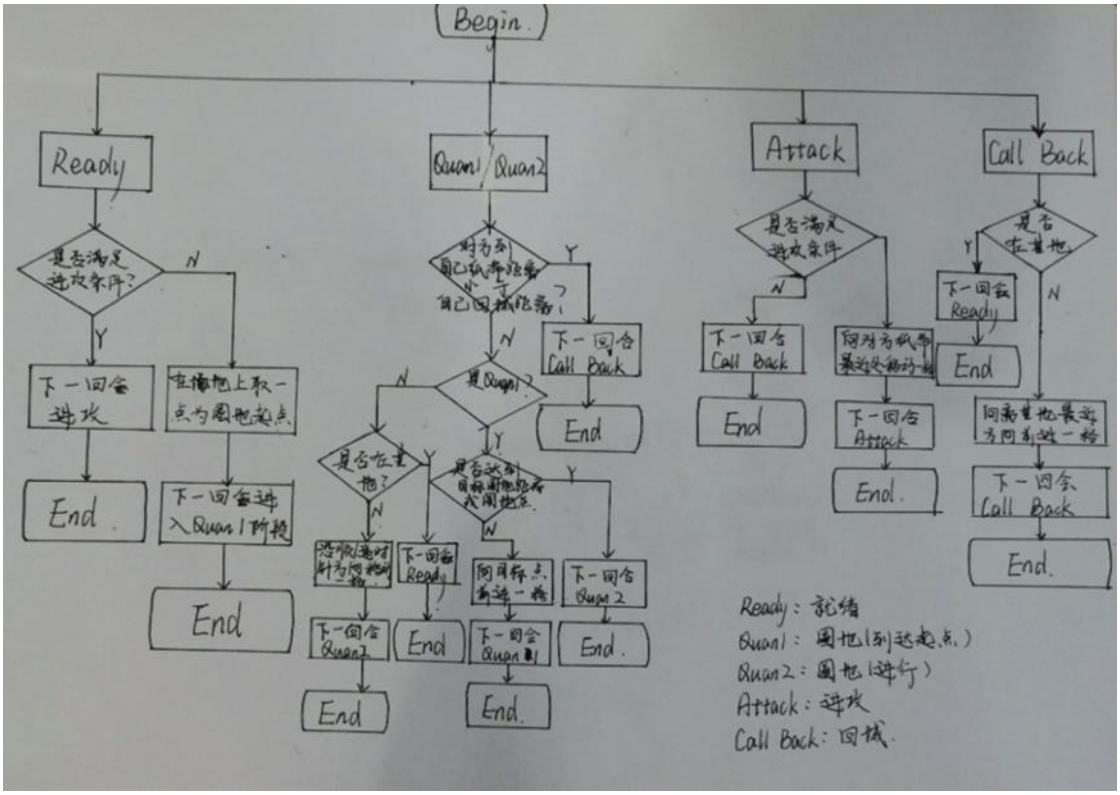
我们最初考虑只圈地、不进攻的算法，只考虑阶梯式圈地这种相对圈地面积较大且较安全的方法；但第二次热身赛后，我们发现这样也会被撞，并且出现对方围上所有边界的情况，和自己撞边界的小 bug。最终，我们添加了进攻算法，并改进了圈地算法。

在最后一次热身赛中，我们位列小组第三，成员们依然在寻找不足。比赛时，纸带出门就被杀，自己圈迷宫，决策时间耗尽三种情况均有出现。解决这些问题后，我们在主动接近对手、主动远离对手两者犹豫。最后，我们认为激进较优，采用 t8 代码。

1.2 算法流程图



最初的想法



t8 代码流程图

1.3 算法复杂度

load 函数中：

dist 函数——O(1)

calc_dist_band——O(n)

calc_dist_border 使用广度有限搜索，计算一个领地的边界以及任一点到领地的最短距离，时间花费较其他函数大，分析为 4*n*m(n、m 分别为长和宽)——O（n2）

play 函数中：

dist，calc_cj，calc_cos——O(1)

move_to 函数使用 for 三个方向的 for 循环——O(1)

min_to_enemy_band 函数使用双层 for 循环，分析为 n*m——O(n2)

综上所述，本算法时间复杂度为 O(n2)。

在实际调用中，对于在圈地 (到达起点)，圈地 (进行)，进攻，回城四个 stage，每一个都进行多次函数调用，实际花费时间依据是否调用 calc_dist_border 函数而定。

所以，预处理搜索部分是运行时间的主要部分。实际运行中，代码调用一次 calc_dist_border 函数大约需要 100ms，其余操作一般可在 10ms 内完成，平均 20 步调用一次 calc_dist_border 函数。

编写函数的时候，考虑到时间有限，在每一次决策的时候，如果时间足够，会多运行一些函数做决策；若时间不足，则终止某些函数。所以在实际运行中，每次的决策不会超时。

2 程序代码说明

2.1 数据结构说明

使用的数据结构类型：

主要是线性数据结构，其中包括：

集合：Python 集合 set 是不重复元素的无序集合。主要用于存储了纸带不同方向矢量方便调用。

列表：Python 列表 list 是任意对象的有序集合，通过索引访问指定元素。在本组的代码中，列表主要用于存储领地的边界点坐标、采用不同策略时触发函数的代号、两点坐标最短距离、处于不同战况下的各项参考因素（位置、距离、剩余时间等），我们也使用了嵌套列表用于存储地图上每一点的领属情况等。

队列：Python 队列是一种有次序的数据集合，特征是新数据项的添加总发生在一端（“尾”端），而现存数据项的移除总发生在另一端（“首”端）。为了计算任一点到领地的最短距离，我们采用了广度优先搜索的队列用于存储坐标。

2.2 函数说明

1、一些准备函数

首先，需要准备一些 loading 时的条件：距离目标位置距离，所在位置距离对手最短进攻距离，前进方向，已有领地及所在位置领属情况，边界位置，剩余时间

（1）计算两点之间距离：

```
def dist(x, y, nx, ny):
    return abs(x - nx) + abs(y - ny)
```

（2）计算某一点到一条直线的最短距离，用于判定能否采取进攻策略的条件之一：

```
def calc_dist_band(pos, pos_list):
    return min([dist(pos[0], pos[1], x[0], x[1]) for x in pos_list])
```

（3）存储边界点坐标函数：利用嵌套列表来存储边界点的坐标，定义四个方向的方向矢量；

```
def calc_dist_border(id, field, band, MAX_DIST):
    direct = ((1, 0), (0, 1), (-1, 0), (0, -1))
    from queue import Queue
    q1 = Queue()
    dist = [list(field[x]) for x in range(stat['size'][0])]
    border= []
```

对整个地图的点进行循环，判断其领属情况。如果是属于自己领地的点，则将距离设置为 0。对一个点四个方向的点的领属情况进行判断，只要其中一个不属于自己的领地，则该点为边界点，坐标存入 dist。

```
    for x in range(stat['size'][0]):
        for y in range(stat['size'][1]):
            if dist[x][y] == id:
                dist[x][y] = 0
                for fa in direct:
                    nx = x + fa[0]
                    ny = y + fa[1]
                    if nx >= 0 and ny >= 0 and nx < stat['size'][0] \
                            and ny < stat['size'][1] and field[nx][ny] != id:
                        dist[x][y] = 1
                        q1.put((x, y))
                        border.append((x, y))
                        break
            else:
                dist[x][y] = MAX
```

在这里我们设置了一个 MAX=1000000 的初始值，对非边界点的点赋值以待后期的更新。

（4）更新最短距离：

我们使用广度优先搜索的算法，利用队列进行一圈圈向外地进行搜索并更新最短距离。

考虑到时间问题，为防止超时，我们在进广度搜索时设置一个 MAX_DIST 的范围，当搜索距离超过 MAX_DIST 时，停止搜索以节约时间。

```
while not q1.empty():
    x, y = q1.get()
    if dist[x][y] >= MAX_DIST:
        continue
    for fa in direct:
        nx = x + fa[0]
        ny = y + fa[1]
        if nx >= 0 and ny >= 0 and nx < stat['size'][0] and \
                ny < stat['size'][1] and band[nx][ny] != id:
            if dist[nx][ny] > dist[x][y] + 1:
                dist[nx][ny] = dist[x][y] + 1
                q1.put((nx, ny))
```

另外，当对手到我方最短距离小于 8，我方回城最短距离小于 8 时，我们认为双方处于掐架局面，此时将最大距离 MAX_DIST 缩小为 10，减少广度搜索的范围来节约时间。

```
x = me['x']
y = me['y']
if dist(enemy['x'], enemy['y'], x, y) < 8 and storage['me_dist'][x][y] < 8:
    MAX_DIST = 10
else:
    MAX_DIST = 100
```

（5）存储不同时刻采取的策略和边界情况

```
storage['calc_dist_border'] = calc_dist_border
storage['calc_dist_band'] = calc_dist_band

storage['stage'] = 0
'''
0:就绪
1:圈地(到达起点)
2:圈地(进行)
3:进攻
4:回城
'''
```

（6）计算当前位置到目标位置两矢量叉乘值和夹角余弦值，用于判定最佳前进方向。

```
def calc_cj(x, y, nx, ny):
    return x * ny - y * nx

def calc_cos(x, y, nx, ny):
    return (x*nx+y*ny)/math.sqrt((x**2+y**2)*(nx**2+ny**2)+1)
```

（7）判定前往目标位置的最佳路径的函数。选择一个与目标位矢量点积大于 0 的方向作为前进方向，并且要注意不能装上自己的纸带。

```
def move_to(x, y, tx, ty):
    faS = direct[me['direction']] + ('S',)
    faL = direct[(me['direction'] + 3) % 4] + ('L',)
    faR = direct[(me['direction'] + 1) % 4] + ('R',)
    ava = []
    for fa in (faS, faL, faR):
        nx = x + fa[0]
        ny = y + fa[1]
        if fa[0] * (tx - x) + fa[1] * (ty - y) >= 0 and nx >= 0 and ny >= 0 \
                and nx < stat['size'][0] and ny < stat['size'][1] \
                and not (nx, ny) in storage['me_band']:
            ava.append(fa[2])
            if fa[0] * (tx - x) + fa[1] * (ty - y) > 0:
                return fa[2]
    return random.choice(ava)
```

（8）计算距离对手最短距离的函数，便于决定是否采取进攻。

```
def min_to_enemy_band(x, y, id):

    re = (10000, 1, 1)
    for nx in range(stat['size'][0]):
        for ny in range(stat['size'][1]):
            if bands[nx][ny] == id:
                re = min(re, (dist(x, y, nx, ny), nx, ny))
    return re
```

（9）计算更新剩余时间和双方领地情况。总时间为 25 秒，我们将实际剩余时间与预计剩余时间作差，ava_time 大于 0 表示时间充足，可以进行下一步操作。此刻局势与我们存储的局势相同表示刚刚更新过。

```
ava_time = stat['now']['timeleft'][me['id']-1]-5 - \
           stat['now']['turnleft'][me['id']-1]*24/2000
updated = fields == storage['last_fields']
```

当时间充足时，对预处理结果进行更新。

```
if not updated and ava_time >= 0:

    storage['last_fields'] = fields
    storage['me_dist'], storage['me_border'] = \
        storage['calc_dist_border'](me['id'], fields, bands, MAX_DIST)
    storage['enemy_dist'], storage['enemy_border'] = \
        storage['calc_dist_border'](enemy['id'], fields, bands, MAX_DIST)
    updated = True
```

（10）预处理己方和对方的领地区域，将己方和对方的距离和边界情况进行更新。

```
if storage['me_dist'][x][y] > 1000:
    MAX_DIST = 100
    storage['last_fields'] = fields
    storage['me_dist'], storage['me_border'] = \
        storage['calc_dist_border'](me['id'], fields, bands, MAX_DIST)
    storage['enemy_dist'], storage['enemy_border'] = \
        storage['calc_dist_border'](enemy['id'], fields, bands, MAX_DIST)
    updated = True
```

2、关于圈地的函数

关于圈地，我们将圈地分成两个阶段，第一阶段为寻找合适的圈地出发点，第二阶段是进行圈地。

首先，我们要考虑圈地和攻击在整个策略中的占比，所以我们设置了 cs0 这个参数用来调整不同局势下进攻的激进程度。当时间充足的情况下，我们进攻的激动度会较大；相反，当我们预计时间不足时，我们会降低进攻的激进度。

```
my_print('s11')
direct = ((1, 0), (0, 1), (-1, 0), (0, -1))
if ava_time >= -1:
    cs0 = 4
else:
    cs0 = 1
```

考虑到已有领地的形状不一，可能出现领地是凹多边形的情况，为了便于己方纸带顺利找到圈地出发起点和圈地路径，我们通过多次比赛不断调整参数，决定先将自己的领地放大 35 个单位，并以 70 为搜索半径寻找距离最近的边界出发点。最后再将领地减少 35 个单位。

```
storage['start_point'] = min(search_area,
    key = lambda pos:
    12 * abs(storage['me_dist'][pos[0]][pos[1]] - 70) +
    4 * dist(pos[0], pos[1], me['x'], me['y']) +
    2 * storage['enemy_dist'][pos[0]][pos[1]] +
    cs0 * dist(pos[0], pos[1], epx, epy) +
    10 * random.random()
    if pos[0] >= 5 and pos[0] < stat['size'][0] - 5 and pos[1] > 5 \
       and pos[1] < stat['size'][1] - 5 and \
       storage['me_dist'][pos[0]][pos[1]] > 1 else 10000

    )
```

我们在热身赛中发现，在面对一些圈地能力相对较弱的敌人时，我们将领地扩大后，会发现整个地图都是己方领地，此时哦我们难以找到合适的圈地出发点，所以我们将强制要求纸带从（50，50）作为圈地初始点。另外，将远离地图的点作为出发点比将地图边界附近的点作为出发点更优。

```
if not (storage['start_point'][0] >= 5 and \
        storage['start_point'][0] < stat['size'][0] - 5 and \
        storage['start_point'][1] > 5 and \
        storage['start_point'][1] < stat['size'][1] - 5):
    storage['start_point'] = (50, 50)
```

找到起始出发点后，我们进入下一阶段——正式圈地。

```
if storage['stage'] == 1:
    my_print('st1')

    if storage['me_dist'][x][y] <= 1:
        storage['out_point'] = (x, y)
    if storage['me_dist'][x][y] >= curr_dist or \
            dist(x, y, storage['start_point'][0], storage['start_point'][1]) < 3:
        storage['start_ang'] = None
        storage['start_point'] = storage['out_point']
        storage['start_ang'] = (x-storage['start_point'][0], y-storage['start_point'][1])
        storage['quan_nearest'] = 1000
        storage['stage'] = 2
        return move_to(x, y, 50, 50)
    return move_to(x,y,storage['start_point'][0],storage['start_point'][1])
```

在这里，为了实时防止敌方的进攻，我们需要动态调整圈地范围。所以我们设置了一个 curr_dist 的参考值，即目前情况下我们期望的圈地范围。

```
dist2 = min(attack_dist2[0], dist(x, y, enemy['x'], enemy['y']))
dist3 = storage['me_dist'][x][y]
if dist2- 2 <= dist3:
    my_print("JP0")
    storage['stage'] = 4

curr_dist = min(storage['calc_dist_band']((enemy['x'],enemy['y']),\
                                          storage['me_band']+[(x, y)])/2-2,90)
```

圈地过程中，需要不断将纸带经过的点坐标添加到’my_band’中。

关于方向设置，我们通过加上 3/1 后模 4 来区别。

```
if storage['me_dist'][x][y] == 0:
    storage['stage'] = 0
faS = direct[me['direction']] + ('A',)
faL = direct[(me['direction'] + 3) % 4] + ('L',)
faR = direct[(me['direction'] + 1) % 4] + ('R',)
x = me['x']
y = me['y']
storage['me_band'].append((x, y))
```

因为存在领地为凹多边形的情况，我们在圈地时会发生较目标圈地大小越来越大的情况，此时我们选择立即回城。

```
if abs(storage['me_dist'][x][y]-curr_dist)-storage['quan_nearest'] > 2:
    storage['quan_nearest'] = min(storage['quan_nearest'], \
                                  abs(storage['me_dist'][x][y]-curr_dist))
    my_print("JP1")
    storage['stage'] = 4
```

另一种情况是敌方纸带距离我方纸带太近，此时也选择立即回城。

```
if curr_dist < 2 or \
        storage['calc_dist_band']((enemy['x'],enemy['y']), \
                                  storage['me_band'])-4<storage['me_dist'][x][y]:
    my_print("JP2")
    storage['stage'] = 4
```

圈地阶段，纸带绕过半圈判定为圈地成功。因为我们选择了当前位置点与起始出发点的连线和边界线的夹角余弦值作为是否绕过半圈的判定条件，考虑到地图时格点图，可能会出现从一个负数突跃到一个正数的情况，所以不把余弦值为 -1 作为判定条件，而是选择 -0.95 作缓冲。

```
if storage['stage'] == 2:
    if not storage['start_ang'] == None and \
            calc_cos(x-storage['start_point'][0], y-storage['start_point'][1],\
                    *storage['start_ang'])<=-0.95:
        my_print("JP3")
        storage['stage'] = 4
    best_fa = (1000, 'A')
    nx = x + faS[0]
    ny = y + faS[1]
```

当纸带到达地图边界附近，无法达到预期圈地目标大小时，则沿着地图边界进行圈地。

```
    if storage['me_dist'][x][y] < curr_dist and \
        (x == 0 or y == 0 or x == stat['size'][0] - 1 or y == stat['size'][1] - 1):
        if fields[x][y] == me['id']:
            my_print("JP0")
            storage['stage'] = 0

        if (nx == 0 or ny == 0 or nx == stat['size'][0] - 1 or \
            ny == stat['size'][1] - 1) and \
            (nx >= 0 and ny >= 0 and nx < stat['size'][0] and \
             ny < stat['size'][1]):
            return 'S'
```

而在这里我们强制要求纸带按照一个方向进行圈地，即要么顺时针要么逆时针。

```
for fa in (faS, faL, faR):
    nx = x + fa[0]
    ny = y + fa[1]
    if nx >= 0 and ny >= 0 and nx < stat['size'][0] and \
            ny < stat['size'][1] and not (nx, ny) in storage['me_band']:
        if storage['start_ang'] == None or \
            calc_cj(storage['start_ang'][0],storage['start_ang'][1],\
                    x-storage['start_point'][0],y-storage['start_point'][1])*\
            calc_cj(x-storage['start_point'][0],y-storage['start_point'][1],\
                    nx-storage['start_point'][0],ny-storage['start_point'][1]) >= 0:
                if abs(storage['me_dist'][nx][ny]-curr_dist)<100 and\
                        len([pos for pos in storage['me_band'] if dist(pos[0],pos[1],nx,ny)<3])<5:
                    best_fa = min(best_fa,(abs(storage['me_dist'][nx][ny]-curr_dist),fa[2]))
```

3、关于进攻的函数

首先，我们要确定己方纸带到对方纸带的最短距离和对方纸带回城的最短距离。

```
attack_dist = min_to_enemy_band(me['x'], me['y'], enemy['id'])
attack_dist2 = min_to_enemy_band(enemy['x'], enemy['y'], me['id'])
```

一开始，我们认为进攻时应该直接奔向敌人所在位置，但考虑敌人的能动性，所以在距离敌人并不是非常近的时候，我们会给敌人位置 x,y 坐标加上一个随机数以预判敌人大致的行动范围，总体上还是在向敌人前进，但同时具有一定的能动性，而不是死死盯着敌人实时位置，这在一定程度上可以减少地图更新频率。

```
if MAX_DIST > 20:
    epx = enemy['x'] + random.randint(-40, 40)
    epy = enemy['y'] + random.randint(-40, 40)
    my_print('s111')
    search_area = [(nx, ny) for nx in range(stat['size'][0]) \
                   for ny in range(stat['size'][1])]
else:
    epx = enemy['x'] + random.randint(-20, 20)
    epy = enemy['y'] + random.randint(-20, 20)
    my_print('s112')
    search_area = [(nx, ny) for nx in range(max(0, x - 20), min(stat['size'][0], x + 20)) \
                   for ny in range(max(0, y - 20), min(stat['size'][1], y + 20))]
```

更新地图后发现敌人回城最短距离要大于我方纸带到对方纸带的最短距离，此时决定靠近进攻，即出发阶段三。

```
if storage['stage'] in (1, 2):

    if updated:
        if storage['enemy_dist'][enemy['x']][enemy['y']] \
                > attack_dist[0] + 1 and attack_dist2[0] > attack_dist[0] + 2:
            storage['stage'] = 3
```

但如果我们回城距离加上到敌方纸带的最短距离仍大于敌方回程最短距离时，我们可是先回城再出来攻击对方。

```
        if storage['enemy_dist'][enemy['x']][enemy['y']] \
                > attack_dist[0] + 2 * storage['me_dist'][me['x']][me['y']] + 2:
            storage['stage'] = 4
    if storage['me_dist'][x][y] > 0:
        storage['me_band'].append((x, y))
    else:
        storage['me_band'] = []
```

如果我方纸带到对方纸带的最短距离要小于对方纸带回城的最短距离，则发起进攻。

```
if storage['stage'] == 0:
    my_print('s0')
    if updated and storage['enemy_dist'][enemy['x']][enemy['y']] > attack_dist[0] + 2:
        storage['stage'] = 3
        my_print('s1')
```

进攻模式的指令为 3，当所处阶段为 3 时，发动进攻，调用 move_to 函数直接靠近攻击对方纸带；当发现当前局面已不利于我方发动进攻时，选择回城，触发回城指令 4。

```
if storage['stage'] == 3:
    my_print('s3', me['x'], me['y'], attack_dist)
    if not storage['enemy_dist'][enemy['x']][enemy['y']] >= attack_dist[0]:
        storage['stage'] = 4
    else:
        return move_to(me['x'], me['y'], attack_dist[1], attack_dist[2])
```

4、关于回城的函数

记录回城路径上的点坐标存储为领地边界点。

```
if storage['stage'] == 4:
    my_print('s4')
    storage['me_band'].append((x, y))
    if storage['me_dist'][x][y] <= 1:
        if (not updated and ava_time >= -1):
            storage['last_fields'] = fields
            storage['me_dist'],storage['me_border']=\
                storage['calc_dist_border'](me['id'], fields, bands, MAX_DIST)
            storage['enemy_dist'], storage['enemy_border'] = \
                storage['calc_dist_border'](enemy['id'], fields, bands, MAX_DIST)
        if storage['me_dist'][x][y] <= 1:
            storage['me_band'] = []
            storage['stage'] = 0
```

设置继续直行、左转、、右转三个方向的参数

```
faS = direct[me['direction']] + ('S',)
faL = direct[(me['direction'] + 3) % 4] + ('L',)
faR = direct[(me['direction'] + 1) % 4] + ('R',)
best_fa = (1000, 'A')
```

在三个方向中取距离边界最近的方向，并且要注意不能撞到自己的纸带。

```
for fa in (faS, faL, faR):
    nx = x + fa[0]
    ny = y + fa[1]
    if nx >= 0 and ny >= 0 and nx < stat['size'][0] and \
            ny < stat['size'][1] and not (nx, ny) in storage['me_band']:
        best_fa = min(best_fa, (storage['me_dist'][nx][ny], fa[2]))
```

当回城最短路径的距离大于当前位置回到领地的最短距离时，更新地图信息，再挑选三个方向中距离领地最近的方向并注意不能撞到自己的纸带。

```
if best_fa[0] > storage['me_dist'][x][y]:
    my_print("BACK_UPDATE")
    storage['me_dist'], storage['me_border'] = \
        storage['calc_dist_border'](me['id'], fields, bands, MAX_DIST)
    for fa in (faS, faL, faR):
        nx = x + fa[0]
        ny = y + fa[1]
        if nx >= 0 and ny >= 0 and nx < stat['size'][0] and\
                ny < stat['size'][1] and not (nx, ny) in storage['me_band']:
            best_fa = min(best_fa, (storage['me_dist'][nx][ny], fa[2]))
```

2.3 程序限制

通过多次比赛，我们发现当领地为凹多边形，我方纸带又恰好把凹陷部分边界作为圈地的起始出发点时，在回城时有很大几率会自己撞上自己的纸带而自杀的错误。

3 实验结果

3.1 测试数据

实验环境说明：

- 硬件配置：CPU core i5，8G

- 操作系统：Windows 10

- Python 版本：Python 3.6

1.warmup2：本次 27 个代码参与，排名 11，以下是对前 10 名与我组的战局的统计分析：

对手	先手	败因
foxtrot_3	对手	在对方领地被撞
	india	在对方领地撞对手； 在空白区被撞击纸带。
Menhera_2	对手	在对方领地被撞； 撞墙自杀； 圈地大小较小；
	india	在对方领地被撞； 圈地大小较小； 撞纸带自杀；
Quebec_2	对手	被撞纸带；
	india	被撞纸带； 圈地大小较小；
KizunaAI_3	对手	在对方领地撞对手；
	india	圈地大小较小； 纸带被撞； 撞纸带自杀；
Whiskey_1	对手	纸带被撞； 撞墙自杀；
	india	纸带被撞； 撞纸带自杀；
Quebec_1	对手	纸带被撞；
	india	纸带被撞；
Foxtrot_2	对手	在对方领域被撞； 纸带被撞； 在对方领地撞对手；
	india	在对方领地被撞；
Echo_1	对手	撞纸带自杀； 在对方领地被撞； 纸带被撞；
	india	纸带被撞；
Whiskey_2	对手	纸带被撞
	india	圈地大小较小； 撞纸带自杀；
Menhera_3	对手	撞纸带自杀； 纸带被撞；
	india	撞纸带自杀； 在对方领地撞对手； 在对方领地被撞；

2.Warmup3:36 个代码参与，排名 14，以下是对前 13 名与我组的战局的统计分析：

对手	先手	败因
foxtrot_1	对手	在对方领地被撞； 纸带被撞； 在对方领地撞对手；
	india	在对方领地被撞；
foxtrot_3	对手	纸带被撞；
	india	在对方领地被撞； 纸带被撞； 在对方领地撞对手；
Menhera_3	对手	被撞纸带； 圈地大小较小；
	india	被撞纸带； 圈地大小较小；
x-ray_1	对手	纸带被撞； 在对方领地被撞； 圈地大小较小；
	india	圈地大小较小； 纸带被撞；
KizunaAI_5	对手	圈地大小较小； 纸带被撞； 在对方领地撞对手；
	india	在领地内撞对手； 在领地内被撞；
Menhera_2	对手	圈地大小较小； 在对方领地撞对方； 撞纸带自杀；
	india	圈地大小较小； 纸带被撞；
Menhera_4	对手	在对方领域被撞； 圈地大小较小； 撞纸带自杀； 纸带被撞；
	india	在对方领地撞对手； 在对方领地被撞； 撞纸带自杀； 纸带被撞； 圈地大小较小；
KizunaAI_4	对手	在对方领地撞对方； 圈地大小较小； 纸带被撞； 在对方领地被撞；
	india	在对方领地被撞； 在对方领地撞对方

foxtrot_4	对手	在对方领地被撞； 纸带被撞； 在对方领地撞对方；
	india	纸带被撞； 在对方领地被撞； 在对方领地撞对方；
Whiskey_1	对手	纸带被撞； 圈地大小较小；
	india	纸带被撞； 在对方领地撞对方； 在对方领地被撞；
Juliet_2	对手	圈地大小较小；
	India	纸带被撞； 在对方领地撞对方； 在对方领地被撞 圈地大小较小；

3.Warmup3 1:36 个代码参与，排名 5，以下是对前 4 名与我组的战局的统计分析：

对手	先手	败因
Menhera_4	对手	纸带被撞； 在对方领地内被撞； 圈地大小较小；
	india	纸带被撞； 圈地大小较小；
Juliet_2	对手	纸带被撞； 圈地大小太小；
	india	纸带被撞； 圈地大小较小； 撞纸带自杀；
Menhera_2	对手	被撞纸带； 圈地大小较小；
	india	圈地大小较小； 在对方领地被撞； 纸带被撞；
Menhera_3	对手	纸带被撞； 圈地大小较小；
	india	圈地大小较小； 在对方领地被撞； 纸带被撞；

4. 正式比赛之小组赛：

在小组赛中我们拿到了第二的好成绩，大多数的胜局以我们进攻成功而结束，可见我们进攻策略的完备。

以下是我们面对实力强劲的对手，对失败战局的分析：

对手	先手	败因
Bravo	对手	回合数耗尽，圈地面积较小（4次）； 纸带被撞；
	india	撞纸带自杀； 回合数耗尽，圈地面积较小（4次）
Quebec	对手	对方领地内被撞； 回合数耗尽，圈地面积较小（2次）； 纸带被撞；
	india	撞纸带自杀 纸带被撞； 对方领地内被撞； 回合数耗尽，圈地面积较小；
Juliet	对手	回合数耗尽，圈地面积较小（3次）； 纸带被撞； 撞纸带自杀；

分析：

在拉锯战中，圈地能力不足是我们的短板；与此同时，在之前的热身赛，我们也多次出现过撞纸带自杀的情况。虽然针对这个情况，代码有所改进，但面对复杂的持续决策，撞纸带自杀的情况依然出现（主要是在计算时间不够的情况下，纸带随机判断方向前进导致的后果）。对于上述两个情况，我们进步的空间仍然相当大。

5. 正式比赛之八进四（对手 lima（E1））：

先手		结果	分析
己方 india	1	回合数耗尽，己方赢	算法相克，双方相向即将碰头时回头，远离后又相互靠近，在最终各自转圈。
	2	回合数耗尽，对方赢	同上
	3	回合数耗尽，对方赢	同上
	4	回合数耗尽，对方赢	同上
	5	回合数耗尽，对方赢	同上

先手		结果	分析
对方：lima	1	在 lima 领域撞击对方，对方赢	在双方领地边界，思虑不周，进攻时自己的领地极易变成对方的。
	2	侧面撞击对手，己方赢	进攻策略生效
	3	撞击对方纸带，己方赢	进攻策略生效。
	4	侧面撞击对手，己方赢	进攻策略生效
	5	侧面撞击对手，己方赢	进攻策略生效。
	6-8	回合数耗尽，对方赢	同己方先手情况
	9-10	回合数耗尽，己方赢	同上

分析：

和 E1 的比赛出现了意外的情况。两种算法互相克制，造成两者在距离很近的时候都开始循环路线，直到回合数用完。而我们采用梯形边界圈地的策略，导致开局的时候圈地面积小，进而在遭遇这种情况时我们组大都以失败结束。但我们不后悔采用梯形边界，这种边界在防御方面优势较大。若要继续改进，我们会针对这种循环的情况，计算双方领地大小，占优势的时候继续循环，不占优势时解除这种状态。观察其他对战，可以看到，我们在进攻方面占显著优势，这给了我们一些慰藉。

3.2 结果分析

本组代码未用到深度学习之类的复杂算法。热身赛前，主要的策略是人脑参与游戏，总结经验，技术人才手动调整参数来让代码升级。热身赛后，我组算法的策略主要是通过历史数据找出自己算法的不足，然后做出相应的改进，这里对对方的策略尽自己的努力做出应对。

热身赛前，通过人脑总结归纳经验，我们的策略重在圈地和防御。计算自己与对手的距离和自己与领地的距离，尽可能大而安全地圈地，与此同时，用来和对手保持一个安全距离。最后的圈地效果较为理想。在热身赛中，我们组的代码在圈地的能力上比较满意，运行时间也在预期之中。但面对攻击性的对手，本策略失效，结果以失败告终。

热身赛后，我们增添了攻击的程序。如果对手与自己的相对位置情况不满足圈地的条件，我们会计算自己进攻或者回到领地保命对应的期望值，然后做出相应判断。这样圈地能力有所下降，但面对进攻的对手，我们更有底气。运行时间有所增加，我们添加了限制决策时间的函数，所以未出现超时失败现象。

4 实习过程总结

4.1 分工与合作

任务分配：

1、开发算法和编程任务：黄荣编写代码，侯华丽负责测试，小组全体成员负责提供算法思想、个人经验。黄荣负责会议组织和琐碎事宜。

2、小组报告：张君曼，祝佳琪，魏论研负责小组报告。祝佳琪，魏论研负责第一、二部分的撰写，张君曼负责报告余下的部分，同时负责报告各部分的整合和最终的排版。

合作讨论方式：

线上：在微信群中进行讨论，通过文字、图片、语音进行交流。

线下：小组会议讨论。

历次组会记录：

1、第一次小组会议

参与人员：全体成员。

时间：2018 年 5 月 25 日 15：00 到 17：00

地点：宿舍

主要内容：

每位组员仔细剖析比赛规则并弄清相关资料。

（未拍照）

2、第二次小组会议

参与人员：全体人员

时间：2018 年 5 月 28 日 21：00 到 22：30

地点：宿舍

主要内容：针对比赛规则进行商讨，玩对应小游戏，根据游戏确定比赛策略和技巧。

会议照片：（未拍照）

3、第三次小组会议

参与人员：全体人员

时间：2017 年 5 月 30 日 14：00 到 17：00

地点：宿舍

主要内容：

进一步弄清课程网站上提供的代码；

搭建了算法的大体框架，初步确定了策略。

（未拍照）

4、第四次小组会议

参与人员：全体人员

时间：2018 年 6 月 4 日 15：00 到 17：10

地点：四教 305

主要内容：在之前算法的框架上进一步完善算法。

会议照片：


5、第五次小组会议

参与人员：全体人员

时间：2018 年 6 月 8 日 18：00 到 21：30

地点：理教 313

主要内容：根据热身赛的结果，分析我们的不足，调整算法。

会议照片：


4.2 经验与教训

我们组一共五位组员，是住在宿舍同一层楼的同学，所以除开组会，我们相互的交流也很多。每一位组员都很负责认真，为这次大作业投入了很多的时间精力。在整个过程中，我们收获了很多，不仅是课堂内的知识，也有组织、沟通能力的提高。

大作业布置之初，我们聚在一起讨论，希望每个人都能参与到算法思路的构建中。但这样大家不知道怎么行动，有拖延的现象，所以在开始两次组会，我们效率很低。故明确的分工还是必要的。

在最开始，我们都对比赛规则了解得不够透彻，因为光看规则确实难以理解。因此，组长组织了一次组会，带领我们玩了几次类似的游戏，这样不仅加深了我们对于游戏规则的了解，也有助于我们思考策略和经验。

为了进一步优化我们的策略，我们特地请教了在对数据结构比较了解的同学，他从大的局面给我们讲解了专家系统， 剪枝等思路。这不仅丰富了我们的知识，也促进了我们对本游戏策略的思考。后来，碍于自己编程水平和比赛判断时长等因素的限制，我们选择了依靠自己的经验编程，并逐步开始编写程序。故适当的自主学习是必要的。

期末大作业渐渐落幕，很高兴能参与到其中以来。最后，谢谢组员，也谢谢所有为了大作业而付出努力的老师和同学们！

4.3 建议与设想

改进：

编程实践仓促，基础设施代码不断在更新，造成一定干扰；

希望能够开发一个友谊赛平台，方便大家互相约战；

编程对我们来说，是一件极其耗费时间和经验的事，放在期末开展，部分同学会有不愿参加的情绪，希望能调整时间。

寄语：

数据结构与算法是一门很好的课程，它提供了期末大作业这样一个考核形式，我们不仅能够学到知识，而且还能锻炼自己的能力，培养团队合作精神。小组合作的过程中，大家是一个整体，每个人都要为了整个小组而努力。同时，希望后来者能积极对待，合理分布时间，享受这个过程。

5 致谢

感谢陈斌老师一学期以来的指导！

感谢与我们进行热身赛的小组！

感谢比赛现场的各位到场同学！

6 参考文献

paper.io play online ： `http://paper-io.com/?referer=paper.io&channel=11`

第十一章　F17_Juliet 报告

郑云泽、胡俊杰、刘煜杭 *、任淼、但浩文

摘要：我们的算法以基础策略为主，采用广搜算法和列表、字典等常规数据类型进行编写，结合了防御、扩张与进攻，通过不同的函数控制不同模式的进行以达到确定安全的情况下扩张与进攻。在实验过程中我们与自己的代码进行了对比，不断增加了扩张和防御性能，最后增加了攻击函数，也和他人的代码有所比较，最终都取得了胜利，但同时也发现了一些不足进行了改进。

关键字：【方形扩张】【估算进程】【就近攻击】【地空排球】

1 算法思想

1.1 总体思路

算法按照最常规的思路进行，即躲避对方攻击，在能保证自己安全的情况下尽可能扩张领地，如果有机会对敌人进行攻击。算法将路径分为几种模式：领地内转移 shift，扩张领地 expand，攻击 attack，返回领地 back。在这几种模式下，路线选择基本是按一定模式进行的，即简单条件下的最大化，因此会有一定的 bug，但是基本上可以应对一些一般情况。

1．方形扩张策略

由于在进行友谊赛时发现很多小组的代码都是以攻击为主，因此他们在领地扩张方面会有一定的缺陷，所以我们采用了以方形扩张为主的扩张方式，这样在保证安全的同时可以更快地扩大领地。在进行小组赛时我们的方形扩张策略就体现了一定的优势：在与 India

组同积分时以较大的领地面积优势取得了小组第一。

2. 估算进程策略

在进行扩张或进行攻击之前，我们会对前进的路线进行估计。比如在进行领地内转移时，会用 shift 函数对下一次的扩张起始点以及扩张终止点进行估计，我们对每一步的前进方向进行了计算，分为一次移动方向和次级移动方向，当一级移动方向存在则优先按照一级移动方向运动，否则按照次级移动方向运动，如果两种移动方向都不存在则直行即可。另外，估算进程还可以用于对是否攻击对方以及是否选择回领地的选择，如果在估计误差范围内可以进行操作，否则就改变路线或者继续进行当前的操作。

3. 就近攻击策略

由于算法主要是进行领地扩张，因此在进行攻击时没有采用一些高端的跟踪攻击或者远程监测的方法，而只是对邻近的对手进行攻击。不过，算法还考虑了一些头对头侧碰的“骚操作”，当可能出现我方与对方互相缠斗的情况时，就计算可能相碰的位置，如果可能被对方撞死则调整路线转使得对方可以被我方撞死。

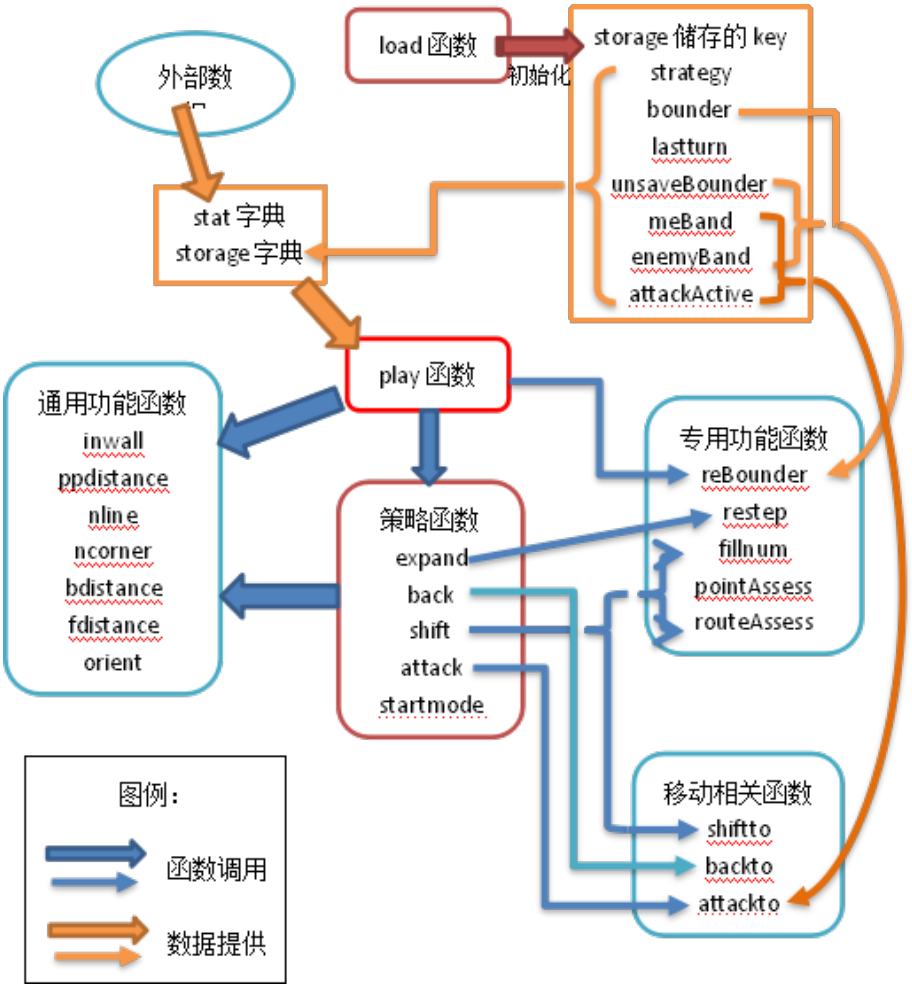
4.CD 冷却策略

在进行领地内转移时，首先要选择下一次扩张的起始点，但是有时会找不到合适的起始点，这时我们就采用了一种 CD 冷却策略：任意方向前进五回合后重新进行检测，找到新一轮扩张点后再开始下一轮扩张。

5．领地间隔刷新策略

由于游戏数据是随时储存的，所以需要对数据进行刷新，但由于游戏本身时间的限制，对所有的数据进行刷新将会耗费过多的时间，因此采用了间隔刷新的策略即每隔六个点刷新一次。对于边角的点也采用了一种答题估计的方法，检测以该点为中心的正方形的四个顶点中有多少被占领从而判断该边界点的归属。

1.2 算法流程图



1.3 算法复杂度

算法整体上来说较为基本，没有特别复杂的算法以及数据结构，而不同的函数算法复杂度有所不同，总体上如下所述：bdistance 函数时间复杂度为 O（n），其中 n 为纸带长度；fdistance 函数时间复杂度为 O（n^2），其中 n 为搜索的距离；fillnum 函数的时间复杂度为 O（n^2），其中 n 为一边的长度即出发不转弯一次能走的长度；reBounder 函数的时间复杂度要分为两种情况来看，首先是自己能够回到领地的情况，这时时间复杂度为 O（n^2），n 为已记录的边界顶点，然后是敌人吃掉自己领地的情况，这时时间复杂度为 O（L），L 为敌方纸带的长度；pointAssess 的时间复杂度为 O（n^2）,n 也代表了一条边的长

度；routeAssess 函数的时间复杂度为 p^2*pointAssess，其中 p 为已记录边界顶点；剩下的策略函数大致分为两种，一种是非模式转换类，此时多为判断语句，时间复杂度为 O（1），另一种就是模式转换类，这类函数需要调用前面的函数，每个回合都需要搜索是否是处于危险或者可以进行攻击，因此时间复杂度为 O（$n^2 + p$），其中 n 为自己或敌人到各自领地的距离，p 为自己或敌人纸带的长度。

2 程序代码说明

2.1 数据结构说明

本次实习作业没有采用一些高级的数据结构和算法，而仅使用了老师上课讲解的广搜算法和列表、字典等常规数据类型。

2.2 函数说明

inwall 函数：

判断点是否在地图内，防止撞墙

```
def inwall(size,point):
    #point[0]为点的横坐标，point[1]为点的纵坐标
    return 0<=point[0]<size[0] and 0<=point[1]<size[1]
```

ppdistance 函数：

计算两点最短路径长度

```
def ppdistance(pa,pb):
    return abs(pa[0]-pb[0])+abs(pa[1]-pb[1])
```

nline 函数：

判断 point 点的东南西北之中可扩张的方向（未占有区域方向）

```
def nline(sta,point):
    direct = ((1, 0), (0, 1), (-1, 0), (0, -1))
    nlinelst=[]#用于储存可扩张的方向
    for dire in range(4):#遍历东西南北四个方向
        x,y=point[0]+direct[dire][0],point[1]+direct[dire][1]#朝某方向行进所得的新坐标
        if inwall(sta['size'],(x,y)):#如果新坐标在地图里
            if sta['now']['fields'][x][y]!=sta['now']['me']['id']:#并且新坐标不在自己领地内
                nlinelst.append(dire)#则储存此坐标
    return nlinelst
```

ncorner 函数：

判断顶点的四个顶点之中有几个可以占领并返回有几个未占有顶点

```
def ncorner(sta,point):
    cnt=0#用于记录未占领顶点的数量
    direct = ((1, -1), (1, 1), (-1, 1), (-1, -1))#东北，东南，西南，西北四个方向
    for dire in range(4):#遍历四个方向的四个顶点
        x, y = point[0] + direct[dire][0], point[1] + direct[dire][1]#某个方向上的顶点坐标
        if inwall(sta['size'], (x, y)):#如果此顶点存在即在地图内
            if sta['now']['fields'][x][y] != sta['now']['me']['id']:#并且此顶点不在自己领地内
                cnt+=1#计数器数值加一
    return cnt
```

bdistance 函数：

算出 centerp 点到某方纸带 band 的最短距离

centerp 是起始点，dire 是起始点的朝向，band 填 storage['meBand'] 或 ['enemyBand']

otherplayer 是某玩家纸卷的坐标（一般是 band 的所有者）

```
def bdistance(centerp,dire,band,otherplayer):
    mindis=ppdistance(centerp,otherplayer)#从otherplayer的纸卷开始遍历
    nearp=otherplayer#用于记录取最小值时对应的纸带上的点
    if len(band)>1:#用于判断是否需要进行循环
        # 对纸带的每个点算距离，取最小值
        for point in band:
            ppdis=2 if orient(centerp,dire,point)=='BB' else 0
            #如果point点在centerp点的正后方，那么需要先进行一步掉头即在正常两点距离基础上加2
            ppdis+=ppdistance(point,centerp)#ppdis用于记录certerp点纸带上点距离
            if ppdis<mindis:#如果certerp点到纸带上点的距离小于原来记录的最小值即mindis
                mindis=ppdis#mindis的值替换为此时的ppdis值
                nearp=point
    return mindis,nearp
```

fdistance 函数：

算出 centerp 点到某一方领地的最短距离

centerp 是起始点,fieldid 是某一方玩家的 id

backUnavail：'No' 是可以搜索正后方，'weak' 是不能搜正后方，'strong' 是不能搜后方某个角度范围

dire 是 centerp 点的朝向，manxdeep 是最大搜索深度即可以搜索的最大范围

输出最近距离和最近点

```
def fdistance(sta, centerp, fieldid, backUnavail, dire=None, maxdeep=None):
    field=sta['now']['fields']
    size=sta['size']
    #在maxdeep没有初始值时要对maxdeep进行计算，即分别在东西和南北方向上到边界的最大距离
    maxdeep= max(size[0] - centerp[0] - 1, centerp[0]) + max(size[1] - centerp[1] - 1, centerp[1])\
        if maxdeep is None else maxdeep#如果maxdeep有初始值则可跳过此步操作
    # 按照距离从近到远的顺序进行广度优先搜索
    for dis in range(maxdeep+1):
        for xdis in range(-dis,dis+1):
            for point in ((centerp[0] + xdis, centerp[1] + dis - abs(xdis)),
                          (centerp[0] + xdis, centerp[1] - dis + abs(xdis))):
                # 如果point点在玩家领地内，则说明该point点就是领地内到centerp点最短距离的点，此时对应的dis就是point点与centerp点之间的最短距离
                if inwall(size,point) and field[point[0]][point[1]]==fieldid:
                    if backUnavail=='No':
                        backeval=True#backeval用于判断可否对point点进行搜索
                    else:
                        ori=orient(centerp, dire, point)#point点在centerp点的大致方位
                        backeval= (ori!= 'BB') if backUnavail=='weak' else not ('BB' in ori)
                        #在'weak'的情况下，只要point点不在centerp点的正后方就可以搜索；在'strong'的情况下
                        #只要point点不在正后方附近的很小范围内就可以搜索
                    if backeval:#如果可以对point点进行搜索，则进行输出
                        return dis,point
    return maxdeep+1,(None,None)#防止出现bug
```

orient 函数：

判断点 point 在参考点 centerp 的相对方向

dire 是 centerp 点的指向

```
def orient(centerp,dire, point):
    # 计算相对方位的，reladic[0]是point在centerp前方的距离，reladic[1]是point在centerp右方的距离
    # reladic中的数据分别为point在东边的距离，point在南边的距离，point在西边的距离，point在北边的距离，
    #当用dire切片之后，就变成point在centerp前边以及右边的距离
    reladic= (point[0] - centerp[0], point[1] - centerp[1], centerp[0] - point[0],
              centerp[1] - point[1], point[0] - centerp[0])[dire:dire + 2]
    if reladic==(0,0):#point与centerp处于同一位置
        return 'FFBB'
    elif abs(reladic[1])*3<abs(reladic[0]):#point大致位于centerp的正前/后方
        if reladic[0]>0:
            return 'FF' if reladic[1]==0 else 'RFF' if reladic[1]>0 else 'LFF'#分别为：正前方，前方稍偏右，前方稍偏左
        else:
            return 'BB' if reladic[1] == 0 else 'RBB' if reladic[1] > 0 else 'LBB'#分别为：正后方，后方稍偏右，后方稍偏左
    elif abs(reladic[0])*3<abs(reladic[1]):#point大致位于centerp的正左/右方
        if reladic[1] > 0:
            return 'RR' if reladic[0] == 0 else 'RRF' if reladic[0] > 0 else 'RRB'#分别为：正右方，右方稍偏前，右方稍偏后
        else:
            return 'LL' if reladic[0] == 0 else 'LLF' if reladic[0] > 0 else 'LLB'#分别为：正左方，左方稍偏前，左方稍偏后
    else:#其余情况则依次归为右前，左前，右后，左后
        if reladic[0]>0:
            return 'RF' if reladic[1] > 0 else 'LF'
        else:
            return 'RB' if reladic[1] > 0 else 'LB'
```

restep 函数：

用于估计前进（直行）扩张需要的步数

start 用于判断是否已经开始扩张

assess 模式是还在领地内 shift 时预算能走的步数

assess 只用于 routeAssess，其中包含三元参数，assess[0] 是目前的位置，assess[1] 是扩张终点，assess[2] 是当前到达目标点所需要的转弯数

```
def restep(sta,stor,dire,start=False,assess=None):
    # 变量名简化
    if assess is None:#如果assess没有初始参数值，则需要进行初始化
        myp=(sta['now']['me']['x'],sta['now']['me']['y'])
        endpx,endpy=stor['strategy']['endp'][0],stor['strategy']['endp'][1]
        turn=stor['strategy']['turn']#预计转弯数
    else:#如果assess已经有初始参数值，则直接使用
        myp = assess[0]
        endpx, endpy = assess[1][0],assess[1][1]
        turn=assess[2]
    enmp=(sta['now']['enemy']['x'],sta['now']['enemy']['y'])
    mxdeep=ppdistance(myp,enmp) if start else \
        bdistance(enmp,sta['now']['enemy']['direction'],stor['meBand'],myp)[0]#计算敌人到自己组带的最短距离，用于计算自己可扩张的距离
    if turn==0:#如果沿当前方向可以直达结束点
        step = min((endpx - myp[0], endpy - myp[1],myp[0] - endpx, myp[1] - endpy)[dire],mxdeep*2//3)
        # 改成*2//3是因为有了更强的防御机制，可以更大胆地扩张；若直达可能被敌人攻击，则需要在保证安全的前提下改变路线
    else:#如果仍然需要拐弯才能到达终点
        walldis=(sta['size'][0]-myp[0]-1,sta['size'][1]-myp[1]-1,myp[0],myp[1])
        step = min(mxdeep*2 // ((turn+1)*3), walldis[dire])#在保证安全的前提下尽可能向地图边界扩张
    return step
```

fillnum 函数：

估算可扩充的领地大小

p1，p2 为任意点

```
def fillnum(sta, p1, p2):
    # 以p1点和p2点为对角线顶点可以确定一个矩形
    nw=(min(p1[0], p2[0]), min(p1[1], p2[1]))#矩形的左上顶点
    se=(max(p1[0], p2[0]), max(p1[1], p2[1]))#矩形的右下顶点
    cnt=0#用于记录样本点的数量
    stepargv=[]#用于记录样本点间隔，第一项是东西向边上的样本点间隔，第二项是南北向边上的样本点间隔
    # 根据p1，p2围城面积的大小取样本点间隔，是跳着选取样本点估计可扩充面积
    for i in range(2):
        #不同长度范围对应不同样本点间隔，最大为5
        for argv in ((10,1),(20,2),(30,3),(40,4)):
            if argv[0]-(se[i]-nw[i])<10:
                stepargv.append(argv[1])
                break
        else:
            stepargv.append(5)
    # 选取样本点估算未成为自己领地的点数
    for x in range(nw[0],se[0]+1,stepargv[0]):
        for y in range(nw[1], se[1] + 1, stepargv[1]):
            if sta['now']['fields'][x][y]!=sta['now']['me']['id']:
                cnt+=1#记录不在自己领地内的样本点数
        return cnt*stepargv[0]*stepargv[1]#返回估算得到的未占领点数即该矩形内可扩充的领地大小
```

shiftto 函数：

估计运行 shift 模式时当前的最佳前进方向

```
def shiftto(sta,stor,start=False):
    direct = ((1, 0), (0, 1), (-1, 0), (0, -1))
    x, y, mydire = sta['now']['me']['x'], sta['now']['me']['y'], sta['now']['me']['direction']
    # direlst是优先的转向方向，sublst是次一级的，当direlst为空时采用
    direlst,sublst= [],[]
    turndic = {mydire: 'F', (mydire - 1) % 4: 'L', (mydire + 1) % 4: 'R'}
    # 在shift模式走到下一个扩张出发点，无cd冷却
    if 'assessCD' not in stor['strategy']:
        ori=orient((x,y),mydire,stor['strategy']['nextstart'])#下一个扩张出发点的相对方向
        for dire in (mydire, (mydire - 1) % 4, (mydire + 1) % 4):#分别查找前、左、右三个方向
            point = (x + direct[dire][0], y + direct[dire][1])#在该方向上前进一步所得到的坐标
            if inwall(sta['size'], point) and turndic[dire] in ori:#如果下一个扩张出发点在目前查找的方向上
                sublst.append(dire)#则先将此方向作为次级转向方向
                if sta['now']['fields'][point[0]][point[1]] == sta['now']['me']['id']:
                    direlst.append(dire)
```

```
    if direlst:
        turn = turndic[direlst[0]]#优先记录一级转向方向
    elif start:#已经找到下一个扩张出发点
        if stor['lastturn'][0] == 'R':
            #如果再上一个拐角处右拐的话则尽量右拐，便于形成闭合曲线，更快地back，同时保证自身安全
            turn = 'R' if inwall(sta['size'], (x+direct[(mydire+1)%4][0],
                    y+direct[(mydire+1)%4][1])) else 'L'
        else:
            #在上一个拐角处左拐的话同理
            turn = 'L' if inwall(sta['size'], (x+direct[(mydire-1)%4][0],
                    y+direct[(mydire-1)%4][1])) else 'R'
    else:
        turn=turndic[sublst[0]]#最后才记录次级转向方向
# shift处于CD冷却中，漫游
else:
    for dire in (mydire, (mydire - 1) % 4, (mydire + 1) % 4):
        point = (x + direct[dire][0], y + direct[dire][1])
        if inwall(sta['size'], point):
            sublst.append(dire)
            if sta['now']['fields'][point[0]][point[1]] == sta['now']['me']['id']:
                direlst.append(dire)
    turn=turndic[direlst[0]] if direlst else turndic[sublst[0]]
return turn
```

backto 函数：

在 back 模式走回自己领地

```
def backto(sta,stor,start=False):
    direct = ((1, 0), (0, 1), (-1, 0), (0, -1))
    myx, myy = sta['now']['me']['x'], sta['now']['me']['y']
    mydire=sta['now']['me']['direction']
    enmx,enmy=sta['now']['enemy']['x'],sta['now']['enemy']['y']
    # direlst是优先的转向方向，sublst是次一级的，当direlst为空时采用
    direlst, sublst = [], []
    turndic = {mydire: 'F', (mydire - 1) % 4: 'L', (mydire + 1) % 4: 'R'}#各方向相对于自身的方位
    antiturn = {'L': (mydire + 1) % 4, 'R': (mydire - 1) % 4,'F':(mydire - 2)%4}#自身方位对应的方向
    endp=stor['strategy']['endp']
    # 确定搜索方向的优先级，先直行还是先转弯
    # 两个Rela分别记录终止点、敌方在前方、右方的距离
    meEndpRela = (endp[0]-myx, endp[1] - myy, myx - endp[0],
                  myy - endp[1], endp[0]-myx)[mydire:mydire + 2]
    meEnmRela = (enmx-myx, enmy - myy, myx - enmx,
                 myy - enmy, enmx-myx)[mydire:mydire + 2]
    if meEndpRela[0]*meEnmRela[0]<=0:
        searchOrder=(mydire, (mydire - 1) % 4, (mydire + 1) % 4)#如果终止点与敌人一前一后，那么优先搜索前方
    elif meEndpRela[1]*meEnmRela[1]<0 or (abs(meEndpRela[0])<abs(meEnmRela[0])
                                          and abs(meEndpRela[1])>abs(meEnmRela[1])):
        #如果终止点与敌人前后方向上处于同一侧，但是一左一右或者前后方向上到终止点较近，并且终止点在左右方向较远
        searchOrder = ((mydire - 1) % 4, mydire, (mydire + 1) % 4) if meEndpRela[1]<0\
            else ((mydire + 1) % 4, mydire, (mydire - 1) % 4)#如果终止点在左边就优先搜左边，否则优先搜右边
    else:
        searchOrder = (mydire, (mydire - 1) % 4, (mydire + 1) % 4)
# 开始按方向搜索可行性
for dire in searchOrder:
    point = (myx + direct[dire][0], myy + direct[dire][1])#沿某一方向前进所得的点
    if inwall(sta['size'], point) and sta['now']['bands']\
            [point[0]][point[1]] != sta['now']['me']['id']:#如果该点在地图内并且没有撞自己纸带
        sublst.append(dire)#那么就可以在次级搜索方向中添加这个方向
        if stor['strategy']['path'][dire] > 0:
            direlst.append(dire)
if direlst:#只要direlst存在就按照这个方向拐弯
    turn = turndic[direlst[0]]
elif start:
    if stor['lastturn'][0] == 'R':
        #如果上一个拐弯处为右拐，那么优先左拐，因为这样可以避免转圈，可以更快地回到领地
        turn = 'L' if inwall(sta['size'], (myx + direct[(mydire - 1) % 4][0],
                    myy + direct[(mydire - 1) % 4][1])) else 'R'
    else:
        #在上一个拐弯处左拐的情况同理
        turn = 'R' if inwall(sta['size'], (myx + direct[(mydire + 1) % 4][0],
                    myy + direct[(mydire + 1) % 4][1])) else 'L'
    stor['strategy']['path'][antiturn[turn]] += 1#记录回家的路径
else:
    turn = turndic[sublst[0]]
    stor['strategy']['path'][antiturn[turn]] += 1
return turn
```

attackto 函数：

attack 模式前往某目标点

```
def attackto(sta,stor):
    myx, myy = sta['now']['me']['x'], sta['now']['me']['y']
    enemyx, enemyy = sta['now']['enemy']['x'], sta['now']['enemy']['y']
    mydire,enmdire = sta['now']['me']['direction'],sta['now']['enemy']['direction']
    aimp=stor['strategy']['endp']#计划攻击的点
    direct = ((1, 0), (0, 1), (-1, 0), (0, -1))
    ori = orient((myx, myy), mydire, aimp)#攻击点在当前位置的大致方位
    direlst,sublst=[],[]#一级搜索方向&次级搜索方向
    turndic={mydire:'F', (mydire - 1) % 4:'L', (mydire + 1) % 4:'R'}#各方向相对当前位置的方位
    # 设定搜索方向的顺序：在前后方向时都是先搜前方，在右边时先搜索右边，在左边时先搜索左边（其余两个方向采取逐渐远离第一方向的顺序）
    searchOrder=(mydire, (mydire - 1) % 4, (mydire + 1) % 4) if (enmdire-mydire)%4==1 else \
        (mydire, (mydire + 1) % 4, (mydire - 1) % 4) if (enmdire-mydire)%4==3 else \
        ((mydire + 1) % 4,mydire, (mydire - 1) % 4) if 'R' in ori else\
        ((mydire - 1) % 4, mydire, (mydire + 1) % 4)
    # 对左前右三个方向搜索可行性
    for dire in searchOrder:
        point = (myx + direct[dire][0], myy + direct[dire][1])#沿某一方向搜索得到的点
        if inwall(sta['size'], point) and sta['now']['bands']\
                [point[0]][point[1]] != sta['now']['me']['id']:#如果该点在地图内且不在我方纸带上
            sublst.append(dire)#则先保存到次级搜索方向
            if turndic[dire] in ori:#如果该方向与攻击点的大致方位一致
                direlst.append(dire)#则直接当作一级搜索方向
# 可能出现头对头攻击
if aimp==(enemyx,enemyy):
    # reladic是攻击点在前面 ，右边的距离
    reladic = (aimp[0] - myx, aimp[1] - myy, myx - aimp[0],
               myy - aimp[1], aimp[0] - myx)[mydire:mydire + 2]
    ppdis=ppdistance((myx, myy), aimp)#我方到攻击点的距离
    # 两者距离为奇数，可以骚操作玩侧碰
    if stor['attackActive']:
        limit = 1 if ppdis==3 else 0
        notOver= reladic[0] not in (1,-1,-2,-3) if ppdis==3 else reladic[0]>0
        #前者为被碰或者将碰上，后者判断是否已经结束比赛即是否已经碰上
        straightF = reladic[1]!=0 if ppdis==3 else True
    # 两者距离为偶数，尽可能防止被侧碰
    else:
        limit=0
        notOver= reladic[0]>0#碰上则比赛结束
        straightF=True
    # 对于不同的方向，有判断其可行的不同标准，存在direEsti中
    direEsti={mydire:notOver and straightF,(mydire - 1) % 4:reladic[1]<-limit,
              (mydire + 1) % 4:reladic[1]>limit}
    for dire in direlst:
        if direEsti[dire]:#如果存储在direlst中的方向可行
            turn=turndic[dire]#直接应用
            break
    else:
        turn=turndic[sublst[0]] if sublst[0]!=mydire else turndic[sublst[1]]\
            if len(sublst)>1 else 'N'#取sublst中不与mydire相同的方向，如果没有则记为N
else:
    turn=turndic[direlst[0]] if direlst else turndic[sublst[0]]#依照优先级选取拐弯方向
return turn
```

rebounder 函数：

刷新边界点（meback 给 True、False，分别表示自己扩回到领地以及敌人回到领地）

```
def reBounder(sta,stor,meback):
    # 自己回到领地，重算边界点
    if meback:
        for point in tuple(stor['bounder']):#遍历原来边界上的所有点
            ncor=ncorner(sta, point)#原来边界点正方形中可以扩张的顶点数
            # point点变为自己领地内或其名消失
            if ncor==0 or sta['now']['fields'][point[0]][point[1]]!=sta['now']['me']['id']:
                stor['bounder'].remove(point)#直接在边界中删除该点
        for point in tuple(stor['bounder']):
            ncor = ncorner(sta, point)
            if ncor==2 or (ncor==1 and nline(sta,point)):
                #有两个可扩张顶点或者有一个可扩张顶点以及至少一个可扩张方向，说明仍在边界上（但必然不是边界顶点，因为边界顶点正方形中一定有三个可扩张顶点）
                for p2 in [(point[0],point[1]+i) for i in range(-4,5) if i!=0]+\
                          [(point[0]+i, point[1]) for i in range(-4, 5) if i!=0]:#搜索point附近的点
                    # 该point离其他记录点过近，且不是顶点，没必要保留，直接删去
                    if inwall(sta['size'],p2) and p2 in stor['bounder']:
                        stor['bounder'].remove(point)
                        break
        # 把刚刚expand模式的记录点即新生成的边界存起来
        for point in stor['unsaveBounder']:
            stor['bounder'].add(point)
        stor['unsaveBounder'].clear()#清空unsaveBounder
```

```
# 敌人回到领地，重算边界点（如果敌人侵占了我方领地那么只有在敌人回到领地之后才生效）
else:
    eaten=False
    for point in tuple(stor['bounder']):#遍历原来的边界点
        if sta['now']['fields'][point[0]][point[1]]!=sta['now']['me']['id']:#如果发现有的边界点不在自己的领地之内了
            eaten=True#就说明这个点被地方侵占了
            stor['bounder'].remove(point)#直接在原来的边界点中删掉这个点
    # 发现有点被敌方吃掉
    if eaten and stor['enemyBand']:
        step = 0
        search = {'x': ((-1, 0), (1, 0)), 'y': ((0, -1), (0, 1)),
                  't': ((-1, -1), (-1, 1), (1, 1), (1, -1)),
                  'start':((i,j) for i in range(-1,2) for j in range(-1,2))}
        # 沿着敌方纸带两侧寻找
        for i in range(len(stor['enemyBand'])-1):
            bandp = stor['enemyBand'][i]#遍历敌方纸带上的点
            if i==0:
                searchdire = 'start'#对于纸带的起始点，根据'start'进行范围搜索
            elif stor['enemyBand'][i-1][0]==stor['enemyBand'][i][0]==stor['enemyBand'][i+1][0]:
                searchdire='x'#如果纸带上连续三个点横坐标相同，说明纸带纵向延伸，进行水平方向搜索
            elif stor['enemyBand'][i-1][1]==stor['enemyBand'][i][1]==stor['enemyBand'][i+1][1]:
                searchdire = 'y'#如果纸带上连续三个点纵坐标相同，说明纸带横向延伸，进行竖直方向搜索
            else:
                searchdire = 't'#对于没有规律的点则搜索四个顶点
            for arg in search[searchdire]:
                point=(bandp[0]+arg[0],bandp[1]+arg[1])#从纸带上某点沿搜索方向得到的点
                # 发现敌方纸带两侧有自己领地即延伸出来的点位于我方领地内
                if inwall(sta['size'],point) and sta['now']['fields']\
                        [point[0]][point[1]]==sta['now']['me']['id']:
                    ncor=ncorner(sta,point)#用于判断这个点是不是我方区域的顶点
                    # 敌方纸带两侧有自己领地的顶点
                    if ncor==3 or (ncor==1 and not nline(sta,point)):
                        stor['bounder'].add(point)
                        step=1#从顶点开始计数
                    # 敌方纸带两侧有自己领地的边界，隔8个点存一次自己的边界点
                    elif step%8==0 and searchdire!='t':
                        stor['bounder'].add(point)
            step+=1#每搜索一个点增加一步
```

pointassess 函数：

评估出发点能扩大多大领地

nowp 自己位置，availp 储存可以的出发策略，dire 出发方向，startp 是从领地扩张的出发位置

endp 领地扩张结束位置，ori 相对方向，step 是扩张步数（单次不转弯）

```
def pointAssess(sta, availp, nowp, dire, startp, endp, ori, step):
    direct = ((1, 0), (0, 1), (-1, 0), (0, -1))
    enemydis = ppdistance(startp, (sta['now']['enemy']['x'], sta['now']['enemy']['y']))#敌人到自己现在位置的距离
    fillcnt=None
    # 出发前的估计
    if abs(endp[0] - nowp[0]) < 4 and abs(endp[1] - nowp[1]) < 4:
        #当前点距离endp足够近，可以认为这两个点都在正方形的一个顶点附近，可以延展出四条边
        #当前位置在endp的相对方向
        enddire = (dire - 1) % 4 if 'L' in ori else (dire + 1) % 4
        p1=(startp[0] + step * direct[dire][0], startp[1] + step * direct[dire][1])#step是一次直线走的总步数，p1是第一个拐点
        p2x= endp[0] + step * direct[enddire][0]#p2是最后一个拐点
        p2x=p2x if 0<=p2x<sta['size'][0] else 0 if p2x<0 else sta['size'][0]-1#首先判断得到的横坐标是不是在地图内，如果不在则就近取到边界
        p2y=endp[1] + step * direct[enddire][1]
        p2y=p2y if 0<=p2y<sta['size'][1] else 0 if p2y<0 else sta['size'][1]-1
        length = step * 4 + 1#一次扩张的最大距离
        if length < enemydis:#确定可以扩张
            fillcnt = fillnum(sta, p1, (p2x, p2y))
    elif 'FF' in ori:#如果endp在出发点的前方
        length = ppdistance(startp, endp) +1
        if length < enemydis:#确定可以扩张
            if dire % 2 == 0:#出发方向平行于x轴的情况
                fillcnt = min(fillnum(sta, startp, (endp[0], min(endp[1] + max(15, step),
                              sta['size'][1] - 1))),fillnum(sta, startp, (endp[0],
                              max(endp[1] - max(15, step), 0))))
            else:
                fillcnt = min(fillnum(sta, startp, (min(endp[0] + max(15, step),
                              sta['size'][0] - 1), endp[1])), fillnum(sta, startp,
                              (max(endp[0] - max(15, step), 0), endp[1])))
    elif ori in ('LF','RF'):#在大致右前方或大致左前方时，可以确保走一段路程可以圈得相当大的面积
        length = ppdistance(startp, endp) + 1#起点和终点分别在正方形的对角
        if length < enemydis:#确定可以扩张
            fillcnt = fillnum(sta, startp, endp)
    else:#其他情况，可以视作起点和终点在正方形的一条边上，因此需要先从起点走到终点的对角
        length = ppdistance((startp[0] + step * direct[dire][0], startp[1]
                             + step * direct[dire][1]),endp) + step + 1#总长度为从起点走到终点的对角再走到终点的距离
        if length < enemydis:#确定可以扩张
            fillcnt = fillnum(sta, (startp[0] + step * direct[dire][0], startp[1] + step * direct[dire][1]), endp)
    if fillcnt:#如果已经算好，那么存储数据
        availp.append({'startp':startp, 'direction':dire, 'endp':endp,
            'efficient': fillcnt / (length+ppdistance(nowp, startp))})
```

expand 函数：

扩张模式

```
def expand(sta,stor,dire=None,endp=None):
    direct=((1,0),(0,1),(-1,0),(0,-1))
    x, y = sta['now']['me']['x'], sta['now']['me']['y']#我当前的位置
    mydire = sta['now']['me']['direction']#我当前的朝向
    # 从别的模式转到扩张模式
    if stor['strategy']['name'] != 'expand':
        stor['strategy']={'name':'expand','endp':endp}#重新布置strategy
        # 当距离敌方太近时，不宜扩张
        if ppdistance((x,y),(sta['now']['enemy']['x'],sta['now']['enemy']['y']))<6:
            stor['strategy'] = {'name': 'expand', 'assessCD': 1}#加一次CD，重新选取扩张出发点
            return shift(sta,stor)#就地内转移
        else:
            if abs(endp[0]-x)<4 and abs(endp[1]-y)<4:#如果快要到达终止点
                stor['strategy']['turn']=2#在保证安全的前提下可以尽量大地扩张
            else:
                ori=orient((x, y), dire, endp)#记录扩张终止点大致方向
                stor['strategy']['turn']=0 if 'FF' in ori else 1 \
                    if 'F' in ori else 2
            stor['strategy']['step']=restep(sta,stor,dire,start=True)#估计扩张所需要的步数
            stor['strategy']['savep'] = {i for i in range(8, stor['strategy']['step'] - 6, 8)}#边上的点每8个存一次
            if (dire-mydire)%4 != 2:#排除dire在mydire正后方的情况，返回dire在mydire的大致方位
                return {0:'N',1:'R',3:'L'}[(dire-mydire)%4]#在正前方的话不用拐弯，因此记为N即None
            # 处理可能dire在mydire正后方的bug，不常见
            else:
                stor['strategy']['step']-=2
                if inwall(sta['size'],(x+direct[mydire][0],y+direct[mydire][1])):
                    stor['strategy']['expandCD']=['R','R','R','L'] if inwall(
                        sta['size'],(x+direct[(mydire-1)%4][0],
                                     y+direct[(mydire-1)%4][1])) else ['L','L','L','R']
                else:
                    stor['strategy']['expandCD'] = ['R','L','L','L'] if inwall(
                        sta['size'], (x + direct[(mydire - 1) % 4][0],
                                      y + direct[(mydire - 1) % 4][1])) else ['L', 'R','R', 'R']
                return stor['strategy']['expandCD'].pop()
    # 之前出现expand的罕见状态，需要CD转向
    elif 'expandCD' in stor['strategy']:
        turn=stor['strategy']['expandCD'].pop()
        if not stor['strategy']['expandCD']:
            del stor['strategy']['expandCD']
        return turn
    else:#处于扩张模式，继续扩张
        endp=stor['strategy']['endp']
        stor['strategy']['step'] -= 1#剩余直行步数
        if stor['strategy']['step']<=0:#直行到指定点
            stor['unsaveBounder'].append((x,y))#记录拐角
            # turn用完，改back模式
            if stor['strategy']['turn']==0:
                turn=back(sta,stor)
            # expand一次转弯
            else:
                stor['strategy']['turn']-=1#剩余拐角数减一
                turn=orient((x,y),mydire,endp)#endp在当前点的大致方位
                # 确定该转向的方向
                if turn[0] not in 'LR':
                    #如果在上一个拐角是右拐，则在不越出地图的前提下尽量右拐，这样可以更快地back，保证安全
                    if stor['lastturn'][0] == 'R':
                        turn = 'R' if inwall(sta['size'], (x + direct[(mydire + 1) % 4][0],
                                                           y + direct[(mydire + 1) % 4][1])) else 'L'
                    #在上一个拐角处左拐的话同理
                    else:
                        turn = 'L' if inwall(sta['size'], (x + direct[(mydire - 1) % 4][0],
                                                           y + direct[(mydire - 1) % 4][1])) else 'R'
                dire=(mydire-1)%4 if turn[0]=='L' else (mydire+1)%4#拐弯方向对应的具体方向
                stor['strategy']['step']=restep(sta,stor,dire)#拐弯之后重新估计扩张所需要的步数
                stor['strategy']['savep'] = {i for i in range(8, stor['strategy']['step'] - 6, 8)}#再次每8个点记录一次
            return turn
        # 继续直行
        else:
            if stor['strategy']['step'] in stor['strategy']['savep']:
                stor['unsaveBounder'].append((x, y))
            return 'N'#直行不拐弯
```

back 函数：

回家模式

```
def back(sta,stor,endp=None):
    x,y=sta['now']['me']['x'],sta['now']['me']['y']#我方位置
    mydire,enmdire=sta['now']['me']['direction'],sta['now']['enemy']['direction']#我方及敌方指向
    enmx,enmy=sta['now']['enemy']['x'],sta['now']['enemy']['y']#敌方位置
    finish=False
    # 从别的模式转到back模式
    if stor['strategy']['name'] != 'back':
        stor['strategy']={'name':'back','path':[0]*4}
        if not endp:#如果没有终止点，则需要寻找终止点（领地中距离自己最近的点）
            endp=fdistance(sta,(x,y),sta['now']['me']['id'],'weak',mydire)[1]
        stor['strategy']['endp']=endp
        xstep,ystep=endp[0]-x,endp[1]-y#终止点与自己的横纵坐标之差
        stor['strategy']['step']=0
        stor['strategy']['path']=[xstep if xstep>=0 else 0,ystep if ystep>=0 else 0,
                                  -xstep if xstep<0 else 0,-ystep if ystep<0 else 0]#分别记录在东西和南北方向上走的距离
        turn=backto(sta,stor,True)
        finish=True
    # 如果离敌方纸带很近，考虑走投无路极限反杀
    elif stor['enemyBand'] and ppdistance((x,y),(enmx,enmy))<12:
        meToField= ppdistance((x,y),stor['strategy']['endp'])#当前位置到终止点的距离
        enmToMe = bdistance((enmx, enmy), enmdire,stor['meBand'], (myx, myy))[0]#敌方击杀我方的最短距离
        meToEnm,enmWeakp = bdistance((myx, myy), mydire,stor['enemyBand'], (enmx, enmy))#我方击杀敌方的最短距离
        enmToField = fdistance(sta, (enmx, enmy), sta['now']['enemy']['id'], 'No')[0]#敌人回领地的最短距离
        if meToField<enmToMe and meToEnm<enmToField:#如果我方可以在被击杀前回领地或者可以在敌方回领地之前将其击杀
            turn=attack(sta,stor,enmWeakp)#进入攻击模式
            finish=True
    # 原来的返回目的地被占领，大喊一声'fxxk'，然后重算新目的地
    if not finish:
        if sta['now']['fields'][stor['strategy']['endp'][0]]\
                [stor['strategy']['endp'][1]]!=sta['now']['me']['id']:
            stor['strategy']={'name':'fxxk'}
            turn=back(sta,stor)
        # 继续back模式
        else:
            stor['strategy']['path'][mydire]-=1#回到领地的路径数减少
            stor['strategy']['step']+=1#步数不断增加
            if stor['strategy']['step']%8==0:#每8步记录一次边界
                stor['unsaveBounder'].append((x, y))
            turn=backto(sta,stor)
    return turn
```

shift 函数：

领地内转移 shift

```
def shift(sta,stor):
    x, y = sta['now']['me']['x'], sta['now']['me']['y']
    start=False
    # 刚回到领地，找下一个出发点，预估下次的最佳扩张路线
    if stor['strategy']['name']!='shift':
        nextone=routeAssess(sta,stor)#选取可以使扩张范围较大的扩张路线（包含扩张起点，扩张终点，扩张方向等）
        if nextone:#如果已经找到下一个扩张点
            stor['strategy']= {'name':'shift','nextstart':nextone['startp'],
                'nextend':nextone['endp'],'expanddire':nextone['direction']}#将这些数据存入strategy中
            start=True
        else:
            stor['strategy'] = {'name':'shift','assessCD':5}# 针对routeAssess没有输出的，给一个5轮的CD冷却时间
    if 'assessCD' in stor['strategy']:
        # 5轮的CD结束，之前找不到合适出发点时积聚的怒气爆发出来，
        # 大喊一声'fxxk'，然后重新计算下一个扩张出发点
        if stor['strategy']['assessCD']==0:#CD时间结束
            stor['strategy']={'name':'fxxk'}
            return shift(sta,stor)#开始找出发点
        else:
            stor['strategy']['assessCD']-=1#CD时间缩短
            return shiftto(sta,stor)#在进行CD冷却时进行漫游
    elif stor['strategy']['nextstart']==(x,y):#如果已经到达扩张出发点
        nextp=stor['strategy']['nextend']#扩张终止点
        dire=stor['strategy']['expanddire']#扩张方向
        turn=expand(sta,stor,dire,nextp)#扩张时的拐点
        return turn
    # 可能计划点已被占领
    else:
        startp=stor['strategy']['nextstart']
        # 计划点被占领，怒吼一声'fxxk'然后重新找点
        if sta['now']['fields'][startp[0]][startp[1]]!=sta['now']['me']['id']:
            stor['strategy']={'name':'fxxk'}
            return shift(sta,stor)
        # 继续在领地内shift
        return shiftto(sta,stor,start)
```

attack 函数：

攻击敌人模式

```
def attack(sta,stor,endp=None):
    myx, myy = sta['now']['me']['x'], sta['now']['me']['y']
    enmx,enmy=sta['now']['enemy']['x'], sta['now']['enemy']['y']
    mydire,enmdire = sta['now']['me']['direction'],sta['now']['enemy']['direction']
    # 从别的模式转到attack模式
    if stor['strategy']['name'] != 'attack':
        if not endp:#没有攻击点时需要先找
            endp=bdistance((myx,myy),mydire,stor['enemyBand'],(enmx,enmy))
        stor['strategy']={'name':'attack','endp':endp}
        turn = attackto(sta,stor)#进入攻击
    # 敌人成功逃脱（出现这个的概率不高，但还是准备）
    elif sta['now']['fields'][enmx][enmy]==sta['now']['enemy']['id']:
        turn = back(sta,stor)#如果敌人逃脱则直接返回我方领地
    # 继续攻击
    else:
        endp=min(stor['strategy']['endp'],stor['enemyBand'][-1],
                 key=lambda p:ppdistance((myx,myy),p))#从原来的攻击点、敌方纸带末端、敌方纸卷选取最近点进行攻击
        stor['strategy']['endp']=endp
        turn= attackto(sta,stor)
    return turn
```

startmode 函数：

开始模式，只在一开始执行

```
def startmode(sta,stor):
    x, y = sta['now']['me']['x'], sta['now']['me']['y']
    mydire = sta['now']['me']['direction']
    stor['attackActive'] = ppdistance((x,y),(sta['now']['enemy']['x'],
                                           sta['now']['enemy']['y']))%2==1
    #判断我方纸卷到敌方纸卷的距离是否为奇数，用于attack函数中倒碰的情况
    for item in ((x - 1, y - 1), (x - 1, y + 1), (x + 1, y + 1), (x + 1, y - 1)):
        stor['bounder'].add(item)#确定初始边界点
    ori = orient((x, y), mydire, (sta['now']['enemy']['x'], sta['now']['enemy']['y']))#记录此时敌方的大致方位
    if 'L' in ori:
        nextp = [point for point in stor['bounder']\
                 if orient((x, y), mydire, point) == 'RF'][0]#当敌方在左边时，往右前方扩张
    else:
        nextp = [point for point in stor['bounder']\
                 if orient((x, y), mydire, point) == 'LF'][0]#当敌方在右边时，往左前方扩张
    stor['strategy'] = {'name': 'shift', 'nextstart': nextp}
    turn = shiftto(sta, stor)#寻找扩张起始点
    antidire = {'L': -1, 'N': 0, 'F': 0, 'R': 1}#方向改变量
    stor['strategy']['expanddire'] = (mydire + antidire[turn]) % 4#扩张方向=初始方向+方向改变量
    stor['strategy']['nextend'] = [p for p in stor['bounder']
            if orient(nextp, stor['strategy']['expanddire'], p)[1] != 'B'][0]#取位于nextp前方一定角度范围内的点作为扩张终止点
    return turn
```

主程序里变量名简化：

```
# 变量名简化
stratedic={'expand':expand,'back':back,'shift':shift,'attack':attack,'start':startmode}
myx,myy=stat['now']['me']['x'],stat['now']['me']['y']
mydire = stat['now']['me']['direction']
enemyx,enemyy=stat['now']['enemy']['x'],stat['now']['enemy']['y']
needShift=False
# 刷新敌人的纸带
if stat['now']['fields'][enemyx][enemyy]!=stat['now']['enemy']['id']:
    storage['enemyBand'].append((enemyx,enemyy))
# 敌人回到领地，检查自己领地是否有保存点被吃掉
elif storage['enemyBand']:
    storage['enemyBand'].append((enemyx, enemyy))
    reBounder(stat,storage,False)
    storage['enemyBand'].clear()
# 刷新自己的纸带
if stat['now']['fields'][myx][myy]!=stat['now']['me']['id']:
    storage['meBand'].append((myx,myy))
# 自己回到领地，更新自己领地边界，如果不attack则要领地内shift
elif storage['meBand']:
    storage['meBand'].clear()
    storage['unsaveBounder'].append((myx, myy))
    reBounder(stat, storage, True)
    storage['strategy']={'name':'Yes'}
    needShift=True
```

```
# back和attack模式有最高优先级
if storage['strategy']['name'] in ('back','attack'):
    turn = stratedic[storage['strategy']['name']](stat, storage)
else:
    finish = False
    enmToField,enmBackp = None,None
    # 检查是否自己有危险，可能要临时返回
    if storage['meBand']:
        meToField,backEndp=fdistance(stat,(myx,myy),
            stat['now']['me']['id'],'weak',mydire)
        enmToMe=min(bdistance((enemyx,enemyy),stat['now']['enemy']['direction'],
                storage['meBand'],(myx,myy))[0],ppdistance((enemyx,enemyy),backEndp))
        #敌人杀死我方的最短距离（直接攻击纸带或者在我方扩张终止点拦截）
        # 简单判断中途拦截可能性
        if myx<enemyx<backEndp[0] or backEndp[0]<enemyx<myx:#若水平方向上敌方在我方纸卷和扩张终止点之间
            enmToMe=min(enmToMe,max(abs(enemyy-myy),abs(enemyy-backEndp[1]))-1)#则沿竖直方向拦截
        if myy<enemyy<backEndp[1] or backEndp[1]<enemyy<myy:#若竖直方向上敌方在我方纸卷和扩张终止点之间
            enmToMe=min(enmToMe,max(abs(enemyx-myx),abs(enemyx-backEndp[0]))-1)#则沿水平方向拦截
        if meToField+2>=enmToMe:#发现有被拦截的可能
            turn=back(stat,storage,backEndp)#立即返回
            finish = True
        # 防止敌人扩充领地使自己离领地距离突然变远
        elif storage['enemyBand'] and fdistance(stat, backEndp,
                    stat['now']['enemy']['id'], 'No', maxdeep=4)[0] < 3:#发现敌方有侵占我方领地的可能
                enmToField, enmBackp = fdistance(stat, (enemyx, enemyy),
                                                 stat['now']['enemy']['id'], 'No')
                if ppdistance(enmBackp,backEndp)<30 and\
                        meToField + 3 >= enmToField:#发现敌方可能比我方先扩充完领地
                    turn = back(stat, storage, backEndp)#立即返回
                    finish = True
    # 检查敌方是否暴露弱点可以进攻
    if not finish and storage['enemyBand']:
        if enmToField is None:
            enmToField,enmBackp=fdistance(stat,(enemyx,enemyy),
                                          stat['now']['enemy']['id'],'No')#敌方回领地的最短距离
        meToEnm,enmWeakp = bdistance((myx, myy), mydire,
                                     storage['enemyBand'], (enemyx, enemyy))#我方攻击敌方纸带的最短距离
        if meToEnm<enmToField:#发现在地方回到领地之前我方可以攻击到敌方纸带
            turn=attack(stat,storage,enmWeakp)#直接攻击
            finish = True
        # 敌方在自己领地，尝试拦截
        elif (not storage['meBand']) and stat['now']['fields']\
                [enemyx][enemyy]==stat['now']['me']['id']:
            meToEnm=min(meToEnm,ppdistance((myx,myy),enmBackp))#最短拦截距离
            #原理与敌方拦截我方相同
            if enemyx < myx < enmBackp[0] or enmBackp[0] < myx < enemyx:
                meToEnm = min(meToEnm, (abs(myy - enemyy)+ abs(myy - enmBackp[1]))//2)
            if enemyy < myy < enmBackp[1] or enmBackp[1] < myy < enemyy:
                meToEnm = min(meToEnm, (abs(myx - enemyx)+ abs(myx - enmBackp[0]))//2)
            if meToEnm+1<enmToField:#发现敌方回到领地之前可以成功拦截
                turn = attack(stat, storage, enmWeakp)#直接攻击
                finish = True
    # 以上都不成立，照常活动
    if not finish:
        # 刚回到自己领地，要shift
        if needShift:
            turn=shift(stat,storage)
        else:
            turn = stratedic[storage['strategy']['name']](stat, storage)
if turn[0] in ('L', 'R'):
    storage['lastturn'] = turn
return turn
```

load 函数：

```
def load(stat,storage):
    # storage['bounder']记录重要的自己领地边界坐标
    storage['bounder']=set()
    # 记录当前策略及信息
    storage['strategy']={'name':'start'}
    # 记录上次转向的方向
    storage['lastturn']='S'
    # 记录未储存的记录节点（边界点），为expand和back模式暂存的
    storage['unsaveBounder']=[]
    # 记录自己和敌方的纸带
    storage['enemyBand'] = []
    storage['meBand'] = []
    # 判断自己攻击应主动还是被动，如果两者相距为奇数则可主动
    storage['attackActive'] = False
```

3 实验结果

3.1 测试数据

一. 实验环境说明：

硬件配置：Inter(R) Core(TM) i5-8250U CPU @ 1.60Hz 1.80GHz 12.0GB

操作系统：window 10 家庭版

Python 版本：3.6.1

二. 测试方法：

1. 参与热身赛进行实战测试

2. 与其他组间进行小范围友谊赛，进行测试

3. 利用所提供的算法就行可行性检验

4. 将改进后的算法与改进前算法对战

3.2 结果分析

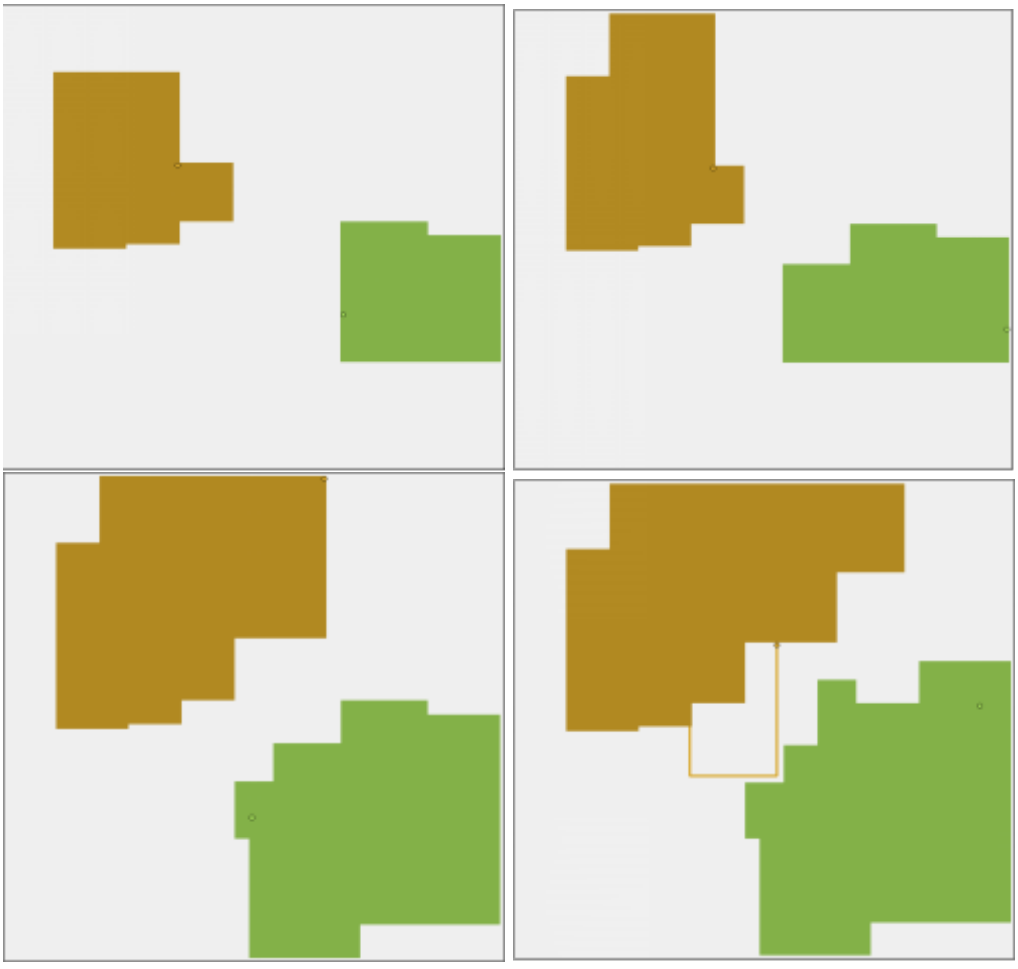
由于我们根据可视化复盘分析了对手的扩张方式、进攻方式、防守方式、各项消耗，在热身赛、实战中，我们发现某些算法之间可能存在相克的情况，故针对不同的战术风格，我们都编写了代码，采用多种战术，在代码中最大程度上体现了兼容性。这样的思想让我

们能够较好应对不同的对手。

3.3 经典战局

我们在比赛前与其他组进行了小范围的友谊赛，来互相改进各自的代码，以在最终的比赛中都拿到好成绩，我们拿来与自己扩张方式风格迥异的一方的比赛作为经典战局进行了分析。当然我们也拿自己更新后的代码与自己之前地代码进行了比赛，其中改变最显著的要数加入进攻之前与加入进攻之后的比赛，我们也拿来其中一场作为经典战局进行分析。

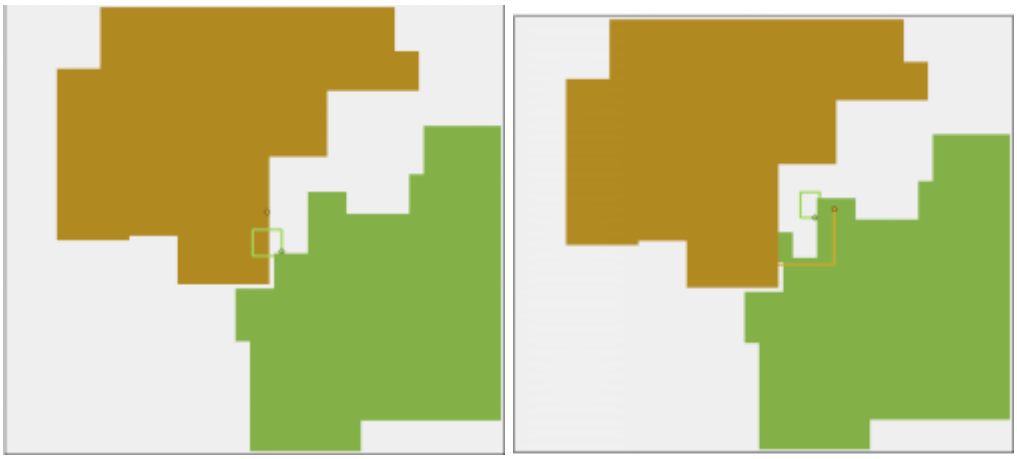
（1）代码改进测试: 加入进攻战术之前与加入进攻战术之后的经典战局（棕色为加入进攻战术之前的一方，绿色为加入进攻战术之后的一方）



由于除进攻战术的添加之外的其他部分战术并没有太大改动，所以开始一段时间内棕

绿双方圈地面积一直处于相似状态，而且双方都在自己的角落里静静地进行着领地地扩张。进攻战术当然只是辅助，只有在有机会一举杀掉对方时才会启动，所以开始的阶段中绿色方也没有进攻的迹象。

其中有一段，棕色一方竟试图主动往绿色方领地扩张，但也只是挑衅行为，并未真正进入绿色方领地。

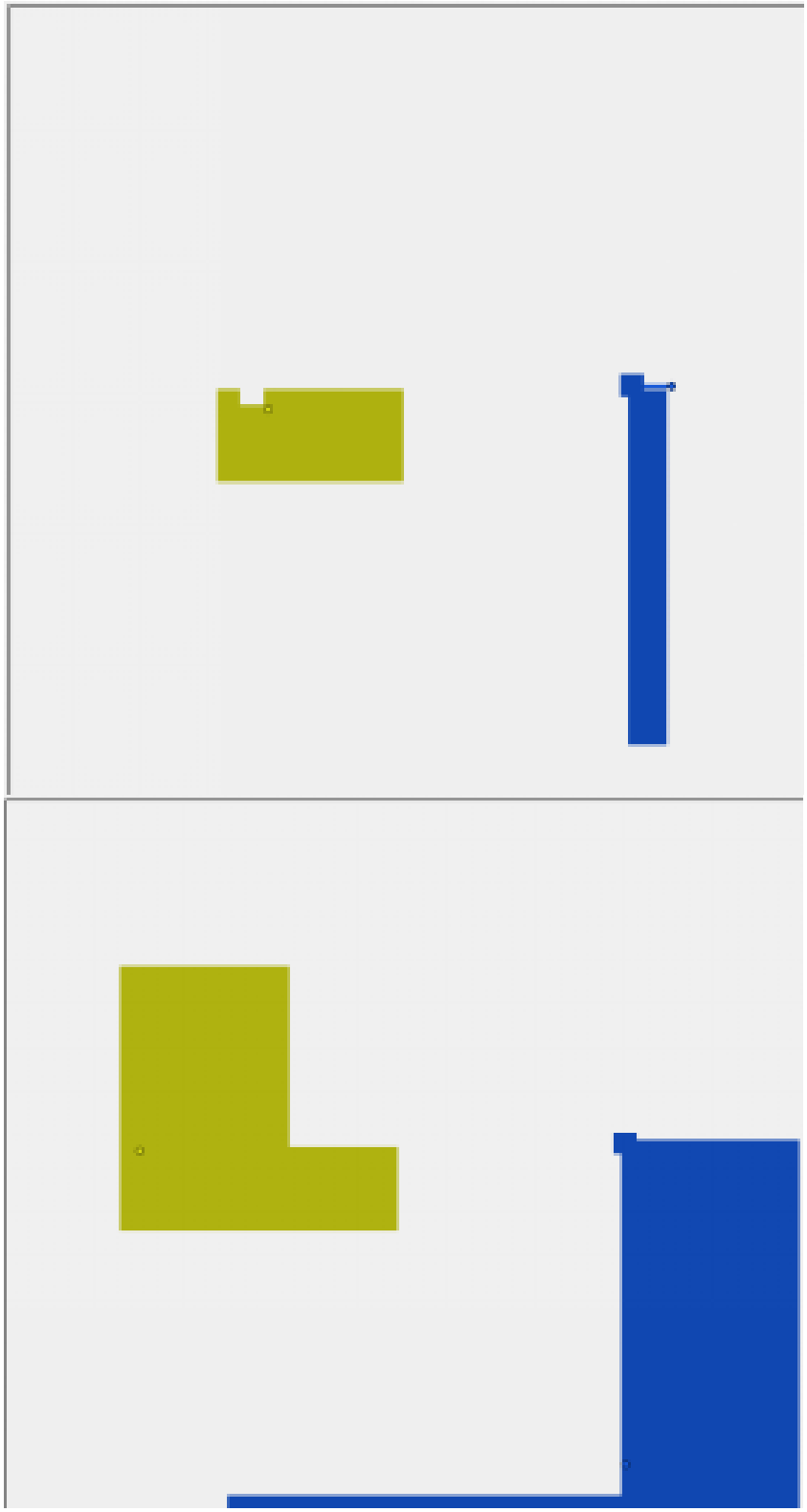


接着，果然是绿色方首先开启进攻状态，进入棕色方领地扩张。由于本来已有的防守战术，棕色方进入防守状态。不过绿色方进攻也是十分灵敏，一察觉对方已经向自己走来，马上回到自己领地，解除了危机。

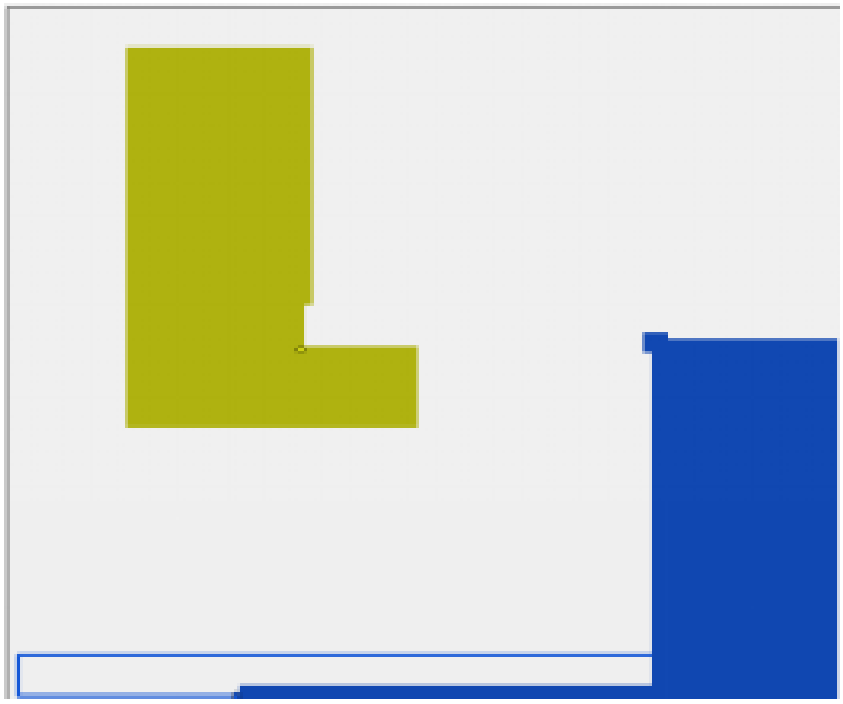
绿色方在原来位置未占到很大便宜，就换了一个位置继续进行扩张，显然棕色一方毫无危机意识，依旧大胆向绿色方领地扩张。不过得益于绿色一方优秀的进攻战术，一发现对方进入自己领地便果断回头转入进攻状态，一举击杀对方，获得胜利。

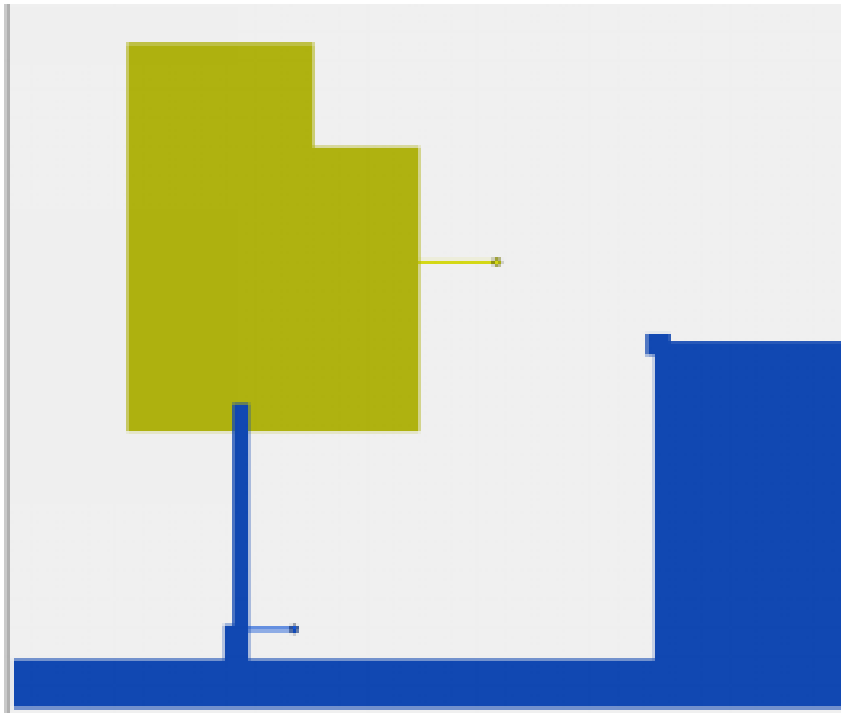
毫无疑问，进攻战术的加入大大提升了战斗力，在该进攻时能果断出击，击杀对方，获得胜利。

（2）我方保守与对方激进的经典对局（绿色为我方，蓝色为对方）



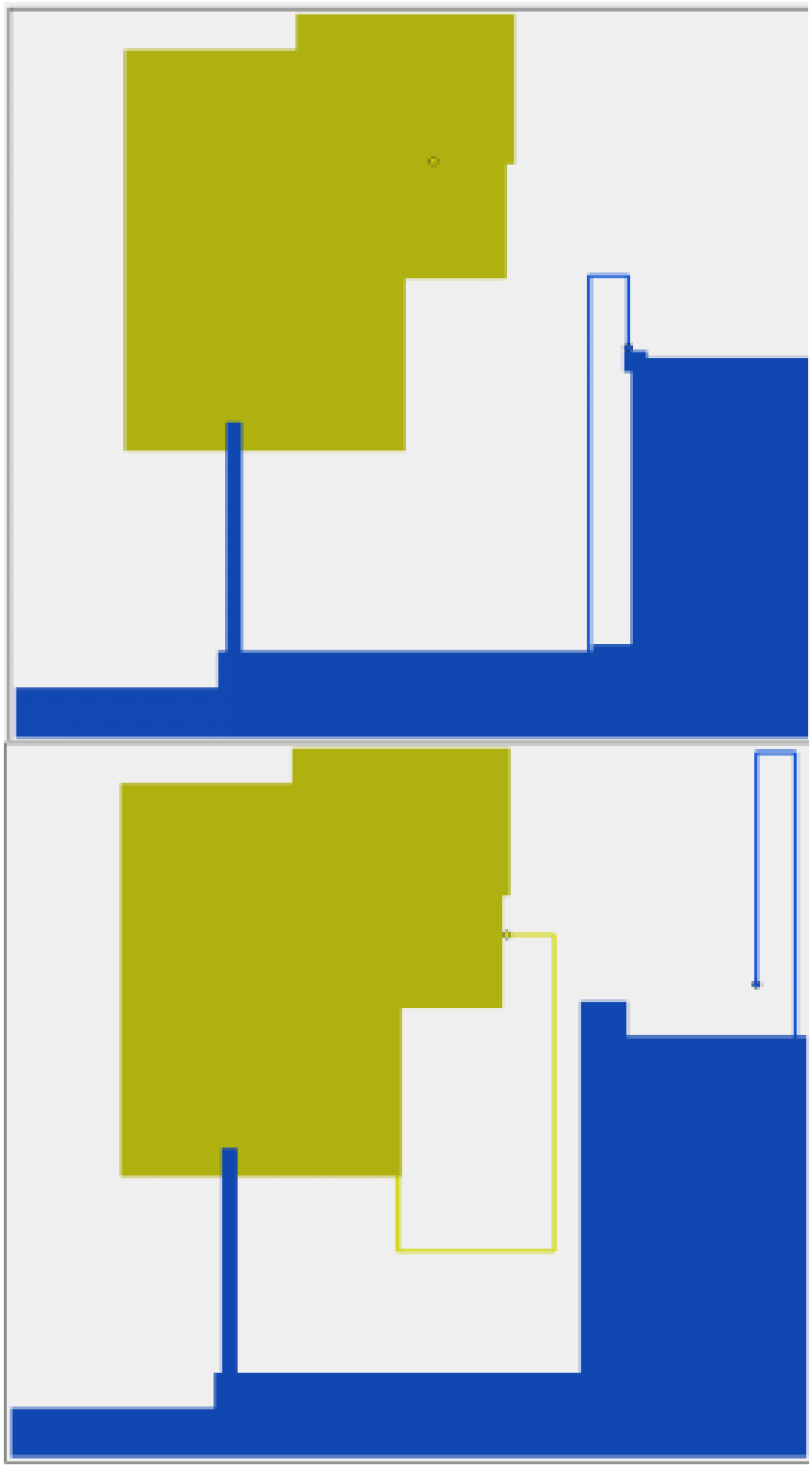
一进入比赛，双方就表现出很大的不同：我方起始的圈地策略就相对比较保守，只进行往自己周围一圈领地辐射型扩张，而对方相对激进的圈地策略刚开始就进行大肆向下增长。我方圈地面积在开始稍占了一点优势，但根据对方的圈地策略可能会在接下来有更好的发展。





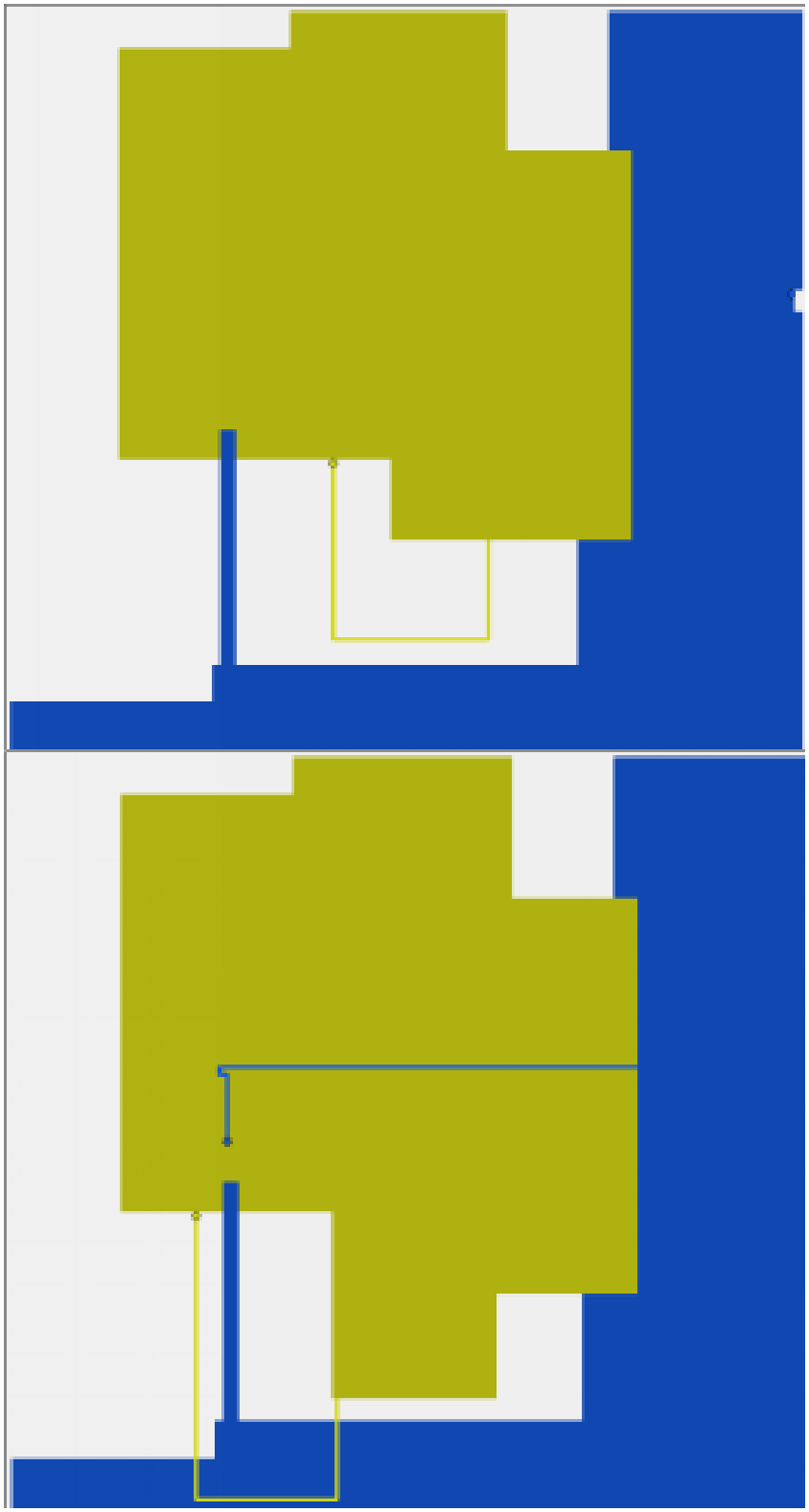
紧接着我方同样延续保守的策略，虽然效果不是很好，但是十分安全，应该是由于对方在下方扩展领地的原因，有意向上方继续扩展领地，而先不与对方进行正面的交锋；对方则将自己的领地扩展到了右边边境，同时用很窄的带状扩展悄悄在下面紧紧贴着边境扩展。

紧接着，我方在向上方不断扩展的同时，试图将领地向同样未被占领的右上方发展；此时对方就开始在我方已有领地下方进行扩张，同样使用几乎条状的方式将自己领地向上扩张，并将延伸到与右方已有领地几乎等高的地方，可以推测下一步对方应该是准备将这块条状区域与右方已有区域进行连接，可以说是十分有效快速的扩张领地的方式，但不得不说十分危险。



应该是察觉到我方不断在右上方扩张的态势，对方采取了回到自己右上方进行防守的策略，而我方也悄悄改变扩张的方向，往我方下面进行扩张，一直在躲着对方，十分安全是一定的了，但也显示出我方进攻的薄弱，一直只是在进行躲避和被动防守。

在这几轮的切磋中双方领地面积几乎没什么差别，不是你稍稍领先于我，就是我稍稍领先于你，几乎不能预测后面的发展局势。可以看到我方已经在对方领地进行着小小的扩张，而对方已经占领了右侧的所有边界，势必要改变扩张方式，要么向左面我方已有领地进行扩张，要么重回下面，利用刚刚在下面的优势进行继续扩张。不得不说刚刚对方在下面的扩张对我方是一个很大的威胁，而且这个威胁一直没有解除。



然后，或许是因为对方有些强迫症的原因，必须将右面领地全部一格不剩的占完才肯罢休，而且出现了失误，一直在一格一格的进行着占领，也耗费了很多时间。我方也充分对方的失误，在圈地领地上与对方拉开了不小的差距。

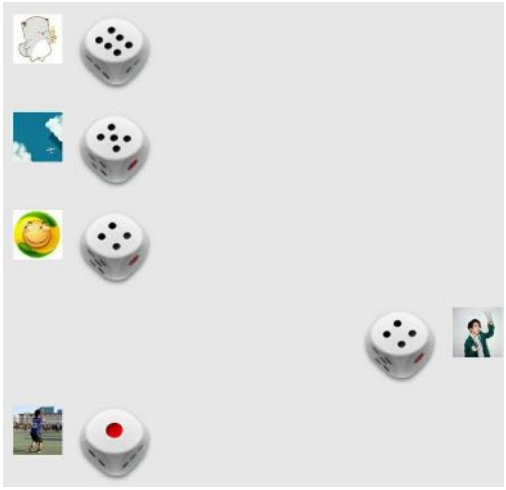
最后便进入了比赛最高潮部分：我方也许是察觉到了对方刚刚留下的危险，便毫不犹豫地将我方领地向下扩展，试图将那一块包含；而此时对方也试图到达那个地方，完成刚刚自己留下的残局。但就差那么几步，我方成功地将刚刚那块领地收入自己囊中，对方根本来不及掉头，便被我方撞击纸带而死，我方获得最终的胜利。虽然我方在这个经典战局取得了最终的胜利成功进行拦截，但也暴露了我方扩展的过于保守，可以在保守扩张与激进扩张间取得平横，或有机地在两种扩张方式间灵活变化，应该会取得更好的领地扩张速度。同时进攻相对较于薄弱，如果与进攻强势的敌人进行比赛的话，根本来不及躲避就会被撞击纸带而死。

4 实习过程总结

4.1 分工与合作

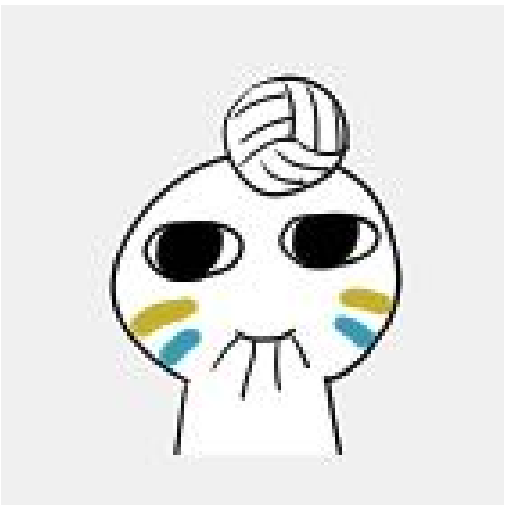
a.5 月 22 日周二——选组长，选头像

鉴于组长并没有很明显的作用，且我们小组的成员都比较谦让，所以决定以微信投骰子作为标准，数字最大的为官方组长，于是最后选定刘煜杭同学称为我队组长。



同时，由于大家都是地空排球队成员，大家很快决定选取和排球有关的照片作为头像，很快决定选择某一排球表情包。但是在但浩文和刘煜杭的“恶搞”之下，上交了小泽学长的微信头像作为本组头像。也代表了我们队队小泽学长的尊重和欢迎!!

（该照片为北大杯小泽学长发球的照片，地空排球队的男队和女队都在今年北大杯取得了第四名，无缘前三甲，尴尬的是，我们的代码也在进入四强后惨遭淘汰没能进入前三甲，这也是地空排球队今年的第三块铁牌。）



b.5 月 23 日—5 月 26 日周六——仔细研究 paper.io 以及任务需求

大家详细阅读老师发布的大作业说明以及要求，并且可以有自己的一些想法，这样也能使讨论时更加流畅。在准备过程中，大家也充分地了解了这个游戏，进行了很多人机对抗和玩家对抗，总结出了一些制胜经验。

c.5 月 27 日周日——第一次小组讨论，制定初步计划

（第一次讨论我们借了教室，在黑板上进行了激烈的作画讨论，但是遗憾的是我们并没有拍照，只有借教室的记录如下）

大家主要就我们应该采取的模式进行了热烈讨论。

首先讨论的是最热门的围城算法，即通过极好的轮回防守绕场地一圈赢得胜利。大家在黑板上作图讲解分析，并考虑时间、步数以及场地大小等因素，发现完全围城在 2000 步内实现有些困难，而且容易被对方阻断而导致失败，所以围城算法可能是很多组都会考虑但是不怎么会采用的方法。但是，我们也将此算法保留，需要进行某些克服此类算法的策

略，以及可以对此算法进行改进（比如不围住所有场地而是尽可能绕大圈）或者可以在某种特定情况下采取这种策略。

之后大家很快毙掉了蚊香算法，即像蚊香一样向四处扩散。虽然防守力很强代码简单，但是效率低下，圈地很少，难以胜利。

最后是普通圈地法，当然就是通过不断在领地周围圈方形区域加大自己的面积。

我们提出的大致思路有：

1. 计算剩余回合数，保证最后一次能圈地，不会停在中间，这样可以避免最后一次

圈地步数浪费，特别是最后一次如果圈地过大。

2. 计算到达敌方的距离与地方能圈地的最短距离进行进攻 and 计算自己能圈地的最短距离与地方能进攻的最短距离进行防守（即快速返回到自己最近的领地中）。这四个距离当然也是最基础的想法，以此可以很好的防御以及进行很好地进攻。当然这四个距离的计算还是需要不同的想法以及函数才能实现。

3. 初步认为我们要以防守为主，进攻为辅，而且胜负的关键应该在于圈地的大小上，所以要将圈地函数多进行研究，必要时可以采取多种战略以应对不同情况。

最后大家决定每个人先继续想出策略并给出能够实行这个策略的基本写代码思路，这样大家就可以在讨论合理性之后完成代码的编写。

d.6 月 3 日周日——小组第二次讨论，分析代码，提出进一步方案

虽然中间一段时间大家忙于大家的事情，对于大作业没有怎么上心，但是组里的胡俊杰不知不觉就写完了自己的一套代码，也让大家感到十分开心。因为避免重复写代码，大家约在了理教二层进行了讨论。讨论内容包括：

1. 由于胡俊杰并没有写代码注释，以至于很难看懂，所以大家听他从头到尾讲了代码的大致思路，并对有些内容提出了见解与讨论。

2. 大家决定分工对胡俊杰的代码写算法分析，这样既能让大家弄懂代码，同时也是报告的任务之一，并决定在算法分析写完后，分成两组人，一组进行代码的实验、分析，另一组对代码功能的强化以及修正 bug。

e.6 月 3 日——6 月 9 日——探讨代码并提出意见

不得不说，看别人的代码真是一个艰辛的过程，不过谁叫我们没有写代码呢。

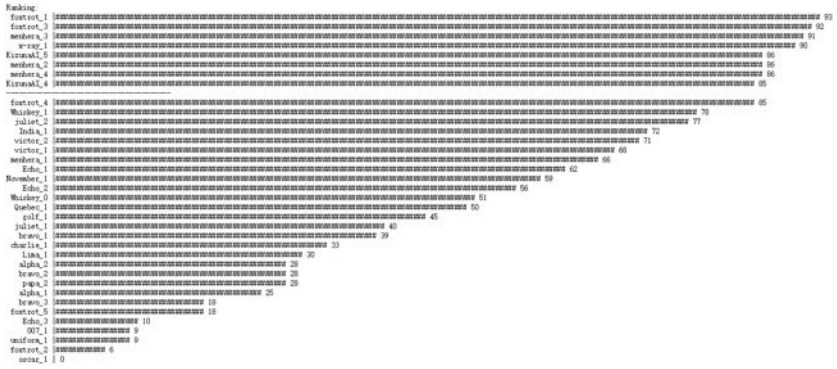
胡俊杰时间有限，所以没有写注释也让大家能够理解，虽然他补注释很快，但我们也觉得有必要看他的代码，所以坚决分工加注释。

当然，其间有很多看不懂的地方，但是在和胡俊杰探讨之后就能知道他的具体意思，也能看到代码的精妙之出。

大家分析代码的时间花了很长，以至于准备看完代码一起研究进攻函数的时候，胡俊杰已经又开始自己写攻击函数了，大家只能在过程中提出一些意见，也感叹胡俊杰写代码之神速，大家也都轻松了不少，因为热身赛临近。

f.6 月 9 日——参加热身赛

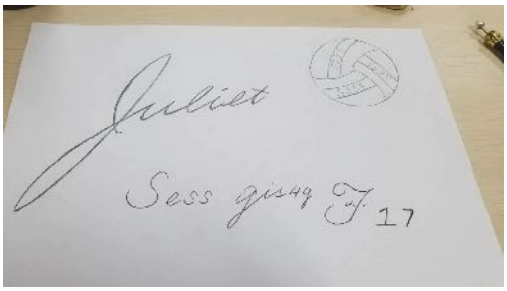
大家抱着试一试的态度将添加攻击函数的代码以及原代码上传参加了热身赛，最后激动地发现，虽然我们总体在靠前的位置，而且参加热身赛的本区小组中，我们排名第一，所以大家对之后的小组出线也有了很大的信心。

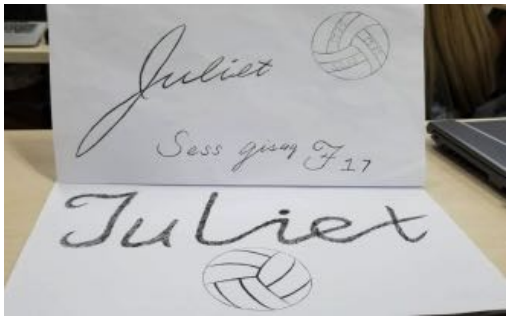


上图中 Juliet2 是添加的攻击代码的函数，同时圈地效率有了很大的提高。Juliet1 则是未改之前的代码，排名靠中。虽然我们是小组第一，但是其实发现 India 组紧次于我们，所以并不能掉以轻心。

g.6 月 12 日——正式比赛

提前一节课，刘煜杭和但浩文提前来到了教室，分别制作了队牌。



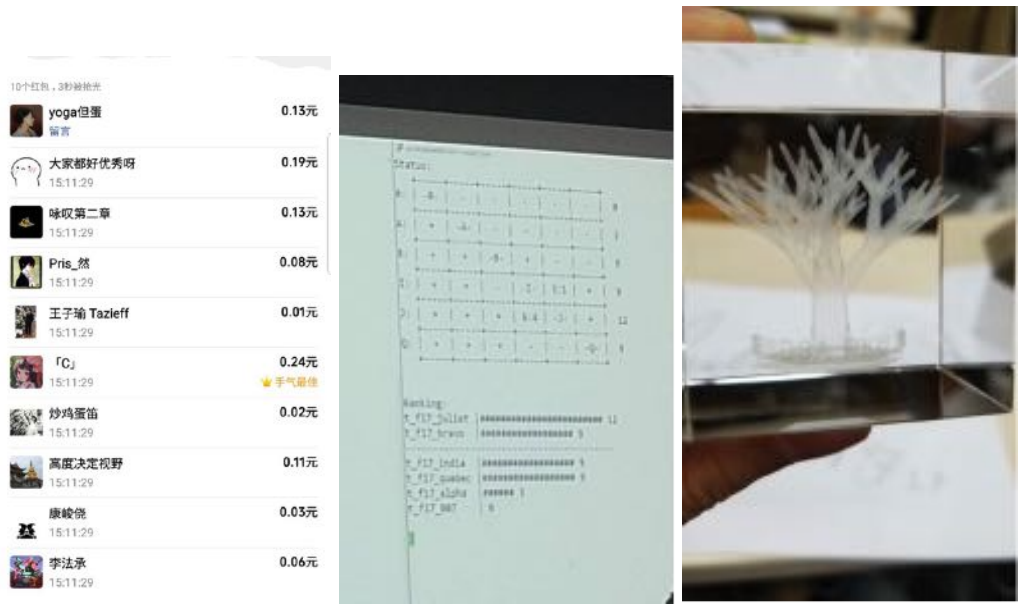


我们在队牌上加入了 gis4g 和排球等元素，总之手绘很好看!!!

很快就要正式比赛了，大家的心情都极其紧张，但是老师很快进行了抽奖环节，我们对组的但浩文因为手速成功抢到了老师提供的充满儿童回忆的大大。

同时，我们组的但浩文和胡俊杰因为预测出线小组成功获得了奖品（奖品前三甲队员才能有的奖品，也弥补了我们并没有进入前三甲的遗憾）

小组赛中，我们分区是比赛的最后分区，也让我们更加焦急，不过很快我们就战胜了四个组获得 12 分，基本稳住出线名额，一切只看我们和 India 的对抗，但遗憾的是，在很长时间的对抗后，我们输给了 India, 这一结果也给了 India 以出线名额，最后双方都积 12 分出线，但我们组平均圈地面积更大，夺得了组一的好成绩，大家十分激动。



赛前抽奖比赛过程竞猜奖品

在四强赛中，我们开局落后，却在最后连续获胜，最后以 11:7 战胜了对手，锁定四强。

很遗憾，在之后的比赛中我们都没有发挥优势，最后取得了第四名的成绩，但大家对这个结果也还是很满意的。

India 组长在赛后帮我们拍的大合照

4.2 经验与教训

胡俊杰：看到小组的代码一步步完善起来，作为框架搭建者的我也非常欣慰。不过调试过程中曾出现过许多 bug，比如各种报错 Error，撞墙撞纸带什么的，一直到比赛前还有一些策略性 bug 未能解决，比如选取返回目标、出发目标等。原来没有添加 attack 策略时以扩张为主，而后加了攻击之后攻击力越来越强，感觉也信心满满的。

关于热身赛，在最后一次热身赛时取得了不错的成绩，并且击败 I 组，不过观看 I 组复盘时发现其潜力依然很强，肯定是小组赛的重点针对目标。虽然正式比赛时输给了 I 组，不过我们凭借优秀的扩张底子，顺利拿到组一。本来我们组设立的目标就是小组出线，这次组一让我们看到了拿牌的希望。不过我一直毒奶，肯定拿不了牌的，别想那么多，结果便也就是刚好的铁牌（噫）。回想起来，原初设立的出线目标已经达成，不管是刚刚出线，还是铁牌，抑或是真的拿牌了，于我们之心也不是很大遗憾罢。

但浩文：有点惭愧在代码上没有帮很大的忙，但是最后比赛那天真的是运气爆棚抽到了好多奖，小组的代码也成功进入四强，也是非常开心吧。一开始是很想参加写代码的，第一次讨论思路时自己其实已经考虑了很多，也想到了一些实现的策略，但是胡俊杰大佬已经开始写还写了很多，为了避免重复就想先研究一下他的再写，可是后面考试多事情也多，慢慢地就越来越没有时间写。在此觉得也算是一个经验吧。首先组里还是要明确一些，有人指挥才能更好地进行，不然大家各自准备会没有效率，所以应该充分发挥组长的作用。所以觉得一开始选的名义组长什么的有点随意，应该定一个完全的组长进行组织会比较好。其次是分工方面，确实应该早点分工早点有任务和目标，这样也能提高整体效率，如果这次没有胡俊杰写代码如此积极的话，我们可能也会像很多别的组一样，在决赛前集体肝很多个晚上。最后是在代码方面，如果几个人都有想法，可以写出两个代码进行更新。

总之这次的大作业是可以做到更好的，也可以为以后的很多工作提供经验。能和球队伙伴们一起参加这次大作业是非常开心的，对这次的成绩也是很满意了，希望明年的数算课会有更多优秀的作品出现！

4.3 建议与设想

1. 建议能开始冠军发言环节，让冠军组的同学在台上简述他们组的代码设想，也能达到大家相互学习的作用。

2. 希望明年大作业最后的评比能区别一下四强和八强，因为我们组最后获得了第四名，虽然我们更有成就感但在奖品，报告排序以及成绩上都是和 5-8 名一模一样的。建议给第四名发个殿军奖，或者能在最后总结报告中开设四强席位。

3. 建议可以开始创意征集，比如今年的最后选题 paper.io 源自一小组的 micro-bit 创意展示，可以给这个小组进行适当的奖励和加分，明年可以进行一个创意征集活动，让大家积极参与，并且能够调动积极性。

5 致谢

感谢陈斌老师以及技术组的油聚以及助教能为我们提供很多基础模块，提供对战平台以及帮我们组织热身赛进行学习。

同时特别感谢 Juliet 组里的胡俊杰同学对代码的编写有卓越的贡献。

最后感谢所有组员的认真配合以及地空排球队的祝福。

6 参考文献

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

第十二章　F17_KizunaAI 报告

姜金廷 * 牟星名 麻一凡 李玥阳

摘要：本组算法采用的是局部最优原理，每走一步都对当前局势进行一次评估，从评估的局势来进行下一步的决策。算法采取的数据结构有栈和列表，主体为线性算法，未涉及非线性结构。实验数据结果为小组未出线。

关键字：固定参数局部最优列表字典栈

1 算法思想

1.1 总体思路

本次纸袋圈地游戏的获胜条件有多种，如采用攻击性的代码来撞击对方纸袋，或是靠可走步数耗尽后的圈地面积大小来决定哪方获胜，或诱导对方纸袋在自身领域内撞击自己而使自己获胜。而我组的算法想要实现的功能是能够综合各种情况，找出执行各种策略的判断条件并以较优的方法来执行此策略，从而实现局部最优。

本组的算法总体思路是每走一步就对局势进行一次评估，按照经大量实验调试后依照经验确定胜率最高的动态参数（确定圈地时向外走的步数的参数 expand_length 与判断局势是否需要撤退的参数 safe）来决定下一步的策略。开场时，先使用 get_out 函数使我方离开初始九宫格，到达边界后再进行下一步策略。每一步的策略有三种：攻击（attack）、撤退（retreat）与圈地（enclose），其中 attack 函数选择最快接近敌方纸带，并尽量减少离己方领地的距离的走法，尽可能保证攻击的效率，同时能够达到向己方领地撤退的目的。而 retreat 函数从使我方缩短最近距离，且远离敌方的方向撤退，也是为了尽可能保证撤退的效率。而 enclose 函数的实现方法是事先生成并保存圈地的路径，并在之后的每一步中判断是否需要 attack 或 retreat，若不用进行这两种策略就继续从事先保存的圈地函数 enclose 生成的路径 path 中一步步调用，来保证局部最优的圈地与攻击、撤退策略的切换。而本算法所采取的节约时间的方法是建立一个 store 字典，来存储场上的对局情况与圈地的决策情况（即圈地时生成的预定路径 path），再依照自身和地敌方所作出的每一步行动来判断是否需要修改此 store 参数，从而大大减少每次都对实时对局情况进行搜索的额外耗时。

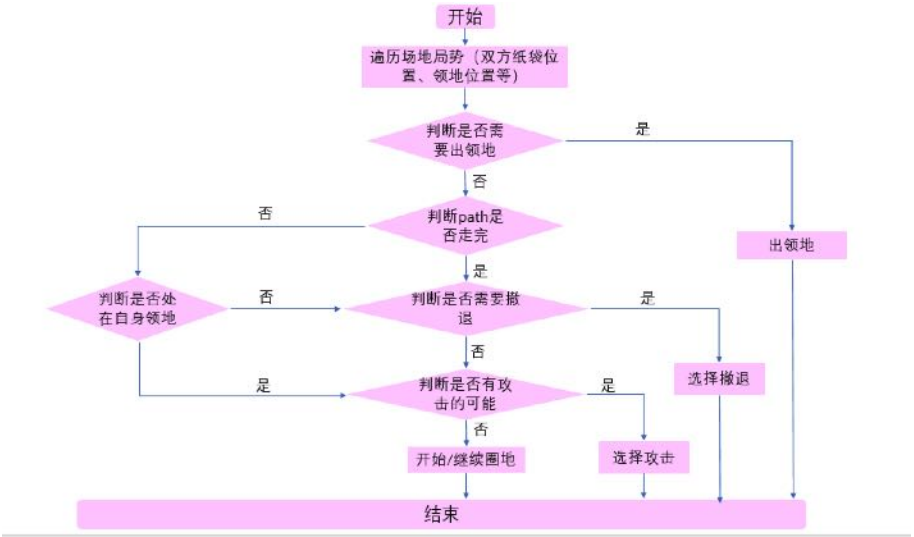
本组代码对撤退、攻击与圈地的决策判定条件为：当圈地函数生成的 path 已经执行完毕而尚未回到领地时，就选择撤退，调用 retreat 函数；若已经回到领地就选择调用 enclose 函数继续圈地，而若我方纸带到我方领地的距离加上事前设定的安全阈值 safe 不小于敌方到我方的纸带距离时，同样选择撤退 retreat 函数。同样，在圈地或攻击途中，若满足撤退条件（即若我方纸带到我方领地的距离加上事前设定的安全阈值 safe 不小于敌方到我方的纸带距离时）同样选择撤退。而不满足撤退条件时，如果满足攻击条件（即敌方纸带到其领地的距离大于我方纸带带敌方的距离，或敌方侵入我方领地时），就调用攻击函数 attack 进行攻击。而如果上述均未满足，就继续执行 path 的一系列操作。

一开始，我们想过用估值函数评定局势，再用搜索算法中的 minmax 算法完成三选一

的决策，再使用 alpha-beta 剪枝算法对函数进行优化。然而，当此代码的 v1.1 版本出现时，代码组发现其耗时过多且效果暂时并不好，对其时间复杂度进行优化也可能较为困难，得不偿失。因此，代码组决定改换思路，采用较简单的人工决策法，从而顺利完成了参赛代码 v2.1 版，并参与了热身赛。之后，从热身赛的结果中发现，代码 v2 在比赛时虽然防守部分较为有效，但攻击效率与圈地效率均不是特别高，比赛时一直绕着自身领地的边界而不主动出边界圈地，导致比赛常由于步数耗尽而圈地面积过小而终结。因此，代码组对圈地函数进行了优化，完成后的 v3 版代码使得圈地效率有了显著的提高，胜率也有了改善。对时间复杂度进行了再次简化，对 bug 进行了一定修正后，便诞生了本组的参赛代码。

本组思路是人工思维下对纸袋走向进行设计的一个较优解，其代码逻辑较为清晰，简洁明了。由于我组不求战术思维的复杂程度一定有多高，只进行了局部最优的判定，所以可读性也相对较强，同时代码的时间复杂度也较低。

1.2 算法流程图



1.3 算法复杂度

本组算法没有用到较复杂的数据结构，所以算法的时间复杂度也不算很高，主要耗时的部分在于对比赛场地上所有坐标的遍历，但由于 store 字典存储了坐标使得不用每走一步都遍历一次，则走一步的时间复杂度最坏是 O(n)，其中 n 为场上坐标的个数。而算法中 store 字典的使用会减少对坐标点的遍历，从而使得所需的时间减少，若局势不变，时间复

杂度甚至可以降到 O(1)，即双方 field 未变，只有纸带加一。

2 程序代码说明

2.1 数据结构说明

本组的算法为线性数据结构，通过人工条件的判断来进行决策。采用的数据结构有圈地函数 enclose() 中用列表所实现的栈 path，保存在自建的字典 store 中方便圈地进行调用，以节省算法运行的时间。而其目的为将人工思维决策的圈地策略存放于栈中，方便存储、传递参数并及时调用。

2.2 函数说明

本组算法中纸袋的行动模块有四个：get_out 函数（用于从我方 field 中走出去）、retreat 函数（用于撤退）、attack 函数（用于攻击）、enclose 函数（用于圈地）。而对场面局势进行评估的遍历函数有 traverse_fields 函数（领地遍历函数）与 traverse_bonds 函数（纸袋遍历函数）两个。以下将逐一进行说明。

```
# 向最近可以出去的位置移动
def get_out():
    maximum = WIDTH + HEIGHT
    coordinate_forward = plus(ME_X, ME_Y, *DIRECTIONS[ME_DIRECTION])
    if is_valid(*coordinate_forward):
        length_forward = me_to_me_field(*coordinate_forward, getout=True)
    else:
        length_forward = maximum
    coordinate_left = plus(ME_X, ME_Y, *DIRECTIONS[turn_left()])
    if is_valid(*coordinate_left):
        length_left = me_to_me_field(*coordinate_left, getout=True)
    else:
        length_left = maximum
    coordinate_right = plus(ME_X, ME_Y, *DIRECTIONS[turn_right()])
    if is_valid(*coordinate_right):
        length_right = me_to_me_field(*coordinate_right, getout=True)
    else:
        length_right = maximum
    length_min = min(length_forward, length_left, length_right)
    if length_min == maximum:
        return None
    elif length_min == length_forward:
        return 'fgetout'
    elif length_min == length_left:
        return 'lgetout'
    elif length_min == length_right:
        return 'rgetout'
```

我们可看出，get_out 函数的作用是当纸袋头在我方领地中，就要选择出去的最近方案。函数中，先用 is_valid 函数来保证纸袋运动有效（不能撞墙、撞到自己纸袋、撞倒在敌方领地），再使用编写的求距离函数 me_to_me_field 分别计算出向左、向右和直走的情况下，纸袋要走出我方领地的需要走的最短距离，而后比较选择这三种走法中最短的走法，并输出。

```
# 从使我方缩短最近距离，且远离敌方的方向撤退
def retreat(known_length):
    maximum = 2 * WIDTH + HEIGHT
    coordinate_forward = plus(ME_X, ME_Y, *DIRECTIONS[ME_DIRECTION])
    if is_valid(*coordinate_forward):
        lf, pf = me_to_me_field(*coordinate_forward, retreat=True, fc=(ME_X, ME_Y))
        length_forward = 2 * lf - min(known_length, length(ENEMY_X, ENEMY_Y, *coordinate_forward),
                                      length(ENEMY_X, ENEMY_Y, *pf))
    else:
        length_forward = maximum
    coordinate_left = plus(ME_X, ME_Y, *DIRECTIONS[turn_left()])
    if is_valid(*coordinate_left):
        ll, pl = me_to_me_field(*coordinate_left, retreat=True, fc=(ME_X, ME_Y))
        length_left = 2 * ll - min(known_length, length(*coordinate_left, ENEMY_X, ENEMY_Y),
                                   length(ENEMY_X, ENEMY_Y, *pl))
    else:
        length_left = maximum
    coordinate_right = plus(ME_X, ME_Y, *DIRECTIONS[turn_right()])
    if is_valid(*coordinate_right):
        lr, pr = me_to_me_field(*coordinate_right, retreat=True, fc=(ME_X, ME_Y))
        length_right = 2 * lr - min(known_length, length(ENEMY_X, ENEMY_Y, *coordinate_right),
                                    length(ENEMY_X, ENEMY_Y, *pr))
    else:
        length_right = maximum
    length_min = min(length_forward, length_left, length_right)
    # 撤退的左拐右拐优先方向与之前扩张圈地的左旋右旋方向相反，避免纸带缠绕到一起
    if store['lexpand']:
        if length_min == length_forward:
            return 'fretreat'
```

```
    # 撤退的左拐右拐优先方向与之前扩张圈地的左旋右旋方向相反，避免纸带缠绕到一起
    if store['lexpand']:
        if length_min == length_forward:
            return 'fretreat'
        elif length_min == length_right:
            return 'rretreat'
        elif length_min == length_left:
            return 'lretreat'
    else:
        if length_min == length_forward:
            return 'fretreat'
        elif length_min == length_left:
            return 'lretreat'
        elif length_min == length_right:
            return 'rretreat'
```

撤退函数 retreat 则是分别使用 is_valid 函数判断直走、左转与右转的撤退行动是否为合法（即会不会直接导致战败），若不合法则设定为最大值，从而退出选择。行动判断为合法后则计算出行动后与我方领地的距离，再将三种选择所导致的与我方距离进行比较，从而能够得出撤退的最优解。比较完毕，选择好策略后返回有效的直行/左转/右转命令，在左右均可时优先采取与先前圈地时相反的行为以尽量减少纸带的缠绕。当主函数判断应该

撤退时，就调用撤退函数，使得纸带能够有效地避开敌方的攻击。代码的主要思维逻辑，就是贪心算法，一步遍历达到单步最优。

```
# 我方主动攻击，选择最快接近敌方纸带，并尽量减少离己方领地的距离的走法
def attack():
    maximum = (WIDTH + HEIGHT) * 3.5
    coordinate_forward = plus(ME_X, ME_Y, *DIRECTIONS[ME_DIRECTION])
    if is_valid(*coordinate_forward):
        length_forward = me_to_enemy_band(*coordinate_forward) + 1.5 * me_to_me_field(*coordinate_forward,
                                                                                      fc=(ME_X, ME_Y))
    else:
        length_forward = maximum
    coordinate_left = plus(ME_X, ME_Y, *DIRECTIONS[turn_left()])
    if is_valid(*coordinate_left):
        length_left = me_to_enemy_band(*coordinate_left) + 1.5 * me_to_me_field(*coordinate_left, fc=(ME_X, ME_Y))
    else:
        length_left = maximum
    coordinate_right = plus(ME_X, ME_Y, *DIRECTIONS[turn_right()])
    if is_valid(*coordinate_right):
        length_right = me_to_enemy_band(*coordinate_right) + 1.5 * me_to_me_field(*coordinate_right,
                                                                                  fc=(ME_X, ME_Y))
    else:
        length_right = maximum
    min_length = min(length_left, length_right, length_forward)
    if min_length == maximum:
        return None
    elif min_length == length_forward:
        return 'fattack'
    elif min_length == length_left:
        return 'lattack'
    elif min_length == length_right:
        return 'rattack'
```

attack 函数则同样是先用 is_valid 函数判定直行、左转、右转的行为是否为合法，若不合法则选择不攻击。再计算我方纸带与敌方纸带的距离，将直行、左转与右转三者的结果进行比较，得出最小值，从而计算出攻击的最优方向。具体来看，当主体程序判断出有攻击的可能时，就调用攻击 attack 函数，其主要的思维逻辑还是一步搜索，目标为达到局部最优。

```
def enclose_left():
    limit0 = max(me_border[0] - store['expand'], 0)
    limit1 = min(me_border[1] + store['expand'], HEIGHT - 1)
    limit2 = max(me_border[2] - store['expand'], 0)
    limit3 = min(me_border[3] + store['expand'], WIDTH - 1)
    if ME_DIRECTION == 3:
        store['path'] = ['lenclose']
        store['path'].extend(['fenclose'] * (me_border[1] - limit0 - 1))
        store['path'].append('lenclose')
        store['path'].extend(['fenclose'] * (ME_X - limit2 - 1))
        store['path'].append('lenclose')
        store['path'].extend(['fenclose'] * (ME_Y - limit0))
    elif ME_DIRECTION == 1:
        store['path'] = ['lenclose']
        store['path'].extend(['fenclose'] * (limit1 - me_border[0] - 1))
        store['path'].append('lenclose')
        store['path'].extend(['fenclose'] * (limit3 - ME_X - 1))
        store['path'].append('lenclose')
        store['path'].extend(['fenclose'] * (limit1 - ME_Y))
    elif ME_DIRECTION == 2:
        store['path'] = ['lenclose']
        store['path'].extend(['fenclose'] * (me_border[3] - limit2 - 1))
        store['path'].append('lenclose')
        store['path'].extend(['fenclose'] * (limit1 - ME_Y - 1))
        store['path'].append('lenclose')
        store['path'].extend(['fenclose'] * (ME_X - limit2))
    elif ME_DIRECTION == 0:
        store['path'] = ['lenclose']
        store['path'].extend(['fenclose'] * (limit3 - me_border[2] - 1))
        store['path'].append('lenclose')
        store['path'].extend(['fenclose'] * (ME_Y - limit0 - 1))
        store['path'].append('lenclose')
        store['path'].extend(['fenclose'] * (limit3 - ME_X))
```

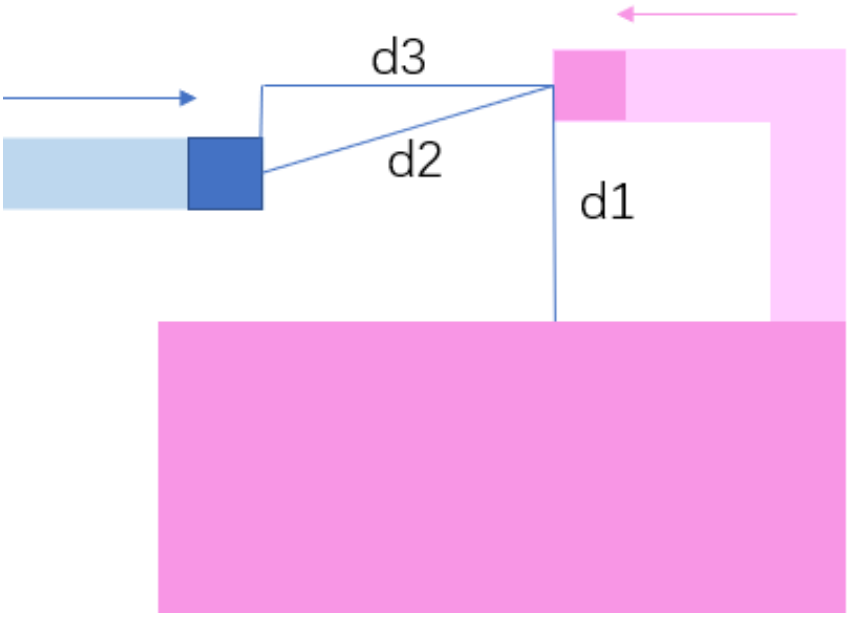
```
def enclose_right():
    limit0 = max(me_border[0] - store['expand'], 0)
    limit1 = min(me_border[1] + store['expand'], HEIGHT - 1)
    limit2 = max(me_border[2] - store['expand'], 0)
    limit3 = min(me_border[3] + store['expand'], WIDTH - 1)
    if ME_DIRECTION == 3:
        store['path'] = ['renclose']
        store['path'].extend(['fenclose'] * (me_border[1] - limit0 - 1))
        store['path'].append('renclose')
        store['path'].extend(['fenclose'] * (limit3 - ME_X - 1))
        store['path'].append('renclose')
        store['path'].extend(['fenclose'] * (ME_Y - limit0))
    elif ME_DIRECTION == 1:
        store['path'] = ['renclose']
        store['path'].extend(['fenclose'] * (limit1 - me_border[0] - 1))
        store['path'].append('renclose')
        store['path'].extend(['fenclose'] * (ME_X - limit2 - 1))
        store['path'].append('renclose')
        store['path'].extend(['fenclose'] * (limit1 - ME_Y))
    elif ME_DIRECTION == 2:
        store['path'] = ['renclose']
        store['path'].extend(['fenclose'] * (me_border[3] - limit2 - 1))
        store['path'].append('renclose')
        store['path'].extend(['fenclose'] * (ME_Y - limit0 - 1))
        store['path'].append('renclose')
        store['path'].extend(['fenclose'] * (ME_X - limit2))
    elif ME_DIRECTION == 0:
        store['path'] = ['renclose']
        store['path'].extend(['fenclose'] * (limit3 - me_border[2] - 1))
        store['path'].append('renclose')
        store['path'].extend(['fenclose'] * (limit1 - ME_Y - 1))
        store['path'].append('renclose')
        store['path'].extend(['fenclose'] * (limit3 - ME_X))
```

```
# 计算当前方向的左侧和右侧领地区域相对大小（用边界点数量简单估算），左侧点多，就进行左旋圈地，反之右旋，以尽量减少圈地效率
def enclose():
    left_direction = turn_left()
    left = right = 0
    if left_direction == 0:
        for coordinate in me_border_coordinates:
            if coordinate[0] > ME_X:
                left += 1
            elif coordinate[0] < ME_X:
                right += 1
    elif left_direction == 1:
        for coordinate in me_border_coordinates:
            if coordinate[1] > ME_Y:
                left += 1
            elif coordinate[1] < ME_Y:
                right += 1
    elif left_direction == 2:
        for coordinate in me_border_coordinates:
            if coordinate[0] < ME_X:
                left += 1
            elif coordinate[0] > ME_X:
                right += 1
    elif left_direction == 3:
        for coordinate in me_border_coordinates:
            if coordinate[1] < ME_Y:
                left += 1
            elif coordinate[1] > ME_Y:
                right += 1
    if left < right:
        store['lexpand'] = False
        enclose_right()
    else:
        store['lexpand'] = True
        enclose_left()
```

圈地函数分为了 enclose_left 函数、enclose_right 函数与 enclose 函数这三个函数。前两个函数的作用是分别求出纸带向左圈地与向右圈地分别生成的路径 path，path 的具体参数（圈地的大小等）由事先设定的阈值决定。enclose_left 函数与 enclose_right 函数生成 path 并将 path 的每一步都存入字典 store 中，方便后续进行调用；而之后的 enclose 函数则是根据当前左右的领地大小比较（左右领地面积以边界点的数量进行估算）来决定应该向左还是向右圈地，

2.3 程序限制

在如下图的情况下，我组的算法会出现无法识别敌方纸带的状况，从而导致被敌方撞死。其原因是由于我组算法所计算的双方纸带的距离是斜线距离 d2，而下图情况下若要及时撤退，正确的做法应该是判断我方纸带与领地的距离 d1 与我方纸带到敌方纸带的距离 d3 的大小，发现 d1 + safe（根据经验设定的与纸带长度有关的安全阈值）> d3 后撤退，而实际上的算法采取的是判断 d2 与 d3，发现 d2 + safe > d3 后选择不撤退，从而被撞击死亡。



3 实验结果

3.1 测试数据

实验环境说明：

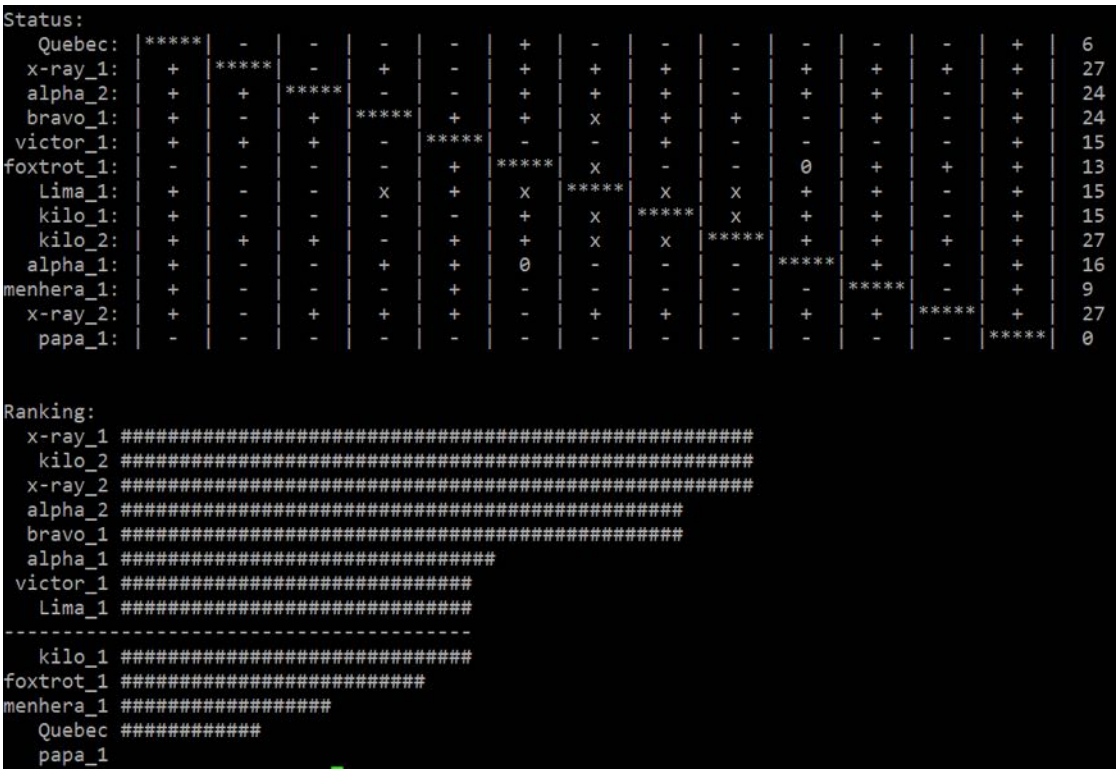
- 硬件配置：（CPU/内存）CPU i7/8G
- 操作系统：（名称/版本）Windows10
- Python 版本：（版本号）python3.6.4/3.5.2

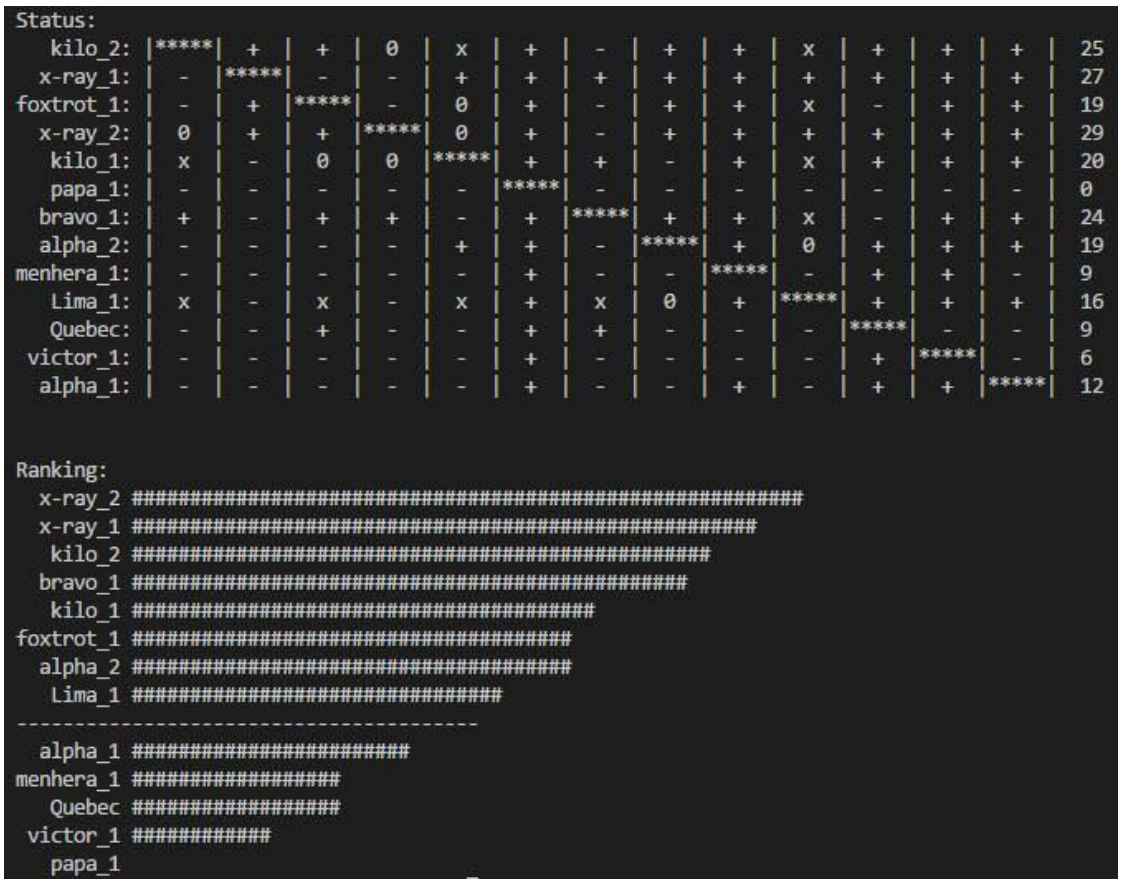
我们采取的测试方法是保持算法框架不变，主要修改算法运行的关键参数（如决定是否撤退的，由纸带长度分为三档的安全阈值 safe 与每次决定圈地面积大小，根据我方纸带头与对方纸带头的距离分三档的圈地尺度 expand_length），根据算法的胜率来最终决定这些参数。而算法实验的结果即为代码中最终呈现出的参数。

```python
def expand_length():
    l = length(ME_X, ME_Y, ENEMY_X, ENEMY_Y)
    if l > (WIDTH + HEIGHT) / 2:
        return 30
    elif l > (WIDTH + HEIGHT) / 6:
        return 20
    else:
        return 10
```

```python
me_band_length = len(me_bands_coordinates)
if me_band_length < 11:
    safe = 5
elif me_band_length < 46:
    safe = 10
else:
    safe = 15
```

我组的代码参加热身赛的结果如下：





3.2 结果分析

本组的测试是改变了参数后相同思路的算法互搏，且两个版本 kilo_1 与 kilo_2 的总体思路没有太大的改变，因此两者互相对决的情况是谁先手谁就可以获得胜利。

在热身赛中，我们的代码暴露出了一些问题，未经优化的 kilo_1 在撤退时算法出现了 bug 而撞死。而面对 foxtrot 这样步步紧逼，圈地的同时时刻保持进攻性的算法，我方只能任由其蚕食领地，大多数对决结果中最终圈地面积是对方较大。而经过了改进后，我组的算法 kilo_2 终于在热身赛中取得了较好的排名。但我们在热身赛后未对算法进行更加彻底的优化，由于单步最优策略的实行，我组的算法存在一些局限性，而圈地的效率仍旧有待加强。我组的代码在圈地前期过于保守，圈地面积较小，导致了圈地能力的较为不足。而经过了代码组同学的强势修改，

实际对决中，由于我组未能想到对现有代码的全新而彻底的优化，导致代码还是存在

许多热身赛时暴露的遗留问题，再加上许多组热身赛保留实力，最终用于参赛的代码相比热身赛有了长足的提高，从而导致了实际小组赛时我组的代码未能出线。其中，我组未能出线的关键即是与 charlie 组的比拼的落后。第 0 局中，我组出现了 2.3 算法局限中的那种情况，这是之前圈地算法面积小时所无法发现的局限。之后，我们出现了与热身赛时对战 foxtrot 相似的情形——面对对方边圈地边蚕食我方领地的行为，我方缺乏强有力的应战进攻措施，导致一味的退让使得最后失败。而之后的我方和 charlie 组出现了镜像循环，二者的思维逻辑出现了明显的相似性，一路顺着边界镜像地行动，而我方的防守最后仍旧出现了漏洞，运动的进攻性没有那么强，导致被步步逼近自己的领地，最后被对方撞击。

我组的代码令人欣慰的一点是其相对较为简洁易懂的算法逻辑与较低的时间复杂度，而这也使得它多少没有那些统筹了多种模块的函数般拥有那样好的性能。而其所耗费的时间在热身赛中就相对而言较小（热身赛时 quebec、menhara 等正式比赛八强出线的代码均出现了思考时间不足的情况），正式比赛时也未与其他参赛函数出现较大的差距。

4 实习过程总结

4.1 分工与合作

本组的分工为：姜金廷，牟星名与麻一凡共同进行主体代码编写，李玥阳负责最后报告的主体部分、对组会的记录及提供建议，姜金廷负责主要协助完善报告，麻一凡与牟星名协助完善报告。期间，姜金廷负责整体框架的构思与谋划，麻一凡与牟星名负责具体部分代码函数的编写，并共同负责代码的测试与热身赛对局情况的查看与复盘。组员之间主要采用微信群聊进行交流，使用`https://coding.net`进行代码的共享与修改。

第一次组会的时间在 5 月 27 日，地点是二教 514 小教室。组会期间讨论了本组的分工安排以及所采用的核心算法（当时初步设想为搜索算法，用 alpha-beta 剪枝进行优化），而第二次组会在 5 月 31 日数算课上进行，组会期间讨论了由于搜索算法的优化复杂与效率低下而全面改用局部最优的贪心算法并重新进行分工，此后均采用献上交流的形式进行测试与复盘的沟通。而且，每次数算课上小组成员会坐在一起进行讨论，进一步交流每周的进度与新的思路。

4.2 经验与教训

数算大作业的过程中，得到一个能够实际参与比赛的 AI，自然是极富有成就感的一件事了，而这一结果所最值得我们骄傲的也就是它真的可以用来作为参赛选手，带着诸君的心血奋战在赛场上这一不容置疑的事实。

实习过程中的教训是不能太过依赖热身赛的成果，要孜孜不觉持之以恒地对代码进行精益求精；同时分工上也应该有进一步优化，全体组员都应当及时跟进最新赛程，从而能够为了代码提供自己的建议。

4.3 建议与设想

By 众组员

1、比赛环境如果能升级到最新版就好了呢

2、如果可以在课程初期想好比赛内容，准备时间或许会更加充裕吧

3、比赛赛制采取天梯制度会比现在更加公平

4、游戏运行时可以加上背景音乐（啥）

后续工作设想：

1、听了一些小组比赛时的总结，觉得搜索算法在某些情况下还是比较有用的，感觉可以基于搜索算法 v1.1 继续探索

2、即使是最终版代码也有很多的逻辑上的 bug 及冗余部分，可以进一步优化代码

3、最终版的参数可以进行更多的试验和调整

4、撞击对手时的进攻思路和防守思路可以进一步优化

5、圈地效率可以有进一步的提升，考虑到与敌方相对位置的更多情况

5 致谢

感谢我们的代码组，在大作业的完成过程中挥洒汗水，起着不可或缺的作用；感谢陈斌老师和各位助教，在课程的学习上使我们受益匪浅；感谢技术组的大佬们搭建比赛平台，使得全部比赛可以正常运行；感谢数算群的各位同学，在思维火花的迸发中让完成大作业的过程寄托了全新的灵感，凝聚了汗水与热血，奏响青春的赞歌（啥）。

6 参考文献

《PC 游戏编程 (人机博弈)》——王小春编著

《他改变了中国》（啥）

第十三章　F17_Lima 报告

周正清 *，叶帆，杨易轩，周一川

摘要：本小组所编写的 play 函数主要由三个部分组成：圈地算法，寻径算法与逻辑算法。每次调用函数时，由逻辑算法判断当前局势，并作出进一步的指令。本函数内部使用了有序表、链表等数据结构，借鉴并改良了用于寻找最短路径的 a* 算法。通过多重逻辑判断、启发式搜索等方式，整个算法的复杂度保持在 O(n2) 的基础上，尽可能地降低了运算时间，加强了函数的对抗能力。

关键字：有序表；链表；启发式搜索；a* 算法.

1 算法思想

1.1 总体思路

本算法将 AI 分为三个部分：圈地算法（三分之一算法）、寻路径算法（astar）和逻辑算法。逻辑算法用于判断 AI 进入进攻、防守或圈地模式之类，统筹其它算法，将它们组装起来，各自发挥最大效用；寻路径算法用于在避开纸带的前提下寻找两点间最短路径，是逻辑、圈地都需要用到的底层算法；圈地算法用于在逻辑运算进入圈地模式后在保证绝对安全的前提下圈尽可能大的地。

在总体思路而言，由于我方近战能力极强，我们采用狼性进攻的想法。在保证每次圈地都绝对安全的前提下，每一次都朝着对方圈地，每一次领地内游走都走向对方，在一百步之内即可与对方短兵相接。这个策略效果十分显著，可以有效遏制敌方的圈地范围，将交战重心从大家都擅长的圈地，转移到我方最擅长的近距离交战上面来。若敌方自顾自逃

跑圈地，我方圈地效率甚至能达到对方的两倍，这也弥补了我们圈地算法的缺点。很多组都在圈地算法上花了很多时间，造成圈地算法全面优于我们的三分之一算法，而在双方交战的算法上就少做了一些功夫，在极短距离下很容易判断失误，最终结果也证明了这一点。

同时，我们也找到了短兵相接的必胜策略。从开局一开始，双方纸卷的距离的奇偶性就已经定下来了。在轮到自己回合时，如果双方距离是偶数，那么我方就一定不可能侧碰对方头部；如果是奇数，那么对方就一定不可能侧碰我方头部。这是在无主领地下双方交战的必胜策略。

而在双方领地交接处交战，我们的策略则以龟缩为主，不给敌方任何攻击我方的机会，一旦找到机会攻击敌方，就一击必杀。

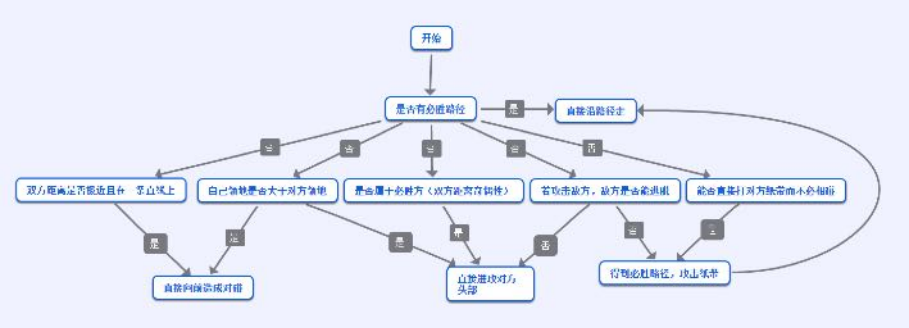
本算法主要采用了有序表、链表，后文将有详细说明。

在第一次开会的时候，我们还没有想到必胜策略，只是大概分了一下任务：一个人写寻径，一个人写圈地，一个人写短兵交战的逻辑。在一周后的第二次会议上面，第一次会议由于考试缺席的大佬提出了必胜策略以及保证圈地绝对安全的思路，给了我们极其重要的启发，自此，我们形成了寻径、圈地、逻辑、参谋四人分工的合作模式，高效率地实现了我们的算法。

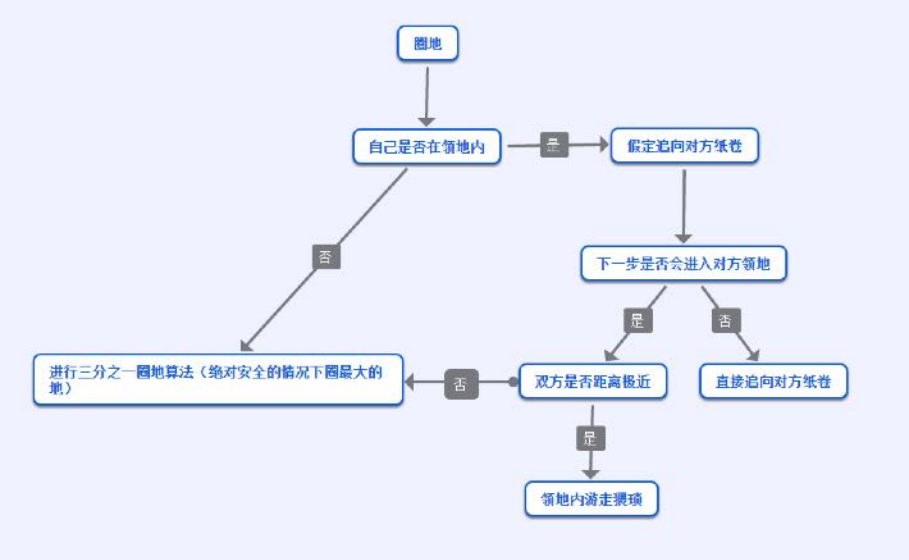
1.2 算法流程图

（1）逻辑算法

需满足所有条件才能进入这些模式，若没有进入任何一个模式，则开始圈地。



（2）圈地算法



上图箭头并不需要满足所有条件，满足一个即可。

1.3 算法复杂度

本算法运行时间主要由寻径算法（astar）耗时、遍历棋盘耗时、逻辑分析耗时为主，且实际运行中三部分耗时大致相当：我们曾对遍历棋盘的函数进行优化，原本需要每一步都遍历完棋盘，现在只需要在纸卷进入领地的时候遍历，圈地时直接添加纸卷坐标进入保存纸带的列表就行了，预计能减少一半的耗时，但优化后的总耗时只减少了五分之一，由后文也可知道寻径和遍历棋盘耗时大致相同，则可推论除此两项外的算法逻辑和它们的耗时都大致相同。它们的时间复杂度都是 $O(n^2)$，这是由于寻径、遍历棋盘都有可能查找遍所有的格点，而逻辑分析需要用到棋盘内存储的点，最大的时候也可能遍历棋盘。

在第一次热身赛里我们的寻径和遍历棋盘算法超时严重，平均三十秒只能走 1500 步。经过多次寻径、遍历、逻辑的三重优化，我们的算法可以在十秒内完成 2000 步，大大提升了计算效率，但时间复杂度仍都是 $O(n^2)$。

2 程序代码说明

2.1 数据结构说明

本程序主要通过 python 库内置的列表、元组与字典结构进行数据的存储，辅以对课程介绍的有序表等数据结构的使用。

在寻径函数 astar 中，通过自定义 OrderedList 类与 Node 类实现了有序的单向链表结构的存储。astar 作用为：在有障碍（即自身的纸带）的情况下寻找两点之间的最短路径。自定义 Node 类用于记录场地内每一点及该点在 astar 函数中需要的相关信息（如与 start 点之间的曼哈顿距离 g，启发距离 h，父节点 father 等），而 OrderedList 则将按 f=g+h 的大小将存储其中的 Node 元素升序排序，通过 get 方法返回 f 值最小的元素。

astar 函数中力图通过单项有序链表避免排序的使用，以此减少寻找最小值所需的计算开销；同时通过单项链表的形式记录了父节点的信息，可进一步在寻径过程中由终点逆向记录最短的路径。

2.2 函数说明

2.2.1 辅助函数：

该部分函数作为计算与写代码的工具，有效的减少编程人员的工作量：

astar(stat, storage, player, destiny):

接收参数：字典 stat、storage；字符串 player（值为’me’或’enemy’）；大小为 2 的元组 destiny（例如 (5,6)）

功能：整个程序核心函数之一，用于寻找一方纸带到某一点的最短路径。

算法策略：考虑到深度与广度优先算法在 100*100 的二维列表中寻径的高计算开销，我们借鉴了 A* 算法【1】，利用估价函数（即前文中的启发距离）进行启发性搜索，以寻找最短路径：取决于该游戏的性质，玩家纸卷只能向东南西北四个方向行走，两点之间所需的最短步数为 x+ y，因此 astar 中选择了曼哈顿距离（即 x+ y）作为估价函数计算路径上每一点的启发距离 h。另一方面，每一点到起点的最短路径长度作为 g 值（通过链表结构将最短路径连接），由此获得 f=g+h 作为评判这一点是否适合被包括在路径中的依据。

通过这样的启发式搜索，最终将得到 f 值最小的 destiny 点：即找到到达 destiny 的最短路径。

但显然这样的 astar 并不能满足我们程序的需要：当 astar 返回的路径与自身方向相反的时候，纸带显然需要改换一条可行的路径；同时，通过 astar 返回的路径回到领地时，纸带围成的面积显然不一定能最大化。

通过调整启发性搜索的优先级、搜索可替代路径，astar 将返回一条平直、可行的路径，符合了我们对于它的期望。

bianli(stat, storage):

接收参数：字典 stat 与 storage

功能：play 函数被调用时首先运行，获取敌我纸带、领地所有点的坐标，以列表的形式存储于 storage['playground']['enemyfields']

goto(stat, direction):

接收参数：字典 stat（matchcore 提供）与大小为 2 的元组所代表的单位向量 direction（如（0,1））

功能：通过 stat 获取自己的方向信息返回转向指令（'l'/'r'/None）。

distance(start, end):

接收参数：大小为 2 的元组记录的点的坐标 start 与 end

功能：返回两点之间的曼哈顿距离（横纵坐标距离差之和）

judge(stat,point1,point2):

接收参数：字典 stat，两个点的坐标 point1、point2。

功能：粗略判断两点之间是否有纸带，以决定是否通过 astar 寻径。

PointToLine(stat, storage, player, line):

接收参数：字典 stat、storage，字符串 player（'me'或'enemy'）以及一个由点的坐标组成的列表。

功能：获取点到线段最短路径。

2.2.2 策略函数

quandi（stat，storage）：

接收参数：字典 stat，storage

功能：向对方纸卷位置出发，圈出最大的地

算法策略：在前期讨论中，我们组总结了如下两条经验：

1. 圈地时应一直向对方的领地出发，挤占对方的生存空间，圈到对方的领地将得到双倍的收益（我方得到，对方损失），同时可以有效的防备“偷鸡算法”以圈矩形为例，受到围栏问题（如下图）的启发，当安全距离为 L 时，圈出边长为 L/3 的正方形面积最大。

例 1 （2012 年湘潭）如图 2，某中学准备在校园里利用围墙的一段，再砌三面墙，围成一个矩形花园 $ABCD$（围墙 MN 最长可利用 25 m），现在已备足可以砌 50 m 长的墙的材料，试设计一种砌法，使矩形花园的面积为 300 m^2.

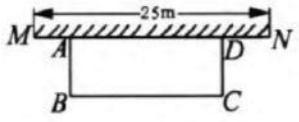


图 2

因此，我们在圈地中尽量保持这个原则，在决策时计算双方的安全距离，在保证安全的情况下进行圈地，一旦出现危险，则立即用 flee 返回领地。

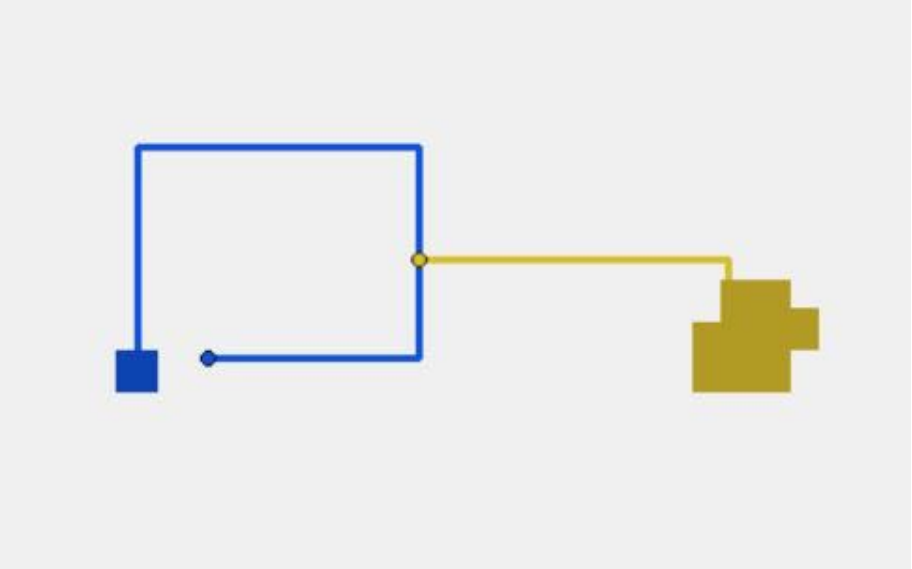
flee(stat, storage, playground)：

接收参数：字典 stat，storage，playground 其中 playground 为 storage[‘playground’]（为了写代码的方便，在参数列表里加了这个参数）

功能：逃跑函数，返回由自己纸带到领地的最短路径，以供逃跑。

2.3 程序限制

1. 三分之一距离的圈地法只适用于己方领地较大，能够提供一堵“围墙”的情况，在程序圈出第一块地的时候，显然己方领地过小，1/3 的距离难以保证自己的安全，如果对方在此时直接冲向我方纸带，将被击杀，如图所示。



2. 在距离较近时，我方采用了龟缩战术，保持自己在领地内的安全，省去了大量的代码，但也导致在近战的时候无法有效的圈地，最终差之毫厘输掉比赛。

3 实验结果

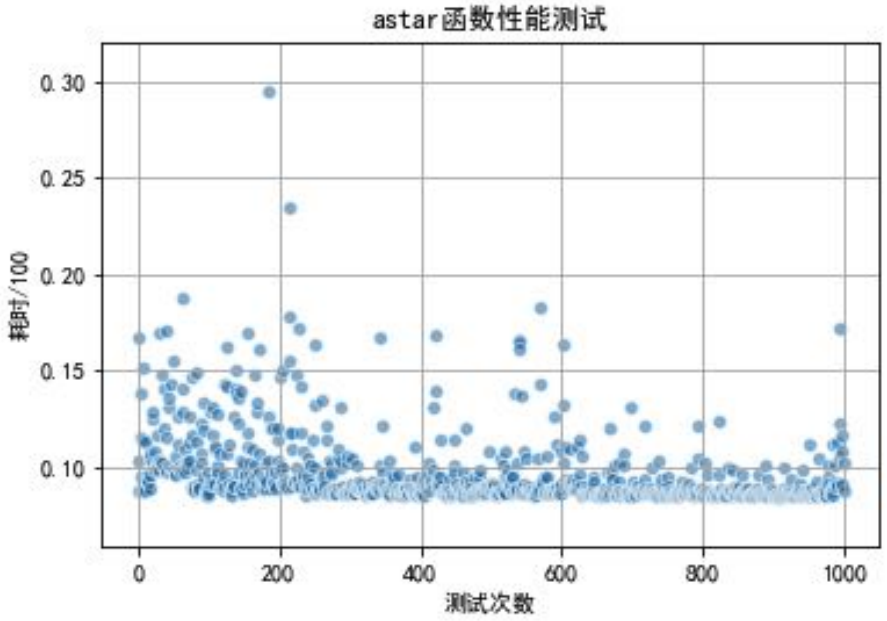
3.1 测试数据

实验环境说明：

1. 硬件配置：Intel(R) Core(TM) i7-7600U CPU @2.80GHz 2.90GHz
2. 操作系统：Windows 10 教育版
3. Python 版本：3.63

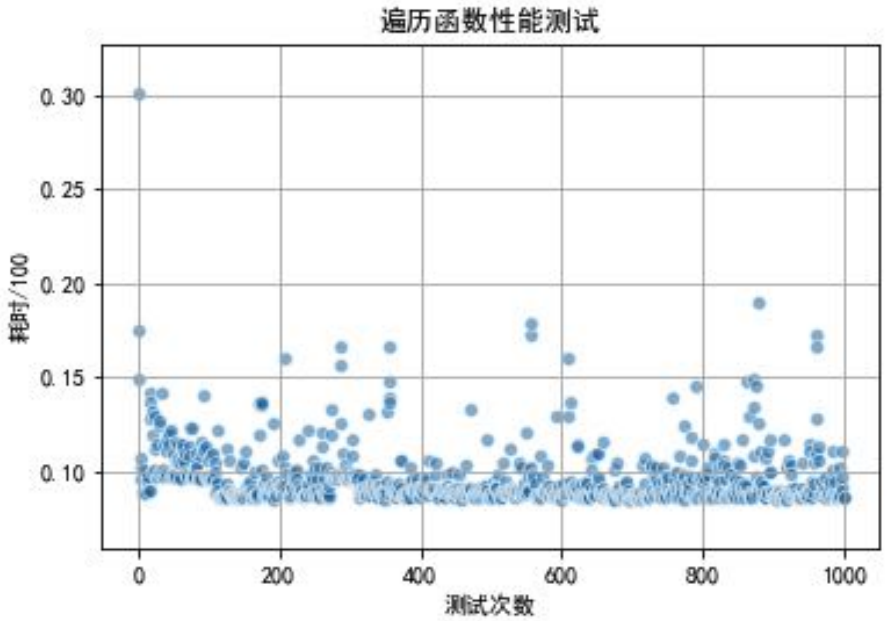
本组在热身赛时先后尝试了油聚圈地，推进城圈地以及最终提交的三分之一圈地函数，热身赛时圈地函数暂时不完善，导致热身赛均未出线。随后我们进行了对主函数各部分耗时的测试实验，在给定的场景中测量耗时，再对算法进行优化，测试代码附在后文。最后在赛前约战 N17 小组进行测试与 debug。战绩 15 胜 5 负。

astar 函数是整个 ai 的基础函数，整体测试与 debug 工作均以 astar 函数为核心展开。



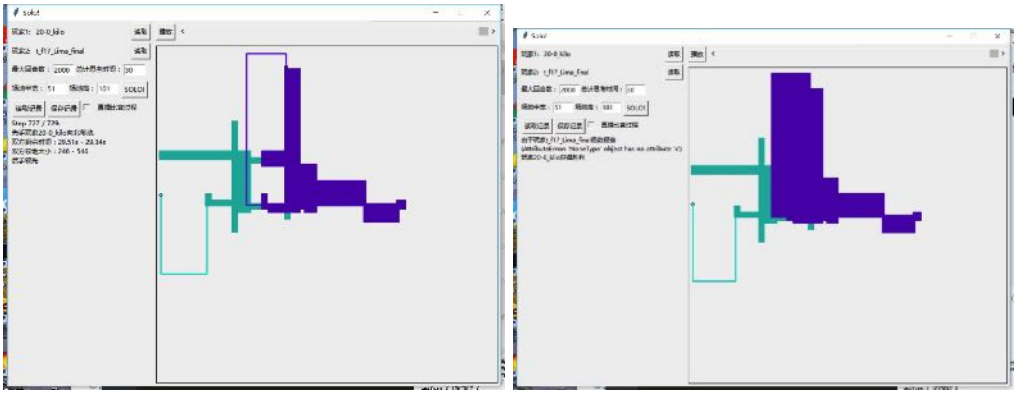
上述测试平均耗时 0.000955s，可以看出 astar 函数耗时大约在 0.001s 数量级以内，且较为稳定。

接下来测试 bianli（遍历）函数：



上述测试平均耗时 0.000951s，遍历函数耗时也在 0.001s 数量级以内，且稳定性较高。

程序耗时稳定在 0.01s 数量级以内，走满 2000 步的时间期望值小于 20s，说明程序性能符合要求。随后进行对战测试，对手是 N17 组某一出线小组，战绩 15 胜 5 负。得益于我组强大的进攻算法，15 次胜利中，大多为直接 KO 对手，接下来分析 5 次失败的原因：函数报错四次——AttributeError：’NoneType’ Object has no attribute ‘x’，在对方领地被撞死一次。



当对方的头部在地图边缘时，使用 astar 函数进行判断时容易出现上述报错。随后我们进行了 debug 改正了上述错误。

3.2 结果分析

在实战对弈过程中，本组算法在攻防方面性能相当优异，近身战斗至少可以做到不败，astar 函数、bianli 函数、play 函数主体逻辑三者用时相当，符合实验预期值。当对手侧重于进攻时，我们通常会以较高的 KO 率赢得比赛；如果对手一旦在防水方面滴水不漏，做到了双方均无法击杀对方，比赛进程就会进入到比圈地面积的阶段。

由于本组算法的强进攻性，设定了贴身战斗时若无法击杀对方，则龟缩在己方领地边界来回走动等待机会的机制。如果双方都有设定这样的机制，在前几次贴身战斗时双方就容易出现各自龟缩在己方领地的循环直到走满 4000 步。这样一来就需要比拼前期圈地的面积，这种比拼圈地面积的循环意料之外的结果。

在四强赛中，对手的圈地算法略弱于我组，常常在圈完第一块地后的贴身战斗中进入循环，最后由于圈地面积略小而战败。而在半决赛中对阵冠军 N 组，对手的防守摆脱与圈地能力相当强，我组难以顺利击杀对手，最后由于圈地面积较小战败。我组龟缩等待机会

的机制与圈地函数的性能仍有提高空间。

4 实习过程总结

4.1 分工与合作

在整个实习过程中，我们线下聚会若干次。一起讨论算法的优化（例如在第二次聚会大型蚊香圈地被推进城取代，第三次聚会推进城被 1/3 取代……），修复代码的漏洞（A-star 的诸多死循环……）。同时，每一次聚会都会根据各人代码的实际难易程度对分工进行微调，防止出现工作量极度不平均的情况。此外，在线上我们随时交流，其余组员可对某一组员遇到的问题提出建议，也可指出对应部分的不足和优化方向。

直到代码提交截至，各组员的具体分工如下：

周正清：A-star 算法的实现；1/3 圈地算法的实现

杨易轩：核心算法的理论提出与计算；推进城圈地算法（后淘汰）的实现

叶帆：战斗函数的实现；各部分代码的整合

周一川：大型蚊香圈地算法（后淘汰）的实现；函数的调试（亦即 debug）


4.2 经验与教训

1. 要尽快将算法付诸为代码，否则会陷入理论与实践不符的陷阱。在预期中，推进城应为最终的圈地函数。在推进城具体实现前，我们用了大量时间进行理论推衍，然而推进城算法完成后，不仅实际圈地面积远小于估计，且在与主函数的对接上出现了诸多 bug。

经多次调试，效果始终不理想，最终不得不放弃而另寻他法。因此，最终的圈地函数没有足够的时间优化，思路比较简单，也导致了我们在半决赛中的折戟。

2. 不要过分迷信高级算法，合适的才是最好的。我们一度打算搜索并剪枝，从而得出每一步最优解，随后发现任意两点间的最优路径在保证安全的情况应该是简单的一次折线，多步搜索产生了大量无效的计算量。除此之外，在近战函数中我们也找到了相对优解，从而减小了计算量。经理论分析和实践证明，棋类 AI 必备的深度搜索在此次实习中并不适用。因此，最终算法的 time left 总是冗余的。

4.3 建议与设想

目前，圈地函数最多只能走四条边（矩形圈地），每次圈地过程中纸带转向的顺逆时针都是确定的，这样简化了计算过程，但相应的限制了圈地面积。因此，在未来的优化中可以改为锯齿形圈地，从而增大圈地面积。

之前已经解释了 1/3 的由来。但是我们未必需要 1/3 带来的绝对安全，适当的修改这一系数可能带来风险，但也会带来更高的收益。此外，也可根据对方接下来的策略在每一步调整该系数，从而使整体更加优化。

5 致谢

感谢清华大学高亦凡同学、北京航空航天大学陈思同学、天津大学武思瑄同学在本次实习中为本小组积极献言献策，提供思路与建议，尤其感谢这三人向我们介绍了 a* 算法，大大优化了本小组算法的用时。

6 参考文献

【1】A* 算法入门 http://www.cppblog.com/mythit/archive/2009/04/19/80492.aspx

第十四章　F17_Menhera 报告

李法承 周赫 李书承 王禹菲 余年年

摘要：通过人工启发设计，小组成员部分地吸纳贪心算法中保证当下情况最优的思想，运用列表、字典、队列等数据结构，编写了根据情况可进入攻击模式、逃跑模式或扩张模式并通过遍历搜索、比对判断得出单步情况下较优走向的 paper.io 算法。最终参赛的版本以攻击性能为主，辅以一定的扩张与防御性能，并且在其余对手均以扩张性为主的小组赛中取得绝对优势。虽然本算法在面对强攻击性对手时表现出劣势，并以止步八强的结果遗憾收场，但其采取人工启发、“贪心”思想及多功能结合的实现方法，仍在算法设计及游戏对抗中具有较高的借鉴价值。

关键字：人工启发；局部最优；队列；遍历搜索；判断语句

1 算法思想

1.1 总体思路

在小组创立之后，我们根据 Line 等社交软件上近日流行的 Menhera-Chan 系列表情，将组名设定为“Menhera”[1]。下文中将以“Menhera 算法”指代小组编写的人工智能程序。

Menhera-Chan 系列表情示例

在 Menhera 算法的设计过程中，小组成员曾经提出过多种构想，包括机器学习、递归搜索等，但经过一定的实践，我们部分地吸纳了贪心算法的思想，利用遍历搜索与比对判断，首先确定了一种优先保障自身安全、以防御及扩张性能为主的算法。该版本算法并未采取普通意义上的“全局策略”，而是通过条件判断“贪心”地保持当前局势下的生存及最优性能扩张，并且在具有十足把握时对敌方发动进攻。

而在后续开发过程中，由于我们注意到分区小组赛内其他队伍也具有优异的圈地性能，故最终改进形成了一个更具备攻击性、同时保有一定防御和扩张能力的参赛版本。以下的代码分析部分将主要以该参赛算法为核心展开。

为了同时保有足够的攻击、防御与扩张性能，我们设计的 Menhera 算法包括四个核心部分：（1）人工启发思想指导的主体控制语句；（2）攻击模式函数；（3）逃跑模式函

数；（4）扩张模式函数。以下将对这四个部分进行分别介绍。

1.1.1 人工启发思想指导的主体控制语句

主体控制流语句的编写采取人工启发式方法，结合观察和思索首先保证我方的生存，并且尽可能确定最优参数。

在每次移动开始前，函数将首先确定不会导致我方“自尽”的走向。如果仅余一个走向，则直接采取该走向。如果剩余走向不止一个但不包括直行走向，说明当前我方位置附近局势可能较为复杂，存在纸带缠绕的现象，应尽快寻找出口，避免“盘蚊香”导致切断自己。

如果剩余走向不止一个且包括直行，函数才会读取各类参数、运行计算，并进行下一步操作。在不同情况下可能需要计算的参数可能包括：

（1）Dekm：敌方切断我方纸带的最短距离 [本文中提到的“距离”，如未特别注明，均按照曼哈顿距离计算。]；

（2）Dmke：我方切断敌方纸带的最短距离；

（3）Deh：敌方返回领地的最短距离；

（4）Dmh：我方返回领地的最短距离；

（5）Dmo：我方距离非己方领地的最短距离；等。

若此时比赛伊始，则 Menhera 算法会使纸卷通过绘制方形，划定一块较大的势力范围，以保有初始优势；若初始圈地已经结束，则会依照我方及敌方情况判断情况（详细的判断条件参见下文模式说明部分），调用适宜的模式函数作出行动。

1.1.2 攻击模式函数

较为注重攻击性能的参赛版本中，设置了两种可以引发攻击模式的条件，具体如下：

（1）当我方有足够把握直接击败敌方时。在计算比对中，如果发现 Dekm 小于 Deh，且满足 Dmke 小于 Dekm 或我方位于自己领地内，则认定为我方有足够把握击杀敌方，从而触发进攻机制。

当敌方进入我方领地且不位于我方领地边缘时（地图边缘在此判断条件下不视为我方领地边缘）。在先前攻击性较弱的版本中，我们并未设置该触发条件。此触发条件大大强化了 Menhera 算法的攻击性，但同时由于条件过于宽泛，在一定程度上限制了原本具有优势

的扩张性能，有待进一步优化。

一旦触发以上条件进入攻击模式，则我方将依次从存储操作的队列（利用双端队列模拟）中通过 FIFO 顺序弹出操作，按照指定攻击路线行进，直到攻击敌方获胜。

1.1.3 逃跑模式函数

当我方未处于攻击模式且面临危险时，控制流语句将调用逃跑模式函数，尽快返回领地，完成圈地的同时避免敌方击杀我方。可能触发逃跑模式函数的条件较多，具体总结如下：

（1）敌方位于我方领地边缘；

（2）存在被敌方切断纸带的风险（Dekm Dmh+10[未特殊标明单位的数值，均默认其单位为游戏中 1 个单元格。]）；

（3）存在与敌方正碰风险（我方与敌方纸卷距离 Dmh+10）；

（4）在领地近距离范围内缠绕复杂（Dmh $<$ 10，且我方在非初始圈地模式或攻击模式下的已行进步数$>$ Dmh+20）；

（5）游戏即将结束，需要尽快圈地扩张（剩余回合数 Dmh+1）。

在逃跑模式下，我方将计算朝不同走向运动后，下一步坐标距离敌方纸卷及我方领地的距离，并返回尽可能保证安全的走向。

1.1.4 扩张模式函数

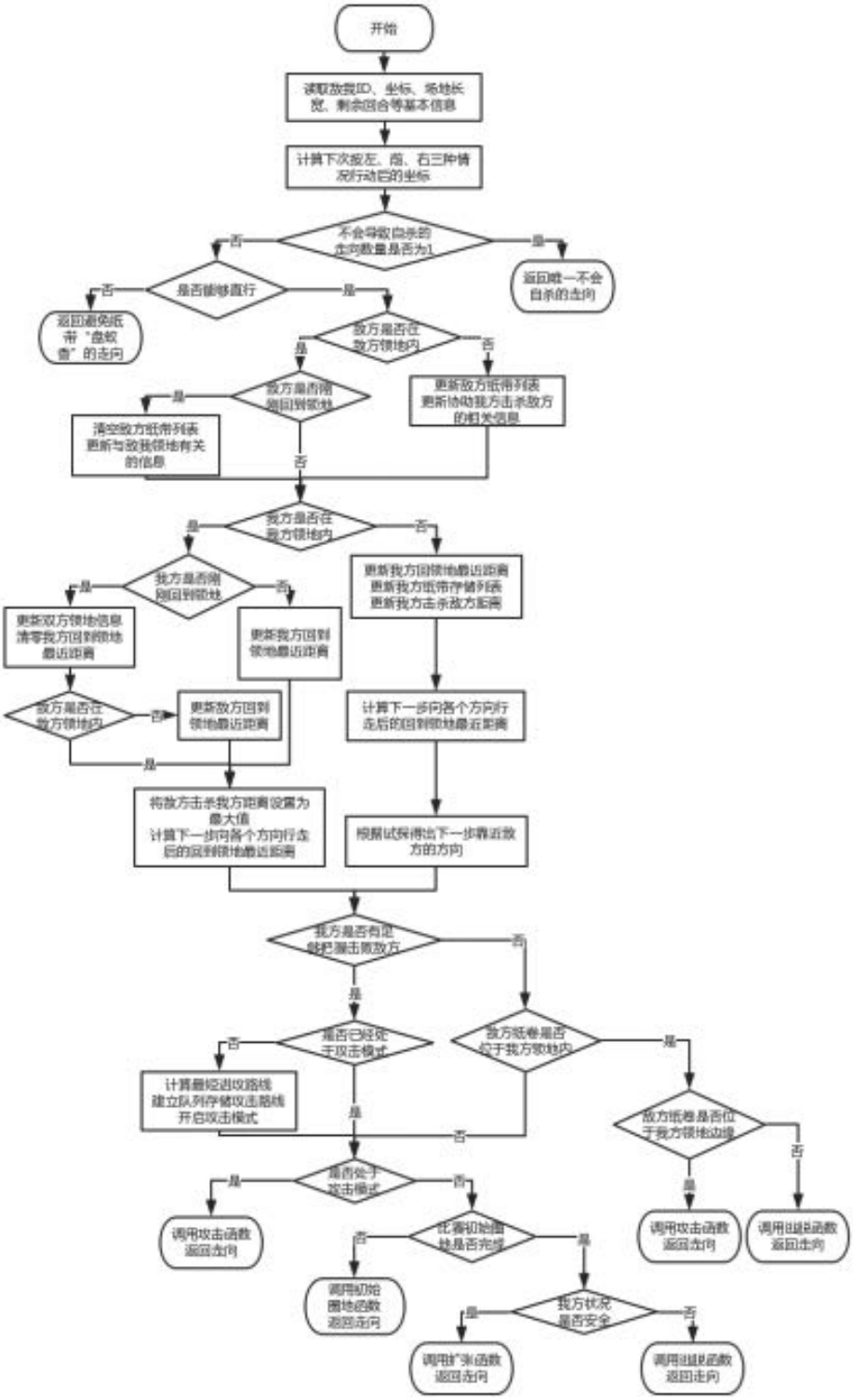
当均不满足以上条件时，视为我方目前安全且不需进攻，控制流语句将调用扩张模式函数，在自保情况下扩大我方领地范围。

在扩张模式下，我方将首先判断自己是否位于领地内。如果位于领地内，则以远离敌方纸卷的方式寻找近距离出口；如果位于领地外，则在避免敌方切断的情况下尽可能扩张。

总而言之，Menhera 的算法吸纳了“贪心”的思想，将“着眼当下”置于“掌控全局”之上，通过化零为整的方式保证了在整盘赛局中的运转进行。

1.2 算法流程图

利用 Process on 在线实时作图工具 [2]，我们绘制了 Menhera 算法中主要函数 play() 单次运行的算法流程图（图 2），示于下方。



Menhera 算法 play() 函数单次运行流程图

1.3 算法复杂度

由于 Menhera 算法涉及在不多种模式下分情况运行，故条件不同时，算法的时间复杂度会产生较大差异。因此，我们将对算法中不同部分在不同情况下的时间复杂度进行详尽分类讨论。

1.3.1 不涉及后续信息更新及模式函数调用的情况

在初始判断时，如果发现仅有一个方向可供运动、无法选择其他方向，或是处于需要避免“纸带盘蚊香”的情况时，则不会进行后续的敌我信息更新与模式函数调用，而是基于较少的信息量返回能够保证安全的走向。

首先，如果发现仅有一个可供运动的方向，此时由于不涉及后续的遍历、计算与比对，算法时间复杂度的数量级仅为 O（1）。

其次，在避免“纸带盘蚊香”的状态下，程序使用了遍历算法，结束条件为完成遍历或者过程中找到了己方纸带。遍历的时间复杂度最小为 O（1），最大为 O（n2）。这是因为，实际情况激活该算法时的一般情况是我方纸卷恰位于己方纸带前，需要盘蚊香算法检测是否会将自己封闭起来；而在我们的算法中，如果纸带较长则会要求自己回到己方领地，该长度参数是固定的，那么将自己封闭起来的范围也是有限的，故此情况下时间复杂度为 O（1）。考虑到 O（1）的极端情况较少，避免“纸带盘蚊香”时的平均算法时间复杂度仍为 O（n2）。

1.3.2 敌我信息更新的时间复杂度

如果敌方或我方纸卷刚刚回到领地，即领地信息发生变化，则需要更新双方领地，遍历整个地图，复杂度为 O（n2）之后判定各个参数是否变化：

如果我方纸卷位于己方领地内，则可能需要更新我方纸卷离最近外出点的距离，在给定范围内搜索外出的最小值，复杂度为 O（1），如果我方距离外出点距离的最小值发生改变，则意味着需要重新遍历搜索离我最近的外出点，复杂度与当前领地大小有关，故无法确定准确值。

而如果我方纸卷不位于己方领地内，则需要在给定范围内搜索距离己方领地的最小距离，复杂度为 O（1）。如果最小值发生改变，则需要重新遍历搜索，复杂度同样与领地大小有关，无法确定准确值。

1.3.3 模式函数判定及调用运行的时间复杂度

如果满足我方开启攻击模式的条件且尚未处于该模式，则需要调用攻击模式函数。首先以 O（n2）的复杂度遍历整个地图寻找敌人的纸带以及攻击点；随后确定进攻路线，即判定选择不同方向后的接近情况，复杂度为 O（1）。即开启攻击模式所需要的总复杂度为 O（n2）。

而如果我方已经处于攻击模式，或是条件不适宜开启攻击模式，则需要按照算好的参数判定当前形势，选择使用的模式：扩张模式、回家模式、进攻模式和方形圈地模式。由于之前已经计算过用到的参数，所以尽管设计许多判断语句，此条件下复杂度仅为 O（1）。

2 程序代码说明

2.1 数据结构说明

Menhera 算法中主要应用的数据结构包括列表、字典与队列（其中队列使用 Python 内置的双端队列 deque 模拟）。以下将对这三种数据结构在该算法中的主要作用进行分别说明。

2.2.1 列表的应用

在本算法中，列表由于其优秀的存储性能及可修改性，主要起到了保存信息的作用。首先，Menhera 算法中使用的全部成对坐标均用长度为 2 的列表存储，既可以按索引提取横、纵坐标的单个数值，又可通过列表形式同时提取，方便了参数的计算。其次，列表还用于存储我方与敌方纸带、领地和空白区域的信息，在局势发生变化时也可及时进行修改。

2.2.2 字典的应用

在 Python 语言中，字典实际上是通过 ADT Map 建构的，其结构是一种键 -值关联的无序集合，而每个关键码都具有唯一性，特点是可以进行高效查找。在本算法中，我们通过字典 storage 存储了大量有效信息，包括目前模式、当前局势、纸带列表等，从而精准快速地辅助算法作出决策。

2.2.3 队列的应用

在本算法的攻击模式启动模块中，为了设计一系列能够避免被打乱的连贯路径，程序会根据启动时的当前局势，通过 attack 函数确定后续运动路线，并且利用双端队列来模拟数据结构队列，利用其 FIFO 的特性存储路径，随后只需依次 dequeue 数据项即可。

2.2 函数说明

在 Menhera 算法中，除了原始模板中已提供的 load() 和 play() 函数外，一共定义了 12 个函数。其中，8 个函数为起到辅助作用的参数计算或数值比较函数，4 个函数为具有模式功能的主要函数（进攻模式、逃跑模式、初始矩形圈地模式、扩张模式）。以下将分别对各函数进行说明。

2.2.1 辅助函数

（1）position(i,j,direction)：用于计算下一步坐标。三个参数均为 int 类型，i 与 j 分别为横纵坐标，direction 为行进方向。

（2）valid_position(i,j)：用于判断坐标的合法性，判断标准包括： 坐标不越界； 不会切断己方纸带； 不会导致被己方纸带包围，无路可走。其中 i 与 j 分别为横纵坐标，均为 int 类型。

（3）min_mh_o(m,n,lst)：用于计算首次离开领地时的最短返回距离。其中 m 与 n 分别为拟定位点的横纵坐标，均为 int 类型；lst 为当前坐标，用长度为 2 的列表来表示。

（4）distance(m,n)：用于计算我方刚离开领地时与敌方纸卷的距离。其中 m 与 n 分别为横纵坐标，均为 int 类型。

（5）updatemo(xx, yy, last)：用于更新我方纸卷当前离空白区域的最近距离。其中 xx 和 yy 为拟定位点的横纵坐标，last 为上次更新时我方纸卷与空白地区最小距离，三者均为 int 类型。

（6）updatemh(xx, yy, last, lastp)：用于更新我方纸卷当前离己方领地的最近距离。其中 xx 和 yy 为拟定位点的横纵坐标，last 为上次更新时我方纸卷与己方领地最小距离，三者均为 int 类型；lastp 为上次更新时存储的我方坐标，用长度为 2 的列表来表示。

（7）updateeh(xx, yy, last)：用于更新敌方纸卷当前离敌方领地的最近距离。其中 xx 和 yy 为拟定位点的横纵坐标，last 为上次更新时敌方纸卷与敌方领地最小距离，三者均为 int 类型。

（8）check(length, centre)：在以上三个状态更新函数计算时，用于核验距离。其中 length 为拟定的距离，为 int 类型；centre 为拟定的坐标，用长度为 2 的列表来表示。

2.2.2 进攻模式函数

进攻模式函数 attack(i, j,dir) 主要用于计算在特定坐标与特定走向下的最短攻击路径，其触发条件已于 1.1.2 节中详细述及，故不再赘述。这里主要介绍其参数与实现方法。

其中，参数 i 和 j 分别为当前横纵坐标，dir 为行进方向，三个参数均为 int 类型。输入参数并调用函数后，程序会读取信息，计算下一步向三个方向行走后的不同状况，包括与敌方纸带的距离以及坐标是否合法，以单步最优为原则，进行一一比对，并确定当前局势下安全且迅速的进攻方向。

2.2.3 逃跑模式函数

逃跑模式函数 go_home() 主要用于计算快速返回己方领地的最佳路径，其触发条件已于 1.1.3 节中详细述及。

go_home() 函数不需额外输入参数，而是直接通过一一比对已计算得出的数值（向各个方向行走后的 Dekm 与 Dmh），以单步最优为原则，判断尽可能远离敌人并快速返回领地的当前步骤最佳方向。

2.2.4 扩张模式函数

扩张模式函数 invader() 主要用于计算快速扩张最佳路径，其触发条件已于 1.1.4 节中详细述及。

invader() 函数不需额外输入参数。执行该函数后，程序会首先判断我方纸卷是否位于领地内，从而以单步最优为原则采取不同策略。如果纸卷当前位于领地外，则会直接通过一一比对已计算得出的数值（向各个方向行走后的 Dekm 与 Dmh），判断尽可能远离敌人及己方领地、从而安全扩大范的当前步骤最佳方向。如果纸卷当前位于领地内，则比对的数值更换为向各个方向行走后的 Dekm 与 Dmo，从而在远离敌人前提下尽快离开领地、开始圈地。

2.2.5 初始矩形扩张模式函数

初始矩形扩张模式函数 square() 主要用于在比赛伊始时，划定一片较大的己方领地范围，从而在初始时获得优势，其触发条件已于 1.1.1 节中详细述及。该函数在初始扩张完毕后即不再被调用。

square() 函数不需额外输入参数。执行该函数后，程序会根据不同初始方向，开始划定矩形。初始矩形划定的情况将被记录在 storage 中，函数将从其中提取有效信息保证正常进行划定，并在圈出第一个矩形后即结束开局扩张。

2.3 程序限制

如前所述，在进攻、扩张与逃跑三种模式下时，程序采用的核心思想为单步最优，并未对全局形势进行考虑；尤其在扩张与逃跑时，算法并未设定一个总体的路径并执行。因此，如果在运行过程中各方信息出现矛盾或局势出现变化，则可能导致纸卷采取形状诡异的迂回路径行进，无法有效利用各回合进行行动（如图 3-1 所示）。此外，由于 valid_position 函数的合法化判定功能仅能对 2 步内避免己方纸带包围的情况进行预判，9 无法进行更为长远的预测，故在迂回路径的情况下，还存在导致“自尽”的风险（如图 3-2 所示）。

图 3-1 迂回路径示意图（红色为我方）

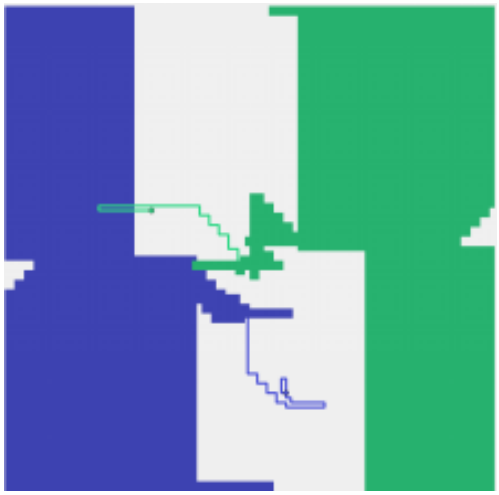


图 3-2 切断己方纸带示意图（蓝色为我方）

3 实验结果

3.1 测试数据

在算法开发过程中，小组内各成员均积极参与了算法测试，故此处列出每位同学在测试过程中使用的实验环境配置（表 1）。

组员姓名	李法承	周赫	李书承	王禹菲	余年年
硬件配置	Intel CORE i5/4G	Intel CORE i5/8G	Intel CORE i7/8G	Intel CORE i5/4G	Intel CORE i5/8G
操作系统	Windows10	Windows7	Windows10	Windows7	Windows10
Python 版本	Python3.6	Python3.6	Python3.6	Python3.6	Python3.6

表 1 测试环境说明表格

3.1.2 测试方法简介

在测试 Menhera 算法的过程中，小组成员主要采用了以下三种实验方式：

（1）与技术组提供的“野怪”AI 对局测试。在程序开发的早期，由于算法还未完全成型，且其他小组开发进程也处于开拓阶段，故初期主要通过与技术组提供的 normal_wanderer 算法进行对战。不过，由于 normal_wanderer 依靠局势作出决策的能力有限，我们编写的早期版本与其对阵时已呈现出绝对的优势，故该种测试方法很快不再使用。

（2）与小组自编程序的先前版本对局测试。该测试方法是我们最为主要的测试方法。通过与先前版本的比较，我们可以更好地确定新版本的优势与劣势，并且作出进一步优化。

（3）与其他小组进行友谊赛。在开发过程中，我们曾经与 F17-KizunaAI（以下简称 K 组）、F17-Lima 与 N17-Xray 三个小组进行过对局。通过友谊赛，参与对局的双方均可注意到自己算法上一些未曾察觉的局限性。有趣的是，K 组的组长与我们小组的核心代码人员是同寝舍友，故在整个开发过程中进行的交流较多。比赛前夕与他们的对决使我们意识到，由于分区内多为扩张型选手，单纯的圈地与扩张算法可能在小组赛中并不占据优势，故最终修改形成了更具攻击性的版本，并在小组赛中以全胜出线。

3.1.3 热身赛结果说明

我们将 4 次热身赛（其中第 4 次为第 3 次在新服务器上的重赛）的结果以表格形式整理如下（见表 2）。由于热身赛中许多小组提交了超过 1 份代码，故为便于观察，本表格中均取每场比赛中每个小组总分最高的版本作为代表，并以组名代码首字母指代小组。表格中“+”表示我方获胜，“-”表示我方失败，“0”表示平局，“/”表示对手未参加该次热身赛。

赛局 \ 对手	0	A	B	C	E	F	G	I	J	K	L	N	O	P	U	Q	V	W	X
Warmup1	/	-	/	/	/	-	/	/	/	-	-	/	/	+	/	+	+	/	-
Warmup2	/	+	+	/	+	+	+	/	/	0	+	/	/	+	/	+	+	-	+
Warmup3	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	-
Warmup3-1	+	+	+	+	+	+	+	+	-	+	+	+	+	+	+	+	+	+	0

表 2 热身赛情况说明表格

由表可见，随着 Menhera 算法的不断更新与完善，在热身赛中的胜率也不断提高。其中，在最后一次热身赛 3-1 中，Menhera 算法更是以优异的成绩获得了全部小组中最高的分数并荣登榜首。

3.2 结果分析

3.2.1 算法策略分析

前已述及，在热身赛与实战对弈过程中，我们使用的算法实际上有较大的差异。我们用于参与热身赛的各个版本代码，均以扩张及防御性能为优势，并且取得了优异的结果。然而，通过分析热身赛的结果及比赛记录，我们发现分区内有大量圈地性能优秀的对手，并且在与他们的某些版本对弈时出现了平局和负局的情况，因此最终改进使用了攻击性强化版本。

实战结果证明，攻击性强化版本在小组赛中以全胜的绝对优势出线，然而很快遇上了一直以极强攻击性著称的对手 F17-Foxtrot 组（以下简称 F 组）。在之前的热身赛中，由于我们编写的防御性算法与他们对决时能够灵活地避免攻击、保证安全，而 F 组的算法又具有较高的时间复杂度，故我方常常可以拖延至对手决策时间耗尽从而取胜。但在正式比赛时，由于进程调度、比赛环境等的区别，F 组并未出现超时现象，我方也由此丧失了优势，最终铩羽而归。

由此可见，无论是在策略性游戏中还是在生活中，根据条件变化作出合理决策，都是一种重要的生活智慧。

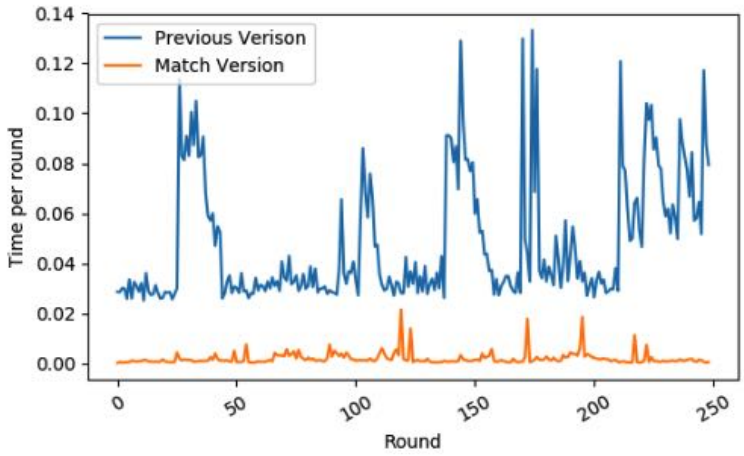
3.2.2 运行时间分析

在 6 月 3 日举行的第一次热身赛中，我们编写的早期算法并未取得理想的战绩。小组

成员随后查看并详细分析了对局结果，发现导致失败的最主要原因是超时——算法中为了计算参数进行了大规模的遍历，此外不少条件判断语句中也需要对列表进行遍历搜索，从而导致耗时大大增加。

经过核心技术人员艰辛的开发历程，最终版本的代码显著减少了所需遍历的次数。尽管主要的耗时步骤仍为遍历，但一方面，算法通过 storage 存储了许多有关信息，并且可以通过时间复杂度更小的算法进行更新；另一方面，在某些不需遍历的条件下，函数将不再进行遍历步骤。改进之后，最终版本算法所需要的运行时间已经可以控制在很低的水平，不再有超时风险。

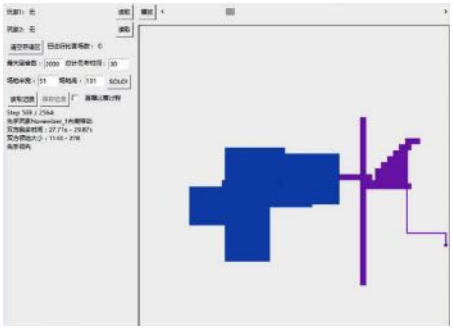
为了将算法时间的改进进行可视化，我们使用编写的最早 Menhera 版本与最终参赛版本，分别与编写过程中一个具有优秀防御性能而无攻击性的版本（该版本代码均作为先手）进行对局，取一次行走完 500 步的耗时数据（测试环境为组长使用的电脑，详见 3.1.1），用 matplotlib 绘制耗时折线图（图 4）并附于下方。由图可见，后续的改进显著优化了运行时长，使得 Menhera 算法具备了更为优异的性能。



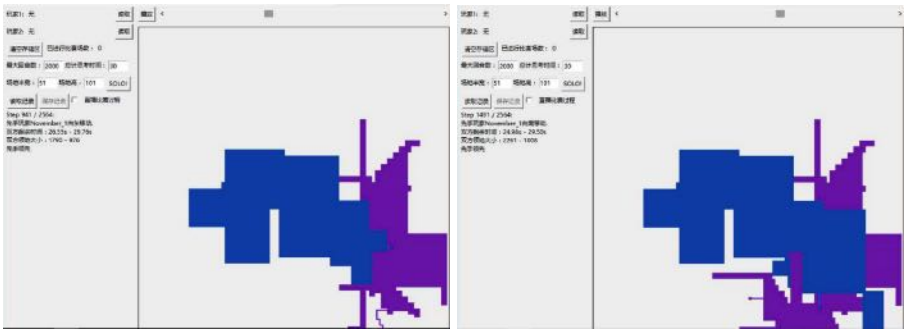
3.3 经典战局

3.3.1 Menhera VS F17-November

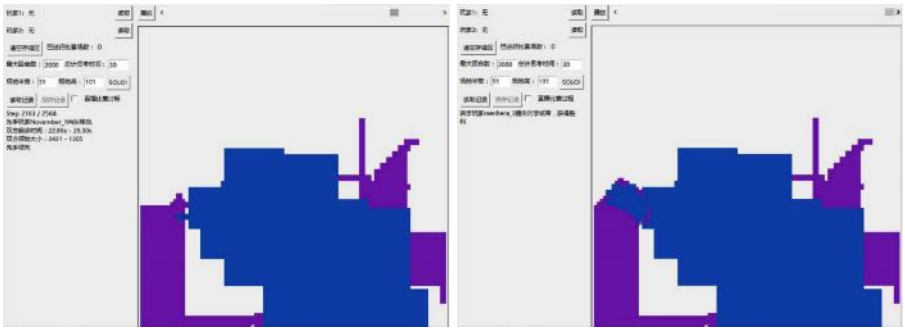
尽管 Menhera 算法在最终正式比赛中仅止步八强，但其在之前的热身赛及小组赛中与 F17 总冠军 November 组对局时均表现出了显著的优势。以下将对 Menhera 与 November 在热身赛中的一局复盘记录进行分析与解读。



如图所示，当时用于参与热身赛的 Menhera 尚未设置开局矩形圈地功能，故在开局时间尚不久时并未表现出明显优势。这种初始时采取长条形的圈地方式，虽有利于辅助搭设后期大范围圈地的框架基础，但面对强扩张型选手时可能存在一定的风险与劣势。



比赛进行了一段时间后，我们可以看到 Menhera 在尽可能避免对手纸卷的接近（该用于热身赛的版本仍为攻击条件判定较弱的版本）。而从比赛耗时可以看出，我方的耗时仅为对方 10% 左右，具有优越的时间性能。

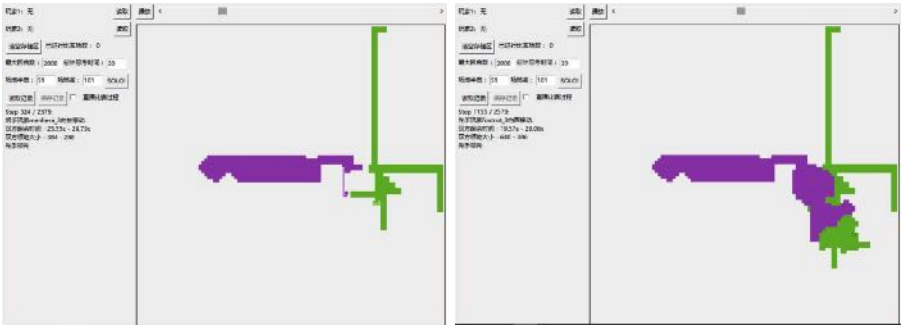


随着战局的逐渐推进，虽然对手圈地暂时领先，但由于其过于接近我方纸卷，露出了一定的破绽。我方及时采取进攻策略，经过一番缠斗，最终切断对手纸带取得胜利。

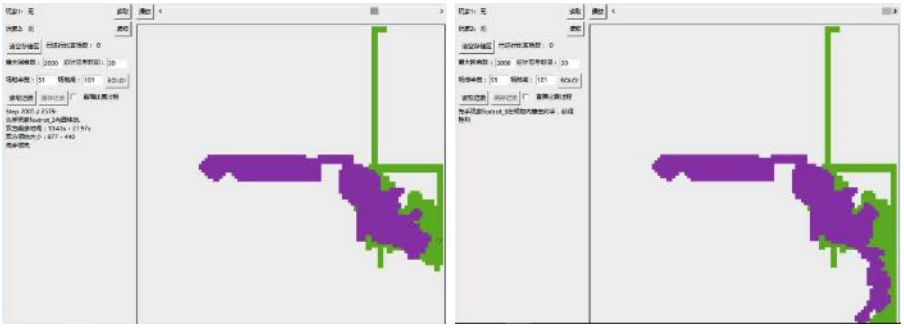
3.3.2 Menhera VS F17-Foxtrot

非常有趣的是，Menhera 算法在八进四比赛中遭遇 F17 的 Foxtrot 组后遗憾落败，而 Foxtrot 组却在决赛中败给了小组赛中负于我方的 November，形成了一种类似于“循环克制”的结构。以下将对 Menhera 与 F17-Foxtrot 在热身赛中的一局比赛记录进行分析。

开局时，我方正常行动，而敌方则意图接近我方进行攻击。



随着敌方的接近，我方试图灵活地作出躲避。而在此过程中，由于我方算法吸纳了局部最优策略的思想，并未考虑总体策略，故领地逐渐被敌方蚕食而难以挽回。不过值得注意的是，在我方仅消耗了 2 秒左右决策时间的情况下，敌方决策时间却只剩余了大约 10 秒。



由于地图边界限制了逃跑范围，而我方又未考虑全局策略，故最终在缠斗至地图低端时被对方撞击而失败。这一事实说明局部最优策略也存在一定的局限性，有待后续改进。

4 实习过程总结

4.1 分工与合作

4.1.1 小组分工

在整个数据结构与算法实习的过程中，小组中的各位成员均积极参与了工作。各个组员负责的主要任务列于下表中（表 3）。

组员姓名	承担工作
李法承	算法思想设计、核心代码编写、代码修改测试
周赫	算法思想设计、核心代码编写、代码修改测试
李书承	算法思想设计、实验报告补充、代码修改测试
王禹菲	算法思想设计、实验报告撰写、代码测试
余年年	算法思想设计、代码注释补充、代码修改测试

4.1.2 小组合作交流

小组成员良好的分工合作是实习进程推动的基础。在完成大作业过程中，我们主要有以下三种合作交流的方式：

（1）组会讨论交流。

我们总共组织了 3 次小组讨论，具体的会议内容及照片记录于下表中（表 4）。

会议次数	第一次组会	第二次组会	第三次组会
会议时间	5 月 22 日 17：00	5 月 23 日 20:45	6 月 3 日 20:30
会议地点	理教 2 层走廊	农园咖啡厅	农园咖啡厅
参会人员	周赫、李书承、王禹菲、余年年	李法承、周赫、李书承、王禹菲、余年年	李法承、周赫、李书承、王禹菲、余年年
成员留影	无	无	如下图
会议内容	提出基本思路构想，确定下次组会时间	Brain storming、确定算法总体思路	算法现场测试及细节改进讨论

（2）成员平时交流。在平时的学习生活中，如果对于算法产生了疑问或改进建议，小组成员均可通过实地交流或社交软件私聊的方式进行沟通。

（3）微信群交流讨论。这是我们小组在后期最重要的交流方式，大家均可在微信群中上传改进版本的代码、测试结果或探讨自己的见解。社交软件为小组成员的良好沟通交流（图 5）提供了便捷的平台。

图 5 小组成员在微信群内愉快地进行交流

4.2 经验与教训

纵观本次实习作业的完成过程，我们认为最重要的经验教训便是：应根据条件变化，合理改进算法设计。

在早期热身赛及小范围友谊战的过程中，Menhera 算法表现出较为明显的优势，甚至在热身赛中曾取得过总排名第一的优异成绩。在早先的几次热身赛中，Menhera 算法均以优秀的防守及扩张性能见长，甚至在第二次热身赛中成为了唯一打败强攻击性选手 F 组的小组。调取复盘记录后，我们发现，其主要原因是我方能够灵活地躲避 F 组 AI 的凌厉进攻；而 F 组算法的时间复杂度明显较高，往往容易超时，在消耗战因耗尽思考时间而落败。

不过，由于竞赛分区内多为强扩张能力选手，小组的核心技术人员编写了攻击性强化的参赛版本。考虑到在后续比赛进程中遭遇 F 组的可能性，两位同学原本准备编写一个专门对抗 F 组的特异性功能模块，但因时间有限最终并未实现。在正式竞赛时，参赛算法于小组赛阶段确实进行了高水平的发挥，后来却败给了之前与我方对局时一直处于劣势的 F 组，未能取得理想的成绩。而最终打败 F 组、获得 F17 联盟总冠军的 November 小组原本在小组赛时负给了 Menhera 算法，这一事实也让我们意识到，paper.io 这种策略性游戏中或许不存在绝对的强弱，而是如同口袋妖怪（Pokemon）游戏中每一世代的“御三家”相互循环克制一样 [4]，不同种类的策略间存在“属性克制”的关系。根据外部条件的变化合理改进算法设计，才是一种智慧的取胜之道。

当然，这种改进算法设计的思想，不仅仅可以用于 paper.io 游戏人工智能算法的设计。例如，我们在作业中曾经研究过快速排序（quick sort）算法的取中值策略——当列表中数字随机排列时，取列表的第一个元素作为参考值是一种妥当的方式；然而，如果列表有一定“人为”的顺序，可以考虑采取随机抽取元素作为中值的方法。显而易见的是，在列表顺序完全随机的情况下，调用 random 库、抽取元素及交换反而会耗费一定的时间，不如简单地抽取首位元素（见图 6-1）；而当列表有某些规律时，随机中值法却又呈现出了明显的优势（见图 6-2）。

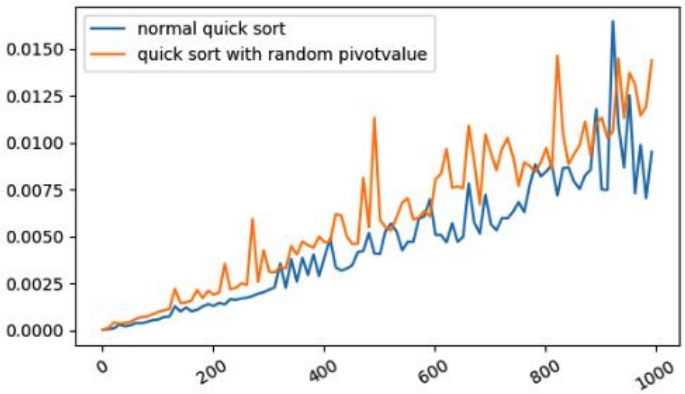


图 6-1 两种取中值方法对于随机列表排序的耗时示意图

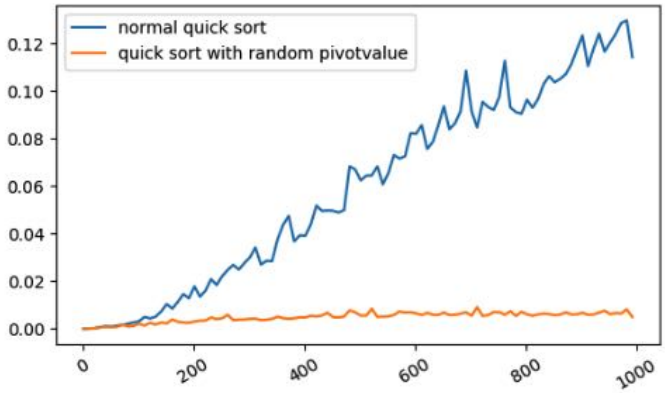


图 6-2 两种取中值方法对于递增列表排序的耗时示意图

总而言之，本次小组作业带给我们最重要的经验教训，便是应当学会根据条件变化合理改进算法设计，并且将这种思想运用于以后的学习与工作之中。

4.3 建议与设想

实习数据结构与算法 B（Python）课程的本次大作业，确实为我们带来了不少宝贵的经历与收获；老师、助教们的辛苦付出，以及技术组成员的高超编程能力，也让我们感到钦佩不已。为了能够让课程的这一实习活动起到更佳的教学结果，我们经过讨论提出了如下建议。

首先，我们认为目前采取的“世界杯”赛制可能会导致竞赛结果具有较大的随机性，在某种程度上或许并不能给予全体同学一个公正、公允的反馈。由于分区的随机性，我

们观察到，许多原本具有强大实力的小组，甚至未能在平均水平较高的区内出线——例如，上文所提到的 K 组，其在热身赛中往往名列前茅，甚至曾在比赛前夕的友谊赛中与 Menhera 算法对决时取得显著优势，却最终未能出线。而经过了解后，我们得知信息与科学技术学院开设的多门编程课程（如计算概论 A、程序设计实习等）中也有编写游戏 AI 的实习作业，但区别在于其比赛是通过“天梯”排位赛的方法进行。考虑到排位赛中运气所能起到的影响较小，以及排名也可以作为连续化得分评估的更好依据，我们提出如下期望与建议：如果条件允许，希望在以后课程的比赛中，可以采取类似的“天梯”排位赛评估方式，以让各位参与实习作业的同学获得更加合理而公正的结果。

其次，关于竞赛现场奖励小零食的设置，我们建议以后选择一些更易清理、有助于维持“品质校园”良好环境的食品。本次比赛提供的“大大卷”泡泡糖确实勾起了大家的童年回忆，但如果有同学将食用完毕的泡泡糖未能进行妥善处理，可能会引起一些麻烦。尽管同学们并不一定是有意而为之，然而若泡泡糖无意间粘在了桌兜内、地面上、墙壁上甚至垃圾桶盖上，或许会给其他学生或本已很辛苦的清洁工带来更多困扰。相比之下，前几届比赛时提供的糖豆或饼干不失为一种更好的选择。

这次实习作业的组织与开展过程总体而言十分顺利，但我们相信，老师与助教也会在条件允许的情况下进一步改进和完善比赛的结果与过程，让数算课程锦上添花。

5 致谢

首先，我们非常感谢陈斌老师、助教学长学姐与北京大学地球与空间科学学院数据结构与算法 B 课程（以下简称 SESSDSA）为我们提供了本次参与竞赛的机会。SESSDSA 在信息科学教学上所作出的大胆创新实践，是各位同学有目共睹的。相比传统教学方式下枯燥的刷 OJ、机考，这样趣味的实习作业不但为我们带来了成就与挑战的乐趣，更能够提升我们应用 Python 语言编程的实践能力。

其次，我们还要感谢两位技术组学长在比赛平台建设与规则完善过程中付出的辛勤努力。在同学们都处于繁忙之中的期末季，两位学长反复进行开发与修改，并且还在课程群与组长群中解答了大家的疑惑，值得各位同学钦佩。

再次，感谢信息科学与技术学院 2017 级人工智能方向的阮思凯同学在实习过程中抽出时间为我们解答了一些疑难问题，并协助小组成员进行了代码修改，推动小组实习工作顺利完成。

此外，感谢 F17-KizunaAI 小组、F17-Lima 小组和 N17-Xray 小组与我们多次进行友谊对战，帮助我们进一步完善算法思想。

最后，可爱的 Menhera 酱为我们提供了名称灵感，并且担当了小组的看板娘! 我们衷心地向 Menhera 酱和画师ぽむ表示感谢!

6 参考文献

[1] 作者ぽむ的 Pixiv 主页：https://www.pixiv.net/member.php?id=2302136

[2] 萌娘百科 -七濑胡桃：https://zh.moegirl.org/%E4%B8%83%E6%BF%91%E8%83%A1%E6%A1%83

[3]Process on 在线实时作图工具：https://www.processon.com/ [4] 神奇宝贝百科：https://wiki.52poke.com/wiki/%E4%B8%BB%E9%A1%B5

第十五章　F17_November 报告

陈逸凡 *、洪涵渝、顾炜东、陆韦宇、徐武辰

摘要：本小组使用树的广度优先搜索查找敌方攻击距离、我方攻击距离、敌方回领地距离、我方回领地距离，以此作为决策的基础，在正方形圈地、进攻、领地内游走等模式之间切换。强制使得运动路径为正方形，基于这一点对算法进行了简化，因此本组决策极其简单，报告较为简短。实际效果良好，在竞赛中取得冠军。

关键字：嵌套列表；递归算法；广度优先搜索；动态规划

1 算法思想

1.1 总体思路

这个比赛想要取得胜利，无非两种情况，要么圈地面积大，要么将对手击杀。再观看了热身赛记录后，我们认为大部分组偏向保守算法，因此提高圈地效率很重要。显然，正方形是效率最高的。在开局，经过讨论，自己后方的土地是迟早得获得的，所以开局采取向后圈地策略，使得自己的领地交大。另外，对于疏于防范的圈地狂魔，我们设计了发起进攻的算法，计算相关距离，满足条件时就按照规划的路线发起进攻。总体来说，这是一种保守的算法。不过几乎所有组的算法都比较谨慎，我们的圈地速度优势很明显，这支持我们最终获得了冠军。

开局在保证自身安全的情况下向后方尽可能大地圈地，确保占有足够的领地。之后使用树的广度优先搜索查找敌方攻击距离、我方攻击距离、敌方回领地距离、我方回领地距离，在保证安全的情况下向着对手方向圈地，这样能对手造成干扰，吞并对手领地。当对

手距离我过近时，停留在领地内转圈保证安全，当对手回领地的最短步数大于等于我进攻对手的最短步数时主动进攻，计算最短距离，利用循环递归找到进攻对手的第一步。

此算法较为简单，易于实现。

1.2 算法流程图

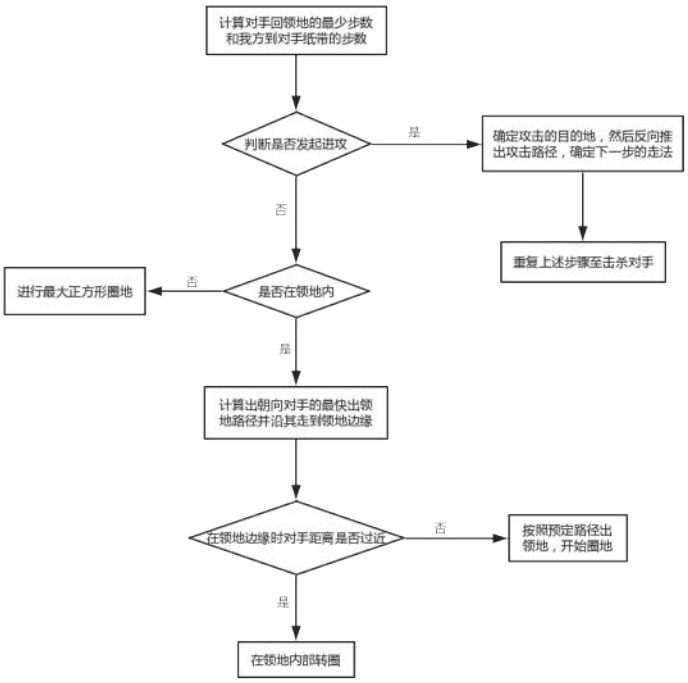


图 1. 本组总算法流程图

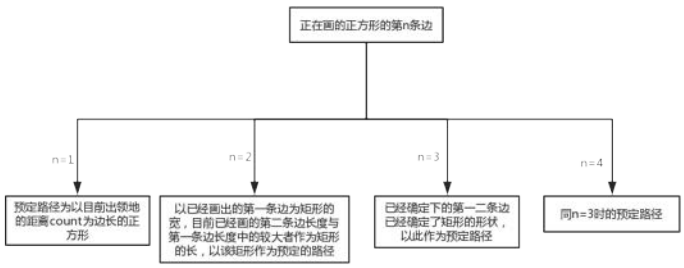


图 2. 圈地时预定的矩形轨迹确定方法

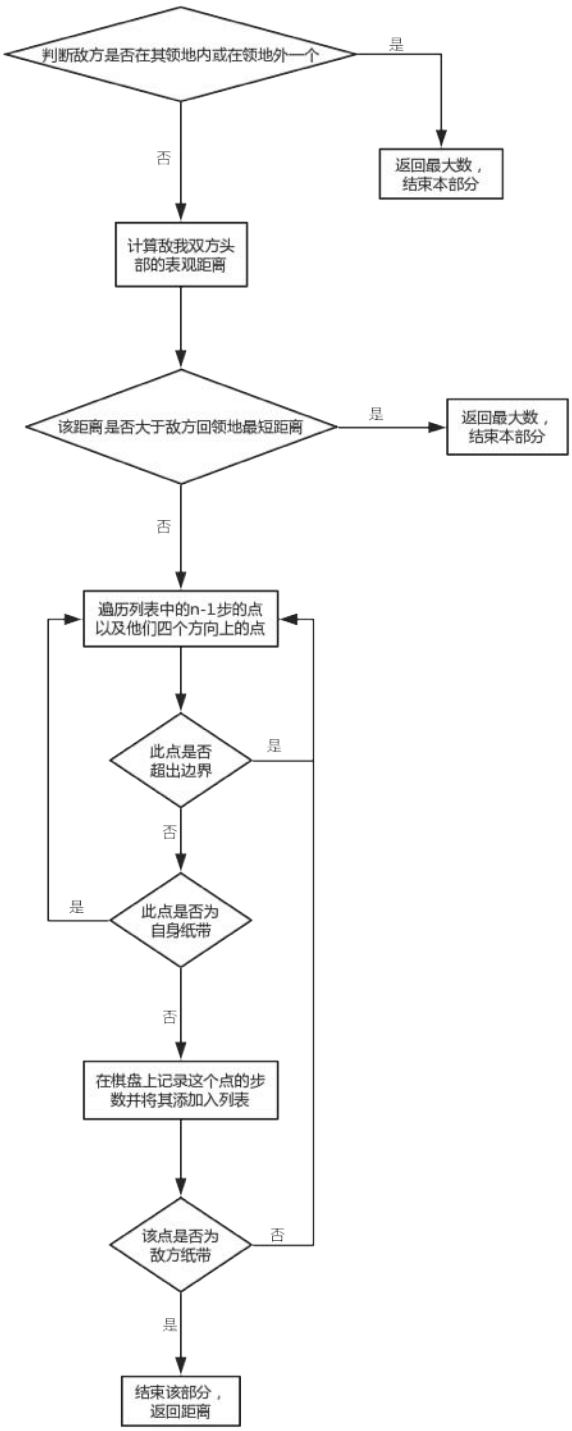

判断敌方是否在其领地内或在领地外一个
是
返回最大数，结束本部分
否
计算敌我双方头部的表观距离
该距离是否大于敌方回领地最短距离
是
返回最大数，结束本部分
否
遍历列表中的n-1步的点以及他们四个方向上的点
此点是否超出边界
是
否
此点是否为自身纸带
是
否
在棋盘上记录这个点的步数并将其添加入列表
该点是否为敌方纸带
否
是
结束该部分，返回距离

图 3. 计算距离部分逻辑

1.3 算法复杂度

1．构造嵌套列表储存坐标时间复杂度为 O（n*n）

2．判断敌我双方是否具有领地时遍历全地图 O（n*n）

3．基于正方形路径估算敌方身体到我方纸带的最短距离时间复杂度为 O（n）

在实际运行中，消耗时间一般不超过 10 秒

2 程序代码说明

2.1 数据结构说明

2.1.1 列表数据结构

圈地时采用效率最高的正方形式圈地，在计算最大圈地方法时使用一个列表储存正方形的四个顶点（Node1、2、3、4）所在位置，位置确定方法如上面给出的矩形轨迹确定方法说明图。

2.1.2 嵌套列表数据结构

计算相关距离时采用的结构是嵌套列表。一个大的列表中包含着很多个列表，这些列表的序号表示这些列表内部的坐标与出发点的距离（此处的距离指的是能避免撞到自身的距离），计算时利用上一个列表中所有坐标，向四个方向拓展，找到符合要求的点添加进新列表。对于寻找路线，从目的地往回找，根据地图上标记的距离，每次找到距离减一的点，循环往复，直到找到当前位置，并给出下一步的位置。

2.2 函数说明

2.2.1 辅助函数

(1) 敌方攻击距离 enemyp(node,enemy) 由于保证我方路径为矩形，则当敌方头部的 x 值处于 Node2、3 的 x 值之间时，最短距离即为其头部 y 值与 Node2、3 的 y 差的绝对值

中的最小值，当敌方头部的 y 值处于 Node2、3 的 y 值之间时同理。两条件均不满足时最短距离为敌方头部到四个顶点距离的最小值。

(2) 回领地距离 homep(node,me,field,storage) 由于保证我方路径为矩形，则仅须遍历该路径，计算非我方纸带和非我方领地的长度即可。

(3) 危险函数 isDangerous(field, nextme, storage, enemy) 通过比较上述两个函数的返回值判断我方是否处于危险。

2.2.2 领地内寻找出领地路径函数 fstart(field,me,enemy,storage) 计算朝向敌方的两个相垂直的方向中出领地最快的方向，确定方向后直线运动出领地。

2.2.3 正方形圈地函数 square(field,me,enemy,storage) 使纸带在确定条件下转弯的函数，转弯的条件一是到达边界；条件二是遇到危险，由辅助函数 isDangerous() 判断。

2.2.4 转圈函数 curcle(field,me,enemy,storage) 在即将出领地时进行判定，若敌方距离太近，则直接在领地内以边长为 1 的正方形转圈，待敌方远离后再出领地。

2.2.5 攻击函数 attack() 当使用广度优先算法计算出精确的攻击敌方距离小于精确的敌方回家的距离时会主动出击，撞击敌方。

因为简化了算法，函数本身的计算比较简单，较为关键的是 4 个 Node 的确定。这种算法类似动态规划，当前状况下能够走的最大矩形取决于前一步的最优解，选择实现拟合路径或者扩大拟合路径的规模，从而得到当前步骤的最优解。

2.3 程序限制

本小组在参加热身赛时可能会出现的报错是 tuple 指数越界，原因是在领地内游走时，若己方头部与敌方头部处于一条直线上时，无法判断哪一个方向是朝向敌方的方向，于是会沿直线走至边界，导致 tuple 指数越界。

经过修改，最终版本在比赛中并未报错，因此我们没有确定可能导致报错的极端条件。但在以下的几种情况会输。

在正方形圈地函数的执行过程中，如果己方领地被吞，则会导致原本找出的最优解无法实现（回领地距离突然变远），可能导致被击杀。

转圈函数执行过程中可能因为领地宽度为 1，转圈也会出领地，然后被击杀。

攻击函数触发时未考虑进攻的同时自己也有可能被撞击。

我们采用了比较保守的圈地方法，可能在圈地效率上会遇到一些问题，但从比赛结果上看来，问题不是很大。

3 实验结果

3.1 测试数据

实验环境说明：

仅使用陈逸凡同学电脑作为测试环境

- 硬件配置：i7-6700HQ CPU 内存：8GB
- 操作系统：Windows10
- Python 版本：Python3.6

最开始利用课程网站自带的 ai 进行测试，能全胜后与自身进行对战，发现其中一方被撞后修改程序，直到几乎每次与自身对战都能走完 2000 回合。在热身赛以及自己组织的友谊赛发现撞墙 bug 以及另外未考虑到的情况，及时修改。

由于小组赛分组中本小组最强的对手是 f 组与 v 组，我们特地找出与 f 组和 v 组在最后一次热身赛中的对战记录（本小组只参加了这一次热身赛）。

以下是热身赛中和 v 组的对战记录

场次	步数	领地大小 (n-f)	胜负	胜负原因
1	2440	3650 - 1053	胜	撞击对手纸带
2	1163	1970 - 800	负	在对手领地内被对手撞击
3	905	1632 - 521	负	在对手领地内被对手撞击

以下是热身赛中和 f 组的对战记录

场次	步数	领地大小 (n-f)	胜负	胜负原因
1	313	393 - 380	负	在对手领地内撞击对手
2	283	366 - 271	负	在对手领地内被对手撞击
3	330	397 - 152	负	在对手领地内撞击对手
4	519	342 - 203	负	在对手领地内被对手撞击
5	359	418 - 242	负	被对手撞击纸带
6	359	480 - 446	负	在对手领地内被对手撞击
7	615	380 - 372	负	在对手领地内被对手撞击
8	304	330 - 166	负	在对手领地内撞击对手
9	314	350 - 124	负	在对手领地内撞击对手
10	327	460 - 256	负	被对手撞击纸带
11	367	369 - 169	负	被对手撞击纸带

热身赛时尚未添加转圈函数，因此总是在离敌方很近时出领地被撞死，修改之后在小组赛中成功击败 f 组与 v 组出线。

由此可见热身赛的重要性以及最终的胜利需要不断地发现 bug 并修改！

3.2 结果分析

小组赛中最大的敌人 foxtrot 和 victor

在与 foxtrot 的战斗中，我们的正方形圈地以及开局的向后圈地发挥了重要的作用，我们两种算法在领地交界处的处理很接近，因此经常在领地交界处各自转圈耗尽回合，利用开局的圈地优势取胜。

在与 victor 的战斗中，我们的向对方纸卷位置圈地策略发挥了重要作用，与 foxtrot 不同，victor 选择全地图逃跑，因此追着他圈地能很有效地扩大自己的优势，唯一需要注意的是要避免在地图边界撞墙。

算法消耗时间在预想范围之内，主要在于计算距离方面消耗。

决赛仍与 f 组对战，对方拿出一个尚未完善的圈地函数，因此被我们成功击杀取得胜利。

3.3 经典战局

以下是与 foxtrot 在决赛中的对局

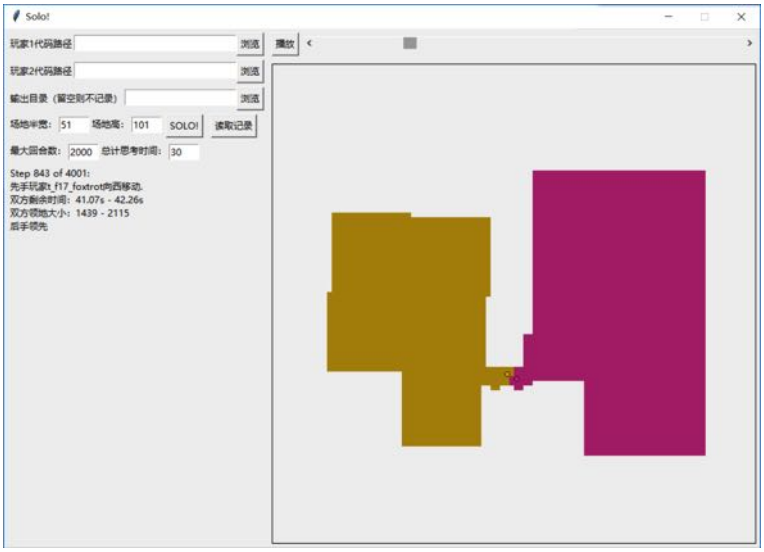


图 4. 决赛局 1

在这次比赛中，我们在开局的较快圈地攒足了缠斗的本钱，在进行到八百多步的时候，我们和对手在交界处各自打圈，因为我们领地较大，最终获胜。

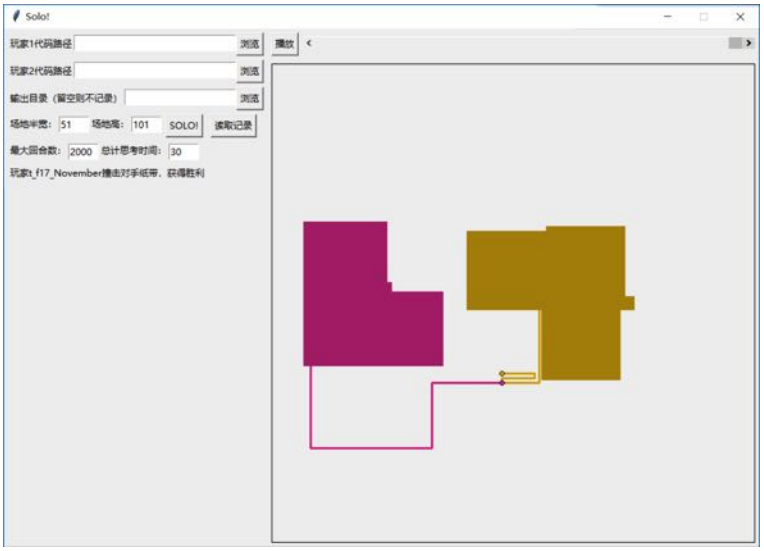


图 5. 决赛局 2

在这个场次中，我方罕见的出现了非正方形的圈地路线，这是因为在某一时刻，经过计算发现我进攻对方纸带距离小于对方回领地距离，于是我方放弃圈地，按照设计的路线将其击杀。

4 实习过程总结

4.1 分工与合作

陈逸凡负责编写计算距离模块，寻找路径模块，参与线下热身赛发现问题；洪涵渝负责编写结点圈地模块，跟踪圈地，以及在领地交界处的细节处理；其他三位同学参与算法总思路，大方向的讨论提出。

本小组第一次在教室共同讨论确定大方向后，接下来的工作为各自完成，线上交流。

以下是第一次组会的照片。地点：四教 509

图 6.. 第一次组会照片

4.2 经验与教训

最重要的就是一定要参加热身赛，就算没办法参加也要参加线下的友谊赛，可以找到自己程序的很多问题。我们取胜的关键就是圈地效率很高，并且在正面交锋时采用保守算法。由于时间仓促，在一些细节上比如自己领地突然消失没有妥善处理，所幸比赛时没有出现这种特殊情况。

在编程方面，不用急于一下子就写出最完善的代码，先写出基本的框架，再加以完善。

写代码时应该做好注释，在完善的时候更加方便。

本组成员均比较拖延，因此本组的代码写的很紧急，对于参赛者来说，一定要早早的找到队友并且早点开始工作。

4.3 建议与设想

本次作业的技术组非常优秀，比赛设施很完备，没有出问题的地方。此次竞赛中大家采取的策略较为一致，希望能够采用允许多样化策略的竞赛规则。

采用小组赛加淘汰赛赛制在一节课内结束比赛有一点点草率，我认为可以延长比赛时间，例如采用一个星期作为一个赛季进行排位赛，每日规定参加排位次数，最终以段位高低分名次。

通过这门课的实习作业, 可以提升自己编程技能，感受 debug 的“快乐”，希望学弟学妹们能积极参与实习作业。

5 致谢

感谢陈斌老师！

感谢各位助教的辛勤付出！

感谢技术组陈天翔，张赖和同学！

6 参考文献

http://gis4g.pku.edu.cn/

第十六章　F17_Oscar 报告

黄赞佑 *、谢东兴、杨恒、段彦琛、姚正

摘要：纸带圈地取胜的核心就是在规定时间内使圈地面积最大或主动出击截断对方获胜，因此参与此次比赛的各组的思想不外乎三种：进攻、防守以及攻防兼备。我们最终的算法思路是离敌人较近时主动攻击，离敌人较远时复制对方。算法的主体包括防撞墙的 wall 函数、攻击敌人的 Kill 函数、在后手起始阶段复制对方操作的 copy 函数以及 play 函数等，主要用到了字典和列表的数据结构。比赛结果为未出线。

关键字：列表，字典，算法；

1 算法思想

1.1 总体思路

纸带圈地的游戏规则是在规定时间内圈地面积最大者获胜，此外，在最大时间内，双方也可主动出击，成功截断对方者胜。因此我们制定了双方距离较大时，敌不动我不动（毕竟不知道对方会采用何种策略，这样胜在稳妥）、当双方距离较近时，则主动寻求时机截断对方纸带的主体思路。

1.2 算法流程图

前提一：保证不碰到墙。—> 前提二：保证不会碰撞自己的织带。—> 判断跟地方的距离选择 1 或 2。—>1：如果接近，就找到最近的织带点而攻击。2：如果较远，copy 对方，等待能够攻击的距离。

1.3 算法复杂度

本算法时间主要用在起始阶段采集对方行动数据及复制和近处主动出击上，由于游戏中设置了思考时间这一限制，我们尽量简化算法，最终算法的复杂度为 O(n)。

2 程序代码说明

2.1 数据结构说明

本组的代码中没有使用太多的数据类型，主要使用的就是基础的 list 类型。在我们的代码中，我们认为这些基础的数据结构已经足以完成我们的代码。结构使用了较为简单的树形结构，逻辑容易理解，对于数据的处理使用了多函数来表示，在结构的建筑上采用了函数互相嵌套的方式，

2.2 函数说明

本程序的代码中有 wall、kill 和 copy 三个函数。

wall 函数的主要功能在于防止 ai 撞墙。函数接收 me、enemy、band、field 几个参数。并且分为 x 轴和 y 轴两个方向来讨论。以 x 轴为例，先求出下一步的格子的 x 轴的坐标。如果不改变方向确实就会撞墙的话，那么就用 choice 函数随机选择一个方向掉头。对 y 轴的讨论相同。如果不会撞墙的话，那么就进入 kill 函数。

kill 函数的功能在于防止撞上自身的纸带。它接收 me、enemy 和 band 三个参数。在确保纸带不会撞上自己的纸带的情况下，纸带按照原来的方向前进。如果在不改变方向的情况下确实会撞上纸带，那么则通过对附近的环境的探查来选择一个尽可能远离自身纸带的方向来移动。

最后的函数是 copy 函数。它接收 me、enemy 为参数。它的主要功能在于在安全的情况下，通过模仿对方的 ai 的行动来行动。并且使得呈中心对称的形势。

最后，三个函数之间可相互切换。实现不同状态之间的切换。

本程序的基本设计思想是首先保证不撞墙，然后不撞到自己的纸带，然后进行模仿。这样是希望在减少冲突的情况下保持至少与敌方相似的领土面积。

2.3 程序限制

本程序在一些情况下会出现严重的失误。这在之前没有发现，实在是个重大失误。本程序在一般情况下会模仿对方纸带的行动，但是由于双方的地形不同，已经走过的路也不同，导致本程序的其他部分，如 wall 函数和 kill 函数会对模仿过程出现干扰。这个对战方式是整体连贯性不好，由于只考虑了每一步之后要怎么走而没有一个比较有大局观的方案导致可能自己的措施比较机械而出现失误。这样就使得模仿过程出现变形，从而十分容易出现撞墙和装纸带的错误。

3 实验结果

3.1 测试数据

硬件配置：CPU：Intel® Core™ i7-7700HQ；内存：16G；操作系统：windows10-1803；Python 版本：3.6；

3.2 热身赛结果

所有参与热身赛的队伍进行了 100 次随机比试，并统计了结果，本组由于技术问题，结果为 0 胜。热身赛结果统计数据如下图：

3.3 比赛结果与分析

本组被分到了 F17 的 South 区，经过区内 6 个队伍循环比赛，本组最终 0 胜，未能出线。一般情况之下能够正常运行，但出乎我们的意料，其他小组的程序都不愿意攻击对方，而是比较防守为主。在跟技术组员提供的简单 AI 的比赛中，能够占优，但对方保守时，程序总是找不到获胜的方法。

4 实习过程总结

4.1 小组讨论记录

第一次讨论（2018.5.27，晚八点于理教一楼）：

本次讨论因时间匆忙，未能拍照，特附录当时群内聊天记录。

本次小组会议主要讨论了分工问题，最终确定了由杨恒、段彦琛、姚正三位同学进行

算法程序部分的编写，其余黄赞佑与谢东兴两位同学进行实习报告的编写。

接着我们对算法的主要思路进行讨论，最初我们准备完全采用防守策略：通过收集每回合结束后双方 x 与 y 轴上距离之和来确定我方所围正方形的大小，即：我方每次所围正方形的边长 =[（双方在 x 轴上的距离 + 双方在 y 轴上的距离）]/4。由于时间关系，未能得到最终的结论。

第二次小组讨论（2018.5.28，晚八点于理教二楼）：

因为前一天对本组应该采用什么策略未能达成一致意见，我们第二天在理教二楼进行了第二次讨论，最终决定由三位同学各自编写自己的算法，最终选出较好的那一个。

4.2 分工与合作

经过讨论，我们最终制定了组员的具体分工。由杨恒、段彦琛、姚正三位同学各编写一种算法。然而由于各种原因，段彦琛、姚正两位同学的程序未能最终完成。所以这两位同学负责了对杨恒同学所编写算法的 debug，以及实习报告关于算法程序分析部分的编写。黄赞佑与谢东兴两位同学对整个过程进行跟踪记录，最后完成实习报告。具体分工如下：

杨恒：代码编写

段彦琛、姚正：代码 debug，实习报告程序代码分析部分编写

黄赞佑、谢东兴：实习报告的编写，跟踪记录，及时反馈技术组要求的变动

4.3 经验与教训

此次小组组成时间较晚，使得小组内程序的编写进行得时间紧促，刚刚赶上热身赛的末班车，也因此对算法未能进行进一步地优化，使得算法中有一个比较明显的缺陷，在边界附近易出现不平常操作导致撞墙。

4.4 建议与设想

在观看比赛的过程中，我们认为对前两名的判定（当有三组及以上分数相同时）有些问题，两组分数相同时用单位时间的占地面积来区分高低有失公平，我们觉得是否可以在下一次比赛时采用当两组分数相同时，通过观察这两组在同一场比赛的数据，让获胜方晋

级的区分方法。

5 致谢

感谢陈斌老师开设这样一种作业形式

感谢陈天翔和张赖和两位技术组员开发了本次比赛的平台与维护

感谢其他小组

感谢本组所有成员的共同努力，虽然没有编程方面的好基础，大家尽到了最大的努力

6 参考文献

《problem solving with algorithms and data structures using python》

—Bradley N.Miller , David L.Ranum

参考的链接：http://interactivepython.org/runestone/static/pythonds/index.html

第十七章　F17_Papa 报告

陈丹丘 *、马涵聪、冉瑾瑜、郭天傲、陈麒安

摘要：我们小组为了实现纸带游戏的 AI 提出了两套方案，其中一套以进攻为首要战略，其出发点是各组 AI 编写不完备导致未考虑周到因素的存在；另一套则是以稳健的风格为主，能够确保纸带的生存。虽然我们在比赛中采用了后者，但我们在对前者的研究中找到了进攻情况下的必胜法，因此亦在本文中留作记录。

关键字：图 BFS 进攻必胜法

1 算法思想

1.1 总体思路

Paper-io 纸带游戏是一款完全信息决策的游戏，根据游戏的获胜条件，我们可以把游戏策略归类为进攻、退防、圈地三大类。我们小组采用的总体思路是根据当前回合敌我双方的情况来调用不同的操作模式，总结起来就是小组大大郭聚提出的战术思想“看一步，走一步；施加压力，全面进攻”。

经过小组成员们的分析，这次大作业不仅仅是算法的比拼，更是代码编写者之间的心理博弈，化用毛主席的游击战十六字口诀，我们可以得出以下细分策略：

1. “敌退我追”，如果对方的 AI 侧重防守，预判到对手未来一两步的进攻会给自己带来威胁从而回防，那么我方主动进攻能够扰乱其行动轨迹，降低其围地的面积；

2. “敌驻我扰”，如果对方的 AI 侧重围地，在判断对方未能造成进攻威胁的时候采用极度优化的方法来进行围地，那么我方主动进攻能够降低其围地效率，并且伺机击杀对方；

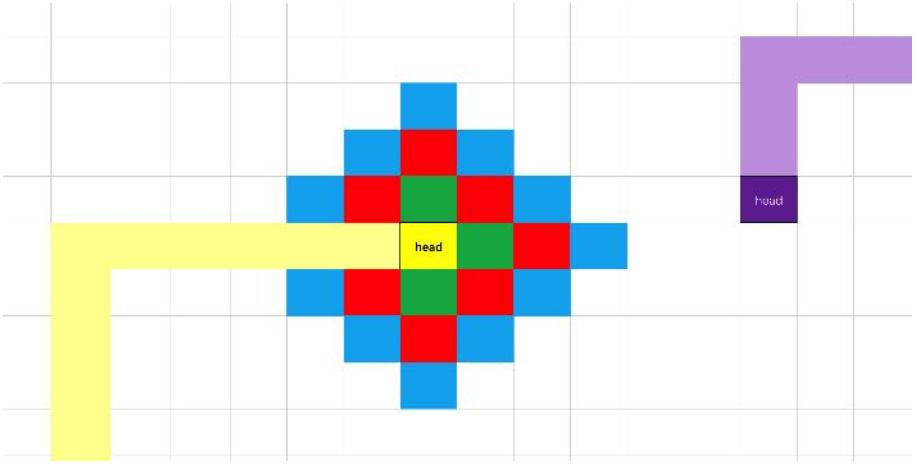
3. “敌进我迎”，如果对方的 AI 侧重主动进攻，那么我方就需要和对方在边值条件、特殊情况的判断上进行比拼，这种情况催生了我们后来发现的“必胜法”；

4. “敌疲我打”，如果对方的 AI 侧重防守反击，即围出了一定面积的地盘后即在领地内等候击杀我方，那么我们只需要在其领地外围巡航，这条巡航路线只需要满足：1. 对方走出领地后无法保证彻底击杀我方；2. 我方潜在的围地面积在不断增加，那么即可根据对方的策略做出相应的应对：1. 若对方离开领地与我方对决，则问题化归为上一种情况；2. 若对方一直在领地内等待进攻，那么我方只需要在剩余回合数花光之前走一条对方无法击杀我方的路径返回领地，则可以靠领地面积获胜。

综上，对对方 AI 施加进攻压力的策略是比较占优势的。“大剑无锋，大巧不工。”在明确“进攻”是我方的首要战略目标之后，我们的“看一步，走一步”战术也随之变得更有针对性了，这体现在：

1. 操作模式的选择上，只要对手（在安全步数内）不会必杀我方，我方就选择进攻，而不是圈地；如果对方对我方发起进攻，我们将主动迎击，而不是退防。

2. 思考时间的分配上，我们会让 AI 对未来两到三步（49 种情况到 144 种情况，未剪枝）进行广搜以确保进攻的有效性，同时在判断安全距离的时候则采用时间复杂度较低的范围判别法。



（预测未来几步，对方同理）

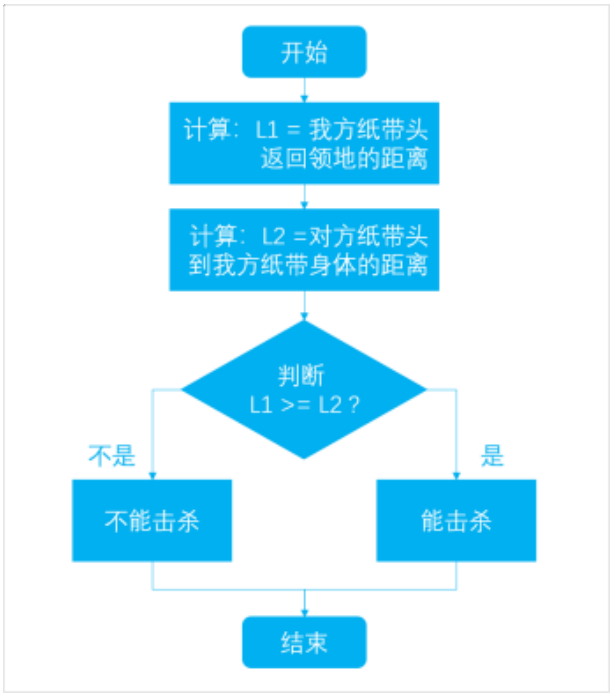
（范围判别法示例）

前文提到的“必胜法”在下文的“经典战局”部分中进行解释，此处不加赘述。

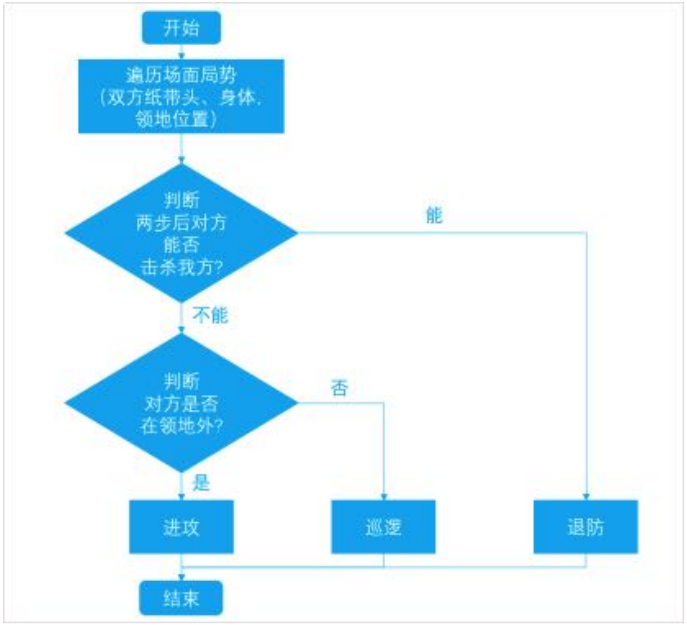
不过实际上，我们小组并没能实现这套方案，在规定的时间内，我们只实现了简易版的 AI，由于它的进攻能力非常弱，所以我们把圈地作为第一战术，主要的优化都围绕圈地效率展开，但最终因为安全距离的判断失误，我方 AI 极易被击杀，这也直接导致了我们没能从小组出线的败绩。

本实验报告希望从我们小组构思的理想版本 AI 和最终成型的现实版本 AI 两方面同时展开，一来可以从多角度体现我们小组对本次大作业的理解与思考，二来为今后的学弟学妹们提供反面案例，望其汲取教训，探索理想算法与具体实现之间有效的桥梁。

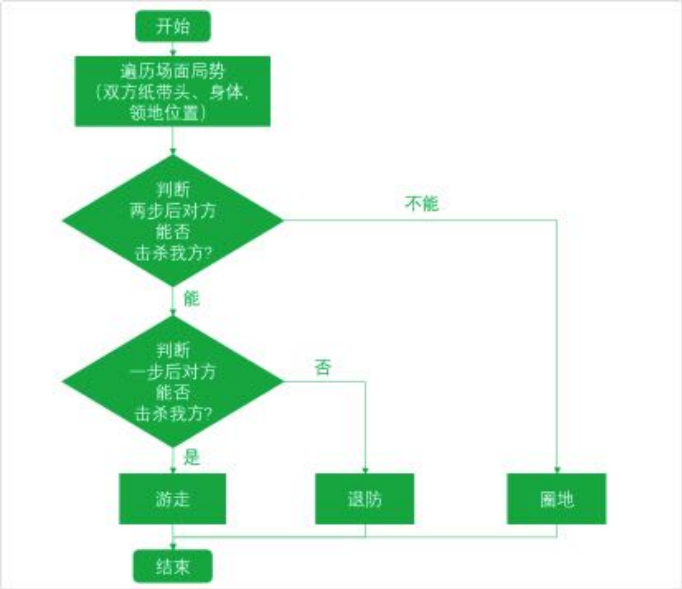
1.2 算法流程图



由于我们组的第一战略目标是进攻打击敌方，因此“击杀”这一概念至关重要。其判别流程如上图。



在理想版的算法流程中，模式的选择可以简化为三种，关键在于我们依靠稳定的进攻算法，能够保证在“白刃战”中大概率战胜敌方，因此只要对方在领地外我方便会选择进攻。



在现实版的算法流程中，可以发现我们的策略非常的保守，只有在判断为非常安全的情况下才会选择外出圈地，否则要么退防要么游走。

1.3 算法复杂度

下面会展示到，无论是理想版的 AI 和现实版的 AI，在判断对方能否击杀我方时都根据范围判别法采取了必要条件，把该部分函数记作 f1(n)；同时理想版 AI 在判断我方进攻以及现实版 AI 在判断游走时均采用了 BFS 算法，把该部分函数记作 f2(n)，则每次函数的执行时间可以记为 O(n1* f1(n)+n2* f2(n))，其中 n2、n2 分别为执行对应函数的次数。

由于 f1(n) = k（常数），f2(n) = O(V+E)，其中图的节点数（V）可认为等于双方边缘格点总数，边数（E）可认为等于节点数的 4 倍，因此影响算法时间复杂度的因素除了剪枝操作（减少 n2）以外还有当前敌我双方所占区域的边缘，在最差情况下可以认为 V = 比赛区域格点总数。

2 程序代码说明

2.1 数据结构说明

我们注意到影响决策的关键在于计算出双方：己方纸带头与己方领地、己方纸带头与对方纸带、对方纸带头与己方纸带之间的距离，同时为了实现预测接下来几步场上的局面，我们考虑编写一个 Paper 类，里面存放着相关的距离信息（调用 BFS 算法计算），还有一些内置方法（如下图）。

```
field, band, me, enemy = stat['now']['fields'], stat['now']['bands'],
                         stat['now']['me'], stat['now']['enemy']

class Paper(self, field, band, me, enemy):
    def __init__(self):
        self.MyHeadtoMyFields = 0
        self.MyHeadtoOpFields = 0
        self.MyHeadtoOpBands  = 0
        self.OpHeadtoOpFields = 0
        self.OpHeadtoMyFields = 0
        self.OpHeadtoMyBands  = 0
        self.myDirection      = ''
        self.opDirection      = ''
    def initialize():              #初始化Paper类
    def getDistance('mode'):       #根据'mode'的名称获取距离
    def updatePosition(position):  #更新纸带的位置
    def nextstep():                #考虑下一步的情况，第三步则再调用一次该方法
        for position in nextPosition:
            Paper.updatePosition(position)
            Paper.initialize()
```

其中，initialize 方法写成外部函数的形式如下：

```
# 初始化，输入当前局面下MyPaper的参数
def initialize(field, me, enemy): # initialize并不是每一步都要更新路径，只有在距离达到一定条件时才更新路径
    myattackresult = FindPathandCountDistance(field, me, 'bands')
    mygobackresult = FindPathandCountDistance(field, me, 'myfields')
    myinvaderesult = FindPathandCountDistance(field, me, 'opfields')
    opattackresult = FindPathandCountDistance(field, enemy, 'bands')
    opgobackresult = FindPathandCountDistance(field, enemy, 'myfields')
    opinvaderesult = FindPathandCountDistance(field, enemy, 'opfields')

    MyPaper['MyHeadtoOpBands'], MyPaper['MyAttackPath'] = myattackresult['dist'], myattackresult['path']
    MyPaper['MyHeadtoMyFields'], MyPaper['MyGobackPath'] = mygobackresult['dist'], mygobackresult['path']
    MyPaper['MyHeadtoOpFields'], MyPaper['MyInvadePath'] = myinvaderesult['dist'], myinvaderesult['path']

    MyPaper['OpHeadtoMyBands'], MyPaper['OpAttackPath'] = opattackresult['dist'], opattackresult['path']
    MyPaper['OpHeadtoOpFields'], MyPaper['OpGobackPath'] = opgobackresult['dist'], opgobackresult['path']
    MyPaper['OpHeadtoMyFields'], MyPaper['OpInvadePath'] = opinvaderesult['dist'], opinvaderesult['path']

    return MyPaper

initialize(field, me, enemy, MyPaper)
```

当中涉及到的 FindPathandCountDistance 方法是一个内嵌了 BFS 算法的路径搜索函数，FindPath 的目的是用列表记录搜索出来的最短路径，以便切换模式的时候调用。

```
def FindPathandCountDistance(field, plr, target):
    result = {'path': [], 'dist': 0}
    distList = [[[None] * stat['size'][1]] * stat['size'][0]]
    tempDist = 0
    flag = False  # 是否找到领地，是否找到纸带
    bandFound = []
    fieldFound = []
    nowSpot = [[plr['x'], plr['y']]]
    tempSpot = []

    if plr == 'me':
        op = 'enemy'
    else:
        op = 'me'
    if storage['band'][op] == [] and target == 'bands':  # 对方没纸带的情况
        flag = True
        result['dist'] = 9999999
```

```
    while not flag:
        for k in nowSpot:
            tempDist += 1
            if not flag:
                for dir in directions:  # 四个方向寻找
                    x1, y1 = k[0] + dir[0], k[1] + dir[1]
                    if x1 in range(len(field)) and y1 in range(len(field[0])) and distList[x1][y1] == None:  # 如果没算过
                        if x1 in range(stat['size'][0]) and y1 in range(stat['size'][1]) \
                                and band[x1][y1] != [plr]['id']:  # 没撞墙，没撞自己
                            if target == 'myfields':
                                if field[x1][y1] == [plr]['id']:  # 找到自己领地
                                    flag = True
                                    fieldFound.append([x1, y1])
                                    result['dist'] = tempDist
                            if target == 'bands':
                                if band[x1][y1] == 3 - [plr]['id']:  # 找到敌方纸带
                                    flag = True
                                    bandFound.append([x1, y1])
                                    result['dist'] = tempDist
                            if target == 'opfields':
                                if field[x1][y1] == 3 - [plr]['id']:  # 找到敌方领地
                                    flag = True
                                    bandFound.append([x1, y1])
                                    result['dist'] = tempDist
                            else:
                                distList[x1][y1] = tempDist
                            tempSpot.append([x1, y1])
                else:
                    break
        nowSpot = tempSpot  # 往外扩一圈
        tempSpot = []
```

```
    if result['dist'] < 9999999:  # 对进攻点，撤退点，入侵点分别找路径
        nowDist = result['dist']
        nowSpot = mySpot
        mySpot = bandFound[0]
        while nowDist > 1:
            for dir in directions:
                x1, y1 = nowSpot[0] + dir[0], nowSpot[1] + dir[1]
                if x1 in range(stat['size'][0]) and y1 in range(stat['size'][1]) and band[x1][y1] != \
                        stat['now'][plr]['id']:  # 没撞墙，没撞自己
                    if distList[x1][y1] == nowDist - 1:  # 只要距离减小了就走，找到一条路径再说
                        nowDist -= 1
                        result['path'].append(nowSpot)
                        nowSpot = [x1, y1]

    return result
```

2.2 函数说明

在理想版的 AI 里，我们能够实现高效的 attack 方法，能根据范围判别法判断是否执行 goback 方法，如果对方在自己领地里“苟”着，我们还可以调用 wander（在对方领地外巡逻）方法。

现将已实现的部分方法记录如下：

```
    def bandUpdate(plr): # 更新我方和对方的纸带坐标存储
        if not storage['band'][plr] == []:
            if band[storage['band'][plr][0][0]][storage['band'][plr][0][1]] !=
                storage['band'][plr][0]: # 如果纸带没了，则清空纸带存储
                storage['band'][plr] = []
        tempBand = [stat['now'][plr]['x'] + directions[(stat['now'][plr]['direction'] + 2) % 4][0],
                    stat['now'][plr]['y'] + directions[(stat['now'][plr]['direction'] + 2) % 4][1]] # 该玩家上一步的纸卷位置
        if field[tempBand[0]][tempBand[1]] != stat['now'][plr]['id']:
            storage['band'][plr].append(tempBand)

    bandUpdate('me')
    bandUpdate('enemy')

    def attack():
        return MyPaper['MyAttackPath'][storage['count']] # 每沿着路径走一步，count就加1

    def goback():
        return MyPaper['MyGobackPath'][storage['count']]
```

不过在现实版的 AI 中，为了保证生存的稳妥性，我们采用了更为保守的策略，对 wander 函数进行了优化，并把 BFS 函数内嵌到 wander 函数中去。

此时的 wander 函数能够保证对方在我方领域附近巡逻时，我方纸带不会随意离开领地，同时伺机反击。其具体实现如下：

```
def ismyBorder(x, y): # 判断(x, y)点是否为己方边界
    if field[x][y] == me['id']:
        for dir in directions:
            x1, y1 = x + dir[0], y + dir[1]
            if field[x1][y1] != me['id']:
                return True
    return False
```

```
def wanderPath(): # 在领地内找最短的路径到达边界，或者贴着边界走
    if not ismyBorder(me['x'], me['y']): # 如果不在边界上
        distList = [[[None] * len(field[0])] * len(field)] # 仍然是广搜的思路
        tempSpot = [[me['x'], me['y']]]
        nowSpot = []
        tempDist = 0
        goal = None
        while goal is None:
            for k in tempSpot:
                for dir in directions:
                    x1, y1 = k[0] + dir[0], k[1] + dir[1]
                    if distList[x1][y1] is not None and goal is None:
                        if field[x1][y1] == me['id']:
                            if ismyBorder(x1, y1): # 找到边界
                                goal = [x1, y1]
                                break
                        distList[x1][y1] = tempDist
            tempDist += 1
            nowSpot = tempSpot
            tempSpot = []
        path = [me['x'], me['y']] # 不用倒过来
        while path[len(path) - 1] != goal:
            for move in moves:
                if pt_dist(MoveToSpot(path[len(path) - 1][0], path[len(path) - 1][1], 'me', move)[0],
                           MoveToSpot(path[len(path) - 1][0], path[len(path) - 1][1], 'me', move)[1], goal[0], goal[1])
                           < pt_dist(path[len(path) - 1][0], path[len(path) - 1][1], goal[0], goal[1]):
                    path.append([MoveToSpot(path[len(path) - 1][0], path[len(path) - 1][1], 'me', move)[0],
                                 MoveToSpot(path[len(path) - 1][0], path[len(path) - 1])])
        path.pop(0)
        return path
```

```
    else: # 如果在边界上，则寻找安全的出去的机会，否则沿着边界走
        def danger(x, y): # 判断是否有危险
            if pt_dist(x, y, enemy['x'], enemy['y']) <= 3:
                return True
            return False
        def alongBorder(x, y): # 沿着边界走
            for move in moves:
                x1, y1 = MoveToSpot(x, y, me['direction'], move)[0],  MoveToSpot(x, y, me['direction'], move)[1]
                if ismyBorder(x1, y1):
                    return [x1, y1]
            path.append([x + directions[me['direction']][0], y + directions[me['direction']][1]]) # 如果找不到边界就直走
            return [x + directions[me['direction']][0], y + directions[me['direction']][1]]
        path = [me['x'], me['y']]
        flag = True # 没有离开边界
        while flag:
            for dir in directions:
                x1, y1 = path[len(path) - 1][0] + dir[0], path[len(path) - 1][1] + dir[1]
                if field[x1][y1] != me['id'] and not danger(x1, y1):
                    flag = False
                    break
            if flag: # 如果不能出去，就沿着边界走
                x1, y1 = alongBorder(path[len(path) - 1][0], path[len(path) - 1][1])[0],
                         alongBorder(path[len(path) - 1][0], path[len(path) - 1][1])[0]
            path.append(x1, y1)
        return path

def wander(): # 如果在自己领地内，并且不能击杀对方，则调用wanderPath计算路径，然后调用wander进行操作
    return MyPaper['MyWanderPath'][storage['count']]
```

2.3 程序限制

```
# 根据当前局面形式选择移动模式
def chooseMode(field, me, enemy):
    safedist = 2
    tempMode = storage['mode']

    # 根据MyPaper的参数选择移动模式
    if MyPaper['MyHeadtoMyFields'] == 0:    # 在自己领地内时
        if 0 < MyPaper['OpHeadtoMyFields'] - MyPaper['MyHeadtoOpBands'] < safedist :   # 这里只是能进攻的必要条件，
            storage['mode'] = 'attack'

        elif MyPaper['OpHeadtoMyFields'] - MyPaper['MyHeadtoOpBands'] > safedist :     # 能出去围地的必要条件，非常安全
            storage['mode'] = 'square'

        else:
            storage['mode'] = 'wander'
```



```
    else:   # 在自己领地外时
        if MyPaper['MyHeadtoMyFields'] + safedist > MyPaper['OpHeadtoMyBands'] :  # 考虑安全距离也跑不掉时就该赶紧跑了
            storage['mode'] = 'goback'
        # 以下情况不用担心我们会被干掉

        elif(
                MyPaper['OpHeadtoOpFields'] > MyPaper['MyHeadtoOpBands'] or    # 必胜情况
                MyPaper['OpHeadtoMyBands'] > MyPaper['OpHeadtoOpFields']       # 对方可能进攻我方时，威吓对方
        ):
            storage['mode'] = 'attack'

        elif MyPaper['MyHeadtoOpFields'] * 2 + 1 > MyPaper['OpHeadtoMyBands'] : # 选择来回是必要条件，最好能找到往返最短路径
            storage['mode'] = 'invade'

        else:   # 不去进攻纸带和入侵对方领地，那就围地吧
            storage['mode'] = 'square'
    if tempMode == storage['mode']:
        storage['count'] += 1
    else:
        storage['count'] = 0
        if storage['mode'] == 'wander':
            MyPaper['MyWanderPath'] = wanderPath()

    return storage[storage['mode']]
```

当前 AI 存在的问题是，在模式选择时我们对必要条件的使用过分频繁，导致“杯弓蛇影”的现象出现；而我们在小组赛中使用的代码则没有考虑到必要条件，往往忽略了返回路径需要绕开自己身体而导致的距离增大，进而导致在领地外被击杀。

3 实验结果

3.1 测试数据及结果分析

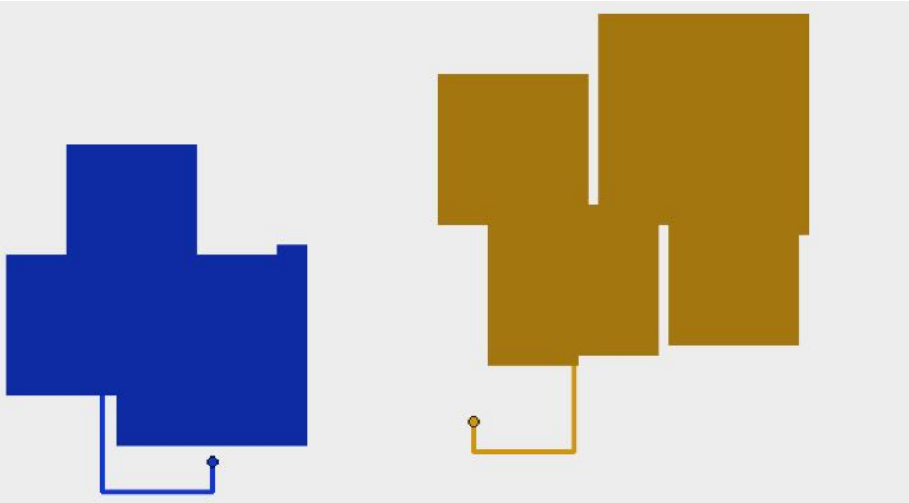
实验环境说明：

硬件配置：（CPU/内存）CPU i7/8G

操作系统：（名称/版本）Windows10

Python 版本：（版本号）python3.6.4

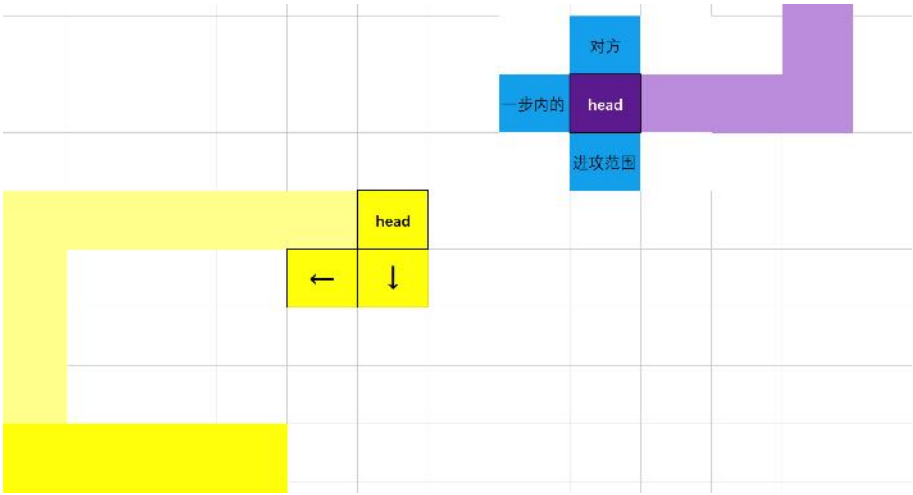
我们在 visualize 中对我们的 AI 和油聚给出的版本进行对比，发现双方都采用了比较稳健的策略。不过由于我方对 wander 函数进行了改进，使得运行中我们总能在领地内沿最短路径躲开对手的追击，因此能在同样的步数内围出更多地进而取胜。



3.2 经典战局

其实必胜法真的非常简单，一句话总结就是“先手优势”。

观察下图，我们发现后手方受到攻击的威胁时，转向至少需要经过两次移动，因此先手方如果选择一直向对方发起进攻，至少能获得三步（第一步是先手方移动的）的优势。然后只需要在这基础上完善圈地的函数，使得后手方“苟”在阵地里时我们能够选择占地取胜即可。



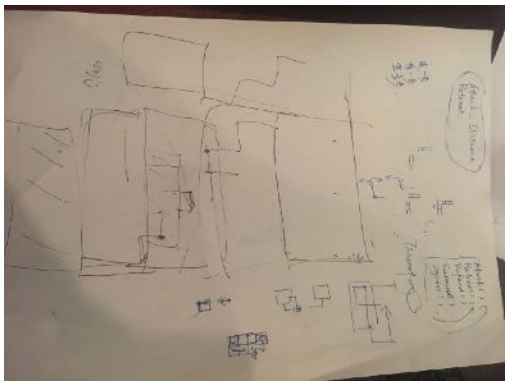
4 实习过程总结

4.1 分工与合作

组长	陈丹丘
代码	马涵聪冉瑾瑜郭天傲
实习报告	陈麒安

第一次组里讨论会：

大家陈述了一下对未来进攻、防守的些许设想，并逐步分析了每个函数的实现方式和功能。大家都对这些设想进行了讨论，我们的进攻和防守方式出现了雏形。但此时还没有深入进行实现，所以还不知道未来的路还很长。



上图是我们第一次讨论时，记录下的进攻与防守思维图。大家的设想还是很不错的。我们先分工，让同学分别编写“field”广度优先搜索函数和“attack”进攻函数。问题是我们忽略了接口实现、读取提供的 solo 函数问题，使得我们之后的代码之路十分艰难。

第二次组里讨论会：

开始编写代码，同学们就代码实现、接口问题进行了争论，同时也修改了当时我们已经初具规模的代码，当时已经陷入了代码实现等问题的纠结之中。实际上着也将是伴随我们代码编写中一直存在的问题；但当时已经对代码进行了优化，我们的攻击点转到了如何让读取函数和我们的代码更好衔接上的问题。

第三次组里讨论会和刷夜写代码：

刚好快到截至日期了，同时又撞上了组长的生日。大家吃了蛋糕然后开开心心的刷夜写代码。地球物理系图书馆成了我们刷夜的主阵地，此时代码实现的差不多了，广度优先搜索的代码也成功的移植到了我们的 load 函数里面。

同时在这里感谢 2016 级王泽鑫学长和 2015 级郭惠昀学姐给我们提供了精神上的大力支持！

4.2 经验与教训

1. 分工编写代码时要有手册

由于我们是第一次合作完成这种大型项目，没有意识到工作手册的重要性，导致我们小组基础功能组和拓展模块组两个独立工作的代码编写小组在工作交接时遇到了接口不对应的窘况，有几个广搜算法没能够在最后的 AI 中实现，直接导致了我们可爱的小蛇战斗力的下降。其中有一个接口上的差异让我们记忆犹新：

```
#   组员1编写的代码
directions = ((1, 0), (0, 1), (-1, 0), (0, -1))      #东南西北的元组
#   组员2编写的代码
directions = [[1, 0], [0, 1], [-1, 0], [0, -1]]      #东南西北的列表
```

而这样因为没有事先统一接口、名称、代码习惯而造成的问题极大地降低了小组合作的效率，因此希望以后的小组引以为鉴。经过讨论，我们得到了一份比较完善的编写手册，以供我们暑假进一步完善代码使用

```
#   函数/方法的命名法：
#   1. 特定函数
    def initialize():
        pass
    def attack():
        pass
    def wander():
        pass
    def goback():
        pass
#   2. 两个单词组合：
    def isBorder():
        pass
    def chooseMode():
        pass
#   3. 三个或以上单词的组合：
    def MoveToSpot():
        pass
    def FindPathandCountDistance():
        pass

#   变量的储存：
#   1. 多数据类：列表
    directions = [[1, 0], [0, 1], [-1, 0], [0, -1]]
#   2. 对应类：字典
    MyPaper = {'MyHeadtoMyFields': 0, 'MyHeadtoOpFields': 0, 'MyHeadtoOpBands': 0,
               'OpHeadtoOpFields': 0, 'OpHeadtoMyFields': 0, 'OpHeadtoMyBands': 0,
               'myDirection': '', 'opDirection': ''}
```

编写手册节选

2. AI 的编写可以从具象模拟到抽象形态

我们收获的最大的惊喜应该就是“必胜法”的发现了，而这居然是源于第二次聚会时晚饭前两位组员在草稿纸上的演算。这提醒了我们，很多时候我们在把任务交给计算机的时候，并没有意识到，许多人脑习以为常的事物、规则，其实在计算机实现算法的过程中是难以得到体现的，因此我们在设计高级的算法时，可以先对人自然性的操作进行模拟，然后再上升到抽象刻画的层面。

这让我们想起上学期计算概论课上遇到的两道题目，一道是“绘制回型数矩阵”，一道是“煎鸡蛋问题”，这两道题目的特点都是：人只要按照常识去做，很容易就能得到正确答案，因此只需要描述人的操作就能 pass，但如果想方设法获得答案的公式解，反而会事倍功半（甚至有可能在期末考挂掉）。

3. 对算法设计与程序编写的反思

我们发现其实很多课堂上讲过的算法或者是网络上的常见算法都可以运用到大作业中去，毕竟那些算法实在是过于经典了，然而我们最终实现的 AI 就只是简单的顺序、选择、循环结构的组合，并没有体现到算法的威力。

这从学习的角度上提示我们，数据结构与算法一课的学习必须基于平时扎实的积累

——把平时接触到的算法进行搜集和归类，记录好它能用于解决什么问题，性能上的优缺点有哪些，并且要自己尝试用代码将之实现一次，甚至可以开一个自己的博客 or GitHub，把自己的学习心得放在上面以便日后翻看和与他人交流。

4.3 建议与设想

对完成大作业过程的的反思

在这里首先要谢谢老师、助教们还有技术组的聚聚们为我们准备了那么精彩的平台，虽然规则在过程中也陆陆续续有些新的解释条款，不过在那么仓促的时间内能够完成那么完善的平台出来，真的是太强啦！

当然我们觉得如果能在期中前后就通知大作业的内容会更有利于提高我们的完成度，而相关的规则也可以在这段时间内在微信群上进一步讨论和完善，同时也让技术组们有更充足的平台搭建时间。

对我们小组未来的设想

在观看了现场比赛后，我们小组有以下改进的想法：

1. 优化搜索过程的剪枝步骤，实现“走一步，算一步”的战略目标。N17 组的巅峰对决让我们意识到，“菜的算法各有各的菜法，优秀的算法总是一样的优雅”，这说明大家的思路确实是相同的，只不过是我们因为畏难的心理而不敢放手去完善这个功能。下一步我们打算利用假期时间通过这个项目巩固我们对搜索算法的了解，并且学习剪枝的相关知识，优化搜索步骤。

2. 完善“必胜法”。我们通过复盘 17 组季军的比赛，发现他们优生的局面和我们推出的“必胜法”的前提条件是一致的，这说明我们是再次因为不敢放手去做而错失了一次良机。有组员提出，这种局面和五子棋的“先手必胜”有异曲同工之妙，我们还可以借此机会研究五子棋的必胜法，从中收获“愉悦的快乐”。

3. 利用这个项目尝试入门深度学习算法。在经历了备赛期间始终顶着“我们这大组居然有深度学习的聚聚，这简直 bug 啊”的心理压力，我们决定秉着“打不过就加入”的 NBA 优良传统，在暑期入坑深度学习。我们相信这个天坑一定充满着我们看不懂的数学公式和算法推演，但是我们相信，除了这次机会之外，我们可能再也没有接近这个领域的机会了（郭聚除外），于是我们打算在假期继续组建学习小组，共同研究“纸带游戏”的深度

学习实现。

对学弟学妹的建议

事实上，课程大作业是对我们算法思维掌握程度的综合检验，换而言之要做好大作业必须是从最开始就有所积累。我们建议明年选修数算的学弟学妹们：

1. 在学习计算概论等入门课之后，就结合网络资源（如廖雪峰老师的 python 教程、学长学姐们辛苦翻译的教材）学习 python 的编程，了解其语法特点和面向对象编程的思想。

2. 可以参考 gis4g 网站上往年的课件，先对一个学期下来整门课的学习框架有所了解，在老师讲授相关内容之前先利用课外资源进行学习，比如《算法》、《代码本色——用编程模拟自然系统》等有趣的基础书，或者是 MOOC 上关于 python 爬虫、数据挖掘、可视化处理等内容的课程，这样的探索能够极大提升我们对本门课程学习的兴趣，也能为大作业的完成打下扎实的基本功。

3. 在课程的过程中，老师会通过作业引导我们“像个程序员”一样学习，比如说作业 H1 是让我们学会了解课外的算法、H3 让我们学会用作图的方法分析算法的时间复杂度、H2 让我们在网络上（如 CSDN 博客）上搜索有关问题其他人的见解，以提高我们的知识水平等等，我们在掌握了这些技能之后要学以致用，比如说在用堆来实现优先队列时，我们就可以自己分析每个接口的时间复杂度（期末考就考到了），在网上浏览相关的问题和算法，这样我们才能不断拓宽自己的知识边界，提高算法的思维。

4. 一般来说，数算课的大作业是大家完成的第一个比较大的程序，这种大型程序的设计和维护都需要耗费比较多的精力，所以学弟学妹们可以在期中考后左右的时间里找来往年的大作业，先和队友们尝试做一做练练手，有了合作的经验之后，面对自己学期的大作业自然会更加的游刃有余。

对课程的反思：建立算法思维认知过程

我们觉得，数算大作业的意义不仅仅在于让我们熟悉课堂上学过的 ADT、算法，更重要的是，让我们在实现大作业的过程中体悟用算法思维来认识事物的过程。在实现本次纸带游戏的过程中，我们对程序设计有了更深入的理解：

我们设计程序的目的是利用计算机解决某一特定的可计算问题，如果把所有被设计出来的程序都看作一个 AI，那么我们所学的算法将会是它的灵魂：从初始条件到预期结果，中间的过程是一个黑箱子，算法实际上就是我们人类利用自身的智慧把黑箱子具象化的一

个过程。比如说我们从预期条件会获得用数据来描述的世界，那我们首先要从中解读出有用的信息，因此需要有排序算法；我们需要从处理过的数据中得到我们想要的结果，因此会有搜索算法；除了课堂上学习过的基础的排序和搜索，我们在处理特殊问题中还发明了许许多多处理特定问题的算法，比如说求第 k 大数算法、遗传算法等等，这些算法是前人智慧的结晶，我们要加以学习，加以运用，而其背后的设计思维理念是我们需要学习的，比如如何根据问题的特点码下“神之一手”，把算法的时间复杂度降低，我们认为能在自己研究的领域获得这样的灵光一闪并加以实现，将会是我们为科学领域做出的极大的贡献。

另一方面，我们所学的、所拓展的数据结构将是它的骨骼、肌肉，正如不同项目的运动员会有不同的身体条件，我们面对不同的问题时也需要使用不同的数据结构，比如说我们对树的搜索中为了实现递归可以引入栈 ADT，我们在进行搜索时可以根据数据间的关系引入图 ADT，而从这些数据结构的独特性出发，我们可以设计出许多特殊的算法来解决问题，比如说字符串匹配，骑士周游问题，但我们认为 ADT 的精髓在于我们可以用一些新的黑箱子来解决目标问题的黑箱子，在某些情况下，我们甚至可以把绝大部分功能包装进 ADT 中，而不需要考虑具体问题繁琐复杂的边界条件、特殊情况等（只需要在外部函数模块解决即可），这极大地解放了我们的思维，降低了我们的思考成本，再次转换到运动上来，一个设计完备的 ADT 好比人体强大的“核心力量”，有好的核心力量，运动员自然可以胜任各类运动竞技的挑战。

经过反思，我们结合数算课的课程教授进程，得到了一些学习数据结构与算法的启发。

学习数据结构与算法，目的在于建立起一套完备的认知过程：认识问题、选取处理对象、获取数据来描述对象、数据预处理以描述问题过程、思考算法以解决问题、设计 ADT 和函数以实现算法、建立对初始问题系统化的认知、认知应用（解决新问题、提高原有问题的效率等）。

第一堂数算课所学的“计算”与“抽象”的概念，其实就是引导我们从程序的角度去认识问题，如何把现实问题抽象成一个个可计算的问题并交由计算机完成，这应该是我们处理问题的第一步。然后根据问题的特征选取我们应该处理的对象，这直接影响了我们往后算法的选取与实现，比如说本次大作业，如何处理纸带的头、身、领地三部分之间的关系将影响到退防距离的判断。获取数据和数据预处理步骤并不在数算课的教学范围之内，但实际上，各式各样的排序算法、统计学公式存在的意义就是帮助我们从纷繁复杂的数据中寻求背后隐藏的规律，比如说 GNSS 系统在实现定位导航功能的第一步就是对数据的获取和预处理，因而此二步骤也是思考算法前的必要铺垫。

完成准备工作后，接下来便是算法的设计与实现环节了。思考算法对初学者来说，要么是设计简单的算法来完成对人类现实操作的模拟，比如上文提到的煎鸡蛋问题，要么是对经典算法的复刻和化用，这便要求我们要把数算课上学到的算法全部掌握并能够有所迁移，同时可以浏览相关书籍、网络资源以了解更多经典算法，并积累其运用条件，把前辈们智慧的结晶为我所用，保持学习将是引来“源头活水”的好办法。而相比于纷繁复杂的算法而言，经典的 ADT 则显得精悍了，栈、队列、链表、树、图..... 这些经典的 ADT 在复杂的算法中无处不在，我们在接触到这些新的 ADT 时一定要将其学透，课件和作业都只是对这些 ADT 的简单展望，更重要的是我们要借助课外资源多了解这些 ADT 的使用条件，相关的经典算法，深刻体会选取这些 ADT 背后的意义，这样做能够让日后实现复杂的算法事半功倍。

最后两步对应的是科学探究中的进一步探究，课堂上一个典例就是字符串。我们在课堂上会接触到和字符串相关的一些问题，但经过查阅资料后我们会发现，字符串其实是一块非常大的研究领域，我们不仅可以关注到自己已学的算法能解决字符串的哪一类问题，还可以在研究其他相关问题是掌握其他工具（比如向量）在这一领域的应用。只有不断打破认知边界，对已经掌握的领域询根问底，才能构建出更完备更具批判性的算法思维。

以上便是我们小组对本门课程、本次大作业的思考，既是作为我们结课的留念，也是供他人参考的一家之言：云山苍苍，天水泱泱。算法之美，山高水长。算法之流，汇涌万江。

5 致谢

本次大作业是在陈斌老师的指导下，技术组聚聚们的努力下，各位助教的讲解下，本组组员共同参与，合作完成的。因此首先要感谢陈斌老师为我们创造了大作业的机会，使得许多志同道合的伙伴们能够聚在一起讨论思考，迸发闪烁的思想火花。其次要感谢技术组的聚聚们以及各位助教在整个大作业期间的耐心指导，从选题到代码到报告，均给予详尽的说明与建议。

最后，衷心感谢本组所有成员的倾心付出：本次大作业中，在组长陈丹丘的带领下，各位组员均身担重责，除了五位成员团聚的两次线下组会外，三位女生采用刷夜工作法写下了 AI 的核心代码，两名男生为其他模块的搭建提供了思路，每个成员对于最终成果的贡献均功不可没。

第十八章　F17_Quebec 报告

郭祉辕、吕悦琪、李子锦、汪昀鸿 *、叶继开

摘要：本程序主要使用的算法为宽度优先搜索算法，通过它计算图上点与区域之间的距离，涉及的数据结构主要有图、队列等。在通过搜索算法获得我和对方回到领地、攻击敌方纸带的四个距离以后，AI 即可判断我方应该攻击对方还是回到领地或是继续圈地。在使用圈地战术时，要将近期利益和远期利益结合起来。最后，还要考虑棋盘的结构对局势的影响，中心和边界的区域分别有不同的作用。分析实验结果，我方 AI 展现出极强的圈地能力，击杀能力也较强，但防守能力较弱。

关键字：队列、图、宽度优先搜索

1 算法思想

1.1 总体思路

Paperio 是一个攻防兼备型的游戏，所以在 AI 的设计上我们主要从这两个方面出发。在进攻端，有两种进攻的方式：吃对方的领地、攻击对方的纸带。在防守端主要需要考虑如何防止被对手撞击。

在进攻端，首先考虑圈地策略。当我方占领区域很小的时候，可认为是从一个点出发圈地回到这个点，在边长一定的情况下，正方形的面积最大；如果我方领地较大时，从边界出发，当周长一定时，圈一个长方形的利益最大。因此，在周长一定时，尽量走直线形成方块，而非走“蛇形”，更有利于圈地。同时，在圈地过程中，AI 需要判断一片区域的“潜在利益”，挑选近期利益较大的方向走；并且，在眼前利益太小的时候要挑选远期利益

较大的方向走。

其次，考虑棋盘结构。由于如果前期占领了中心区域后，对方只能在边缘地带发展，而边缘地带是由“扁平长方形”构成的区域，圈地效率不会很高。因此，我们认为中心地带的价值会较大。经过热身赛，我们发现，如果前期能占领四个边界的较大区域，那么在后期可能能够迅速覆盖整个棋盘。因此，我们认为边界地带的价值也很大。基于这两点对棋盘结构的分析，我们将棋盘中心赋较大的权，边界通过“补偿”手段在某些时候增大其利益评估。同时，由于吃对方领地时对方领地减少，我方领地增加，因此对对方领地的估值应当大于对空地的估值。

再次，考虑撞击对手纸带。由于这种情况有风险，我们仅在“时机非常成熟”的时候执行这种操作。为评估风险、预计击杀可能，我们需要测量我们纸卷到对方纸带的距离、对方回领地的距离、对方击杀我的距离等一系列参数。这些参数的测定，大都是通过宽度优先搜索实现。在某些情况下，距离差距明显时，就不需要通过搜索实现。因此，我们通过预处理排除了一些不需要精确测距的情况。在参数测量完毕后，只有在“击杀长度小于他回领地长度，他杀我的距离大于我杀他的距离”这种情况下，我们才启动击杀策略。

在防守端，主要考虑被击杀的可能。通过测量对方到我方纸带的距离，我方回领地的距离，判断是否他能够击杀我方。距离的测量通过 BFS 进行，通过预处理也可以排除一些不需要精确测距的情况，减少搜索时间。

本程序主要使用图和队列这两种数据结构。其中，整个棋盘通过图的数据结构进行存储；而在 BFS 中，将使用队列这种数据结构存储接下来要遍历的节点。

算法策略文字描述如下：

首先执行测距模块

1. 对距离测量进行预处理，并测量“他到我纸带的距离”、“他回领地的距离”、“我到他纸带的距离”、“我回领地的距离”

其次执行“击杀、被击杀模块”：

2. 如果发现“我到他纸带的距离小于他到我纸带的距离”且“他回领地的距离大于我到他纸带的距离”且“回合数大于我击杀他的步数”，那么他回不去领地且无法击杀我且我可以击杀他，于是我们直接冲向他的纸带

3. 如果发现“我回领地的距离”小于“他到我纸带的距离”减去 4（一步失误最坏造

成 4 步距离减少）、大于等于“他到我纸带的距离”加 1 时，立刻回领地

4. 如果发现“我回领地的距离”小于“他到我纸带的距离”加 1 时，无法回领地，因此冲向他的纸带

最后执行“圈地模块”：

5. 判断我方是否走了较远的距离却还在我方边界附近徘徊，如果是，则直接回领地

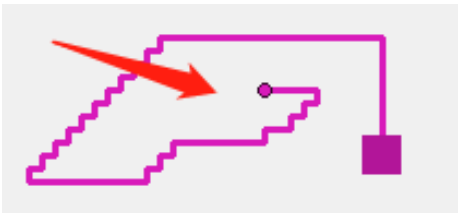
6. 如果我方在自己领地内部时，判断能够走的方向中哪一个方向的近期利益最大，如果他们的利益都很小，则返回远期利益较大的方向

7. 如果我方在领地外时，对每个方向的期待利益、走了这一步是否会死进行判断，如果走原来方向的利益不会死且超过所有方向期待利益的 80% 时，则选择走原来方向（通过这样来减少走曲线），否则挑选期待利益最大的方向走

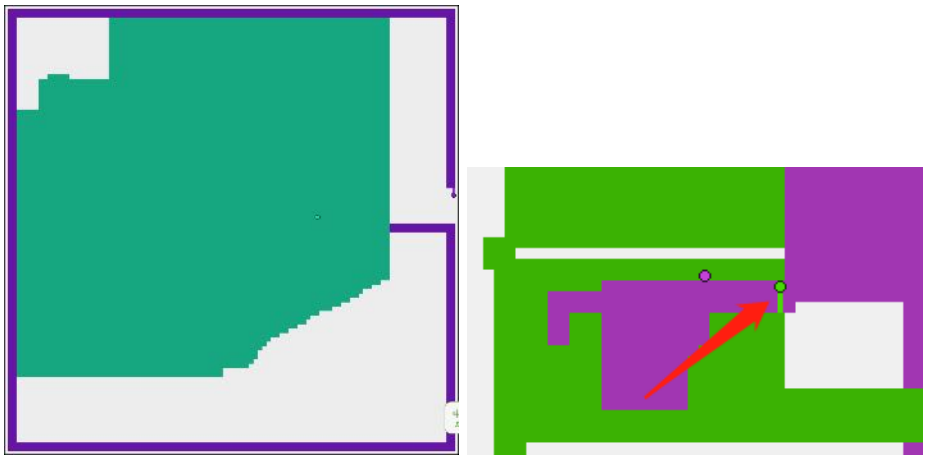
在研究之初，我们希望基于传统棋类剪枝算法“alpha-beta 剪枝算法”来进行设计。然而由于棋盘规模太大，该算法计算的步数十分有限，对整个棋局发展无法有效预测和估计，因此放弃了此算法，转而采取现在的基于策略的 AI 设计。

最初版的 AI 考虑到 BFS 只能搜索一条最短路径、且可能此路线可能并非最佳路线，因此在测量距离的函数中搜索了三条最短路径。同时，最初版没有对距离测量做预处理，短期利益函数中使用的广度优先搜索对一个方向搜索十层做利益估计。因此，最初级的 AI 计算非常缓慢，十分容易超时。

为了提高优化计算速度，第二版 AI 减少了最短路径搜索条数，添加了距离预处理函数，并将短期利益函数改成了对某一确定范围内的少数确定点做利益估计。通过这些方法，第二版 AI 计算速度有显著提升。然而，第二版 AI 会走入自己的“势力范围”之内（如下图），并且过分追求短期利益，没有全盘估计能力。因此，增加了一个判定是否走入自己势力范围内的函数，和一个估计全盘利益的函数。改进后的 AI 能力又有所增强，但我们发现它有时会在离自己领地很近的地方周游，不回自己的领地，这会导致其圈地效率较低。因此后来又增加了判定是否它在自己领地周围游弋不回领地的函数。

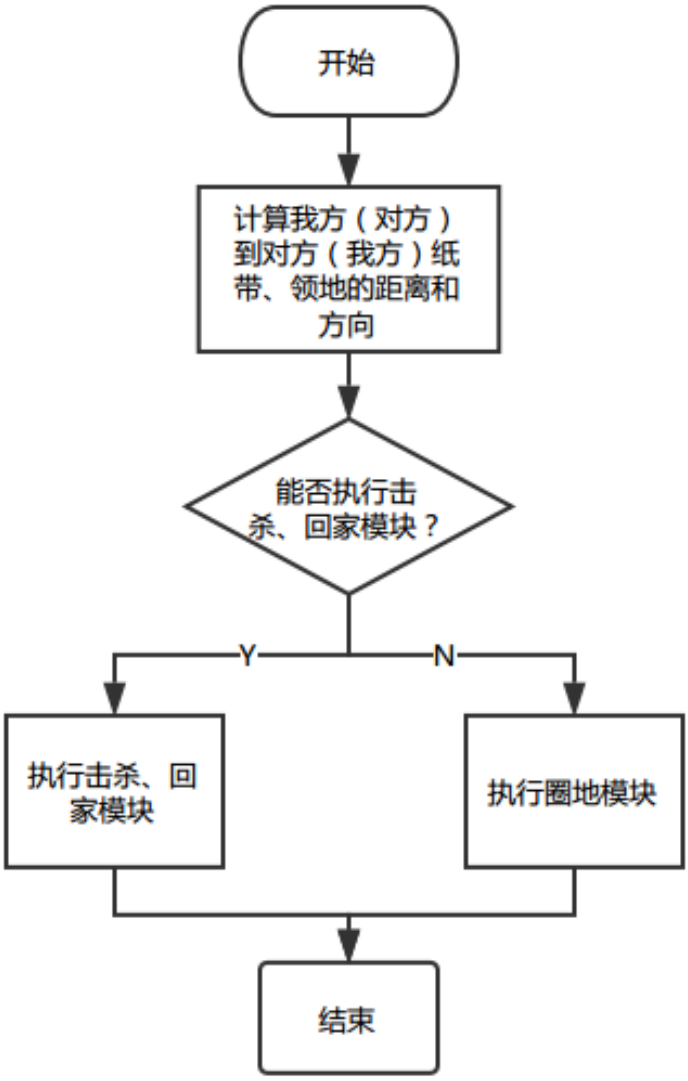


在最初的热身赛中，这个 AI 的表现比较好。然而，接下来的一次热身赛出现了一个“围边界”的 AI（如左图）。这让我们意识到棋盘的结构应该对估值产生影响。棋盘中间部分可以压制对手发展，棋盘的边缘能让后期圈地面积迅速增大。因此，我们增加了对棋盘不同位置赋权的做法。

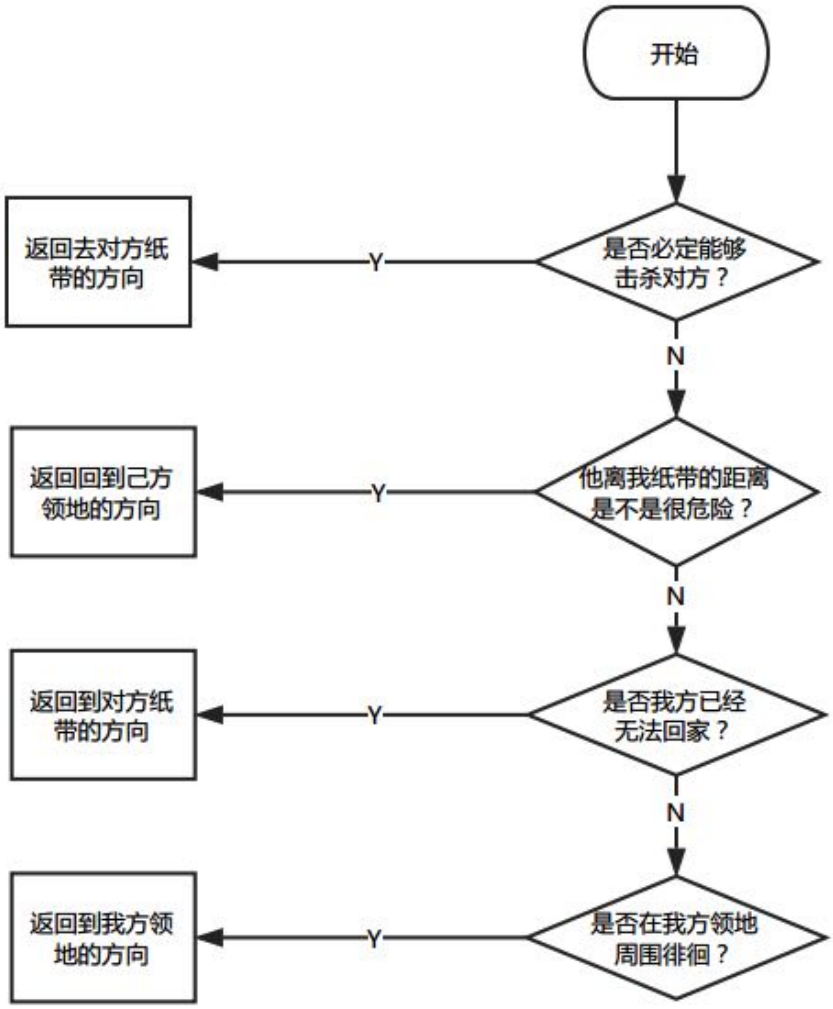


在后来的自对弈过程中，这个 AI 展现出惊人的圈地能力。在刚开始的一段时间内，该 AI 在 500 步大约可以占领面积 2000 的领地。然而，它并不能有效的保证自己的安全，容易被击杀。尤其是与人类玩家对弈时，其对敌人领地的贪婪导致它屡屡被击败（如左图）。经过研究，我们发现这个 AI 使用的测量距离函数并不能保证其绝对存活。首先，它只能测量在当前纸带情况下的击杀距离，不能考虑到对方可能攻击新形成的纸带，因此可能在回领地的路上被对方击杀。其次，由于局势动态变化，我方领地可能被对方吞噬，这个测距函数所测出来的回领地的领地上的那一个点有可能在接下来的几步被对方吃掉（这一点尤其表现在进攻对方领地的时候），导致回领地距离增大，因此这也可能导致我方 AI 被击杀（如左图）。对于第一个问题，可以通过在计算回领地的最短路径时保存最短路径上的点，并测量对方到新形成的纸带和原有纸带的最短距离来确定是否需要回领地。对于第二个问题，需要判断我方返回领地的目标点是否会被吞噬。这必须测算对方的是否将形成一个包含此点的闭合区域，即必须判断此点是否被对方的边界包围。这不得不遍历该点周围的在对方领地外的所有点，如果每次都测算会导致极大的计算量增加。在交战中，我们发现我方 AI 被击杀的情况大都是因为回领地的目标点被对方占领导致的，第一种情况出现的较少。然而由于可能增大较大计算量导致超时，我们并没有采取防止这种情况出现的方法。

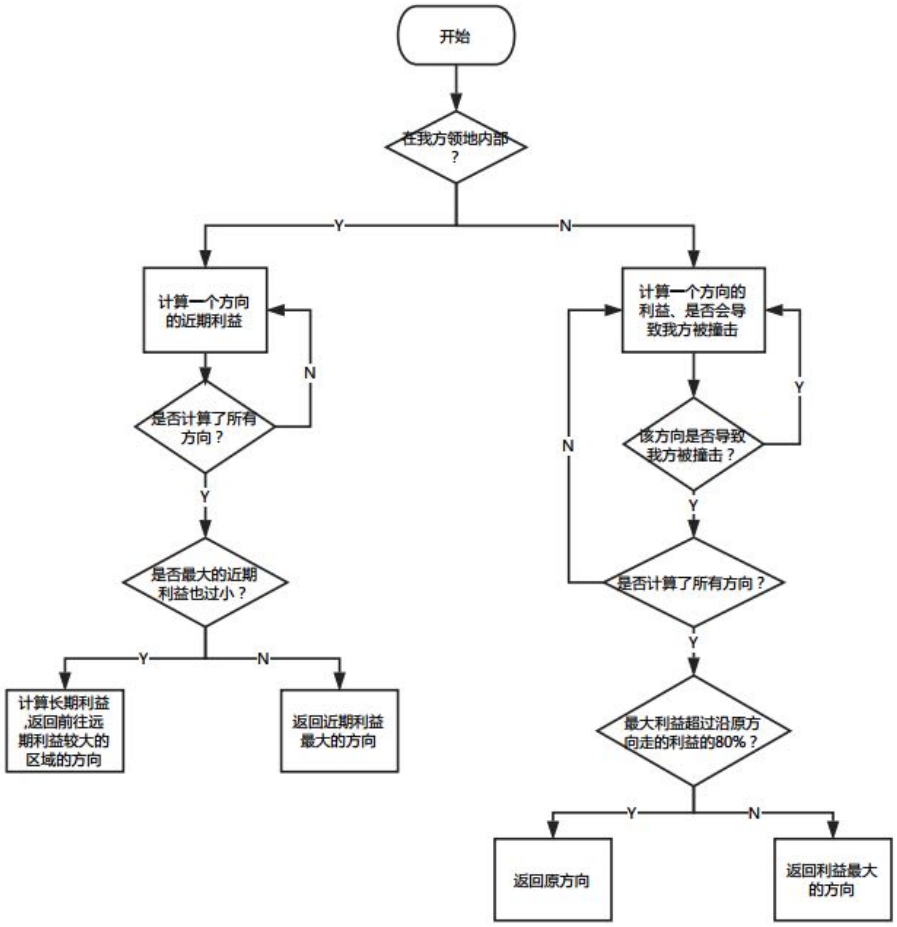
1.2 算法流程图



总体流程


开始
是否必定能够击杀对方？
Y
返回去对方纸带的方向
N
他离我纸带的距离是不是很危险？
Y
返回回到己方领地的方向
N
是否我方已经无法回家？
Y
返回到对方纸带的方向
N
是否在我方领地周围徘徊？
Y
返回到我方领地的方向

击杀、回领地模块



圈地模块

1.3 算法复杂度

整个主函数 strategy 由 7 个函数组成：to_my_field，to_his_band，renew_band_boundary，mt_benefit，lt_benefit，hollow，convert。其中，renew_band_boundary、mt_benefit、lt_benefit、hollow、convert 函数都是 O(1) 复杂度的。to_my_field 和 to_his_band 函数使用了 BFS 算法，时间复杂度为 O(n+e)，其中 n 和 e 分别为达到目标节点前遍历的所有点和边的数目。而这里的点和边的数目，随着初始节点与目标节点的距离 d 增长呈平方倍 (d2) 增长。

主函数 strategy 中，对 to_his_band 做了优化，如果精确计算出的回到领地的距离小于粗略估计的对方碰撞纸带的距离，那么就不必调用 to_his_band 函数，因此测距部分的

时间复杂度为 BFS 复杂度 O(n+e)。strategy 里的击杀、防守模块中，只有在我方已经无法回到领地、必须击杀对方的情况下可能会再次调用 To_his_band 计算击杀方向，其他情况下都是 O(1) 复杂度的操作。因此击杀、防守模块的复杂度约为 O(1)。圈地模块中，如果己方纸卷处于我方领地内部时，计算三个方向的近期利益复杂度为 O(1)，计算远期利益的复杂度也是 O(1)，所以如果在己方内部的时间复杂度为 O(1)。而若在我方领地外部，需要计算三个方向的期待利益以及如果走了该方向是否会被击杀，其中会调用 To_my_field 函数。因此若在我方领地外部，则时间复杂度为 O(n+e)。

由算法最初会进行 renew_band_boundary 操作，时间复杂度为 O(1)。因此，综合三个主要模块的时间复杂度，我们可以知道在最好的情况下（对方离我方纸带很远且我方能执行击杀—防守模块、我方纸卷在我方领地内部），只需要测算我方纸卷回领地的距离，其它计算都是 O(1) 复杂度的，因此算法的时间复杂度为 O(n+e)，其中 n 和 e 随着我方纸卷到我方领地的距离增长而呈平方倍增长。最坏情形为对方离我方纸带比较近，不能执行击杀模块。这种情况下，需要精确计算我方与对方纸带的距离、计算走三个方向后我方回领地的距离等，时间复杂度也是 O(n+e)，其中 n 和 e 与我到我方领地、我到对方纸带、对方到我方纸带、对方到自己领地这四个距离的最大值的平方呈正相关。

因此，整个算法的时间复杂度在最好情况下与我方回领地的距离的平方呈正相关，在最坏情况下与我方回领地距离、我方击杀对方距离、对方回领地距离、对方击杀我方距离这四个距离的最大值的平方呈正相关。

在实际运行中，使用 glory_of_mankind 进行人机大战，发现 AI 纸卷离 AI 领地较远时，游戏速度明显变慢，而离领地很近的时候游戏速度很快。在实际比赛中，当我方纸卷离领地很近时，时间开销很小，而很远时，时间开销很大。和分析基本一致。

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为图和队列。fields 和 bands 都通过二维数组表示，BFS 中使用一个列表作为队列。

2.2 函数说明

1.To_his_band 函数

计算某一方到对方的纸带的最近距离，返回到他纸带的最近距离和第一步应该往哪里走。接收参数：player_id , player_pos , he_pos , fields , bands，分别为我方 ID、位置，对方位置，领地情况，纸带情况。返回值：一个含有两个元素的元组，第一个值是我方到对方纸带的最近距离，第二个值是我方应该走的方向

采用了宽度优先搜索的算法，生成一个队列保存需要遍历的节点，生成一个 None 二维数组 pre_matrix 保存每个节点的“前驱节点”。每次从对列中出队一个节点，遍历其邻居节点，将需要遍历的邻居入队，并在 pre_matrix 中记录该邻居的前驱结点。直到找到他的纸带，记录找到的对方纸带的位置。从他的纸带的位置开始，通过 pre_matrix 回溯到出发点的位置，找到第一步应该走的方向和步长。

2.To_my_field 函数

计算某一方到自己领地的最近距离，返回回到我方领地的最近距离和第一步应该往哪里走。

接收参数：player_id , player_pos , me_dir , fields , bands，分别为我方 ID、位置、我方原来所走的方向，领地情况，纸带情况。

返回值：一个含有两个元素的元组，第一个值是我方到我方领地的最近距离，第二个值是我方应该走的方向。

算法和 To_his_band 相同。

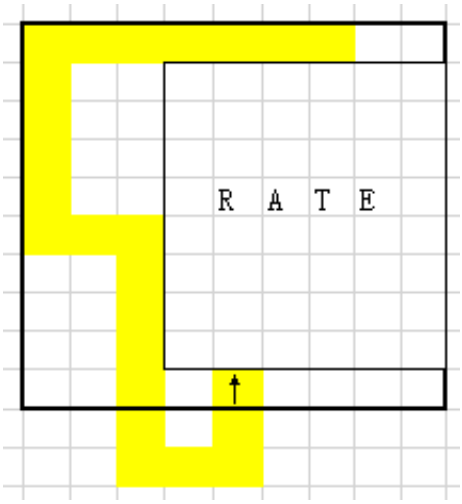
mt_benefit 函数

计算我方某位置某方向的近期利益，返回该利益的值。

接收参数：me_dir , me_pos，分别为我选择走的方向、我的位置。返回值：该方向的近期利益。

首先，在我选择走的方向的正前方形成一个 12*12 的正方形区域，在这个区域内计算期待利益。为了减少计算的点数目，每隔距离 3 取一个点，形成 16 个需要计算的点。生成 score1 和 score2 两个量，分别保存真正的盘内利益和补偿的边界利益（提高 AI 对边界的偏好）。为棋盘赋权，由于中部的利益差不多，越往边界权重下降越快，因此采取非线

性函数 log 函数作用于该点。其表达式如下图所示。遍历这 16 个点，如果该点越界，则 score2 增加 1*weight，如果没有越界，空地则 score1 增加 1*weight，对方领地则 score1 增加 2*weight。同时，考虑到能够走的点并非这整个区域（由于我方纸带的存在，有的面积无法进入），因此探测当前位置前、左、右三个方向我方的纸带。如果探测到了我方纸带或探测超出了该 12*12 正方形的区域，则停止对该方向的探测。通过探测到的纸带边界形成的矩形区域，可以粗略测定该 12*12 区域内纸卷能达到的面积，通过这个面积除以 12*12 作为比率 rate。如果 score1 小于 2 时，认为没有实际利益，则返回 0。否则返回 (score1+score2)*rate，表示在我方能达到的区域内实际利益和边界补偿利益的和。



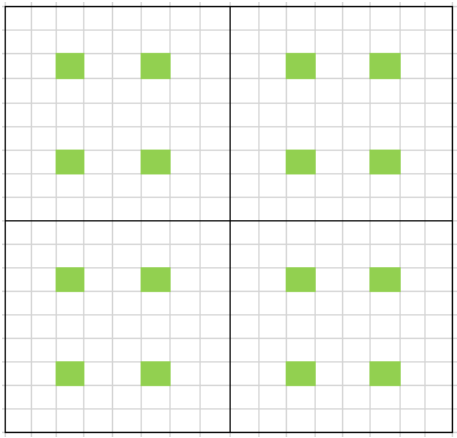
4.lt_benefit 函数

将棋盘分成四份，计算每一个区域内的期待利益，返回通往期待利益最大的区域的方向。

接收参数：me_pos , me_dir，分别是我方纸卷的位置，我方当前的方向。

返回值：前往利益最大区域的方向。

首先将棋盘等分为四份：左上、左下、右上、右下。在每一个域内取四个点，如果该点是空地则估值为 1，对方的地则估值为 2。将每个区域内的四个点的估值相加，作为这个区域的估值。为了更快的跳出局部极值，我们不选择纸卷现在所在的区域作为将要前往的区域。在剩下的区域内，选择一个估值最高的区域作为将要前往的区域，并将该区域的中点作为准备前往的点。计算要前往该处应当走的方向，返回该方向。



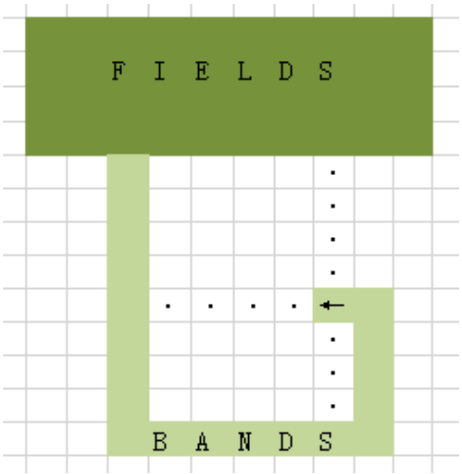
5.hollow 函数

判定是否所走的方向走近了“我方控制范围”，返回判定结果。

接收参数：me_pos , me_dir，分别表示玩领地当前位置、玩领地采取的方向。

返回值：0 或 1，0 代表没有走进我方控制范围，1 代表走进了我方控制范围。

从我方位置，向前方、左方、右方三个方向搜索，直到找到我方的纸带（得分加 2）、领地（得分加 1）或越界。如果最后得分大于 3，说明有两个方向有纸带、或者三个方向均有我的棋子并至少有一个方向上有我的纸带，在这种情况下判定为走近了我方领地，否则判定没有走进我方领地。

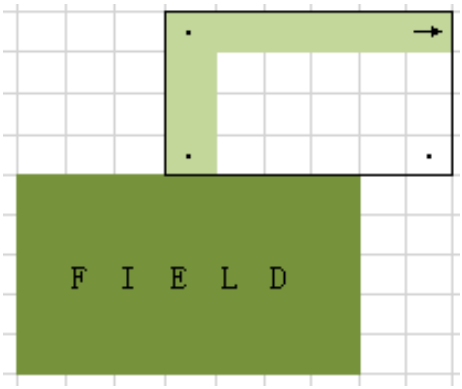


6.renew_band_boundary 函数

更新 storage 保存的纸带边界，无返回值。

接收参数：me_pos , me_id，分别表示玩家的位置、玩家 id。

storage 内储存了双方玩家的纸带边界四个顶点的位置。当玩领地回到自己领地时，更新该玩领地的纸带边界四个顶点为当前位置；当玩家在领地外运动时，如果当前位置超出了原来纸带的边界则更新边界。



7.strategy 函数

AI 的主要函数，包含所有策略，返回行走方向。

接收参数：fields , bands , me_pos , he_pos , me_dir , he_dir，分别表示场地情况、纸带情况、我方位置、对手位置、我方行走方向、对方行走方向。

返回值：0 到 3，表示所决定走的方向。

首先，通过 To_my_field 函数计算我方回领地的距离，然后计算对方到我方“纸带边界”的距离。这个距离一定小于等于到我纸带的距离。因此如果这个距离大于我回领地的距离，那么对方一定无法击杀我，因此将这个距离作为对方到我纸带的距离。否则，就使用 To_my_band 函数精确计算对方到我方的距离。然后用相同的方式计算我到对方纸带的距离以及对方返回领地的距离。其后的策略，在算法流程图和算法思想中已经充分说明，这里不再赘述。

2.3 程序限制

本程序主要有三个较大的限制：

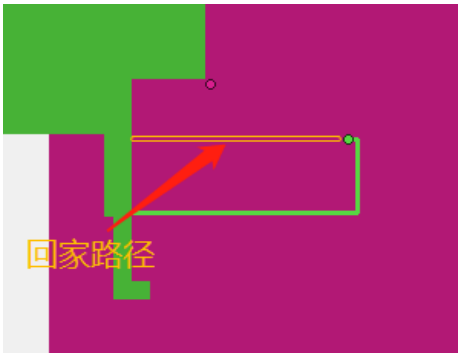
1. 在己方几乎占领全图的时候，纸卷会在一个地方逗留，不继续圈地或击杀对手

2. 不能排除对方碰撞我方回领地路上形成的新纸袋的情况

3. 不能判断预计返回的领地上的点是否接下来会被对方吃掉

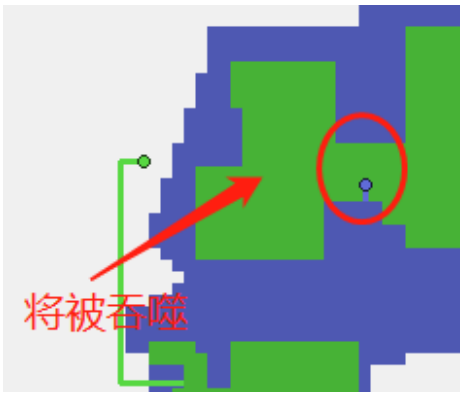
第一个限制的出现是因为 lt_benefit（远期利益估计函数）在每个区域只选取了四个点进行评估，四个区域一共也只有 16 个点参与了评估，如果我方已经占领了较大领地时，就会导致 lt_benefit 认为四个区域的利益都是 0，远期利益估值函数便失去了意义。由于返回的方向是利益最大的方向，而四个区域利益相同，按照搜索顺序，每次返回的都会是‘右’。这就会使得 AI 在一个区域内不停的打转，走不出这个区域，最后被翻盘。为解决这个问题，可以让 lt_benefit 函数多计算一些节点，增大预估范围，这能使我方在占领大部分领地后，预估函数仍能预测到一些没有被占领的区域。然而在实际比赛中，这种限制出现的次数非常少，因为我方很少能够快速占领大半部分的棋盘，所以这个限制没有被优化。

第二个限制的出现是因为 To_my_field 函数在计算回领地距离的时候，没有考虑到新形成的纸带会被对方攻击的情况（如左图所示）。在 BFS 搜索过程中，考虑的是静态棋盘，不能预测对方动向，其预测结果仅仅是“如果对方不动，我方回领地时的最短路径长”。这种预测在多数情况下问题不大，然而如果对方离 To_my_field 预测的回领地路线很近的时候，就会导致距离估值失效。为了解决这个问题，可以在 To_my_field 计算出一条回领地路线后，将这条路线保存下来，然后使用 To_his_band 测量对方纸卷到现有纸带和‘虚拟纸带’之间的距离，帮助判断我方是否在被击杀的危险之中。



第三个限制的出现和第二个限制出现的原因相似，不过这个限制的出现是因为我方回领地所要前往的那个点可能会在接下来被对方吞噬（如左图所示），导致我方回领地路径变长。这也是因为 To_my_field 只能计算静态距离导致的。这种情况导致的我方无法回领地而被对方击杀的情形屡次出现，尤其表现在我方进攻对方领地的时候。为解决这个问题，必须判断我方某些地方是否被对方所围，并即将被吃掉。而要检查被围情况，必须遍历围绕该点的对方边界，检查是否即将闭合。如果每一步都进行判定，可能导致较大的时间消耗，因此我们没有选择这种破除限制的方式。根据观察，我们发现这个问题出现的少的组，

他们的领地较为“紧密”且领地边界“平缓”（如：delta1）、圈地时所走出的图形趋于扁平的方形。让己方领地成为大块的、边界突出的地方较少，可以使对方进攻我方领地时需要在我方领地内逗留较长时间，更容易被击杀，增大到对方占领我方纸卷附近领地的困难。这种做法有其防御上的优势之处，然而必然导致圈地效率受到限制，和我方注重圈地的设计思路相悖，因此我们并没有采取这种策略。直到最后，我们仍没有找出一个在我方策略框架下，破除这个限制的较好方法。



还有一点可能导致程序出错的情况就是，如果我方纸带发现走三个方向都必定会死，或者对方可以击杀我方而我方无法击杀对方的时候，程序就会认为已经输了，不会有返回值，最后撞墙而死。

3 实验结果

3.1 测试数据

实验环境说明： 硬件配置：Intel(R) Core(TM) i7-4720HQ CPU @ 2.60GHz 2.60GHz RAM 8.00GB; 操作系统：Windows 10 领地庭中文版; Python 版本：Python 3.6.4;

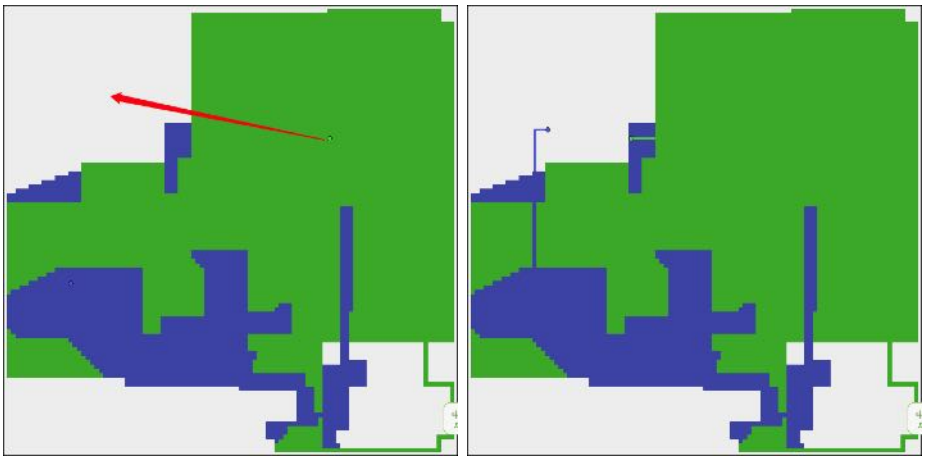
首先通过 AI_normal_wanderer 对本 AI 进行测试，将其中的 bug 修复并观察其行为。然后通过自对弈的方式观察 AI 的不足之处，进行修改。最初时，观察发现该算法容易超时，后来增加了预处理，并改进了 me_benefit 函数，将运算速度提高了许多。然后，在热身赛中，我们发现此 AI 有时会陷入局部最优，无法关注全局，因此增加了 lt_benefit 函数。同时，观察到有的组实施了“圈边”的策略，让我们意识到了边的重要性，因此在 me_benefit 函数中增加了边补偿。

热身赛场次	排名（名次/队伍数）	失败原因
1	10/13	走进自己控制范围、自杀、路径过于曲折
2	3/26	无法估计回领地路径的安全性
3	20/36	程序报错、无法估计回领地路径的安全性

在最后一次热身赛之后，修复了报错的 bug，并增加了“如果在领地附近徘徊则直接回领地”策略。

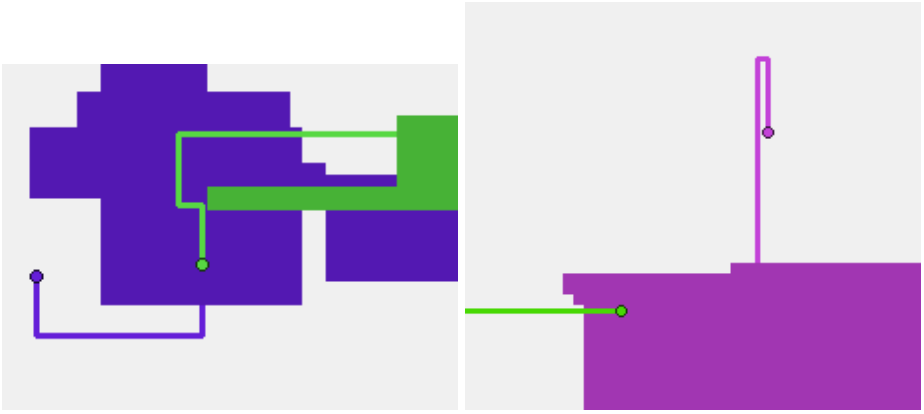
3.2 结果分析

1.lt_benefit 函数能让我方探测到远期利益较大的区域并前往（下图为比赛实况）



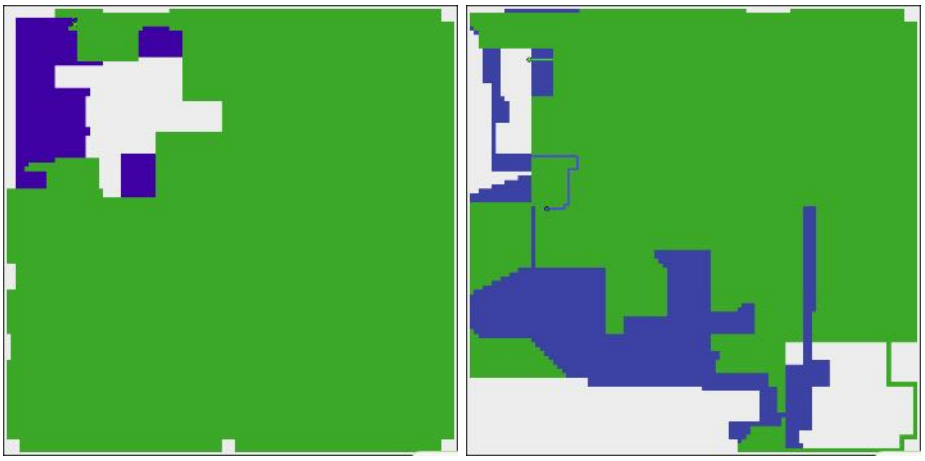
几乎所有对战过程中，lt_benefit 都能够有效引导我方纸卷走出局部利益较小的区域，这个函数作用较大。

2. 击杀、防守模块能让纸卷击杀对方、防止被击杀



在绝大多数场合，击杀模块都能有效击杀对方。而防守模块，在对方攻击我方回领地路线、侵吞我方领土时无法生效。在其他情况下，防守模块表现良好。

3. 对边界的补偿策略使得我方在后期能够围出极大的面积（下图均为比赛实况）



进行边界利益补偿的做法，在前期没有优势，而在后期，会产生巨大优势。在多数情况，如果比赛能进行 1000 steps 的场次，最终的圈地面积都是我方 AI 更大，有时甚至能够占领超过 80% 的领地。

4. 运算速度分析

算法的运行速度和预期的几乎相同，在远离己方领地时运算时间迅速增长，在自己领地附近时则运算时间很短。整个程序的主要时间开销还是在测量四个距离的时候。如果离领地较远可能导致四个距离都需要测量，并且由于计算开销和里领地距离的平方成正比，所以离领地很远的时候会导致计算量迅速暴增。整个比赛过程中，我方程序运行时间较为

稳定，从未超过 30s，性能良好。

4 实习过程总结

4.1 分工与合作

1. 分工与合作

小组成员：郭祉翀、吕悦琪、李子锦、汪昀鸿 *、叶继开

合作交流方式：

（1）通过线上交流的方式讨论小组分工，交流基本思路并分享有关资料和作业进度。包括各版代码的更新，热身赛记录的讨论在内的事件都通过线上交流完成。

（2）在二教的树下讨论区进行圆桌会议。当遇到一些不可描述的事件以及出现一些难以理解的 bug 或者是策略问题时，组长会提前通知大家进行线下见面会，群策群力解决问题并提出更好的策略。

分工：（1）代码编写：叶继开负责了基础代码的编写，以及各版代码的更新以及优化；汪昀鸿负责统筹全局并参与策略的设计与优化；吕悦琪与叶继开共同参与了代码的调试工作，郭祉翀负责策略的设计与优化，李子锦负责了全局策略的制定以及根据录像对代码策略大方向进行调整的工作。

（2）实验报告：叶继开负责算法思想，程序代码说明及实验结果三个部分，汪昀鸿负责其他几个部分的编写。

2. 小组活动：

（1）第一次组会

时间：2018/5/25

内容：通过讨论，确定了小组的基本方针，对可能出现的局势以及应对的策略讨论出了大概的框架。本次组会初步拟定采用 alpha-beta 剪枝算法计算最优路线。

（2）第二次组会

时间：2018/6/5

内容：对算法的再度修改以及对策略的调整以及优化。在之前的热身赛中由于出现了大量超时的案例，敦促我们对代码的运行进行优化。这次组会的主要内容就是降低代码运行的时间以及对热身赛中出现的一些问题进行修正。

（3）第三次组会

时间：2018/6/10

内容：比赛前最后的挣扎。针对热身赛的结果，提出了一个圈地能力更强但是更容易被击杀的策略并进行代码的修改。（P.S: 三次组会中都有一人负责照相故未出镜，实际上每次组会都是全员到齐了的）

4.2 经验与教训

组长感言：很高兴能够担任 Q 组的组长啦，感觉作为组长我其实十分的尸位素餐呢。感谢各位组员的努力，尤其要感谢叶继开大佬的实力 carry，真的是拯救了全组成员的身家性命。在我看来，我们组还是有很多地方做的还是相当不错的，其一是我们小组拥有高远的理想，为了做出迫真的效果，我们小组尝试从许多不同的角度进行思考，包括从击杀，圈地，对方策略的模拟等等角度综合，前瞻地制定策略；其二是我们小组善于发现问题，也善于解决问题，通过模拟赛事发现了超时这一巨大的问题后，我们齐心协力提出策略优化了算法，不仅大大降低了算法的运行时间，还在处理的过程中使算法的功能更进一步，同时解决了自杀的问题，最终的比赛中虽然棋差一着，却也没有掉链子；其三是小组成员处理事情的效率很高，每次线下讨论都能够在一定时间内得出比较满意的结果，可以看出大家对这份大作业都是花了心思的，都是有备而来。

毕竟金无足赤，人无完人，更何况我们小组也并没有取得出线的成绩，绝对说不上是完美，在完成大作业的过程中也的确出现了几个不容忽视的问题。其一是分工的不明确，工作量分配的不均匀，负责编写代码的叶继开同学的工作量明显要大于其他人，这不仅加大了他的压力，也使得本组代码的完成进度不是很符合预期；其二是信息来源的缺乏，在这次非对称性的程序对抗中，我们小组没有能很好地共享与发现新的信息，使得思路或多或少地有些局限性与闭塞性，对他人思路的不了解也使得我们的代码设计的目的性与直接性并不是那么的强，甚至导致了赛前的不自信，对代码功能的再一次修改反而降低了代码的稳定性。这也是我们以一名之差饮恨收场的原因之一。

组员感言：

吕悦琪：很高兴能够作为 Quebec 组的一员和组内其他同学一起合作完成这次大作业。对于大作业我最大的印象就是团结：开组会的时候大家一起挑错，提出改进策略的各种方法，面对层出不穷的 bug 不曾放弃。虽然最后 Quebec 在分区对抗中惜败未能出线，但是重在参与嘛，在这个过程中我从优秀的队友们那里学到了很多知识。能见证并全程参与这次大作业，我感到很荣幸。

郭祉翀：在这次参与讨论完成数算大作业的过程中，不仅是关于对组内大佬的相关知识的了结，最大的还是教会了我讨论的重要。在这次打磨代码中我们历经数次讨论，解决基础的策略问题，最后关于计算时间和 bug 在一起头脑风暴。很多问题都在其中得到了或多或少的改善。虽然最后惜败，但是毕竟学到了很多很多，也并不觉得丧气。重要的是我

们都有所提升，在过程中磨练了自己，这才是最重要的意义吧。

叶继开：第一次写大作业，收获很多，感悟也很多。

首先，我发现团队合作的重要性不止在于‘人多写代码的力量大’上，更重要的是不同的思维会相互激发。比如，一开始我写的搜索算法代码跑的非常慢，一位组员看了我的算法之后对它做了一点微小的改进，减少了一个大数组的使用从而极大的提高了计算速度。并且，在设计后期，我受到了一开始设计的束缚，不愿意改框架，而和队友的讨论帮助我们破除了原来的框架。

其次，团队可以提高做作业的积极性。原来以及写作业的时候很拖延，并且做到差不多就可以了。而有队友之后就会感到一种责任感，会把自己的代码写的更认真。明确的分工和每个人不同的 ddl 也会帮助消除拖延症。

最后，写一次大作业对自己的算法运用能力有很大提高。从对问题建模，确定数据结构，到确定策略和算法以及后来的实现，每一步事实上都是曲折的。尤其是一开始对策略的设计和对算法的选择，这个过程让我深刻明白了知道多少算法其实并没有那么重要，对策略的分析很多时候比知道算法更重要。

李子锦：这次的大作业中，我们在热身赛以及与别人对打的过程中发现了很多不足，可以说是挑战不断。但是我们小组的成员都非常团结，大家一起出谋划策，思考算法应该如何改进，在刚开始的时候我们的代码一度面临超时的窘境，偶尔也会出现代码行为十分奇怪的情况，但是大家都齐心协力，共同思考，在原有代码的基础上一起找出了更快且战斗力较强的算法，并在最后一次热身赛时在 17 组中得到了第三的好成绩。虽然最后的结果事与愿违，我们的算法第一局就在惊险中出线，但是我相信，我们一起努力过了就不会后悔。在编写代码以及共同商讨战略的过程中，我们小组都非常团结，很庆幸能与大家一起合作，不管结果怎样，很开心和大家一起写代码的过程！

4.3 建议与设想

建议：

1. 建议增加分组的随机性，自行组队会导致独自选这门课的同学分组困难，也会导致大佬抱团现象的产生，使得评分难以做到公正。

2. 建议在正式比赛前让每一小组提交一份算法初步设想，这样有助于比赛的顺利进行

以及各队的自我纠错，也可以增加小组的积极性。

3. 尽量一次性地给出基础设施代码，减少后续的工作量。

4. 如果时间允许的话可以增加淘汰赛的环节，防止由于实力分配不均而导致一些组拥有出线实力却在强强竞争中落败而失去出线机会。

5. 竞赛的时间安排可以进行调整，不要放在压力特别大的期末季，或者是尽量在学期初就给出大作业选题，这样会有更多的时间进行准备。

对学弟学妹们的寄语：

数算有三好，学了忘不了：

（1）拓宽生活经历，培养健全人格（即使为了大作业而熬夜，你也不能放弃，不能绝望，要对生活充满信心，相信现在的熬夜是为了将来更多的熬夜打下坚实的基础）；

（2）增长知识见闻，学习新技能（条条大路通 CS，学好数算说不定将来就多一口饭吃）；

（3）提升团队协作能力；

总之，陈斌老师的数算课，是你没有体验过的船新版本，选一下，玩一年，是兄弟，明年就和我一起来。（重修警告）

5 致谢

感谢国家，感谢党，感谢人民，感谢社会主义新时代。尤其要感谢博学多才，幽默风趣的老师，勤勤恳恳的助教，以及辛辛苦苦负责接口编写以及赛事情况统计的技术组们。也要感谢那些参加了模拟赛为我们提供了比赛数据的同学们，正是靠着这些比赛回放我们才能一次次地调整策略来做出一份完成版的代码。

6 参考文献

算法与数据结构 C 语言描述 (第二版). 张乃孝

http://www.cnblogs.com/IThaitian/p/3616550.html

http://www.cnblogs.com/pengyingh/articles/2396432.html

第十九章　F17_Romeo 报告

项洋 *、刘瀚林、耿力、王福涛、张师维

摘要：本篇报告描述了本小组的数据结构与算法课程实习作业：用于实现纸带圈地的代码。具体包括其运行原理，结果分析和编写过程。作业成果没有使用太多复杂的技巧，主要通过函数的相互调用实现运行模式的转换，在小组对抗赛中也取得了不错的成绩。

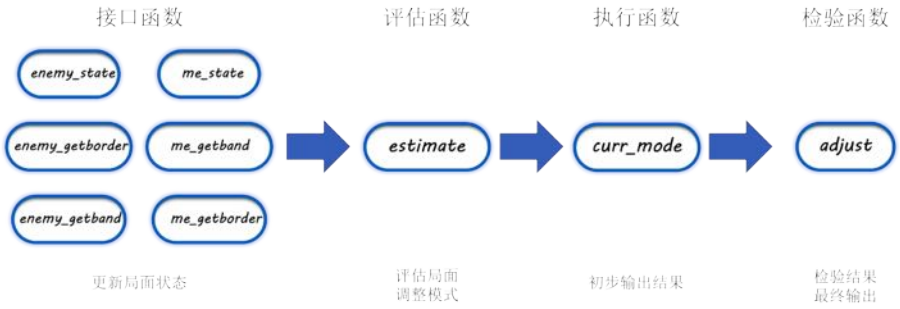
关键字：列表，字典，动态对抗，通宵作战

1 算法思想

1.1 总体思路

在圈地游戏进行的过程中，双方的位置和纸带信息均是公开的，我方的思路是通过接收和收集对方的位置信息，结合我方的位置和纸带长度进行状况的分析和比较，在进攻，防御和圈地最大化等策略中，选择较优解，然后执行，由于在每一次移动前都进行了判断，所以移动的方式是一种防守和圈地最大化为主的弱进攻性算法。

1.2 算法流程图



1.3 算法复杂度

算法运行过程中先运行更新我方和对方状态的函数 me_state 和 enemy_state，这两个函数仅有一步判断，复杂度为 O(1)

更新纸带状态的函数 me_getband 和 enemy_getband 复杂度为 O(1)。纸带头在领地外, 执行 append，或者纸带头在领地内，仅存在一步判断。纸带头回到领地，清空列表，这三种情况复杂度都为 O(1)。

更新领地边界的函数 me_getborder 和 enemy_getborder 的复杂度随情况不同而不同。一般情况下，函数主体并不会运行，存在一步判断过程，复杂度为 O(1)。当己方纸带头回到领地时，存在一个遍历纸带的过程，复杂度为 O(n)，而当对方纸带头回到领地时，存在遍历场地的过程，复杂度为 O(n^2)。而遍历过程大概每经过 n(和纸带长度同量级) 步运行一次，所以估计实际量级是 O(n)。

estimate 函数用于评估并且判断调用哪个执行函数，由于其中最多只有一重循环，所以复杂度为 O(n)。

各级执行函数，包括离开领地的 leave 函数，决定进攻的 attack 函数，决定防守的 defend 函数，决定回领地的 goback 函数，领地内游走的 wander 函数，还有拐弯的 turn 函数，由于其中最多只存在对纸带或领地边界的遍历，只有一重循环，复杂度为 O(n)。

最后校正结果的 adjust 函数，由于仅存在几步判断，复杂度为 O(1)。

实际运行时先运行更新状态的六个函数，然后运行评估状态的 estimate 函数，各级执行函数，最后校正 adjust 函数，总体时间复杂度大致为 O(n)。

实际运行中总共用时一般在 15 秒左右。

2 程序代码说明

2.1 数据结构说明

本 AI 函数使用的数据结构大部分是线性结构，以 python 中自带的列表来实现。还存在一些映射结构，以 python 自带的字典实现

2.2 函数说明

1. 接口函数

（1）me_state 函数

直接接收传入的 stat 原始数据，得到我方上一步是否回到领地，输出 True/False。

具体实现方式是缓存此时己方纸带头位置处的领地属性，并与上一步的缓存比较，从而判断其是否在上一步回到领地

（2）enemy_state 函数

同 me_state 函数，得到敌方上一步是否回到领地。

（3）me_getband 函数

直接接受传入的 stat 原始数据，更新全局变量 storage 中我方纸带 storage[‘me_band’] 的数据。

检测到我方在领地外，在队列末尾添加当前纸带头位置元组：当我方回到领地，清空队列：当我方在领地内，不操作全局变量

（4）enemy_getband 函数

同 me_getband 函数，更新全局变量 storage 中敌方纸带 storage[‘enemy_band’] 的数据

（5）me_getborder 函数

直接接受传入的 stat 原始数据，更新全局变量 storage 中我方纸带 storage[‘me_border’

] 的数据。

当己方纸带头回到领地时，将我方纸带中的元素全部添加到 storage[‘me_border’] 中，并遍历 storage[‘me_border’]，去除其中实际位于领地内的点。而当对方纸带头回到领地时，遍历场地，去除 storage[‘me_border’] 中不在我方领地边界上的点。一般情况下，仅存在判断过程，函数主体并不会运行。

（6）enemy_getborder 函数

同 me_getborder 函数，更新全局变量 storage 中敌方纸带 storage[‘enemy_border’] 的数据

2.Estimate 函数

接受原始数据和之前六个接口函数更新的全局变量信息，评估局面，决定是否开始攻击或者是否取消攻击开始防御。

具体估计方法是通过遍历双方纸带和领地边界，得到敌方返回领地最短距离、我方攻击敌方纸带最短距离、我方返回领地最短距离，预判我方回领地路径并得到敌方攻击到我方纸带的最短距离。当我方攻击敌方最短距离小于敌方回领地距离，执行攻击函数。当敌方回到领地 (成功逃脱)，转为防守函数。

3.Adjust 函数

接受执行函数的结果，以及全局变量中己方纸带的信息。当执行下一步操作时会导致撞墙或自杀时强行调整方向。

虽然撞墙或自杀的情况一般不会出现，但一旦出现结果十分致命。当调整函数判断下一步将撞墙时，由于不清楚何种条件下会出现此种情况，处理方式也是简单地选择一个不会自杀、撞墙或将自己绕死的方向并输出，避免死亡。

4.Turn 函数

接受原始数据和之前六个接口函数更新的全局变量信息。根据我方当前的方向，获取如果方向未发生改变下一帧时的预计坐标。

如果发现与我方纸带中某一坐标重合，则检测这个坐标在 storage[‘me_band’] 中对应的 index 值，上溯 index 减小的那一点，并让纸带头拐向 index 值较小的那一端，从而避免纸带绕成环困住自己。否则，由当前坐标与边界突出点的相对位置确定转向——向边界突出点的大方向转向。此后，再在此次转向的基础上确定新方向与新的预计坐标，判断

是否越界或碰撞我方纸带，从而决定是按照此转向还是变更转向。

5.Wander 函数

接受原始数据和之前六个接口函数更新的全局变量信息。当我方在自己领地内时，找到到我方边界的最短路径并垂直边界出领地。

通过遍历我方领地边界点（不包括场地边界），并寻找最近的出领地的点及其对应路径，到达此点，并垂直边界出领地。

6.Leave 函数

接受原始数据和之前六个接口函数更新的全局变量信息。以实现在保证安全时，控制离开领地的距离，以达到最大圈地的目标。

首先按照当前方向确定下一帧时的预计坐标，及时进行转向从而防止越界或是碰撞我方纸带。之后通过遍历获取我方坐标到敌方纸带的最小进攻距离和敌方坐标到敌方边界的最小逃跑距离，比较两者相对大小决定是否进行进攻。之后再通过遍历获取敌方坐标到我方纸带的最小防御距离，一旦最小防御距离的三方之一小于等于我方出领地距离，即转向并开始防御。若以上情况皆未发生，则继续前行，以达到最大圈地目标。

7.Defend 函数

接受原始数据和之前六个接口函数更新的全局变量信息。用于防御对方潜在攻击保证己方安全。

首先按照当前方向确定下一帧时的预计坐标，及时进行转向从而防止越界或是碰撞我方纸带。之后通过遍历获取我方坐标到我方边界的最小返回距离，同时模拟最近返回路线，通过遍历获取敌方坐标到我方纸带与模拟返回路线的最小防御距离，当最小防御距离只比最小返回距离略大时，开始返回边界；当最小防御距离小于最小返回距离时，开始执行玉碎战术，疯狂进攻；以上情况皆未发生，则继续前行，以达到最大圈地目标。

8.Attack 函数

接受原始数据和之前六个接口函数更新的全局变量信息。目的是发动攻击置对方于死地。

Attack 函数在防御函数失效后就会发动。Attack 函数表面上是攻击函数，其实内部也包含了防守函数 protect，就是在保证自己不被自己的纸带绕死，不会越界的前提下，进行疯狂的拼命的攻击。函数被调用时，首先会根据当前的方向确定下一帧的预计坐标。如果

这个预计坐标是己方纸带区域或会出界，就会调用 protect，而不优先进行攻击。在有了上述那些基本保证后，函数通过遍历获得己方纸带到敌方纸带最小距离的点的坐标 M，并由此确定己方纸带的追击路径，进行下一步的攻击。这里值得注意的是上述的点 M 会在己方纸带攻击的过程中会发生变化，这就需要即时获得 M 的位置，即要不断更新，重新遍历。总的来看纸带在调用 Attack 函数后就像个勇士，朝着 M 点不断进攻！

9.Goback 函数

接受原始数据和之前六个接口函数更新的全局变量信息。用于使纸带返回自己领地。

首先按照当前方向确定下一帧时的预计坐标，及时进行转向从而防止越界或是碰撞我方纸带。之后根据我方坐标与边界突出点的坐标的相对位置，沿最短路线快速返回边界，以躲避进攻并完成圈地。

2.3 程序限制

1，出领地的时候可能出现描边走，形成浪费。

原因：wander 函数出领地时会直接选择垂直于边行动，如果在出现直角的边界时，就会由于这样的垂直走程序，进入描边，在效率上形成浪费。

2，防守函数执行不到位或者过于保守。

原因：因为执行程序的时候，以保证自己回到领地优先，所以某些时候会出现牺牲效率的情况，而有时候，又因为判断失误，导致进入 estimate 函数，出现莽撞进攻的危机。

3 实验结果

3.1 测试数据

实验环境说明：

- 硬件配置：（CPU/内存）i5-6300 8G
- 操作系统：（名称/版本）win10
- Python 版本：（版本号）python3.7

使用了官方配备的 visualize 代码来进行试验测试。

由于过程中没有输出，测试数据只能定性分析。详见 3.2 结果分析部分。

3.2 结果分析

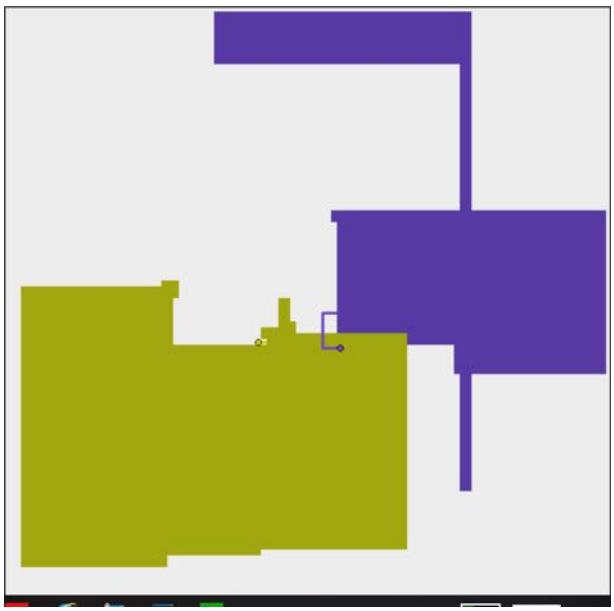
算法在对战过程中，不乏优点和智障之处，而一场战斗可以体现出几乎所有函数的执行过程。现在借用下面这个图分析一下：

我方纸带：黄色

对方纸带：蓝色

刚开始圈地的时候，由于对方纸带较远，我方先采取了圈地最大化的策略，迅速取得了一大块黄色的领地，在这个过程中，体现了我方以安全偷鸡为主的策略，追求利益的最大化。

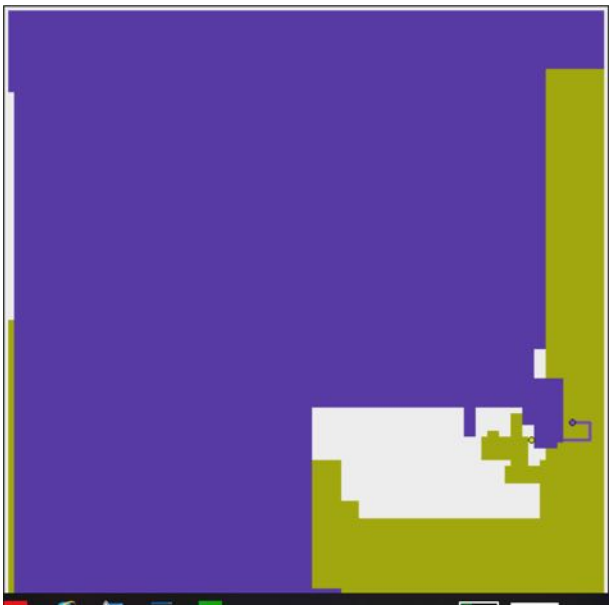
但是，对方自然不同意，便出现了在中间部分的交锋，在小范围内，经由我方 defend 函数判断，由于距离较远，所以可以安全回到领地，于是我方纸带决定先迅速回到领地，再从领地出发，开始执行圈地程序。



但是，对方程序不依不饶，继续追逐我方纸带，在地图的中部进行缠斗，通过 estimate 函数判定后，我方纸带选择执行进攻函数，在中部与对方发生了进攻战，由于双方都有

defend 函数控制着双方的战斗，所以仅仅是在小范围内形成斗争后，迅速回到领地，继续各自的圈地。

在这幅图中，则是很好的显示出了我方函数的问题，由于过于保守的防守程序，以及相对而言，仅仅只是在回不到领地时才进入的 estimate 函数，我们被对方的偷鸡策略包围，在圈地面积上被反超出现了劣势。



4 实习过程总结

4.1 分工与合作

项洋、刘瀚林、耿力、王福涛为代码编写小组，张师维负责实习报告的总结。其中，项洋负责编写各函数的接口以及代码的修改和调试，刘瀚林负责编写圈地最大化相关函数，耿力负责编写防守函数以及相关算法的修改和调试，王福涛负责编写进攻函数。

4.2 经验与教训

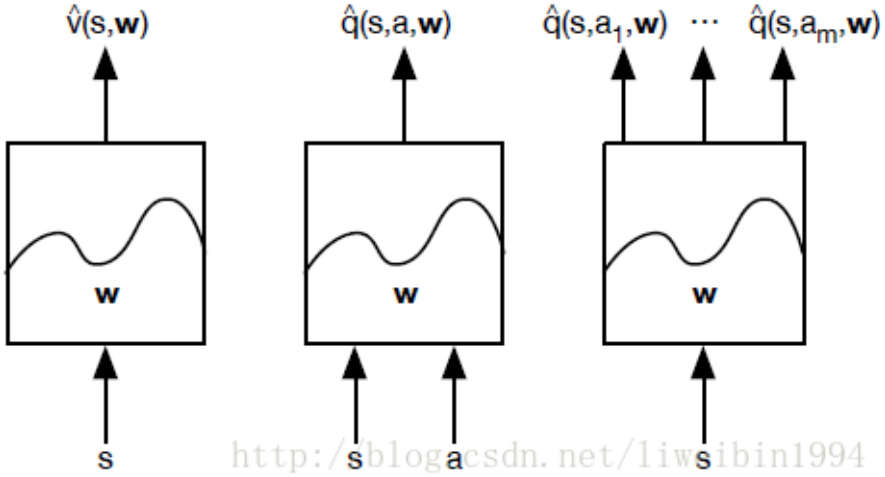
整个算法的成型经历了许多次的讨论，分析，放弃，与再思考。尽管在尝试的过程当中，经立过初始目标的构思错误，编写过程中的各种修改策略，放弃复杂构思以及对自身策略的重构与修改，以下将详细记录这耗尽心血看待自己的“智障儿子”的出生过程。

1，初见圈地：

在第一次的小组开会前，我们先分别学习了相关的作业要求和大致的完成方式，并分别试玩了 paper.io。由于第一次开会前，小组内部并没有人尝试具体编写相关算法，所以第一次的开会虽说目的是进行使用的算法的策略，但实际上更多的是一次天马行空的畅所欲言，以加深大家对于这次大作业的理解。在这次讨论中，大家提出了两个可能的实现方法，一种是项洋同学提出来的，利用价值函数进行判断，通过预先设立判定方式来对纸带的行进进行控制，另一种是刘瀚林同学提出的，尝试编写机器学习类的算法，通过长时间的自我博弈，来提高算法判断的能力，以期在有效时间内寻找较优解。尽管在之后的尝试中，发生了各种各样的问题，但是，这次的讨论至少给了我们两个大致的方向。

2，一周的失败学习和尝试：

当开始尝试编写代码的时候，我们决定两条路径并行，由项洋和王福涛负责价值函数算法的尝试，刘瀚林和耿力负责尝试机器学习类算法，然而，这两次尝试最终都以失败告终。价值函数的编写，这样一个策略，最重要的地方便是价值函数。因为在这样一个圈地游戏中，每个位置就是一个状态，这样的状态是连续的我们很难用表格来表示，所以我们需要用价值函数来逼近，价值函数是用一个函数进行估计（estimate value function with function ）approximation) 依赖的是如下图所示的一个输入输出结构：



我们可以利用调整参数向着负梯度移动的方式，去寻找局部的最小值。可以通过以下这个函数表示：

- Goal: find parameter vector $\mathbf{w}$ minimising mean-squared error between approximate value fn $\hat{v}(s,\mathbf{w})$ and true value fn $v_\pi(s)$

$$J(\mathbf{w}) = \mathbb{E}_\pi\left[(v_\pi(S) - \hat{v}(S,\mathbf{w}))^2\right]$$

- Gradient descent finds a local minimum

$$\begin{aligned}\Delta\mathbf{w} &= -\frac{1}{2}\alpha\nabla_{\mathbf{w}}J(\mathbf{w}) \\ &= \alpha\mathbb{E}_\pi\left[(v_\pi(S) - \hat{v}(S,\mathbf{w}))\nabla_{\mathbf{w}}\hat{v}(S,\mathbf{w})\right]\end{aligned}$$

- Stochastic gradient descent *samples* the gradient

$$\Delta\mathbf{w} = \alpha(v_\pi(S) - \hat{v}(S,\mathbf{w}))\nabla_{\mathbf{w}}\hat{v}(S,\mathbf{w})$$

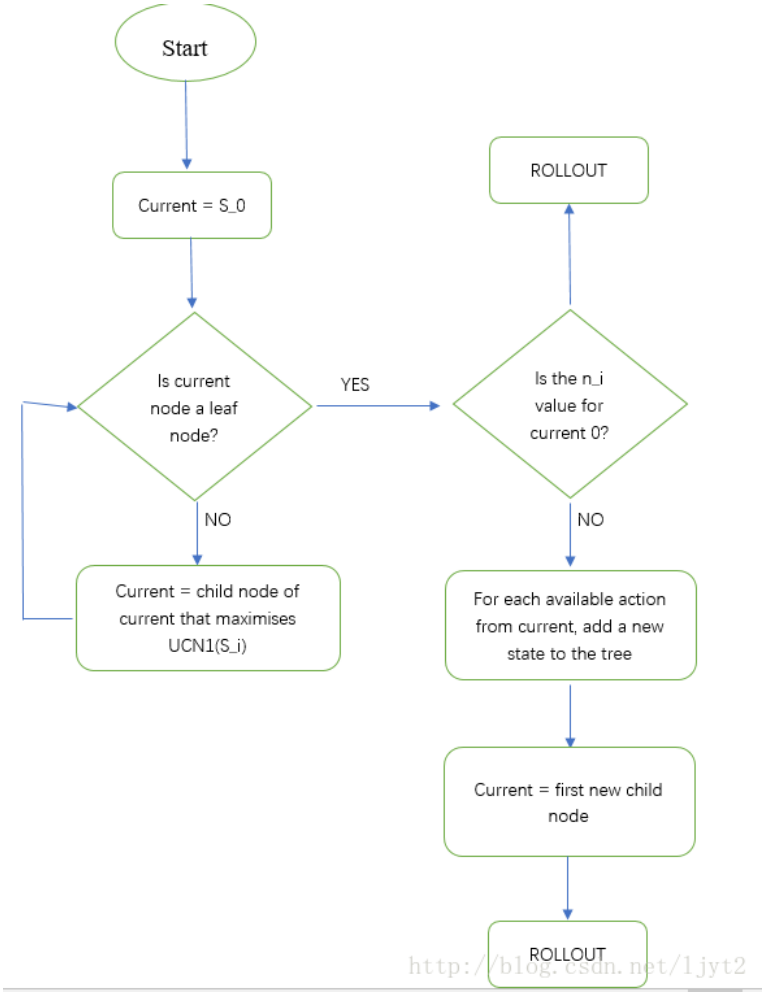
- Expected update is equal to full gradient update

这样的一个查找过程中，通过预设算法的方式，便可以寻找到解决的方案。但是，由于难以写出这个价值函数, 经过小组讨论，没有想到如何用一个合适的算法，对已知的信息进行判断，同时，由于没有学习过程，我们没有监督数据，需要用估计值代替真实值，这个过程的实现，本应该利用线性价值组合来近似，但是在尝试过程中，由于解空间状态过多，而宣告失败。

同期，刘瀚林和耿力组对于机器学习类算法的尝试也宣告破产。在设想之处，由于蒙特卡罗算法不需要先验知识，所以在搜集解空间的信息方面有了相对的便利。由于是个随机采样的算法，采样的数据会根据之前的采样结果进行调整，通过一下四步模拟：

1. Tree traversal:
2. Node expansion
3. Roll out (random simulation)
4. Back propagation。

利用以下流程图进行相关的运算



通过这样的一个类似的方案，反复迭代，在设计的时间区间内，寻找一个最优策略，

并执行。这样的算法，避免了解空间很大时候对于数据的处理问题的同时，还可以在学习的过程中提高能力，寻找最优解。听起来不错的一个方法..... 但是，我们都由于专业知识和关于数据结构与算法认知的局限性，只能临时学习相关的算法，参考了《Reinforcement Learning An Introduction》, 也学习了相关的课件。在《A Survey of Monte Carlo Tree Search Methods》中，我们本来寄希望于搜索树零和博弈的..... 但是学习过相关知识后我们发现自己只能堪堪看懂而无法利用在圈地游戏中，这个方案也宣告失败。

凌晨两点的全家，已经开始发出了嘲讽笑容的组员.JPG

3，绝地求生地编写“智障儿子”

认清了现实的困境以及重新评估了自身实力之后，小组成员决定，重新规划编写程序的方案和思路，紧紧贴合地球与空间科学学院的口号“仰望星空，脚踏实地。”尽管仰望星空的过程中失败了，但是我们依旧学到了许多知识，而现在，是脚踏实地编写可行算法的时候了。

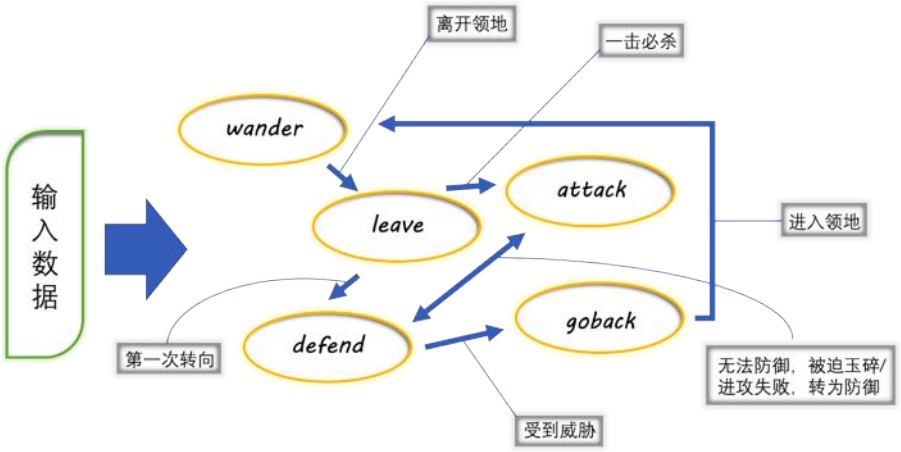
初始的思路分为以下几部分：

1，编写无对手时圈地最快的方案。

2，编写判断之后策略的函数：即防守，进攻，还是继续利用圈地最大化策略。

3，当发现对方有靠近趋势时，开始利已给的 defend 函数进行判断，当可以回去时，利用圈地较优策略，回到领地内，当无法逃脱时，进入 attack 函数，进行时刻对着对方纸带的饿暴力攻击模式。

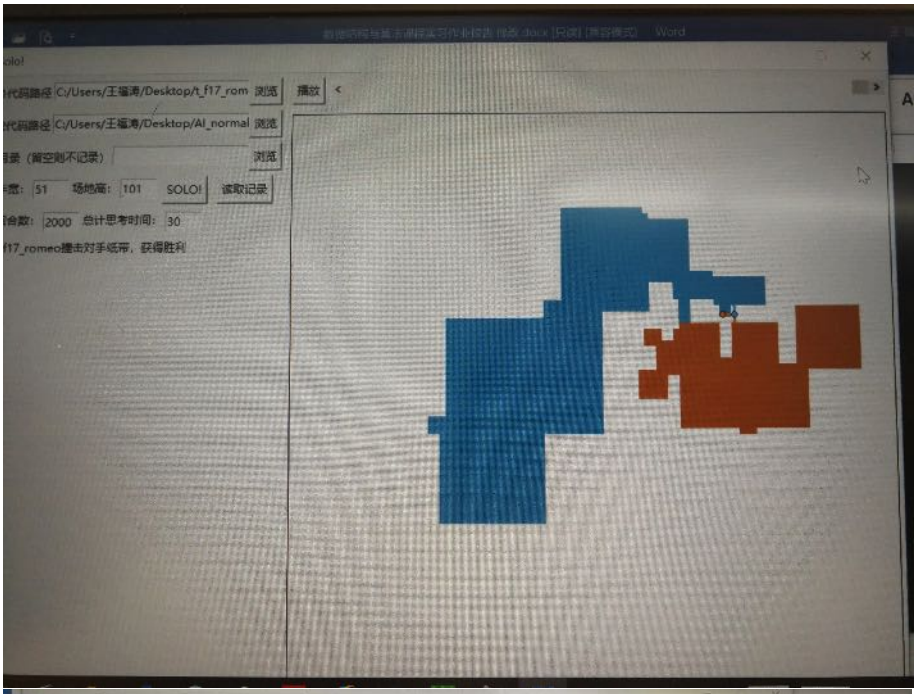
4，于是，形成了以下的内部流程图：

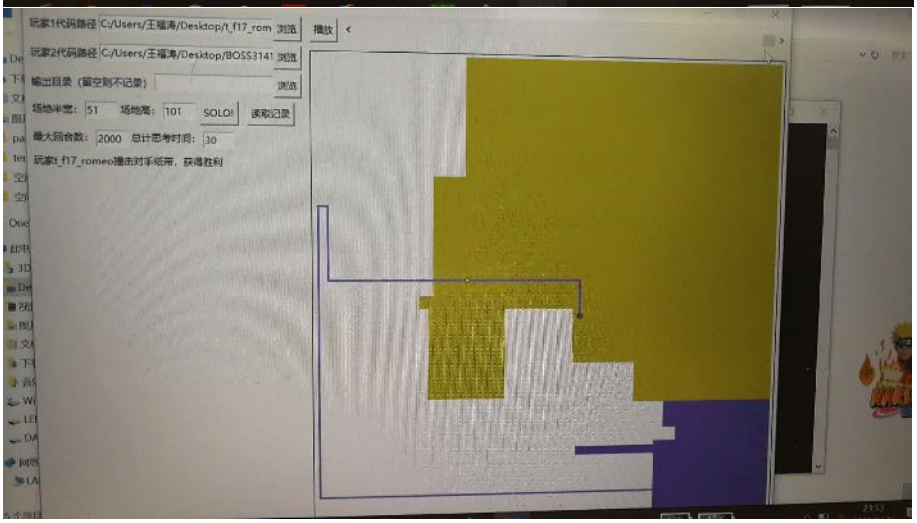


尽管是一个看似并不高大上的程序，但是重在思路清晰，执行性高。大家迅速分工，由项洋编写各接口，耿力编写 defend 函数，王福涛编写 attack 函数，刘瀚林编写圈地最大化的函数和 wand 函数。具体的算法流程参考以上，终于，在连续两天的努力后，智障儿子终于出生了，尽管还有许多 bug 没有 de，参数调试也没有到位，但是，终归是依赖着偷鸡和奇葩到组员也难以领会的走法，赢了测试函数，某组员甚至开心的连夜发了朋友圈：

5，比赛前的调试：

在经过了参数的调节和 bug 的修改后，这个简约却不简单的算法打出了比预想中更好的战绩，依赖着快速的判断和果决的行走，在参赛与其余小组的交锋中，取得不错的战绩，以下便是友谊赛比赛图：





4.3 对于此次编写大作业的感想

王尔德曾经说过“我们都身处沟壑，但总有人仰望星空。”在这次的数算大作业中，我们无疑是处于沟壑中的，没有足够的知识储备和代码编写能力，有的只是一腔热血，和属于大一新生的初生牛犊的挑灯夜战能力。但是，我们不曾后悔没有大腿作陪，也不后悔第一周的仰望星空，正是在只能依靠自己的环境中，我们快速提高了自己对于算法的认知，对于如何修改 bug 的心得，以及在价值函数和蒙特卡洛算法的学习中，所感受到的算法的奥妙和魅力，我们相信，也同时坚信着这次大作业的学习和编写过程，将是我们在大学生活中难以忘怀的艰难而愉快的，与小组同学共同合作奋进的学习过程。

“构思算法和编写代码，真快乐啊!”这不是王尔德说的，也不是鲁迅说的，仅仅，是属于这个小组五位成员的肺腑之言。

5 致谢

感谢北京大学地球与空间科学学院的陈斌老师，他开设的他开设的数据结构与算法课程给了我们这个编写大作业程序的机会，他在课上所讲的知识是我们完成代码的基础。

感谢数据结构与算法课程的 7 位助教：易超、陈旭、赵宇、冀锐、滕沅建、缪舜、张颢丹。

感谢给我们提供自己算法进行对战的两个小组。

6 参考文献

https://github.com/chbpku/paper.io.sessdsa/tree/master/AI

《数据结构与算法 -期末大作业》陈斌

https://blog.csdn.net/liweibin1994/article/details/79206489

CSDN 博客 - 强化学习：价值函数的逼近

Cameron B. Browne, Member, IEEE, Edward Powley, Member, IEEE, Daniel Whitehouse, Graduate Student Member, IEEE, Simon M. Lucas, Senior Member, IEEE, Peter I. Cowling, Member, IEEE, Philipp Rohlfshagen, Member, IEEE, Stephen Tavener, Diego

Perez, Spyridon Samothrakis, Graduate Student Member, IEEE, and Simon Colton, “A Survey of Monte Carlo Tree Search Methods” in IEEE TRANSACTIONS ON COMPUTATIONAL INTELLIGENCE AND AI IN GAMES,VOL.4,NO.1,MARCH 2012

Richard S. Sutton and Andrew G. Barto,” Reinforcement Learning: An Introduction (second edition)”

第二十章　F17_Tango 报告

杨一龙 *，吴政霖，姜晓飞，侯远樵

摘要：本算法的基本原理是根据在我方回合敌方纸带头与我方纸带头距离的奇偶性划分为优势和劣势情况。再将决策的算法分为三个核心部分：进攻、防守、圈地。在优势劣势情况下，根据不同的判断条件来决定调用这三个算法的优先级。涉及了栈、队列和列表的数据结构，分别利用了其 LIFO、FIFO 和快捷存储数据的性能。本算法主要采用博弈算法，根据不同答案局势来调用不同的策略。对于实验数据结果，本算法对于主要组成部分均进行了测试，数据结果符合算法的目的。

关键字：栈，队列，列表，博弈算法

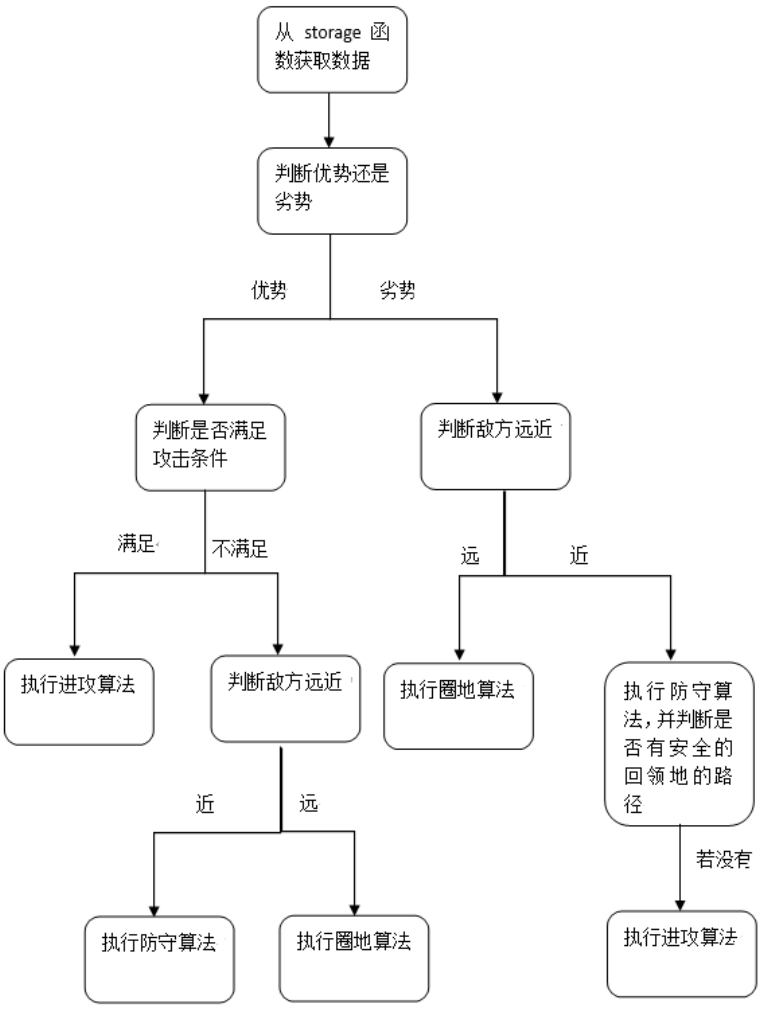
1 算法思想

1.1 总体思路

本算法首先把游戏过程分为了调用数据和生成策略两部分，调用数据部分较为简单。对于生成策略部分，我们首先发现如下事实：对于仅有纸带头的最简单情况而言，当我方纸带行走时，如果我方纸带头与对方纸带头之间的距离为奇数，则我方纸带头与对方纸带头撞击时必定是我方纸带头先手，此时可保持不败。故我们首先编写了进攻算法：当我方纸带行走时，如果我方纸带头与对方纸带头之间的距离为奇数，且我方纸带头与敌方纸带头的距离小于敌方纸带头回领地的最短距离，则朝着敌方纸带头的方向行走，若敌方纸带头“逃跑”，则朝着敌方纸带上最接近我方纸带头的点前行进攻。此算法最终用在优势情况下进攻和劣势情况下无法回领地时的拼死一博。之后，由于这是一个圈地游戏，我方在敌方距离我方比较远时采用圈地算法，以效率最高的方式进行圈地。此外，当敌方距离我方

较近且不满足进攻条件时，我方采用防守算法，以自家领地为依托防止敌方的侵袭。综上所述，本算法根据不同答案局势来调用不同的策略，以达到利益最大化的目的。

1.2 算法流程图



1.3 算法复杂度

对于数据的调用，算法核心为二维列表的遍历，最好情况：轨迹不发生闭合时，遍历的点数为常数，时间复杂度为 O（1），最坏情况：发生闭合时，遍历的点最大正比于 n^2，时间复杂度 O（n^2），平均为 O（n^2）。

对于各种策略：

进攻代码的核心为对敌方纸带头及敌方纸带区域的遍历，平均时间复杂度为 O（n^2）。

圈地代码的核心为对我方纸带及敌方纸带区域的遍历，平均时间复杂度为 O（n^2）。

防御代码中，在纸带头前方有领地的情况下，设边界点数量为 n，则平均需要比对 n/2 次才能判断前方是否有领地，所以复杂度为 O(n)。

在纸带头侧面有领地的情况下，算法复杂度的主要贡献来自于选择回领地的路径部分。设纸带头到边界平均距离 d。在选择回到领地的边界点时，平均需要考虑 n/2 个边界点，而在计算回到每个边界点所需距离时，因为需要判断路径上是否有自己的纸带，所以每个边界点要调用 wholeplate 判断路径上点是否有纸带。所以复杂度为 O(n*d)。

在纸带头侧面和前方都没领地的情况下，主要是判断出该情况所需的算法占主要部分，复杂度为 O(n)。

故总体而言，本算法的时间复杂度为 O（n^2）。

2 程序代码说明

2.1 数据结构说明

在圈地算法内，定义的函数 Enemyturn 中使用了栈的数据类型，虽然没有调用专门的 Stack 类，但是栈的特性和功能，比如压栈和弹栈，都通过对于内置列表结构的相应操作实现。

在防守算法中，定义了 coordinate 类：一个点的 xy 坐标。并且将接下来几回合的决策都存在一个队列中，利用了队列“FIFO”的特性——先做的决策先执行。

2.2 函数说明

调用数据部分：本部分较为简单，故主要说明接口和作用。

getTheNumber(stat, i, j)

作用为通过调用 stat 中的 fields 和 bands 判断点的类型，获取该点在二维列表内的编号，各个编号的含义如下所示：

a0 我方领域，无纸带

a1 我方纸带头在我方领域

a2 我方纸带在我方领域

a3 对方纸带头在我方领域

a4 对方纸带在我方领域

b0 对方领域，无纸带

b1 对方纸带头在对方领域

b2 对方纸带在对方领域

b3 我方纸带头在对方领域

b4 我方纸带在对方领域

c0 空白

c1 我方纸带头在空白领域

c2 我方纸带在空白领域

c3 对方纸带头在空白领域

c4 对方纸带在空白领域

operate_ij(wholePlate, myOutline, myPaperTrail, enemyOutline, enemyPaperTrail, i, j)

作用为利用栅格化的数据判断场地信息的拓扑性质，判断一个点 (i, j) 的种类并加入相应的列表

Traverse_all(stat, storage)

作用为遍历场地内的一个矩形区域，该区域包含了纸带头经过的所有点, 返回新的 wholePlate, myOutline，enemyOutline, myPaperTrail, enemyPaperTrail

Traverse_part(stat, storage)

作用为仅遍历一个回合内纸带头可能影响到的区域的点，返回新的 wholePlate, myOutline, enemyOutline, myPaperTrail, enemyPaperTrail

isClosed(stat, storage)

作用为判断路径是否在本回合发生闭合。如果纸带头上回合在领域外，本回合在内，判定闭合

isInsidePoint(id, p, i, j)

作用为判断某已知的领域点 (i, j) 是否为某 id 内点, p 为标准二维列表。定义内点：若某点周围 8 个点都是与之 id 相同的领域点，则认为此点是内点，否则为边界点。

storageUpdated(stat, storage)

作用为每回合（包括第一回合）更新 storage 中所有数据

storageUpdated_record(stat, storage)

作用为每回合更新 record

进攻函数部分：

Condition(enemyOutline,sstat,wholePlate)

条件函数，判断与对方在远离领地的敌方遇见且我方优势。

relative_direction(stat):

判断两点纸带头相对位置的函数。

返回一列表，第一项表示东西方向相对位置，第二项表示南北方向相对位置。

attack(myOutline,myPaperTrail,enemyDirection,myDirection,enemyOutline,stat,wholePlate):

进攻函数，用于调整纸带头方向。返回一字符，其首字母代表方向如何改变。思路在算法总体思路中已述，解决对方未“逃跑”时的进攻情况。

foolishattack(enemyOutline,stat,wholePlate):

进攻函数，用于调整纸带头方向。返回一字符，其首字母代表方向如何改变。思路在算法总体思路中已述，解决对方“逃跑”时的进攻情况。

圈地函数部分：

dis_meandmyOut()，

作用：得到本方纸带头和边界的距离，用于敌我相对距离的安危程度，返回一个参数

表，三项分别为距离，最近点的横坐标，纵坐标

参数：myX,myY,myOutline(本方边界)，即本方横纵坐标，边界

思路：采取最简单的方式，即遍历边界，保存距离最小值找到和纸带头最近的一点。这里距离的定义是最朴素的绝对值相加定义。

dis_enemyandmyband()

作用：得到对方纸带头和本方纸带的距离，和前一函数共同起判定作用。返回列表，三项意义同 1 函数

参数：myPaperTrail(本方纸带),enemyX,enemyY

思路:，没有采用复杂的思路，同样采取简单方式实现，遍历纸带，保存最小值。

wall()

作用：判断本方是否即将撞墙。返回布尔值（冲突或不冲突）

参数：纸带头位置，本方方向

思路：直接通过与边界的距离判断，如果小于 3 就视为存在撞墙的潜在危险

circle()

作用：主要圈地函数，通过调用存储区 record 获取所有信息，确定当前所在的阶段数，内部设置并调用一些局部函数，达到决策目的。返回左转，右转或直行

参数：record（存储的比赛情况），本方边界，本方纸带，本方方向，双方纸带头坐标。

思路：在构思时考虑到圈地效率最大时较易实现的方案是正方形，正方形分四条边，尽量走齐这四条边，如果对手足够靠近自己，则为了避免对手的威胁，暂时舍弃正方形的方案，从第三阶段开始以最快的速度回到领地中。判断形势的依据是 1，2 两个函数的返回值之比。处理特殊情况时，对于撞墙，视为对手即将接近情况，提前进入下一阶段，对于纸带冲突情况，先判断自己的纸带有没有部分在左侧再判断左转还是右转。此后直行一直到前方和左右没有纸带，然后转向原来的方向继续执行原计划。既然是四条边，那么圈地过程可以分为四个阶段，分别处理 3 次转弯后的路线，在衔接处用存储的参数判断。在前两步为了保证接近于正方形，不判断是否应以最短路回领地，而在第三阶段倒转指向时才判断，这样虽然具有一定的风险，但考虑到圈地效率问题，只好冒险以求最大利益。

选择利用存储区是因为直走时什么时候应该转弯，转弯时向哪个方向都需要记住之前

的决策，而每次调用函数无法记录这些信息，因此只好在函数内部读取和改动存储。

lefthand()

作用：判断本方纸带是否在左侧，返回布尔值（是或否）。

参数：本方坐标，本方方向，本方边界

思路：考虑不撞带问题时，想从转弯的根源上解决指代冲突后是否还有危险，因此在避免冲突之前就预先考虑接下来的转弯怎样才能避免死胡同。其实本函数在设计上较为粗略，绝对，遍历本方纸带，只要发现有点在左侧，就返回 True，然后控制 circle 向右转。

防御函数部分：

主要以两个函数为主。

defense 函数：

在场地外敌人靠近时调用，返回值是 True 则表示可以回到领地，False 则不可以。函数通过将操作存在一个队列里来进行决策。

参数有：存储操作的队列、领地边界列、纸带轨迹、我方纸带头方向、敌我 x 和 y 坐标、场地边界。

定义了 coordinate 类：一个点的 xy 坐标。

首先判断纸带头前方有没有领地，如果有的话，则决策直接往前方走，在操作队列中存储相应个数的操作符。这里没有判断途中是否有纸带，但如果有，则在直行到纸带前方时通过 play 函数最后的防自杀程序转向，从而转变为其他情况。

如果判断前方没有领地，则再判断纸带头两侧是否有领地。如果有的话，则考虑 L 型（直走转弯再直走）或 U 型（直走转弯直走转弯再直走）路径回到领地。遍历领地边界的每个点，先判断按照 L 型回到领地是否可行（路径上是否有纸带），如果不可行，再考虑 U 型的（一次次增加 U 的边长，即增加第一段直走和最后一段直走的距离）。在确定了每个点的路径后，计算每个路径的长度和这段路径带来的面积收益。最后在长度小于敌人离我方纸带最短距离的路径中选择一个面积收益最大的路径，并返回 True。如果没有这样的点，则返回 False。

如果侧面也没有领地，则在领地中抽取一个点，如果这个点在前方（包括斜前）上，则继续前行。这样在走一些回合后，可以转化为前面的第二种情形。

defensiveback 函数:

在领地内时调用。起到离开领地并追击敌方的作用。根据敌方相对方位，像相应方位移动。

2.3 程序限制

本程序采用的判断较少，故一般不会出现运行时间过长的情况，算法中存在防撞墙以及防撞自己的部分，故在这种临界状态下不受影响。限制算法的主要为其思路及处理，算法本身限制较少。

3 实验结果

3.1 测试数据

实验环境说明：

由于算法拼接起来较为庞大，故测试分块进行，采用模拟情况调用算法，比较理论结果与实际结果差别的方法进行测试，测试结果均符合理论结果。相关照片已表明分属哪一部分。由于圈地算法较为复杂，过程简述如下：

由于本算法只是总算法的一部分，不具有单独和 AI 对抗的能力（因为没有基本的数据处理系统），因此测试的难度比较大。为了在能力范围内最大限度地还原真实情况，本代码的测试采用了中间结果输出和读入手写场地混合的方式。

1，中间结果输出说明

输出的主要内容是 storage 存储的内容，每一步打印其所有内容，有时辅助以一些中间函数的返回值，比如测试中打印 lefthand 函数，dist_enemyandmyband() 函数，dist_meandmyOut() 函数返回值。

附带的图片是一次完整的圈地。见初始图 0，00。

以下并非全部步骤

一开始在领地里

False# 是否纸带冲突

False# 对手是否在右手边

1000 0# 敌人离本方纸带的距离

None# 下一步的决策

'dhistory': [3, 3, 3, 2, 1, 1], 'distant2': 0, 'mydhistory': [1], 'conflict': False, 'mycurrent': 0, 'urgent': False, 'home': 'y': None, 'x': None, 'mystep': 1#storage 的所有内容（依次是）：对手行进历史，正方形边长，纸带冲突，当前步数，是否紧急，紧急回归点

初始步顺利直走，更新 mystep,mydhistory

False

False

8 1

None

'urgent': False, 'home': 'y': None, 'x': None, 'distant2': 0, 'mycurrent': 0, 'mydhistory': [0], 'mystep': 1, 'dhistory': [3, 3, 3, 2, 1, 1, 1], 'conflict': False

继续直走

False

True

6 2

None

'home': 'y': None, 'x': None, 'distant2': 0, 'mydhistory': [0], 'urgent': False, 'dhistory': [3, 3, 3, 2, 1, 1, 1, 1], 'conflict': False, 'mystep': 1, 'mycurrent': 0

马上就要转弯了

False

True

4 3

l

'conflict': False, 'mystep': 2, 'mycurrent': 2, 'dhistory': [3, 3, 3, 2, 1, 1, 1, 1, 1], 'mydhistory': [0, 3], 'home': 'y': None, 'x': None, 'urgent': False, 'distant2': 3

我们发现程序选择了右转，这是符合实际的，而且更新了 mydhistory，边长 distant2,mycurrent 当前边上的步数,mystep 当前阶段数。（由于二维列表的写法和场地是对称的，因此显示的左转其实是右转）

False

True

5 4

None

'distant2': 3, 'dhistory': [3, 3, 3, 2, 1, 1, 1, 1, 1, 1], 'urgent': False, 'mydhistory': [0, 3], 'mystep': 2, 'conflict': False, 'mycurrent': 3, 'home': 'x': None, 'y': None

False

True

6 5

l

'dhistory': [3, 3, 3, 2, 1, 1, 1, 1, 1, 1, 1], 'mydhistory': [0, 3, 2], 'mystep': 3, 'urgent': False, 'home': 'x': None, 'y': None, 'distant2': 3, 'mycurrent': 2, 'conflict': False

再度转弯，更新数据

False

True

4 3

1.3333333333333333# 输出 temp 值

None

’mydhistory’: [0, 3, 2], ’conflict’: False, ’dhistory’: [3, 3, 3, 2, 1, 1, 1, 1, 1, 1, 1], ’mycurrent’: 4, ’distant2’: 3, ’urgent’: True, ’home’: ’y’: 2, ’x’: 3, ’mystep’: 3

False

True

3 2

1.5

l

’conflict’: False, ’distant2’: 3, ’urgent’: True, ’mydhistory’: [0, 3, 2, 1], ’home’: ’y’: 2, ’x’: 3, ’mycurrent’: 2, ’dhistory’: [3, 3, 3, 2, 1, 1, 1, 1, 1, 1, 1], ’mystep’: 4

右转，这是合理的，因为最近边界点在（3，2）

False

True

1 0

None

’dhistory’: [3, 3, 3, 2, 1, 1, 1, 1, 1, 1, 1], ’mystep’: 1, ’home’: ’x’: None, ’y’: None, ’mycurrent’: 0, ’urgent’: False, ’distant2’: 0, ’conflict’: False, ’mydhistory’: [1]

回到领地，全部清空后回到第一步。

综上，这是一个圈地全过程，前提是对手足够远（虽然在测试时做不到真正的足够远），本算法能完成圈地的任务，但最终的使用还需要和其他部分相互融合。

下面是几个特殊情况处理。

(图片名按顺序代表步骤顺序)

首先是撞带情况构造图见“撞带”图

l

’distant2’: 3, ’urgent’: False, ’home’: ’x’: None, ’y’: None, ’dhistory’: [3, 3, 3, 2, 1, 1, 1, 1, 1, 1, 1], ’mystep’: 4, ’conflict’: True, ’mydhistory’: [1, 0, 3, 2, 1], ’mycurrent’: 2

其次是撞墙，见‘撞墙’图

l

’distant2’: 3, ’dhistory’: [3, 3, 3, 2, 1, 1, 1, 1, 1, 1, 1], ’mycurrent’: 2, ’home’: ’y’: None, ’x’: None, ’conflict’: True, ’mydhistory’: [3, 0, 1], ’urgent’: False, ’mystep’: 2

遇到墙直接转到下一步，并更新 mydhistory

本算法未参加热身赛。

3.2 结果分析

在实战对弈过程中，本算法的三种子算法均有出现，也都进行了优秀的对抗。运行时间上本算法基本上符合预期，未出现超时现象，主要的运行时间开销发生在开始的调用数据以及圈地或者进攻算法的环节。

4 实习过程总结

4.1 分工与合作

4.1.1 分工

经过近一周的准备与思考，本小组组员对于问题的解决各自有了一定的思路。于是在当周周五晚进行了第一次讨论。在讨论中，组员们畅所欲言，提出了各有特色的思路，汇集成一个总体的方针（总思想前面在流程图部分已经介绍过了，这里不再赘述）。在这个方针的基础上，本组以“思路提出者为用到本人思路的部分算法编程”为基本原则进行了任务的分工：组长杨一龙负责防守反击的部分，即先回领地，后出击对手的代码；吴政霖负

责与对手正面交锋的必胜代码部分，策略即不断通过逼近压制使对手无路可走；侯远樵负责编写在对手离本方距离很远时本组圈地效率最高的正方形圈地法（至于为什么选择正方形，源于数学上的考虑，当然同等周长下圆的圈地效率是最高的，单元的实现太过复杂而且对于横平竖直的场地也不很合适）。姜晓飞主要负责基础代码，即接收到每场的游戏数据后进行一些预处理，将数据整理打包供前三位组员的函数调用。每位同学在编写完毕后将独立进行 debug，

在代码实现结束后，杨一龙同学和姜晓飞同学不辞辛苦，承担起了合并代码的任务，其内容主要包括变量名的统一，衔接代码也就是函数调用条件的补充，以及最终的 debug 过程。最后本实验报告的各函数说明部分也由四人分别完成并组合。

4.1.2 合作

思路汇总

如上所述，第一次讨论本组组员将思路汇总与讨论取舍，比如讨论进攻算法中的奇偶性对决策优势与劣势的影响，讨论进攻与防守的关系以及可能的权值计算问题。虽然这次讨论中的一些想法最后并未实现，但大体的思路就是在本次思路汇总中确定的。

2) 代码汇总

编写完各自代码后，本小组进行了第二次讨论交流。在交流中，每位组员介绍自己代码的具体思路，让小组成员大致了解各自代码和其他人代码的相对关系，比如哪里要调用别人的代码，哪里有重复，哪里可以互补，同一个算法谁写的更完整等等。第三次讨论中，组员们一面继续实现数据的预处理，一面将进攻，防守，反击三部分内容合在一起，并加上必要的调用条件

3) 定期交流

在编写代码的过程中，小组组员随时保持交流，适时调整战略，并尽量根据彼此的情况调整自己的代码，尽可能做到代码中一些关键变量的统一。经过平时的交流，小组改动了原先思路中难以实现的部分，比如在最大圈地时兼顾反击，在反击之后自动恢复原先的圈地战略。如果产生好想法组员会立即在微信群中与大家分享，和大家一起评判，这样不断在各自的代码基础上实现微调。

4.2 经验与教训

【1】经验

本组在开发时做的不错的地方在于及时进行了任务的明确分工，这使得组员在编写代码时能够集中精力长期地解决一个问题，并留出了足够长的时间进行反思和调整。除此之外，本组能随时交流，互相建言献策，为完成任务扫除了很多障碍，这也是值得延续的。

他山之石，可以攻玉。其他小组的经验也可以指导我们以后的合作。我们学到了可有另一种开发模式，集美名组员先自行开发一个简单的完整代码，之后互相比赛，选出胜率最高的代码。之后的工作就围绕这个代码的修改提高展开，这样做或许更有集中性和完整性。这为以后可能的合作提供一种模式的选择。

【2】教训

总比赛结果来看，本小组发挥并不好，在实际比赛时遇到了一些各自测试时没有遇到的新问题，可以说这次比较失败的比赛使我们吸收了一定的经验教训。

思路方面，本组在考虑最佳算法时更多地关注了一些静态的情况，对于对手的策略没有进行更加细致的关注，在功能的划分上有些绝对。在考虑综合决策时没有对于更加智能化的算法进行探索，在数据结构方面的构想也较为简单。另外，本组没有进行更高级更智能算法的查找和学习，没有利用课程外的或者课程不做重点的内容帮助自己形成思路。

编写代码方面，本组虽然在组员之间进行了及时的沟通，但实际合并代码略显仓促。平时在编写的进程中就应该及时合并代码，这样就可以避免最后出现的各组员算法不兼容，过于独立的现象与问题。在实际调用时出现各功能的绝对先后性，使得整体性打折扣。

调代码方面，组员在测试自己的代码时有些情况没有考虑到，使得最终决策出现与实际情况的偏差，这与代码测试的难度大有一定的关系，因为每个程序都只是一部分，在没有基础数据处理的情况下很难独立对战随机 AI，因此只能自己手写棋盘，构造情况，而情况的构造就会有误差，出于种种原因进行了一些无效的测试，不完整的测试。另一方面作为算法初学者，我们本身编写代码的经验不够丰富，仅凭自己有限的知识考虑各种情况，就会出现一定的失败风险。今后我们会加倍努力，使自己有深厚的编程功底，进而有时间有精力探索更高级的代码。

汇总方面，本组在交流代码的过程中虽然彼此了解了代码的思路，但有时面对队友的代码还是会一头雾水，对于怎样改进自己的部分以形成互补不够清晰。汇总的时间比较短，

而且可能没有达到最大的参与度，导致合并时对于一些技术问题没有得到及时的交流。

4.3 建议与设想

对于本类型编程课题的建议

问题与要求方面：这个课题本身很具有研究价值，可以培养编程学习者的创新能力，巩固编程基础，提供实际运用编程知识的机会。但我们建议减少要求的改动，比如基础对战平台的调整，因为有时大量的时间会花费在根据新要求，新的运行环境修改变量名，甚至修改原先思路，对策赶不上变化，会造成一些时间的浪费。此外我们希望除了基础平台外，切实增进队与队之间的交流，增加一些思路或者实现方法上的帮助。

评分标准方面，本小组认为具体记分可以采用更加多元化的方法，即使不算为分数，

也可以评定比如“最佳创意奖”“最佳 KO 奖”“最佳防御奖”“最佳圈地效率奖”等等。这样可以发现一些中间被淘汰算法的优点。毕竟有一些算法可能只是遇到了过于强大的对手，而让一些优势不能充分发挥出来。

5 致谢

感谢陈天翔同学对于本组一些基本技术问题的回答与帮助。感谢陈斌老师以及各位助教一学期以来的帮助。

6 参考文献

Problem Solving with Algorithms and Data Structures

廖雪峰的 Python 教程

第二十一章　F17_Uniform 报告

余圣杰 * 李南鸽方景行赵佳迪

摘要：Paper.io 为不完全信息动态决策游戏，我们以设计战术的方式，结合博弈论与计算权重的思想设计了我们的算法。经过细致解读 paper.io 的原始代码并研究其规则，基于队列等数据结构，我们设计了此次比赛的算法。在我们经过完善的最终版本中，总的体系是根据场上实时情况，结合深度优先搜索，在充分保险的情况下做出最优决策，然后执行。

关键字：广度优先搜索队列 list dict

1 算法思想

1.1 总体思路

我们小组总体上采用比较保守的思路，主要是圈地，不是以攻击为目标的玩法。

首先，确保我方的每一个决策都不会“自杀”。即自己的纸卷不会撞到自己的纸带，且自己的纸卷不会撞墙。

在进行圈地时，对于路径的决策，采取一定不会被对方杀掉的情况下的最优解。如果我方纸卷处于自己领地的边界处，且与对方的纸卷距离过近，则在我方领地边缘徘徊，以确保不会被对方杀死。当我方可能有被杀危险时，立即找到最近的路线逃回自己领地。

只有当我方有极大把握（具体条件将在后文给出）杀掉对方时，才开始进攻对手，其余情况不会主动出击。

具体地，我方纸卷所处的状态分为 8 类（8 种 state）：

0：从内部出去/1：出去后第一条边/2：出去后第二条边/3：第三条边/4：第四条边/5：逃回家/6：杀/7：沿边界走/8：其他

每一轮 play 函数开始后，首先保存下可能用到的数据，对场上情况进行更新，然后开始利用决策函数选择接下来的动作。

决策的开头，若我方处于某些特定状态，判断一下是否需要先改变状态。

如果我方处于状态 0/1/2/3，则说明我方正在圈地中，则进行一系列判断，判断是否进行以及是否进入下一状态。

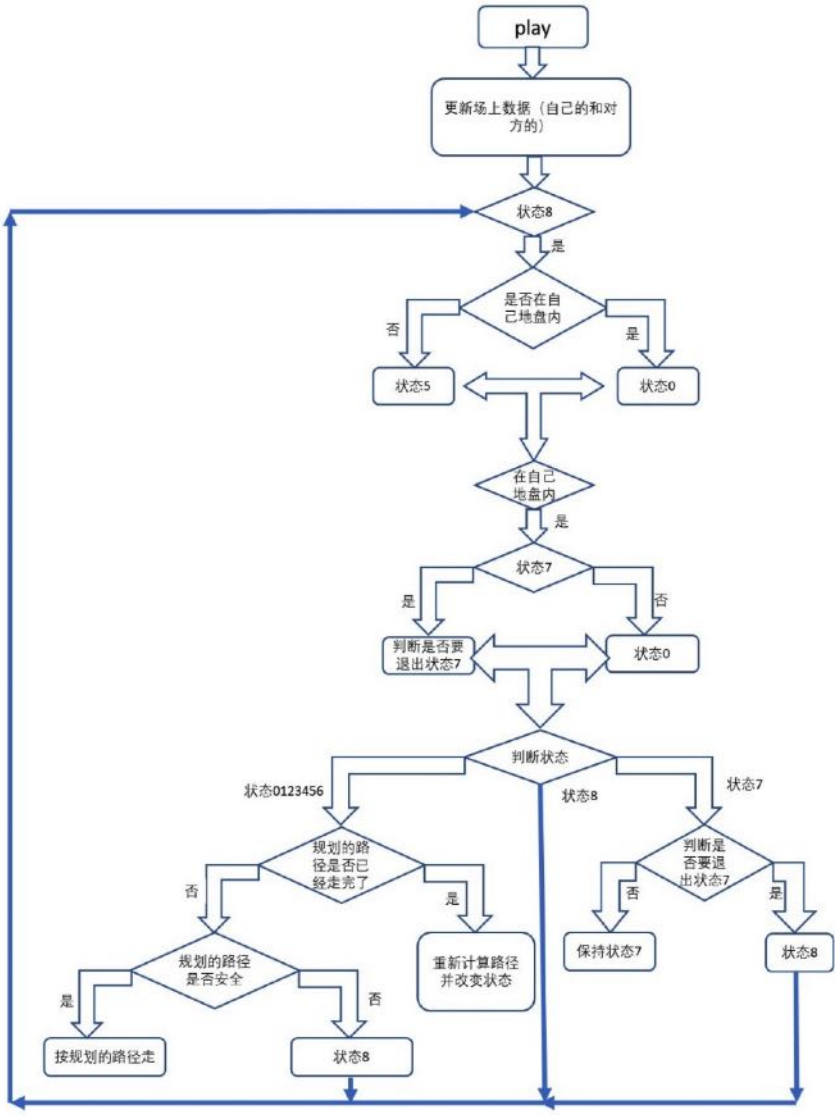
如果我方处于 4/5/6，则分别按照计算的路径行进。

以上状态中，如果情况生变（如发现路径不安全等）则进入状态 8。

如果我方处于状态 7，则沿边界徘徊。

如果我方处于状态 8，则需要新一轮的决策。

1.2 算法流程图



1.3 算法复杂度

整个决策中只有找路径的 BFS 需要 O(n^2) 时间，而决策是 O(1) 或 O(n) 的，所以算法复杂度是 O(n^2) 的。

2 程序代码说明

2.1 数据结构说明

在 find_min_dist 函数里的宽度优先算法里，使用了队列数据结构。

2.2 函数说明

1.play 函数

首先介绍主体函数 play，在 play 中用到的其他函数暂时仅说明其达到的效果，而具体实现方式将分列各个函数，逐一说明。

```
20  def play(stat, storage):
21
22      from random import random, shuffle
23      now = stat['now']
24      fields = now['fields']
25      bands = now['bands']
26      me = now['me']
27      enemy = now['enemy']
28      _x = me['x']
29      _y = me['y']
30      _d = me['direction']
```

保存常用数据。

```
526     update()
527     return quick_choice()
```

先 update（）更新各项数据，然后进行决策。

2.find_min_dist 函数

```
32  def find_min_dist(bb, x=me['x'], y=me['y'], d=me['direction'], aim=[], \
33                    elude=[], find_path=False):
34      if bb[x][y] in aim:
35          return [] if find_path else 0
36      que = [(x, y, d)]
37      front = 0
38      elude.append(-1)
39      bb[x][y] = -1
40      flag = False
41      while not flag and front!=len(que):
42          xx, yy, dd = que[front]
43          for i in (0, -1, 1):
44              nxtd = (dd+i)%4
45              nxtx = xx+storage['direction'][nxtd][0]
46              nxty = yy+storage['direction'][nxtd][1]
47              if storage['in_graph'](nxtx, nxty) and (bb[nxtx][nxty] not\
48                                                      in elude):
```

```
                que.append((nxtx, nxty, nxtd))
                if bb[nxtx][nxty] in aim:
                    flag = True
                    break
                bb[nxtx][nxty]=-1
            front+=1
        path = []
        tmp = len(que)-1
        xx, yy, dd = que[-1]
        while tmp>=0:
            if que[tmp][0] == xx and que[tmp][1] == yy:
                path.append(que[tmp])
                dd = que[tmp][2]
                xx-=storage['direction'][dd][0]
                yy-=storage['direction'][dd][1]
            tmp-=1
        nxtd = path[-1][2]
        choose = []
        for tmp in range(len(path)-1, 0, -1):
            dd = nxtd
            nxtd = path[tmp-1][2]
            for i in (-1, 0, 1):
                if (dd+i)%4==nxtd:
                    choose.append(storage['equal'][i])
                    break
        return choose if find_path else len(choose)
```

此函数用于寻找最短路线，使用宽度优先搜索算法。地图 bb 是一个二位列表，大小和游戏区域相同。x 是开始点的 x 坐标，未指定时默认自己当前 x 坐标；y 是开始点的 y 坐标，未指定时默认自己当前 y 坐标；d 是开始时的方向，未指定时默认自己当前方向。aim 是一个列表，只要在地图 bb 上找到 aim 内的元素，就算搜索成功，默认是空列表。elude 是一个列表，代表在地图 bb 上不能走过的区域，默认是空列表。find_path 是选项，为 True 时返回走的路径的选择，为 False 时只需返回最短路的长度。

3. find_path_kill 函数

```
def find_path_kill(find_path=True):
    board = []
    for row in bands:
        board.append(list(row))
    return find_min_dist(board, me['x'], me['y'], me['direction'], \
            aim=[enemy['id']], elude=[me['id']], find_path=find_path)
```

此函数用于找到我杀敌人的最短路径，需要注意路上不能碰到自己的纸带

4. find_path_in 函数

```
    def find_path_in(find_path=True):
        def find_path_in_helper():
            a = 0
            b = storage['dist'](order[0][0], order[0][1], enemy['x'], enemy['y'])
            if storage['kill_me']!=-1 and storage['kill_me']\
                    <=len(storage['escape_me'])+2:
                for i in range(1, len(order)):
                    if b<storage['dist'](order[i][0], order[i][1], enemy['x'],\
                                         enemy['y']):
                        a = i
                        b = storage['dist'](order[i][0], order[i][0], \
                                            enemy['x'], enemy['y'])
            else:
                a = 0
            return a
        board = []
        for row in bands:
            board.append(list(row))
        if storage['band_enemy'] == [] or storage['size'][enemy['id']]>\
                storage['size'][me['id']]:
            board[enemy['x']][enemy['y']] = me['id']
        board[me['x']][me['y']] = -1
        que = [(me['x'], me['y'], me['direction'])]
        front = 0
        flag = True
        while flag and front!=len(que):
            x, y, d = que[front]
            order = []
            for i in (0, -1, 1):
                nxtd = (d+i)%4
                nxtx = x+storage['direction'][nxtd][0]
                nxty = y+storage['direction'][nxtd][1]
                if storage['in_graph'](nxtx, nxty) and board[nxtx][nxty] \
                        not in (-1, me['id']):
                    order.append((nxtx, nxty, nxtd))
                    board[nxtx][nxty] = -1
            while order!=[]:
                p = find_path_in_helper()
                que.append(order[p])
                if fields[order[p][0]][order[p][1]] == me['id']:
                    flag = False
                    break
                order.pop(p)
            front+=1
        path = []
        tmp = len(que)-1
        xx, yy, dd = que[-1]
        while tmp>=0:
            if que[tmp][0] == xx and que[tmp][1] == yy:
                path.append(que[tmp])
                dd = que[tmp][2]
                xx-=storage['direction'][dd][0]
                yy-=storage['direction'][dd][1]
            tmp-=1
```

```
136         nxtd = path[-1][2]
137         choose = []
138         for tmp in range(len(path)-1, 0, -1):
139             dd = nxtd
140             nxtd = path[tmp-1][2]
141             for i in (-1, 0, 1):
142                 if (dd+i)%4==nxtd:
143                     choose.append(storage['equal'][i])
144                     break
145         return choose if find_path else len(choose)
146 
147     def find_path_out(find_path=True):
148         board = []
149         for row in fields:
150             board.append(list(row))
151         return find_min_dist(board, me['x'], me['y'], me['direction'], aim=[None,
```

此函数用于计算我逃跑（即回到自己地盘）的最短路。由于逃跑的时候尽量不能让对方碰到自己，所以要在所有的最短路中选择距离敌方纸卷最远的路径。方法是将每一个选择的路径按距离敌方纸卷的最近距离从大到小排序。

5. find_path_out 函数

```
148     def find_path_out(find_path=True):
149         board = []
150         for row in fields:
151             board.append(list(row))
152         return find_min_dist(board, me['x'], me['y'], me['direction'], \
153                     aim=[None, enemy['id']], elude=[], find_path=find_path)
```

此函数用于计算我出门的最短路，只要是 None 和敌方的覆盖都能算作“出门”

6. find_path_escape 函数

```
155     def find_path_escape(find_path=False):
156         board = []
157         for row in fields:
158             board.append(list(row))
159         for x, y in storage['band_enemy']:
160             board[x][y] = enemy['id']+2
161         return find_min_dist(board, x=enemy['x'], y=enemy['y'], d=enemy\
162 ['direction'], aim=[enemy['id']], elude=[enemy['id']+2], find_path=find_path)
```

此函数用于计算敌人逃跑的最短路（假设敌人够聪明不会碰到自己的纸带）。

7. judge_kill_escape 函数

```
164     def judge_kill_escape():
165         if storage['kill_enemy']!=-1 and len(storage['kill_enemy'])\
166                 <=len(storage['escape_enemy']):
167             storage['state'] = 6
168             storage['path'] = storage['kill_enemy']
169             storage['begin_path'] = now['turnleft'][me['id']-1]
170         elif storage['kill_me']!=-1 and len(storage['escape_me'])>=\
171                 storage['kill_me']-4:
172             storage['state'] = 5
173             storage['path'] = storage['escape_me']
174             storage['begin_path'] = now['turnleft'][me['id']-1]
175         elif storage['state'] == 5:
176             storage['state'] = 4
177             storage['escape_me'] = find_path_in(find_path=True)
178             storage['begin_path'] = now['turnleft'][me['id']-1]
```

此函数用于判断是否要杀敌人或者逃跑。只要杀敌人的距离小于等于敌人逃跑的距离，就可以选择杀敌人；只要我逃跑距离和敌人杀我距离相比太少就要选择逃跑；不需要逃跑且原来状态是逃跑的，说明已经逃离追杀，可以从容地回去。

8. update 函数

```
180     def update():
181         if bands[me['x']][me['y']] == me['id']:
182             storage['band_me'].append((me['x'], me['y']))
183             storage['kill_me'] = storage['dist'](me['x'], me['y'], \
184                                                  enemy['x'], enemy['y'])
185             for x, y in storage['band_me']:
186                 storage['kill_me'] = min(storage['kill_me'], \
187                             storage['dist'](x, y, enemy['x'], enemy['y']))
188             storage['escape_me'] = find_path_in(find_path=True)
189         else:
190             if storage['band_me'] != []:
191                 tot = 0
192                 for row in fields:
193                     tot+=row.count(me['id'])
194                 storage['size'][me['id']] = tot
195             storage['band_me'] = []
196             storage['kill_me'] = -1
197             storage['escape_me'] = []
198         if bands[enemy['x']][enemy['y']] == enemy['id']:
199             storage['band_enemy'].append((enemy['x'], enemy['y']))
200             storage['kill_enemy'] = find_path_kill(find_path=True)
201             storage['escape_enemy'] = find_path_escape(find_path=True)
202         else:
203             if storage['band_enemy'] != []:
204                 tot = 0
205                 for row in fields:
206                     tot += row.count(enemy['id'])
207                 storage['size'][enemy['id']] = tot
208             storage['band_enemy'] = []
209             storage['kill_enemy'] = -1
210             storage['escape_enemy'] = []
211         judge_kill_escape()
```

此函数用于更新场上的状态。

如果当前在圈地，那么我的纸带，杀我的最短距离，我逃跑的最短路径都需要更新。

其中计算各个路径的时间复杂度最高，因为调用的是 find_min_dist 函数，使用的是 BFS 算法，时间复杂度为 O(n^2)。

如果当前在自家地盘，则可能需要更新的有自家面积。

对敌人数据的更新和上面的类似。

9. is_safe 函数

```
213     def is_safe(c):
214         nxtd = (me['direction']+storage['equal'][c])%4
215         nxtx = me['x']+storage['direction'][nxtd][0]
216         nxty = me['y']+storage['direction'][nxtd][1]
217         if nxtx == enemy['x'] and nxty == enemy['y']:
218             return fields[nxtx][nxty]!=enemy['id']
219         else:
220             return storage['in_graph'](nxtx, nxty) and bands[nxtx][nxty]!=\
221                    me['id']
```

此函数用于判断这个选择是否安全。“安全”的选择需满足条件有：不在敌方区域内碰到敌方纸卷、不出界、不碰到自己纸带。

10. is_border

```
233     def select_3():
234         c = storage['path'][0]
235         nxtd = (me['direction']+storage['equal'][c])%4
236         l = storage['kill_me']//5-1
237         if l<=2:
238             storage['path']=storage['escape_me']
239             storage['begin_path']=now['turnleft'][me['id']-1]
240             storage['state']=5
241         nxtx = me['x']+storage['direction'][nxtd][0]
242         nxty = me['y']+storage['direction'][nxtd][1]
243         j=1
```

此函数用于判断点 (x, y) 是不是自家区域的边界。

11. select_3 函数

```
233     def select_3():
234         c = storage['path'][0]
235         nxtd = (me['direction']+storage['equal'][c])%4
236         l = storage['kill_me']//5-1
237         if l<=2:
238             storage['path']=storage['escape_me']
239             storage['begin_path']=now['turnleft'][me['id']-1]
240             storage['state']=5
241         nxtx = me['x']+storage['direction'][nxtd][0]
242         nxty = me['y']+storage['direction'][nxtd][1]
243         j=1
```

```
244         while j<l:
245             nxtx+=storage['direction'][nxtd][0]
246             nxty+=storage['direction'][nxtd][1]
247             if storage['in_graph'](nxtx, nxty) and fields[nxtx][nxty]!=me['id']:
248                 j+=1
249             else:
250                 break
251         storage['begin_path'] = now['turnleft'][me['id']-1]
252         if j>len(storage['escape_me']) and storage['escape_me'][0] == 'f':
253             storage['state'] = 5
254         else:
255             storage['path'] = [c]+['f']*(j-1)
```

选择第三条边的长度，单独写一个函数是因为代码要复用。

12. quick_choice 函数

```
257     def quick_choice():
258         if storage['state']==8:
259             if fields[me['x']][me['y']]==me['id']:
260                 storage['state'] = 0
261                 storage['path'] = find_path_out(find_path=True)
262                 storage['begin_path'] = now['turnleft'][me['id']-1]
263                 try:
264                     storage['path'].pop()
265                 except IndexError:
266                     pass
267                 storage['begin_path'] = now['turnleft'][me['id']-1]
268             else:
269                 storage['state'] = 5
270                 storage['path'] = storage['escape_me']
271                 storage['begin_path'] = now['turnleft'][me['id']-1]
```

当本身处于状态 8，如果在自己地盘内就进入状态 0，否则进入状态 5。

```
272         if fields[me['x']][me['y']]==me['id']:
273             if storage['state'] == 7:
274                 if storage['dist'](me['x'], me['y'], enemy['x'], enemy['y'])>10:
275                     storage['state']=0
276                     storage['path'] = find_path_out(find_path=True)
277                     storage['begin_path'] = now['turnleft'][me['id']-1]
278                     try:
279                         storage['path'].pop()
280                     except IndexError:
281                         pass
282             else:
283                 storage['state']=0
284                 storage['path']=find_path_out(find_path=True)
285                 storage['path'].pop()
286                 storage['begin_path']=now['turnleft'][me['id']-1]
287         else:
288             if len(storage['escape_me']) >= now['turnleft'][me['id']-1]-5:
289                 storage['state'] = 5
290                 storage['begin_path'] = now['turnleft'][me['id']-1]
```

如果在自家地盘内，且原来状态是 7（也就是沿着边界走），则判断是否要退出状态，敌我距离足够大即可以退出状态 7 进入状态 0 也就是要出去圈地了；原来状态不是 7 则直

接进入状态 0 因为出击不会有危险。选择出去的最短路，同时把最后一步删掉因为场上情况瞬息万变所以到了边界还不一定真的出去。如果不在自家地盘内，如果回去的路径长度和余下的回合数相当，那么就回去保证把这块地圈了不浪费，否则就不做改动，后面再进行决策。

下面是对应每个状态的行走方案：

```
339         if storage['state']==1:
340             try:
341                 c = storage['path'][storage['begin_path']-now['turnleft']\
342                     [me['id']-1]]
343                 if is_safe(c):
344                     return c
345                 else:
```

```
318                 else:
319                     storage['path'] = []
320                     for i in (0, -1, 1):
321                         nxtd = (me['direction']+i)%4
322                         nxtx = me['x']+storage['direction'][nxtd][0]
323                         nxty = me['y']+storage['direction'][nxtd][1]
324                         if storage['in_graph'](nxtx, nxty) and fields[nxtx]\
325                                 [nxty]!=me['id']:
326                             storage['path'].append(storage['equal'][i])
327                             break
328                     j=1
329                     while j<l:
330                         nxtx+=storage['direction'][nxtd][0]
331                         nxty+=storage['direction'][nxtd][1]
332                         if storage['in_graph'](nxtx, nxty) and fields[nxtx]\
333                                 [nxty]!=me['id']:
334                             storage['path'].append('f')
335                             j+=1
336                         else:
337                             break
338                 storage['begin_path']=now['turnleft'][me['id']-1]
```

如果处于状态 0，如果规划的路径尚未走完，那么返回已经计算好的选择，并判断该选择是否安全：若安全则进行，不安全则进入状态 8；如果规划的路径已经走完，需要开始圈地进入状态 1，并需要计算路径：计算圈地第一条边走多远，即找对方一定不能杀了我的最大值。

```
339         if storage['state']==1:
340             try:
341                 c = storage['path'][storage['begin_path']-now['turnleft']\
342                     [me['id']-1]]
343                 if is_safe(c):
344                     return c
345                 else:
```

```
            else:
                storage['state'] = 8
        except IndexError:
            storage['state'] = 2
            l1 = len(storage['path'])
            storage['path'] = []
            l = storage['kill_me']//4-1
            if l<=2:
                storage['path']=storage['escape_me']
                storage['begin_path']=now['turnleft'][me['id']-1]
                storage['state']=5
            board = []
            for row in fields:
                board.append(list(row))
            choseleft = False
            choseright = False
            nxtd = (me['direction']-1)%4
            nxtx = me['x']+storage['direction'][nxtd][0]
            nxty = me['y']+storage['direction'][nxtd][1]
            if storage['in_graph'](nxtx, nxty) and board[nxtx][nxty]!=\
                    me['id']:
                for x, y in storage['band_me']:
                    board[x][y]=me['id']+2
                tmp=board[nxtx][nxty]
                board[nxtx][nxty]=me['id']+2
                path = find_min_dist(board, x=me['x'], y=me['y'], d=\
me['direction'], aim=[me['id']], elude=[me['id']+2], find_path=True)
                if len(path)>=storage['kill_me']:
                    choseright = True
                board[nxtx][nxty]=tmp
            nxtd = (me['direction']+1)%4
            nxtx = me['x']+storage['direction'][nxtd][0]
            nxty = me['y']+storage['direction'][nxtd][1]
            if storage['in_graph'](nxtx, nxty):
                for x, y in storage['band_me']:
                    board[x][y]=me['id']+2
                tmp=board[nxtx][nxty]
                board[nxtx][nxty]=me['id']+2
                path = find_min_dist(board, x=me['x'], y=me['y'], d=\
me['direction'], aim=[me['id']], elude=[me['id']+2], find_path=True)
                if len(path)>=storage['kill_me'] and board[nxtx][nxty]!=\
                        me['id']:
                    choseleft = True
                    return quick_choice()
                board[nxtx][nxty]=tmp
            if not choseleft and not choseright:
                l2, l3 = 0, 0
                nxtd = (me['direction']-1)%4
                nxtx = me['x']
                nxty = me['y']
                while l2<l:
                    nxtx+=storage['direction'][nxtd][0]
                    nxty+=storage['direction'][nxtd][1]
                    if storage['in_graph'](nxtx, nxty) and \
                            fields[nxtx][nxty]!=me['id']:
                        l2+=1
                    else:
                        break
                if storage['in_graph'](nxtx, nxty):
                    L2 = l1+l2
                else:
```

```
405                 else:
406                     L2 = 2*l1+l2
407                 nxtd = (me['direction']+1)%4
408                 nxtx = me['x']
409                 nxty = me['y']
410                 while l3<l:
411                     nxtx+=storage['direction'][nxtd][0]
412                     nxty+=storage['direction'][nxtd][1]
413                     if storage['in_graph'](nxtx, nxty) and \
414                             fields[nxtx][nxty]!=me['id']:
415                         l3+=1
416                     else:
417                         break
418                 if storage['in_graph'](nxtx, nxty):
419                     L3 = l1+l3
420                 else:
421                     L3 = 2*l1+l3
422                 if l1*l2/L2>l1*l3/L3:
423                     storage['path'] = ['l'] + ['f']*(l2-1)
424                 else:
425                     storage['path'] = ['r'] + ['f']*(l3-1)
426             elif choseleft and choseright:
427                 storage['path'] = storage['escape_me']
428                 storage['state'] = now['turnleft'][me['id']-1]
429             elif choseleft:
430                 storage['path'] = ['l'] + ['f']*(l2-1)
431             else:
432                 storage['path'] = ['r'] + ['f']*(l3-1)
433             storage['begin_path'] = now['turnleft'][me['id']-1]
```

当处于状态 1 时，如果规划的路径尚未走完，那么返回已经计算好的选择，并判断该选择是否安全：若安全则进行，不安全则进入状态 8；如果规划的路径已经走完，则进入状态 2，并需要计算圈地第二条边走多远。首先需要决定往哪个方向走，我按照圈地效率来决定：圈地效率定义为圈地面积和圈地所用步数之比。分别计算向左和向右的圈地效率，取效率更高的方向。计算圈地步数时，需要考虑实际是用两条边就可以封起来，还是要三条边或者更多，会不会碰到边界等等。以及，由于不确定因素太多，边的长度的上限不会真正决定这条边，因此这个效率是估计的。而我们在实测中发现，这个方案确实可以很好的提高圈地效率。

```
434         if storage['state'] == 2:
435             try:
436                 c = storage['path'][storage['begin_path']-now['turnleft']\
437                     [me['id']-1]]
438                 if is_safe(c):
439                     return c
440                 else:
441                     storage['state'] = 8
442             except IndexError:
443                 if random()>0.2:
444                     l = storage['kill_me']//5-2
445                     nxtx, nxty = me['x'], me['y']
446                     j = 0
447                     while j<l:
448                         nxtx+=storage['direction'][me['direction']][0]
449                         nxty+=storage['direction'][me['direction']][1]
450                         if storage['in_graph'](nxtx, nxty) and fields[nxtx]\
451                                 [nxty]!=me['id']:
452                             j+=1
453                         else:
454                             break
455                     storage['path'] += ['f']*j
456                     if j>0:
457                         c = 'f'
458                         if is_safe(c):
459                             return c
460                         else:
461                             storage['state'] = 5
462                     else:
463                         storage['state'] = 3
464                         select_3()
465                 else:
466                     storage['state'] = 3
467                     select_3()
```

如果处于状态 2，如果规划的路径尚未走完，那么返回已经计算好的选择，并判断该选择是否安全：若安全则进行，不安全则进入状态 8；如果规划的路径已经走完，则计算还要不要延长圈地第二条边或者圈地的第三条边走多远。要不要延长是随机的：如果 random() 给出大于 0.5 的值并且还能走下去（就是走下去不会被对方碰到纸带），那么就继续走下去，否则就转弯进入第三条边。这里有一个参数 0.5，这是经过迭代发现效果比较好的一个选择。实际代表第二条边期望要走两段。

```
    if storage['state'] == 3:
        try:
            c = storage['path'][storage['begin_path']-now['turnleft']\
                [me['id']-1]]
            if is_safe(c):
                return c
            else:
                storage['state'] = 8
        except IndexError:
            if random()>0.2:
                l = storage['kill_me']//5-2
                j = 0
                nxtx, nxty = me['x'], me['y']
                while j<l:
                    nxtx = nxtx+storage['direction'][me['direction']][0]
                    nxty = nxty+storage['direction'][me['direction']][1]
                    if storage['in_graph'](nxtx, nxty) and fields[nxtx]\
                            [nxty]!=me['id']:
                        j+=1
                    else:
                        break
                storage['path'] += ['f']*j
                storage['begin_path'] = now['turnleft'][me['id']-1]
                if j>0:
                    c = 'f'
                    if is_safe(c):
                        return c
                    else:
                        storage['state'] = 5
                else:
                    storage['path'] = storage['escape_me']
                    storage['state'] = 5
            else:
                storage['state'] = 4
                storage['path'] = storage['escape_me']
                storage['begin_path'] = now['turnleft'][me['id']-1]
    if storage['state'] == 4:
        try:
            c = storage['escape_me'][0]
            if is_safe(c):
                return c
            else:
                storage['state'] = 8
        except IndexError:
            storage['state'] = 5
            storage['path'] = storage['escape_me']
            storage['begin_path'] = now['turnleft'][me['id']-1]
        pass
```

如果处于状态 3，如果规划的路径尚未走完，那么返回已经计算好的选择，并判断该选择是否安全：若安全则进行，不安全则进入状态 8，也就是需要重新做决策；如果规划的路径已经走完，则计算要不要延长圈地第三条边或者进入圈地最后一步。此处要不要延长也是一个随机的选择：若 random() 给出大于 0.2 的值，那么就接着走下去，否则进入最后一条边。

```
504         if storage['state'] == 4:
505             try:
506                 c = storage['escape_me'][0]
507                 if is_safe(c):
508                     return c
509                 else:
510                     storage['state'] = 8
511             except IndexError:
512                 storage['state'] = 5
513                 storage['path'] = storage['escape_me']
514                 storage['begin_path'] = now['turnleft'][me['id']-1]
515             pass
```

如果处于状态 4，即圈地的最后一步，则走最短路回去即可。如果情况有变，如原来规划好的路径被圈走了，重新规划最短路回自己的区域。

```
516         if storage['state'] == 5:
517             try:
518                 c = storage['escape_me'][0]
519                 if is_safe(c):
520                     return c
521                 else:
522                     storage['state'] = 8
523             except IndexError:
524                 storage['escape_me'] = find_path_in(find_path=True)
525                 storage['begin_path'] = now['turnleft'][me['id']-1]
526                 c = storage['escape_me'][0]
527                 if is_safe(c):
528                     return c
529                 else:
530                     storage['state'] = 8
```

若处于状态 5，则用距离敌方纸卷最远的最短路回自己区域。

```
532         if storage['state'] == 6:
533             try:
534                 c = storage['path'][storage['begin_path']-now['turnleft']\
535                     [me['id']-1]]
536                 if is_safe(c):
537                     return c
538                 else:
539                     storage['state'] = 8
540             except IndexError:
541                 storage['path'] = find_path_kill(find_path=True)
542                 storage['begin_path'] = now['turnleft'][me['id']-1]
```

如果处于状态 6，则可以杀敌，直接按照选好的能碰到对方纸带的最短路杀过去即可。

```
544         if storage['state'] == 7:
545             if fields[me['x']][me['y']] == me['id']:
546                 choices = [-1, 0, 1]
547                 shuffle(choices)
548                 for i in choices:
549                     nxtd = (me['direction']+i)%4
550                     nxtx = me['x']+storage['direction'][nxtd][0]
551                     nxty = me['y']+storage['direction'][nxtd][1]
552                     if storage['in_graph'](nxtx, nxty) and \
553                             is_border(nxtx, nxty):
554                         return storage['equal'][i]
555             else:
556                 storage['state'] = 8
```

如果处于状态 7，沿着区域的边界走不出头（当自己和敌方距离过小时使用）。

```
558         if storage['state'] == 8:
559             return quick_choice()
```

如果一轮判断和改变状态等操作结束后由于种种原因需要重新决策，就递归再做一次选择

2.3 程序限制

1. 当对方圈一块地成功，可能会使本来规划好的能够回到自己区域的路变成不可行的，可行路径突然变长了很多，导致容易在此过程中被对方杀掉。

2. 当敌我距离过近且我方在自己地盘内时，我方会在边界游走。但是当我方地盘过小而敌方一直在旁边时，我方永远无法出自己的地盘。

3 实验结果

3.1 测试数据

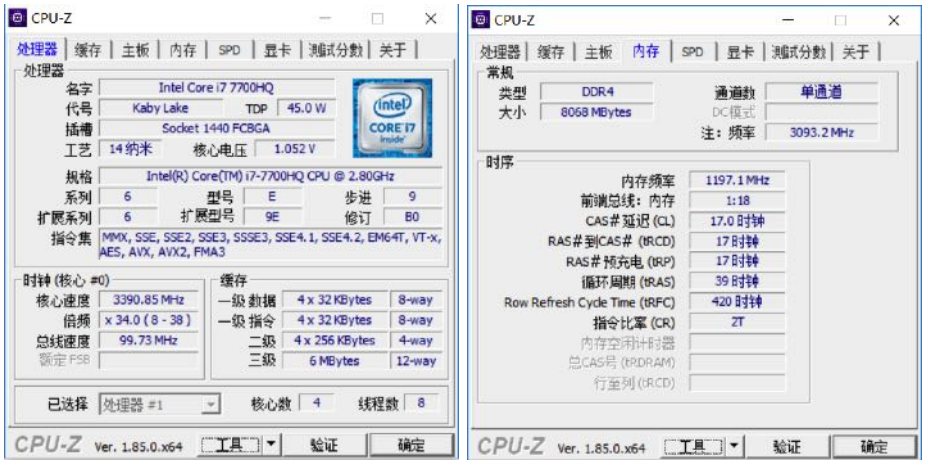
实验环境说明：

CPU：Intel Core-i7 7700HQ，主频 2.8GHz，睿频 3.7GHz，功耗 45W

内存：单通道 8GB，DDR4 内存，频率 2400MHz。

操作系统：Windows10，版本 1803。

Python 版本：3.6.5



测试方法：

1. 把自己的代码作为双方进行比拼，观察设计的功能是否能够实现。

2. 参加热身赛。

热身赛结果：总计 3 胜 32 负，积 9 分排名第 34。

对战方	007_1	Echo_1	Echo_2	Echo_3	India_1
结果	负	负	负	负	负
对战方	KizunaAI_4	KizunaAI_5	Lima_1	November_1	Quebec_1
结果	负	负	胜	负	负
对战方	Whiskey_0	Whiskey_1	alpha_1	alpha_2	bravo_1
结果	负	负	负	负	负
对战方	bravo_2	bravo_3	charlie_1	foxtrot_1	foxtrot_2
结果	负	负	负	负	胜
对战方	foxtrot_3	foxtrot_4	foxtrot_5	golf_1	juliet_1
结果	负	负	负	负	负
对战方	juliet_2	menhera_1	menhera_2	menhera_3	menhera_4
结果	负	负	负	负	负
对战方	oscar_1	papa_2	victor_1	victor_2	x-ray_1
结果	胜	负	负	负	负

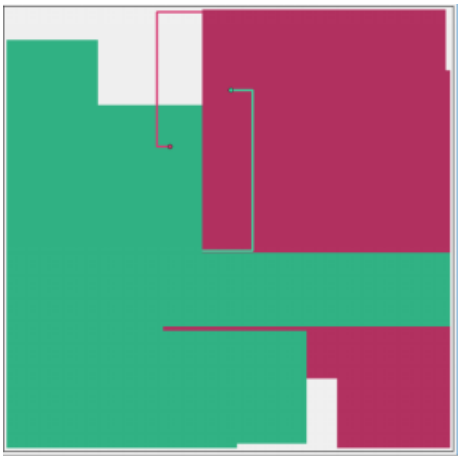
3.2 结果分析

从所有测试局中挑选两局进行分析：

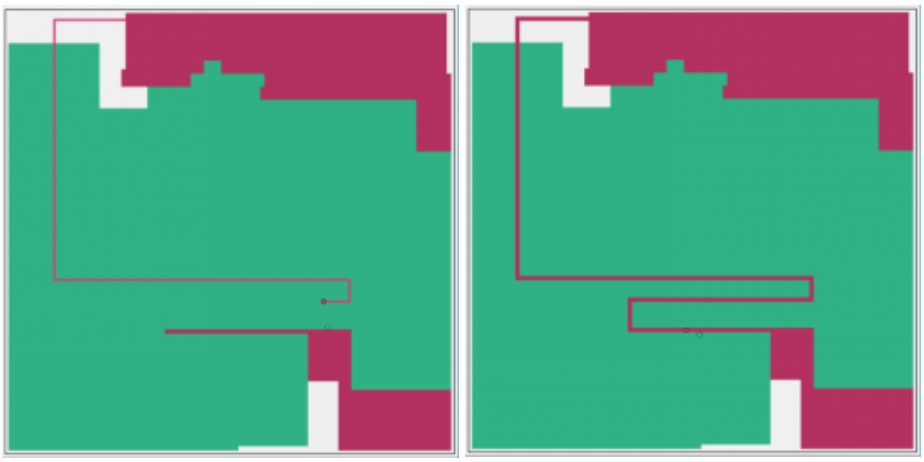
第一局：

后手方发现有追杀先手方成功的可能，正在追杀先手方，先手方在沿最近的路径逃回地盘。

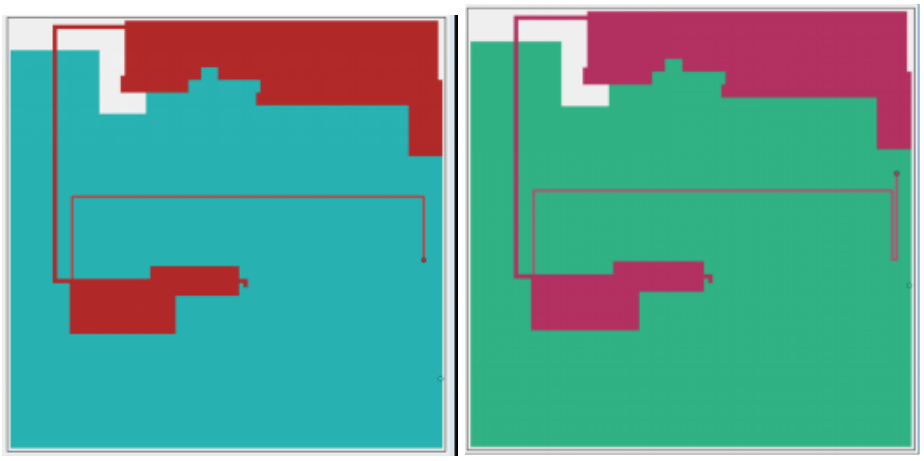
后手方和先手方都正处于自己地盘的边界，并且与对手距离过近，此时双方均沿着边界移动，避免走出自己地盘后有被杀风险。



双方互相在抢占对方的地盘。

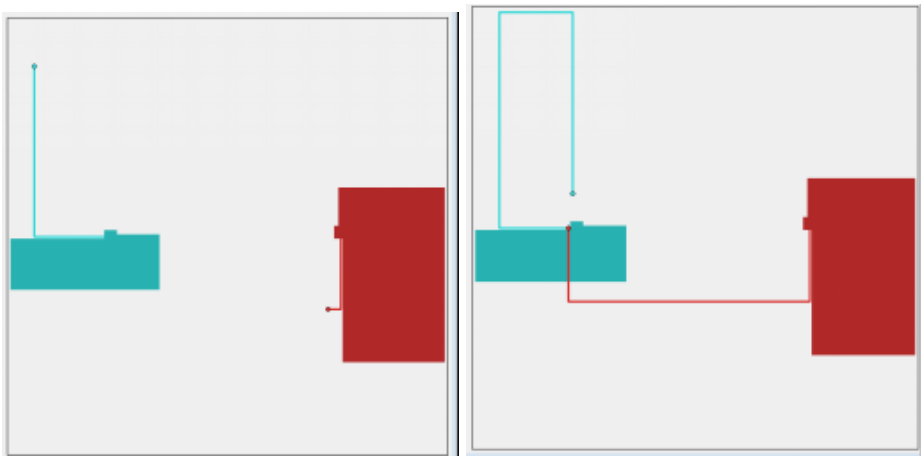


由于先手方拦腰截断了后手方的地盘，导致后手方圈地失败，被杀的风险也增大了。



后手本来打算拓宽可能的圈地面，发现先手开始追杀立即掉头沿最近路线返回自己地盘。

第二局：



比赛进行一段时间后，后手方计算出可以杀掉对手的路径，并沿此路径杀掉对手。

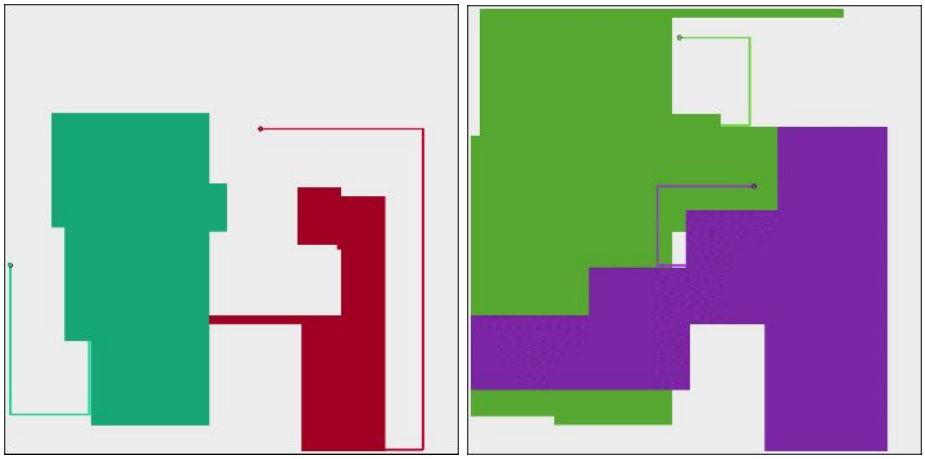
由上述两场测试局可见，我们设计的几个相应情况下函数功能能够较好地实现。算法在运行时间上也基本在预期之中，没有出现超时等问题，主要的运行时间开销发生在搜索环节。

3.3 经典战局

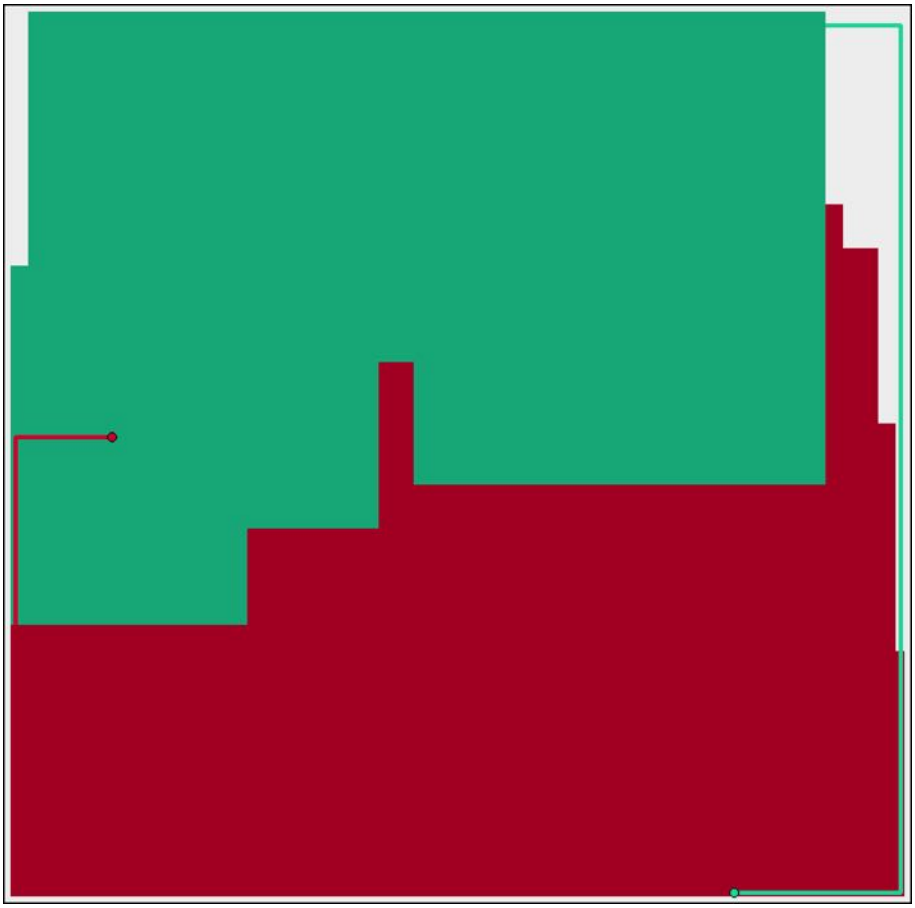
在热身赛中，由于我方代码还未完全调试好，后手采取的镜面方案由于代码出现失误，多次发生了撞墙情况而失败，但先手情况下我组所编写的代码发挥正常，在圈地、追杀对手、逃跑等方面实现情况良好，以下为两场经典战局：

(1) 我方先手：

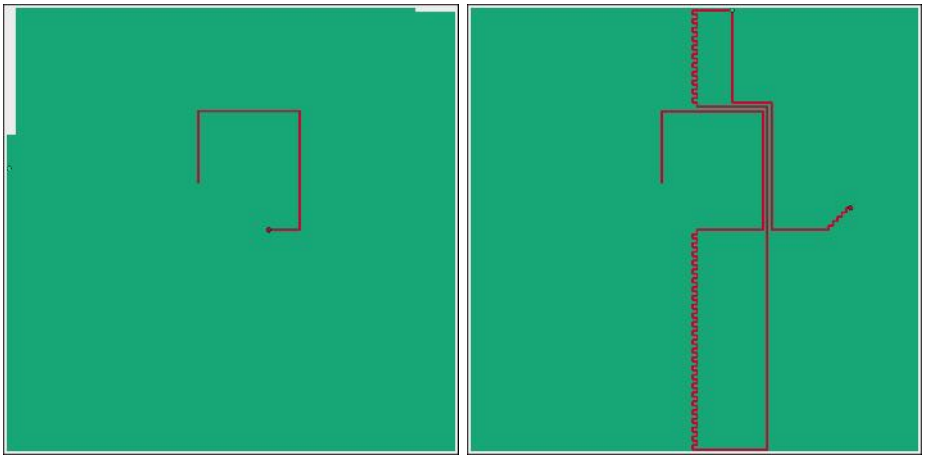
首先，双方激烈的进行圈地竞争：



此时，我方抓住红方离我方较远的机会，准备将全图的地圈入我方地盘，红方对此危机并未察觉，仍然在小规模的圈地：



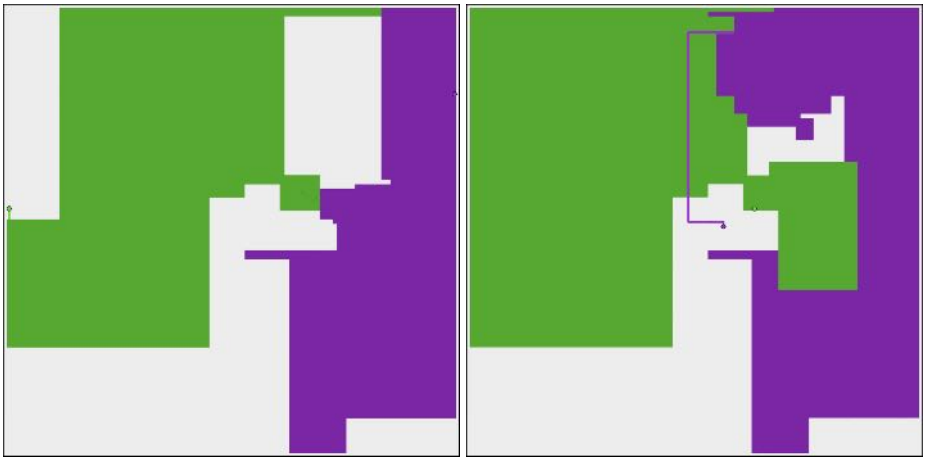
最终，我方占领的红方全部地盘，由于规则限定，红方此时没有地盘，已经是必死局面，在一番挣扎之后红方最终被我方击中纸带，我方胜利。但这里实际上也暴露出了我方过于保守的问题，在红方必败的情况下我方首先选择圈完剩余边角的地盘，最后才杀死对方：



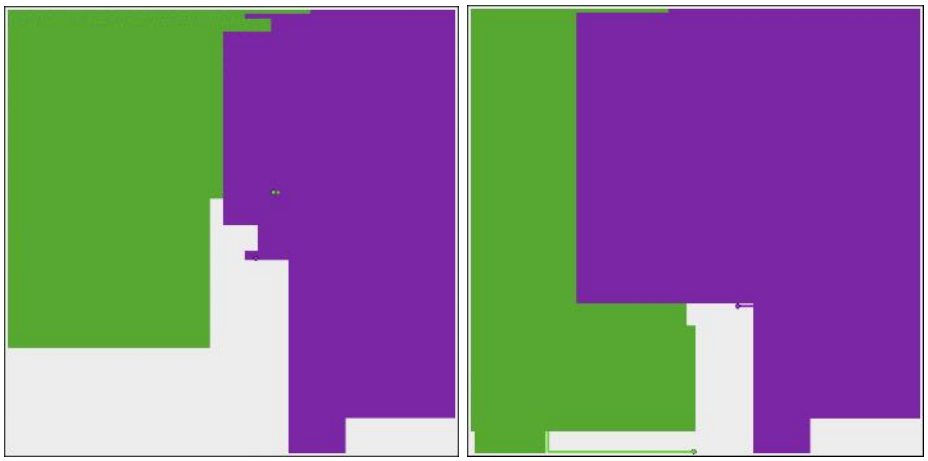
(2) 我方后手：

在经过一系列的调试和改良之后，我组放弃了后手镜面的思路，修正了撞墙的 bug，在后手局中也取得了可贵的反杀式胜利。

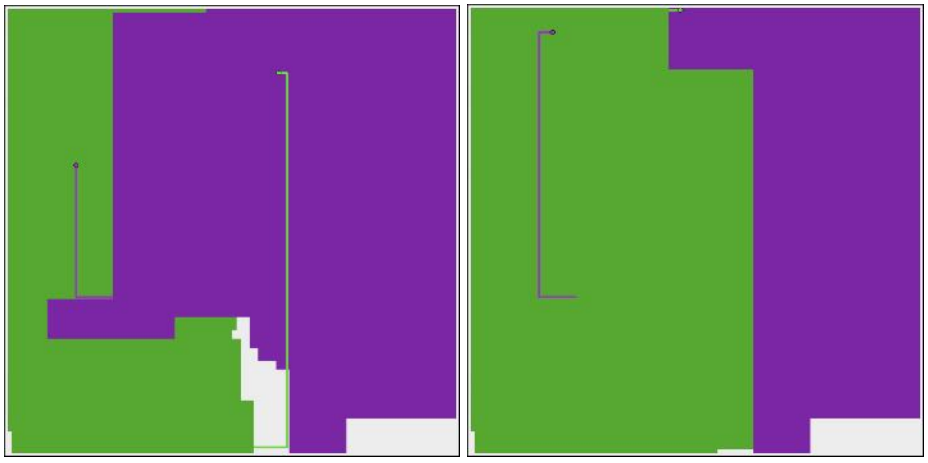
起初，双方进行圈地竞赛，绿方取得暂时性的领先：



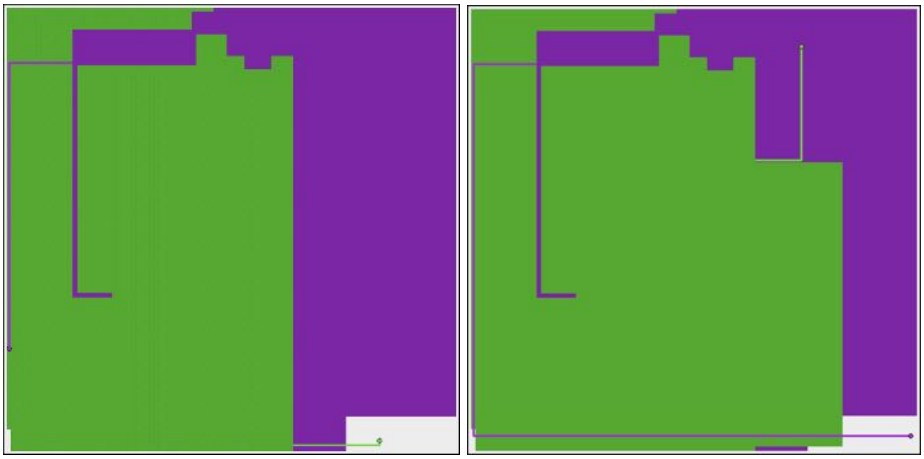
我方则是在右侧形成包围，以最短的线路圈下对方较大面积的一块地盘，并乘胜追击：



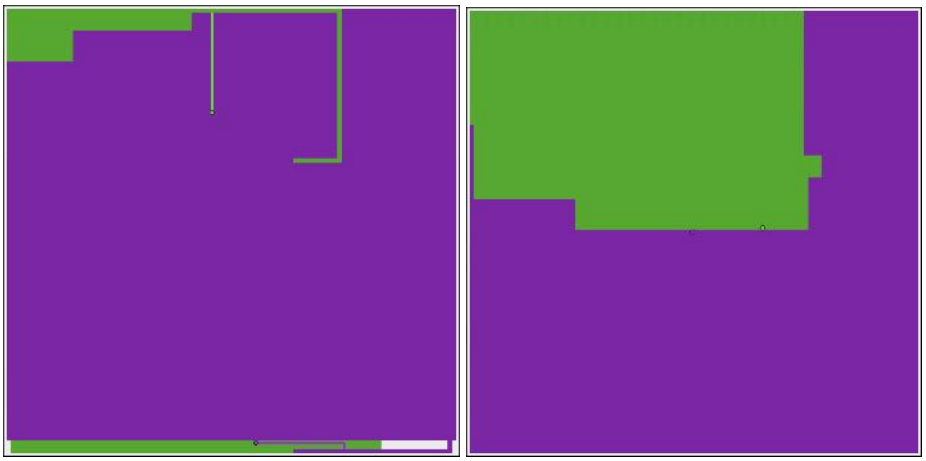
但此时我方未能意识到绿方在双方距离较远之时试图圈下大面积的地盘，导致在 2000 步时领先优势毁于一旦：



但我方以彼之道，还施彼身，同样在绿方距我方较远时背水一战，伺机圈下大块地盘：



成功之后我方抵御了对方最后的反击，赢下这一局精彩的反杀战：



4 实习过程总结

4.1 分工与合作

本小组大体上采取分工为代码组和报告组的方式，余圣杰和李南鸽同学主要负责代码的编写，方景行和赵佳迪同学主要负责撰写实习报告并对算法的思想提出建议，完成大作业的过程中四名组员交流、协作，共同进步。小组组建了微信群进行平时的交流讨论，上传代码改善操作，并举行了多次组会集中分析。

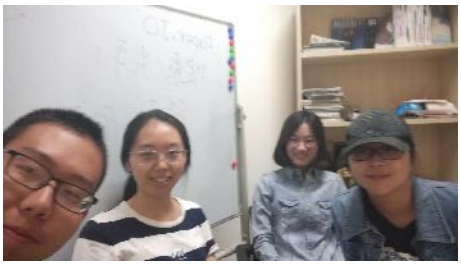
5 月 22 日，在数算课程布置了期末大作业并提出组队要求后，四人小组快速成立，并

建立了微信群用于交流，初步确立了分工情况，推举余圣杰同学作为组长。

5 月 29 日，在初步分析了大作业要求，在微信群中进行讨论、编写出初步代码之后，我们召开了第一次组会，探讨了尝试采取增强学习算法的可能性。

6 月 1 日，经过李南鸽同学对增强学习方法的研究和大家对此方法的疑问和讨论，我们最终决定仍旧遵循最初圈地为主的算法思想，放弃增强学习，采用更加脚踏实地的圈地搜索算法。

6 月 8 日，小组在决定放弃增强学习算法，初步完成了圈地代码之后，我们召开了第二次组会，大家并没有为过程中的曲折而感到挫败，而是积极讨论现有算法的优势与不足之处，在多次调试代码的过程中，大家对于代码都提出了创造性的优化建议。



6 月 11 日，比赛前夜我们在微信群中对最终的代码进行了调试，在确定没有 bug 之后我们怀着激动而紧张的心情期待第二天的比赛。

6 月 12 日，我们虽然在比赛中以小组第三的成绩遗憾出局，但大家都在完成此次期末大作业的过程中增进了对 python 和数据结构与算法的理解，还探究了增强学习的思想，每位组员都收获颇丰。

4.2 经验与教训

由于 paper.io 游戏不确定性强、可玩度高，比起黑白棋等二人零和有限的博弈过程类的游戏，不存在所谓的必胜方法，但规则又较为简单，所以本小组最初预想用增强学习（reinforcement learning）的方式训练随机算法，并且在前期讨论了一些算法：

minimax

思路：

这是我们首先想到的最简单的一个方法。

minimax 很适合对战棋类，深度搜索 dfs 我方和敌方的可能行走路线，通过树的形式，在我方层返回较大 alpha 值，在敌方返回较小 beta 值，中间配合剪枝的步骤节省时间和空间，最后选取其中的最佳结果，使得不论对方如何行动，保证我方利益最大化且不论我方如何决策，使得对方利益最小化。

这其中，alpha 和 beta 值直接使用当前局面双方的面积之差，如果能够击杀对方，则直接使用正无穷，如果给了对方击杀自己的可能，则使用负无穷。

但是在对算法的讨论中，我们发现，尽管每次行动时双方的可能行走方法只有三种，也即搜索程度可能达到很深，但是将需要开辟大量的存储空间来存储每一层时的局面状态，这将很大程度上限制 minimax 算法的效果，实际上不太好实现。

policy gradient

在阅读完一些 reinforcement learning 的文档后，我们学习了一些简单的机器学习算法来入门，包括 TensorFlow 文档里的鸢尾花种类识别程序，这是通过大量的已知数据训练 AI，使之判断鸢尾花种类的准确率上升。通过这个样例，我初步了解了搭建一个神经网络需要的 env、action、reward 几个部分，并尝试了 tensorflow、numpy 的常用函数。后来，我们看了一些增强学习的实践案例，包括 gym 中的 cartpole，但是 gym 模块对于非 gym 游戏的学习环境搭建参考价值不大。

在借鉴了一些教程后，我们又尝试了用神经网络训练机械手，使之可以快速反应碰触

随鼠标控制移动的小滑块。

在这个实践中，我们发现搭建 main 函数的框架非常清晰，也即：

env = Env( ) # 首先构建训练的 environment

参数初始化，包括 episode 和每个 episode 中的 steps 值

rl = # 选取一个决策函数，机器手的教程中使用的是比较难的 DDPG，paper.io 里面原打算使用 policy gradient 但最后没写出来…

Def train( ) # 训练函数，重复 episode 次，并将训练的数据记录下来

def eval( ) # 评估函数，调用时先清空当前的环境，重新搭建新的环境。然后根据已经收集到的训练数据，判断是否达到了收敛，如果达到了收敛，则本次训练的算法有所进步可以作为下一次训练的陪打。

Env( )

环境搭建的思路：

Env( ) 中包括一个 Environment 类

首先是构造函数 ___init___：用来初始化开始时的状态，对于纸袋圈地游戏即为初始时我方纸卷和纸带头

Def step( ) # 对于某个操作（例如左转、右转、直行），计算出数据的变化即可

Def reset( ) # 所有变量回复初始值，用于 evaluate 时判断训练的效果

在将这些模板运用到纸袋圈地游戏时，我们遇到了很大的困难——

首先是搭建的环境并不是独立的，而是双方交互 + 界面交互的，然后是由于对 TensorFlow 中的函数不够熟悉，很难有顺畅的思路在框架中补充写出完整代码。

缺少好的训练算法。这个是导致我们最后转为非 reinforcement learning 的最终原因，在小组讨论中，我们认为用一般的训练程序，从随机算法开始训练，在没有 GPU 的条件下，很难使得结果收敛，也即很难快速训练出能超过自己写的决策函数的策略。同时，如果只采用单一的策略训练，结果是否普适还有很大的不确定性。由于担心浪费过多的精力和时间，我们最后采用了比赛中呈现的策略函数。

完成大作业的过程中，我们经历了多次探索中的曲折发展。增强学习的方案被小组讨论放弃、热身赛中因代码存在漏洞而折戟沉沙，但我们没有因这些困难而气馁，组员们整

理心情，从头再来，吸取之前的经验和教训，完善新的策略函数代码，在小组中强敌林立的情况下仅以一名的差距遗憾出局。

不过在更改为策略函数后，虽然我们都对算法提出了建议并进行了完善，但最终没有脱出保守圈地的思想，因为采取的是一定不会被对方杀掉的情况下的最优解，当出现被对方杀死的威胁时就尽快回到自己的领地或在自己的领地徘徊，某种意义上一旦对方圈地面积大于我方就会陷入困局，没有增加代码策略的多样性，这也是导致我们最终遗憾出局的原因之一。

4.3 建议与设想

虽然本次 paper.io 比赛设计十分新颖，过程十分有趣，但由于安排在春季学期期末，题目在之前又很少接触，大部分同学不可避免的无法在完成代码过程中拿出全部的精力，同学们编写、调试代码的时间相对较短，最终提交的代码也不能完全代表小组的算法思想。其次，由于实现赢得 paper.io 对战的代码还是有一定难度，对 python 基础不高的同学相对有些不友好，尤其当分组中有比较强大的队伍存在使得晋级希望渺茫时，对大家的热情有一定的打击。

对未来选修这门课的学弟学妹们，我们想说这是一门非常有趣味、有内涵、有意义、有价值的课。在陈斌老师的课堂上，我们不仅能在轻松的课堂氛围中收获知识、提高自身的编程能力，还能在期末大作业的小组合作中亲身体验 Python 语言的优美之处，增强用算法解决问题的能力，同时还培养了团队合作能力，通过这门课能收获的知识与能力是远超学分和绩点可以衡量的，也期望这门课越办越好，每年都有更加有趣的赛事。

5 致谢

感谢陈斌老师在这一学期中带来的趣味丰富、意义深远的数算课程和紧张激烈、妙趣横生的 paper.io 比赛。

感谢各位助教的答疑和讲解。

感谢设计的可视化对战程序和对战平台的陈天翔等同学。

感谢王浩男学长为我组在增强学习方面提供了详细的指导。

感谢组长余圣杰同学的辛勤付出和有力监督，感谢各位组员的团结协作和戮力同心。

6 参考文献

[1] [美] Thomas H.Cormen,[美] Charles E.Leiserson,[美] Ronald L.Rivest,[美] Clifford Stein 著, 殷建平, 徐云, 王刚等译. 算法导论 [M]. 北京: 机械工业出版社,2013

第二十二章　F17_Victor 报告

伍峻琦 *、丁聪、周志竞、孙维来、杨状

摘要：采用最传统的防守—进攻普通策略，通过测定己方纸带、己方领地、对方纸带、对方领地四者相互间的最短路径规划行进路线（进攻或者回避）。算法中涉及迭代、动态规划、排序和搜索等数据结构。在最终的竞赛中，本组设计的算法列小组第三，遗憾未能出线。（P.S. 小组的一二名最终分获冠亚军，小组赛中我们组的算法与一二名算法比赛各有胜负，说明我们还是有争冠的实力的，此段话是我们安慰自己设计的算法被淘汰后失落的心情的，老师或者助教可以忽略，溜。）

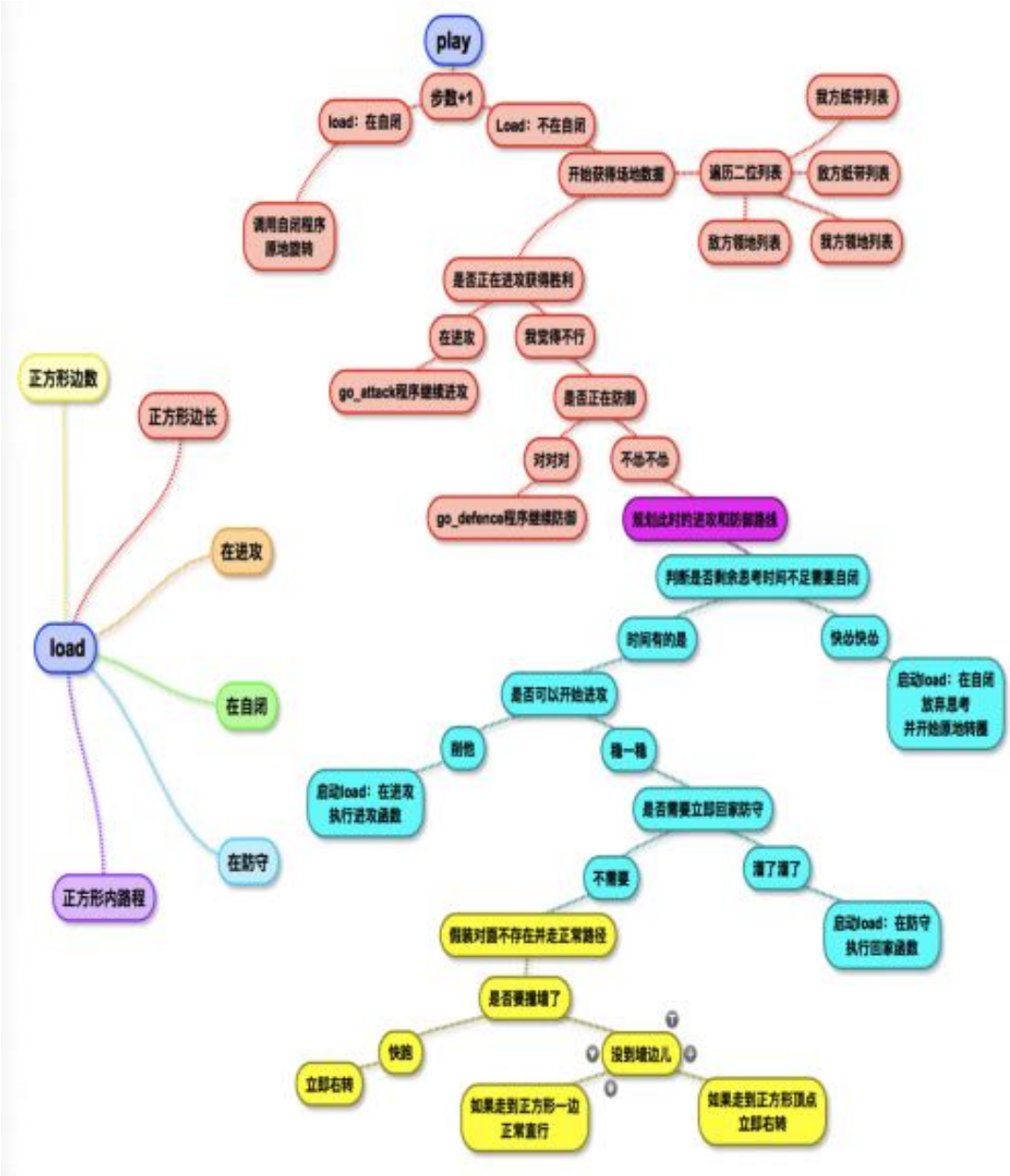
关键字：迭代，动态规划，排序，搜索

1 算法思想

本组算法采用最传统的防守—进攻普通策略，即设计函数，判断己方到达对方纸带最短路径和到达自身领地最短路径，为了简化程序节约时间，本程序在正常圈地时只会右转（并不影响圈地效率）。当判定地方可能杀死自己时回家，我方一定能杀死对方时进攻。此外，本程序的最初 11 个版本均有超时（或超时到电脑死机无法运行）的问题，因此本程序在剩余思考时间不足时会放弃思考，进行简单的领域内原地转圈围地保证自身安全。

1.1 总体思路

1.2 算法流程图



1.3 算法复杂度

根据本算法各个函数步骤划分：

1. 首先是判定撞墙的程序 knock_wall()，这个程序的时间复杂度 O(1)

2. 寻找回家最短路径的程序 checkroad() ，与出发进攻的 go_attack()，出发防御的 go_defence()，这三个程序需要遍历前进方向上的纸带列表，复杂度为 O(n)

但是获得纸带列表需要遍历整个地图，时间复杂度为 O(n^2), 故这个函数的时间复杂度为 O(n2)。

3. 判断我方进攻距离、敌人进攻距离、我方防守距离、敌方防守距离的几个函数 min_distance_meattack()，min_distance_enemyattack()，min_distance_medefence()，以及 min_distance_enemydefence()，要遍历某个方向的全部地图，和纸带列表，复杂度为 O(n^2)

最终，这个程序的时间复杂度为 O(a*n^2) [a 是常数]

然而这个算法并不能仅仅靠 O（）来判断运行时间，比如，我们在 AI_Victor_4(4) 时尝试将以上 O(n^2) 的搜索：

```
mybandlst, myfieldlst, enemybandlst, enemyfieldlst = [], [], [], []
for x in range(size[0]):
  for y in range(size[1]):
    if fields[x][y] == id_me:
      myfieldlst.append([x, y])
    elif fields[x][y] == id_enemy:
      enemyfieldlst.append([x, y])
    if bands[x][y] == id_me:
      mybandlst.append([x, y])
    elif bands[x][y] == id_enemy:
      enemybandlst.append([x, y])
```

的 mybandlst 和 enemybandlst 替换成了准确度略有下降的 O(n) 复杂度的算法：

```
if fields[x_me][y_me] != id_me:
  stat['mybandlst'].append([x_me, y_me])
  if stat['wo'] == 1:
    stat['wo'] = 0
  elif stat['wo'] == 0 :
    stat['mybandlst']=[]
```

```
    stat['wo'] = 1
    for x in range(size[0]):
      for y in range(size[1]):
        if fields[x][y] == id_me:
          stat['myfieldlst'].append([x, y])
        elif fields[x][y] == id_enemy:
          stat['enemyfieldlst'].append([x, y])

if fields[x_enemy][y_enemy] != id_enemy:
  stat['enemybandlst'].append([x_enemy, y_enemy])
  if stat['ta'] == 1:
    stat['ta'] = 0
  elif stat['ta'] == 0 :
    stat['enemybandlst'] = []
    stat['ta'] = 1
    for x in range(size[0]):
      for y in range(size[1]):
        if fields[x][y] == id_me:
          stat['myfieldlst'].append([x, y])
        elif fields[x][y] == id_enemy:
          stat['enemyfieldlst'].append([x, y])
```

但是实际运行时间并没有减少，而是比原始代码提前 3200 步超时!!

2 程序代码说明

2.1 数据结构说明

我们尝试过使用面向对象的代码，建立各种数据结构和自定义类，但是我们在测试过程中发现非常容易超时，最后决定用普通的赋值、判断、循环等语句，即便如此时间的消耗仍然很快，不得已又定义了自闭函数，即在决策时间所剩无几时原地旋转维持现状。

2.2 函数说明

（1）首先是 knock_wall() 函数，用于判断下一步会不会撞墙，由此决定下一步策略。

```
    def knock_wall(x_me, y_me, d_me, size):
        x_nextS, y_nextS = x_me + storage['next'][d_me][0], y_me + storage['next'][d_me][1]
        x_nextR, y_nextR = x_me + storage['next'][d_me + 1][0], y_me + storage['next'][d_me + 1][1]
        if x_nextS < 0 or x_nextS >= size[0]:
            return 'not_S'
        if y_nextS < 0 or y_nextS >= size[1]:
            return 'not_S'
        if x_nextR < 0 or x_nextR >= size[0]:
            return 'not_R'
        if y_nextR < 0 or y_nextR >= size[1]:
            return 'not_R'
        else:
            return 'go_on'
```

x_me，y_me 是此时的坐标，根据目前的方向 d_me = stat[‘now’][‘me’][‘direction’]，计算得到直走或右转时的下一步坐标：(x_nextS，y_nextS)，(x_nextR，y_nextR)，判断这几个坐标有没有越界（<0 或 >=size()），越界的话返回信息“不能直走”或“不能右转”，只要不越界就按原来的走法继续即“go_on”

（2）简单的 distance() 函数

```
def distance(x1, y1, x2, y2):
    return abs(x1 - x2) + abs(y1 - y2)
```

本游戏中走的步数比几何距离更有意义，故直接取 |x 差 |+|y 差 | 作为两点间距离

（3）check_next() 函数

```
def check_next(x_me, y_me, d_me, id_me, x_enemy, y_enemy, fields, size):
    if knock_wall(x_me, y_me, d_me, size) == 'go_on':
        x_nextS, y_nextS = x_me + storage['next'][d_me][0], y_me + storage['next'][d_me][1]
        x_nextR, y_nextR = x_me + storage['next'][d_me + 1][0], y_me + storage['next'][d_me + 1][1]
        id_S, id_R, id_now = fields[x_nextS][y_nextS], fields[x_nextR][y_nextR], fields[x_me][y_me]
        distance_S = distance(x_nextS, y_nextS, x_enemy, y_enemy)
        distance_R = distance(x_nextR, y_nextR, x_enemy, y_enemy)
        if id_now == id_me:
            if id_S != id_me and id_R == id_me:
                if distance_S <= 5:
                    return 'go_R'
            elif id_S == id_me and id_R != id_me:
                if distance_R <= 5:
                    return 'go_S'
            elif id_S != id_me and id_R != id_me:
                x_nextL, y_nextL = x_me - storage['next'][d_me + 1][0], y_me - storage['next'][d_me + 1][1]
                id_L = fields[x_nextL][y_nextL]
                if id_L == id_me and distance_S <= 5 and distance_R <= 5:
                    return 'go_L'
                else:
                    return 'go_R'
        return 'go_on'
    if knock_wall(x_me, y_me, d_me, size) == 'not_S':
        x_nextR, y_nextR = x_me + storage['next'][d_me + 1][0], y_me + storage['next'][d_me + 1][1]
        id_R, id_now = fields[x_nextR][y_nextR], fields[x_me][y_me]
        distance_R = distance(x_nextR, y_nextR, x_enemy, y_enemy)
        if id_now == id_me:
            if id_R != id_me:
                x_nextL, y_nextL = x_me - storage['next'][d_me + 1][0], y_me - storage['next'][d_me + 1][1]
                id_L = fields[x_nextL][y_nextL]
                if distance_R <= 5 and id_L == id_me:
                    return 'go_L'
        return 'go_R'
```

```
if knock_wall(x_me, y_me, d_me, size) == 'not_R':
    x_nextS, y_nextS = x_me + storage['next'][d_me][0], y_me + storage['next'][d_me][1]
    id_S, id_now = fields[x_nextS][y_nextS], fields[x_me][y_me]
    distance_S = distance(x_nextS, y_nextS, x_enemy, y_enemy)
    if id_now == id_me:
        if id_S != id_me:
            x_nextL, y_nextL = x_me - storage['next'][d_me + 1][0], y_me - storage['next'][d_me + 1][1]
            id_L = fields[x_nextL][y_nextL]
            if distance_S <= 5 and id_L == id_me:
                return 'go_L'
    return 'go_S'
```

id_me 是 stat[‘now’][‘me’][‘id’] 即我方的 id 号。

在 knock_wall() 返回 go_on 时说明直走和右转都不会撞墙，用 field 取到前方、右方、原位的领地属性，再用 distance() 取到前方一格、右方一格与敌方的距离。如果在自己的领地里，前方不是自己领地而右方是、直走离对方太近（<=5）就选择右转，同理判断能不能直走，最后判断左方的可行性，在三个方向都不是自己领地或者对方离自己太近的时候最终选择右转。前面这些都是在防止刚出自己领地就被杀，如果不在自己领地就没有这样的风险，返回 go_on 就好，后面有回家的算法。

在 knock_wall() 返回 not_S 或者 not_R 时，说明向对应方向走会撞墙死亡。仍要防止刚出自己领地就被杀的情况，只不过这时少了一种可选的方向，优先左转，不行就走另一个方向。

（4）判断我方进攻距离、敌人进攻距离、我方防守距离、敌方防守距离 min_distance_meattack()，min_distance_enemyattack()，min_distance_medefence()，min_distance_enemydefence() 函数

四个函数比较相似这里就放在一起说。我方进攻距离是这样计算的：遍历对方纸带，由于我方的方向基本为直走和右转，这里要看对方是否在我们的右前方。在此之外，还要通过 checkroad() 函数判断两点间能否被很好地连通（有时自己的纸带会将路拦住），可以的话找到离自己最近的那一个点，最后返回最近点的距离和其坐标。

```
def min_distance_meattack(enemybandlst, mybandlst, x_me, y_me, d_me):
    mindistance = 999
    getpoint = [None, None]
    n = len(enemybandlst)
    if n != 0:
        for i in range(n):
            x_to, y_to = enemybandlst[i][0], enemybandlst[i][1]
            if d_me == 0 and (x_to < x_me or y_to < y_me):
                continue
            if d_me == 1 and (x_to > x_me or y_to < y_me):
                continue
            if d_me == 2 and (x_to > x_me or y_to > y_me):
                continue
            if d_me == 3 and (x_to < x_me or y_to > y_me):
                continue
            if checkroad(x_me, y_me, x_to, y_to, mybandlst, d_me) is False:
                continue
            newdistance = distance(x_me, y_me, x_to, y_to)
            if newdistance <= mindistance:
                mindistance = newdistance
                getpoint = [x_to, y_to]
    return [mindistance, getpoint]
```

计算敌人进攻距离的时候不仅要考虑自己的纸带，还有考虑回家路上的“危险”，因此还要遍历自己的“防守路径”defence_potention

```
def min_distance_enemyattack(mybandlst, defence_potention, x_enemy, y_enemy):
    mindistance = 999
    getpoint = [None, None]
    n1 = len(mybandlst)
    n2 = len(defence_potention)
    if n1 != 0:
        for i in range(n1):
            x_to, y_to = mybandlst[i][0], mybandlst[i][1]
            newdistance = distance(x_enemy, y_enemy, x_to, y_to)
            if newdistance <= mindistance:
                mindistance = newdistance
                getpoint = [x_to, y_to]
        for i in range(n2):
            x_to, y_to = defence_potention[i][0], defence_potention[i][1]
            newdistance = distance(x_enemy, y_enemy, x_to, y_to)
            if newdistance <= mindistance:
                mindistance = newdistance
                getpoint = [x_to, y_to]
    return [mindistance, getpoint]
```

计算我方防守距离。身处领地、没有纸带时防守距离自然是 0。这里排除领地全在左后方的情况，在连接的比较好的情况下找到最短距离和对应的点

```
def min_distance_medefence(myfieldlst, mybandlst, x_me, y_me, d_me):
    mindistance = 999
    getpoint = [None, None]
    if len(mybandlst) == 0:
        mindistance = 0
    else:
        for i in range(len(myfieldlst)):
            x_to, y_to = myfieldlst[i][0], myfieldlst[i][1]
            if d_me == 0:
                if y_to < y_me or (y_to == y_me and x_to < x_me):
                    continue
            if d_me == 1:
                if x_to > x_me or (x_to == x_me and y_to < y_me):
                    continue
            if d_me == 2:
                if y_to > y_me or (y_to == y_me and x_to > x_me):
                    continue
            if d_me == 3:
                if x_to < x_me or (x_to == x_me and y_to > y_me):
                    continue
            if checkroad(x_me, y_me, x_to, y_to, mybandlst, d_me) is False:
                continue
            newdistance = distance(x_me, y_me, x_to, y_to)
            if newdistance <= mindistance:
                mindistance = newdistance
                getpoint = [x_to, y_to]
    return [mindistance, getpoint]
```

因为不知道敌方策略，敌方防守距离按最简单的方法去算

```
def min_distance_enemydefence(enemyfieldlst, enemybandlst, x_enemy, y_enemy):
    mindistance = 999
    getpoint = [None, None]
    if len(enemybandlst) == 0:
        mindistance = 0
    else:
        n = len(enemyfieldlst)
        for i in range(n):
            x_to, y_to = enemyfieldlst[i][0], enemyfieldlst[i][1]
            newdistance = distance(x_enemy, y_enemy, x_to, y_to)
            if newdistance <= mindistance:
                mindistance = newdistance
                getpoint = [x_to, y_to]
    return [mindistance, getpoint]
```

（5）min_distance_interattack() 函数，用于在领地较大时计算与敌方正碰的最短距离

```
def min_distance_interattack(attack_potention, x_enemy, y_enemy):
    mindistance = 999
    getpoint = [None, None]
    n = len(attack_potention)
    if n != 0:
        for i in range(n):
            x_to, y_to = attack_potention[i][0], attack_potention[i][1]
            newdistance = distance(x_enemy, y_enemy, x_to, y_to)
            if newdistance <= mindistance:
                mindistance = newdistance
                getpoint = [x_to, y_to]
    return [mindistance, getpoint]
```

（6）check_road() 函数

```
def checkroad(x_from, y_from, x_to, y_to, mybandlst, d):
    min_x, min_y = min(x_from, x_to), min(y_from, y_to)
    max_x, max_y = max(x_from, x_to), max(y_from, y_to)
    n = len(mybandlst)
    if min_x == max_x:
        for i in range(n):
            x_check, y_check = mybandlst[i][0], mybandlst[i][1]
            if x_check == x_from and min_y < y_check < max_y:
                return False
        return True
    elif min_y == max_y:
        for i in range(n):
            x_check, y_check = mybandlst[i][0], mybandlst[i][1]
            if y_check == y_from and min_x < x_check < max_x:
                return False
        return True
    if max_x - min_x == 1 and max_y - min_y == 1:
        if d == 0:
            for i in range(n):
                x_check, y_check = mybandlst[i][0], mybandlst[i][1]
                if x_to > x_from:
                    if x_check == x_to and y_check == y_from:
                        return False
                else:
                    if x_check == x_from and y_check == y_to:
                        return False
            return True
```

我们在多次超时之后决定仅采用判断和循环来组织这里的函数，因此这里只展示一小段，后面的代码思想基本一致，不再贴出。

本函数的功能就是看两个点是否比较好地连接在一起。首先取到坐标的最值，即得到两点的“外切矩形”，再判断我方的纸带是否切断了矩形使得不走矩形外的路径两点无法连接，没有这样的纸带就返回 True，否则 False。

（7）go_attack() 函数

```
def go_attack(x_me, y_me, d_me, potentionlst, mybandlst):
    x_nextS, y_nextS = x_me + storage['next'][d_me][0], y_me + storage['next'][d_me][1]
    x_nextR, y_nextR = x_me + storage['next'][d_me + 1][0], y_me + storage['next'][d_me + 1][1]
    n1 = len(mybandlst)
    n2 = len(potentionlst)
    for i in range(n1):
        check_x, check_y = mybandlst[i][0], mybandlst[i][1]
        if check_x == x_nextR and check_y == y_nextR:
            return 'next_S'
        elif check_x == x_nextS and check_y == y_nextS:
            return 'next_R'
    for i in range(n2):
        check_x, check_y = potentionlst[i][0], potentionlst[i][1]
        if check_x == x_nextS and check_y == y_nextS:
            return 'next_S'
        elif check_x == x_nextR and check_y == y_nextR:
            return 'next_R'
    return 'next_S'
```

首先遍历我方的纸带，防止进攻时撞上自己的纸带，选择另一个方向；在遍历设计好的进攻路线，按正确的路径走即可。

（8）go_defence() 函数

```
def go_defence(x_me, y_me, d_me, potentionlst, myfieldlst, mybandlst):
    x_nextS, y_nextS = x_me + storage['next'][d_me][0], y_me + storage['next'][d_me][1]
    x_nextR, y_nextR = x_me + storage['next'][d_me + 1][0], y_me + storage['next'][d_me + 1][1]
    n1 = len(myfieldlst)
    n2 = len(mybandlst)
    n3 = len(potentionlst)
    for i in range(n1):
        check_x, check_y = myfieldlst[i][0], myfieldlst[i][1]
        if check_x == x_nextS and check_y == y_nextS:
            return 'last_S'
        elif check_x == x_nextR and check_y == y_nextR:
            return 'last_R'
    for i in range(n2):
        check_x, check_y = mybandlst[i][0], mybandlst[i][1]
        if check_x == x_nextR and check_y == y_nextR:
            return 'next_S'
        elif check_x == x_nextS and check_y == y_nextS:
            return 'next_R'
    for i in range(n3):
        check_x, check_y = potentionlst[i][0], potentionlst[i][1]
        if check_x == x_nextS and check_y == y_nextS:
            return 'next_S'
        elif check_x == x_nextR and check_y == y_nextR:
            return 'next_R'
    return 'next_S'
```

前方、右方能直接回到领地就回，如果是纸带就选择另一个方向，在此之外走规划好的路径

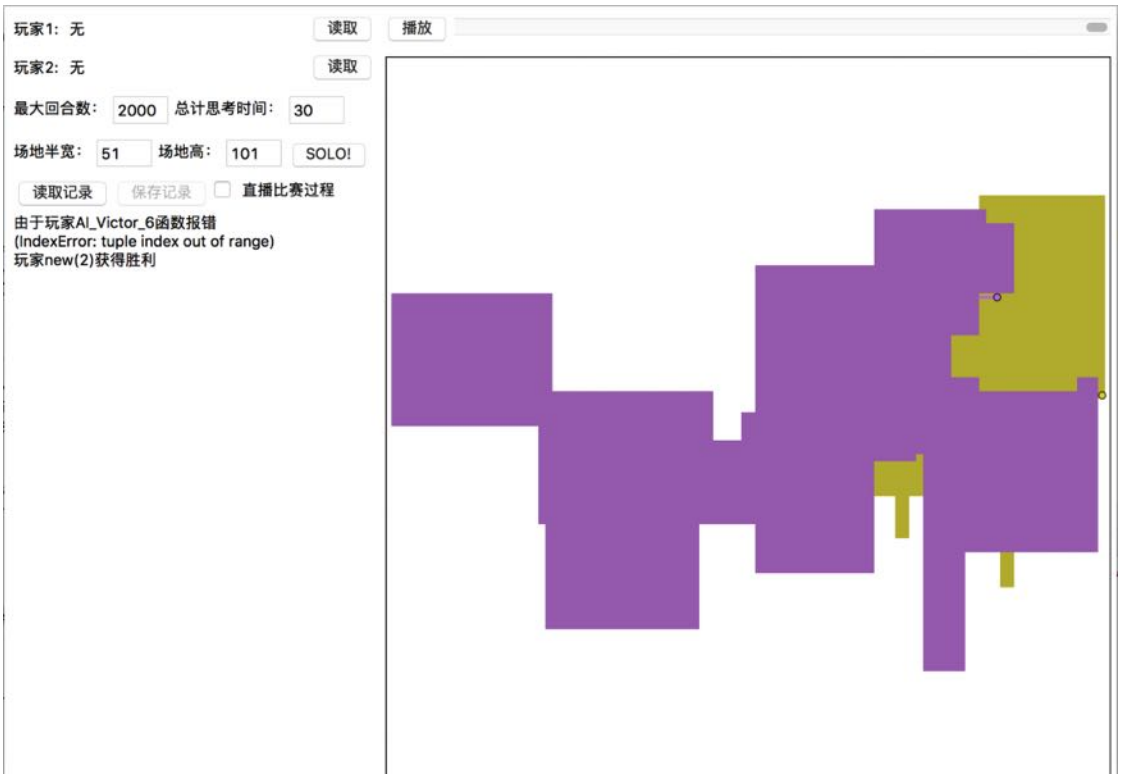
（9）自闭函数 keep_self()

```
def keep_self(x_me, y_me, d_me, id_me, fields, size):
    x_nextS, y_nextS = x_me + storage['next'][d_me][0], y_me + storage['next'][d_me][1]
    x_nextR, y_nextR = x_me + storage['next'][d_me + 1][0], y_me + storage['next'][d_me + 1][1]
    x_nextL, y_nextL = x_me - storage['next'][d_me + 1][0], y_me - storage['next'][d_me + 1][1]
    if knock_wall(x_me, y_me, d_me, size) == 'go_on':
        if fields[x_nextS][y_nextS] == id_me:
            return 'go_S'
        if fields[x_nextR][y_nextR] == id_me:
            return 'go_R'
        if fields[x_nextL][y_nextL] == id_me:
            return 'go_L'
        if fields[x_me][y_me] != id_me:
            return 'go_back'
        return 'go_R'
    elif knock_wall(x_me, y_me, d_me, size) == 'not_S':
        if fields[x_nextR][y_nextR] == id_me:
            return 'go_R'
        if fields[x_nextL][y_nextL] == id_me:
            return 'go_L'
        if fields[x_me][y_me] != id_me:
            return 'go_back'
        return 'go_R'
    elif knock_wall(x_me, y_me, d_me, size) == 'not_R':
        if fields[x_nextL][y_nextL] == id_me:
            return 'go_L'
        if fields[x_me][y_me] != id_me:
            return 'go_back'
        return 'go_R'
```

用于在决策时间即将耗尽时的保命程序。首先确认三个方向的坐标，前方、右方不会撞墙的话就按能回到领地走、不能就右转；knock_wall() 否认掉一个方向的话就少一种选择，能回领地就回，不能就右转。

2.3 程序限制

由于考虑的比较周全，我们的程序一般不会出错，但在练习时出现了如下图所示的问题



这是一种边界情况，即左方为墙而在领地内、前方和右方都不是自己领地，我们思考了很久，目前仍不清楚出错的地方。

另一个要说的是我们一定会被敌方杀死的情况。我们回家的程序是不包含左方和正后方的规划的，当领地被切割到我们失去左右方的领地、再一条狭长的领地里行走时非常容易被在附近的敌方击杀。

3 实验结果

3.1 测试数据

硬件配置：Inter(R) Core(TM) i7-7500U CPU @2.70GHz 2.90GHz

操作系统：Windows 10

Python 版本：python 3.6

1、测试阶段结果：（1）Victor_1: 选择与技术组的内置 AI 函数以循环赛的方式进行

测试

AI 函数	战绩
AI_4x9	68 胜 37 负 15 平
AI_6x6	78 胜 27 负 15 平
AI_7x12	83 胜 35 负 2 平
AI_random_1	32 胜 88 负 0 平
AI_random_2	33 胜 87 负 0 平
AI_random_3	30 胜 90 负 0 平
Victor_1	79 胜 39 负 2 平

（注：后面的 Victor 函数版本由于技术组的循环赛函数不能使用，故无详细的测试数据）

（2）选择 AI-glory_of_mankind 函数来测试 Victor 函数

2、热身赛阶段测试结果

第 1 次热身赛

Victor VS 对象	Victor 先手战绩	Victor 后手战绩
Alpha_1	0 胜 4 负 0 平	2 胜 8 负 0 平
Alpha_2	2 胜 5 负 0 平	4 胜 6 负 0 平
Kilo	5 胜 5 负 0 平	1 胜 5 负 0 平
Bravo	0 胜 4 负 0 平	2 胜 8 负 0 平
Foxtrot	0 胜 10 负 0 平	0 胜 2 负 0 平

第 2 次热身赛

Victor VS 对象	Victor 先手战绩	Victor 后手战绩
Foxtrot_2	3 胜 7 负 0 平	0 胜 4 负 0 平
Foxtrot_3	0 胜 10 负 0 平	0 胜 1 负 0 平

India	6 胜 4 负 0 平	6 胜 4 负 0 平
Alpha_2	9 胜 1 负 0 平	8 胜 2 负 0 平
Alpha_3	10 胜 0 负 0 平	10 胜 0 负 0 平

第 3 次热身赛

Victor VS 对象	Victor 先手战绩	Victor 后手战绩
Foxtrot_3	4 胜 4 负平	5 胜 5 负 0 平
Foxtrot_4	3 胜 2 负平	2 胜 8 负 0 平
India	6 胜 4 负 0 平	4 胜 5 负 0 平
November	4 胜 6 负 0 平	2 胜 1 负 0 平
Oscar	7 胜 2 负 1 平	8 胜 1 负 1 平

3、友谊赛测试结果

（第三次热身赛由于修改了部分参数，于是和分区其他组约定了友谊赛）

（1）第一次友谊赛

时间：2018.6.11 21：30

小组：Victor、Foxtrot、November

形式：各两两对抗 10 局，其中先手、后手各 5 局

结果：由于忽略了记录，大致结果五五开

（2）第二次友谊赛

时间：2018.6.12 13：30

小组：Victor、November

结果：Victor 5 胜 3 负 0 平（攻击对方纸带取胜 2 局，对方自杀获胜 3 局，我方报错

负 1 局，我方被撞击负 2 局）

4、实战阶段结果

由于分区有一小组代码出现问题，分区 6 各小组并没有完全赛完，暂列第 3，未出线。

3.2 结果分析

1、测试阶段

（1）Victor_1：只具备圈地功能，不具备防守和攻击功能

代码分析：除了 AI_7x12 在战绩上领先于 AI_Victor_1 之外，其余的内置 AI 均输给 AI_Victor_1，在针对小范围的内置圈地 AI，AI_Victor_1 在圈地效率上具有优势的，前提是在给定的回合数内并没有发生碰撞（内置 AI 的小范围圈地减少了两者发生碰撞的频率），而且针对 AI_random_1、2、3 系列由于它们的随机性，AI_Victor_1 则具有更大的优势；但是在与 AI_7x12 的对战中，AI_Victor_1 的圈地效率仍高于 AI_V7x12，但是由于 AI_Victor_1 本身不具有防守能力，当两者圈地范围接近时，具备更优圈地效率的 AI_Victor_1 更容易被攻击（圈地效率高则意味暴露在外的纸带更多，更容易被攻击）。

（注：往后的版本由于循环赛函数不能使用，故缺少对战结果数据）

（2）Victor_2：具备圈地和防守功能，不具备攻击能力

代码分析：新添加的防守（躲避）函数 gohomesuccessfully 具有一定防守功能，但是在测试过程中仍会出现 bug，对于全局的信息把握不够准确，不能正确判断对方纸卷的位置并做处正确判断。

（3）Victor_3：具备圈地、防守和一定的攻击能力，算法初步形成，存在超时问题

代码分析：在接受 AI-glory_of_mankind 的攻击时不能及时躲避，纸卷的轨迹和函数中设定的指令不一致

（4）Victor_4：基本解决了上一代代码耗时问题，并且加设探头函数

代码分析：（1）小组选择以减少循环语句而增加判断语句的方式来减少耗时，具有一定成都的效果，但是仍不能解决少数情况的超时；（2）探头函数的出现提高了在离开自己领地时候的安全性，提高了自身的容错率，确保纸卷自身在离开领地前自己周围一定范围内是安全的，同时在对方靠近我方纸卷的时候能在相对安全的防卫内做出躲避或是进攻的

反应，如何反应取决于对方与我方纸卷的位置和对方离开对方领地的距离的比较。

（5）Victor_5：加设“自闭”函数使 AI 在时间将要耗尽时“自闭”，并且调整了部分参数

代码分析：“自闭”函数很好的解决了超时问题，在正常运行其他函数的情况下若时间所剩无几时还未响应时能最快的作出最简单的反应，而放弃原来的耗时指令；这样的简单指令能够及时做出响应，解决了超时的问题。

（6）Victor_6：在赛前与分区其他小组友谊赛后所确定的最终版本

代码分析：无明显 bug，不存在超时问题，圈地效率仍不高，主动进攻能力较弱，但是能及时躲避对方的进攻。

2、热身阶段

（1）第一次热身

（图中被攻击方为我方纸卷）

结果分析：

a、圈地效率偏低；

b、几乎不具备攻击能力，获胜场中多数是以对方报错二获胜，在回合数结束比圈地以及攻击上均没有优势；

c、躲避函数不够成熟，几乎不能躲避对方纸卷的攻击，从图中可以看出我方纸卷检查不到对方纸卷的位置而觉察不到危险；

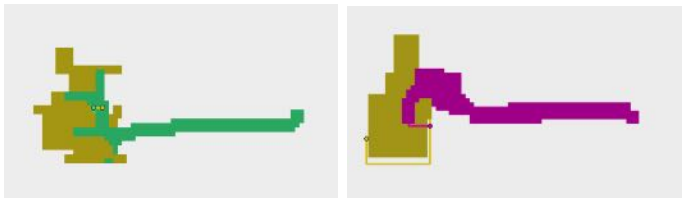
d、自身有 bug，存在超时或自杀等情况；

（2）第二次热身

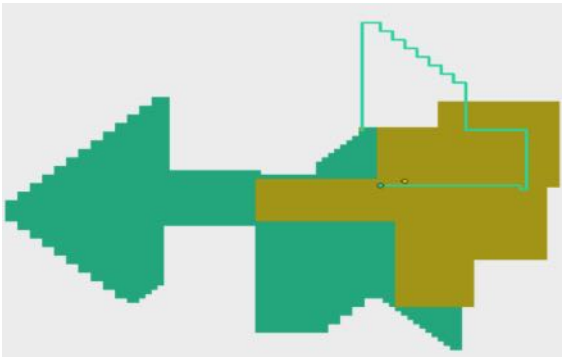
结果分析（如图所示）：

a、tantou（探头）函数不能顾及后方区域；

b、不能确定对方纸卷位置，导致不能正确躲避而受攻击；



c、如下图所示，攻击函数存在问题，在我方领地内与对方纸带相距很小的距离任由对方穿梭而无攻击行为；

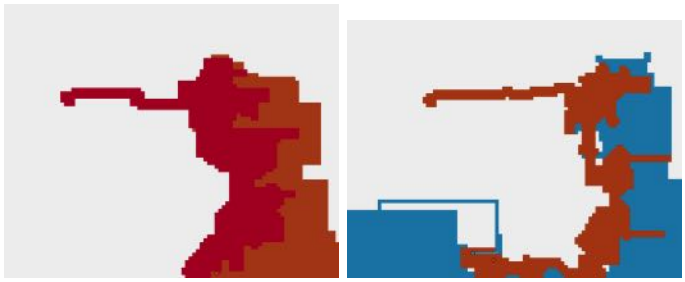


d、已经能够较好的躲避对方纸卷的的攻击，能够及时安全回到我方领地。

（3）第三次热身

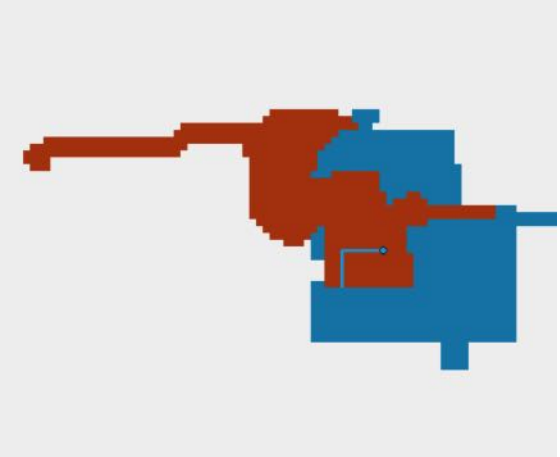
结果分析（如图所示）：

a、Victories 函数已经具备相当好的躲避能力，在对战中经常耗尽回合数；



b、攻击能力具有很大的提升，一般不主动出击，tantou（探头）函数具有相当优良的“进攻 or 防守”的判断能力，在对方靠近时能够在有一点能够把握的前提下发起攻击。

c、函数出现少量撞墙情况，偶尔对于对方纸卷的判断不准确而出现失误不能及时回家。



3、友谊赛阶段

（最后一次热身赛后由于存在少量 bug，于是小组修改了部分参数，在实战之前选择与同一分区的其他小组进行了友谊赛来测试参数的修改是否合理）

（1）第一次友谊赛

结果分析：

a、Victor_6 无明显 bug，函数总体以判断句为主，很少有循环语句，多数情况下对方代码报错撞墙儿我方获胜；

b、knockwall（撞墙）函数综合考虑多种情况，临界判断比较准确，纸卷几乎没有出现撞墙的情况；

c、圈地不占优势，回合结束则因圈地小告负；

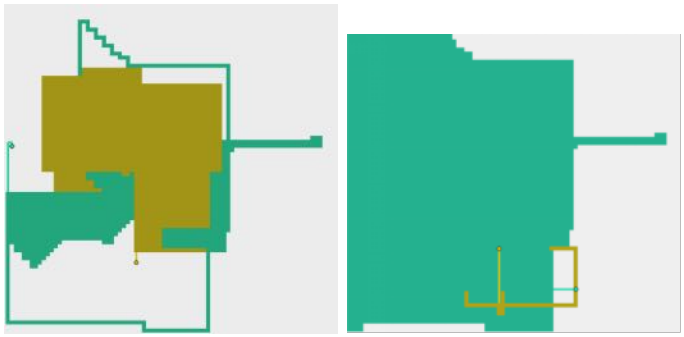
d、在具有相当高的把握的时候我方能击杀对方，tantou（探头）函数会考虑纸卷在离开我方领地前其周围的一定小范围的属性，所以我方纸卷在与对方相隔较远时多以圈地和躲避为主，在小范围内才会主动攻击，而且此时纸卷离我方领地较近，在攻击失败的情况下也能安全“回家”，具有一定的容错率。

（2）第二次友谊赛

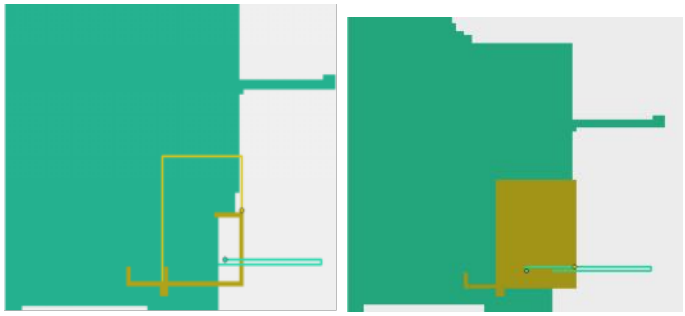
分析：版本与 Victor_5 几乎一致，结果分析同上一次友谊赛

3.3 经典战局

（1）第二次热身赛 Victor VS India（黄色方为 Victor；绿色方为 India）

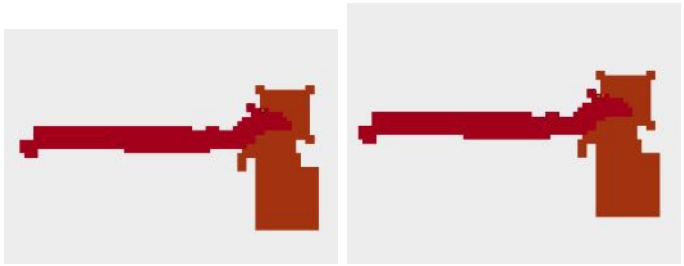


过程：a、India 欲包围我方进行收割，万幸之下 Victor 还仅存部分领地（不至于失去全部领地属性而导致追尾自杀，留下了残血反杀的机会）

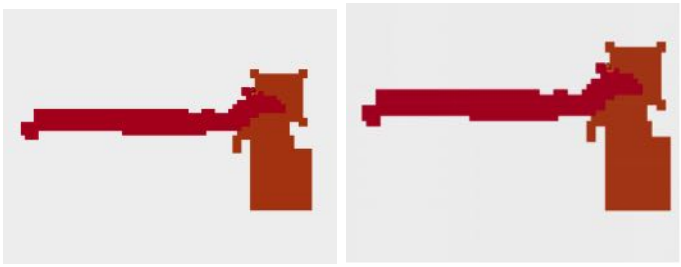


b、Victor 迅速将仅存的领地利用形成小领地，让正在浪的对方纸卷留在我方领地，Victor 迅速追上准备回家的 India 纸带，在其回家之前实现反杀。

（2）第三次热身赛 Victor_4 VS Foxtrot_3（左边暗红色为 Foxtrot；右边暗橘色为 Victor）



过程：a、面对以攻击力强著称的 Foxtrot 函数步步入侵与紧逼，Victor 临危不乱与之周旋；熟知 Foxtrot 的套路先粘着你，在你周围盘旋而放弃圈地，只待对手犯错进入它的领地后再进行猛攻；此时 Victor 孤军深入，进入到对方领地，果然 Foxtrot 紧随其后，欲追杀 Victor；



b、Foxtrot 在靠近 Victor 纸带的时候离开了自己的领地，Victor 抓住机会一招回马枪将 Foxtrot 击杀。

4 实习过程总结

4.1 分工与合作

本次小组成员分工大致如下：

伍峻琦	组长兼核心代码编写
丁聪	核心代码编写
周志竞	对代码进行测试，并找出经典战局
杨壮	分析经典战局，并找出原核心代码中的相关问题，与两位主要代码编写者进行商榷，并一起进行修改。
孙维来	分析经典战局，并找出原核心代码中的相关问题，与两位主要代码编写者进行商榷，并一起进行修改。

合作交流的方式主要是定期开小组讨论，确定大家进展的情况，并且大家在一起进行一些程序上的测试比如与给定的代码进行比试，如本代码与本身相互调试，如本代码与人

类（我们组员操控）进行比试，并依此发现本代码中的小问题，同时我们也对于游戏初期的圈地方式和与对方 AI 博弈方面展开了深入的探讨。

下面是我们组会的照片：

4.2 经验与教训

关于本次大作业的经验教训，我觉得有几下几点：

1. 我们与很多组的发展方式并不相同，从代码的基本架构，代码采取的策略，可以说都不是由仅仅一个人完成的，而是大家共同讨论的结果，从圈地策略的讨论，到如何选择进攻与防守的方式，都是由各位同学集思广益的。

2. 我们在进行比赛之前，也跟某些小组私下里进行了友谊赛，在这些在对局中，我们有发现了各自代码的一些问题，也让我们更好的改进了代码的策略。

3. 关于我们可以改进的地方，我觉得可能是如果我们可以在代码的博弈方面再改进一些就好了，大家可以在多进行一点头脑风暴，想到一些更加好的策略。

4.3 建议与设想

建议：在竞赛方面的小组赛方面可以在赛制上进行一定的改良，让某些十分特殊的小组也有展示的机会，并且这种博弈游戏那些输的一方在对阵所谓最终的八强是不一定输的，

所以下次比赛的赛制有待商榷。

5 致谢

此次大作业竞赛中小组通力合作，各组员均为算法的改进与更新有不可磨灭的贡献。虽然我们小组设计的算法最终没能在小组中出线，但我们感谢我们每个人接近一个月的付出。同时，对于本次数算大作业设计的技术组和老师助教团队，感谢辛勤的付出，提供了一个很棒的算法平台和竞赛平台，让期末的大作业竞赛更有趣也更具公平性。另外，我们组还要感谢同一个赛区的 F 组和 N 组，我们三个组在比赛前的晚上进行了一次友谊赛，三个组讲的算法互相切磋，其间找出了各组算法上的漏洞，促进了各组算法实力的增强。

总之，此次数算大作业是有趣的，更是有意义的，通过此次大作业，我们对于数据结构的很多知识有了更深入的了解，对于算法的编写也更为高效、娴熟。

6 参考文献

竞赛平台：

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

相关介绍文件：

- sessdsa2018-paper.io.pdf
- match_core 代码分析.pdf
- 文档 0605.pdf
- AI_Template.pdf

（以上文档为竞赛平台中的指示说明文档）

数据结构构建参考：

- 张乃孝、陈光、孙猛，《算法与数据结构》，高等教育出版社，第 3 版
- 殷人昆，《数据结构》，清华大学出版社，第 1 版

第二十三章　F17_Whiskey 报告

柯赵轲 * 李烨星许鹏程颜松昆

摘要：（原理以优先级：攻击；回家；平行扩张；直线扩张为顺序。主要是保守策略。距离判定依照绝对值算法，所有判定在距离判定的基础之上。涉及：列表，字典。结果概述：算法处于中上水平）

关键字：（List Dict ）

1 算法思想

先判断能否一定能击杀对方，如果可以的话就将对方击杀，游戏结束。

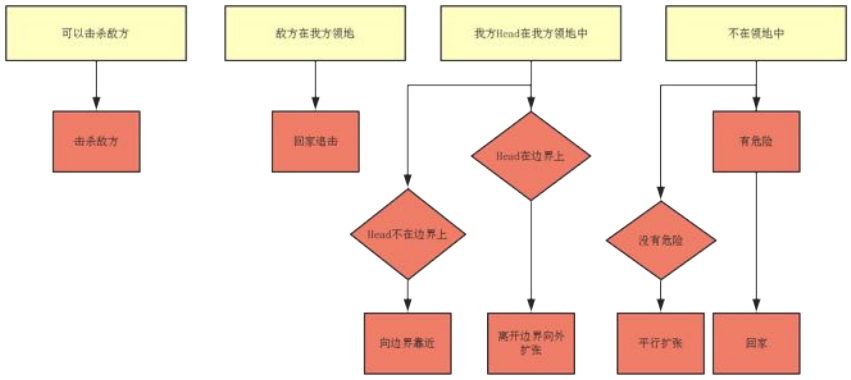
如果敌方在我方领地内，就回头去追击敌方

如果我方 Head 在自家领地中，则向最近边界靠拢并离开边界圈地。

如果我方 Head 在自家领地之外，先判断是否处于安全状态，如果敌方有机会击杀我方，就回家，不再扩张。

1.1 总体思路

1.2 算法流程图



1.3 1.3 算法运行时间复杂度分析

由于代码中运用的是简单的 if 语句，所以判断时间用得比较短，这也是相比其他小组代码的优势之一。每一步的判断最多用到一重循环，算法复杂度近似等于 O（n）。

2 程序代码说明

2.1 数据结构说明

代码并没有用过多的数据结构，大部分都是使用 if 语句来进行判断，枚举各种可能的情况，并作出相应的对策。

2.2 函数说明

传入一个点和一个点阵，得到这个点到这个点阵的最短距离 (粗略算法）

def TheShortest(x,y,array):

计算该点到点阵每个点的曼哈顿距离，并求出最小值，记录该点的位置，由此可以判断回家的最小距离

是否进攻判断

def Whetherattack():

判断是否进攻。判断依据：我方 Head 到敌方纸带的最小距离小于敌方 Head 到我方纸带最小距离 +3（保证敌方无法击杀我方）且我方 Head 到敌方纸带的最小距离 +2 小于敌方回家的最小距离（保证敌方无法逃脱）

是否回家

def WhetherBack():

判断是否回家，判断依据：敌方到我方纸带的距离的二分之一小于等于我方回家的最短距离，或者敌方到我方纸带的距离小于我方回家的最短距离 +4，这样可以防止被对面击杀。此外，由于回合数的限制，为了保证面积最大化，在结束之前回家。

是否平行扩张判断

def WhetherStayOutside():

判断是否可以安全进行扩张。判断依据：敌方到我方纸带的距离小于我方回家距离 +7。这样可以比较好的进行扩张。

2.3 程序限制

在函数 Whetherattack() 中判定是否要回家存在一些限制，在一些极端情况下，由于我方移动一步有可能使敌方纸带到我方纸带的最短距离缩小 2，导致判定失败，从而可能出现被击杀的可能。

3 实验结果

3.1 测试数据

实验环境说明：

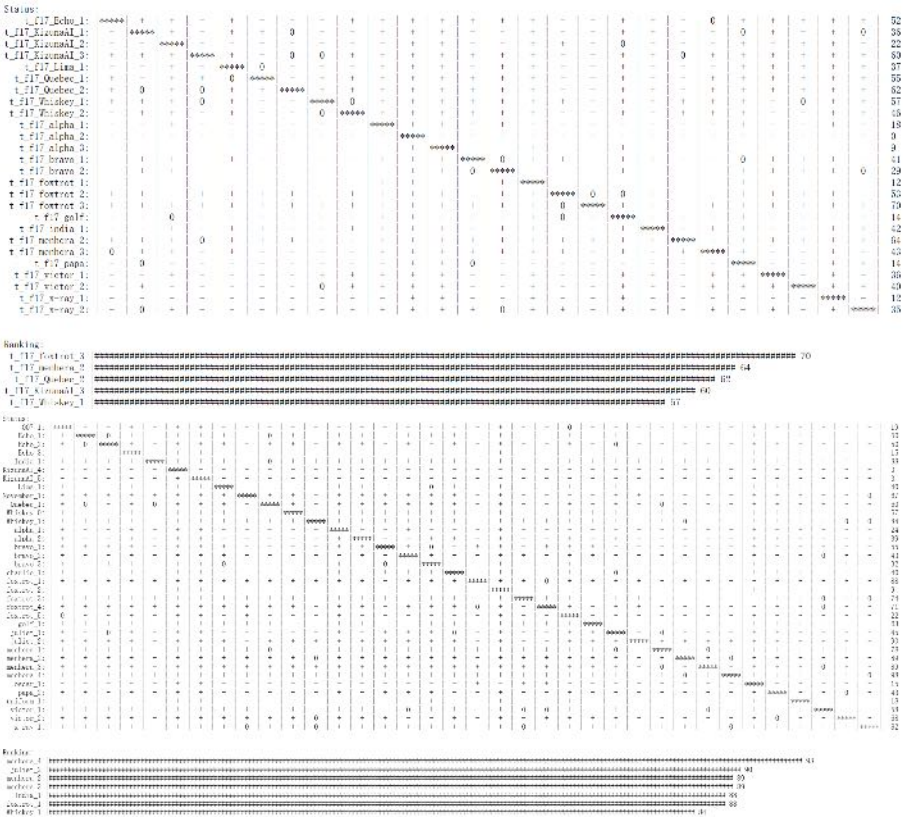
- 硬件配置：CPU: Intel(R)Core™ i7-8650U CPU @ 1.90Hz 2.11GHz 内存：16GB
- 操作系统：Windows 10 1803

- Python 版本：Python 3.6.4

测试：利用课上给出的对战平台和自己对战以及和其他两组代码（A B）对战分析。

测试结果：和自己对战几乎持平；和 A 对战以 6：11 失利；和 B 对战以 11：3 胜利。

（下面两个图是参加热身赛的结果）



3.2 结果分析

测试：

与自己对战：与自己对战基本对称，都是较为保守的策略，运气成分较大。主要用于 BUG 的检查，与自己对战过程中，无明显 BUG。

与 A 对战：A 的基本对战思路是跟进对方，进行近身缠斗，比较克制本算法。我方算法的扩大思路基本是最后一个优先级，所以对于此算法无法有效的扩大地盘。本算法在缠斗方面采取消极躲避措施，是为了防止强攻导致失败。这个策略防止我方在离开领地时恰

好被敌方碰撞，并且第一优先级的攻击方式也使得我方可以在领地内击杀对手。但是由于地方的缠斗思路，以及我方急需扩大领地的矛盾，导致我方死亡原因大多是己方领地被敌方包围导致自身领地瞬间变为地方领地，失去领地后被击杀（这里我方没有考虑到领地突变问题，因为算法过于复杂）。

与 B 对战：B 的基本对战思路与我方相同，都是远离对方领地扩张。我方算法在排除：攻击，回家。这两点后，采取直线向前的思路，这种算法策略可以将所占的领地尽量扩大，避免不必要的拐弯。所以我方领地的占地效率大于地方。同时，我方在近战缠斗方面采取消极躲避措施，但是地方的近战模式并非跟紧缠斗，无法圈住我方，并且在进攻上冒进，使得我方的第一优先级的攻击策略得以施展，从而在近身战也能够取得胜利。

热身赛：在热身在过程中，只有第一组采取了近身缠斗的作战方式，所以在与第一组的作战中，和 A 组有类似的局面，不再分析。我方算法结构比较简单，所以时间复杂度很低，运行时间基本在 5s 以内。对于其他排在我方前面的队伍。主要分析是我方的回家策略的漏洞（在前面已经分析过），但是比较占地面积，我方的算法基本上是占优势。

实战对弈：在最后的比赛过程中，由于我方参赛小组排在前面的队伍基本采用了近战缠斗跟踪的方式虽然我方在圈地以及时间复杂度的方面领先，但是由于我方的近战采取消极模式，导致战斗失利。平局数比第二名多了一句，败局只输给了小组第一，惜败。

主要时间运行开销：我方算法在领地计算方面采取边界遍历，时间复杂度大大降低，同时对于回家等等距离判定同样采用边界遍历。时间大多画在判定是否攻击，是否回家。主要是本算法着重于保守策略，所以扩张领取的判定需要大量的时间。实际上本算法的时间复杂度还是很低，比赛时间大多可以控制在 5s 以内。

3.3 经典战局

如图是和 B 的对战过程。我方与 B 均不会主动与敌方碰撞，所以我方在扩张方面非常有优势，从本局可以看出我方（绿色），的圈地非常完美，将大部分纳入领地。

4 实习过程总结

4.1 分工与合作

本小组有 4 名成员，组长是柯赵轲，组员包括许鹏程、颜松昆、李烨星。具体分工为：四人共同参与前期的游戏体验以及算法初步讨论；柯赵轲和许鹏程负责代码的编写；李烨星、柯赵轲和许鹏程负责调试以及测验；四人共同讨论优化和升级（柯赵轲和许鹏程贡献大）；颜松昆和李烨星负责报告的总结和书写。

交流形式主要分为线上（微信群）和线下（组会）两种形式。线上讨论主要是对算法的讨论以及对和其他组比较的讨论，线下组会主要是测验和调试代码。

本组进行了 3 次组会，第一次讨论了分工，初步交流了对算法的设想；第二次组会初

步完成了代码并进行了初步的测试并交流了进一步改进的想法和策略；第三次组会时代码基本达到最终版，进行了进一步的调试，并和其他组进行了比较，对自己的代码进行了最后的升级。在三次组会之前之间与之后，都进行了大量的线上交流，保障了大作业的整体进度以及代码的一次次的升级换代。

除了上述方式，组内同学还在各种一起上的课的课间（偶尔还有课上…）对大作业进行了讨论。

4.2 经验与教训

本次实习过程比较满意的有以下几点：

1. 编写代码前对游戏的熟悉以及每个人对算法设想的初步贡献。保证了后期工作的顺利快速进行。

2. 随时随地的交流，包括微信和各种零碎时间的见面，使得想法可以快速交流。

3. 及时和其他组交流，并对自己的代码做出改进。

4. 及时的总结和精益求精的精神，模拟对战后及时分析自己代码的疏漏并做出改进。

5. 拿到任务后及时的分工以及强大的执行力，保证了工作的效率。

感觉需要改进的有以下几点：

1. 线下的交流应该增多，线上交流有时回复不及时，而且很多时候说不清楚，会延缓进度。

2. 对多种算法的尝试还不够。在最初确定方向之后，一直是对这一种思路进行改进和微调，并没有大修大补，进行其他方法的尝试。

3. 闭门造车。并没有参考太多其他的文献，主要是靠自己的想法和编写。

4.3 建议与设想

希望在组队的时候可以提高队伍内成员的多样性，感觉今年的分组大多数是同年级同专业的人组队，而且很多是之前就已经认识的，这导致队内想法整体上不够丰富。跨专业跨年级的组队也许会让每个组内多样化的想法碰撞出不一样的火花。另外希望在最开始确定对战平台的 python 版本，本组组员之一的舍友的小组碰到了由于版本不同导致代码出错，重改后战力大大下降的问题。针对上述组内多样化缺失的问题，大胆设想可以把各个组的代码模块化，分为不同的功能板块，比如进攻、防守，然后进行重组，看看能否有更强的效果。

另外希望在最后的对战中通过多种方式决定最后的成绩，比如本组算法在时间复杂度上很有优势，但是在最终环节并没有体现。

对学弟学妹的话：数据结构与算法是一门很有意思也很实用的课程，陈斌老师的课程也非常充实有趣。对 python 语言的学习只是这门课程的第一步，也是最简单的一步，之后对不同算法的学习和理解，才是这门课程最内涵，最有价值的部分。信息技术已经渗透到所有的学科领域，数据处理和分析也成为各学科必须的基础研究方法，希望你们在学习过程中能继续加强计算思维的训练，这会为你们未来的学习科研提供强大的支持。

5 致谢

感谢北京大学地球与空间科学学院的陈斌老师，他开设的他开设的数据结构与算法课程给了我们这个编写大作业程序的机会，他在课上所讲的知识是我们完成代码的基础。

感谢数据结构与算法课程的 7 位助教：易超、陈旭、赵宇、冀锐、滕沅建、缪舜、张

颢丹。

感谢给我们提供自己算法进行对战的两个小组。

6 参考文献

数据结构与算法 2018 春季课程课件:http://gis4g.pku.edu.cn/course/pythonds/

数据结构与算法 _ 北京大学 _MOOC:https://www.icourse163.org/course/0809PKU015-1002534001

第二十四章　F17_X-ray 报告

宣泽远 *、余江晖、何鑫、许睿安、于之恒

摘要：我们的算法基于自己构建的基础设施模块（对地图信息进行了集成化的整合，构建了双方边界等信息），分为进攻、圈地、返回、穿刺四种模式（从基础指令实现逐层往上汇集功能，达到直接将函数组合为接近自然语言的目的，指挥函数实现不同战术），根据地图现有信息确定执行何种模式，从而构建出完整的 AI 算法。我们的整体思路是按照专家系统的思想人工确定 AI 执行逻辑并分模块加以实现。在参数的确定上并没有来得及应用用机器学习的想法训练算法，是我们的一点缺憾。本组的算法没有涉及到十分复杂的数据结构，而采用了启发式规则，故而时间复杂度基本能够控制在 O(1)。最终我们的 AI 取得了较好的成绩，在竞赛中成功打入了八强。

关键字：专家系统 基础设施构建　模式选择

1 算法思想

1.1 总体思路

1. 算法总体思路：人工启发下控制的多模块命令执行

启发我们的主要是专家系统这一思路，我们模拟人做出决策的过程，进行程序构架。

首先，我们选取的基础操作方式为：

选定模式——生成 order——维护 order

Order 指一串储存在 storage 中的指令，由生成 order 阶段产生，并指导接下来的步骤，当上一个 order 被完成（或被其他高优先级判断中断时），将会根据当前的模式选取调

用的模式函数，并重新生成下一步的指令，这一点大幅度节约了程序的运行时间，也保证了 AI 运动目标的一致性。

当一个 order 被执行时，主程序将会维护这个 order（比如判断原计划是否可行，是否可以扩展原计划，是否应该发动进攻等），并根据具体盘面情况作出修改操作。

我方纸带的运动分为攻击模式（attackmode）、圈地模式（basemode）、返回模式（backmode）和穿刺模式（stinmode），每回合根据目前我方所处的模式、我方与敌方到双方纸带和双方领地的距离来执行主函数逻辑判断，明确之后几步所要完成的基本操作（攻击对手、执行圈地、返回领地或者执行穿刺）并生成所需的指令。而模块间切换的时机的则由人工模拟得到的一些参数/上级函数来实现控制。

2. 采用的主要数据结构 & 算法策略

本组采用的数据结构比较朴素，主要为递归/线性的列表、元组、字典等；我们将主要的精力放在了战术策略研究与函数模块实现上。

在算法策略方面，我们以通常最大价值圈地、必要时穿刺来分割对手领地的圈地策略为基础，附加在对方处于必死状态和纸带过长状态时进攻、在己方遭遇危险时返回的模式。

并且，在针对不同策略的敌人时我方通过人工调整参数来实现相应的针对性策略：对于以广范围圈地意图包围我方的对手，我方将更频繁的切换至穿刺模式以切断对手的包围圈；对于以进攻为目的逼近我方的对手，我方将在圈地选择起点终点时更多的选在敌人的周围，以确保我方不会从对方面前逃开。

在这之中，人工启发式的调整参数来变更我方的策略是重中之重，通过我们对于该游戏的理解来引导 AI 的趋向是十分方便且比较准确的。

3. 从最初设想到实现的过程

我们最初的设想是从公用基础函数建设开始，不断搭建语言框架，在这些较高级有效的语言框架之上可以非常容易地编写灵活的战术模块，制订不同战法（mode）。

在实现过程中，全组同学先各自独立编写了初级的程序以熟悉操作、积累经验，而后在第一次组会中确定了 order 指令这一体系，分工三大模块（attack，base，back）进行编写（穿刺模块为后来根据实际战术需要加入的）。

宣泽远首先编写了这个框架下的一个实验性程序并参加热身赛。在获得了一定积累后，小组成员开始在由余江晖和宣泽远编写的基建函数框架内加入各自的模块，最后通过主程

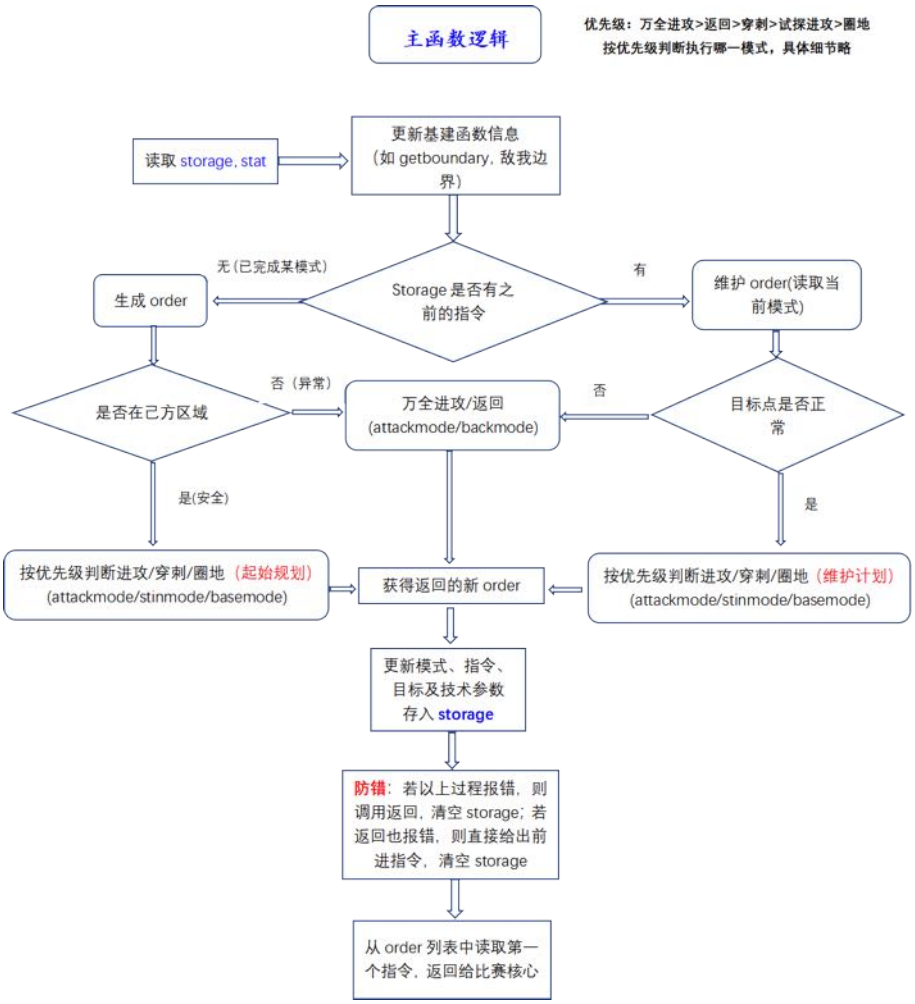
序逻辑将这些模块组合，形成了三个函数——X-Ray 终稿、X-Ray（针对纯圈地）、X-Ray（针对纯进攻），这三个函数即在该框架下不同战术的实现。

最后，小组成员不断在调试中改变不同模式行为的参数设置，以达到最好的效果。

略微遗憾的是，由于最后的淘汰赛中并没有给予我们复盘以观察对手的行为模式，我们无法针对对手选择合适的程序，除 X-Ray 终稿以外的两个函数并没能发挥作用。

1.2 算法流程图

每一模块的分图见 2.2 函数说明部分.



1.3 算法运行时间复杂度分析

关于算法的时间复杂度，我们主要从三个方面分析：

1，基建模块——基础数据更新

2，功能模块时间复杂度

3，剪枝操作、不同情况的开销不同

其一，在基建模块中，每次更新的是边界范围函数和边界函数。前者只更新己方最远边界点的坐标，时间可忽略；而后者相比大多数遍历所有点的求边界点程序则大大节省了运行时间，由于区域变动都发生在敌我圈地成功时，故只需此时将原有纸带的所在点遍历一遍。时间复杂度随前一时刻纸带长度而变，每回合的开销只有 O（n）量级。

其二，由于进攻、圈地、穿刺、返回的函数会因情况不同而选择不同的判断函数，故时间复杂度与盘面情况有很大关系，此处只给出平均意义上的时间。

1、进攻函数：

2000 回合的耗时大概在 1 秒左右，进攻分为两个步骤，能否进攻判断和进攻路线规划，在能否进攻判断的地方，有粗略判断和精细判断，粗略判断的时间复杂度为 O(n/5)，精细复杂度为 O（n），都在可控的范围内，并且是先调用粗略判断，在调用精细判断的。

在满足判断后，进攻规划的方面，有规划路线和路线判断两个方面进行耗时，在规划路线时，会生成三条路线，需要对路线上的没一点都检验，所以耗时为 O（3n），检测路线也同样要点点检验，会耗时，同样为 O(3n)，但总体而言，都属于 O(n) 数量级。

总的来说，进攻模块的调用为 O（n）数量级。

2、圈地函数：

2000 回合耗时在 3 至 4 秒，圈地分为两个模块，路线规划与路线维护。路线规划模块使用了几种遍历边界点集合、判断区域开闭的函数，由于基建函数中已经给出了边界点集合，并且使用了改良的探测开闭集合方式（时间复杂度在 O(n) 左右，可达到任意的精度，详见程序解释——smallrobotdetect），整体的时间复杂度平均在 O() 量级，并且由于路线规划不需要步步进行，使得耗时并不长。

路线维护主要是步步进行的危险判断，耗时也在 O(1) 量级。

3、穿刺函数：

2000 回合耗时在 2 秒左右，每一次调用时均判断，探测的时间复杂度为 O(n)，路线规划的时间复杂度为 O()，由于穿刺次数在整局中不超出十位数的量级，大多数判断都在极短时间内判否，故总开销被削减了。

4、返回函数

2000 回合耗时不确定，最低时在 1 秒以内，最多有过总耗时 10 秒附近的情况，由于返回函数是在出现异常和进攻未果的情况下调用的，在一局中一般只调用个位数的次数，但当对手与我方发生作用，出现较为激烈的交锋时，调用次数会上升。

就算法本身而言，由于返回模块使用了改良的路线规划，并不直接穷举可能路径、所有点。如果调用返回函数的当下位置离己方边界点只有一格的距离，亦即使用“贴边返回”以及“走出一步就返回”时，时间复杂度为 O(1)。如果不是上述两种情形，但是离边界点又不是太远的时候，时间复杂度也是 O(1)，因为无论场地多大，只会分析附近的 5x5 矩形。如果离边界点比较远时，时间复杂度大体上退化成 O(n^2)，也就距离边界点越远所需耗时越长，以一个相对快的增长趋势，其中还取决于己方纸带分布的复杂度。如果使用“All 模式”，所需耗时就不取决于与边界点的距离，是所有返回方式中最为耗时的一种。在实战情况之下，“贴边返回”以及“走出一步就返回”是最常被触发的，其次是搜索附近 5x5 矩形的方式，相对的“All 模式”极少使用，也就是说返回模块通常时间复杂度落在 O(1)，特别糟的情况变成 O(n^2)。

2 程序代码说明

2.1 数据结构说明

本组算法中采用的主要数据结构为线性数据结构，主要由 Python 基本数据类型构成。这里重点解释一下有关“以 order 指令控制 AI”这一基本想法的实现。

首先先要说明的是，我们每一个模块都不仅仅是为了规划下一步，而是为了执行一次进攻、圈地、返回而规划了接下来的一系列步骤。而这些规划的步骤都被我们输入到 order 指令列表：storage[‘order’] 中（这相当于一个队列），这是我们在 play 函数中对外界的唯一接口，即我们只会从这个队列里输出下一步的指令作为我方的函数返回值。

当需要进行模式变更时，我们也通过对 storage[‘order’] 的调整来实现：比如在执行模糊进攻当中判断本次进攻失败而需要执行返回模块时，我们会删除 storage[‘order’] 中

的指令并更改为返回模块所呈递上来的新指令。

这种控制 AI 的方式能够实现单一的信息输入、输出流，防止了计算浪费的出现。

2.2 函数说明

由于整个 play 函数太过庞杂，我们将对每一模块构建单独的流程图，并将其中重要且编写较为出色的函数源代码附在这里进行说明：

2.2.1 公用基础函数（宣泽远、余江晖）

goto(mode, startpoint, endpoint, direction, orient=’shun’)

——完成从一点移动到另一点的基础操作，输出一个 order 列表：

对各种情况（主要是不同的起始朝向所引起的恶果）进行了分类讨论，由于主要是对各种情况的具体探讨，故不在这里过多赘述。但事实上它大大简化了各模块思考和编写的难度，被广泛应用于我们的各个模块之中。

getboundary() ——每回合分别维护我方和敌方的领地边界：

该函数基于这样一点考虑：只有在双方中任一方圈地成功的回合才会发生边界点的改变。所以我们首先判断有没有一方圈地成功，再在圈地成功的回合通过圈地成功前的纸带来维护边界点。这是一个贪心算法，虽然仅针对本次任务，但确实有极高的效率，相较于其他穷举算法，大幅节约了时间。

i. 判断有无圈地成功：

```
#如果我方在此回合圈地成功
if not storage['mybands'] and storage['former_mybands']:
    myfield_edge = myfield_edge + storage['former_mybands']
```

这里我们在 storage 中维护一个我方实时的纸带集合和我方上一回合的纸带集合，当我方此时刻纸带集合为空（storage[‘mybands’] 为空）但上一回合纸带不为空（storage[‘former_mybands’] 不为空）时，标志着我方本回合圈地成功。同样的，敌方是否在本回合圈地成功用随时维护的敌方纸带信息 storage[‘enemybands’] 以及 storage[‘former_enmybands’] 来判断。

而纸带信息的具体维护则通过一个专门的函数实现，每回合判断头的位置是否在己方

领地内，如果不是则将头的位置添加进纸带集合之中。

ii．如果我方圈地成功对我方边界点（myself_edge）的维护：

首先将我方上一时刻的纸带集合点全部添加到边界点集合中，我方目前真实的边界点集合应该已经被完全包含在内。

```
myfield_edge = myfield_edge + storage['former_mybands']
```

之后再判断每一个集合中的点是不是我方边界点。

```
for i in myfield_edge:
    #如果i的每一边不是边框就是我方地盘，则不是边界
    if i[0] == 0 or isInbound((i[0] - 1, i[1]), me):
        if i[0] == MAX_W - 1 or isInbound((i[0] + 1, i[1]), me):
            if i[1] == 0 or isInbound((i[0], i[1] - 1), me):
                if i[1] == MAX_H - 1 or isInbound((i[0], i[1] + 1), me):
                    deleteList.append(i)
```

最后删去需要删除的点。

```
#当deleteList不为空时，将其中最后一个坐标点从我方边界中删去
while deleteList:
    point = deleteList.pop()
    storage['boundary'][point[0]][point[1]] = 0
    myfield_edge.remove(point)
```

这样就完成了一次对我方边界点的维护。完全相同的，我们可以对敌方圈地成功对敌方边界点（enemyself_edge）做如上方式的维护，这里不再赘述。

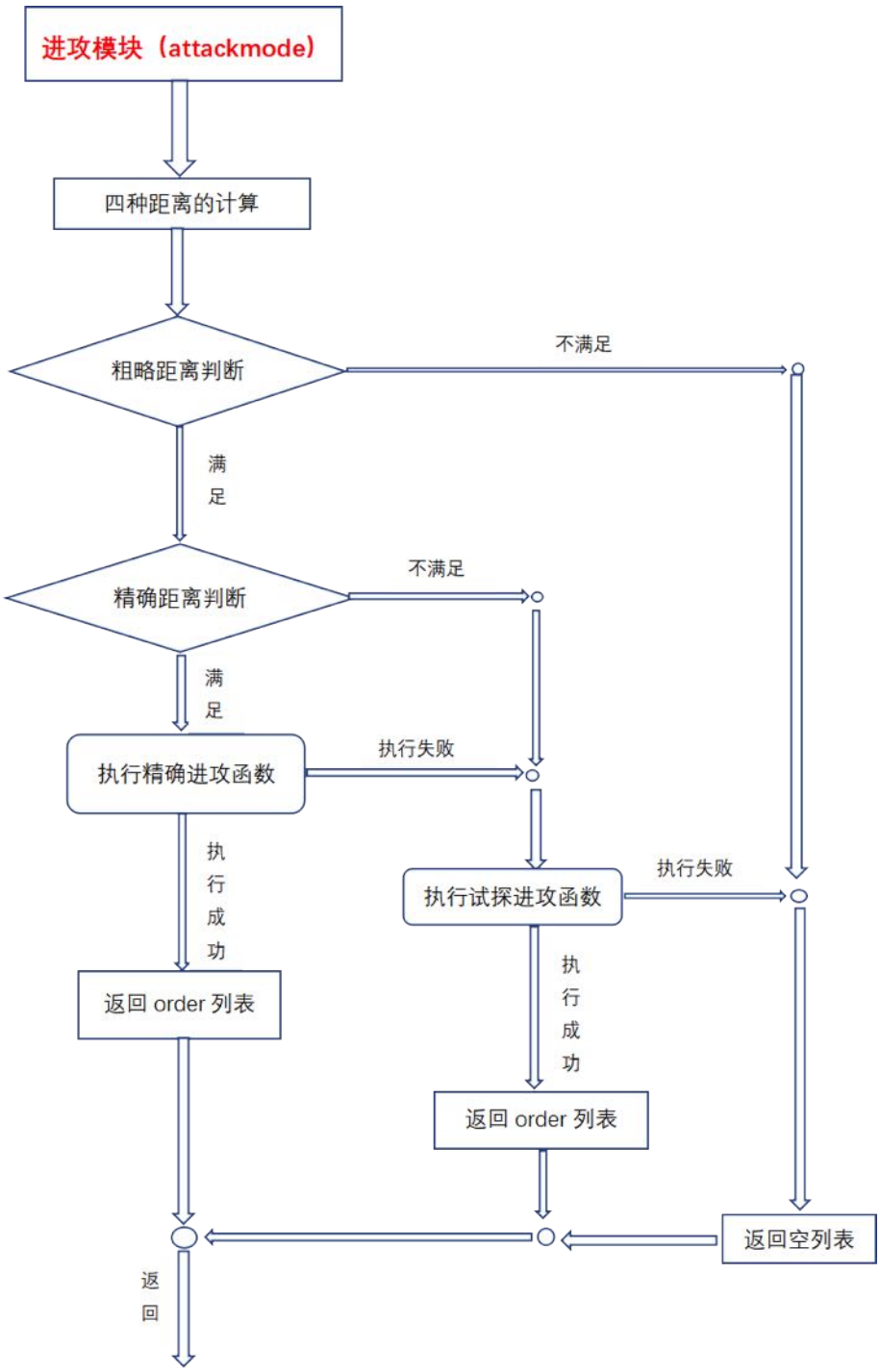
iii．如果敌方圈地成功对我方边界点（myself_edge）的维护：

首先我们发现当敌方圈地成功时只有敌方圈地前形成的纸带的周围四个点有可能是我们的新的边界点。故而我们先得到敌方上一回合的纸带集合，对其中每一点的周围四个点进行判断，如果是我方的点则是我方的边界点。

```
#若我方边界有修改，对于敌方纸带上的每一点，若周围有我方领地则为我方边界
if mychange:
    for j in storage['former_enemybands']:
        if isInbound((j[0] - 1, j[1]), me) and (not (j[0] - 1, j[1]) in myfield_edge):
            myfield_edge.append((j[0] - 1, j[1]))
            storage['boundary'][j[0] - 1][j[1]] = me.id
        if isInbound((j[0] + 1, j[1]), me) and (not (j[0] + 1, j[1]) in myfield_edge):
            myfield_edge.append((j[0] + 1, j[1]))
            storage['boundary'][j[0] + 1][j[1]] = me.id
        if isInbound((j[0], j[1] + 1), me) and (not (j[0], j[1] + 1) in myfield_edge):
            myfield_edge.append((j[0], j[1] + 1))
            storage['boundary'][j[0]][j[1] + 1] = me.id
        if isInbound((j[0], j[1] - 1), me) and (not (j[0], j[1] - 1) in myfield_edge):
            myfield_edge.append((j[0], j[1] - 1))
            storage['boundary'][j[0]][j[1] - 1] = me.id
```

这样就完成了对我方边界点的维护。完全相同的，当我方圈地成功时亦可以对敌方边界点做如上方式的维护，这里同样不再赘述。

2.2.2 attackmode（何鑫、于之恒）

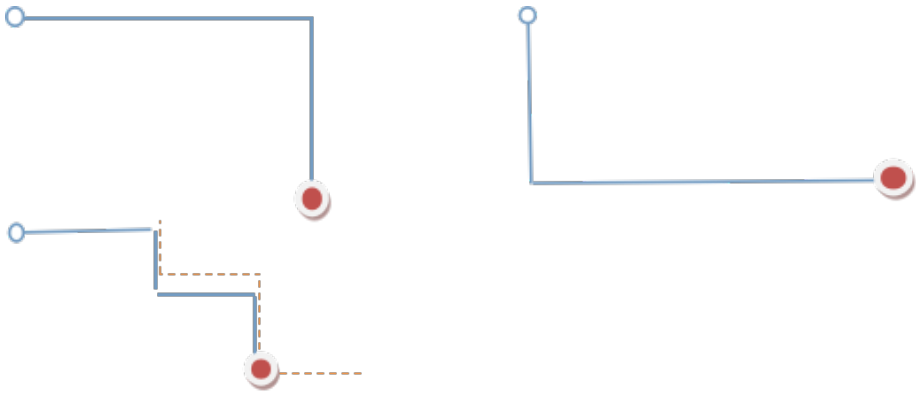


函数中的主要思路是通过计算进攻，回领地的路线等方法判断能不能进攻，并返回进攻的模式和路径，同时检测危险，有防守的作用，实现的具体思路如下：

第一步，模糊判断，因为每一次行动前都需要调用精功函数的模块，所以先通过间隔遍历的方法，对纸带和边界点不完全的遍历一边，在节省时间的同时大概的计算出一个对方的回领地距离，我方的回领地距离，对方攻击我方纸带的距离，我方攻击对方纸纸带的距离，如果可以攻击，就进入下一层判断。

第二步，精确判断，在完成了模糊判断之后，大致上是可以进行攻击了，此时，可以调用需要消耗许多时间的精确判断模块。调用精确判断的思路与上一步一样，不同的是间隔式遍历改为逐点遍历纸带和边界点，这样会消耗许多的时间，但考虑到调用的次数并不多且进攻的风险性很大，所以觉得这样还是可以接受的，然后就是一组判断，对方的回家距离要高于我方的进攻距离，对方的进攻距离要小于我方的回家距离。这样就可以说是可以进攻了，于是进入第三步 -精确制导，如果不满足，考虑到大多数的函数其实并不完美，那么进行试探性质的进攻

第三步：精确制导和试探性进攻，在完成了距离的判断之后，如果可以完美无缺的进攻，就执行精确进攻，精确制导的过程中，除了要通过算法找一条路之外，还要判断己方的进攻路径会不会被对方拦截，，在确认不会之后，才会进行真正无敌的“进攻模式”。返回进攻的指令，开始屠杀。而在不满足精确制导的情况下，会尝试性的进行试探性进攻。这个阶段，会检测己方的进攻距离小于对方回家距离 +5，己方的回家距离小于对方进攻距离 +5，这样可以增加进攻的机会，与此同时，在试探性进攻中，会点点检测“危险”，即干不掉别人的情况下还回不了家。这时会调用回家的函数，这样就能确保无虞了。



这是进攻模块找路径的三种方法，会先尝试从上面找笔直的进攻路径，然后找下面的进攻路径，最后找一条受限制的路径，橙色线区域即是不可到达点，得绕开才行

下面是本模块中所使用函数的具体介绍：

1. 七个计算距离的函数

（1）def mydis_to_home_0(stat, storage):

（2）def mydis_to_home_1(stat, storage):

（3）def enemydis_to_home_0(stat, storage):

（4）def enemydis_to_home_1(stat, storage):

（5）def attack_rough_me(stat, storage):

（6）def attack_rough_enemy(stat, storage):

（7）def attack_fine()

（1），（2）均为计算自己的头回到自己区域的距离。（3），（4）均为计算敌人的头回到他的区域的距离。（1），（3）为对粗略距离的计算。（2），（4）为对精确距离的计算。（5），（6）分别为我方进攻敌方和敌方进攻我方的粗略距离。（7）为对我方进攻敌方的精确距离。

这七个函数均以 stat,storage 作为参数。（1）（3）（5）（6）的返回值均为一个距离，（2）（4）（7）的返回值为一个距离和一个元组（目标点的坐标）

计算精确距离的方式为遍历所有可选点，算出其与目标点的距离。计算粗略距离则是用同样的方式，但是改为在可选点中每隔几个选一个点，以减小计算量。

注意点：计算的粗略距离与精确距离要满足一定的放缩关系，如自己的粗略进攻距离与精确距离相比只能放大，敌人的粗略进攻距离只能缩小。当初始点与目标点在同一条直线上时，若初始方向与行进方向相反，距离要加 2（需要多转两次）。

2. 两种规划进攻路线的函数

（1）def y_side_attack(y_me_finecoordinate, y_me_x, y_me_y):

（2）def h_attackway(myhead1, target1, myband1):

这两个函数均为已知初末点，规划进攻路线的函数。（1）的参数为初末点，（2）的参数为初末点和当前的纸带集合。返回值均为一个列表，列表元素为进攻路线的点坐标。

在路径上没有自己纸带的时候使用函数（1），（1）的路径为先沿着 x 方向，再沿着 y 方向或者是先沿着 y 方向，再沿着 x 方向。在路径上有自己纸带的时候使用函数（2），（2）的路径为绕开自己纸带的最小距离路径。

注意点：这两个函数所生成的列表中的坐标可能有重复的，这时用相关的去重函数对这个列表进行去重处理。该列表不包含初始位置，包含末位置。

3. 从已有路径中选出最优路径的函数

def enemy_intercept(road1, road2, road3, enemyhead):

该函数以三条已算出的进攻路线和敌人头的坐标作为参数，返回这三条路径中的最优路径和相对应的敌人拦截距离（拦截距离为敌人在它当前位置进攻我方进攻路经上点的距离）。

通过遍历每条路径上敌人的进攻距离得到该路径的拦截距离，算出三条路径的拦截距离，选择其中最大拦截距离的作为最优路径。

注意点：若该点在自己的领地内，则敌人无法进攻这点，无需计算这一点的拦截距离，其他时候都需要计算。

4. 精确进攻函数

def attack1(stat, storage, my_attack_pos, my_attack_dis1)

执行精确进攻（一定能进攻成功），参数为进攻目标点和进攻的精确距离。返回值为可行的路径的点列。

先找出三条可能的进攻路径。并找出其最优路径，判断敌人的拦截距离与我方的进攻距离的关系。若拦截距离比较大，就说明该路径可行，就返回对应的列表。若拦截距离较小，就说明该路径不行，返回空列表。

5. 试探进攻函数

def attack2(stat, storage, my_attack_pos, my_attack_dis, my_re_dis,enemy_attack_dis, enemy_re_dis)

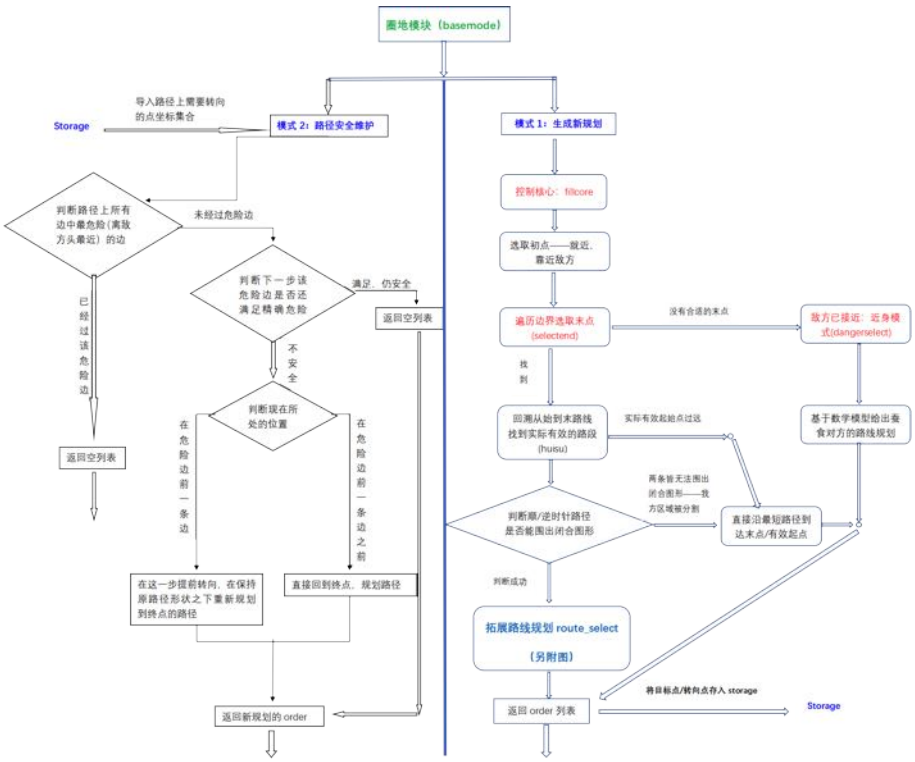
执行试探进攻，参数为我方的进攻点，进攻距离，回家距离，敌方的进攻距离，回家距离。返回一个指令列表和一个表示当前进攻模式的字符串。

由于已经不满足精确进攻，不能一定撞死对手。但考虑到对手可能在我方试探进攻时没有及时采取应对的措施而被有机可乘。相当于在有一些机会的时候自己主动制造进攻机会。先找出三条可能的进攻路径。并找出其最优路径，判断其是否满足试探进攻的条件，若满足，则返回对应的指令列表和模式；若不满足，则返回空列表。

遗憾：由于试探进攻具体实现上的困难，我们未能写出完全符合预期的函数，可能出现的失误会导致进攻将自己陷入危险之中，所以最后的代码中对试探进攻的触发条件限制很大，以至于它的功能实现的较少。

注意点：该函数需要每次执行，以判断最新情况下的策略和路径规划。

2.2.3 basemode（余江晖、宣泽远）



圈地模块 (basemode) 主要分为路线规划与安全维护两个模式。

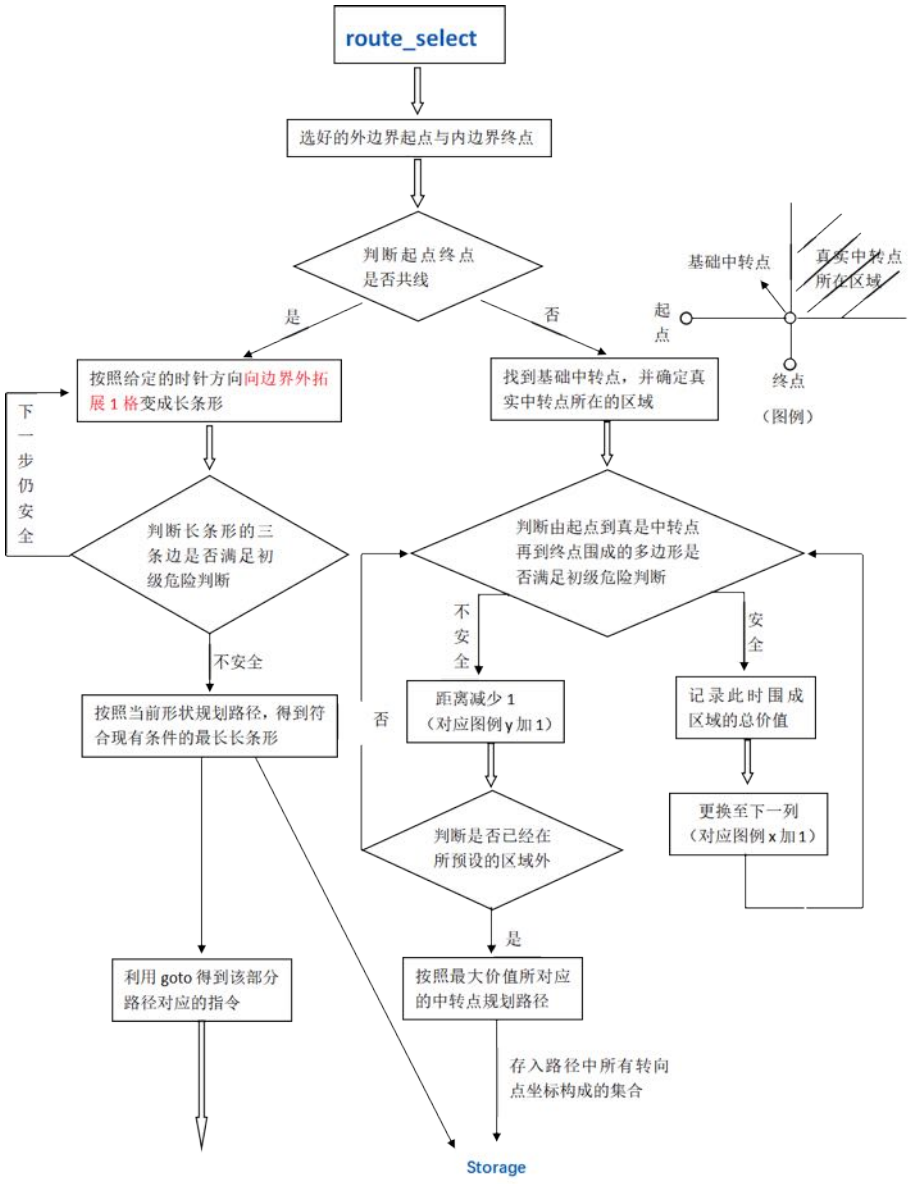
I．当前接收到的指令为重新规划路线时，会进入以下步骤:

选取初点 (fillcore 函数内置)，在就近范围内选取合适初点，并以此为起点执行选取末点的操作。遍历己方边界点，通过粗略判断由初点和目标点构成的方形中有效圈地的面积 (detecvalue 函数)，粗略给出待选的末点，这样的选取保证了从初点直接走到末点是安全的，并且使得程序能够敏锐地发现“凹”的图形。接下来对基础路线进行扩展和修正。

另外，当无法选到合适的末点时，说明需要进行向外的拓展。另外，如果敌方已经逼近，会调用近身选点 (dangerselect 函数) 进行缠斗。

在已经选好初末点后，将进行拓展路线规划部分（route_select 函数）以求更大的圈地价值（在这里我们把敌方领地、空地、己方领地的价值分别设置为 3、1、0）。具体的步骤实现见下页附图及说明。

附图：route_select()——拓展路线规划



在上述流程图中整个路线规划过程已经讲述的比较清晰了，这里我们强调三个细节：

第一是最终路线的形状，在起末点共线时如图一所示，不共线时如图二、三所示：

第二是不共线时对于最优真实中转点的选取。首先可以理解，每一个真实中转点都对应一条路径，而这条路径有对应着一个圈地得到的价值，所以问题其实就是找到最大价值圈地对应的中转点。在具体实现中，我们采用了先预设一个能拓展的最远距离，然后在每列中寻找可能拓展的最远位置（此时必然是该列中具有最大价值的）最终确定最优点的方式，这要比遍历真实中转点所在区域要强的多。

第三是对初级危险判断的解释：这里我们所做的并不是比较敌人进攻的距离和我方规划中剩余距离来规避进攻的“精确危险判断”，而是放宽条件到了敌人进攻可以比我方剩余步数少敌我头距离的二分之一，从而争取增大单次圈地的面积。

II. 当前接收到的指令为路线安全维护时：

由前所述我们知道，route_select 得到的路径是不能确保安全的，所以需要配套使用一个能够动态实时地判断安全与否的“精确危险判断”，即是这个路线安全维护（danger_detect 函数）。下面是具体流程：

首先从 storage 中调取当前路径中的转向点集合（在每次执行 route_select 时会将最后得到的转向点集合存储至 storage 中），从而确定路径上每条边的位置并找到距离敌人最近的边，我们称之为危险边（下图红色边），而当前我方所处的边为黑色边。

当我们处于图一的情况时，称为处于危险边的前一条边，在判断下一点出现危险时会提前转向，走蓝色虚线以代替红色危险边；当我们处于图二的情况时，由于情况会比较复杂，我们在下一点遇到危险时会选择走蓝色虚线直接返回终点。

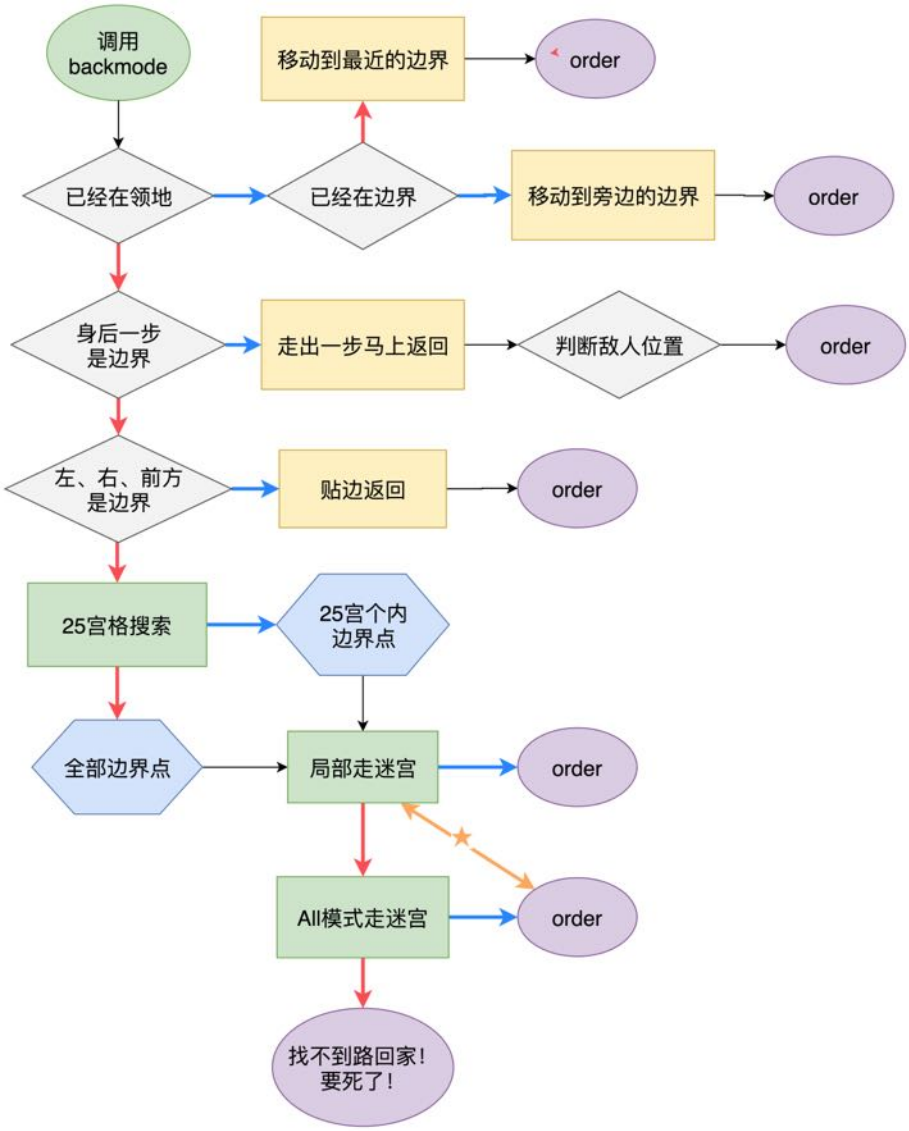
可以看到，这种操作尽可能地保留了原先规划的路径形状，事实证明这样能够获得相对较好的区域价值。

当然，这之中会出现危险边不止一条的情况，我们都将想办法把它化归为一条危险边

的状态来处理。

2.2.4 backmode（许睿安）

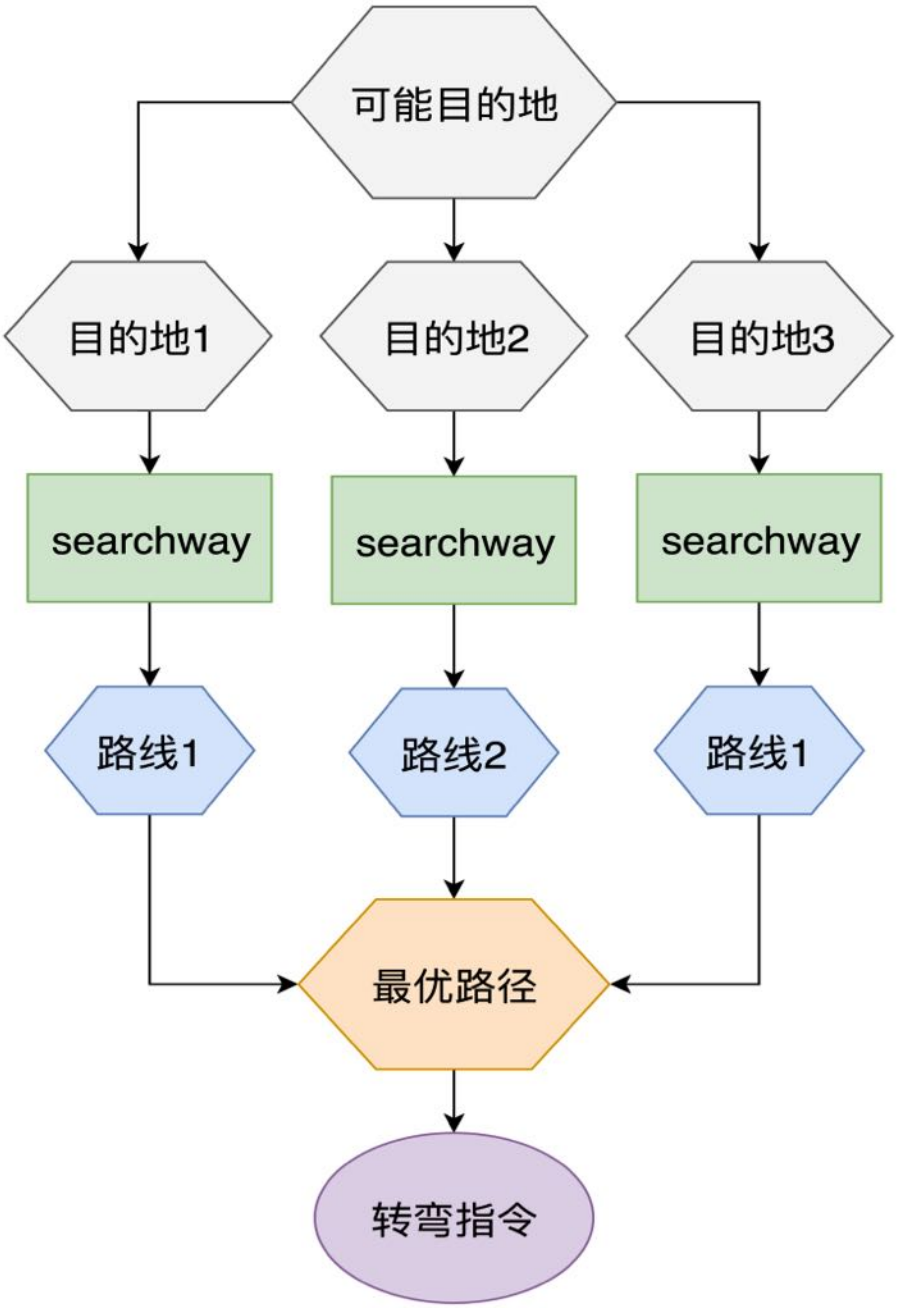
I. 总体流程图



流程图中红色箭头表示“True”、蓝色箭头表示“False”。返回方式一共有 4 种，优先级由高到低为排列为：“走出一步就返回”、“贴边返回”、“非 All 模式（局部走迷宫）”、“All 模式走迷宫”。除此之外，还有当调用返回模式时已经在己方领地的应变方式。其中“All 模式 1 走迷宫”产生的“order”必须再经过修正，在流程图上表示为“黄色星

号箭头”，详细说明放在后面。

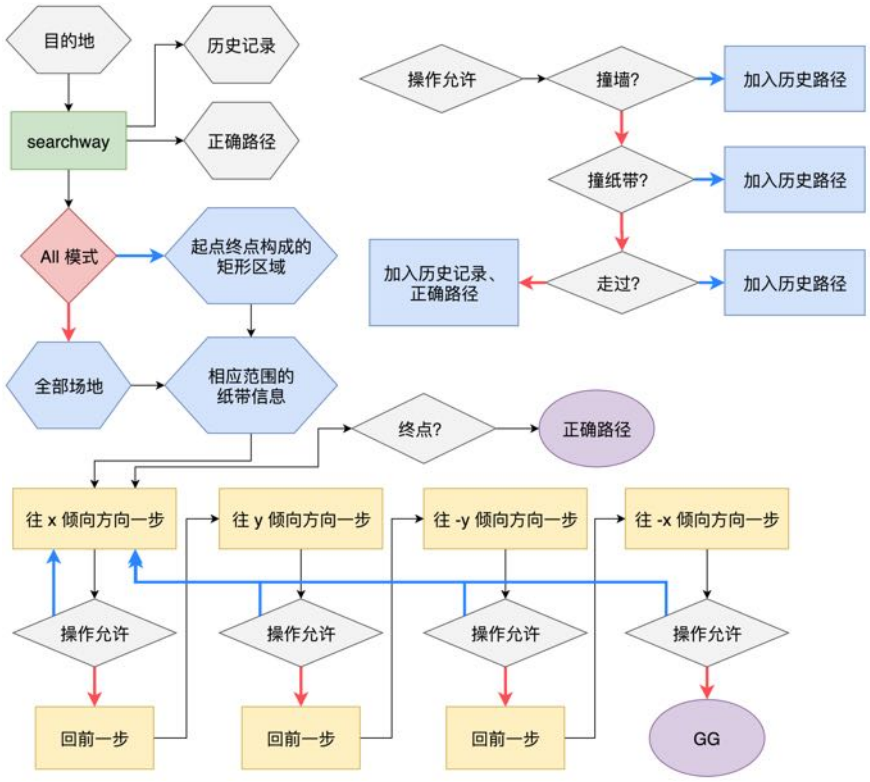
II．寻找最近边界点流程图



流程图中红色箭头表示“True”、蓝色箭头表示“False”。将待搜索的边界点逐一规划一条最短路径，之后再从其中挑出距离最短的边界点及其路径，最后，使用“坐标 -转向转

换函数”，将一串连续的位置坐标配上初始方向转换成转向指令。

III. 搜索路径流程图

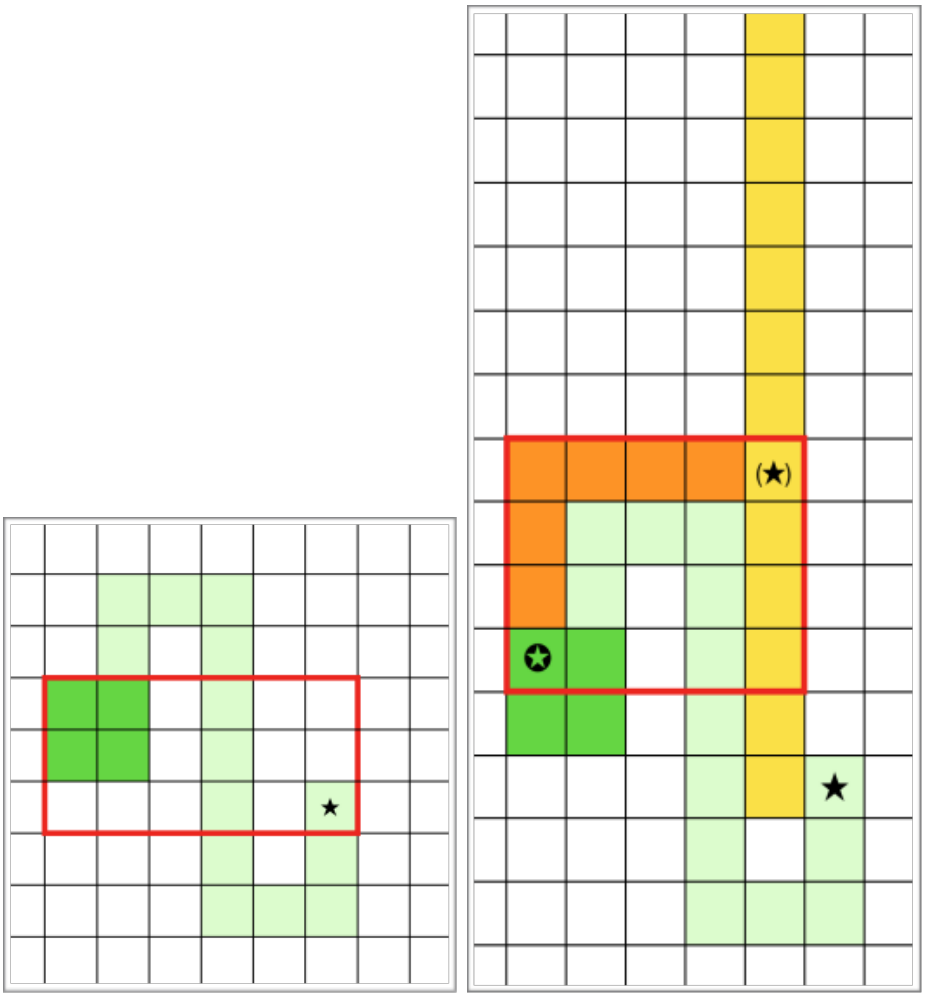


流程图中红色箭头表示“True”、蓝色箭头表示“False”。寻找從给定位置到指定终点的一条不会撞上自己纸带也不会撞墙的最短路径。主要思路为，每次首先尝试前往终点的方向，如果该方向可行，继续尝试该方向；如果不可行，退回原位置，尝试另一方向。重复直至抵达终点。其中不可行的依据就是使用“判断操作可行性函数”，当撞墙、撞纸带、或是该点走过了，都会判断不可行。具体实践上采用 2 个变量“历史记录”和“正确路径”，只要走过便将该点加入“历史记录”，如果该点不可行则在“正确路径”中删除该点。如果没有使用图中的红色“All 模式”，将调用场地切割函数限定搜索范围，反之则全场搜索。

IV. 场地切割函数和“All 模式”修正说明

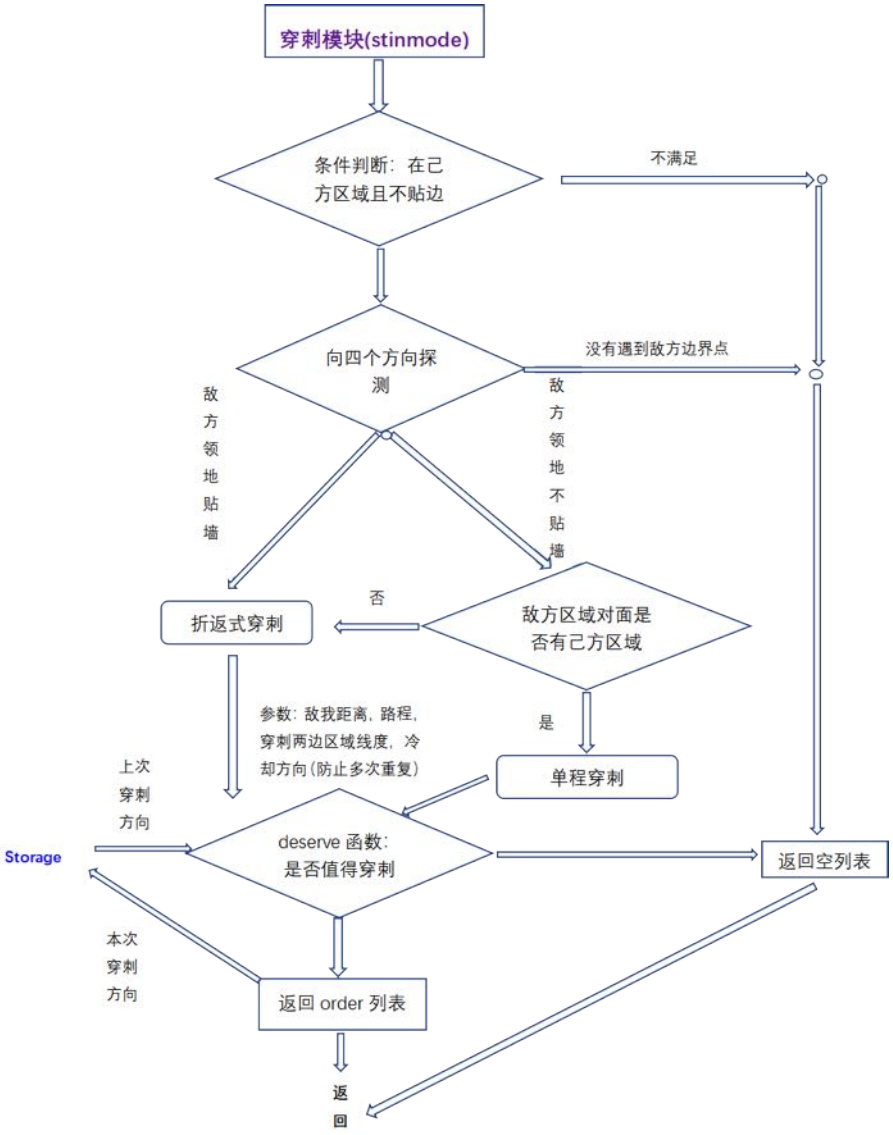
将含有全场纸带信息的列表切片，以起点、终点为矩形的 2 个对顶点。搜寻路径时仅考虑切割后的矩形范围，亦即规划路径中不能超出矩形区域。如果两点 x 值相同，取宽为 3 的矩形。如果两点 y 值相同，取高为 3 的矩形。切割的目的是降低数据大小，加快运算

时间，避免多走了一些不必要的步数。缺点是在某些极为特殊的纸带分布情形，切割部分矩形的方式将无法找到任何一条路径回到边界点。例如左下图情况。浅色区域为己方纸带，深色为己方领地，星号表示己方纸带的头。图中情况如果使用“场地切割函数”，最大范围也只会切割出红色矩形的范围，然而在此区域内，是不存在一条不撞上己方纸带的路径回到己方领地，但是这并不代表无法返回，只是因为强制切割造成的缺陷，如此之下就不能使用切割的方式，而是必须把全场的纸带信息都考虑进路径当中。如右下图所示，黄色为“All 模式”下规划出的路径，因为没有限制范围再加上算法中方向优先级的原因，他会不断前进直到碰到墙才转弯。所以为了修正路径，在黄色路径上的每一点都使用“非 All 模式”再规划一次路径，如图中所示，在括号星号的位置成功利用“非 All 模式”规划出橘色的路径，然后保留前半段的黄色路径，并修改后半段路径为橘色路径。



如此完成“All 模式”的路径修正。

2.2.5 stinmode（宣泽远）



穿刺模块主要包含分割和局部包围这两方面。

分割：由于圈地为导向的函数大多采用了在周边包围对手，形成闭合图形的战术，分割这一策略即基于贯穿对手领地、使得己方领地与空地/墙连通，以达到更高的圈地效率和彻底的分割。

局部包围：当敌方侵入我方领地时，会形成一种类似于“口袋”的凸起结构，一旦从

敌方的后部连通我方的区域，则会构成闭合图形而将对方区域重新夺回，并使得对手处于只有纸带的危险状态，这一点在近战缠斗时会使己方具有很大的威胁性。

在具体实现上，当我方头在自己的领地中时，便会触发穿刺判断，向四个方向探测敌方边界点。一旦遇到，判断其后是否有己方边界，以确定是往返贯穿还是局部包围。

当目标确定，会生成路线规划以评估危险性和价值比 (deserve 函数)，当满足条件时，触发穿刺。

另外，由于多次在同一方向穿刺是不必要的，我们引入了穿刺冷却——前一次的穿刺方向会被储存在 storage 里，在一定回合数内，不会再考虑相同方向的穿刺。

2.3 程序限制

由于每一个模块都内置了防错操作（针对意料之外的情况如何应对），以及主函数采用了 try-except 式结构防止任何编写上的 bug，可以认为我们的程序不会因报错中断比赛。

同时，动态规划圈地路线使对手永远不可能对我方纸带产生威胁

唯一可能被对手撞击情况是在死局/返回模块报错时，会返回直接前进的指令。

3 实验结果

3.1 测试数据

实验环境说明：

- 硬件配置：cpu：i7-7700HQ 内存：DDR4-2400 8G
- 操作系统：windows10
- Python 版本：3.6

测试方法 1：左右互搏

过程：本组在已经完成了终稿的情况下，又编了两套不同的程序，一共三套程序，分别应对纯进攻的函数，纯圈地的函数和攻防兼备的函数，对每套程序进行了总计约 60 场的“循环赛”模式的测试。

结果：无一例外都是跑到了 4000 回合，圈地的效果在不同的程序之间也有极大的差异。且当应对特化的函数时，最初的平衡的 X-Ray 终稿更容易在圈地上占下风。

测试方法 2：人机对战

过程：在已经完成了主体函数的情况下，采用人类智能的方法，与三套不同的程序进行对战，人类以灵活的思路瞄准程序的弱点采用包围的方式进行作战。

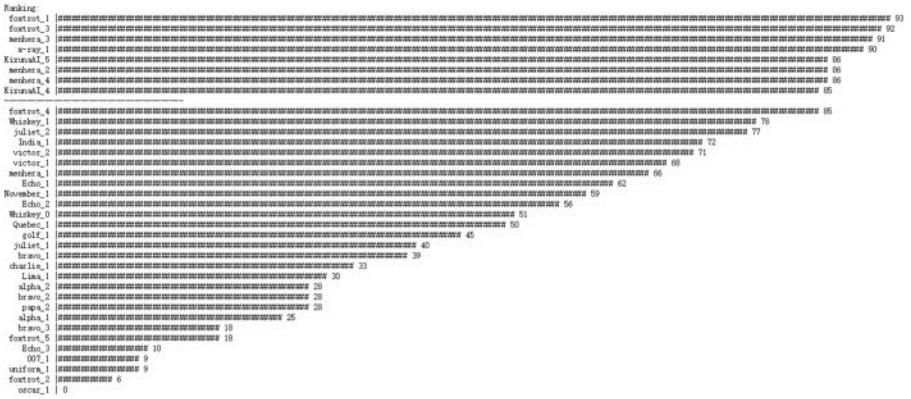
结果：人类大多数情况下会由于失误被 ai 干掉，但是在能达到 4000 回合的情况下，常以圈地的大优势取得胜利。

测试模式 3：热身赛

第一次热身赛：我们小组的两个代码与另外一个小组并列第一

第二次热身赛：我们小组的代码还在攻坚阶段，使用的是上个版本的代码，且由于平台未安装 numpy 库而频繁报错，最终处于中下

第三次热身赛：基本上是与参加竞赛的代码一致，排名在第四，只输给了 foxtrot 组的代码



3.2 结果分析

首先从算法整体战术上来说，由于我们在最大化圈地之中拥有 basemode 的多重危险判断外加 backmode 的快速返回能力，故 X-Ray 几乎没有被对手在 4000 回合内击杀的情况；同时由于我们加入了 attackmode 进攻模块，使得我们能够把握住多数一击必杀的机会赢得比赛。这是我们能够在小组赛中脱颖而出的基础。

其次就是在圈地中加入 stinmode 穿刺模块的局部思路。在最后两次热身赛中，我们看到很多组都采用了广范围搜索圈地意图包围的战术思路，而当我们加入了 stinmode 以后，X-Ray 得以在不疲于追赶在外围的敌人的基础上不断地破坏敌方的区域连通性，从而使得敌方的广范围圈地无功而返，我们可以通过 3.3 经典战局看到这一点。

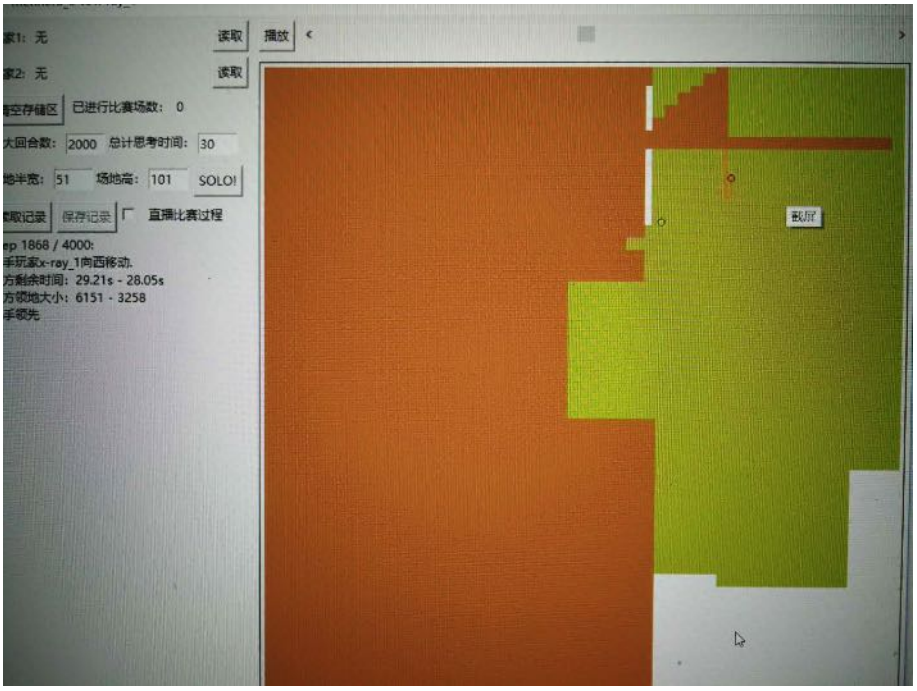
此外，我们还针对热身赛排名靠前的小组量身定做了 X-Ray 的两个特化版本，预期会有比较好的效果，但非常可惜的是它们没能有出场的机会。

最后在耗时方面，由于我们极少使用搜索等时间复杂度高的算法，故不用担心这方面的问题。耗时最长的模块一般为返回模块，这也是由函数要求精度高的特点所导致的。

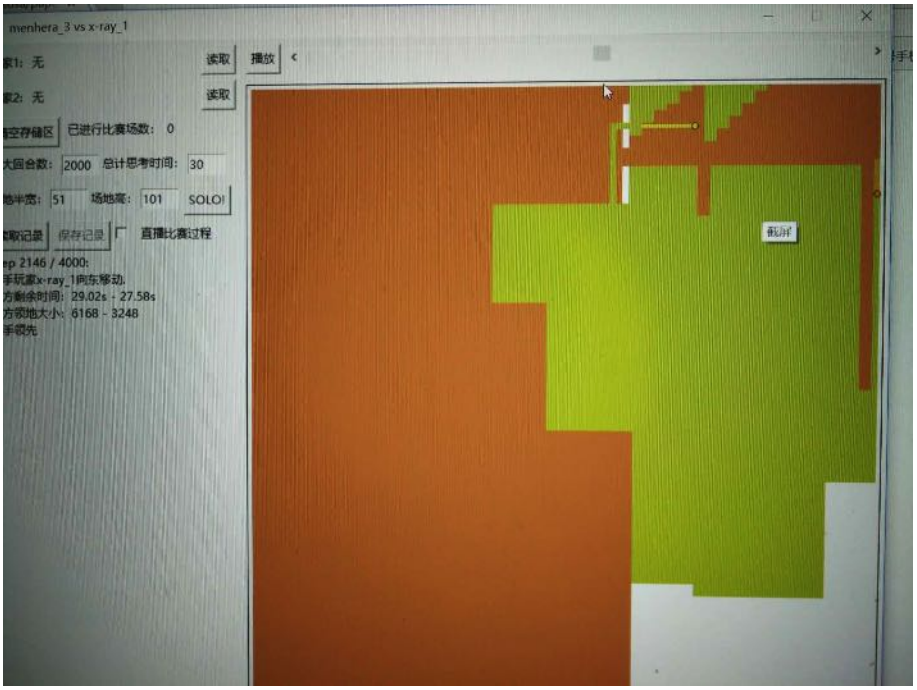
3.3 经典战局

第三次热身赛与 menhera 的对局中，有一局反包围来反败为胜的对局，

对局过半时，处于大劣势，对方开始尝试包围：



而后我方开始利用穿刺和针对性选点来展开精彩的反包围：



最终赢得了比赛（可以看到穿刺对敌方区域连通性的破坏）：

4 实习过程总结

4.1 分工与合作

4.1.1 基本情况与个人的工作量说明

本组成员：宣泽远 *、余江晖、何鑫、许睿安、于之恒

合作与交流的方式：

I. 以 38 楼 235 寝室为平日主要讨论地点（这里有三位组员）

II. 通过微信群和 coding.net 网站搬运代码源文件进行共同作业

III. 以每周 1-2 次的频率在二教预约教室开展小组集中会议

分工：

1. 算法开发与编辑：战术思考上群策群力；基础代码实现上由宣泽远、余江晖实现公用基础函数模块与圈地模块（其中包含有穿刺模块，由宣泽远完成），何鑫、于之恒实现攻击模块，许睿安实现返回模块。同时组长宣泽远负责总览全局，控制整个大作业的进程。

2. 小组报告：余江晖负责各次组会中的会议记录；余江晖与宣泽远按照报告模板拟定大体框架并完成主体部分的制作；何鑫、许睿安、于之恒针对各自负责的模块整理出该部分的运行逻辑及函数的介绍，并对本次小组大作业给出自己的感想。

个人工作量说明：

何鑫：

进攻模块：四个计算距离的函数（自己和敌人回家的距离），已知初末点规划曲折进攻路线的函数，从已有路径中找到最优路径和其被拦截距离的函数，将坐标点列转化为指令点列的函数，精确进攻函数，粗略进攻函数。

其他：跑主程序代码，找 bug，看与其他组的比赛复盘。

于之恒：

进攻模块：四个计算距离的函数（敌我双方的进攻距离的模糊和精细），已知出点和末点，规划两条直白的路线，分别向上和向下。对于计算好的路径，对路径的检验和返回合格路径。进攻模块的主体函数，模糊进攻和精确进攻函数，校验函数。

其他：观看比赛录像，人工 debug，跑主程序调试。

余江晖：

公用基础函数：

寻找边界点集合的 getboundary 函数及附加的功能函数

圈地模块：

确定起末点后的路径规划函数 route_select、通过危险判断对已规划路径的

维护函数 danger_detect，及附加的功能函数

针对性程序：

X-Ray 特异化（强化进攻）

其他：

历次组会记录，大作业报告编写，参与战术制定，

主力 debug（因为写的函数出的 bug 最多（沮丧））

许睿安：

改写递归实现的走迷宫算法，减少调用函数的开销；

实现不撞己方纸带并以最短路径返回领地的搜索；

改进算法复杂度，选择切割部分场地，并且优先考虑周围不可能撞上纸带的范围，先从这个区域内寻找边界点；

解决搜寻最短路径时方向优先级的判别问题；

针对频繁调用的情况，各别编辑返回算法，降低算法复杂度；

改进返回模块，完善调用时已经在己方领地点应对方案。

宣泽远：

试验性函数：全部内容

最终程序：

程序框架思路：order 体系，主函数逻辑，组织分工体系

基础设施建设：

goto 函数（strict/free 模式），用于从任意方向不碰壁地到达全盘任一点

各类小的转换、距离计算函数

basemode（圈地模块）：

粗略、精细价值评估，末点选取函数 (detectvalue/valuepiont/selectend)

路线规划函数中的开闭判断，危险判断函数 (isopen/smallrobotdetect/predangerdetect)

圈地模块控制核心逻辑 (fillcore)

stinmode（穿刺模块）：

全部内容，包括穿刺危险评估、价值评估、包围探测、穿刺冷却、开闭判断等（具体函数略）

主函数控制逻辑：

全部内容 + 额外防错机制

针对性程序：

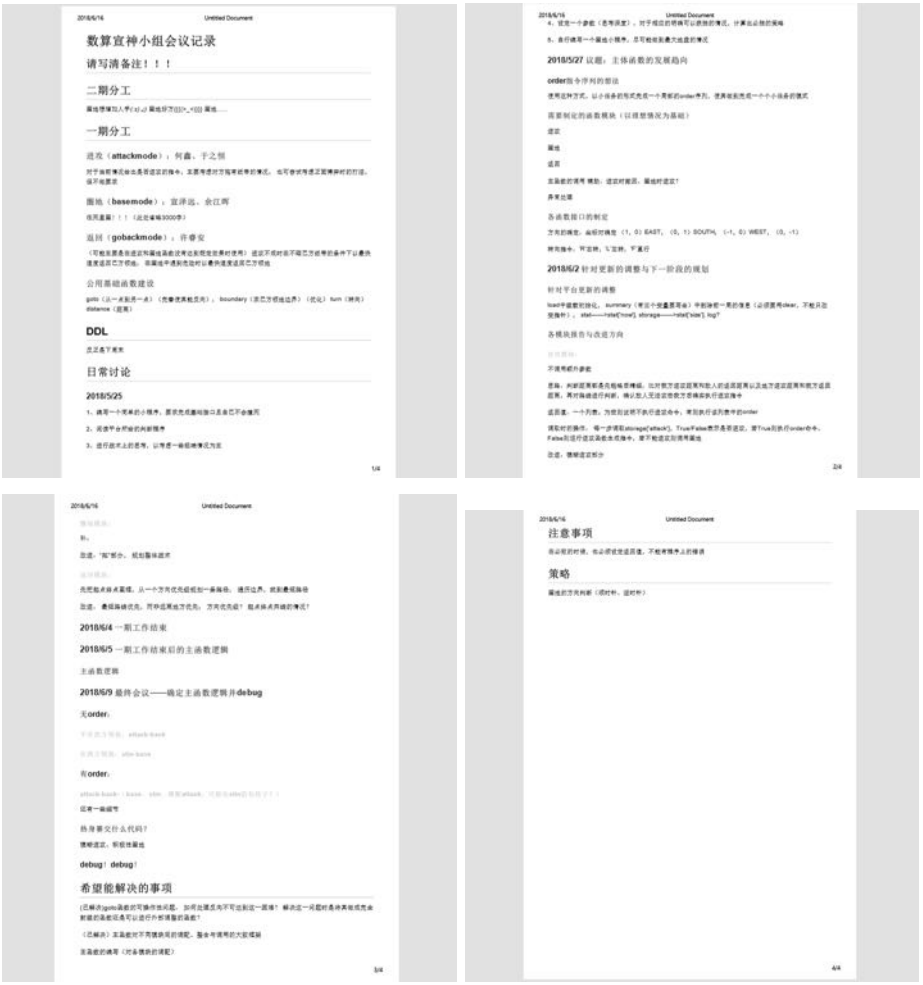
改写参数和判断逻辑：X-Ray 对纯圈地/X-Ray 对纯进攻

其他：

组织工作、大作业报告、debug（真是难 (kong) 忘 (bu) 的回忆。。。。。。）

4.1.2 历次组会记录

首先附上最终的会议记录图：


数算宣神小组会议记录

请写清备注！！！

二期分工

一期分工

进攻（attackmode）：何鑫，于之恒

圈地（basemode）：宣泽远、余江雨

返回（gobackmode）：许睿安

公用基础函数建设

DDL

日常讨论

2018/5/25

2018/5/27 议题：主体函数的发展趋向

order指令序列的思路

2018/6/2 针对更新的调整与下一阶段的规划

2018/6/4 一期工作结束

2018/6/5 一期工作结束后的主函数逻辑

2018/6/9 最终会议——确定主函数逻辑并debug

无order：

有order：

debug！debug！

希望能解决的事项

注意事项

策略

I．小组初建 5.22 理教 208 数算课上

本年度数算大作业“纸带圈地”开始组队，我们小组终于在此时成立了！部分组员（主要是 235 宿舍的三人）私下的膜法较量之后，最终决定由膜法值最高的宣泽远同学担任本组组长，而鉴于他本人比较聚的事实，其他组员对此决定没有异议。在这之后，我们本

着佛系选择的心态，选 (xia) 取 (xuan) 了 X-Ray 的组名与小组头像（来自某次失败的延时星空摄影）。最终，星空下弯曲的 X-Ray 小组就此诞生了。

X-Ray 小组头像

II. 第一次组会 5.25 地学楼 321

组长宣泽远对大作业基本情况以及前期大家玩 paper.io 手游得到的经验进行总结，并提议仔细阅读平台所给程序中的主要判断逻辑。

余江晖在 coding.net 上创建了项目，并提供了本组的大致工作思路：用模块搭建的方式构建 AI 算法逻辑，并计划使用一些参数来控制各模块的切换。

大家对战术进行了一些讨论后决定脚踏实地，首先编写一个能够完成部分功能的小函数（完成圈地或者自己不会撞自己的 random 走位），在了解该任务的难度后再做打算。

III. 第二次组会 5.27 二教 521


本次大家带来了自己所编写的具有各种缺陷的函数，但仍能看到一些雏形：宣泽远、余江晖分别编写了一个能够独自完成圈地的函数；何鑫、许睿安、于之恒则尝试编写不会自己撞自己纸带的函数，在纸带构成的几何图形上有了一定的认识。

在宣泽远的主持下，大家依次发表了对于 play 函数逻辑的想法，最终一致同意建立 storage[‘order’] 指令序列以方便对于不同模块间的切换，并以此为基础建立最终的主函数运行逻辑。

在余江晖的提议下，大家统一了各个函数接口。

各函数接口的制定

方向的确定：坐标对确定（1，0）EAST；（0，1）SOUTH；（-1，0）WEST；（0，-1）

转向指令：'R'右转；'L'左转；'F'直行

宣泽远将模块分为进攻、圈地、返回三部分，并进行了人员分配，制定了一期分工，预期一周后提交成果。

一期分工

进攻（attackmode）：何鑫、于之恒

对于当前情况给出是否进攻的指令，主要考虑对方拖有纸带的情况；也可尝试考虑正面博弈时的打法，但不做要求

圈地（basemode）：宣泽远、余江晖

完善对起末点的选择；每次规划一个让价值最大化的路线

返回（gobackmode）：许睿安

（可能主要是在进攻和圈地函数没有达到既定效果时使用）进攻不成时在不碰己方纸带的条件下以最快速度返回己方领地；在圈地中遇到危险时以最快速度返回己方领地

公用基础函数建设：宣泽远、余江晖

goto（从一点到另一点）（完善使其能反向）；boundary（求己方领地边界）（优化）turn（转向）distance（距离）

IV．第三次组会 6.2 二教 523

首先由何鑫、许睿安、宣泽远分别汇报了进攻、返回、圈地模块的思路和已完成的功能，经过大家的讨论确定了每个模块现存的比较重大的问题以及进一步的改进方向。

随后宣泽远根据目前看到的可能会被对手全包围整体吃掉的情况提出了穿刺对手领地从而破坏其连通性的思路，并着手将其实现。

各模块报告与改进方向

进攻模块：

不调用额外参数

思路：判断距离都是先粗略后精细；比对我方进攻距离和敌人的返回距离以及地方进攻距离和我方返回距离；再对路线进行判断，确认敌人无法攻击我方后确实执行进攻指令

返回值：一个列表。为空则说明不执行进攻命令；有则执行该列表中的order

调取时的操作：每一步调取storage['attack']，True/False表示是否进攻，若True则执行order命令、False则运行进攻函数生成指令，若不能进攻则调用圈地

改进：模糊进攻部分

圈地模块：

思路：依照起末点的所围成的最大价值来选点，为处理特殊情况考虑了开闭区间的判断，对确定起末点后的基础路线进行了最大价值规划

改进："拓"部分——更贪心的圈地并辅以实时危险判断，规划整体战术，"补"模块的计划与编写，穿刺的应用

返回模块：

先把起点终点画框，从一个方向优先级规划一条路径；遍历边界，找到最短路径

改进：最短路线优先，而非远离地方优先；方向优先级？起点终点共线的情况？

V. 第四次组会 6.9 二教 523

开心的同志们

（左侧：宣泽远、于之恒、何鑫，右侧：余江晖、许睿安）

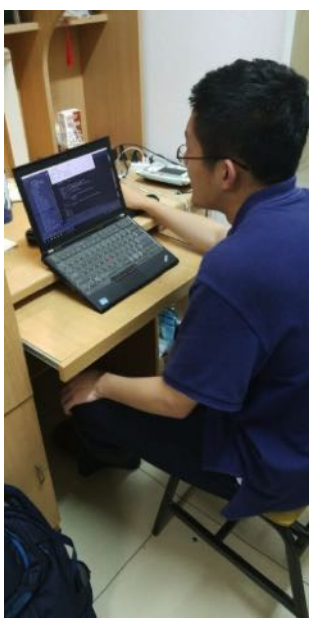
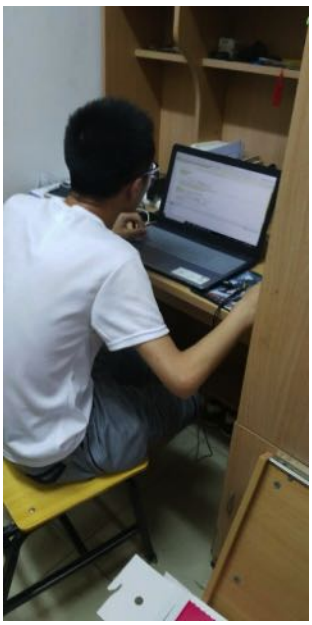
在宣泽远的主持下，大家经讨论确定了主函数 play 对各模块的调用逻辑，各个模块均据此调整各自的函数接口。

何鑫、于之恒完成了全部进攻模块；许睿安改良了返回模块，使其能够单次确实地给出到区域边界点的最短距离；宣泽远、余江晖分别完成了穿刺模块和圈地时危险判断部分。

小组基本进行至集中 debug 阶段，基本确立了由已经编写完进攻模块的于之恒、何鑫来主要测试代码，并将对应的问题分别交给该部分负责人宣泽远、余江晖、许睿安，且由这三人来修改自己所写的函数。

VI. 决战前夜 6.11 38 楼 235

Debug（左右图分别为余江晖、何鑫）

针对之前热身赛为排名靠前且与我们交手战绩出色的小组量身制定战术，通过 X-Ray 终稿的参数控制和攻守的平衡性来实现这一点。完成了 X-Ray 针对纯圈地模式（加强穿刺）和 X-Ray 特异化攻击模式（在对攻中不后退）。

V. 聚餐（手动斜眼笑）

4.2 经验与教训

这一次的大作业，毫无疑问是给予了我们很大的锻炼和能力的提升的。

我们小组的最大特点是没有“大腿”（手动滑稽），所以反而也促进了我们更加紧密地合作。最欣慰的一点是，由于一开始统一讨论和制定框架，每个组员都很快地找到了自己的状态和工作方法，并且在模块组装时几乎没有遇到交流和接口的障碍。当我们看到一个程序从最底层基建的’L’/’R’指令逐渐构建成可以轻易利用现有函数、参数接口组装战术的状态时，不得不说是很有成就感的。

唯一有一丝遗憾的是，在进入最终的阶段之后我们没有足够的时间将战术调整的功能通过程序自身实现，以至于最后在我们没有能够及时看到对手的复盘而调整自己的函数情况下，无法改变函数功能，由于模式的克制关系而惜败。

另外的一点是，对于这类排他性的比赛活动，做好 AI 智能化的搭建固然重要，但在时间较短、无法达到高程度的智能时，或许我们应该注意对具体规则的利用，也即同时进行针对“人为”战术的研究。

总的说来，这一段经历带给我们的不仅是成绩，更是一种历练与美好的回忆吧。

——宣泽远

很高兴能和 X-Ray 小组的大家一起完成这次大作业，能够真正的体会一次小组共同完成编程项目的经历对我来说是十分快乐而满足的。

在这次的算法设计中，我个人认为在最初设置的 order 指令体系与基础函数构建为后续的工作打下了坚实的基础，是我们本次作业中的一大亮点。这不仅仅是因为它们降低了功能实现上的难度，也因为这种平台搭建式的思路能够不断给予我们新的启发而指导下一步的行动。但比较可惜的是，我们在最后处理各个模块的联系、确定各模块应该实现的功能上似乎还欠考虑，导致最后几天有一点混乱，做出的成品也没有达到我们最好的预期。不过这也是模块分工体系会带来的问题，相信有了这次大作业的经历我们以后会做的更好。

——余江晖

我觉得这次的竞赛有别于平时上课很不一样的是：他是完全开放性的问题。之前课堂上学习到的经典问题都有漂亮的解法、使用特定的数据结构可以游刃有余地处理，但是 paper.io 并没有，甚至我们连要处理的“问题”是什么就思考、讨论了很久，虽然在玩网站版时操作极为简单，甚至也不需要制定策略，只要在危急关头按一下键盘即可；但是当我们要设计 AI 时，这套思路完全行不通，我们认为的一个个简单的指令，却要通过一个又一个判断条件、回圈循环、优先级判定，才能达成最基础的指令。在实践这些功能的时候，并没有明确的算法或是数据结构的指示，这些抽象的概念只能从我们学过的知识点中拼拼凑凑，不断尝试、测试可行性，每一次的构思都透过人脑进行一遍又一遍的沙盘推演，或许是我们还以人类的思维方式来控制 AI，但毫无疑问地，经过这次的机会，我们彻底了解到要实现一个 AI 绝不是件易事。当工作进展到中期时，我们体认到面对这种未知的开放性问题，想找到最优解近乎不可能，在任何时刻的必胜法并不存在，甚至每一个算法都有其着重的特色，在各种功能实践上只能寻求相对好的解法，并且努力优化、除错，这一点就

与课堂上那些漂亮的代码截然不同。当我们面临两个方案的选择时，一方面要考虑完成后的效益，一方面还要衡量代码本身的时间复杂度、函数调用的开销、使用的限制等等技术层面的问题，这些无不是在应用这学期所学到的知识，使之更臻完善。

此外，组内合作也是我们完成竞赛的一大考验，面对一个庞大的问题，如何分而治之彼此独立、却又彼此融合成了关键，明确接口、界定好变量、规范的返回，这些都是平常一个人完成编程作业时不需要多加留意的，但在这次的合作中，我们切身体会到协力合作的重要，唯有如此，才能充分发挥每个人的能力。这些训练及经验，可能不会在期末试卷上考察，也没办法完全展现在竞赛结果上，但是对于我们未来处理问题的助益却是无比深远且深刻的。很开心能够有这个机会与身边的同学一起完成这份大工程，虽然累，但却获益匪浅。

——许睿安

4.3 建议与设想

总的来说，这一次的实习大作业是十分吸引人的，并且各方面的基础设施（除了某次突然改接口导致全盘都得改 QAQ）也是非常完善的。如果能够在竞赛过程中加入更多的因素（比如多方面对战）以及在最后的比赛中能够用系统及时地向各队反映当前状况，或许就更好了。

另外，虽然这次作业的规则改编自一个成熟的游戏，但是由于只是双人的对战（而非原来的多人），使得在交锋过程中出现了“谁先退谁吃亏”的现象。个人觉得这在一定程度上造成了过多的死循环和较为单一的战术选择，或许在以后的规则制定中可以更多地考虑这方面的因素。

5 致谢

首先要感谢陈斌老师和助教大大们给我们创造了一个体验合力完成一个大项目的机会，并向我们介绍了“纸带圈地”这样一个有趣而又容易上手的游戏，指导我们进行以前想都不敢想的 AI 制作；也要感谢技术组的二位大佬，为我们搭建了一个对战平台，从而为算法竞赛创造了条件。

此外要感谢 X-Ray 小组每一位组员的付出，虽然大家都是第一次尝试编写这种“大工

程”，但每个人都努力的思考工作、积极的出谋划策。我们虽然可能没有“大腿”，但从零开始思考战术、编写代码、debug，每一个人都将它们做了个遍，最终才有了 X-Ray 小组的完成品。衷心的感谢大家对于这一次大作业的努力，相信大家都有着许许多多的收获。

第二十五章　F17_Yankee 报告

陈羲 杨芯 赵阳 吴天阔 唐镛 马浩男

摘要：通过 python 进行编程；记录和处理数据都使用线性结构，涉及元组、列表、字典、队列等；主要通过贪心算法进行找点，并通过队列进行行动指令记录；找点时保证朝一个方向走，防止撞带；在和老师提供的 ai 进行对战时战绩不错，但实战时出现卡死的情况，代码尚需进一步分析改进。

关键字：线性结构 队列 贪心算法 单向找点

1 算法思想

（1）总体思路：

我们的策略主要是圈地。在领地内部时寻找最佳的出去点，在边界时找最佳的目的点（目的点的优劣是以离领地的距离和安全性来衡量，粗略地认为离领地远并且绝对安全的点是好的点），如果到达后安全就继续找下一个点，遇到危险时找回来的点，在这里我们简单地把回来的点暂时定为出去的点，然后检查行动队列，一旦遇到边界，该点即为回来的点。同时在找延伸点时一直朝一个方向，返回时从另一侧走回来。

（2）采用的数据结构与算法：

我们采用线性数据结构，主要涉及元组、列表、字典、队列等，而算法方面主要是贪心算法，即在选取出去的点时，只保证当前的点最佳即可。

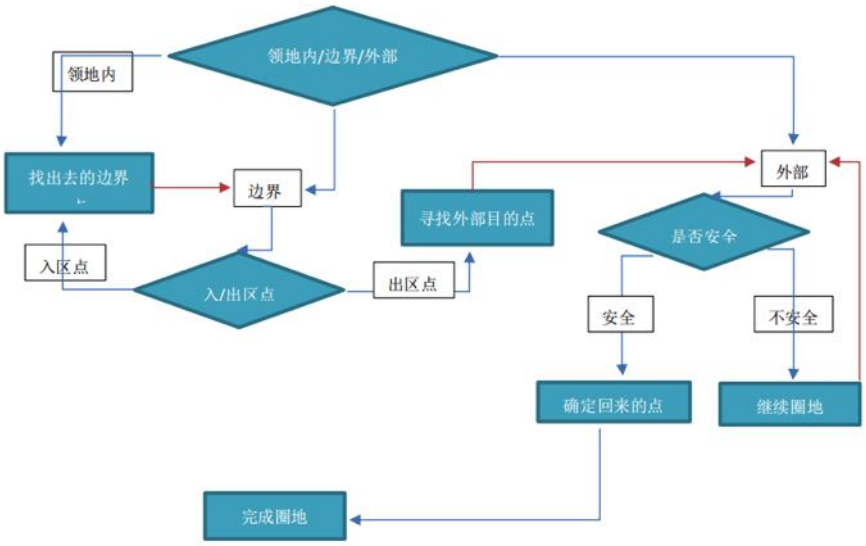
（3）算法策略及其完善过程：

刚开始我们的策略大致和总体思路里的差不多，唯一的区别在于怎么找回来的点。刚

开始我们找回来的点没有比较保险的策略，只是就近找点，这样就会很容易出现撞击自身纸带的情况，于是我们决定将回来的点暂时设为出去的点，然后检查行动队列，一旦遇到边界，该点即为回来的点。这样改动之后撞击自身的情况基本不存在了。

1.1 总体思路

1.2 算法流程图



1.3 1.3 算法运行时间复杂度分析

设棋盘上所有点的数量为 n，代码中的各种队列长度、目的点的个数均为 k，自己的领地上所有点为 m。

则 findpoint 函数、creatpoint 函数、disofhome 函数、firstqueue 函数、nextqueue 函数、backhome 函数时间复杂度为 O(k)，demaxlen 函数为 O(√n)，findout 函数为 O(k√n)，checkqueue 函数为 O(n+k)。（以上都是最坏情况，最好情况是 O(1)）

阅读代码发现我们的 AI 时间复杂度最高的地方在于判断安全与否并找回家点的代码，其平均时间复杂度为 O(m(k+n))。

```
if actionQueue.isEmpty() and findQueue.isEmpty():
    a = distance(direction, myx, myy, othx, othy)//5
    if fields[myx][myy] != myid:        #当前点不在自己领地内：

        # 后面是生成第二次的最大化的队列，初始时设置边长是3a，然后逐渐缩小至一个安全的路径
        while not findQueue.isEmpty():
            findQueue.dequeue()
        while not tempQueue.isEmpty():
            tempQueue.dequeue()

        g = 3 * a
        nextQueue(tempQueue, myx, myy, size, g, out_pointlist, statelist)              #生成队列，因为
        start_x, start_y = tempQueue.items[-1]                                  #记录出发点
        backHome(tempQueue, start_x, start_y, out_pointlist, statelist)            #生成回家队列
        dis_home = disOfHome(tempQueue, start_x, start_y, out_pointlist, statelist, fields, myid)
        safe = checkQueue(tempQueue, othx, othy, fields, myid, bands, dis_home)          #检查安全性
        while not safe:                     #不安全
            while not tempQueue.isEmpty():
                tempQueue.dequeue()             #清空生成的队列
            g = g - 1                           #缩小边长再试探
            nextQueue(tempQueue, myx, myy, size, g, out_pointlist, statelist)
            start_x, start_y = tempQueue.items[-1]
            backHome(tempQueue, start_x, start_y, out_pointlist, statelist)
            dis_home = disOfHome(tempQueue, start_x, start_y, out_pointlist, statelist, fields, myid)
            safe = checkQueue(tempQueue, othx, othy, fields, myid, bands, dis_home)
            if g == 0:
                break
```

综上可知代码的时间复杂度平均为 O(m(k+n))。

2 程序代码说明

2.1 数据结构说明

在代码中，我们主要运用了元组，列表，字典，队列等线性结构，这些数据类型的使用为我们的算法实现提供了很大的便利。

列表：作为一种相当基础的结构，我们在算法中大量的使用了列表。如边界点的记录，敌我纸带的记录等都用到了列表，并通过遍历列表的方式完成了寻找回家边界点，躲避撞击自己的纸带，计算敌人攻击自己与自己攻击敌人的距离等具体想法，特点是简单而且实用，至于具体的实现放在函数部分给予说明。

元组：元组在算法中被大量用于记录点的坐标的操作，部分的被用作记录方向的方式，如（x1,y1）表示坐标，（1，0）表示向东的方向等，具有清楚明了，操作简单的特点。

字典：字典在攻击函数模块被用到，由于每一个点都会产生很多的可用信息，所以想到用字典对该点的这些信息进行记录，使调用更加方便，结构更加清晰。

队列：队列是我们主要使用的数据结构类型，它被用来存放行进过程会经过的的点，即 actionqueue；或者被用来记录去边界点的队列，即 findqueue。由于队列具有有序性，和单一可操作性，每次处理只用处理队列队首的点，它给我们的操作带来很大的便利，具

体实现会在函数说明中具体说明。

2.2 函数说明

（1）基建函数：

由于这部分函数比较简单，我们集中起来简单说明：

a)Isback 函数：用来判断点是不是在前进方向的反方向上，返回布尔值

b)Isedge 函数：用来判断点是不是在墙上，返回布尔值

c)Attack 函数：用来判断是否能够进行攻击，有三方面考虑：自己攻击敌人的最短距离，敌人攻击自己的最短距离，敌人最快的回家距离，当自己的攻击距离比其他两者都小，攻击条件成立。

d)distance 函数：传入两点坐标及前进方向，若在正背后，则返回两点距离加 2，若否，返回曼哈顿距离

e)mht 函数：给两点，返回横纵坐标差的和

f)bandPoint 函数：传入场地相关参数，按横纵坐标遍历，把所有判定为自己的纸带的点加到纸带列表中去，返回纸带列表

g)edge 函数：传入场地相关参数，扫描全场，若左右，或上下归属不同，则判定为边界点，并把该点加到边界列表里去，返回边界列表

h)findpointx 函数，findpointy 函数：传参为两点坐标，按照先走 x 或先走 y 的原则走以这两个点为对顶点的正方形的两条邻边，把这两条边上的点都加到行动队列里面去，准备后续操作

i)creatpoint：传入行动队列，状态列表，两个坐标，根据状态列表暂存的状态来决定调用 findpointx 还是 findpointy 函数，保证前后的行动规则一致，来避免纸带自撞的情况。

j)disofhome 函数：调用 back home 函数，遍历该函数生成的行动队列，返回实际的回家距离。

（2）算法核心函数：

a)edmaxlen 函数：该函数功能是对某一边界点，延申的找四条对角线上在保证不回区，不被敌人撞击和不撞墙的情况下找到能够出界的最长距离，参数需要传入点坐标，场

地信息，敌方位置等参数，返回一个距离作为该点的优良程度评判标准。这里我们采用了贪心算法的思想，不管之后的情况如何，只要保证当前点较佳即可，另外我们粗略的将出区距离长作为圈地效率高的评判标准，本身存在一定的不合理性。

b)findout 函数：调用 edmaxlen 函数就近的搜索我方位置附近的最佳出界点，并返回该点；如果找的点恰好在墙上，则无法很好的做出判断，这里会有些小问题。

以上是在领地内找到最佳的出区点的函数部分，

c)gopoint 函数：算法的输出核心，在这里传入行动队列和我方位置等，根据相对关系，返回左转，右转，直走的信息，是函数整个的面向外围的接口。

d)expandfirst 函数：到达边界点，沿当前的方向出界，并根据传入的安全距离作为边长确定一个当前方向下的正方形的对顶点，把该顶点作为目标点返回。

e)expandnext 函数：作为函数第一次出界后的下一个找点的函数，与 expandfirst 函数类似，但需要注意的是我们把横纵坐标差加入考虑范围，因为第一的函数可能会遇到墙等情况而没有走严格的正方形，在之后这个函数可能不仅仅作为第二次扩张的选点函数等原因，我们保守的把坐标差为零的情况做单独考虑。另外，我们还通过传入一个出界点的方式判断了上一步的行进方式，，确保这一次的行进规则与上一步相同，以保证自杀的情况不会发生。

f)firstqueue 函数：用来生成第一次向外扩张的行动队列。参数有行动队列，我方坐标，方向，场地信息，安全距离，出界列表状态列表等，调用这个函数时，就把当前的我方坐标加入出界列表头，由于调用时间的关系，可以保证这个坐标就是出界点，以备后用；内部调用 expandfirst 函数，找到可行点，并按照既有方向决定先走 x 还是先走 y 并记录，这里用到了状态列表 statelist 做记录用。

g)nextqueue 函数：传参与 firstqueue 基本相同，在生成队列前需要根据状态列表的参数确定扩张规则，不再赘述，功能与 firstqueue 也基本一致，但这个函数最后调用了 backhome 函数，就是说我们的函数一般只会扩张两次，两次后就必须回家了，也是出于次数多之后会有自杀情况的考虑。

h)backhome 函数：传参最主要的是行动队列和出区点，由于出区点一定位于区域中所以把出区点作为回家的入区点一定能回家，同样要保证与前几步有相同的行进规则。功能与前两个函数基本一致。

i)disofhome 函数，判断自己回家需要长距离，返回该距离。做法是传入当前位置和出

区点位置坐标，根据 back home 函数确定的路径，将其遍历后可得回家长度。

以上是算法出区扩张选点及生成行动队列的部分

j)checkqueue 函数：用来检查队列安全性。传参有一个队列，敌人坐标，场地信息，回家距离等，虚拟一条还没有跑完到家的纸带，遍历该纸带，与敌人位置计算一个相对距离，判断与回家距离的大小关系，若有比回家距离大的，则返回 false，若无，则返回 true。

以上是用来保证我方行动不会被攻击的函数部分。

（3）主函数部分：

主函数部分没有解释参数的必要，我们主要对函数想法做一说明。我们的函数可以说时建立在两个队列的基础上的，actionqueue 和 findqueue。前者用来存储出区后的行为方式，后者用来存储入区后找出区点的行为方式，这两个队列的空与不空可以用来判断当前我们应该执行什么操作。以后以行动队列和找点队列代指这两个队列。

A. 行动队列和找点队列都空：

一般这种情况出现在纸带到达出区点，刚刚进入领地或者完成第一次扩张后，包括最开始的扩张。若此时不在区内，则为完成第一次扩张。此时暂时用第二次扩张函数确定一个 tempqueue，再用检测安全性的办法判断该距离下是否安全，若否则缩小距离，重复尝试，直到保证安全，缩小到零为止，之后调用回家函数，并把该队列加到行动队列中去执行；若此时在区内，判断上步状态，若在外，则是刚刚进入领地，调用找出区点的函数，找到出区点，加入找点队列，记录状态；若在内，则是从入区点走到了出区点，调用第一次扩张函数，检查安全性，记录状态。

B. 行动队列不为空：

要及时判断，若上步在家当前步显示进入己方领地，表明虽然和预期不符但是仍然由于某种情况使我方返回领地，假设我们圈好了地盘，那么我们直接置空行动队列并调用找出界点函数准备出界，并记录状态。

C. 找点队列不为空：

D. 要及时判断，若找点函数在某些情况下虽然没到达预想的最佳点但是遇到了边界，就置空当前的找点队列直接判定为出界，调用第一次扩张队列准备出界。

（3）其他函数：

我们也尝试了编写攻击函数部分，主要的思路是通过避开我方纸带找到一条可以攻击对方纸带的最佳路径，并通过相同算法找到敌人攻击我方的最佳路径，以及计算出敌人回家的最短路径，当我方攻击路径小于敌方攻击路径和回家路径时，我方发起进攻。但是就编写结果来看似乎这部分函数内部出现了一些问题，导致调用之后程序就会崩溃，所以最后我们没有把这部分函数加到我们参赛用的代码中去，比较遗憾。

2.3 程序限制

（1）实际限制：

我们的算法逻辑似乎是完整的，但是编写效果并不和我们预期的一致。我们在测试中主要遇到了如下的几种现象：

1. 纸带会在领地内绕圈，并不像我们预期的那样只要一进领地就立刻找到一个出区点立刻出区，这里我们会浪费大量的回合数，推测可能是某些情况有疏漏造成的。

2. 纸带到达墙后不会判断是不是墙直接撞出去而被判负，似乎是因为到达边界后我们的函数不会给出转向指令使纸带保持原来的方向导致的。

3. 纸带有时会画出完全不合理的安全距离而无法自保使得被对手撞死

（2）逻辑限制：

1．我们的逻辑比较简单，就是对行为进行拆分再组装起来，在我们的行为还没有结束之前的一些情况的变化可能会对我们造成不利，如我们选定的回家点可能在我们回家之前就被划走导致我们没有回家但是没有后续的行为操作等。

2．还有就是方案本身的问题，如选出界点操作本身不能保证我们在自己的领地内游走而是出界一段距离，若就近找不到点则可能下一次找到的点会浪费相当的回合数；清空队列的操作也很值得商榷因为我们无法判定我们是不是圈好了一块地就开始了下一次的扩张，导致图形不封闭等等

3．由于是团队合作在某些函数的接口上出现了小的矛盾，如 exmaxlen 函数找到的最佳点的最佳扩张方式本身并没有传递出来，后面的函数找到那个点但是选择的扩张方式并没有用到函数本身的期望等等。

4．再有就是我们攻击函数功能的不足会使我们的进攻优势很小，加上扩张效率不高，只采用了多个同方向正方形的扩张方式，使我们在两种方式的赢面都有不少提升空间。

3 实验结果

3.1 测试数据

实验环境说明：在 solo 平台上基于以下配置环境

硬件配置：（CPU/内存）Intel(R)Core(TM) i7-7700HQ CPU@2.80GHz / 8GB

操作系统：（名称/版本）WIN10

Python 版本：（版本号）3.6.5

我们在 solo 平台上用我们的 AI 和技术组提供的三个 AI 分别进行了三十次对战实验，没有出现异常情况，实验结果如下表

对阵 AI	对战总次数	获胜次数	获胜情况	失败情况	耗时情况
AI_simple_wanderer	30	22	15 次以面积取胜，7 次在领地内被对手撞击	3 次撞到边界，3 次撞到自身，2 次在对方领地撞到对方	千回合耗时约 5 秒，最大回合数 2000 时平均耗时 8.36 秒
AI_simple_wanderer	30	28	16 次以面积取胜，12 次在领地内被对手撞击	2 次撞到边界	千回合耗时约 5 秒，最大回合数 2000 时平均耗时 7.91 秒
AI_dumb_random	30	30	14 次以面积取胜，16 次在领地内被对手撞击	无	千回合耗时约 5 秒，最大回合数 2000 时平均耗时 6.32 秒

3.2 结果分析

生效策略：从表中数据可以看出，我们的 AI 在圈地方面表现不错，没有一次被对方以面积取胜，所以我们的圈地策略是有效的，并且在圈地的同时较好地保证了自己的安全，

没有被对方撞击到。

时间开销：时间开销比较稳定，千回合耗时大概 5 秒，虽然我们算法的数量级是 O(m(k+n))，但是在实际开销中用时并不高，这是因为我们采用的主要算法是贪心算法，AI 分析的数据量并不大。

4 实习过程总结

4.1 分工与合作

小组介绍与分工：

一起来自地球与空间科学学院的六个男生组成了我们这支队伍。由于每个团队成员都有各自的长处和特点，实在是难以割舍任何一个人，于是我们在经过一番思考和讨论后决定冲破一个队伍最多 5 个人的限制，成为数据结构与算法课堂上为数不多的 6 人小队。

我们的组长陈曦同学是经过民主评议选出的，主要负责团队的组织，重要大方向的把握，距离 ddl 所剩时间的掌控，团队例会的召开，以及与老师和助教的沟通。

技术组由杨芯赵阳马浩男三位同学以及组长陈曦组成，主要负责主代码的编写，代码中的 debug，对于所有代码中出现的各种问题的解决。

思维组由全体成员构成，所有人都提供了大作业中的点子和最基础的构想。

报告组由吴天阔唐镛马浩男三位同学组成，主要负责例会安排与记录（包括会议文字和照片的记录），所有资料的汇编，以及最后实习作业报告的编写。

我们合作与交流的方式由线上微信群的交流和线下在组长宿舍每周 3 次的例会组成。

4.2 经验与教训

得意之处：六个人的队伍，在分工方面，可以照顾到每一个队员的长处和短处，可以很好的发挥每个人的优势。

可以改进的方面：对于时间的把握有些不准确，导致最后因为缺少几天时间的原因没能够把代码做的更好些。最终在比赛时出现莫名原因而卡死。

4.3 建议与设想

对本次实习作业的建议：对于完成作业时间的消耗过于巨大，导致其他课程学习时间极其紧张。

对学弟学妹的寄语：一定要把握好距离任务结束还剩下多少时间，把握好自己小组的效率，这样才不会出现像我们组一样的遗憾。

大胆设想：能否不限于二者对战，而是三者？或者更多？

5 致谢

感谢陈斌老师发起这次活动，并在活动中提供了一些战略性建议；

感谢助教提供的建议以及在组长记错提交组员时间的情况下帮忙补救；

感谢技术组提供了对战的平台和若干 ai 进行对战；

6 参考文献

http://gis4g.pku.edu.cn/course/pythonds

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

https://blog.csdn.net/caoshulin1989/article/details/53068674

第二十六章　F17_Zulu 报告

夏一飞，胡禛海，彭镜宇，葛孟帅，胡伟一

摘要： 本文介绍了 Zulu 小组算法的基本思路和各个函数的代码和功能，以及对于程序的分析。该算法属于保守型，以圈地为主，但仍有攻击功能，判断出一定击杀对手时会激发，遗憾的是该功能未在比赛前实现。算法复杂度为 $O(1) \sim O(n)$，全局总耗时不超过 1 s。算法的结构很简单，位于领地内就调用领地内走法函数 generate_goto()，位于领地外就调用圈地函数 route()，两个函数会自己判断场面局势，并返回走法。文中对这些函数做了详细介绍。算法的优势就在于圈地效率较高，但由于此算法保守性的本质，存在很多限制，比如近战会怂，打也不敢，围也不敢。小组内每个人都有相应的分工，大家协力合作完成了算法的实现。小组没有出线，组员对于比赛结果并不满意，并与成绩优秀的小组进行了沟通，在本文的最后介绍了大家的一些反思和收获。

关键词： 贪心策略;　动态调节;　专家系统

1 算法思想

1.1 总体思路

本次实习作业的场景是完全信息决策，我们需要在确保安全与尽可能高效地圈地之间找到平衡点，策略不外乎是保守型、进攻型与二者兼顾型。经讨论，我们决定采用保守策略，即在确保自身绝对安全的情况下，尽可能多地圈地，并兼顾特殊情况进行防御性进攻。

本组的思路大致包括 3 个原则：

1) 绝对不能死！对于 AI 而言，在能绝杀的情况下，是能够确保击杀的，所以当我们的 AI 探测到如果走出下一步就可能让自己陷入必死的境地（在最快的回领地路线上仍会被对方截断），就会折回；

2) 尽可能方形地扩大领地。在绝对不死的前提下，（在会被绝杀的边缘疯狂试探），只要不被剪断纸带就尽量向外规划领地；

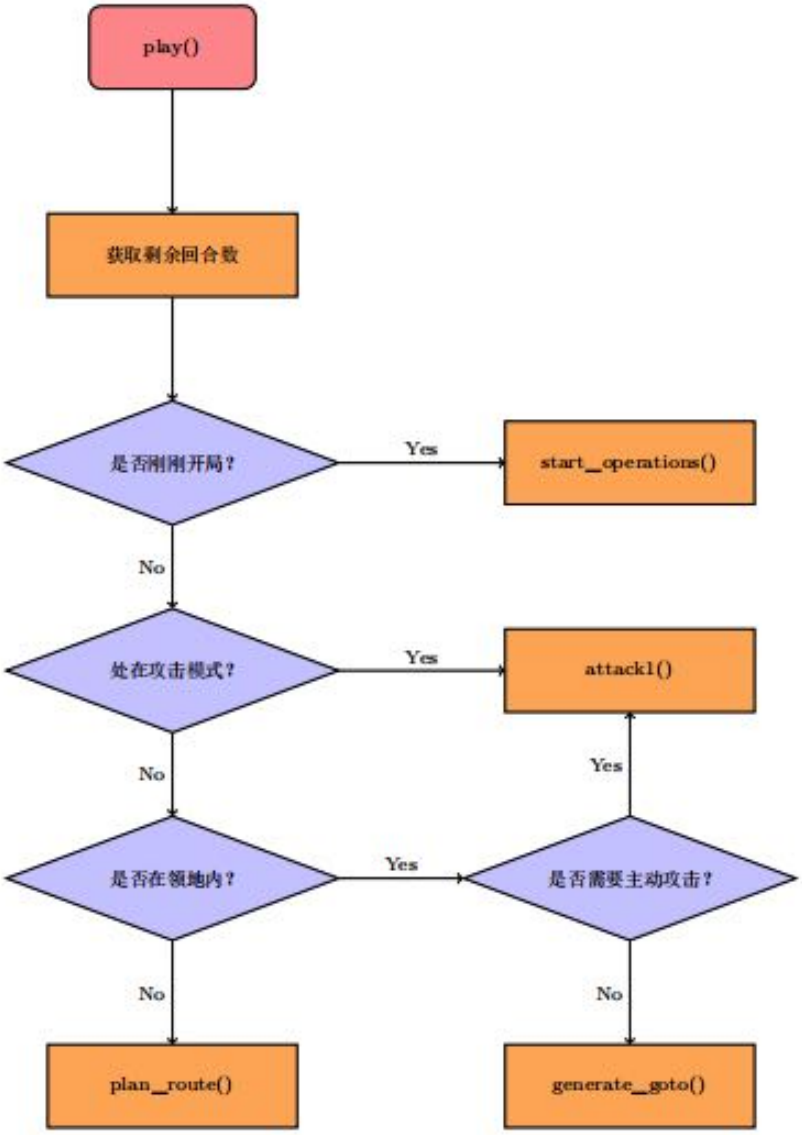
3) 在能够确保击杀对方的前提下立刻出击，将对手扼杀在领地之外。

而此三原则确立后，其实 AI 每一步的前进都已经确立了：防御、扩展和攻击。而再深入到实际操作层级时，又需要一些可复用的具体功能。

首先是扩张。在游戏的开局就需要扩张，而扩张又不可避免遭遇防御的问题：如何评估自己所规划的扩张纸带的安全性问题，这个函数完全集成在 route 函数里：在前一步规划的路径中进行适当的扩充，计划好新的路径直到返回领地，并评估此路径对方是否能从他所在的位置必然打到：如果会被绝杀，则退缩，按原路径继续走；如果安全，则按新规划的路径前进。

然后就是攻击。本功能在比赛时尚未完全成型。攻击分为两个部分，一个是在对方将要将自己“团团围住”时触发，粉碎地方的邪恶阴谋；另一个是在对方暴露出破绽，已经可以绝杀对方时（即满足能找到安全的路径切断敌方的纸带，并且对方无法在这之前回到领地）主动出击，一举将对方击杀，而此功能在比赛之前并未得到太多重视，因为以为所有组都会有安全性评估与相应的防御机制。直到比赛开始时才意识到也会有很多进攻组抓住防御为主的 AI 缺乏进攻机制，而进行一些实际十分“危险”的操作，而挤占对方生存空间；临时编好了代码，但最后并未能得以施行。

1.2 算法流程图



1.3 算法复杂度

我们的算法主要时间耗费在领地内走法判断函数和领地外围地函数上，时间复杂度为 $O(1) \sim O(n)$. 对于领地内走法判断函数，只有回到领地后第一次调用时需要遍历整个领地边界，生成一个列表记录之后的走法，设边界的尺度为 n，时间复杂度就为 $O(n)$，之后在边界内只需按照之前生成的走法列表走，时间复杂度为 $O(1)$；对于领地外围地函数，函数

每走一步都会判断规划的路径是否安全，需要每次检测自己领地在某一行或列的边界点，此操作时间复杂度为 $O(n)$，还需计算敌人攻击的最短距离，扫描规划路径上每一个点，若有 n 点，时间复杂度为 $O(n)$，所以此函数时间复杂度为 $O(n)$.

以上分析了最为耗时的两个函数，由于一开始就很小心，我们的函数时间复杂度是很小的，最大只有 $O(n)$，发现每一局总时间不超过 1 s，看来当时担心超时实在是多余了。

2 程序代码说明

2.1 数据结构说明

除了列表、字典等 Python 基本类型外，我们还定义了如下的自定义类，将大作业平台提供的各类接口统一进行了封装：

(1) Block 类：

主要用于存储自己或对方领地的按照 y 坐标不同分类的点的集合，附带功能有：

a) 得到区域精确边界点集；

b) 得到涵盖区域最小矩形外框边界；

c) 得到某一边的边界点；

d) Block 的取并及取差运算（用于更新场地）。

(2) Trail 类：

以列表的形式记录某一方已走过的 band 或计划要走的路径。支持通过路径列表生成操作列表，及操作列表生成路径列表。

2.2 函数说明

主逻辑

```
def normal():
    if stat['now']['fields'][stat['now']['me']['x']]\
       [stat['now']['me']['y']] == storage['my_id']:
        storage['route'] = {'OriPoint': [], 'temp': [1, 0, 0],\
                            'opt': [], 'flag': 0}
        if storage['edge_going']:
            return attack1()
        else:
```

圈地模式

```
def normal():
    if stat['now']['fields'][stat['now']['me']['x']]\
       [stat['now']['me']['y']] == storage['my_id']:
        storage['route'] = {'OriPoint': [], 'temp': [1, 0, 0],\
                            'opt': [], 'flag': 0}
        if storage['edge_going']:
            return attack1()
        else:

            attack1_finding()
        return generate_goto()
    else:
        if storage['edge_going']:
            return attack1()
        return plan_route(storage['turn_to_direction'])
```

(1) Plan_route 函数：

描述：该函数以假定纸带移动出领地后按照直走、转向、直走、转向（与前一转向方向相同）、直走的路径回到领地为基础，每走一步判断安全性并调整预设路径，且针对特殊情况有相应的微小的调整。

调用时机：当我方纸带处于非我方领地时

输入：事先预置的方向

判断与返回：判断分为三个分支，分别是纸带位于预设路径的第一、第二、第三条边上时。

1) 位于第一条边时，我们假定的路径的第二条边长与第一条边长度一致，因此确定了两条边的长度，而第三条边的长度可根据第二个转向点与我方领地边界来计算得出。如果预设的三条边长之和小于安全距离（敌我距离），且直走一格后三条新的预设边长之和大于安全距离则转向。

2) 位于第二条边时，先走到边长与第一条边一致时（由于领地可能会变化，每走一步仍做判断，不安全则转向）。之后仍然与位于第一条边相同进行判断，如果直走一格后两条新的预设边长之和大于安全距离则转向。

3) 位于第三条边时，一直前进回到领地。

Plan_route 函数中调用了几个比较重要的函数：

1) EnemyToUs 函数：该函数输入为四个 tuple 类型的点，即预设路径三条边上的四个点。由四个点的 x, y 值可以将三条边上的所有点存入一个 List 里，然后计算对方纸带到每个点距离的最小值，即是安全距离。返回该最小。

2) EdgePoint 函数：该函数输入为一个 tuple 类型的点，一个字符

串（'top','bottom','left' 或者 'right'）。如果字符串是 'top' 或者 'bottom'，那么就根据输入点的纵坐标找到我方领地该纵坐标下顶端或者底端的点。如果字符串是 'left' 或者 'right'，那么就根据输入点的横坐标找到我方领地该横坐标下的最左侧或者最右侧的点。输出为 tuple 类型。

3) Slice_dge 函数：在 Block 类里，根据输入的横坐标（纵坐标）找到在该坐标左边/右边（上面/下面）领地的 top 或者 bottom 值（left 或者 right 值）。输入是一个点（tuple 类型），两个字符串。返回 int 类型数值。

(2) Go_out 函数：

本函数目的是寻找一个最优的出发点。

具体算法：对边界点进行遍历，计算每一个点的收益值，选取收益最大的点，同时对一些特殊情况进行单独处理。

收益值计算公式如下：

$$f_{\mathrm{p}} = \alpha \times \min\{d_2, d_3, d_4\} - d_1$$

其中，

- α：权重因子，具体值待探索与调整
- d_1：出发点到当前点的距离
- d_2：己方点与敌方点当前距离的 $\frac{1}{3}$
- d_3：出发点沿出发方向到边界的距离
- d_4：从出发点出发并转弯后到边界的距离

部分代码分析如下：

函数无需参数，直接从全局的 stat 以及 storage 中调用。my_id 获取了己方的属性，my_point 和 enemy_point 分别为己方点和地方点的当前位置坐标，my_field 为己方当前领地的集合（类型为 Block），BorderLine 则为己方领地的外边界点集合。

```
def Go_out():
    my_id = stat['now']['me']['id']
    my_point = (stat['now']['me']['x'], stat['now']['me']['y'])
```

```
    enemy_point = (stat['now']['enemy']['x'], stat['now']['enemy']['y'])
    my_field = storage['my_field']
    BorderLine = my_field.get_borderline()
```

对于领地边界上的点进行遍历，从中选取最合适的 StartPoint，但己方点不参与评估，同时将离场地边界过近的点也排除在外。当 StartPoint 仍为 None，即不存在满足上一个循环进入条件的点的时候，放宽限制条件随机选择一个点；否则返回 StartPoint.

```
for point in BorderLine:
    if (point[0] != my_point[0] and point[1] != my_point[1])\
        and 17 < point[0] < storage['x_len'] - 18
        and 17 < point[1] < storage['y_len'] - 18:
        # Some operations to get StartPoint.
if StartPoint != None:
    return StartPoint
else:
    for point in BorderLine:
        if point != None and 1< point[0] < storage['x_len'] - 2\
            and 1 < point[1]< storage['y_len'] - 2:
            outpoint = point
            break
    return {'x':outpoint[0], 'y':outpoint[1], 'out':1, 'turnto':'L'}
```

(3) generate_goto 函数：

只要现在在领地内，调用本函数，就能得到下一步应该走的方向。

当被调用时，本函数会判断是否刚刚进入领地：

1) 如果是刚刚进入领地，就会在调用 Go_out 函数获取下次圈地的出发点，然后在 storage['trace_goto'] 里存放一条轨迹 (如果原本有数据就被覆盖)。接着，在 storage['trace_goto'] 里读取第一个数据，然后把它删掉。

2) 如果不是刚进入领地，就总是在 storage['trace_goto'] 里读取第一个数据，然后把它删掉。

返回值：下一步的方向 ('L' 或 'R' 或 'None').

进攻模式

attack1 函数

attack1 函数用于发现敌人想要通过游戏场地边界围所有地的阴谋并将其领地切断。首先 attack1_finding 函数检测场地各边上有无敌军，为了降低时间复杂度，我们选取了一些样本点经行扫描，如果敌军已经分布在三条边上，就马上出动，调用 attack1 进行破坏。

场地维护

(1) get_points_of函数：

以某点为中心，一定宽度搜索周围领地，找出指定类型的点，并存储在字典 points 中，存储方式与 Block 中点的方式相同。

```
def get_points_of(flag, centre_x, centre_y, radius=5):
    points = {}
    if flag[0] is 'f':
        status = 'fields'
    elif flag[0] is 'b':
        status = 'bands'
    else:
        return False
    for y in range(max(centre_y - radius, 0),\
                   min(centre_y + radius, storage['y_len'])):
        t = False
        for x in range(max(centre_x - radius, 0),\
                       min(centre_x + radius, storage['x_len'])):
            if str(stat['now']['fields'][x][y]) == flag[-1]:
                if t is not True:
                    t = True
                    points[y] = set()
                this_point = (x, y)
                points[y].add(this_point)
    return points
```

(2) update_field函数：

更新的逻辑，某方从外部回到领地时触发，搜索并更新。

```
def update_field(I_arrive, she_arrives):
    if I_arrive:
        new_me = Block(storage['get_points_of'](\
                       storage['flag_my_field'],\
                       stat['now']['me']['x'],\
                       stat['now']['me']['y'],\
                       radius=storage['my_out_ticks']),\
                       storage['flag_my_field'])
        storage['my_field'].add(new_me)
        storage['her_field'].sub(new_me)
        storage['my_out_ticks'] = 0
    if she_arrives:
        new_her = Block(storage['get_points_of'](
                        storage['flag_her_field'],\
                        stat['now']['enemy']['x'],\
                        stat['now']['enemy']['y'],\
                        radius=storage['her_out_ticks']),\
                        storage['flag_her_field'])
        storage['her_field'].add(new_her)
        storage['my_field'].sub(new_her)
        storage['her_out_ticks'] = 0
    if I_arrive or she_arrives:
        storage['my_field'].border_update()
        storage['her_field'].border_update()
```

场地维护的相关逻辑如下：

a) 初始化双方领地，并储存在 storage 中；

b) 每回合记录上回合本回合的双方状态，如果某方上回合在场地外下回合在场地内，则以到达点为中心，该方该次外出所行进的半径，并通过 Block 类的并和差更新双方 Block 领地（A 回到领地，则搜索该点附近 A 领地的点，A 的领地并上新搜到的点集，B 的领地

减去新搜到的点集）；

c) 如果上回合在场地内，本回合在场地外，则初始化一个集合，并将该点的元组加入到集合里；如果本回合仍在场地外，则添加该点到集合；如果回到场地则清空此集合。

2.3 程序限制

1) 圈地函数的路线是理想的三边路线。有时会出现因为己方领地形状特殊走出两边路径就回到领地的情况，这时因为不存在的第三边在 route 中设计的路线比理想路径要差。

2) 设计的理想路线只考虑敌我距离和己方回到领地距离。如果在圈地过程中，己方领地被对方圈走了一块，改变了 route 中各参数值，此时就可能出现报错或纸带乱走的情况。

3) 在近距对战中，我们的算法为了确保绝对安全，并不能显示出良好的对战能力。

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 – 7200U @ 2.50GHz; RAM : 8.00 GB
- 操作系统：Windows 10 版本 1803
- Python 版本：3.6.3

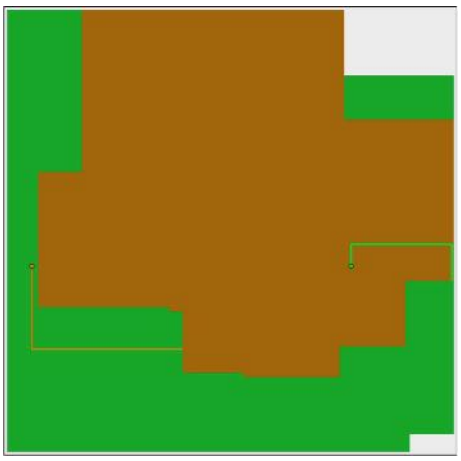
我们使用大部分都用人机对打的方式测试算法，并与隔壁小组的算法进行友谊赛。

3.2 结果分析

如果纯圈地，不进行战斗，算法的圈地效率是很高的，不输于人脑，但人类主动攻击个别情况会被杀，敌人靠近，受到干扰后这个很怂的 AI 圈地效率也会很小，经人机测试，防止敌人从场地边缘围地的函数也是有效的。

与技术组给的几个算法比赛我们的算法都是全胜，我们的算法与其他组提供的陪练 AI 的比赛，胜率达到了 12.

3.3 经典战局



tick 数耗尽，以圈地效率取胜

4 实习过程总结

4.1 分工与合作

夏一飞（队长）：实现了 generate_goto() 函数，并负责各模块之间的统筹工作。完成了实习报告的排版工作。

胡禛海：实现了算法会用到的基本的类，为各模块提供接口。

胡伟一：实现了场外圈地函数 route().

葛孟帅：实现了场内走法函数 generate_goto().

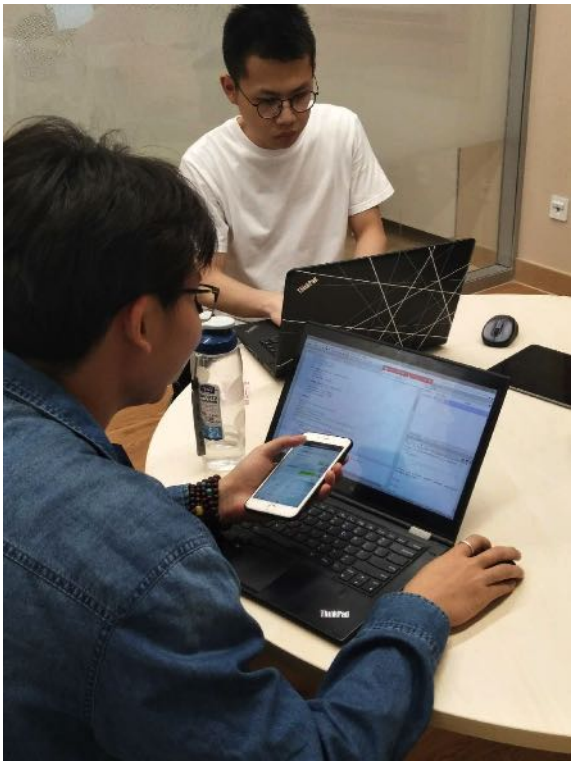
彭镜宇：实现了防止对手在边界围地函数 attack1().

该分工只是大致的，实际上我们也常会帮助队友实现部分功能，每个人的贡献都不限于自己的分工。

下面是我们组会上拍摄的一些照片：

仔细论证“三步走”圈地战略

讨论算法实现细节



组会上汇报各自工作

4.2 经验与教训

我们最初的想法是采用极大极小值剪枝算法：每一步有直走、左转、右转三个选择，通过特定的函数给某时刻棋盘状态下的一个格点打分，然后将分数储存在一个三叉树中，用极大极小值算法由叶向上传递分数，最终返回最优的分数及其选择的移动方向。

然而，由于棋盘很大，对局情况多样，状态函数比较难以确定。在尝试过程中，首先区分了己方领地、纸带，未被圈地部分，对方领地、纸带几个区域，以某一个格点到双方距离为最基础的变量，总体思想是离我方越远、离敌方越近数值越高。同时用安全评估函数对格点的打分进行调整，最后判断其走向。但是实际操作过程中发现纸带的走势非常杂乱，而且简单的打分很难让它回到领地进行有效圈地。想要纸带更为智能地移动必须要有很好的打分函数作为支撑，而在短时间针对万格的棋盘写出这样的函数的确非常困难，在艰难地写了一段时间以后，我们最终放弃了该种算法。

4.3 建议与设想

我们的最终算法实际上只有围地功能，而不能主动攻击，一直希望我们的算法可以增加攻击函数，但是由于开始较晚时间不够，非常遗憾，这个功能比赛前一直没有实现。另外，与比赛成绩优秀的小组比较，我们得到一点教训，认为不应该花费太多时间在讨论如何围地的具体实现上，因为我们自己也不精通此游戏，自己讨论不可能得出最佳方案，相比之下，其他小组尝试不同的策略，对算法进行大量的的测试，不断得到新的灵感，不断对代码进行改进，获得了很好的效果。希望以后类似的活动中可以学会如何在短时间内得到一个好的算法。

5 致谢

感谢陈斌老师给我们带来这门精彩的课程。感谢技术组大佬们提供的接口和思路。

6 参考文献

大作业项目网站https://github.com/chbpku/paper.io.sessdsa