数据结构与算法课程实习作业报告

张广欣* 胡皓然 李英楠 王雪峰

摘要：（简要介绍算法原理，涉及的数据结构与算法，实验数据结果概述）

关键字：启发式算法，广度优先搜索，无向图

1 算法思想

1.1 总体思路

（介绍算法的总体思路，采用的主要数据结构与算法，采用的算法策略，算法从最初设想到实现确定的过程）

我们起初希望实现的是尽量把兵力铺开，即尽可能多的占领无人节点，因此最先想到了 BFS 算法，对每次遍历到的新节点进行派兵操作，并将其标记为 visited 状态。然而我们发现使用这种算法会导致某些节点的兵力大量囤积，只进不出（我们称之为貔貅节点）。为了解决出现貔貅节点的问题，我们考虑改进算法。

于是我们在第一回合以敌方大本营为起点进行 BFS 搜索，对整个地图的结构进行了分析。此后，将兵力从离敌方大本营较远的位置迁移至相距其更近的位置，从而避免了貔貅节点的出现。

在作战策略上，我们先考虑了使用 MCTS 算法，然而发现此问题中的估值函数很难设计，且程序的时间限制(100 ms)不太能够允许此算法所要求的大量的搜索。故我们放弃了这一思路，决定改用启发式算法，通过观察并学习天梯排名靠前的算法的可视化战况中体现的作战策略来试图达成“师夷长技以制夷”的目标。

小组成员通过对战局仔细的研判，发现我方的节点大致可以分成三类：扩张地盘、对前线进行补给、以及与敌方节点遭遇——需要对战的节点。根据从观察高手代码的战况表现中所习得的派兵习惯，我们依据回合数来决定后方向前线派兵以后所剩余的兵力应为多少，从而保证前线能持续得到较多的兵力供应，并且后方能够可持续发展。

1.2 算法流程图

算法的总流程图示意如图 1，外部 player_func 函数的流程图示意如图 2。

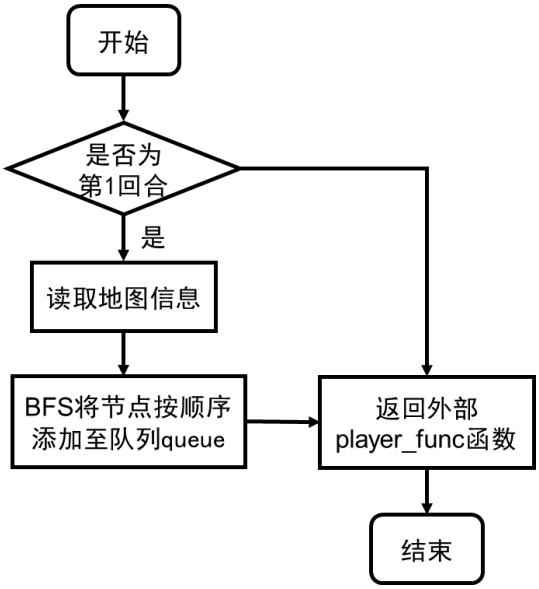

图 1 算法的总流程图

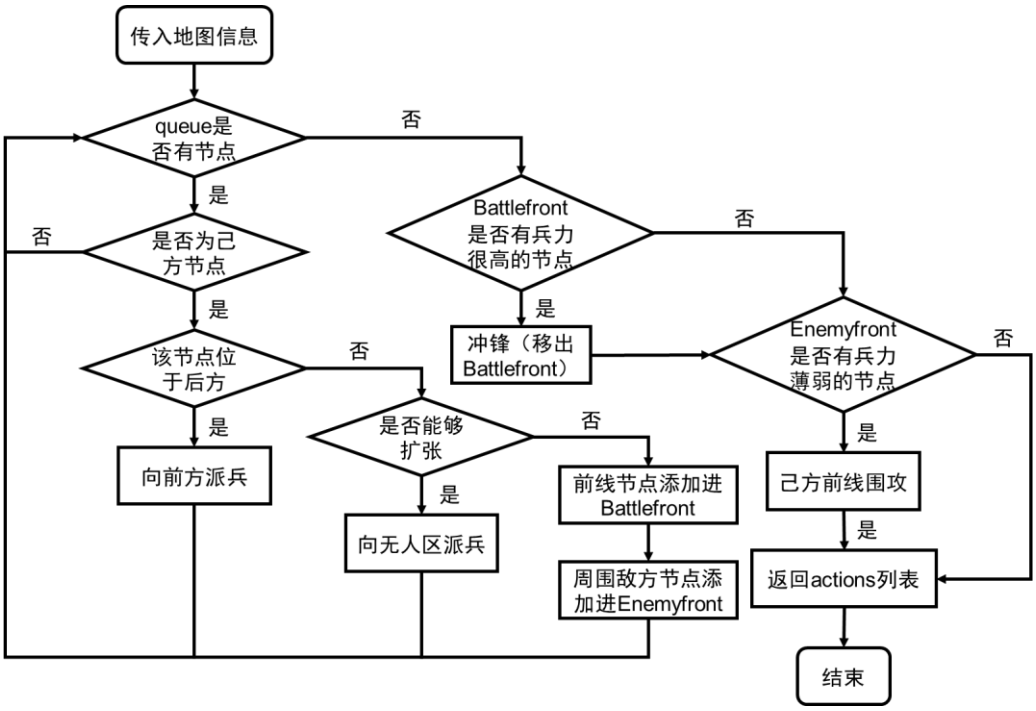

图 2 外部 player_func 函数的流程图

1.3 算法运行时间复杂度分析

（估算算法运行时间的主要构成，估计复杂度的大 O 数量级，在实际运行中的时间开销如何）

本代码的结构可以分为两大部分。第一部分是对于地图的 BFS，时间复杂度为 O(V+E)，即节点的数量加边的数量。第二部分则是对每个节点的派兵操作，时间复杂度为 O(V*E)。具体原因是，要对每个节点都操作一遍，即遍历一遍节点，复杂度为 O(V)；另外，针对于每一个节点，都对其进行了状态的判断——即识别其相邻的所有的节点，然后根据周围的节点情况进行相应的派兵操作，这与边数有关，复杂度 O(E)，叠加起来复杂度就是 O(V+E)。通过 for 循环的叠加数量也能大致判断出来数量级。

总结一下，就是算法时间复杂度为 O(V*E)，既与节点数量成正比，也和边的个数成正比。

2 程序代码说明

2.1 数据结构说明

（说明算法中采用的主要数据结构、自定义类等，对课程上介绍的线性、树、图等基本数据结构有什么扩展和改进）

算法主要采用的结构就是列表。

在对地图（无向图）进行 BFS 搜索的过程中，将节点按照顺序排成了列表（队列）self.mapqueue，然后按顺序遍历里面的节点，进行派兵操作。

对兵力部署进行维护时也使用了列表（这里其实使用字典能优化一些，但是对时间复杂度没

什么影响就没优化），存储了每个节点进行已有的派兵操作后的兵力。
另外，针对技术组一开始提供的 random 代码，对 player_class 类进行了优化，新添了 self.mapdic、self.mapqueue、self.count 三个属性，分别记录该地图的节点遍历顺序、节点的“地理位置”、已进行的回合数。前两者能够保证不会每个回合都对地图 BFS 搜索浪费时间，第三者能够保证实现一些启发式规则，比如前期保留兵力少，后期保留兵力多等等。

2.2 函数说明

```
def player_func(map_info,player_id,dicnode,queue2,count):
    actions = []
    power1 = [0]  #对应每个节点派完兵之后的兵力
    for i in range(len(map_info.nodes)-1): #没派兵时，就是当前节点兵力
        power1.append(map_info.nodes[i+1].power[player_id])

    def judge(node):  #用于判断节点的当前情形,-1 是攻击，0 是扩张，1 是补给
        adjuncts = map_info.nodes[node].get_next().copy()  #把相邻节点存储在列表里
        spread = False  #看节点旁边有没有空的地方
        for i in adjuncts:
            if map_info.nodes[i].belong == -1:
                spread = True
            elif map_info.nodes[i].belong == 1 - player_id:
                return -1  #面临攻击

        if spread == True:
            return 0  #扩张
        else:
            return 1   #进行补给

    if count <= 6: #根据回合数选择后方补给线的保留兵力
        great = 9
    elif count <= 9:
        great = 16
    elif count<= 10:
        great = 29
    else:
        great = 48

    def chooseAction(node,power1,great):  #对 node 的派兵进行操作
        situation = judge(node)
        adjuncts = map_info.nodes[node].get_next().copy()  #把相邻节点存储在列表里
        adjunct = []  #adjunct 是需要派兵的节点的 list
        troop = []  #troop 与 adjunct 一一对应，是对应的节点的派兵数量
        nodepower = int(map_info.nodes[node].power[player_id])
```

```
        if situation == 1:    #补给
            for i in adjuncts: #把兵力往前方输运
                if dicnode[i] < dicnode[node]:
                    adjunct.append(i)
            #调整合适的输运数量，确保不越界，且能保留 great 的兵力
            totalnumber = max(power1[node]-great, 0)
            totalnumber = min(totalnumber, nodepower)
            if totalnumber>0:
                troop = [totalnumber/len(adjunct)]*len(adjunct)

        elif situation == 0:  #扩张
            for i in adjuncts: #把需要扩张的点记录下来
                if map_info.nodes[i].belong == -1:
                    adjunct.append(i)

            expand_number = nodepower // 9  #扩张的节点数
            #如果节点本身内部空虚，就少扩张一个，防止发出的兵损耗完了，不值得
            if power1[node] < (expand_number + 1) * 9:
                expand_number -= 1

            if expand_number < len(adjunct):
                adjunct = adjunct[0:expand_number]

            totalnumber = max(power1[node] - 9, 0)  #派兵的总兵力，保留 9 个守家
            totalnumber = min(totalnumber,nodepower)
            if len(adjunct) != 0:
                troop = [totalnumber/len(adjunct)] * len(adjunct)

        for i in range(len(troop)):       #对 adjunct 里的节点派兵，添加进 actions
            if troop[i] != 0:             #对每个节点派兵数量从 troop 中读取
                actions.append((node,adjunct[i],troop[i]))
                power1[adjunct[i]] += troop[i]-troop[i]**0.5
                power1[node] -= troop[i]

        return power1  #维护兵力部署

    battlefront = []
    enemyfront = []
    for i in queue2:  #遍历节点进行操作
        if map_info.nodes[i].belong == player_id:
            #如果节点在前线，添加入列表 battlefront 中
            if judge(i) == -1:
                battlefront.append(i)  #节点添加到己方前线
                #判断哪些是对方前线的节点，添加到 enemyfront
```

```
                for j in map_info.nodes[i].get_next():
                    if map_info.nodes[j].belong != player_id:
                        enemyfront.append(j)

            #如果不是前线，调用 chooseAction 函数，同时维护了兵力部署
            else:
                power1 = chooseAction(i,power1,great)

    for node in battlefront:  #判断己方前线的某个节点是否有大量兵力
        nodepower = int(map_info.nodes[node].power[player_id])
        #兵力很足，否则跳过冲锋环节
        if nodepower >350:
            #移除 battlefront，直接冲锋，不考虑后续的两面夹击
            battlefront.remove(node)
            adjuncts = map_info.nodes[node].get_next().copy()
            attacknode = adjuncts[0]
            #挑选要攻击的节点，离敌方大本营越近越好
            for i in adjuncts:
                if dicnode[i]< dicnode[attacknode]:
                    attacknode = i
            #添加攻击指令，维护兵力部署
            actions.append((node,attacknode,nodepower))
            power1[node] -= nodepower

    for node in enemyfront:  #判断敌方是否有薄弱节点，如果是，则两面包夹
        #敌方节点的相邻节点
        adjuncts = map_info.nodes[node].get_next().copy()
        #敌方节点兵力
        nodepower = map_info.nodes[node].power[1-player_id]
        #我方节点总兵力
        opponentpower = 0
        ourlist = []
        for i in adjuncts:
            #将 adjuncts 中的我方节点找出来
            if i in battlefront:
                opponentpower += map_info.nodes[i].power[player_id]
                ourlist.append(i)
        #如果我方合力夹击能打得过就打
        if opponentpower > nodepower * 1.2:
            for i in ourlist:
                battlefront.remove(i)
                #选择出兵数量，添加进 actions
                movepower = max(0,power1[i]-28)
                movepower = min(movepower,int(map_info.nodes[i].power[player_id]))
```

```
                if movepower > 0:
                    actions.append((i,node,movepower))
                    #维护兵力部署列表
                    power1[i] -= movepower

    return actions

class player_class:
    def __init__(self, player_id:int):
        self.player_id = player_id
        self.mapqueue = []  #用于派兵时对节点进行遍历
        self.mapdic = {}  #存储每个节点离敌方大本营的位置
        self.count = 0

    def player_func(self, map_info):
        self.count += 1
        if self.count == 1:
            queue = []   #用于扫描节
            if self.player_id == 0:
                queue.append(len(map_info.nodes)-1)  #将对方老巢加入队列，设定为 0
                self.mapdic[len(map_info.nodes)-1] = 0
            else:
                queue.append(1)
                self.mapdic[1] = 0

            while queue:    #每次 pop 队列第一个出来，将它的相邻节点加入队列（如果没搜索过）
                node1 = queue.pop(0)
                self.mapqueue.append(node1)   #记录节点的搜索顺序，它的逆序就是派兵顺序
                node1_next = map_info.nodes[node1].get_next()
                for i in node1_next:
                    if i not in self.mapdic:
                        queue.append(i)
                        self.mapdic[i] = int(self.mapdic[node1] + 1)
            self.mapqueue.reverse()   #queue2 的顺序就是等会遍历节点的顺序

        return player_func(map_info,self.player_id,self.mapdic,self.mapqueue,self.count)
```

主程序 player_func 函数：

输入地图信息 map_info，玩家信息 player_id，节点 BFS 后的层次信息 dicnode，节点的遍历顺序 queue2，回合数 count。返回玩家的派兵操作 actions。

采用 BFS 算法判断节点与敌方大本营的距离，用贪心算法依次遍历每个节点，来保证兵力除了扩充地盘外，都往前线调动。采用列表 power1 记录当前我方的兵力分布。采用启发式规则，根据当前回合数 count，判断后方保留兵力。调用 judge 函数和 chooseAction 函数，判断单一节点的状态并行动。如果节点在前线对战，采用“一往无

前”和“两面包夹”两种策略。

如果前线单一兵力超过 350，则全员出动，攻击离敌方大本营最近的薄弱位置。

如果对方某一前线节点兵力较少，其相邻的我方几个节点总兵力大于它的 1.2 倍，则我方这些节点都向其出兵，保证最后我方节点剩余 28 个兵（可以是后方补给上来的兵力）。

judge 函数：

输入节点信息 node，返回节点的当前状态 (0 代表有空地盘可以扩张，1 代表需要往前线补给兵力，-1 代表在前线，面临攻击)。

功能：判断节点的当前情形。

采用遍历相邻节点的策略。如果相邻节点有敌方节点，返回-1；如果没有敌方节点，且周围有空节点，返回 0；其余返回 1.

chooseAction 函数：

输入节点信息 node，兵力分布 power1，后方补给线保留兵力数量 great。将派兵操作记录到列表 actions 中，并返回新的兵力分布 power1。

采用启发式规则，如果该节点要往前线补给，则保留 great 数量的兵。向前方节点派兵总数量是“当前节点兵力减 great”和“当前节点可支配兵力”中的最小值。如果要扩张，需要保证扩张后新节点的兵力大于等于 6，否则不利于可持续发展，要减少往外扩张的节点数。如果要战斗，交给 player_func 函数处理。

要更新 power1，即：当前节点兵力要减去派出的兵力，对象节点的兵力要增加收到的兵力。

player_class 类中 player_func 函数：

2.3 程序限制

程序在运行时未发现存在 bug。可惜更新的一代版本有一点小 bug，没有及时改好，只能用旧版本去比赛）

3 实验结果

3.1 测试数据

实验环境说明：

- 硬件配置：CPU：Intel(R) Core(TM) i7-8750H CPU @ 2.20GHz 2.21 GHz
  内存：8.00 GB
- 操作系统：Windows 10 家庭中文版
- Python 版本：Python 3.8.3

（说明采用了何种测试方法来对算法进行测试，测试结果如何，如果有编写测试程序请列出，以表格或者图表形式列出参加热身赛的结果）

本代码主要采用以下三种方式进行测试：

1.利用开发文档中自带的 debugger 生成 Json 文件在可视化界面进行逐帧分析（如兵力运输、战斗策略等是否达成目的），该测试方法主要运用于编写代码前期 debug 环节，主要目的是代码实现；

2.在自由对战平台上将不同版本（优化前后）的代码相互对战实现自我测评，以判断是否达到优化效果；

3.在天梯平台与大佬们的高版本代码切磋，分析对方的更优策略以及我方代码的短板，并进行相应的针对性修改。

基于以上测试方法实现的代码更迭情况见下表：

代码版本	修改操作	取得的进展
Idiot_1	初代代码，基于给定的随机 AI 算法进行一定程度上的修改，使之不会后退。	对任务的实现有了初步了解，作为初代代码起码能跑，完胜原来的 AI 代码。
Idiot_2	对铺场操作添加 BFS 算法。	更好的实现铺场任务，将铺场效率最大化。
Idiot_3	记录每一步操作的节点数变化，将后方兵力保持在一定水平。	改善了后方兵力增长率，但单纯使用 BFS 算法造成节点兵力堆积的问题无法解决。
Idiot_4	抛弃原有的 BFS 架构，添加蒙特卡洛算法与估值函数。	理论上可以解决兵力堆积问题，但严重超时。
Idiot_5	削减了蒙特卡洛算法搜索的数量，将节点分为后方补给与前线战斗，只在前线进行搜索操作。	部分缓解了超时情况，但在节点数量达到一定值时仍会超时，且蒙特卡洛并没有达到预期的效果。
Idiot_6	引入启发式规则，将节点分为补给、运输以及战斗节点，分别写函数指挥节点行为。	解决了铺场的节点堆积问题，但前线过于疲软，缺乏进攻性和防守能力。
Idiot_7	修改了战斗节点相关的函数，对双方节点兵力进行判断并作出相应的行动。	增加了前线节点的防守能力，但相对来说过于保守。
Idiot_8	在前线兵力达到一定数量的情况下主动出击，攻击对方的薄弱节点。	部分场景下侵略性得到加强，但有时会导致主力节点向反方向“逃窜”。
Idiot_9	对 Idiot_8 的参数进行优化	前线的攻击与防守行为变得更加合理。
Idiot_10*（未实现）	添加了前线支援与撤退函数，在兵力处于劣势时更加理性。	Bug 过多，debug 时间不够充分导致该想法未实现。

最终提交代码版本为 Idiot_9。热身赛结果如下：

代码名称	attack-N
等级分	1543.02
参与对战数	25130

获胜局数	12096
胜率	48.1%
排名	12
是否出线	否

3.2 结果分析

（说明在测试、热身赛和实战对弈过程中，本算法的哪些策略起了何种作用？效果如何？算法在运行时间上是否在预期之中？主要的运行时间开销发生在什么环节？）

总体来说，代码在排位赛中的表现整体符合我们的预期。本算法的铺场策略与前八名选手的代码效率基本上不相上下，问题主要出在算法的进攻效率略显单调，无法实现前线的进退调兵与支援。

从积极的一面来考虑，本代码不管是从运行时间还是函数实现上来看均较为稳定，从来没有出现过“按兵不动”的情况。在第一回合中，该算法实现了对地图的整体遍历并根据节点的相对位置进行赋值，以此防止了位于死路的节点不断堆积兵力，实现了兵力的合理运输；位于前线的战斗节点，在能力不足以歼灭对方时按兵不动，实现了对兵力的保存与防御；对于大兵力的前线节点，实现了在特定情况下的“冲家”战略。总而言之已经达到了令人比较满意的效果。

本算法的运行时间开销在于对每个节点进行遍历并进行派兵操作，但其实该操作所占用的时间也并不长，因为算法的时间复杂度本身就较低。

3.3 经典战局（可选）

（如果在实习过程中发现本算法有令人惊奇的表现（包括与其他小组算法对战过程），可在本节中加以描述）

4 实习过程总结

4.1 分工与合作

（说明小组分工，合作与交流的方式，历次组会记录（照片！））

张广欣 胡皓然 李英楠 王雪峰

4.2 经验与教训

（总结实习过程中的工作经验，有哪些方面是得意之处，哪些方面可以改进）

4.3 建议与设想

（请对本次实习作业提出建议，在组队、基础设施代码、竞赛等哪些方面可以改进？对选修明年课程学弟学妹的寄语；实习作业后续工作的大胆设想）

5 致谢

（请感谢下对本小组实习过程有贡献的人和事）

6 参考文献

（列出实习过程中用到的参考资料、网站链接等）