City_526 代码说明文档

潘俊生 姚力源 黄贤

一、对任务、系统的理解

本次比赛的道路数据取自北京三环以内的道路，路网结构为由车道 lane 组成道路 road 和路口 junction，道路通过路口连接形成路网，作为车辆起点终点的兴趣点 poi 分布于道路上，车辆通过某一条车道进出各 poi。我们认为，在所给情境下，堵车的主要原因是汽车进出兴趣点以及路口处的车辆交汇。

二、算法思路

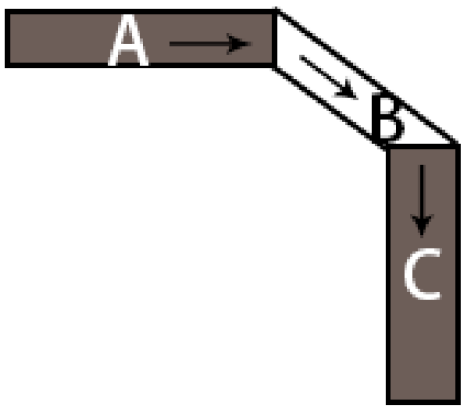
我们的思路是对于每一条拥堵的车道，向前回溯其之前的车道，分析其车流的来源，统计其之前的各车道对该车道车流的贡献量，并将车流贡献量最大的车道列入限行规则库中，在对所有拥堵车道分析完毕，得到完整的限行规则库后，再通过枚举抽样的方式从中对限行车道进行离散化得到限行规则，防止连续拥堵路段被全部限行，反而降低通行效率。多次循环取优化结果最优的方案，对其进行仿真、获得新的交通状况后，重复上述步骤进行迭代，以获取进一步优化。

我们的算法主要分为以下四个部分：

第一部分是对当前交通状况的统计分析，获取拥堵车道的 id。在当前限行规则下（初始为无限制）运行一次仿真，得到当前限行规则下的交通情况"traj.txt"。对每一辆汽车在主要考察的半小时内的行驶情况进行跟踪，当车辆在某时刻的行驶速度(准确说是在上一个 10 秒内行驶的距离)低于设定的阈值时，认为车辆在当前路段堵车，于是将此车当前所在车道计入堵车路段目录。

第二部分是寻找已知车道的前驱车道。具体功能是对于一条已知的车道，遍历所有车道信息寻找出连接该车道的前驱车道与前驱车道的前驱车道。

第三部分是计算已知车道 A 对其接下来延伸的车道 B 和 C 的车流的贡献量，其中车道 A 和 C 是位于两条相邻道路（road）的车道，车道 B 是连接 AC 的路口车道。具体指若一辆车上一时刻在车道 A，下一时刻就行驶到接下来的车道 B 或 C 上，则记录车道 A 对 C 的车流贡献量加一。在观测的 30 分钟内对目标车道进行观察，记录这 30 分钟内车道 A 对 C 贡献了多少车流量。



第四部分的功能是将以上三部分的功能整合，得到最后的优化方案。具体指

通过第一部分得到拥堵车道 id 列表，调用第二部分代码获取拥堵车道列表中的每个车道的前驱车道，即拥堵车道的车流来源(可能有很多条)，再调用第三部分代码计算拥堵车道的各车流来源对拥堵车道的车流贡献量，将每一条拥堵车道的车流来源中车流贡献量最大的车道列入限行规则库，通过枚举抽样的方式从中选出部分车道后得到优化方案。此时即完成了一代优化。

迭代的具体方式如下：

首先，在不设置道路限行规则时直接进行车流仿真，得到"traj.txt"，将"traj.txt"文件拷贝至代码运行文件夹下，执行"deal.py"代码，即可得到初代限行规则库"access.txt"。之后执行"randomizefun.py"代码从限行规则库中进行枚举抽样，枚举该次抽样的抽样间隔以及抽样位置，抽取其中的部分限行道路编号，每次枚举抽样得到一个"access1.txt"文件，即为本代所添加的限行规则，将其内容续至整体限行规则"access0.txt"文件中，即完成首次迭代。将每次枚举抽样得到的新"access0.txt"在平台中进行仿真，获取本限行规则下的平均车辆行驶时间，其中结果最优的，作为本代的最优抽样结果。

将上一代最优解仿真得到的"traj.txt"再次拷贝至代码运行文件夹下，执行 deal 获取本代的限行规则库，再使用 randomizefun 进行枚举抽样，并将枚举抽样得到的新"access1.txt"续在上一代最优解的整体限行规则"access1.txt"最后，直至找到本代的最优解，即完成了下一次迭代。

三、具体代码、注释

Deal_Comma.py 文件：

```
import json


def throughput(rl, bl, ol, tj):  # throughput(拥堵车道,路口车道,拥堵车道的前驱车道,交通数据)
    dic1 = dict() # 生成空字典
    thruput = 0 # 表示"拥堵车道的前驱车道"对"拥堵车道"的车流贡献量
    for i in range(90, 270): # 在观测的 30 分钟内
        data = list(tj[i].split()) # 读取某一时刻的车辆信息
        cnt = len(data)
        for j in range(1, cnt, 3): # 对每一辆车进行观察
            if data[j + 1] == ol: # 如果该车在"拥堵车道的前驱车道"上
                dic1[data[j]] = data[j + 1] # 则将该车计入字典
        data = list(tj[i + 1].split()) # 读取下一时刻的车辆信息
        cnt = len(data)
        for j in range(1, cnt, 3): # 再观察每一辆车
            if data[j] in dic1: # 如果这辆车在上一时刻在"拥堵车道的前驱车道"上
                if data[j + 1] == rl or data[j + 1] == bl: # 且这一时刻在"拥堵车道"或"路口车道"上
                    thruput = thruput + 1 # 则该车属于"拥堵车道的前驱车道"对"拥堵车道"贡献的车流
        dic1.clear() # 清空字典供下次循环使用
```

```
    return thruput # 输出贡献量
```

上面的 throughput 函数用于计算"拥堵车道的前驱车道"对"拥堵车道"的车流贡献量，即前文所提的第三部分。核心思路为寻找上一时刻在"拥堵车道的前驱车道"、下一时刻在"拥堵车道"或"路口车道"上的车辆并计数

```
def getLaneLink(lane_id, Lane2predec):
# 输入参数为目标车道以及前驱车道列表
    Linklist = []
    for info in Lane2predec[lane_id]:
        if Lane2predec[info] == []:  # 如果前驱的前驱为空则跳过
            continue
        templist = []
        templist.append(lane_id)
        templist.append(info)  # 前驱车道 id
        templist.append(Lane2predec[info][0])  # 前驱的前驱车道 id
        Linklist.append(templist)
    return [Linklist]
```

getLaneLink 函数用于寻找目标车道之前的车道，即前文中的第二部分。核心思路就是遍历所有车道，查找其前驱车道及前驱车道的前驱车道，值得注意的是一条车道可能有多组前驱车道及前驱车道的前驱车道

```
with open('map.json', 'r', encoding='utf8') as fp: # 读取道路信息文件
    json_data = json.load(fp)
header = json_data[0]
laneList = [] # 车道列表
roadList = [] # 道路列表
junctionList = [] # 路口列表
poiList = [] # 兴趣点列表
for info in json_data:
    if info.get('class') == 'lane': # 如果类型是"车道"
        laneList.append(info.get('data')) # 则将该车道信息计入车道列表，下面三个同理
    elif info.get('class') == 'road':
        roadList.append(info.get('data'))
    elif info.get('class') == 'junction':
        junctionList.append(info.get('data'))
    elif info.get('class') == 'poi':
        poiList.append(info.get('data'))
```

以上部分代码将车道、道路、路口和兴趣点的信息整理成列表，以便于后续的查阅

```
lane2parent = {}  # 标记所属的路口 id 或路 id
```

```
lane2predec = {}  # predecessorid  前驱车道字典
lane2succes = {}  # successorid    后继车道字典
lane2junc = {}  # 获得某车道 相连的路口
 # Lane2parent、lane2predec、lane2succes 通过线性遍历 laneList 列表进行赋值。
for info in laneList:
    lane2parent[info.get('id')] = info.get('parent_id')
    lane2predec[info.get('id')] = info.get('predecessor_ids')
    lane2succes[info.get('id')] = info.get('successor_ids')
for info in laneList:
    # ane2juc(value 为 key 车道 id 相连的路口，如果属于路口，则为单值，即路口 id；如果属于路，则元素为列表，列表第一个元素为前驱的路口 id，第二个元素为后继的路口 id)
    if info.get('parent_id') > 300000000:  # 区分车道属于路口还是属于路在于 parent_id 首位数字，首位为 2 代表属于路，首位为 3 代表属于路口
        lane2junc[info.get('id')] = info.get('parent_id')
    else:  # 如果车道不属于路口，则找到该车道的上一个路口和下一个路口 实现过程中发现，其前驱后继均属于路口。即一条车道若属于路，其前驱后继(若存在)均属于路口。
        b = lane2predec[info.get('id')]  # 获得前继列表
        prejunc = []
        for a in b:
            if lane2parent[a] > 300000000: #若属于路口
                prejunc.append(lane2parent[a])
            else:
                prejunc.append('0') # 用来观察是否存在特殊情况 实现中发现并不存在
        b = lane2succes[info.get('id')]  # 获得后继列表
        sucjunc = []
        for a in b:
            if lane2parent[a] > 300000000: #若属于路口
                sucjunc.append(lane2parent[a])
            else:
                sucjunc.append('0')
        lane2junc[info.get('id')] = {'pre': prejunc, 'suc': sucjunc}
print("finish reading road info")
```

以上部分将车道信息进一步整理，得到关于车道前后连接、与路口连接等信息的字典，便于后面的查阅

```
with open('traj.txt', 'r') as f: # 读取车辆行驶数据
    tj = f.readlines()
dic_poi = dict() # 以有兴趣点的车道 id 为索引的字典，内容为兴趣点在该车道上的位置。用于查阅哪些车道上有兴趣点
for i in poiList:
    i['driving_position']['lane_id']
    dic_poi[str(i['driving_position']['lane_id'])] = tuple(
```

```
        dic_poi.get(str(i['driving_position']['lane_id']), "n")) + tupl
e(
            [i['driving_position']['s']]) # 遍历所有兴趣点，以所在车道 id 为
索引、所在位置为内容记录
```

以上部分是整理了兴趣点信息，获得车道 ID 对应兴趣点信息的字典

```
# d = float(data1[0])
dic1 = dict() # 两个用于记录相邻两时刻车辆信息的字典
dic2 = dict()
dic_lane = dict() # 记录拥堵车道的字典：索引为拥堵车道的 id，内容为汽车拥堵的
位置
threshold = 1 # 判断车辆是否拥堵的速度阈值
distance = 0 # 用于记录车辆移动距离
for i in range(90, 270): # 在观测的 30 分钟内
    data = list(tj[i].split()) # 读取某一时刻车辆行驶信息
    cnt = len(data)
    for j in range(1, cnt, 3): # 遍历该时刻的每一辆车
        dic1[data[j]] = (data[j + 1], float(data[j + 2])) # 以车辆 id 为索
引，记录所在车道及位置
    data = list(tj[i + 1].split()) # 读取下一时刻车辆行驶信息
    cnt = len(data)
    for j in range(1, cnt, 3): # 遍历每一辆车
        dic2[data[j]] = (data[j + 1], float(data[j + 2])) # 记录车辆 id、
位置
        if data[j] in dic1: # 如果该车在上一时刻仍在路网中
            if dic1[data[j]][0] == dic2[data[j]][0]: # 且上一时刻与当前时
刻在同一车道上
                distance = dic2[data[j]][1] - dic1[data[j]][1] # 则计算
这段时间内的行驶距离
                if distance < threshold: # 如果行驶距离小于阈值
                    dic_lane[data[j + 1]] = tuple(
                        dic_lane.get(data[j + 1], "n")) + tuple(
                            [float(data[j + 2])]) # 则判定该车已被堵住，将
其所在车道及位置计入拥堵车道字典
    dic1.clear()
    dic2.clear() # 清空字典供下次循环使用
```

以上部分是分析了当前的交通状况，提取出拥堵车道的 id，即前文中的第二部分。核心思路为同时读取相邻两时刻的车辆行驶信息，寻找两个时刻都在同一车道的车，计算其移动距离，与阈值比较、记录。

```
print('finish analyzing traj')
Acc = open("access.txt", "w+") # 打开记录文件
for jamlane in dic_lane.keys(): # 遍历每一条拥堵车道
```

```
    prelaneList = getLaneLink(int(jamlane), lane2predec) # 用
getLaneLink 函数得出拥堵车道的前驱车道及前驱车道的前驱车道
    maxIndex = 0 # 记录对拥堵车道车流量贡献最大的前驱车道的 id
    maxThroughput = 0 # 单条前驱车道对拥堵车道车流贡献量的最大值
    for prelanei in prelaneList[0]: # 遍历每一条前驱车道及前驱车道的前驱车
道
        if str(prelanei) in dic_poi: # 如果前驱车道上有兴趣点
            continue # 则忽视后面的判断，因为有兴趣点的车道不能删
        currentThroughput = throughput(str(prelanei[0]), str(prelanei[1
]), str(prelanei[2]), tj) # 用 throughput 函数计算各前驱车道的前驱车道对拥堵
车道的车流贡献量
        if currentThroughput > maxThroughput:
            maxIndex = prelanei[2]
            maxThroughput = currentThroughput # 记录车流贡献量最大的车道
id 及贡献量
    Acc.write(str(maxIndex)) # 将车流贡献量最大的车道 id 计入文件
    Acc.write('\n') # 转行
Acc.close() # 关闭记录文件
```

以上部分整合了之前各部分的功能，得到限行规则库，即第四部分的主要功能。思路是寻找对每一条拥堵车道车流量贡献最大的车道，将该车道 id 计入待删目录

randomizefun.py 文件：

```
import os


def randomizefun(prag, mod):  # 从样本列表中进行随机抽样
    f = open('access.txt', 'r')  # 打开限行规则库
    ac = f.readlines()  # 将文件按行读取成 list
    Acc1 = open("access1.txt", "w+")  # 打开写入目标文件
    for i in range(1, len(ac)):  # 逐行进行判断
        for j in prag:  # 枚举每一个取模余数
            if i % mod == j:  # 同余则抽样
                Acc1.write(ac[i])  # 将抽样结果写入输出
                break  # 找到一个同余项则退出
    Acc1.close()  # 关闭写入目标文件
```

以上部分用于对现行规则库进行抽样，并写入新的“access1.txt”文件

在寻找最优结果时，执行以下代码，通过枚举抽样的方式，寻找最优的抽样参数，以获得更低的平均车辆运行时间。

```
# for mod0 in range(20, 30):  # 枚举基本抽样间隔
#     for i in range(1, 5):  # 枚举抽样间隔得倍数
#         mod = mod0 * i  # 获取本次得实际抽样间隔
#         for a1 in range(0, mod):  # 枚举第一个抽样位置
#             for a2 in range(a1, mod):  # 枚举第二个抽样位置
#                 for a3 in range(a2, mod):  # 枚举第三个抽样位置
```

```
#                     for a4 in range(a3, mod):  # 枚举第四个抽样位置
#                         for a5 in range(a4, mod):  # 枚举第五个抽样位置
#                             for a6 in range(a5, mod):  # 枚举第六个抽样
位置
#                                 randomizefun([a1, a2, a3, a4, a5, a6]
, mod)  # 执行抽样
#                                 os.system('pause')  # 停止程序，运行仿真
评估限行结果
```

根据我们的运行所得到的抽样参数，复现提交结果时，仅需按照如下代码逐行取消注释并执行即可。四次迭代依次对应如下语句中的一条，每次解除一行的注释并执行即可。

```
randomizefun([1, 5, 10, 17, 22], 24)  # 第一次迭代抽样
# randomizefun([2, 3, 11, 20, 26, 27], 48)  # 第二次迭代抽样
# randomizefun([2, 3, 12, 20, 38, 39], 48)  # 第三次迭代抽样
# randomizefun([2, 7, 13, 19, 25, 37], 48)  # 第四次迭代抽样
```

四、结果与复现方法

由于时间及电脑性能有限，我们仅完成了部分枚举抽样，得到的结果并不一定为最优解。如果能够完成全部枚举，则可能能够得到更优的限行规则。

具体复现方法：

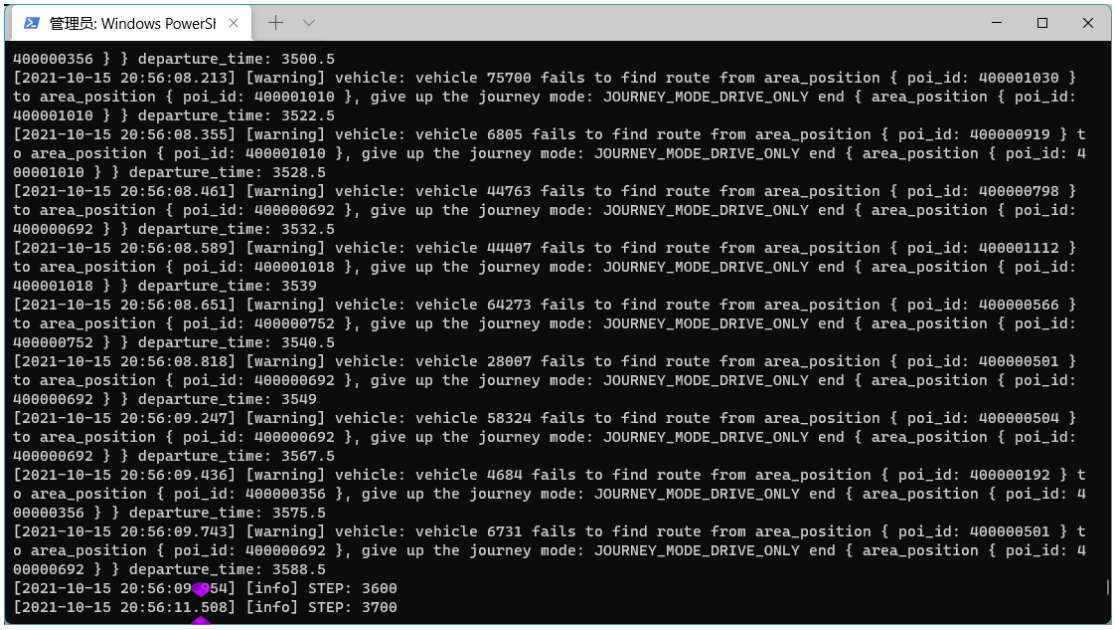
在无限制通行时，得到第 0 代"traj.txt"，执行"deal.py"后，得到第 1 个限行道路库"access.txt"。取基本抽样间隔为 24，间隔倍数为 1，抽样位置为 1，5，10，17，22 时，即执行 **randomizefun([1, 5, 10, 17, 22], 24) # 第一次迭代抽样**得到的结果"access1.txt"为较优解，平均车辆行驶时间为 748.26s。

在将"access1.txt"作为用于提交的"access0.txt"，并完成仿真后，得到第 1 代的"traj.txt"。执行"deal.py"后，得到第 2 个限行道路库"access.txt"。取基本抽样间隔为 24，间隔倍数为 2，抽样位置为 2，3，11，20，26，27 时，即执行 **randomizefun([2, 3, 11, 20, 26, 27], 48) # 第二次迭代抽样**，得到结果"access1.txt"，将其内容续在"access0.txt"之后，平均车辆行驶时间为 708.01s。

在使用限行规则"access0.txt"完成仿真后，得到第 2 代的"traj.txt"，执行"deal.py"后，得到第 3 个限行道路库"access.txt"。取基本抽样间隔为 24，间隔倍数为 2，抽样位置为 2，3，12，20，38，39 时，即执行 **randomizefun([2, 3, 12, 20, 38, 39], 48) # 第三次迭代抽样**，得到的结果"access1.txt"，将其内容续在"access0.txt"之后，平均车辆行驶时间为 694.19s。

在使用限行规则"access0.txt"完成仿真后，得到第 3 代的"traj.txt"，执行"deal.py"后，得到第 4 个限行道路库"access.txt"。取基本抽样间隔为 24，间隔倍数为 2，抽样位置为 2，7，13，19，25，37 时，即执行 **randomizefun([2, 7, 13, 19, 25, 37], 48) # 第四次迭代抽样**，得到的结果"access1.txt"，将其内容续在"access0.txt"之后，得到的结果较优，平均车辆运行时间为 685.67s，即为提交得版本。

在执行仿真时，我们注意到提示出了如下警告：



报错的车辆似乎并未完成出发，且没有出现在"time.txt"及"traj.txt"中。

为了解决这一问题，我们尝试在获取限行道路库时检测待添加的道路上是否有兴趣点，甚至只限行个位数条相距甚远的道路等各类解决方案，但这一问题仍然存在。由于我们对于车辆的路径规划原则并不悉知，同时也不能实时观察到道路情况，不能查看无法出发的车辆的更新规划，我们始终无法分析出该问题出现的原因，这个问题也存在于我们的最终提交中，它可能会对最终结果产生或正向或负向的影响（如果无法出发的车辆运行时间较短，则最终结果可能略优；如果无法出发的车辆运行时间更长，则可能最终结果略差）。