第六届湖南省研究生数学建模竞赛

题 目 交通信号灯全局优化调控策略

摘 要：

随着社会经济的发展，城市交通问题越来越引起人们的关注，如何采用合适的交通信号灯调度方法，最大限度缓解城市的交通拥堵状况成为了交通运输管理和城市规划部门亟待解决的主要问题。本文主要研究交通信号灯全局优化调控策略问题，具体包括两个问题，一是在给定路网结构、车辆行程计划等信息的条件下来设计交通信号灯全局调控策略，二是在道路发生故障时设计局部交通信号灯调控策略。期望在无故障情况下或者发生故障的情况下，利用交通信号灯全局调控策略或局部调控策略，最小化城市车辆通行总时间，从而实现城市车辆总得分最大化。

针对问题一，本文在题目的简化假设条件下建立数学模型，首先生成随机的初始交通灯调度策略，然后利用遗传算法，将每个交通灯调度策略看做单独的基因，通过多次的交叉、遗传、变异，最终得出一个最优的全局交通灯调度策略。本文的实验结果表明，除一辆车超时外，其余车辆分数行驶过程都比较顺畅，在某些路口略有停顿，较好的解决了全局交通灯调度策略问题。

针对问题二，本文在问题一的基础上，对故障产生后的情况建立新的数学模型。本文先找出需要修改路径的车辆，然后利用堆优化的 Dijkstra 算法，找出对应车辆的最短后续路径，将其最短后续路径上的交通灯策略作为局部优化对象，最后通过遗传算法，找出一个局部最优的交通灯调度策略。根据实验结果，新规划的路线等待时间分别为 4 秒、15 秒和 16 秒，说明了故障发生后局部交通灯调度策略能够很好地满足题目要求。

最后，本文对模型的优缺点进行了分析评价，优点是本文的模型用数学语言完善的描述了交通灯调度策略问题，可以较为直观的解决无故障和故障发生两种情况下的交通灯调度策略问题，最终能够得到最优的交通灯调度策略，缺点是交通灯调度策略的设计空间较大，自变量较多，本文建立的模型需要花费较长时间进行多次迭代才能收敛到全局最优。

关键词：交通灯调度策略，遗传算法，堆优化，Dijkstra 算法

一、问题背景和重述

1.1 问题背景

交通信号灯在城市交通中发挥了巨大的作用，可以说是无处不在。现在很多城市的交通信号灯已经互联，可以远程控制每一盏灯每一种信号的时长。能否实现对交通灯的全局优化调控，以提高道路通行能力，减少通行时间，是一个亟待解决的问题。

1.2 问题的简化假设条件

（1）城市路网由路口和道路组成，路口和道路均有唯一的名称。道路与路口相连，路口是道路的起点或终点。每一条道路均为单向通行，连接两个不同的路口。但是可能有方向相反的两条路连接相同的两个路口，这对应了现实中双向通行的道路。

（2）连接两个路口的道路，同一方向至多有一条。每条道路中间不会与其他道路交叉。如果出现道路不交叉无法在平面上排布的情况，可对应现实中的隧道或者立交桥。

（3）每个路口至少是一条道路的终点，另一条道路的起点。

（4）车辆行驶速度相同，因此可用畅通状态下的通行时间表示一条道路的长度。并且，车辆行驶速度不受其他车辆影响。

（5）每条道路对应设置一个交通灯，用于控制该条道路上的车辆能否进入该道路终点处的路口。由于道路与交通灯一一对应，因此使用道路名称指代其交通灯。

（6）路口的交通灯指的是所有以该路口为终点的道路的交通灯。

（7）交通灯只有红色、绿色两种。红灯表示停止，交通灯为红色时，车辆要停在所在道路上，不能穿过该道路终点的路口。绿灯表示车辆可以通行，车辆可以穿过该路口，并开往以该路口为起点的其他道路。

（8）当一条路的交通灯为红灯时，所有到达该道路终点的车辆排队等待绿灯，绿灯之后可以通行。

（9）在任何时刻，每个路口最多只有一个交通灯是绿灯。当一条道路的绿灯亮起时，只有这条道路上的车辆可以通行，并开往其他道路，其他道路的交通为红色，其他道路上准备进入该路口的车辆都必须排队等待绿灯之后才可以通行。

（10）忽略车辆的几何尺寸，不考虑车辆排队时候的空间要求。

（11）当交通灯变为绿灯之后，排队等待的车辆中的第一辆车可以立即通过路口，没有延迟，每一辆车通过路口的时间大小为一秒，即，如果绿灯持续 n 秒，则排队等待的队列中前 n 辆车恰好能够通过路口，其他正在排队等候的车辆只能等待下一个绿灯。

（12）已知全部车辆的行程计划，即，由一系列道路组成的行驶路径，并且不能改变。

（13）每辆车的行程经过任何一个路口至多一次。

（14）初始状态，即在 0 时刻，所有的车辆都在各自路径上第一条道路的终点处，该道路的交通灯若为红灯，则等待绿灯；若为绿灯，则准备移动。如果有多辆车按照自己的行程计划行使，遵守交通灯的指示，到达路径中最后一条道路的终点为止，其行程结束，立即被移出，不再考虑。即，车辆不需要在最后一条道路的终点处排队，也不需要进入下一个路口。

1.3 问题的调度方案要求

每个路口可以独立设置交通灯的调度方案。一个路口的交通灯调度方案是该路口的交通灯在一个周期内亮起绿灯的顺序以及持续时长，然后不断重复。调度方案是一个列表，每一项包含交通灯所在道路名称及绿灯持续时间，按照绿灯亮起的顺序排列。每个交通灯在一个周期内至多亮起绿灯一次。所有交通灯的初始状态为红灯。因此，某些交通灯可以不在方案中出现，表示这些交通灯一直保持红灯状态，不出现绿灯。

1.4 问题重述

问题 1：假设全市路网结构、车辆的行程计划、车辆行驶速度等信息都不变，且已知，每个路口的交通信号灯都以各自独立控制的固定周期变换信号，研究交通灯全局优化调控策略，分析算法适用的城市路网规模，给出具体的调度方案。记 D 表示总时长，若某辆车 j，在 Tj 时刻到达终点，F 为车辆按时到达终点的基本得分，则该车的得分为：

$$M_j = \begin{cases} F + (D - T_j), & T_j \le D \\ 0, & T_j > D \end{cases} \tag{1}$$

问题一的优化目标如下，其中，V 为车辆总数。

$$Max \sum_{j=1}^{V} M_j \tag{2}$$

问题 2：若某时刻某条道路上突发道路设施故障，需要立即维修，导致该道路无法继续通行，在问题 1 的基础上，讨论此类事件对交通灯调度方案的影响，设计局部调整调度方案的方法，给出调整后的调度方案。

二、问题假设与符号说明

2.1 问题假设

（1）假设车辆不需要反应时间，交通灯变绿后最在前面排队的车辆瞬间通过，交通灯变红后车辆不会出现闯红灯的状况。

（2）假设所有车辆匀速行驶，不通车辆的速度相同，不考虑因为地形、天气和人为等因素造成加速减速或者故障的情况。

（3）假设连续几个交通灯较近不会带来影响，在交通灯调度过程中只需考虑优化车辆得分。

（4）假设没有车辆经过的道路交通灯全为红色。

（5）假设问题 2 中发生故障后，经过该道路但尚未通过的车辆在重新选择后续行使路径时只依据“里程最短”原则，而不依据交通灯的密集程度或其他原因。

（6）假设问题 2 中发生故障后的道路在该问题全过程中都无法再次通行。

2.2 符号说明

符号	说明
D	总时长，单位秒
T_j	车辆 j 到达终点时刻

P_j	车辆 j 按时达到终点的基本得分
I	车辆总数
S	道路总数
B	道路起点的路口 ID
E	道路终点的路口 ID
L	车辆从该道路起点到终点所需的时间，单位秒
$C001$	车辆 ID，以大写字母 C 开头，之后三位整数。
P	车行程路径的道路总数
J_i	指代交通灯
P_i	交通灯 i 绿灯持续时间
Z_{Ji}^{A}	包含交通灯 Ji 的路口 A 的交通灯调度周期
S^A	路口 A 的交通灯状态
C_{Jn}	交通灯 J_n 的颜色，1 表示绿色，0 表示红色
T_i^J	车辆 i 在交通灯 j 处等绿灯的时间
T_j^P	车辆 j 的排队等待时间
T_{ij}^L	车辆 i 在道路 j 上的通行时间
T_j^Z	车辆 j 的总通行时间
M_j	车辆 j 的总得分
L^*	问题 1 总路径集合
T_i^{ebf}	存在路径 ebf-ffef 时，车辆 i 到路口 ebf 的时间
R^*	存在路径 ebf-ffef 的车辆
R^{**}	在 ebf-ffef 的车辆，且 T=30 秒还未通过的车辆子集
V_i^{ebf}	车辆 i 从路口 ebf 到终点的道路集合
$T_i^L(v_i)$	车辆 i 以 v_i 路径从 ebf 到达终点的道路长度
$T_i^Z(v_i)$	车辆 i 以 v_i 路径从 ebf 到达终点的道路总长度

三、问题 1 建模与求解

3.1 问题分析

问题 1 简化了实际问题，不考虑全市路网结构、车辆行使速度等因素的变化，只考虑道路长度，交通灯的变化周期等因素。每个路口的交通灯都以各自独立控制的固定周期变换信号，车辆按照所给规则行使，问题 1 需要我们给出所有的交通灯的变化周期的方案，优化目标是所有车辆的总同行时间，需要考虑的条件是

同一路口交通灯只有一个为绿色和车辆的行使规则等。

交通灯的调度需要我们根据车辆评分来确定，要达到一个较高的车辆得分，就需要车辆的总通行时间最短，而总通行时间又由车辆的道路通行时间，车辆的路口通行时间和排队托或者等待交通灯的时间。我们可以从车辆的总通行时间下手，先初始化一个交通灯调度方案，然后利用遗传算法等解决方案来根据所达到的车辆得分来优化初始交通灯调度方案，多次迭代后可以获得一个较为好的交通灯调度方案。

3.2 模型建立

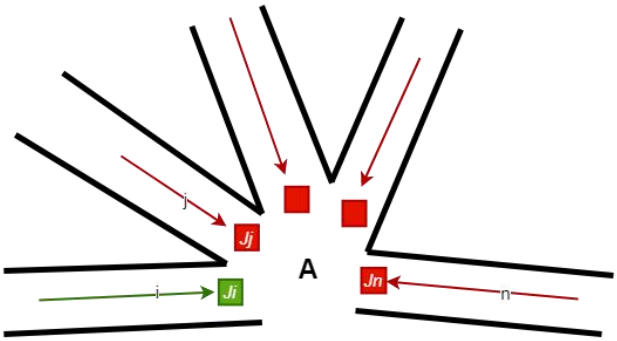

图 1 路口示意图

（1）交通灯策略

当一个路口只是一条道路的终点时，那么调度策略就比较简单，此路口可以设置全为绿灯，，这里不需要此类情况讨论。当一个路口是多条道路的终点时，当每条道路的绿灯先后顺序和绿灯持续时间确定时，就可以确定该条路口的交通灯策略，就可以通过该策略得到任意时刻该路口所有交通灯的颜色，从而可以确定在任意时刻该口路需不需要等待或者需要等待多久时间才能通行。

如图 1 所示，当路口 A 为多条道路（i，j，…，n）的终点时，若道路i上的交通灯J_j先亮绿灯，道路j上的交通灯J_j第二个亮绿灯，……，道路n上的交通灯J_n最后一个亮绿灯，交通灯J_j的绿灯持续的时间为P_i，交通灯J_j的绿灯持续时间为P_{j}，……，交通灯J_{n}的绿灯持续时间为P_{n}，则此路口 A 的交通灯周期为$Z_{J_{\mathrm{i}}}^{A}$

$$Z_{Ji}^{A}=P_i+P_j+\cdots+P_n=\sum_{k}^{k\in\{i,j,\cdots,n\}}P_k \tag{3}$$

交通灯J_i在时刻 T 的颜色C_{j_i}，其中 1 表示绿色，0 表示红色：

$$C_{Ji}=\begin{cases}1,0\le\left(T\ \%Z_{Ji}^{A}\right)<P_{Ji}\\0,P_i\le\left(T\ \%Z_{Ji}^{A}\right)<Z_{Ji}^{A}\end{cases} \tag{4}$$

交通灯J_j在时刻 T 的颜色C_{j_j}：

$$C_{\mathrm{Jj}}=\begin{cases}1,\ \ P_i\le\left(T\%Z_{\mathrm{Ji}}^{A}\right)<P_i+P_j\\0,0\le\left(T\%Z_{\mathrm{Ji}}^{A}\right)<P_i,\ \ P_i+P_j\le\left(T\%Z_{\mathrm{Ji}}^{A}\right)<Z_{\mathrm{Ji}}^{A}\end{cases} \tag{5}$$

交通灯J_n在时刻 T 的颜色C_{Jn}：

$$C_{Jn}=\begin{cases}1, & P_n \le \left(T\%Z_{0i}^{A}\right) < Z_{Ji}^{A} \\ 0, & 其他\end{cases} \tag{6}$$

此时路口 A 的交通灯的状态 S^A 可表示为

$$S^{A}=\begin{cases}(1,0,\cdots,0), & 0 \le \left(T\%Z_{Ji}^{A}\right) < P_i \\ (0,1,\cdots,0), & P_i \le \left(T\%Z_{Ji}^{A}\right) < P_i + P_j \\ \cdots \quad \cdots & \\ (0,0,\cdots,1), & Z_A - P_n \le \left(T\%Z_{Ji}^{A}\right) < Z_{Ji}^{A}\end{cases} \tag{7}$$

（2）在交通灯处的等待时间

当一辆车行驶到一条道路的终点，即一个路口的交通灯前时，如果该交通灯为绿色，则不需要等待，此时等待时间为 0，如果该交通灯为红色，则需要等待交通灯变成绿色才可以通行。若 T_{ji}^{Ji} 时刻一辆车 j 到达一个路口 A，车辆 j 在路口 A 等待的交通灯为 Ji，Ji 为此路口第 n+1 个变为绿灯的交通灯，Z_{Ji}^{A} 为包含交通灯 Ji 的路口 A 的交通灯的周期，此时需要等待交通灯 Ji 变绿时间可表示为

$$T_j^J=\begin{cases}\sum_{k=0}^{n}P_k - T_{ji}^{Ji}\%Z_{Ji}^{A}, & T_{ji}^{Ji}\%Z_{Ji}^{A} < \sum_{k=0}^{n}P_k \\ Z_{Ji}^{A} - T_{ji}^{Ji}\%Z_{Ji}^{A} + \sum_{k=0}^{n}P_k, & T_{ji}^{Ji}\%Z_{Ji}^{A} \ge \sum_{k=0}^{n+1}P_k \\ 0, & \sum_{k=0}^{n}P_k \le T_{ji}^{Ji}\%Z_{Ji}^{A} < \sum_{k=0}^{n+1}P_k\end{cases} \tag{8}$$

其中 P_1、P_2、P_3……为前 n 个交通灯的绿灯持续时间。在一个交通灯策略确定好后，等待时间是车辆经过这个路口需要额外付出的时间，等待时间越小越好。

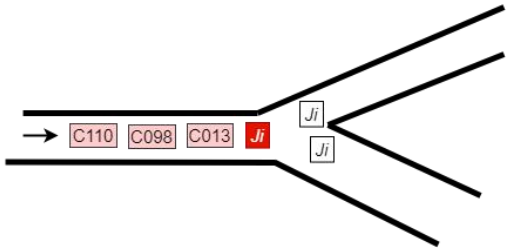

图 2 车辆排队等待示意图

（3）排队等待时间

如图 2 所示，当至少两辆车同一时刻出现在同一条道路终点路口处的时候，后面的车会发生等待，等待前边的车辆通过该路口之后此辆车才可通行。ID 号小的排在前边先通过，ID 号大的车辆排在后边通过，此时 ID 号大的车会发生等待，产生额外的等待时间，此时间会影响车辆的总通过时间，会降低车辆通过的总得分。若在 T 时刻车辆 j 前面有 m 辆车且出现同一路口，车辆 j 等待时间可以表示为：

$$T_j^P = m\ ,\ \ m \ge 0 \tag{9}$$

其中 m 为车辆 j 前面的车辆数。因为问题规定了每辆车通行的时间为 1 秒，那么车辆前边有 m 辆车时，就需要等待 m 秒，等待时间为 m。

（4）道路通行时间

车辆从起点路口开往终点路口时，根据道路长度的不同，有不同的通行时间。由于问题规定车辆的速度不变，规定车辆在道路上的通行时间等于道路的长度。查看 streets.txt 可以得知车辆在某一条道路上的通行时间。车辆 i 在道路 j 上的道路通行时间为

$$T_{ij}^{L} = L_j, \quad L_j \in L^* \tag{10}$$

L^* 为所有道路的集合，其中 L_j 通过查询附录 2 streets.txt 文件可得。

（5）车辆总通行时间

车辆总通行时间是影响车辆最后得分的关键指标，车辆的总通行时间包括了 200 辆车各自通行时间的总和，而每一辆车的通行时间又包括车辆的交通灯等待时间，排队等待时间和道路通行时间。v_i 为车辆走的路径，$U(v_i)$ 表示车辆 j 走路径 v_i 要经过的所有路口，车辆 j 的总通行时间可以表示为

$$T_j^Z = \sum_{i}^{i \in U(v_i)} \left(T_{ji}^{J} + T_{ji}^{P} + T_{ji}^{L} \right) \tag{11}$$

（6）优化目标

问题的优化目标是车辆的得分，问题要求改变交通灯的策略来使得得分最大化。D 表示总时长，若某辆车 j，在 Tj 时刻到达终点，则该车的得分为

$$M_j = \begin{cases} F + (D - T_j), & T_j \le D \\ 0, & T_j > D \end{cases} \tag{12}$$

其中 F 为车辆按时到达终点的基本得分，F 和 D 可根据附录 2 中 parameters.txt 文件可得分别为 250 和 856，

$$T_j = T_j^{Z} \tag{13}$$

则问题一的优化目标为：

$$Max \sum_{j=1}^{V} M_j \tag{14}$$

即问题一可建模为：

$$Max \sum_{j=1}^{V} M_j ,$$

$$
s.t.\begin{cases}
T_{ji}^{J}=\begin{cases}\sum_{k=1}^{n}P_k-T_{ji}^{Ji}\%Z_{Ji}^{A}, & T_{ji}^{Ji}\%Z_{Ji}^{A}<\sum_{k=1}^{n}P_k\\ Z_{Ji}^{A}-T_{ji}^{Ji}\%Z_{Ji}^{A}+\sum_{k=1}^{n}P_k, & T_{ji}^{Ji}\%Z_{Ji}^{A}\geq\sum_{k=1}^{n+1}P_k\\ 0, & \sum_{k=1}^{n}P_k\leq T_{ji}^{Ji}\%Z_{Ji}^{A}<\sum_{k=1}^{n+1}P_k\end{cases}\\
T_{ji}^{P}=m\\
T_{ij}^{L}=L_j\\
T_j^{Z}=\sum_{i}^{i\in U(v_i)}\left(T_{ji}^{J}+T_{ji}^{P}+T_{ji}^{L}\right)\\
M_j=\begin{cases}F+\left(D-T_j\right), & T_j\leq D\\ 0, & T_j>D\end{cases}\\
T_j=T_j^{Z}\\
m\geq 0,\ n\geq 0\\
L_j\in L^{*}
\end{cases}
\tag{15}
$$

3.2 数据集分析

该问题的求解空间较大，情况比较复杂，通过数据集分析我们可以发现不同的路口存在不同的情况，通过分析这些情况能够对有效地缩减问题的求解空间，简化要处理的数据，提高算法的效率。

通过简单的统计，在给的数据集中一共有 200 多辆存在 2000 多个路口，但是大多数路口的情况比较简单，例如下面图 3 的情况，虽然可能有很多车辆经过该路口，但是从每条道路上来的车辆都只有一辆，在这种情况下，很容易证明得到，对于某一路口有 n 条道路 $l_1,l_2,l_3,.....,l_n$ 的情况下，如果每条路会经过的车辆数 $C_k=1$，那么根据题目要求，就总能找到一种交通灯调度方案使得每辆车到达时刚好为绿灯，能够直接通过。

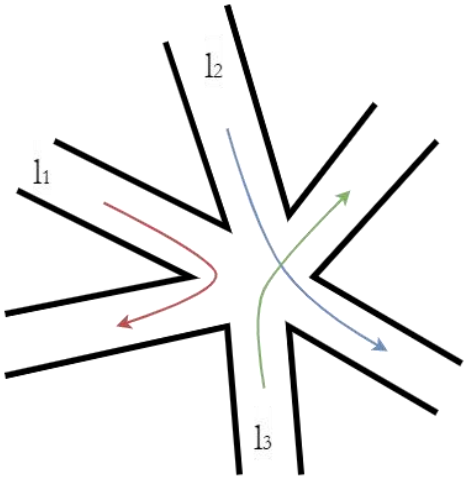

图 3 路口车辆情况示例

那么对于道路数量$l_{\max} \geq 2$且至少一条道路的车辆数$C_k \geq 2$的情况，例如下面图 4 的示例，就可能找不到一种合适的调度方案能够使得每辆车到达时恰好为绿灯，这种路口我们可以认为可能产生冲突，所以我们只需要先考虑这些特殊的路口交通灯的调度方案。通过大概统计这种点大概有 70 个，分布如图 5 所示。

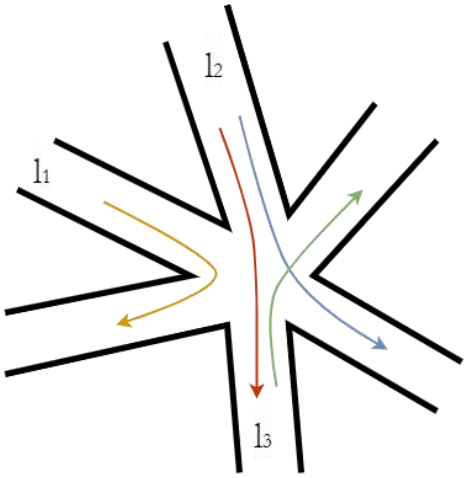

图 4 路口车辆情况示例

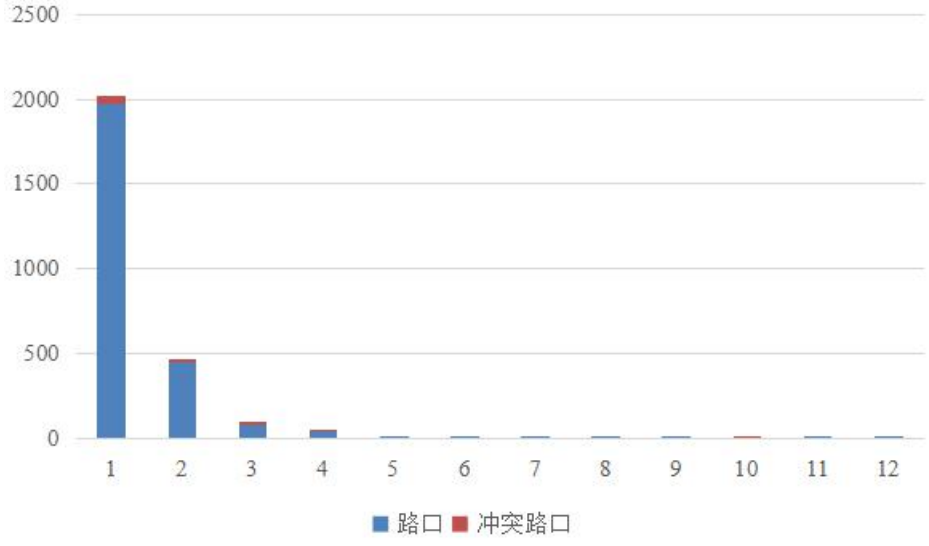

图 5 路口情况统计

还有再考虑路口交通灯调度策略的相互影响，分析一些比较简单的情况，示例如下面图 6 所示，如果先调整其中一个交通灯的调度策略，那么其中一条通往

其他路口的到达时间也就会随之改变，势必影响另一个路口的交通灯调度策略。

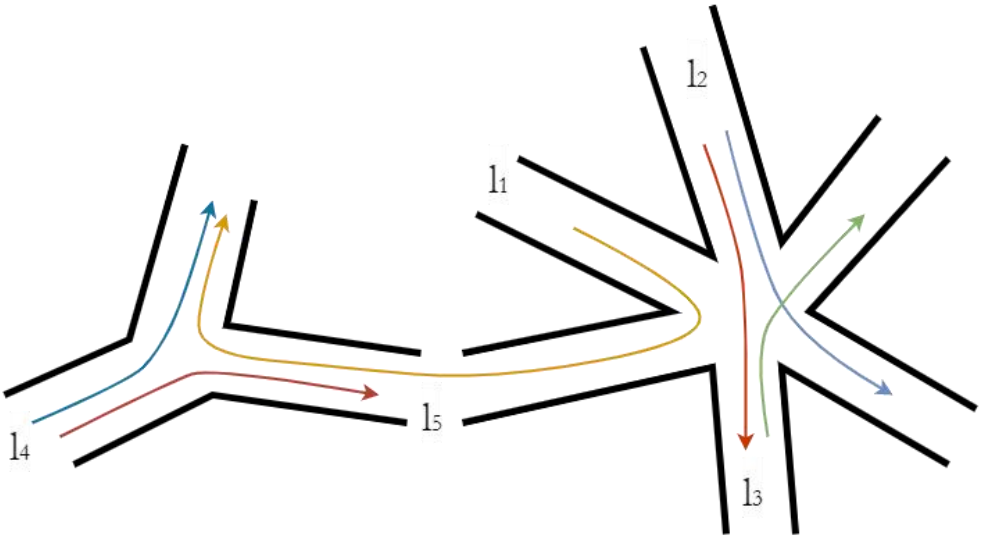

图 6 多路口关联示例

3.3 求解思路

通过对数据集的分析可以对整个求解空间进行简化，通过对路口的筛选对数据集进行预处理，将那些不会产生冲突的点进行与所连接的道路先进行融合，这样就可以认为是一条合成的道路，这样处理之后就可以得到由 70 个点组成的简化图状结构。

根据对数据集的分析发现如果采用搜索的求解方法不太可取，因为难以对每个路口交通灯的调度策略依次进行规划，也不能根据时间进行动态调整，因为每一个调度策略都会相互调整。因此在这种情况下难以逐步进行调整，只能通过一次次迭代不断逼近最优的全局调度。

解决这种问题可以用到的算法有很多，遗传算法，模拟退火算法，蚁群算法，粒子群算法等。遗传算法是根据大自然中生物体进化规律而设计提出的，模拟达尔文生物进化论的自然选择和遗传学机理的生物进化过程的计算模型，是一种通过模拟自然进化过程搜索最优解的方法。该算法通过数学的方式，利用计算机仿真运算，将问题的求解过程转换成类似生物进化中的染色体基因的交叉、变异等过程。在求解这种较为复杂的组合优化问题时，可以将每个路口的调度策略看作一类基因，通过不断组合、变异和适应进行优化，最终得到一个最优的交通灯调度策略组合。

3.4 求解算法介绍

我们针对该问题所采用的算法是遗传算法，其中的基因是由那 70 个路口的交通灯调度策略组成。

3.4.1 算法整体过程

算法整体框架采用的是遗传算法，细节处略有不同，下面图 7 是整体大概的算法流程：

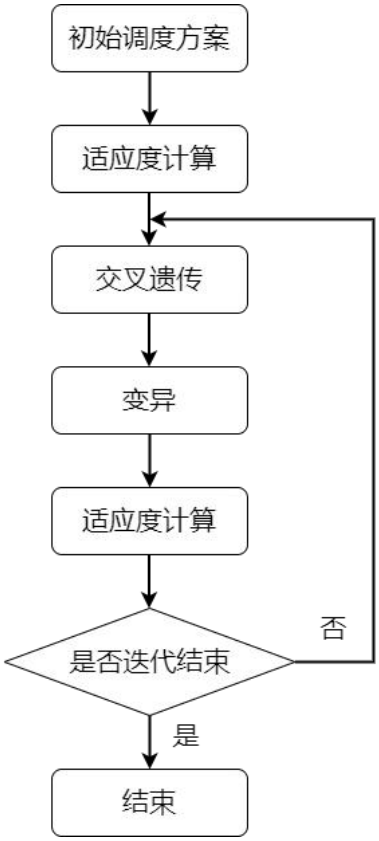

图 7 遗传算法流程图

3.4.2 种群基因表示

一个种群的基因是由各个路口的交通灯调度策略组成的，我们可以将其设计成一个类结构，所以一个种群的基因的长度为 70，即需要考虑冲突的路口数。

依照题目要求，单个基因的状态空间可以是无限的，因为周期时长没有确定，所以要在求解的过程中对周期设置一个合理的上限，初始的基因状态可以随机从设置的状态空间当中进行选取。这部分与一般的遗传算法不太一样，基因状态空间的大小会最终影响收敛效果，可能需要很长的收敛时间才能够找到较好的基因状态。

3.4.3 种群适应度计算

种群适应度代表了该种群的基因组在当前条件的适应能力，在该问题中就是最终分数的计算，我们在确定了调整策略之后就可以计算好每辆车在完成对应的路线之后的分数，我们就用这所以车的分数和当成该种群的评价分数。

而某个种群适应度的计算就是该种群分数与所有种群分数的比例，适应度越大，那么该种群的基因遗传给下一代的概率就越大。

3.4.3 调度策略表示和计算

按照题目的要求，交通灯的调度方案由道路名称顺序和对应绿灯时间构成。在知晓到达该路口时间的情况下，我们可以计算出每条道路上的车等待时间，用以计算最后分数。交通灯调度策略示例如下表：

表 1 交通灯调度示例

道路名称	eff-ad	acd-ad	df-ad	eca-ad
绿灯时间/s	4	6	2	4

如果道路 df-ad 上车辆到达时间为T_{df-ad}，那么该车在该道路的堵车时间可以计算为：

$$\nabla t=\begin{cases}0,4\le T_{df-ad}\%T_{总}<10\\(4+T_{总}-T_{df-ad}\%T_{总})\%T_{总},其他\end{cases} \quad (16)$$

3.4.4 变异的选取

变异是遗传算法中很重要的一环，能够跳出局部最优进而搜索全局最优解。在该问题中，单个基因的变异就是该路口的交通灯调度策略的变化，我们所采用的变异方案就是从状态空间中随机选取一个状态当作变异的调度策略，这种变异策略也是具有一定合理性的，因为变异的方向是不确定的，而且大多数变异都是朝着坏的方向，而随机从状态空间中选取大多数也不一定能够使整体分数变大使适应度提高。

而变异率则表示的是每次交叉遗传后每个基因的变异概率，虽然单个基因的变异概率很小，但是由于基因个数多导致种群中总会有基因产生变异从而可趋向更优解。

3.6 实验过程与结果分析

根据上面提出的求解算法进行实验，计算最终全局交通灯调度的最优解。在实验中我们设置了 100 个种群，变异率为 0.01，迭代次数为 1000 次，下面图 8 是迭代次数与全局得分的一个关系图，可以看出随着迭代次数的增加，得分是不断提升的，虽然提升过程较慢，而且过程中波折不断，这些都是因为每个基因的状态空间过大导致的，因为分数的设置不太合理，整体所有种群适应度也没有很大区分。

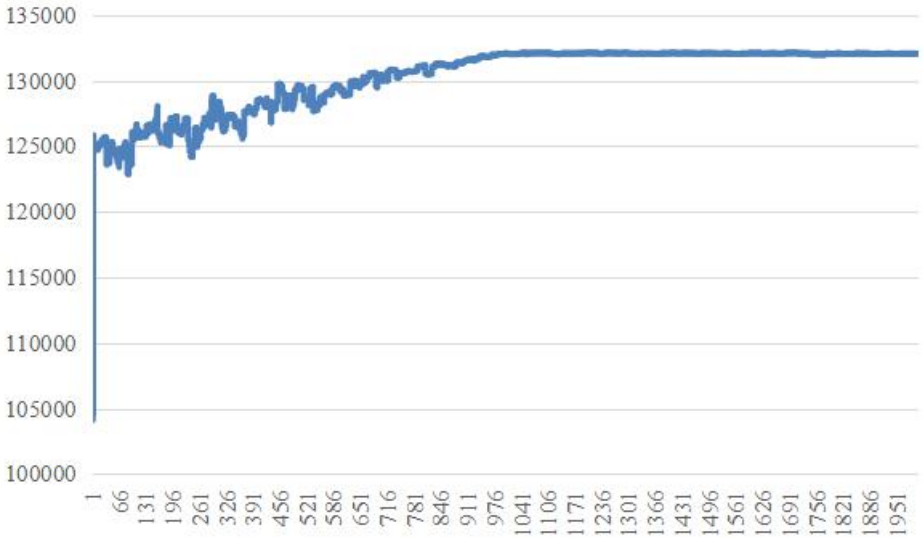

图 8 总得分变化图

为验证全局交通灯调度策略趋于最优，我们对各个容易产生冲突的路口车辆总等待时间进行统计如图 9 所示。可以看出大多数的路口交通灯调度策略以达到最优，因为车辆在这些路口的等待时间都是零，另外还有极小一小部分路口会有车等待，但是等待时间并不大，通过对比数据可以发现这些路口情况都比较复杂，很容易产生冲突，对于这些路口难以完全调度为顺畅通过，因此我们这种调度方案是比较可行的，可以认为全局最优。

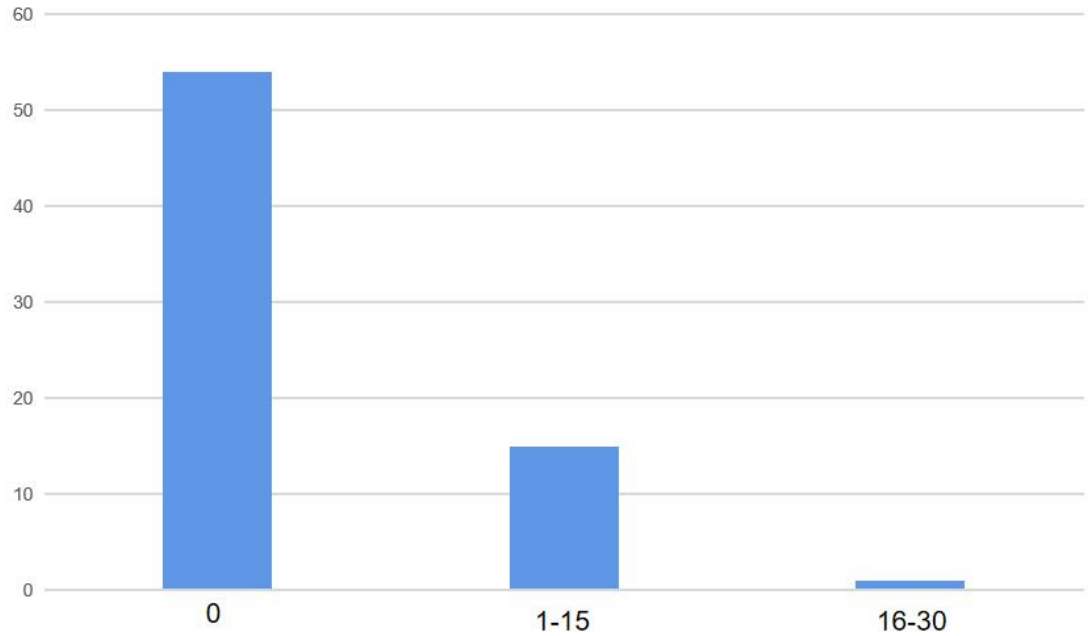

图 9 等待时间统计图

四、问题 2 建模与求解

4.1 问题分析

问题 2 在问题 1 的基础上加入了突发道路设施故障，在某一时刻道路的信息会改变，如果以这一时刻的状态为新的起点，那么这就和问题 1 类似。在发生故障之后，其中一条路段无法通行，已经通过故障路口的车辆将继续前进，不在新的调度方案中考虑，问题 2 的新调度方案中只需要考虑原本需要通过该路口但是因为故障需要修改路径的车辆，问题 2 可以通过局部调整调度方案来优化车辆的总通行时间。

在 30 秒发生故障之后，已经到到达或者通过故障道路的车辆不受故障影响，受到故障影响的车辆需要进行最短路径选择和局部车辆得分优化，最短路径选择可以通过贪婪等算法求得，局部车辆得分的优化可以在找出最短路径作为受故障影响车辆的后续路径后，根据问题 1 的解法求个一个局部较好的车辆得分，从而获得一个较好的交通灯调度方案。

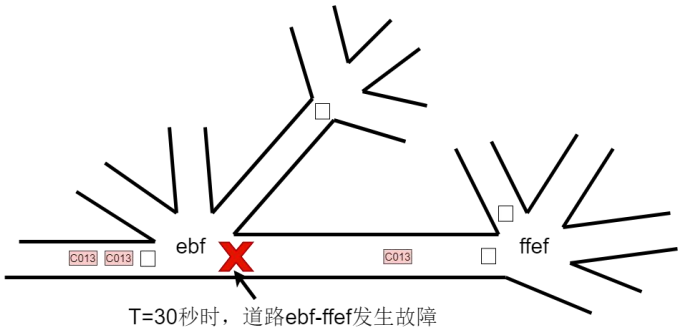

图 10 路段故障示意图

4.2 问题模型

如图 8 所示，ebf-ffef 道路在时刻 30 秒发生故障。故障发生后，该道路立即永久封闭，其他道路正常通行。故障发生时，已经通过路口 ebf 的车辆按原计划继续行使，道路故障不对这些车辆造成影响。原定计划中有 ebf-ffef 这条路的车

辆但是在时刻 30 秒时还没通过路口 ebf 的车辆将将受到道路故障的影响，这些车辆需要按照路径最短原则重新选择后续路径。

在 30 秒前，交通灯调度同问题一，在 30 秒故障发生后，需要新的调度。

（1）车辆到达 ebf 路口所需的时间

发生故障后只需要考虑部分车辆，现在我们需要找到所有原本路径中需要途经 ebf-ffef 的车辆。根据附录 2 中 cars.txt 文件遍历查询存在 ebf-ffef 的车辆，得到一个车辆的子集 R^*，现在只需要考虑集合 R^* 中的车辆。首先需要找出集合 R^* 中 T=30 秒还未通过 ebf 的车辆。车辆 i 到路口 ebf 的时间为 T_i^{ebf}，则

$$T_i^{ebf} = \sum_{j=1}^{q} \left(T_{ij}^J + T_{ij}^P + T_{ij}^L \right) \tag{17}$$

其中 $i \in R^*$，q 为车辆 i 到路口 ebf 一共需要经过的路口数量，可以通过查询附录 2 中 cars.txt 文件得到，T_{ij}^J、T_{ij}^P、T_{ij}^L 分别为车辆到达路口 ebf 前，经过一个路口所需的交通灯等待时间，经过一个路口的所需的排队等待时间，经过路口前的道路通行时间，这三个数据通过问题 1 获得。

（2）T=30 秒还未通过路口 ebf 的车辆子集

根据问题 2，故障发生后车辆不能进入 ebf 路口，而 T_i^{ebf} 是车辆 i 刚进入路口 ebf 所需的时间，因此需要将 T_i^{ebf} 减去 1 秒，即可得知车辆 i 到达 ebf 路口前所需的时间。通过将这个时间与 30 秒相比，就可得知哪些车辆在 T=30 秒故障发生时还没有通过路口 ebf。令 T=30 秒还没通过路口 ebf 的车辆的集合为 R^{**}，则

$$R^{**} = \left\{ i | i \in R^*, T_i^{ebf} - 1 \le 30 \right\} \tag{18}$$

（3）路径规划

根据问题 2，所有计划经过 ebf-ffef 道路还未经过的车辆，到达路口后立即按照里程最短原则重新选择后续行驶路径，这里是对属于集合 R^{**} 的车辆的路径规划。通过读取附录 2 中 streets.txt 文件和 cars.txt 文件可知路网信息和车辆的目的地，本文需要根据路网信息，找到每一辆车到达其目的地的最短路径。由于道路比较多，不能直接通过遍历的方法找出每个车各自的最短路径。

令 V_i^{ebf} 为车辆 i 从路口 ebf 到终点的道路集合，v_i 为 V_i^{ebf} 其中一种走法，j 为 v_i 中道路之一，则车辆 i 以路径 v_i 从 ebf 到达终点的道路时间 $T_i^{L(v_i)}$ 为：

$$T_i^L(v_i) = \sum_{j}^{j \in v_i} L_j \ , \ \ v_i \in V_i^{ebf} \ , \ \ i \in R^{**} \tag{19}$$

其中 L_j 为道路 j 的长度通过读取附录 2 中 streets.txt 文件获得。

$U(v_i)$ 为路径 vi 中包含的路口数，因为车辆通过每一个路口的最短时间为 1 秒，相当于路径长度加一，在计算最短路径的时候需要包含路口的数量。车辆 i 以路径 v_i 从 ebf 到达终点的总长度 $T_i^Z(v_i)$ 为：

$$T_i^Z(v_i) = T_i^L(v_i) + U(v_i) * 1 = T_i^L(v_i) + U(v_i) \ , \ \ v_i \in V_i^{ebf} \ , \ \ i \in R^{**} \tag{20}$$

要寻求最短路径则需要遍历集合 R^{**} 求得：

$$Min\left(T_i^Z(v_i) \right) , \ \ v_i \in V_i^{ebf} \ , \ \ i \in R^{**} \tag{21}$$

求得的最短路径之后，本文即可对最短路径上的交通灯进行局部调度。

重新调度后车辆总通行时间根据问题一的解题过程可知，车辆总通行时间影响车辆最后得分，车辆的总通行时间包括了 200 辆车各自通行时间的总和，而每一辆车的通行时间又包括车辆的交通灯等待时间，排队等待时间和道路通行时间。若 e 表示车辆 j 要经过的所有路口，车辆 j 的总通行时间可以表示为

$$T_j^Z = \sum_{i=1}^{\mathrm{e}} \left(T_{ji}^J + T_{ji}^P + T_{ji}^L \right) \tag{22}$$

（4）优化目标

问题的优化目标是车辆的得分，问题要求改变交通灯的策略来使得得分最大化。D 表示总时长，若某辆车 j，在 Tj 时刻到达终点，则该车的得分为

$$M_j = \begin{cases} F + \left(D - T_j \right), & T_j \le D \\ 0, & T_j > D \end{cases} \tag{23}$$

其中 F 为车辆按时到达终点的基本得分，F 和 D 可根据附录 2 中 parameters.txt 文件可得分别为 250 和 856，

$$T_j = T_j^{\mathrm{Z}} \tag{24}$$

问题二的优化目标为

$$Max \sum_{j=1}^{V} M_j \tag{25}$$

则问题二整个模型可以建立为：

$$Max \sum_{j=1}^{V} M_j$$

$$
s.t.\begin{cases}
Min\left(T_i^Z\left(v_i\right)\right) \\
T_{\mathrm{k}}^{ebf}=\sum_{j=1}^{q}\left(T_{kj}^J+T_{kj}^P+T_{kj}^L\right) \\
R^{**}=\left\{u|u\in R^*,T_u^{ebf}-1\le 30\right\} \\
T_i^L\left(v_i\right)=\sum_{j}^{j\in v_i}L_j \\
T_i^Z\left(v_i\right)=T_i^L\left(v_i\right)+U\left(v_i\right)*1=T_i^L\left(v_i\right)+U\left(v_i\right) \\
T_j^Z=\sum_{i=1}^{\mathrm{e}}\left(T_{ji}^J+T_{ji}^P+T_{ji}^L\right) \\
M_j=\begin{cases}F+\left(D-T_j\right), & T_j\le D\\ 0, & T_j>D\end{cases} \\
T_j=T_j^{\mathrm{Z}} \\
k\in R^* \\
i\in R^{**} \\
v_i\in V_i^{ebf}
\end{cases}
\tag{26}
$$

4.3 求解思路

在解决完问题 1 之后，问题 2 的解决相对比较简单，但是需要对优化对象进行筛选。问题 1 已经找出了所有车辆在所有路径都正常通行的情况下，交通灯的最优调度策略，在发生故障之后，要考虑的路径变了，要优化的交通灯变少了。通过对数据集的分析，本文发现涉及到故障路段的车辆较少，加上在 30 秒故障发生时，已经有部分车辆通过该路段，因此所需考虑调度的交通灯较少。

问题 2 在考虑调度交通灯之前，先需要找到具体的优化路段，首先需要找出在 30 秒故障发生后将要通过故障路段但是还没有通过故障路段的车辆，在故障发生之后，这些车辆的后续行使路径需要改变。可以根据车辆的目的地，利用求最短路径的算法求出最短路径，然后在问题 1 已有的解题方法稍作改进即可解决问题 2。

解决问题 2 中最短路径的算法有很多，包括 Dijkstra 算法、SPFA 算法\Bell man-Ford 算法、Floyd 算法\Floyd-Warshall 算法、Johnson 算法、A*算法等，以上算法都可以解决问题 2 中求最短路径的问题。Floyd 等算法实现起来简单，但是其效率较低，根据丰富的使用经验，本文选择 Dijkstra 算法求解车辆的最短路径问题。再求出最短路径后，再次使用问题 1 中的遗传算法，对问题 2 进行局部的优化调度。

4.4 求解算法

问题 2 使用了一个优化的 Dijkstra 算法和问题 1 使用过的遗传算法，问题 1 中已经介绍过遗传算法的使用，这里不在赘述。本节主要讲优化的 Dijkstra 算法

在调度车辆后续最短路径的问题上的应用。

4.4.1 最短里程算法

Dijkstra 算法使用了广度优先搜索解决赋权有向图或者无向图的单源最短路径问题，算法最终能够得到一个最短路径树。该算法常用于路由算法或者作为其他图算法的一个子模块。但是 Dijkstra 算法具有一个很大的不足，那就是在有向图很大的情况下，整个算法所耗费的时间会大大增加，所以要降低时间复杂度，用空间换时间。

在该问题中采用的方法是堆优化的 Dijkstra 算法，在原来的 Dijkstra 算法中每次搜索到未标记点的距离需要对所有的点进行搜索，而采用堆结构的话搜索速度大大缩短，搜索的时间复杂度从 $O(n)$ 降为 $O(\log n)$，对于该问题中总共 8000 多个路口来说，搜索时间大概能够缩短为原来的 1/600。

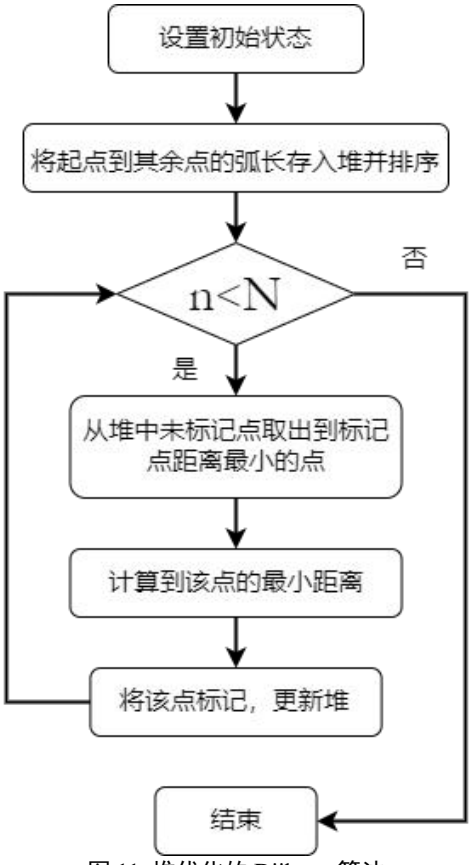

图 11 堆优化的 Dijkstra 算法

4.4.2 局部调度调整算法

在计算出新的车辆路径之后，整个路线结构也随之改变，如果对全局再采用问题 1 的求解方法，那么可能整个过程比较复杂。另外按照题目要求的局部调整，调整的调度方案就可以是一种局部最优的调度方案。

那么针对某条新道路为局部的话，按照问题 1 的思想，先找出可能产生冲突的路口，对于这些路口而言，局部最优的调度策略就是不改变其他车辆通过该路口的等待时间，尽可能缩短新路线在路口的等待时间。所采用的算法与问题 1 一样，而不同的地方是交通灯调度的状态空间和基因长度。下图是新路线可能产生冲突的路口示意图。

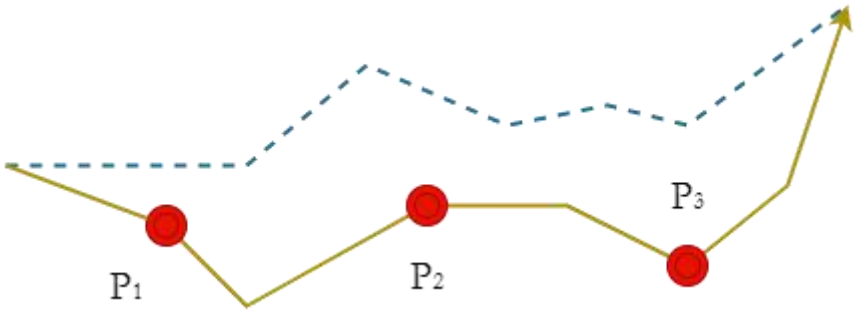

图 12 新路线冲突路口

4.5 实验过程与结果分析

首先是计算 30s 后需要改道的车辆的集合，然后采用堆优化后的 Dijkstra 算法计算 ebf 到原来终点的最短路径，结果如下表 2 所示。

表 2 新路径规划

C016	dbh-fjai-eggf-ebbe-ebi-ghfc-edaj-ebf-dci-cci-f-bhej
C035	bch-djei-ccii-ceag-eeg-bjif-dadh-ejje-egd-ebf-ejae-c-bcj-fj-fjh-bb-ijh-bhi-cddh
C050	gg-djjc-cede-dci-ebf-ejae-c-bcj-fj-fjh-bb-bjfh

新规划的路径只有 3 条，下图 13 是这 3 辆车行进状态，横向是时间，绿色代表正在行车，红色代表正在等待，可以看出新规划的路线等待时间并不长，在合理的区间，而且采用所提出的局部规划策略对其他车辆的路线没有任何影响，所以在全局的影响也很小，所以这种局部规划调整是可行的。

图 13 新路线行进状态图

同样的我们也如第一问一样统计一下可能发生冲突的路口等待时间情况如下图 14 所示，还对比了问题 1 的情况，可以看出整体状况略有不足，这是因为在局部进行了微调，导致整体比问题 1 的情况略差，但也说明了对全局影响不大，而且需要调整的路口不多。

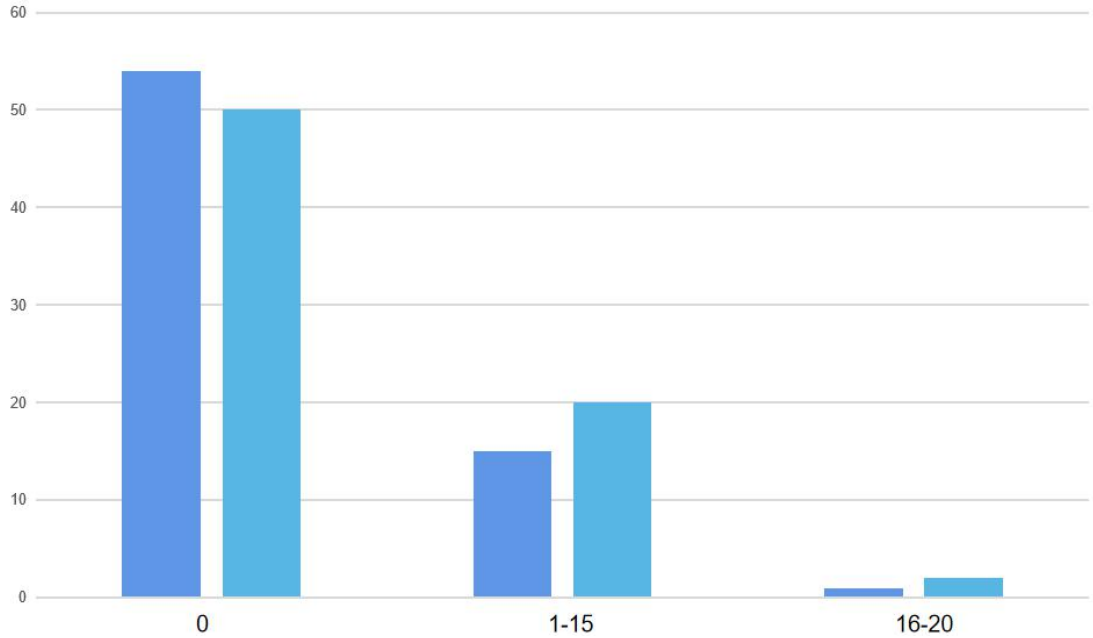

图 14 冲突路口等待时间情况统计

五、总结

本文以交通信号灯调度策略设计为背景，在给定路网结构、车辆行程计划等信息的条件建立数学模型，通过遗传算法和堆优化的 Dijkstra 算法分别解决了无道路故障和有道路故障下的交通灯调度策略问题。根据实验结果，本文所设计的交通灯调度策略能够很好的满足题目要求。

5.1 模型的优点

（1）在模型建立过程中，用数学语言完善地描述了问题一和问题二不同情况下的交通灯调度策略问题，并通过有效的假设简化了问题。

（2）在数据分析时，问题一针对城市路网规模较大的问题，该模型通过分析车辆的路径对路口和道路进行筛选，减少了优化对象，一定程度上减少了计算量。问题二只考虑车辆最短路径上的交通灯优化，实现了局部优化，避免了重新全局优化带来的巨大运算量。

（3）在模型求解过程过程中，对交通灯调度策略问题使用了遗传算法，将每个交通灯策略作为单独的基因，进行多次的交叉、遗传、变异，减少了计算量，算法较为直观，避免了经典优化方法的复杂计算。对车辆的最短路径规划问题使用了堆优化的 Dijkstra 算法，避免了使用经典 Dijkstra 算法在路口节点数量过多、搜索空间较大的情况下带来的巨大时间开销。

5.2 模型的缺点

（1）模型自变量较多，敏感性较强，在修改车辆路径较前端的交通灯策略时，对后续交通灯策略和其他车辆的行使时间有很大的影响，不同的修改策略有较大的开销差异。

（2）本文的算法较为单一，效率不高，需要大量的时间进行多次迭代才能得到有效的结果。

参考文献：

[1] 丘东林. 城市路网交通流动力学建模及动态交通灯控制策略研究[D].广西师范大学,2021.
[2] 钱锐. 基于宽度优先搜索算法的交通灯智能调控研究[D].杭州电子科技大学,2020.
[3] 邵明莉,曹鹗,胡铭,章玥,陈闻杰,陈铭松.面向优先车辆感知的交通灯优化控制方法[J].软件学报,2021,32(08):2425-2438.
[4] 蔡含宇. 基于道路实时容量的交通灯智能控制算法研究[D].西安理工大学,2020.
[5] 陈海洋,金晓磊,牛龙辉,刘喜庆.基于改进克隆选择算法的区域交通灯配时优化[J].计算机工程与应用,2020,56(09):272-278.

附录

附录一 问题 1 遗传算法

```
import math
import random

Population_num=10
DNA_size=70
Mutations_rage=0.01
iterations_num=1000


class crossing:
    loads=[]
    light_dispatch = []
    weight_T=[]
    def __init__(self,name,num_loads,loads):
        self.name=name
        self.num_loads=num_loads
        self.loads=loads.copy()
        self.start_list=[]
    def dispatch(self,p):      #交通灯调度函数
        self.light_dispatch=[x for x in range(self.num_loads)]
        self.T = 0
        for item in self.light_dispatch:
            self.start_list.append(self.T)
            self.T = self.T + random.randint(1,math.ceil(100*(1-p)))
        random.shuffle(self.light_dispatch)


class population:
    def __init__(self,DNA,fitness,score,true_score):
        self.DNA=DNA.copy()
        self.fitness=fitness
        self.score=score
        self.true_score=true_score

def get_pop(probability,old_populations):
    for i in range(Population_num):
        if old_populations[i].fitness<probability:
            continue
        else:
            return old_populations[i].DNA
```

```
def Mutations(DNA,p):
    for i in range(DNA_size):
        if random.random()<Mutations_rage:
            DNA[i].dispatch(p)
    return DNA

def create_new_population(old_populations,p):
    new_populations=[]
    sum_score=0
    max_score=0
    cross_poit = random.randint(0, DNA_size)
    max_population=0
    for i in range(Population_num):
        father_DNA=get_pop(random.random()*100,old_populations)
        mother_DNA=get_pop(random.random()*100,old_populations)
        son_DNA=father_DNA[:cross_poit]+mother_DNA[cross_poit:]
        son_DNA=Mutations(son_DNA,p)#变异
        DNA_score,true_score=get_score(son_DNA)
        son=population(son_DNA,100,DNA_score,true_score)#种群信息由 DN
A，适应度，分数组成
        if max_score<DNA_score:
            max_score=DNA_score
            max_population=i
        sum_score+=true_score
        new_populations.append(son)
    with open("./result.txt","a",encoding="utf-8") as f:
        f.write("%s\n"%max_score)
    print("种群 id:",max_population,"max_score:",max_score)
    return calculate_fitness(sum_score, new_populations)

def calculate_fitness(sum_score,new_populations):
    fitness=0
    score_list=[x.true_score for x in new_populations]
    score_list=np.array(score_list)
    min_score=min(score_list)
    score_list=np.array([1/(x.true_score-min_score+1) for x in new_population
s])
    sum_score=score_list.sum()
    for i in range(Population_num):
        fitness += (new_populations[i].true_score-min_score+1) / sum_score
 *100
```

```
            new_populations[i].fitness=fitness
        return new_populations

import numpy as np

#车辆途经道路
car_ways=[]
all_ways=[]
with open("cars.txt","r",encoding="utf-8") as f:
        lines=f.readlines()
        for line in lines:
            line=line.strip("\n")
            item_arr=line.split(",")
            car_ways.append(item_arr[2:])
print(car_ways[0])

#路口名 2id,道路列表
cross2id={}
all_road=np.zeros(shape=(8000,8000))
with open("streets.txt","r",encoding="utf-8") as f:
        lines=f.readlines()[1:]
        for line in lines:
            line = line.strip("\n")
            item_arr = line.split(",")
            for i,cross1 in enumerate(item_arr[0].split("-")):
                cross2id[cross1]=int(item_arr[i+1])
            all_road[int(item_arr[1])][int(item_arr[2])]=int(item_arr[3])

#车辆途经道路冲突列表
ways_clash=np.zeros(shape=(8000,8000))
for car_way in car_ways:
        for item in car_way:
            start_cross,end_cross=item.split("-")
            start_crossid=cross2id[start_cross]
            end_crossid=cross2id[end_cross]
            ways_clash[start_crossid][end_crossid]=ways_clash[start_crossid][end_cr
ossid]+1

ways_mask=ways_clash.copy()
ways_mask[ways_mask>1]=1
count_car_reach=np.sum(ways_mask,0)
print(np.max(count_car_reach))
```

```
used_road=all_road*ways_mask
index_clash=np.argwhere(ways_clash>1)[:,1]
value_clash=count_car_reach[index_clash]
all_share_way=np.argwhere(ways_clash>1)[:,1]

#车辆途经道路的 id
car_way_ids=[]
for car_way in car_ways:
    temp_list=[]
    for item in car_way:
        start_cross,end_cross=item.split("-")
        start_crossid=cross2id[start_cross]
        end_crossid=cross2id[end_cross]
        temp_list.append("%s-%s"%(start_crossid,end_crossid))
    car_way_ids.append(temp_list)

#初始化规则
roles={}
start_roles=[]
start_populations=[]    #初始种群
start_DNA=[]
for item in list(set(index_clash)):
    col=ways_mask.transpose()[item]
    index_cross=np.argwhere(col>0)
    cross_with_road=col.sum()
    roles[str(item)]={}
    sort_=[]
    for i,cross in enumerate(index_cross):
        #name:[green_start_time,green_time,T,pass_time]
        roles[str(item)][str(cross[0])+"-"+str(item)]=[i,1,int(cross_with_road),all_r
oad[cross[0]][int(item)]]
        sort_.append(str(cross[0]))
    start_DNA.append(crossing(item,int(cross_with_road),sort_))

def find_cross_info(start_DNA,end_id):
    for item in start_DNA:
        if item.name==int(end_id):
            return item
    return -1

def get_score(start_DNA):
    cross_wait_time={}
```

```
    for index in index_clash:
        cross_wait_time[str(index)]={"wait_time":0,"T":1,"num":0}
    clash_road = np.argwhere(ways_clash > 1)
    rest_counter = {}
    score=0
    for i, item in enumerate(car_way_ids):
        now_pass_time = 1
        clash_ways = []
        for it in item[1:]:
            if int(it.split("-")[1]) not in index_clash:
                now_pass_time += all_road[int(it.split("-")[0])][int(it.split("-")
[1])]
            else:
                now_pass_time = now_pass_time + all_road[int(it.split("-")
[0])][int(it.split("-")[1])]
                clash_ways.append([it, now_pass_time])
                end_id = it.split("-")[1]
                cross_info=find_cross_info(start_DNA,end_id)
                T=cross_info.T
                score+=T
                order=cross_info.light_dispatch
                start_list=cross_info.start_list
                roads=cross_info.loads
                index_time={}
                for x,key in enumerate(roads):
                    if len(roads)==1:
                        index_time["%s-%s" % (key, end_id)] = [order[x], st
art_list[x], T]
                        break
                    if x+1==len(roads):
                        index_time["%s-%s" % (key, end_id)] = [order[x], st
art_list[x], T]
                    else:
                        index_time["%s-%s"%(key,end_id)]=[order[x],start_list
[x],start_list[x+1]]
                # if now_pass_time % T != index_time[it]:
                #     wait_time = T - (now_pass_time % T)
                # else:
                if now_pass_time%T>=index_time[it][1] and now_pass_time%
T<index_time[it][2]:
                    wait_time=0
                else:
```

```
                    wait_time = (T-now_pass_time%T+index_time[it][0]*inde
x_time[it][0])%T
                if cross_wait_time[end_id]["wait_time"]:
                    cross_wait_time[end_id]["wait_time"]+=wait_time
                    cross_wait_time[end_id]["T"] =T
                    cross_wait_time[end_id]["num"] = cross_info.num_loads
                else:
                    cross_wait_time[end_id]["wait_time"]=wait_time
                    cross_wait_time[end_id]["T"] = T
                    cross_wait_time[end_id]["num"] = cross_info.num_loads
                now_pass_time = now_pass_time + wait_time
            if it != item[-1]:
                now_pass_time += 1
        rest_counter[str(i)] = {"now_pass_time": now_pass_time, "clash_ways":
 clash_ways}
    # m_list=[]
    # for item in cross_wait_time.values():
    #     T=item["T"]
    #     num=item["num"]
    #     m_list.append((num*T-item["wait_time"])/T)
    # sum_m=np.array(m_list).sum()
    # print(sum_m)
    sum_m = 0
    m_list = []
    for item in rest_counter.values():
        time = item["now_pass_time"]
        if time <= 858:
            m = 250 + (858 - time)
        else:
            m = 0
        sum_m += m
        m_list.append(m)
    return sum_m,score

start_sum_score=0
start_max_score=0
for i in range(Population_num):
    for j in range(DNA_size):
        start_DNA[j].dispatch(0)
    DNA_score,true_score=get_score(start_DNA)
    print(DNA_score)
    if DNA_score>start_max_score:
```

```
            start_max_score=DNA_score
        start_sum_score+=DNA_score
        start_populations.append(population(start_DNA,100,DNA_score,true_score))
print("start_max_score:",start_max_score)

old_populations=calculate_fitness(start_sum_score,start_populations)

for i in range(iterations_num):
        new_population=create_new_population(old_populations,i/iterations_num)
        old_populations=new_population.copy()
#  result_roles=[]
#  result_score=0
#  for item in old_populations:
#          if result_score<item.score:
#                  result_score=item.score
#                  result_roles=item.DNA
#  np.save("./temp_result",result_roles,allow_pickle=True)
#  s=np.load("./temp_result.npy",allow_pickle=True)
```

附录二 问题 2 遗传算法

```
import math
import random

Population_num=10
DNA_size=72
Mutations_rage=0.01
iterations_num=1000


class crossing:
        loads=[]
        light_dispatch = []
        weight_T=[]
        def __init__(self,name,num_loads,loads):
                self.name=name
                self.num_loads=num_loads
                self.loads=loads.copy()
                self.start_list=[]
        def dispatch(self,p):       #交通灯调度函数
                self.light_dispatch=[x for x in range(self.num_loads)]
                self.T = 0
                for item in self.light_dispatch:
```

```
                self.start_list.append(self.T)
                self.T = self.T + random.randint(1,math.ceil(100*(1-p)))
            random.shuffle(self.light_dispatch)


class population:
    def __init__(self,DNA,fitness,score,true_score):
        self.DNA=DNA.copy()
        self.fitness=fitness
        self.score=score
        self.true_score=true_score

def get_pop(probability,old_populations):
    for i in range(Population_num):
        if old_populations[i].fitness<probability:
            continue
        else:
            return old_populations[i].DNA

def Mutations(DNA,p):
    for i in range(DNA_size):
        if random.random()<Mutations_rage:
            DNA[i].dispatch(p)
    return DNA

def create_new_population(old_populations,p):
    new_populations=[]
    sum_score=0
    max_score=0
    cross_poit = random.randint(0, DNA_size)
    max_population=0
    for i in range(Population_num):
        father_DNA=get_pop(random.random()*100,old_populations)
        mother_DNA=get_pop(random.random()*100,old_populations)
        son_DNA=father_DNA[:cross_poit]+mother_DNA[cross_poit:]
        son_DNA=Mutations(son_DNA,p)#变异
        DNA_score,true_score=get_score(son_DNA)
        son=population(son_DNA,100,DNA_score,true_score)#种群信息由 DN
A，适应度，分数组成
        if max_score<DNA_score:
            max_score=DNA_score
            max_population=i
```

```
            sum_score+=true_score
            new_populations.append(son)
        # with open("./result_task2.txt","a",encoding="utf-8") as f:
        #         f.write("%s\n"%max_score)
        print("种群 id:",max_population,"max_score:",max_score)
        return calculate_fitness(sum_score, new_populations)

def calculate_fitness(sum_score,new_populations):
        fitness=0
        score_list=[x.true_score for x in new_populations]
        score_list=np.array(score_list)
        min_score=min(score_list)
        score_list=np.array([1/(x.true_score-min_score+1) for x in new_population
s])
        sum_score=score_list.sum()
        for i in range(Population_num):
            fitness += (new_populations[i].true_score-min_score+1) / sum_score
 *100
            new_populations[i].fitness=fitness
        return new_populations

import numpy as np

#车辆途经道路
car_ways=[]
all_ways=[]
with open("cars.txt","r",encoding="utf-8") as f:
        lines=f.readlines()
        for line in lines:
            line=line.strip("\n")
            item_arr=line.split(",")
            car_ways.append(item_arr[2:])
newcars_ways=[]
with open("newcars.txt","r",encoding="utf-8") as f:
        lines=f.readlines()
        for line in lines:
            line=line.strip("\n")
            item_arr=line.split(",")
            newcars_ways.append(item_arr[2:])
car_ways[15]=newcars_ways[0]
car_ways[34]=newcars_ways[1]
car_ways[49]=newcars_ways[2]
```

```
#路口名 2id,道路列表
cross2id={}
all_road=np.zeros(shape=(8000,8000))
with open("streets.txt","r",encoding="utf-8") as f:
    lines=f.readlines()[1:]
    for line in lines:
        line = line.strip("\n")
        item_arr = line.split(",")
        if int(item_arr[1])==415 and int(item_arr[2])==5545:
            continue
        else:
            for i,cross1 in enumerate(item_arr[0].split("-")):
                cross2id[cross1]=int(item_arr[i+1])
            all_road[int(item_arr[1])][int(item_arr[2])]=int(item_arr[3])

#车辆途经道路冲突列表
ways_clash=np.zeros(shape=(8000,8000))
for car_way in car_ways:
    for item in car_way:
        start_cross,end_cross=item.split("-")
        start_crossid=cross2id[start_cross]
        end_crossid=cross2id[end_cross]
        ways_clash[start_crossid][end_crossid]=ways_clash[start_crossid][end_cr
ossid]+1

ways_mask=ways_clash.copy()
ways_mask[ways_mask>1]=1
count_car_reach=np.sum(ways_mask,0)
print(np.max(count_car_reach))
used_road=all_road*ways_mask
index_clash=np.argwhere(ways_clash>1)[:,1]
value_clash=count_car_reach[index_clash]
all_share_way=np.argwhere(ways_clash>1)[:,1]

#车辆途经道路的 id
car_way_ids=[]
for car_way in car_ways:
    temp_list=[]
    for item in car_way:
        start_cross,end_cross=item.split("-")
        start_crossid=cross2id[start_cross]
```

```
            end_crossid=cross2id[end_cross]
            temp_list.append("%s-%s"%(start_crossid,end_crossid))
        car_way_ids.append(temp_list)

#初始化规则
roles={}
start_roles=[]
start_populations=[]    #初始种群
start_DNA=[]
for item in list(set(index_clash)):
        col=ways_mask.transpose()[item]
        index_cross=np.argwhere(col>0)
        cross_with_road=col.sum()
        roles[str(item)]={}
        sort_=[]
        for i,cross in enumerate(index_cross):
            #name:[green_start_time,green_time,T,pass_time]
            roles[str(item)][str(cross[0])+"-"+str(item)]=[i,1,int(cross_with_road),all_r
oad[cross[0]][int(item)]]
            sort_.append(str(cross[0]))
        start_DNA.append(crossing(item,int(cross_with_road),sort_))

def find_cross_info(start_DNA,end_id):
        for item in start_DNA:
            if item.name==int(end_id):
                return item
        return -1

def get_score(start_DNA):
        cross_wait_time={}
        for index in index_clash:
            cross_wait_time[str(index)]={"wait_time":0,"T":1,"num":0}
        clash_road = np.argwhere(ways_clash > 1)
        rest_counter = {}
        score=0
        for i, item in enumerate(car_way_ids):
            now_pass_time = 1
            clash_ways = []
            for it in item[1:]:
                if int(it.split("-")[1]) not in index_clash:
                    now_pass_time += all_road[int(it.split("-")[0])][int(it.split("-")
[1])]
```

```
            else:
                now_pass_time = now_pass_time + all_road[int(it.split("-")
[0])][int(it.split("-")[1])]
                clash_ways.append([it, now_pass_time])
                end_id = it.split("-")[1]
                cross_info=find_cross_info(start_DNA,end_id)
                T=cross_info.T
                score+=T
                order=cross_info.light_dispatch
                start_list=cross_info.start_list
                roads=cross_info.loads
                index_time={}
                for x,key in enumerate(roads):
                    if len(roads)==1:
                        index_time["%s-%s" % (key, end_id)] = [order[x], st
art_list[x], T]
                        break
                    if x+1==len(roads):
                        index_time["%s-%s" % (key, end_id)] = [order[x], st
art_list[x], T]
                    else:
                        index_time["%s-%s"%(key,end_id)]=[order[x],start_list
[x],start_list[x+1]]
                # if now_pass_time % T != index_time[it]:
                #     wait_time = T - (now_pass_time % T)
                # else:
                if now_pass_time%T>=index_time[it][1] and now_pass_time%
T<index_time[it][2]:
                    wait_time=0
                else:
                    wait_time = (T-now_pass_time%T+index_time[it][0]*inde
x_time[it][0])%T
                if cross_wait_time[end_id]["wait_time"]:
                    cross_wait_time[end_id]["wait_time"]+=wait_time
                    cross_wait_time[end_id]["T"] =T
                    cross_wait_time[end_id]["num"] = cross_info.num_loads
                else:
                    cross_wait_time[end_id]["wait_time"]=wait_time
                    cross_wait_time[end_id]["T"] = T
                    cross_wait_time[end_id]["num"] = cross_info.num_loads
                now_pass_time = now_pass_time + wait_time
            if it != item[-1]:
```

```
                    now_pass_time += 1
            rest_counter[str(i)] = {"now_pass_time": now_pass_time, "clash_ways":
 clash_ways}
    # m_list=[]
    # for item in cross_wait_time.values():
    #     T=item["T"]
    #     num=item["num"]
    #     m_list.append((num*T-item["wait_time"])/T)
    # sum_m=np.array(m_list).sum()
    # print(sum_m)
    sum_m = 0
    m_list = []
    for item in rest_counter.values():
        time = item["now_pass_time"]
        if time <= 858:
            m = 250 + (858 - time)
        else:
            m = 0
        sum_m += m
        m_list.append(m)
    return sum_m,score

start_sum_score=0
start_max_score=0
for i in range(Population_num):
    for j in range(DNA_size):
        start_DNA[j].dispatch(0)
    DNA_score,true_score=get_score(start_DNA)
    print(DNA_score)
    if DNA_score>start_max_score:
        start_max_score=DNA_score
    start_sum_score+=DNA_score
    start_populations.append(population(start_DNA,100,DNA_score,true_score))
print("start_max_score:",start_max_score)

old_populations=calculate_fitness(start_sum_score,start_populations)

for i in range(iterations_num):
    new_population=create_new_population(old_populations,i/iterations_num)
    old_populations=new_population.copy()
result_roles=[]
result_score=0
```

```
for item in old_populations:
    if result_score<item.score:
        result_score=item.score
        result_roles=item.DNA
np.save("./temp_result_2",result_roles)
```

附录三 问题 2 最短路径算法

```
import numpy as np

#test
a=[[0,1,2],[1,2,4]]
a=np.array(a)
b=np.sum(a,0)

#车辆途经道路
car_ways=[]
all_ways=[]
with open("cars.txt","r",encoding="utf-8") as f:
    lines=f.readlines()
    for line in lines:
        line=line.strip("\n")
        item_arr=line.split(",")
        car_ways.append(item_arr[2:])
print(car_ways[0])

#路口名 2id,道路列表
cross2id={}
all_road=np.zeros(shape=(8000,8000))
with open("streets.txt","r",encoding="utf-8") as f:
    lines=f.readlines()[1:]
    for line in lines:
        line = line.strip("\n")
        item_arr = line.split(",")
        for i,cross1 in enumerate(item_arr[0].split("-")):
            cross2id[cross1]=int(item_arr[i+1])
        all_road[int(item_arr[1])][int(item_arr[2])]=int(item_arr[3])
print(np.argwhere(all_road[415]>0).__len__())
s=cross2id["ffef"]
all_road=all_road.tolist()
nums=[]
for i,row in enumerate(all_road):
    for j,col in enumerate(row):
```

```
            if col!=0:
                if i==415 and j==5545:
                    continue
                else:
                    nums.append([i,j,col])
id2cross={v:k for k,v in cross2id.items()}

import heapq
class graph:
    def __init__(self,num,ma):
        self.edge={}
        self.dic=[ma]*(num)
    def add(self,u,v,w):
        if u in self.edge:
            self.edge[u].append((v,w))
        else:
            self.edge[u]=[(v,w)]
    def dijkstra(self,start,n):
        dis=self.dic
        dis[start]=0
        heap=[]
        path = [[]] * len(dis)  # source 到其他节点的最短路径
        path[start] = [start]
        visit=[0 for i in range(len(dis))]
        for i in self.edge[start]:
            dis[i[0]]=i[1]
            heapq.heappush(heap,(i[1],i[0]))# i[1]为该边权值，i[0]为该点，从 s
tart 点到其余点的边入堆
            path[i[0]]=[i[0]]  # 再其后添加已找到节点即为 sorcer 到该节点的
最短路径
        while heap!=[]:
            vic=heapq.heappop(heap)# 弹出最小边
            if visit[vic[1]]==1:# 如果该点已经弹出过，则不再松弛
                continue
            visit[vic[1]] = 1# 记录弹出点
            if vic[1] not in self.edge:# 防止有向图中出度为 0 的点
                continue
            for i in self.edge[vic[1]]:
                if dis[i[0]]>dis[vic[1]]+i[1]:# 判断是否松弛
                    dis[i[0]]=dis[vic[1]]+i[1]# 松弛边
                    heapq.heappush(heap,(i[1],i[0]))# 松弛过边则把新边权
值入堆
```

```
                    path[i[0]] = path[vic[1]][:]  # 复制 source 到已找到节点的上一节点的路径
                    path[i[0]].append(i[0])  # 再其后添加已找到节点即为 sorcer 到该节点的最短路径
            self.dic=dis
            return path
        def printf(self):
            print(self.dic)
if __name__ == "__main__":
    # nums = [[0,2,10],[0,4,30],[0,5,100],[1,2,5],[2,3,50],[3,5,10],[4,5,60],[4,3,20]]
    dis_list=[cross2id["bhej"],cross2id["cddh"],cross2id["bjfh"]]
    n, m = 8000, len(dis_list)
    # n,m=6,8
    g=graph(n,1000000000000)
    for i in nums:
        if i[0]==415:
            print(i)
        g.add(i[0],i[1],i[2])
    path=g.dijkstra(cross2id["ebf"],n)
    g.printf()
    with open("./newcars.txt","w",encoding="utf-8") as f:
        for p in path:
            print(p)
            if p!=[] and p[-1] in dis_list:
                f.write("%s\n"%p)
```