土壤采样点最优路径和工作时间均衡化问题

摘要

研究最优路径规划问题对提高土壤采样工作的效率具有重要意义。本文建立最短路径模型和均衡化优化模型，运用图论知识、Floyd-Warshall 算法、回溯策略、深度优先搜索算法、最近邻算法、贪心算法、禁忌搜索算法等方法研究多点位之间的最优路径问题和任务均衡化问题。

针对问题一，首先从图论的角度出发，将其转化为寻找连接 8 个采样点的最短哈密顿通路问题。采用 Haversine **公式**来计算所有采样点之间的最短路径，以此构建邻接矩阵，通过**回溯算法**不断递归，找到最短哈密顿路径；随后采用 python 的 geodesic 方法直接用函数计算出所有采样点之间的最短路径，使用 Floyd-Warshall **算法**通过不断迭代找到所有点之间的最短路径，通过**回溯策略嵌套深度优先搜索** DFS 构建一个访问所有点的路径,找到一个访问所有点的最短单向路径及其时间。对比两种方法，发现结果一致，得到**最优路径为** 44 -> 61 -> 158 -> 83 -> 147 -> 100 -> 31 -> 115；**最短时间约为** 401.53min。

针对问题二，首先进行数据预处理，将数据从 WGS 84 坐标转换为 UTM 坐标，随后采用 K-means **聚类算法**将转换后的经纬度坐标初步聚类为 28 个簇，并使用**贪心算法**对聚类后的数据集进行调整，使得前 27 个簇都已经包含全部 8 个采样点，令第 28 个簇包含剩余的 6 个采样点，并输出每个簇。运用第一问的模型计算出每个簇的最短时间路径，求解结果存放在**题二时间.xlsx** 中。对输出结果进行以时间升序的冒泡排序，得到最优路径的工作时间的**最大值为** 474.5002759056946 **min，最小值为** 327.1029846448918 **min。**

针对问题三，先对问题二的方案进行检验。因为问题二中工作时间最大值小于 8.5 小时，所有**没有超过最大限时现象**。采用标准差和洛伦兹曲线进行均衡性检验，得到总体标准差是： 26.82523171286905，**Gini 系数**为 0.03294262157395665，故问题二的方案**均衡性良好**。

随后建模计算取消 8 个点限制后的均衡化方案，首先仍然对 222 个数据点建立任意两点之间的行程的时间矩阵，使用**基于最近邻矩阵策略的贪心算法**构建路径的初始解，接着通过**领域搜索策略**优化初始解，然后使用**禁忌搜索算法**在优化解中搜索最优解，得到最短工作时间及其具体路径并输出，对每个路径内的数据点尝试移动和交换，以尽可能平衡每天的工作时间，使最终工作时间的**极差**不大于 50min，最终输出 Pareto **前沿**上的解决方案，**存放在题三时间.xlsx 中**。可知完成时间缩短到 **25 天，平均工作时间为** 473.39min。

针对问题四，使用函数构建请求 URL，通过高德地图 API 和腾讯地图 API 分别获取 8 个采样点之间的距离和时长，随后套用问题一的最短路径规划模型，比较两个地图的最短时间并选择较短的工作时间和规划路径输出。由于实时情况动态变化，编写自动运行脚本，获取每隔 30min 的 API 调用数据并重新运行，共得到 30 个样本点，存放在 **Q4.xlsx** 中。对样本点进行分析，发现**最短工作时间在** 410.0min **到** 411.5min **之间，一般使用高德地图导航，最优路径规划主要为** 83 -> 158 -> 100 -> 147 -> 31 -> 115 -> （61 -> 44/44->61）。

关键词：哈密顿路径　Floyd-Warshall 算法　回溯策略　禁忌搜索算法

一、问题重述

1.1 问题背景

土壤普查工作旨在通过对国土各类土地资源状况的排查，指导国家的农业生产和土地资源规划，具有至关重要的意义。我国地形复杂，土地种类多样，对土壤资源采样工作的顺利进行造成了严重阻碍，因此，有必要建立路径规划和任务均衡化的最优模型，提高工作效率。

1.2 问题要求

问题一：在本问题中，需要对给定的 8 个采样点规划出最优路径，使得加上通行时间后得到的工作时间最短。通过建立数学模型计算最短工作时间，并输出最优路径规划。

问题二：在本问题中，需要采用就近原则，以每天工作 8 个点位为基准，通过建立数学模型，对所有的 222 个采样点划分出每一天的点位最优路径及最短工作时间，并给出所有工作时间的最值。

问题三：在本问题中，需要对问题二中的方案进行时限和均衡化评价，并通过建立数学模型，给出取消 8 个点位限制后，在满足均衡化情况下的每一天的点位最优路径及最短工作时间。

问题四：在本问题中，需要调用地图开放平台的 API，采用专车通行方案，给出 8 个采样点之间的最优路径规划和最短工作时间。

二、问题分析

2.1 问题一的分析

问题一属于在单一约束条件下求解最优解的问题,该题属于旅行商 TSP 问题的变式，实际上就是要求我们求解最短哈密顿通路。采用 Haversine 算法或 Python 第三方库 geopy 都可以计算出两个经纬度坐标点之间的直线距离，通过回溯算法不断递归即可得到最短哈密顿路径。又由于题目已经把速度为 20km/h 给出，故即可得到题目要求的最短工作时间和对应的路径规划。

2.2 问题二的分析

问题二中，要求我们按每天 8 个样本点对题目给出的 222 个样本点划分，分成 28 天完成任务。由于天数已经给定，实际上就是需要我们按就近原则进行聚类分簇，将原有数据分成 28 个簇即可。又由于要求每天都要采样 8 个点，实际上就是要求进行均衡化，我们先把最后一天的 6 个点拿出来，对其他都是 8 个点的簇使用贪心算法进行均衡化操作，即可输出每一天的点位最优路径及最短工作时间，再通过排序即可得到所有工作时间的最值。

2.3 问题三的分析

问题三有两个部分。

第一部分属于模型评价问题，要求判断问题二中的方案是否超过最大工作时间限制，是否均衡。检验是否超出只要和最大值比较，检测均衡化只需要引入相应的参数，采用标准差和洛伦兹曲线中的基尼系数作为衡量指标，即可明确问题二的方案是否均衡。

第二部分属于在多个约束条件下寻找最优解问题，直接求解比较困难。采用启发式算法寻优，我们可以采用基于最近邻矩阵策略的贪心算法构建初始解，领域搜索策略优化初始解，禁忌搜索算法在优化解中搜索最优解。再对对每个路径内的数据点尝试移动

和交换，即可得到最短工作时间及其具体路径并输出。

2.4 问题四的分析

在问题一的基础上多了对地图开放平台的 API 调用。只要将请求时间和距离数据得到的结果放入问题一模型中即可得到要求的8个采样点之间的最优路径规划和最短工作时间。另外需要注意实时路况是动态的，因此实现多次调用得到一定量样本点以后分析，可以获得更完善的结论。

三、模型假设

1. 假设对于所有采样点，任意两个采样点之间都是直线可达的，任意两点间的行车距离都以直线距离计。
2. 假设对于所有采样点，它们占有的陆地之间都是平整的，没有海拔高度差异
3. 假设地球是一个带有曲率的，理想化的椭球体
4. 对于问题四，假设使用的地图开放平台的算法和数据集可信，地图精确有效
5. 对于问题四，假设工作组都在北京时间的工作日，白天进行出行，土壤普查勘探工作
6. 对于问题四，假设工作组出行的日期内没有重大事件影响，如地质灾害，气象灾害，封路等造成出行时间或者路径收到较大影响的事件

四、符号说明

符号	说明	单位
d	两点之间的球半径	Km
λ_1、λ_2	两个点的纬度	度（°）
φ_1、φ_2	两个点的经度	度（°）
dLatitude	两个点的纬度差	度（°）
latitude1、latitude2	两个点的纬度	度（°）
dLongitude	两个点的经度差	度（°）
Longitude1、longitude2	两个点的经度	度（°）
Radius	地球半径	Km

五、模型的建立与求解

5.1 问题一的模型建立与求解

对于本题的求解我们采用了两种算法，分别在 MATLAB 和 Python 中建模得到结果

算法一：使用 Haversine 公式，使用哈密顿路径的概念+回溯算法

算法二：调用 python 中 geopy 库的 geodesic 方法，

使用 Floyd-Warshall 算法+回溯策略+深度优先搜索 DFS

5.1.1 算法一概念说明

1）Haversine 公式

Haversine 公式即半正矢公式，它利用特定地点的经度和纬度值，计算两点间的大圆距离（也称为正交距离），正交距离定义为地球表面或球体上两点之间的最短距离。[1]为了保证模型的建立符合实际情况，首先对 Haversine 公式进行简要说明。

$$\mathrm{Haversine}(d/r) = \mathrm{haversine}(\varphi 2 - 1) + \cos(\varphi 1)\cos(\varphi 2)\mathrm{haversine}(\lambda 2 - \lambda 1) \quad (1)$$

其中 d 表示两点之间的球半径，$\lambda 1, \lambda 2$ 表示两个地点的纬度，$\varphi 1, \varphi 2$ 表示两个地点的经度。

于是得到两点之间的纬度差

$$\mathrm{dLatitude} = \mathrm{latitude2} - \mathrm{latitude1} \quad (2)$$

两点之间的经度差

$$\mathrm{dLongitude} = \mathrm{longitude2} - \mathrm{longitude1} \quad (3)$$

在此基础上得到 Haversine 公式

$$a = \sin(\mathrm{dlatitude}/2)\text{^}2 + \cos(\mathrm{latitude1}) * \cos(\mathrm{latitude2}) * (\sin(\mathrm{dlongitude}/2))\text{^}2$$

$$c = 2 * \mathrm{Math.atan2}\big(\mathrm{Math.sqrt}(a), \mathrm{Math.sqrt}(1-a)\big) \quad (4)$$

$$\mathrm{distance} = \mathrm{Radius} * c$$

2）最短哈密顿路径

结合图论中哈密顿路径的概念，本问题实际上可以转化为寻找最短哈密顿通路的问题。同样对最短哈密顿路径进行简要说明。

哈密顿路径由数学家 W.R.Hamilton 和 T.P.Kirkman 提出，是指访问图中每个顶点一次且仅一次的路径。[2]以下图为例，从所有顶点经过一次的通路可能是路径{A-E-D-C-B}，也可能是路径 {A-B-C-D-E}。将每两点之间的距离相加，可以得到路径 1 的总长度是 22，而路径 2 的总长度是 19。因此路径 2 比路径 1 更好。由此可知，对于图上给定的一些顶点，我们总能找到一条连接所有点的总长度最小的通路，即最短哈密顿通路。

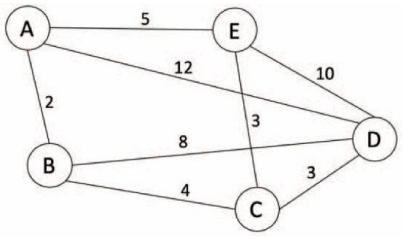


图 1 哈密顿通路示意图

5.1.2 算法一模型建立与求解

1）在主函数中，首先将每一个采样点作为索引顺序为 1 的采样点依次尝试，得到当序号为 115 的采样点的索引为 1 时距离最短。

2）通过 Haversine 公式（4）计算出任意两个采样点之间的球面距离，并将其填充成为如下图所示的邻接矩阵。

表 1

序号	115	31	44	61	83	100	147	158
115	0	0.280455796	3.014192742	2.663012622	2.10812742	1.073928525	1.055769583	2.88555539
31	0.280455796	0	2.776293116	2.422758065	1.827689003	0.825326215	0.78483461	2.605544497
44	3.014192742	2.776293116	0	0.355165564	1.628205927	2.543900584	2.361221181	1.406312926
61	2.663012622	2.422758065	0.355165564	0	1.34824309	2.197749054	2.012169683	1.285238417
83	2.10812742	1.827689003	1.628205927	1.34824309	0	1.199328269	1.09957905	0.793233883
100	1.073928525	0.825326215	2.543900584	2.197749054	1.199328269	0	0.205282416	1.988688165
147	1.055769583	0.78483461	2.361221181	2.012169683	1.09957905	0.205282416	0	1.892619382
158	2.88555539	2.605544497	1.406312926	1.285238417	0.793233883	1.988688165	1.892619382	0

***为方便在论文中表示，没有写成矩阵的形式**

3）使用**回溯算法**在给出的邻接矩阵中找到最短哈密顿路径。
该算法的具体流程如下：

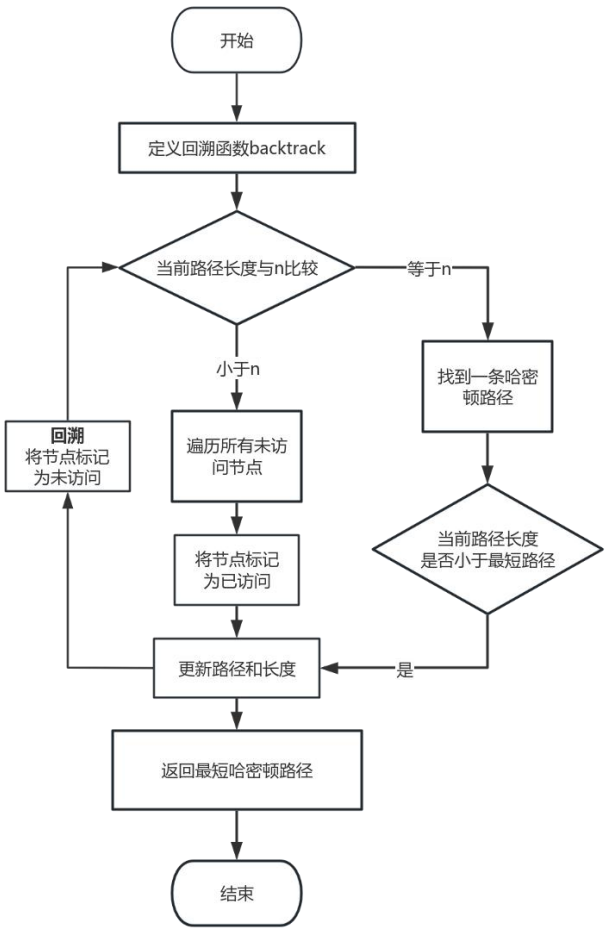


图 2 ：回溯算法流程

函数通过回溯算法实现

***Step1* 定义 backtrack 递归函数:**定义回溯函数 backtrack，用于递归地寻找路径。

***Step2* 判断当前路径长度与当前最短路径长度 n 的关系：**若当前路径更短，则更新最短路径和长度 n;若当前路径长度小于 n,则遍历所有未访问的采样点。

***Step3* 递归调用 backtrack 函数：**初始时仅访问了第一个采样点。故调用 backtrack 函数，从第一个采样点开始搜索。

对于每个未访问采样点，将其标记为已访问，通过不断递归调用 backtrack 函数，更新路径和长度，最后返回最短哈密顿路径。

***Step4* 回溯：**在回溯时首先将已访问的采样点标记为未访问。然后进行下一轮递归。

4）最后将计算得到的最短路径相加，得到最终结果为 4.8443km，代入速度 20km/h，得到路上耗费的最短时间为 14.5329min。于是得到工作组完成当天工作的最短时间是（52+50+55+52+38+39+48+53+14.5329）min，即 401.5329min。
得到的最短哈密顿路径如下图所示。

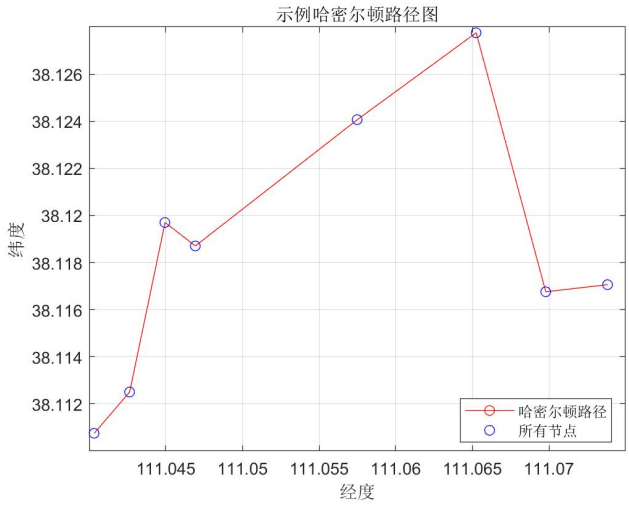


图 3 最短哈密顿通路

5.1.3 算法二的模型建立与求解

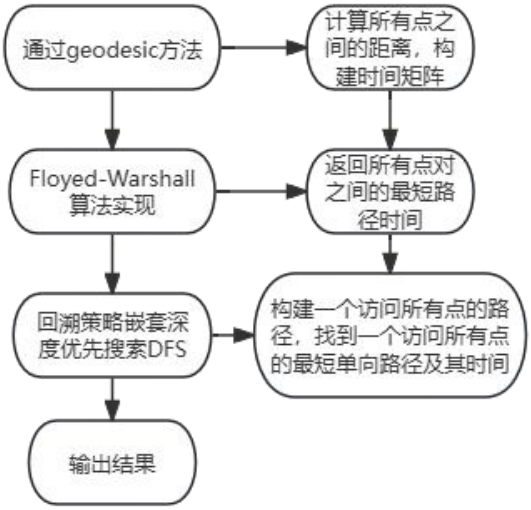


图 4　算法二流程图

1）我们注意到，python 直接提供了 geopy 库来计算各类测地线距离（即考虑地球曲率的最短距离）。而其中的 geodesic 方法可以用来计算地球上两点之间的距离。

2）读取给定的 8 个采样点的经纬度数据，通过 geodesic 方法计算所有采样点之间的距离，并按照 20km/h 的速度将其全部转化为所有采样点之间的时间，填充到一个时间矩阵中。所有的对角线位置为 0，因为每个采样点到达自己的最短时间为 0。而其他位置的数据则可被视为连接两个采样点的边的权重，即最短到达时间。

时间矩阵：

[[0.0, 8.347711508345261, 7.284561086593386, 5.484948133816309, 2.472231865715798,
0.8416969717902941, 2.352595972268699, 7.821626761178003],
[8.347711508345261, 0.0, 1.068014503347556, 4.89169463035579, 7.649591689453603,
9.062239957096297, 7.100484429503311, 4.216513869482587],
[7.284561086593386, 1.068014503347556, 0.0, 4.048335274880765, 6.608461614249632,
8.006116656282249, 6.050704455951583, 3.8504519996202413],
[5.484948133816309, 4.89169463035579, 4.048335274880765, 0.0, 3.604173082888594,
6.326592954016837, 3.302625266804037, 2.3827661155033835],
[2.472231865715798, 7.649591689453603, 6.608461614249632, 3.604173082888594, 0.0,
3.2180051639743157, 0.6165782337409859, 5.975355587548573],
[0.8416969717902941, 9.062239957096297, 8.006116656282249, 6.326592954016837,
3.2180051639743157, 0.0, 3.1656665401489605, 8.66198955887592],
[2.352595972268699, 7.100484429503311, 6.050704455951583, 3.302625266804037,
0.6165782337409859, 3.1656665401489605, 0.0, 5.684812154016353],
[7.821626761178003, 4.216513869482587, 3.8504519996202413, 2.3827661155033835,
5.975355587548573, 8.66198955887592, 5.684812154016353, 0.0]]

3）随后我们使用 Floyd-Warshall 算法，通过对所有时间的不断迭代找到所有采样点之间的最短时间。

Floyd-Warshall 算法：Floyd-Warshall 算法由 Robert Floyd 和 Stephen Warshall 提出，是一种用于求解图中所有采样点之间最短路径的动态规划算法。其核心思想是利用

一个中间采样点来尝试更新任意两个采样点之间的最短路径。

以下是 Floyd 算法确定最短路径的具体步骤：

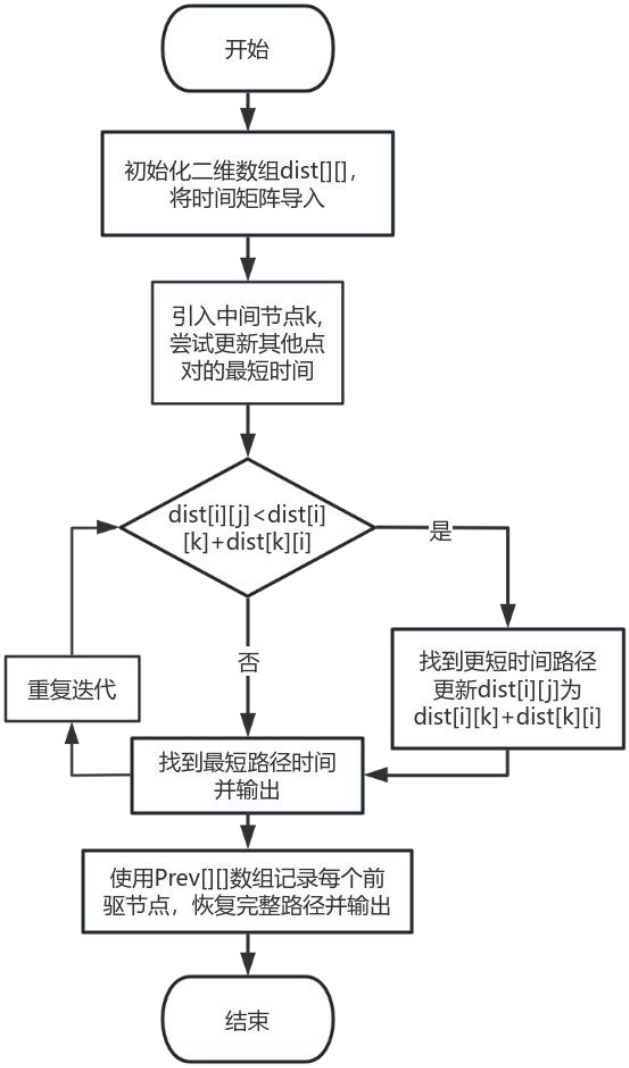


图 5　Floyd-Warshall 算法流程

***Step1* 初始化：**创建一个二维数组 dist[][]，并将得到的时间矩阵中的每一个节点对的最短时间的数据和位置填充到这个二维数组中。dist[i][j]表示节点 i 和节点 j 之间直接相连的边的权重。

***Step2* 引入中间节点：**对于图中的每一个节点 k（从 1 到 V，其中 V 是节点的总数），将其作为中间节点来尝试更新其他所有节点对之间的最短路径时间。

***Step3* 更新最短路径时间：**比较 dist[i][j]和 dist[i][k] + dist[k][j]的大小。对于每一对节点(i, j)，检查是否存在一个通过中间节点 k 的路径，使得从 i 到 k 再到 j 的路径长度比直接从 i 到 j 的路径更短。如果 dist[i][k] + dist[k][j] < dist[i][j]，则更新 dist[i][j]为 dist[i][k] + dist[k][j]。这表示找到了一个更短的路径从 i 到 j，且这条路径通过中间节点 k。

***Step4* 重复迭代：**对每一个中间节点 k 重复步骤 3，以确保考虑了所有可能的中间节点组合。在每次迭代中，都会尝试更新所有节点对之间的最短路径时间。

***Step5* 结果：**当所有的中间节点都被考虑过之后，dist[][]数组中存储的就是图中所有节点对之间的最短路径时间。

***Step6* 画出最短路径：**使用二维数组 prev[][]来记录到达每个节点的前驱节点。在更新 dist[][]的同时，也更新 prev[][]，使之恢复出完整的路径。

4）最后使用**回溯策略**来确定连接所有采样点且用时最短的通路。

与方法一中提到的回溯策略类似，不再重复说明。但在本算法中，我们在回溯策略中**嵌套了一个深度优先搜索 DFS 函数**，用于递归地构建路径。在搜索过程中，通过标记采样点是否被访问来避免重复访问，并记录最短时间及对应的路径。最后，函数遍历所有采样点作为起点，尝试构建路径，并返回最短时间和路径。

5）经过以上步骤，我们得到的最优路径为:44 -> 61 -> 158 -> 83 -> 147 -> 100 -> 31 -> 115

最短时间为: 401.53436495652545min

5.1.4 算法比较

对比两种算法得到的结果，可知最优路径完全一致，最短工作时间的误差非常非常小，进一步证明了我们模型的可靠性以及求解结果的精确性。

表 2　两种算法的比较

	算法 1	算法 2
最优路径	44 -> 61 -> 158 -> 83 -> 147 -> 100 -> 31 -> 115（或从 115 到 44）	
最短时间	401.5329min	401.53436495652545min
绝对误差	0.001464956525min	
相对误差	0.00036484096955%	

5.2 问题二的模型建立与求解

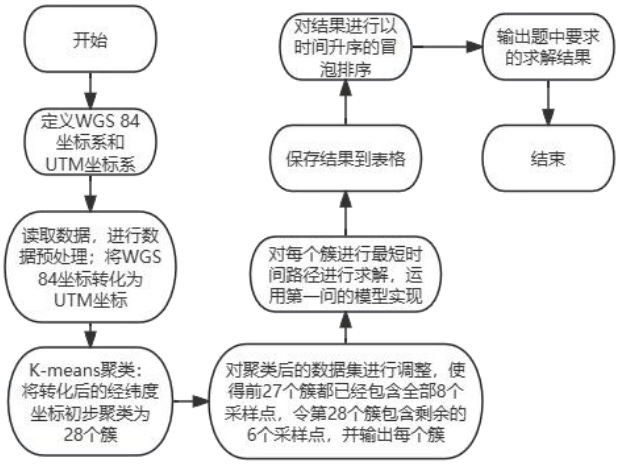


图 6　问题二流程图

5.2.1 坐标系转换

首先分别定义 WGS 84 坐标系和 UTM 坐标系。随后读入 8 个采样点的经纬度数据，并将其从 WGS 84 坐标系转换为 UTM 坐标系。也就是把用经纬度刻画的地心坐标系通过特定的投影算法转换成平面坐标系，把球面坐标转换为平面坐标，方便给定的 8 个采样点在小范围尺度下的距离计算。

5.2.2K-means 聚类：

K-means 聚类算法是将采样点划分为 k 个簇，使得每个簇内的采样点相似度较高，而不同簇的采样点相似度较低的一种常见的聚类算法。因为其在确定簇数的情况下有很好的聚类效果，而本问题中刚好需要在给定了天数（即簇数）的情况下进行聚类，所以非常适合。

使用 K-means 聚类算法对给定的 222 个采样点进行分组，按题目每天 8 个采样点的要求，可知要分成 222/8 28 组。于是我们将 28 作为给定的 K-means 聚类簇数，通过迭代地分配簇和更新聚类中心，使得簇内相似度高而簇间相似度低。由此初步得到一个分簇聚类后的结果，但需要注意的是，此时每个簇中的采样点个数并不是均衡的。所以在此基础上，我们还要对 28 个簇里，每一天的工作时间进行均衡化调整。

5.2.3 贪心算法进行均衡化调整：

于是我们使用贪心算法对聚类之后的簇进行均衡化调整。首先规定一个簇中采样点最多为 8 个，最少为 6 个，由此确定数量大小不符合需要调整的簇。不断将采样点过多

的簇中的采样点移到采样点过少的簇中，直到他们的采样点容量都符合容量要求。由此我们达到了一个全局的最优解，并将调整完的分簇情况输出。
下图展示了调整完之后的分簇情况，不同颜色的点所在的簇不同。

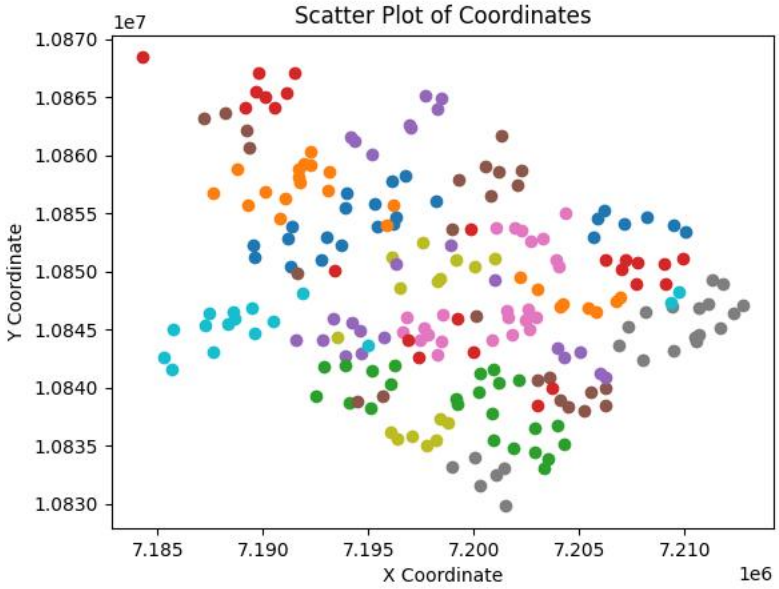


图 7　分簇可视化结果

附录三中展示了得到的 28 个簇的可视化结果。

5.2.4 套用问题一的最短路径规划模型：

在此基础上，使用问题一中得到的模型，对每一簇给定的采样点计算最短哈密顿路径，得到每一天的采样最优路径，及相应的工作时间，并记录在题二时间.xlsx（附录四）中。结果如下图所示。

表 3　问题二结果

天数	采样点序号	最短时间/min	最优路径
1	[6, 12, 21, 99, 106, 127, 148, 170]	407.41703781679877	21 -> 6 -> 170 -> 127 -> 99 -> 12 -> 148 -> 106
2	[26, 31, 83, 100, 115, 140, 147, 158]	394.88830033044263	158 -> 83 -> 147 -> 100 -> 31 -> 115 -> 140 -> 26
3	[3, 81, 96, 112, 135, 177, 221, 222]	419.98625349577384	3 -> 221 -> 222 -> 135 -> 112 -> 96 -> 81 -> 177
4	[36, 48, 58, 65, 98, 109, 121, 212]	442.0714054140091	98 -> 48 -> 36 -> 65 -> 121 -> 109 -> 58 -> 212
5	[47, 105, 124, 134, 139, 151, 167, 192]	401.1866983184913	124 -> 105 -> 151 -> 139 -> 134 -> 192 -> 47 -> 167
6	[66, 95, 114, 178, 180, 219,	448.5523734554857	180 -> 114 -> 220 -> 66 -> 95 -> 219 -> 178 -> 211

	220, 211]		
7	[19, 20, 44, 61, 200, 202, 203, 204]	421.7530657949371	19 -> 204 -> 203 -> 20 -> 200 -> 202 -> 44 -> 61
8	[2, 4, 49, 67, 142, 156, 157, 164]	409.34534478788	164 -> 157 -> 4 -> 67 -> 142 -> 2 -> 49 -> 156
9	[24, 39, 45, 131, 175, 213, 214, 194]	414.3415234984422	214 -> 213 -> 39 -> 175 -> 24 -> 45 -> 131 -> 194
10	[33, 51, 89, 126, 152, 193, 179, 169]	445.6728092221021	89 -> 33 -> 126 -> 152 -> 51 -> 193 -> 169 -> 179
11	[9, 22, 32, 34, 55, 130, 154, 163]	417.0665611735286	130 -> 34 -> 32 -> 9 -> 22 -> 154 -> 55 -> 163
12	[35, 41, 75, 79, 104, 107, 145, 162]	407.80539928884775	145 -> 162 -> 104 -> 35 -> 41 -> 75 -> 107 -> 79
13	[59, 87, 93, 122, 125, 168, 184, 185]	421.6897822100704	122 -> 59 -> 125 -> 168 -> 87 -> 93 -> 185 -> 184
14	[46, 60, 69, 73, 80, 92, 133, 174]	399.53381410661774	174 -> 80 -> 92 -> 46 -> 60 -> 133 -> 73 -> 69
15	[7, 23, 28, 42, 50, 62, 171, 172]	415.4213689932219	50 -> 172 -> 23 -> 42 -> 7 -> 28 -> 171 -> 62
16	[14, 18, 37, 40, 102, 138, 153, 176]	408.85977634629353	14 -> 37 -> 40 -> 176 -> 153 -> 18 -> 102 -> 138
17	[84, 86, 88, 90, 91, 94, 118, 136]	409.0841704020767	118 -> 88 -> 91 -> 136 -> 94 -> 86 -> 90 -> 84
18	[17, 70, 119, 132, 141, 155, 165, 166]	427.995171711283	166 -> 141 -> 165 -> 132 -> 119 -> 17 -> 70 -> 155
19	[8, 15, 43, 54, 63, 68, 76, 82]	403.81908807958564	43 -> 68 -> 76 -> 8 -> 54 -> 82 -> 63 -> 15
20	[16, 30, 85, 120, 186, 187, 188, 189]	436.7782419672873	16 -> 189 -> 120 -> 30 -> 85 -> 186 -> 188 -> 187
21	[1, 11, 57,	443.8190385565755	57 -> 1 -> 110 -> 78 -> 217 -> 218 -> 11 -> 208

	78, 110, 217, 218, 208]		
22	[27, 77, 113, 116, 144, 160, 207, 173]	417.8038332492774	77 -> 27 -> 160 -> 116 -> 113 -> 144 -> 173 -> 207
23	[13, 25, 97, 128, 143, 190, 191, 195]	389.10572053664384	13 -> 128 -> 143 -> 25 -> 191 -> 195 -> 190 -> 97
24	[38, 198, 197, 209, 196, 159, 146, 201]	474.5002759056946	197 -> 198 -> 159 -> 146 -> 196 -> 209 -> 201 -> 38
25	[29, 53, 72, 205, 206, 199, 137, 129]	435.82949411118767	29 -> 72 -> 53 -> 205 -> 206 -> 129 -> 199 -> 137
26	[64, 103, 108, 117, 101, 216, 215, 210]	465.8091795102142	:215 -> 216 -> 210 -> 101 -> 103 -> 64 -> 117 -> 108
27	[5, 10, 52, 56, 71, 123, 149, 150]	415.7196630629414	5 -> 10 -> 149 -> 71 -> 123 -> 150 -> 56 -> 52
28	[74, 111, 161, 181, 182, 183]	327.1029846448918	111 -> 183 -> 161 -> 74 -> 182 -> 181

5.2.5 时间结果的按升序的冒泡排序

排序结果: [['第 28 天', 327.1029846448918], ['第 23 天', 389.10572053664384], ['第 2 天', 394.88830033044263], ['第 14 天', 399.53381410661774], ['第 5 天', 401.1866983184913], ['第 19 天', 403.81908807958564], ['第 1 天', 407.41703781679877], ['第 12 天', 407.80539928884775], ['第 16 天', 408.85977634629353], ['第 17 天', 409.0841704020767], ['第 8 天', 409.34534478788], ['第 9 天', 414.3415234984422], ['第 15 天', 415.4213689932219], ['第 27 天', 415.7196630629414], ['第 11 天', 417.0665611735286], ['第 22 天', 417.8038332492774], ['第 3 天', 419.98625349577384], ['第 13 天', 421.6897822100704], ['第 7 天', 421.7530657949371], ['第 18 天', 427.995171711283], ['第 25 天', 435.82949411118767], ['第 20 天', 436.7782419672873], ['第 4 天', 442.0714054140091], ['第 21 天', 443.8190385565755], ['第 10 天', 445.6728092221021], ['第 6 天', 448.5523734554857], ['第 26 天', 465.8091795102142], ['第 24 天', 474.5002759056946]]

5.2.6 工作时间最值结果

整个周期内工作时间最短的是第 28 天，当天最优时间为 474.5002759056946 min
整个周期内工作时间最长的是第 24 天，当天最优时间为 474.5002759056946 min

故得到所有最优路径的工作时间的最大值为 474.5002759056946 min，最小值为 327.1029846448918 min。

5.3 问题三的模型建立和求解

5.3.1 对问题二的方案进行时限和均衡化检验：

1）时限检验首先对问题二得到的方案进行检验，判断其是否存在工作组每天工作时间超过限时或工作时间不均衡的现象。由问题二的结果可知，工作时间的最大值是474.5002759056946 min，和题目要求的 8.5h（=510min）进行比较，发现并没有超过最大限时。所以可知问题二中的方案每一天都在最大工作时间范围内。

2）均衡化检验，我们使用标准差和洛伦兹曲线来衡量。

1. 标准差计算：

计算得到总体标准差是： 26.82523171286905

因此，在阈值为 0.08 的条件下，数据均匀性检验结果为: 均匀

这说明工作时间的分配具有良好的均衡性。

2.洛伦兹曲线：

首先对洛伦兹曲线进行说明[3]。洛伦兹曲线主要用于描述和比较一个国家或地区内收入分配不平等状况，其思想也广泛被用于绘制类似曲线，观察数据分布的均衡性。我们将 28 个工作时间升序排序后，计算累计前 X%的天数的总时长占 28 天总时长的百分之几，这个数值就是对应 X 的 Y，洛伦兹曲线就是这一函数的图像。

当总时长平均分配到 28 天中时，得到完全平等线。我们把这两条曲线进行绘制，得到下图所示，可以看到实际洛伦兹曲线与完全平等线非常贴合。

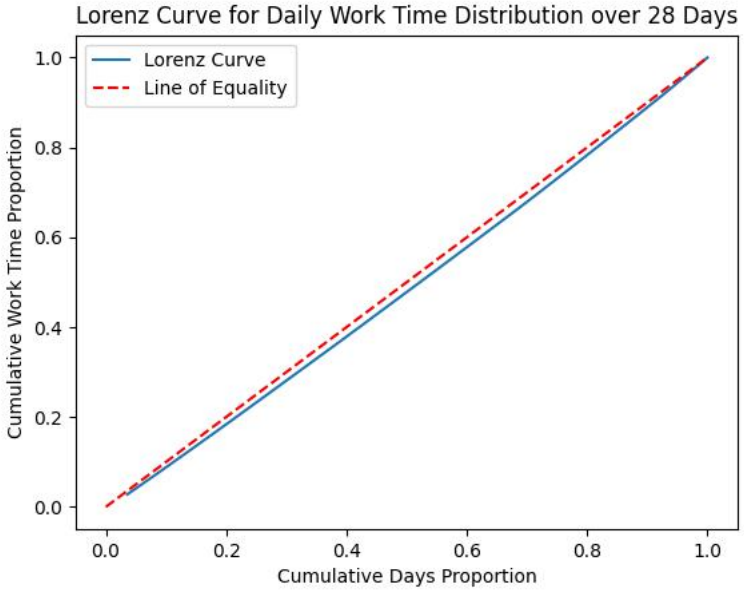


图 8 问题二洛伦兹检验曲线图

注释：洛伦兹曲线：Lorenz Curve

完全平等线：Line of Equality

3.计算基尼（Gini）系数。

不平等面积=洛伦兹曲线与完全平等线之间的面积

Gini 系数=2*（不平等面积/正方形面积）

由此得到 Gini 系数为： 0.03294262157395665

由于基尼系数的值域为[0,1]，约接近 0 表明数据的均衡性越高。由此可知，从洛伦兹曲线的角度衡量，我们的问题二的工作时间分配具有良好的均衡性。

综合上述，问题二工作时间的分配**非常均衡**。

5.3.2 取消每天 8 个点位数的限制，能在工作时限内尽快完成土壤采样工作的均衡化的方案[4]：

这一问属于较为复杂的在多约束条件下寻找最优解的问题。因此，我们采用（元）启发式算法来近似求解，尽可能地逼近最优状态。

在此我们借鉴了开放式多车辆路径问题（Open Multi-Vehicle Routing Problem, OMVRP）的解决思路，将车辆数类比天数，即将问题转化为“一辆车完成一个路径上的任务，每辆车的路径经过点不重叠，使用最少的车辆数完成所有任务，且每辆车的路径长度保持均衡性”的最优解的问题。

该问题有以下特点：

开放式路径：车辆完成任务后无需返回起始点。

多车辆服务：需要分配多辆车共同完成配送任务。

路径不重叠：每个客户点只能被一辆车访问。

最少车辆数：目标是使用尽可能少的车辆完成所有任务。

路径长度均衡：在尽可能少使用车辆的同时，需要保持每辆车的路径长度相对均衡。

建模之前，首先对情形进行假设：

1、假设天数是车辆数，采样点是客户点，路径不重叠；

假设配送中心拥有多辆车

2、假设车辆在完成任务后不必返回配送中心，而是可以选择停靠在路径上的某个客户点。

3、假设车辆行驶的距离为两点之间的直线距离。

假设每个客户只能被一辆车服务一次。

4、假设客户点的位置是已知的，且固定不变。

5、假设车辆的行驶速度可以视为常数。

5.3.3 模型建立和求解过程

在实际建模过程中，我们先后使用了最近邻算法，贪心算法，禁忌搜索算法，最优等方法来尽可能逼近最优解。简要流程如下图所示。

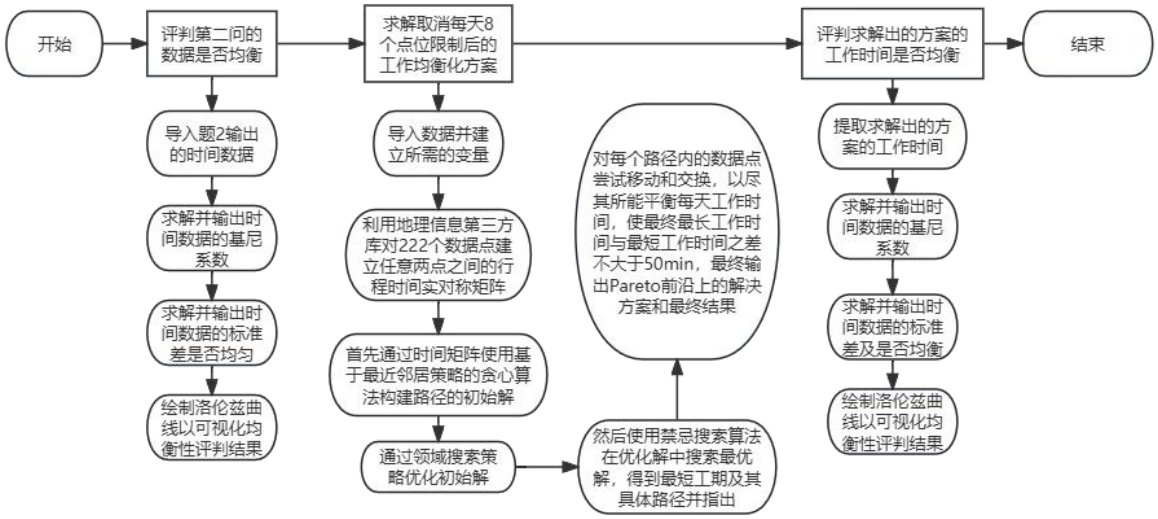


图 9　问题三流程图

1）**算法介绍：**为方便理解，我们对使用到的算法进行简单介绍。

最近邻算法（KNN）

是一种基于实例的学习方法，通过计算样本之间的距离来进行分类或回归预测。其目标是找到与待预测样本最相似的 K 个训练样本，并根据这 K 个样本的标签进行预测。其对异常值敏感，对于大规模数据集的分类有较好的效果，但是计算复杂度也较高。

最近邻算法和问题二中使用过的贪心算法结合，特别适于优化 KNN 算法进行分类或回归时，因为特征维度很高引起的计算复杂度会显著增加问题。通过使用贪心算法的思想进行特征选择，即每一步选择对当前分类或回归效果提升最大的特征，可以减少特征维度并降低计算复杂度。

禁忌搜索算法（TS）

是一种元启发式随机搜索算法，旨在通过避免陷入局部最优解来寻找全局最优解，特别适用于解决类似本题的复杂优化问题，且被大量应用于 TSP 旅行商问题中。它的算法思想是从一个初始可行解出发，通过选择一系列的特定搜索方向作为试探，以实现目标函数值的最大化或最小化。为了避免陷入局部最优解，算法采用了一种灵活的“记忆”技术，即禁忌表，对已经进行的优化过程进行记录和选择，以指导下一步的搜索方向。

Pareto 最优

Pareto 最优（Pareto Optimality），是经济学和管理学中的一个重要概念，它描述了一种资源分配的理想状态。在实际应用中很难达到绝对的 Pareto 最优状态，我们使用 Pareto 最优检查的方法来评价资源配置的效率，即工作时间的均衡度。

2）**具体步骤：**

Step1 首先利用 geopy 库对 222 个采样点建立任意两点之间的行程时间的实对称矩阵（即时间矩阵），接着对时间矩阵使用基于最近邻矩阵策略的贪心算法构建路径的初始解。对时间矩阵中的每对采样点的最短时间迭代构建每条路径。在每条路线的构建过程中，选择与当前采样点距离最近且能满足路线限制条件的任务，将其添加到当前路径中，并从未分配采样点中移除。当无法找到满足条件的最近采样点时，结束当前路径的构建。最后，返回包含所有路径的列表。

Step2 通过领域搜索策略优化初始解，考虑多次交换，通过不断尝试交换路线中的任务来寻找局部最优解，以期得到更好的路径安排。

计算初始最佳路径的评价函数值。通过三重循环遍历所有可能的一次交换的组合。对于每一次交换，先判断交换后的新路径是否有效。如果新路径有效，则计算新路径的评价函数值；如果新路线的评价函数值比当前最佳路线的评价函数值小，则更新最佳路径和最佳评价函数值。最终在循环结束后返回最佳路线。

Step3 然后使用禁忌搜索算法在优化解中搜索最优解，得到最短工作时间及其具体路径并输出，其通过不断迭代和对每个路径内的采样点尝试移动来逐步优化路线，同时使用禁忌列表避免重复的交换操作，以尽可能平衡每天的工作时间，使最终最长工作时间与最短工作时间之差不大于 50min。

Step4 最终输出 Pareto 前沿上的解决方案和最终结果，并使用 cost, balance 和 routes 进行进一步的处理。

3）**求解结果：**由于求复杂的最优解问题时往往很难真正得到最优结果，只能尽可能地逼近完全均衡化状态。所以我们在均衡化过程中，将工作时间的极差尽可能地缩小。最后限于算力和算法优化问题，选定极差为 50%，结果详见题三时间.xlsx（附录四）

调整以后得到以下结果：

表 4　问题三结果

天数	路径	时间（min）
1	[1, 57, 110, 78, 218, 11, 19, 204, 203]	479.7075739
2	[169, 142, 157, 4, 67, 166, 179, 80, 174]	469.718539
3	[3, 221, 222, 135, 81, 177, 37, 14, 198]	472.4060621
4	[5, 10, 149, 77, 123, 150, 56, 52, 84]	477.0665739
5	[6, 212, 211, 21, 106, 148, 12, 99, 144]	474.274913
6	[7, 42, 23, 172, 50, 62, 171, 28, 207]	483.7492323
7	[54, 82, 63, 15, 129, 158, 83, 147, 100]	447.0962305
8	[32, 107, 75, 41, 35, 104, 162, 79, 145]	454.0171548
9	[13, 128, 215, 143, 216, 190, 195, 191, 146]	472.8354763
10	[16, 124, 105, 151, 194, 47, 167, 120, 30]	477.6457462
11	[93, 119, 132, 165, 141, 164, 2, 49, 156]	469.3831052
12	[18, 153, 176, 40, 197, 185, 184, 59, 182]	474.2774384
13	[44, 200, 202, 20, 201, 199, 43, 68, 137]	483.244774
14	[101, 130, 154, 55, 173, 163, 170, 127, 22]	476.016786
15	[175, 39, 213, 214, 131, 45, 122, 161, 74]	452.7374123
16	[26, 140, 115, 31, 155, 70, 53, 205, 206]	471.7669238
17	[113, 160, 116, 77, 103, 64, 117, 121, 58, 38]	490.2683428
18	[29, 72, 138, 102, 125, 87, 168, 183, 111]	478.9108842
19	[33, 89, 51, 152, 187, 188, 186, 85, 189]	488.737309
20	[34, 208, 178, 114, 220, 66, 95, 219, 180]	477.0802028
21	[36, 65, 98, 27, 108, 109, 76, 91]	481.487291
22	[217, 61, 60, 46, 92, 133, 69, 73, 17]	470.1502653
23	[159, 90, 86, 94, 196, 136, 139, 193, 134, 192]	484.905018
24	[88, 118, 209, 210, 96, 112, 25, 97, 126]	479.4668882
25	[48, 181, 9, 24, 8]	447.7497349

同时得到均衡性检验的结果：

数据 Gini 系数为： 0.012817639900845848

总体标准差是： 11.451830021872741

检验标准差是否有超出阈值:

[False False False False False False True False False False False False False False False False False False False False False False False True]

在阈值为 0.08 的条件下，数据均匀性检验结果为: 均匀

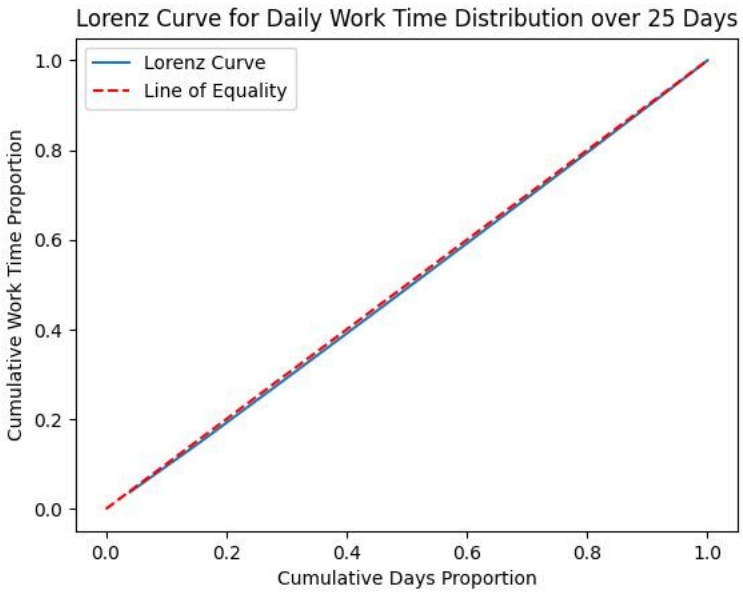


图 10　问题三洛伦兹检验曲线图

注释：洛伦兹曲线：Lorenz Curve

完全平等线：Line of Equality

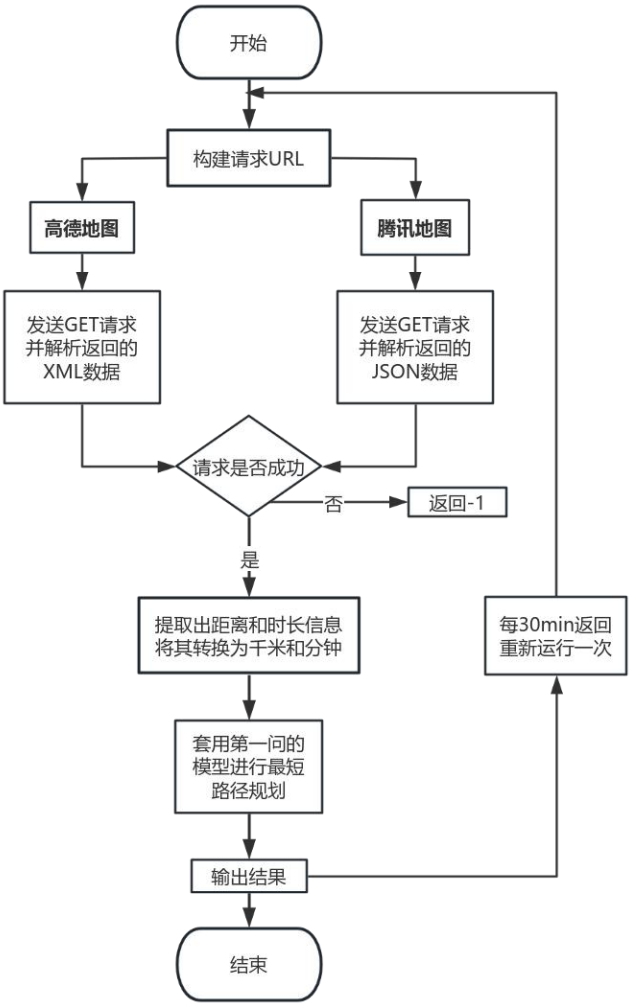
由可视化结果和 Gini 系数可以看出，其均衡性非常高，在问题二的基础上也有了很大的提升。

5.4 问题四的模型建立与求解

在问题一模型的基础上，我们接入高德地图和腾讯地图的 API。

需要说明的是，由于百度地图无法根据给定的经纬度定位到这 8 个采样点的位置，所以我们只分别调用了高德地图和腾讯地图的 API 进行最优路径规划。

具体步骤如下方所示:



***Step1* 通过高德地图 API 和腾讯地图 API 获取两个位置之间的距离和时长：**使用函数构建请求 URL，其中包含了高德地图和腾讯地图的 API 的密钥和起始位置、目标位置的信息。对于高德地图，发送 GET 请求并解析返回的 XML 数据；对于腾讯地图，发送 GET 请求并解析返回的 JSON 数据。

***Step2* 提取距离和时长信息：**请求成功时，分别从两个地图开发平台提取出距离和时长信息，并将其转换为千米和分钟。

***Step3* 套用问题一的最短路径规划模型：**将获得的距离和时长信息以和问题一相同的方式先转化为时间矩阵，随后套用最短路径规划模型得出最优路径和最短工作时间。

***Step4* 动态获取数据：**通过自动运行脚本每隔 30min 运行一次程序，重复以上流程获得动态监测的结果。

图 11　问题四算法流程

由于算法和数据集上存在差异，对于同一时刻高德地图和腾讯地图规划得到的最短时间和最优路径也有差异。因此我们同时使用两者进行路径规划，随后将结果进行比较后，选择更短的时间和对应的路径作为最优路径方案输出。

值得注意的是，由于实际路况一直在变化，所以工作组的最短工作时间和最优路径（采样顺序）也会一直变化。

为了尽可能得到明确的最优方案，我们在一天中的不同时刻，对两个平台的数据进行多次调用，将每次得到的结果汇总，确定了大致的最短时间范围。

在这里我们创建了自动运行的脚本，通过每三十分钟调用一次 API，记录了 2024 年 7 月 26 日从 0:30-15:30 期间连续的 30 个样本点,对汇总结果进行分析，得到表格 Q4.xlsx（见附录四）

并将最短工作时间绘制成如图（12）所示的散点图。

图 12　最短时间散点图

即在不考虑突发情况等重大因子影响的情况下，工作组的最短工作时间大致为410min 到 412.5min 之间，平均值为 411.50967741936min,总体波动很小。

又考虑到工作组一般在白天工作,所以最短工作时间在 410.0min 到 411.5min 之间，一般使用高德地图导航，最优路径规划主要为 83 -> 158 -> 100 -> 147 -> 31 -> 115 -> （61 -> 44/44->61）。

六、模型的评价

6.1 模型的优点

1. 算法多样性：文件中涵盖了多种类型的算法，包括路径查找（Floyd-Warshall）、组合优化（回溯法）、数据聚类（K-means）、启发式搜索（贪心算法和禁忌搜索算法）等，这种多样性使得在面对不同问题时可以选择最合适的算法。
2. 高效解决方案：通过应用 Floyd-Warshall 算法和贪心算法等，能够快速找到问题的近似或最优解，尤其是在路径规划和资源分配等场景中表现出色。
3. 灵活性和适应性:算法的选择和应用可以根据具体问题的需求进行调整和优化,例如，通过调整 K-means 聚类算法中的聚类数或禁忌搜索算法中的参数，来适应不同的数据集或问题规模。
4. 在问题三求解取消每天 8 个采样点数限制后工作时间均衡化的方案中，我们通过类比开放式多车辆路径问题（OMVRP），将所需车辆数类比工作天数，使用基于最近邻居的贪心算法来构造满足路径不重叠的约束，并尽量使用较少的车辆的初始解，再邻域搜索策略来优化初始解，尝试交换不同路径上的客户点，或者合并和拆分路径，以减少车辆数并保持路径长度的均衡性，并通过禁忌搜索算法搜索最优解，最后使用 Pareto 前沿法来搜索多个目标之间的最优权衡，通过均衡性检验说明了我们模型求解的可靠性。
5、在问题四中，我们通过使用高德地图和腾讯地图两个 API，并比较使用两者后的时间，取最优方案，选用最优 API，并总结了常用路径，计算了平均时间，有助于工作人员动态规划自己的工作顺序，尽量减少自己的工作时间，体现了我们的人文关怀。

6.2 模型的缺点

1. 算法复杂性：不同算法具有不同的时间和空间复杂度，对于大规模数据或复杂问题，可能会遇到计算效率低下或资源消耗过大的问题。
2. 参数敏感性：部分算法对参数的选择非常敏感，如 K-means 聚类的初始质心选择、

禁忌搜索的禁忌列表长度等，这些参数的选择直接影响算法的性能和结果。
3. 全局最优性的挑战：除了 Floyd-Warshall 算法能保证找到全局最短路径外，其他启发式搜索算法（如贪心算法和禁忌搜索算法）通常只能找到近似最优解，可能无法确保全局最优性。

七、参考文献

[1] V. Hegde, T. S. Aswathi and R. Sidharth, "Student residential distance calculation using Haversine formulation and visualization through GoogleMap for admission analysis," 2016 IEEE International Conference on Computational Intelligence and Computing Research (ICCIC), Chennai, India, 2016, pp. 1-5, doi: 10.1109/ICCIC.2016.7919699. keywords: {Data mining;Data visualization;Google;Organizations;Industries;Clustering algorithms;Earth;Haversine Formulae;KMeans Algorithm;Admission Analysis},
[2] K.Singh, S. K. Bedi and P. Gaur, "Identification of the most efficient algorithm to find Hamiltonian Path in practical conditions," 2020 10th International Conference on Cloud Computing, Data Science & Engineering (Confluence), Noida, India, 2020, pp. 38-44, doi: 10.1109/Confluence47617.2020.9058283. keywords: {Conferences;Cloud computing;Data science;Hamiltonian Path;Travelling Salesman Problem;optimization;Heuristic;Algorithms;Web-application},
[3]高子健,初良勇.考虑订单优先级带时间窗的多车型开放式车辆路径问题研究[J].科学技术与工程,2024,24(6):2521-2529

附录

附录一 支撑材料列表

支撑材料	
源代码	calculate_adjacency_matrix calculate_path_distance find_shortest_hamiltonian_path main Q1.Floyd-Warshall算法+回溯与dfs Q2.K-Means聚类+规划 Q3.基于最近邻居的贪心算法+邻域搜索策略+禁忌搜索算法+Pareto前沿法 Q4.GD_TX_API-FW-DFS.py Q4_run_script
数据	Q1.邻接矩阵 Q4.output Q4 题2时间 题3时间
图片	Q2.Lorenz Curve Q2.分簇结果 Q2.最短哈密尔顿路径生成图 Q3.Lorenz Curve
文献	Identification of the most effcient algorithm to find Hamiltonian Path in practical conditions.jsp Student residential distance calculation using Haversine formulation and visualization through GoogleMap 考虑订单优先级带时间窗的多车型开放式车辆路径问题研究 我国区域体育产业均衡性发展研究-基于基尼系数和洛伦兹曲线

附录二 程序代码

代码 1：mian.m(包含 4 个代码文件)

```
%main
coordinates = [
    38.110755,111.040331; %6
    38.112507,111.042637;
    38.117061,111.073839;
    38.116762,111.069797;
    38.124064,111.057493;
    38.119702,111.044954;
    38.118705,111.046929;
    38.127754,111.065254;

];%将每一个采样点作为索引顺序为 1 的采样点依次尝试，得到当第六个采样点的索
```

引为 1 时距离最短。

```
adjacency_matrix = calculate_adjacency_matrix(coordinates);

shortest_hamiltonian_path =
find_shortest_hamiltonian_path(adjacency_matrix);
disp('最短的哈密顿路径：');
disp(shortest_hamiltonian_path);
total_distance = calculate_path_distance(shortest_hamiltonian_path,
adjacency_matrix);
disp('最短的哈密尔顿路径的总距离：');
disp(total_distance);
```

```
%calculate_adjacency_matrix
% 假设 adjacency_matrix 是一个 n × n 的邻接矩阵，表示完全图的权重
function adjacency_matrix = calculate_adjacency_matrix(coordinates)
    n = size(coordinates, 1);
    adjacency_matrix = zeros(n);

    % 计算采样点之间的球面距离，并作为权重填充邻接矩阵
    for i = 1:n
        for j = 1:n
            if i ~= j
                lat1 = deg2rad(coordinates(i, 1));
                lon1 = deg2rad(coordinates(i, 2));
                lat2 = deg2rad(coordinates(j, 1));
                lon2 = deg2rad(coordinates(j, 2));

                % 使用 Haversine 公式计算球面距离
                R = 6371; % 地球半径（单位：公里）
                dlat = lat2 - lat1;
                dlon = lon2 - lon1;
                a = sin(dlat/2)^2 + cos(lat1) * cos(lat2) * sin(dlon/2)^2;
                c = 2 * atan2(sqrt(a), sqrt(1-a));
                distance = R * c;

                % 填充邻接矩阵
                adjacency_matrix(i, j) = distance;
            end
        end
    end
end

% 给出每个采样点的经纬度坐标
```

```
%find_shortest_hamiltonian_path
```

```
function shortest_hamiltonian_path =
find_shortest_hamiltonian_path(adjacency_matrix)
    n = size(adjacency_matrix, 1);
    shortest_path = [];
    shortest_length = Inf;

    path = [1];
    visited = zeros(1, n);
    visited(1) = 1;

    function backtrack(current_node, current_length, path)
        if length(path) == n
            % 当路径长度为 n 时，我们找到了一条哈密顿路径
            if current_length < shortest_length
                shortest_length = current_length;
                shortest_path = path;
            end
            return;
        end

        for next_node = 1:n
            if ~visited(next_node)
                visited(next_node) = 1;
                backtrack(next_node, current_length +
adjacency_matrix(current_node, next_node), [path, next_node]);
                visited(next_node) = 0;
            end
        end
    end

    backtrack(1, 0, path);
    shortest_hamiltonian_path = shortest_path;
end
```

```
%calculate_path_distance
function total_distance = calculate_path_distance(path, adjacency_matrix)
    n = length(path);
    total_distance = 0;
    for i = 1:n-1  % 遍历路径中的边（不包括回到起点的边）
        total_distance = total_distance + adjacency_matrix(path(i),
path(i+1));
    end
end
%将最小距离依次相加得到结果
```

代码 2：Q1.Floyd-Warshall 算法+回溯与 dfs

```
import pandas as pd
from geopy.distance import geodesic

# Floyd-Warshall 算法实现，返回所有采样点对之间的最短路径长度
def floyd_warshall(graph):

      num_vertices = len(graph)
      dist = [[graph[i][j] for j in range(num_vertices)] for i in
range(num_vertices)]

      for k in range(num_vertices):
            for i in range(num_vertices):
                  for j in range(num_vertices):
                        if dist[i][j] > dist[i][k] + dist[k][j]:
                              dist[i][j] = dist[i][k] + dist[k][j]
      return dist

# 使用回溯策略来构建一个访问所有采样点的路径，并找到最短时间
def find_path(dist, num_vertices):

      visited = [False] * num_vertices
      path = []
      min_time = float('infinity')

      def dfs(node, current_path, current_time):
            nonlocal min_time, path
            visited[node] = True
            current_path.append(node)

            if len(current_path) == num_vertices:
                  if current_time < min_time:
                        min_time = current_time
                        path = current_path[:]
            else:
                  for next_node in range(num_vertices):
                        if not visited[next_node]:
                              dfs(next_node, current_path, current_time +
dist[node][next_node])

            visited[node] = False
            current_path.pop()
```

```
      # 从每个节点开始尝试
      for start_node in range(num_vertices):
            dfs(start_node, [], 0)

      return min_time, path

"""
# 示例数据和调用
graph = [
      [0, 10, 15, 20],
      [10, 0, 35, 25],
      [15, 35, 0, 30],
      [20, 25, 30, 0]
]"""
#----------读取数据----------
df=pd.read_excel("附件 1：xx 地区.xlsx")
points_index=[31,44,61,83,100,115,147,158]
points_detail=[]
speed=20
dwell_time=[0,0,0,0,0,0,0,0]
graph=[]

#----------提取所需点位----------
for i in range(len(points_index)):
      points_detail.append(list(df.iloc[points_index[i]-1]))
      dwell_time[i]=points_detail[i][3]
#print(dwell_time)
#print(points_detail)

#----------计算时间矩阵----------
for i in range(len(points_detail)):
      graph.append([])
      for j in range(len(points_detail)):
            graph[i].append(geodesic((points_detail[i][2],
points_detail[i][1]), (points_detail[j][2],
points_detail[j][1])).kilometers/speed*60)
#print(travel_time_matrix)
num_vertices = len(graph)

# 计算所有点对之间的最短路径
dist = floyd_warshall(graph)

# 找到一个访问所有点的最短单向路径及其时间
min_time, path = find_path(dist, num_vertices)
```

```
#----------输出结果----------
print("-----------------------------------------------------------------
----------------")
print("最短时间:", min_time+sum(dwell_time))
print("最优路径为",end=":")
for i in range(len(path)-1):
      print(points_index[path[i]],end=" -> ")
print(points_index[path[-1]])
```

代码 3：Q2.K-Means 聚类+规划

```
import pandas as pd
import numpy as np
from sklearn.cluster import KMeans
import matplotlib.pyplot as plt
from pyproj import Proj, transform
from geopy.distance import geodesic


# ----------Floyd-Warshall 算法实现，返回所有点对之间的最短路径长度
----------
def floyd_warshall(graph):
    num_vertices = len(graph)
    dist = [[graph[i][j] for j in range(num_vertices)] for i in
range(num_vertices)]
    for k in range(num_vertices):
        for i in range(num_vertices):
            for j in range(num_vertices):
                if dist[i][j] > dist[i][k] + dist[k][j]:
                    dist[i][j] = dist[i][k] + dist[k][j]
    return dist


# ----------使用回溯策略深度搜索 dfs 来构建一个访问所有点的路径并找到最短时
间----------
def find_path(dist, num_vertice):
    visit = [False] * num_vertice
    path = []
    min_time = float('infinity')
    def dfs(nodes, cr_path, cr_time):
        nonlocal min_time, path
        visit[nodes] = True
        cr_path.append(nodes)
        if len(cr_path) == num_vertice:
```

```
            if cr_time < min_time:
                min_time = cr_time
                path = cr_path[:]
        else:
            for nextNode in range(num_vertice):
                if not visit[nextNode]:
                    dfs(nextNode, cr_path, cr_time + dist[nodes][nextNode])
        visit[nodes] = False
        cr_path.pop()

    # ----------从每个节点开始尝试----------
    for startNode in range(num_vertice):
        dfs(startNode, [], 0)
    return min_time, path


# ----------使用贪心算法调整分组----------
def adjust_cluster(clusters):
    # ----------确定哪些组需要调整----------
    cluster_s = np.array([len(cluster) for cluster in clusters])
    targer_s = 8
    last_cs = 6

    # ----------使用贪心算法进行调整----------
    while not all(cluster_s == targer_s) and not (
            np.all(cluster_s[:-1] == targer_s) and cluster_s[-1] ==
last_cs):
        # ----------找到需要移除点的组和需要添加点的组----------
        too_big = np.where(cluster_s > targer_s)[0]
        too_small = np.where(cluster_s < targer_s)[0]

        if too_big.size > 0 and too_small.size > 0:
            # ----------选择一个太大的组和一个太小的组----------
            from_idx = too_big[0]
            to_idx = too_small[0]

            # ----------检查选定的太大组是否不为空----------
            if len(clusters[from_idx]) > 0:
                # ----------从太大的组中移除一个点，并添加到太小的组中
----------
                point_to_move = clusters[from_idx].pop()
                clusters[to_idx].append(point_to_move)

                # ----------更新组大小----------
```

```
                cluster_s[from_idx] -= 1
                cluster_s[to_idx] += 1
            else:
                # ----------如果选定的太大组为空，则跳出循环并打印错误消息
----------
                print("Error: Attempted to pop from an empty cluster.")
                break
        else:
            # ----------如果没有太大或太小的组可以调整，就打破循环
----------
            break

    return clusters

# ----------定义 WGS 84 坐标系----------
wgs84 = Proj(proj='latlong', datum='WGS84')

# ----------定义 UTM 坐标系（以区域 33N 为例）----------
utm = Proj(proj='utm', zone=33, datum='WGS84')

# ----------读取数据----------
df = pd.read_excel("附件 1：xx 地区.xlsx")
point_original = df[['JD', 'WD']].values.tolist()

# ----------将 WGS 84 坐标转换为 UTM 坐标----------
x, y = transform(wgs84, utm, df['JD'].values, df['WD'].values)
points=list(zip(x,y))

# ----------使用 K-means 聚类----------
num_clusters = int(np.ceil(len(points) / 8))  # 计算需要的聚类数量
kmeans = KMeans(n_clusters=num_clusters, random_state=1).fit(points)
labels = kmeans.labels_

# ----------聚类结果转换为分组----------
clusters = [[] for _ in range(num_clusters)]
for i, label in enumerate(labels):
    clusters[label].append(points[i])

# ----------调整后的分组----------
adjusted_clusters = adjust_cluster(clusters)

# ----------打印聚类结果----------
#for i, cluster in enumerate(adjusted_clusters):
#    print(f"Group {i + 1}: {cluster}")
```

```
results= {}
# ----------汇总聚类结果----------
for i, cluster in enumerate(adjusted_clusters):
    groups=[]
    for j in range(len(cluster)):
        groups.append(points.index(cluster[j])+1)
    results[f"第{i+1}天"]=groups

#file_name=0
for i in adjusted_clusters:
    # ----------提取 x 和 y 坐标----------
    x = [co[0] for co in i]
    y = [co[1] for co in i]

    # ----------绘制散点图----------
    plt.scatter(x, y)
    plt.xlabel('X Coordinate')
    plt.ylabel('Y Coordinate')
    plt.title('Total Scatter Plot of Coordinates')
    #plt.title(f'{i}Scatter Plot of Coordinates')

    # ----------保存每个簇的散点图----------
    #plt.savefig(f"2.{file_name+1}.png")
    #file_name += 1
    #plt.show()

# ----------保存总图----------
plt.savefig("2.total.png")
plt.show()

timedata=[]
for key, value in results.items():

print("-----------------------------------------------------------------
----------------")
    print(f"{key}: {value}")
    points_detail = []
    speed = 20
    dwell_time = [0] * len(value)
    graph = []
    for i in range(len(value)):
        points_detail.append(list(df.iloc[value[i] - 1]))
        dwell_time[i] = points_detail[i][3]
```

```
    for i in range(len(points_detail)):
        graph.append([])
        for j in range(len(points_detail)):
            graph[i].append(geodesic((points_detail[i][2],
points_detail[i][1]),
                                     (points_detail[j][2],
points_detail[j][1])).kilometers / speed * 60)
    # print(travel_time_matrix)
    num_vertices = len(graph)
    dist = floyd_warshall(graph)
    min_time, path = find_path(dist, num_vertices)
    timedata.append([key,min_time + sum(dwell_time)])
    print(f"{key}最短时间:{min_time + sum(dwell_time)} min")
    print(f"{key}最优路径为", end=":")
    for i in range(len(path) - 1):
        print(value[path[i]], end=" -> ")
    print(value[path[-1]])

print("----------------------------------------------------------------
----------------")

# ----------保存时间结果----------
df2=pd.DataFrame(timedata,columns=["日期","工作时长/min"])
df2.to_excel("题 2 时间.xlsx",index=False)

# ----------对时间结果进行冒泡排序----------
for i in range(len(timedata)):
    for j in range(len(timedata)-1,i,-1):
        if timedata[j][1]<timedata[j-1][1]:
            timedata[j],timedata[j-1]=timedata[j-1],timedata[j]

# ----------输出最终结果----------
print("----------------------------------------------------------------
----------------")
print("综上所述:")
print("1、排序结果:",timedata)
print(f"2、整个周期内工作时间最短的是{timedata[0][0]}，当天最优时间为
{timedata[0][1]} min")
print(f"3、前 27 天内工作时间最短的是{timedata[1][0]}，当天最优时间为
{timedata[1][1]} min")
print(f"4、整个周期内工作时间最长的是{timedata[-1][0]}，当天最优时间为
{timedata[-1][1]} min")
print("----------------------------------------------------------------
----------------")
```

代码 4 包括三个部分
Part One:计算时间平衡函数，检验均衡性
Part Two:处理取消每天 8 个采样点数的限制时的工作均衡化方案的函数
Part Three:主程序

代码 4：Q3.基于最近邻居的贪心算法+邻域搜索策略+禁忌搜索算法+Pareto 前沿法

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import inequality
from geopy.distance import geodesic
import copy

"""
    ######################
    Part One:计算时间平衡函数，检验均衡性
    ######################
"""



# ----------计算洛伦兹曲线函数----------
def calculate_lorenz_curve(work_times):
    #param work_times: 一个包含 28 天每天工作时间的列表
    #return: 两个列表，分别表示累计天数占比和累计工作时间占比
    # 将工作时间转换为 numpy 数组，并进行排序
    sorted_work_times = np.sort(np.array(work_times))
    # 计算总工作时间
    total_work_time = np.sum(sorted_work_times)
    # 计算每天工作时间的占比
    work_time_proportion = sorted_work_times / total_work_time
    # 计算累计天数占比（即索引占比）
    cumulative_days_proportion = np.arange(1, len(sorted_work_times) + 1) /
len(sorted_work_times)
    # 计算累计工作时间占比
    cumulative_work_time_proportion = np.cumsum(work_time_proportion)
    return cumulative_days_proportion, cumulative_work_time_proportion



# ----------绘制洛伦兹曲线函数----------
def plot_lorenz_curve(cumulative_days_proportion,
cumulative_work_time_proportion,time):
    #cumulative_days_proportion: 累计天数占比的列表
    #cumulative_work_time_proportion: 累计工作时间占比的列表
    # 绘制洛伦兹曲线
    plt.plot(cumulative_days_proportion, cumulative_work_time_proportion,
```

```
label='Lorenz Curve')
    # 绘制完全平等线（对角线）
    plt.plot([0, 1], [0, 1], 'r--', label='Line of Equality')
    # 设置图表标题和标签
    plt.title(f'Lorenz Curve for Daily Work Time Distribution over {time}
Days')
    plt.xlabel('Cumulative Days Proportion')
    plt.ylabel('Cumulative Work Time Proportion')
    # 显示图例
    plt.legend()
    # 显示图表
    plt.show()


# ----------计算标准差函数----------
def Standard_deviation_test(data):
    # 计算平均值
    mean_value = np.mean(data)
    # 计算样本的标准差
    std_dev = np.std(data, ddof=1)  # ddof=1 表示自由度的更正
    # 计算样本数据的总体标准差
    population_std_dev = np.std(data, ddof=0)  # ddof=0 表示使用总体标准差
的公式
    print("总体标准差是：",population_std_dev)
    threshold = 2.16  # 对应 95%置信区间
    # 检验标准差是否超出阈值
    is_outlier = np.abs(data - np.mean(data)) > threshold *
population_std_dev
    print("检验标准差是否有超出阈值:\n",is_outlier)
    # 数据均匀性检验
    balance_threshold = 0.08
    is_balanced = (population_std_dev / mean_value) < balance_threshold
    print(f"在阈值为{balance_threshold}的条件下，数据均匀性检验结果: {'均
匀' if is_balanced else '不均匀'}")



"""
    ########################
    Part Two:处理取消每天 8 个点位数的限制时的工作均衡化方案的函数
    ########################
"""
# ----------定义 Solution 类，代表解决方案的各项指标----------
class Solution:
```

```
    def __init__(self, routes, time_matrix):
        # 该类用于直接计算总成本和平衡值
        self.routes = routes
        self.cost = sum(calculate_route_time(route, time_matrix) for route
in routes)
        self.balance = calculate_time_balance(routes, time_matrix)

# ----------基于最近邻居的贪心算法构建初始解----------
def greedy_initial_solution(time_matrix):
    una_tasks = list(range(n_tasks))
    routes = []
    while una_tasks:
        current_route = []
        current_task = una_tasks.pop(0)
        current_route.append(current_task)
        while una_tasks:
            closest_task = None
            min_distance = float('inf')
            for task in una_tasks:
                if time_matrix[current_task][task] < min_distance:
                    closest_task = task
                    min_distance = time_matrix[current_task][task]
            if can_add_to_route(current_route, closest_task, time_matrix):
                current_route.append(closest_task)
                una_tasks.remove(closest_task)
                current_task = closest_task
            else:
                break
        routes.append(current_route)

    return routes


# ----------邻域搜索策略优化----------
def local_search(routes, time_matrix):
    best_routes = copy.deepcopy(routes)
    best_obj = calculate_objective(best_routes, time_matrix)

    # ----------考虑一次进行多次交换----------
    for i in range(len(routes)):
        for j in range(i + 1, len(routes)):
            for k in range(j + 1, len(routes)):
                # ---------- 一次执行两次交换----------
                try:
```

```
                    new_routes = swap_tasks(routes, i, 0, j, 0)
                    new_routes = swap_tasks(new_routes, j, 0, k, 0)
                    if is_valid_solution(new_routes, time_matrix):
                        objective = calculate_objective(new_routes,
time_matrix)
                        if objective < best_obj:
                            best_routes = copy.deepcopy(new_routes)
                            best_obj = objective
                except IndexError:
                    continue

    return best_routes


# ----------禁忌搜索算法搜索最优解----------
def tabu_search(routes, time_matrix, tabu_list_size=10,
max_iterations=1000):
    current_routes = copy.deepcopy(routes)
    current_obj = calculate_objective(current_routes, time_matrix)
    best_routes = copy.deepcopy(current_routes)
    best_obj = current_obj
    tabu_list = []
    iteration = 0
    while iteration < max_iterations:
        neighbor_routes = []
        for i in range(len(current_routes) - 1):
            for j in range(i + 1, len(current_routes)):
                if not current_routes[i] or not current_routes[j]:
                    continue
                for task_i in range(len(current_routes[i])):
                    for task_j in range(len(current_routes[j])):
                        if (i, task_i, j, task_j) not in tabu_list:
                            try:
                                new_routes = swap_tasks(current_routes, i,
task_i, j, task_j)
                                if is_valid_solution(new_routes,
time_matrix):
                                    neighbor_routes.append(
                                        (new_routes,
calculate_objective(new_routes, time_matrix)))
                            except IndexError:
                                continue

        if not neighbor_routes:
```

```
            break

        neighbor_routes.sort(key=lambda x: x[1])

        selected_routes, selected_objective = neighbor_routes[0]
        if selected_objective < best_obj:
            best_routes = copy.deepcopy(selected_routes)
            best_obj = selected_objective

        current_routes = copy.deepcopy(selected_routes)
        current_obj = selected_objective

        tabu_list.append((i, task_i, j, task_j))
        if len(tabu_list) > tabu_list_size:
            tabu_list.pop(0)
        iteration += 1

    return best_routes


# --------------------其他辅助计算函数--------------------
# ----------计算路线总时间----------
def calculate_route_time(route, time_matrix):
    total_time = sum(time_matrix[route[i]][route[i + 1]] for i in
range(len(route) - 1))
    total_time += sum(dwell_time[route[i]] for i in range(len(route)))
    return total_time


# ----------判断是否可以添加任务到当前路线----------
def can_add_to_route(route, task, time_matrix):
    new_route = route + [task]
    return calculate_route_time(new_route, time_matrix) <= max_work_time


# ----------计算总成本----------
def calculate_total_cost(routes, time_matrix):
    return sum(calculate_route_time(route, time_matrix) for route in routes)


# ----------判断是否是有效的解决方案----------
def is_valid_solution(routes, time_matrix, min_tasks_per_route=1):
    if any(len(route) < min_tasks_per_route for route in routes):
        return False
```

```
    return all(calculate_route_time(route, time_matrix) <= max_work_time for
route in routes)


# ----------交换任务----------
def swap_tasks(routes, route_index1, task_index1, route_index2,
task_index2):
    new_routes = copy.deepcopy(routes)

    # 确保 route_index1 和 route_index2 有效
    if route_index1 >= len(new_routes) or route_index2 >= len(new_routes):
        raise IndexError("Route index out of range.")

    # 确保 task_index1 和 task_index2 在各自路线的有效范围内
    if task_index1 >= len(new_routes[route_index1]) or task_index2 >=
len(new_routes[route_index2]):
        raise IndexError("Task index out of range.")

    task1 = new_routes[route_index1].pop(task_index1)
    task2 = new_routes[route_index2].pop(task_index2)

    new_routes[route_index1].insert(task_index1, task2)
    new_routes[route_index2].insert(task_index2, task1)

    return new_routes


# ----------计算时间均衡性----------
def calculate_time_balance(routes, time_matrix):
    times = [calculate_route_time(route, time_matrix) for route in routes]
    average_time = sum(times) / len(times)
    return sum((time - average_time) ** 2 for time in times) / len(times)


# ----------Pareto 最优检查----------
def is_pareto_optimal(candidate, solutions):
    for solution in solutions:
        if all(candidate[i] >= solution[i] for i in range(len(candidate)))
and any(
                candidate[i] > solution[i] for i in range(len(candidate))):
            return False
    return True
```

```
# ----------更新 Pareto 前沿----------
def update_pareto_frontier(frontier, candidate, time_matrix):
    candidate_solution = Solution(candidate, time_matrix)
    key = (candidate_solution.cost, candidate_solution.balance)
    if key not in frontier or candidate_solution < frontier[key]:
        frontier[key] = candidate_solution


# ----------计算目标函数----------
def calculate_objective(routes, time_matrix, weight_cost=0.1,
weight_balance=0.9):
    cost = calculate_total_cost(routes, time_matrix)
    balance = calculate_time_balance(routes, time_matrix)
    return weight_cost * cost + weight_balance * balance


# ----------计算路线时间----------
def balance_routes(routes, time_matrix, threshold):
    # 计算当前均衡性
    times = [calculate_route_time(route, time_matrix) for route in routes]
    average_time = sum(times) / len(times)

    # 当存在明显的不平衡
    while max(times) - min(times) > threshold:
        # 找到最长和最短的路线
        longest_route_index = times.index(max(times))
        shortest_route_index = times.index(min(times))

        # 将任务从最长路径移动到最短路径
        for i, task in enumerate(routes[longest_route_index]):
            new_routes = copy.deepcopy(routes)
            new_routes[longest_route_index].remove(task)
            new_routes[shortest_route_index].append(task)
            if is_valid_solution(new_routes, time_matrix):
                routes = new_routes
                times = [calculate_route_time(route, time_matrix) for route
in routes]
                break

    return routes


"""
    ######################
```

```
    Part Three:主程序
    ######################
"""
if __name__ == '__main__':
    # ----------评判问题二的数据是否均衡----------
    # 计算 Gini 系数
    df2 = pd.read_excel("题 2 时间.xlsx")
    t2_time = df2['工作时长/min'].values.tolist()
    gini = inequality.gini._gini(t2_time)
    print("问题二数据 Gini 系数为：", gini)
    Standard_deviation_test(np.array(t2_time))

    # 绘制洛伦兹曲线
    cumulative_days_proportion, cumulative_work_time_proportion =
calculate_lorenz_curve(t2_time)
    plot_lorenz_curve(cumulative_days_proportion,
cumulative_work_time_proportion, len(t2_time))

    # ----------导入数据，建立必需的变量----------
    df = pd.read_excel("附件 1：xx 地区.xlsx")
    points_origin = df[['JD', 'WD']].values.tolist()
    point_index = df['序号'].values.tolist()
    speed = 20
    dwell_time = df['完成工作所需时间（分钟）'].values.tolist()
    graph = []
    pareto_frontier = {}
    points = df[['JD', 'WD', '完成工作所需时间（分钟）']].values.tolist()
    max_work_time = 8.5 * 60  # 最大工作时间，单位：分钟
    n_tasks = len(points)
    timedata = []

    #----------计算时间矩阵----------
    for i in range(len(points_origin)):
        graph.append([])
        for j in range(len(points_origin)):
            graph[i].append(geodesic((points_origin[i][1],
points_origin[i][0]),
                                     (points_origin[j][1],
points_origin[j][0])).kilometers / speed * 60)
    #print(graph)

    # ----------调用算法，解决问题----------
    initial_routes = greedy_initial_solution(graph)
    optimized_routes = local_search(initial_routes, graph)
```

```
    final_routes = tabu_search(optimized_routes, graph)

    # ----------输出初步结果----------
    for i, route in enumerate(final_routes):
        route_time = calculate_route_time(route, graph)
        print(f"Route {i + 1}: Tasks {route}, Total Time: {route_time}
minutes")

    # ----------平衡路线----------
    final_routes = balance_routes(final_routes, graph, 50)
    final_routes_solution = Solution(final_routes, graph)
    update_pareto_frontier(pareto_frontier, final_routes, graph)

    # ----------输出 Pareto 前沿上的解决方案----------
    for (cost, balance), solution in pareto_frontier.items():
        routes = solution.routes
        # 使用 cost, balance 和 routes 进行进一步的处理
        print(f"Cost: {cost}, Balance: {balance}, Routes: {routes}")

    # ----------输出最终结果----------
    for i, route in enumerate(final_routes):
        route_time = calculate_route_time(route, graph)
        timedata.append([f"第 { i + 1 } 天", route_time])
        print(f"Route {i + 1}: Tasks {route}, Total Time: {route_time}
minutes")

    # ----------保存最终时间结果----------
    df3 = pd.DataFrame(timedata, columns=["日期", "工作时长/min"])
    df3.to_excel("题 3 时间.xlsx", index=False)

    # ----------评判问题三的数据是否均衡----------
    t3_time = df3['工作时长/min'].values.tolist()
    gini3 = inequality.gini._gini(t3_time)
    print("问题三数据 Gini 系数为：", gini3)
    Standard_deviation_test(np.array(t3_time))

    # 绘制洛伦兹曲线
    cumulative_days_proportion2, cumulative_work_time_proportion2 =
calculate_lorenz_curve(t3_time)
    plot_lorenz_curve(cumulative_days_proportion2,
cumulative_work_time_proportion2, len(t3_time))
```

代码 5：Q4.GD_TX_API-FW-DFS.py

```
import pandas as pd
import requests
import xml.etree.ElementTree as ET
import json


# ----------高德/腾讯地图 API 密钥----------
GDKEY = "39baae6eae5068c851ac1048a7b1d8e2"
TXKEY = "UARBZ-QEWWB-VBKUH-NGJEJ-NJVUQ-VQBKO"

# ----------通过高德地图 API 获取距离、时长----------
def gd_getDistance_and_time(start, end):
    url =
f"https://restapi.amap.com/v3/direction/driving?origin={start}&destinatio
n={end}&extensions=all&output=xml&key={GDKEY}"
    res = requests.get(url)
    root = ET.fromstring(res.text)
    if root.find('.//status').text == "1":
        return (float(root.find('.//distance').text)/ 1000,
round(float(root.find('.//duration').text)/60,1))
    else:
        print(root.find('.//info').text)
        return -1

# ----------通过腾讯地图 API 获取距离、时长----------
def tx_getDistance_and_time(start, end):
    url =
f"https://apis.map.qq.com/ws/direction/v1/driving/?from={start}&to={end}&
waypoints=&output=json&callback=cb&key={TXKEY}"
    res = requests.get(url)
    json_data = json.loads(res.text)
    if json_data["status"] == 0:
        return (float(json_data["result"]["routes"][0]["distance"])/1000,
float(json_data["result"]["routes"][0]["duration"]))
    else:
        print(json_data["message"])
        return -1

# ----------Floyd-Warshall 算法实现，返回所有点对之间的最短路径长度
----------
def floyd_warshall(graph):
    num_vertices = len(graph)
    dist = [[graph[i][j] for j in range(num_vertices)] for i in
```

```
range(num_vertices)]
    for k in range(num_vertices):
        for i in range(num_vertices):
            for j in range(num_vertices):
                if dist[i][j] > dist[i][k] + dist[k][j]:
                    dist[i][j] = dist[i][k] + dist[k][j]
    return dist

# ----------使用回溯策略来构建一个访问所有点的路径，并找到最短时间
----------
def find_path(dist, num_vertices):
    visited = [False] * num_vertices
    path = []
    min_time = float('infinity')
    def dfs(node, current_path, current_time):
        nonlocal min_time, path
        visited[node] = True
        current_path.append(node)
        if len(current_path) == num_vertices:
            if current_time < min_time:
                min_time = current_time
                path = current_path[:]
        else:
            for next_node in range(num_vertices):
                if not visited[next_node]:
                    dfs(next_node, current_path, current_time +
dist[node][next_node])
        visited[node] = False
        current_path.pop()

    # 从每个节点开始尝试
    for start_node in range(num_vertices):
        dfs(start_node, [], 0)
    return min_time, path

# ----------读取数据----------
df=pd.read_excel("附件 1：xx 地区.xlsx")
points_index=[31,44,61,83,100,115,147,158]
points_detail=[]
dwell_time=[0]*len(points_index)
gd_distanse_graph=[]
gd_time_graph=[]
tx_distanse_graph=[]
tx_time_graph=[]
```

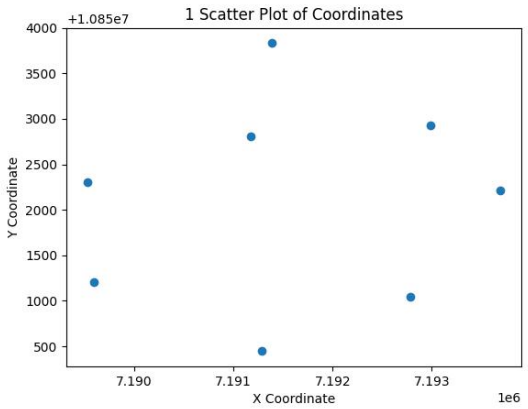
```
#----------提取所需点位----------
for i in range(len(points_index)):
    points_detail.append(list(df.iloc[points_index[i]-1]))
    dwell_time[i]=points_detail[i][3]

#----------计算时间矩阵----------
for i in range(len(points_detail)):
    gd_start=str(points_detail[i][1])+","+str(points_detail[i][2])  #gd
    tx_start = str(points_detail[i][2]) + "," + str(points_detail[i][1])
#tx
    gd_distanse_graph.append([])
    gd_time_graph.append([])
    tx_distanse_graph.append([])
    tx_time_graph.append([])
    for j in range(len(points_detail)):
        gd_end=str(points_detail[j][1])+","+str(points_detail[j][2]) #gd
        tx_end = str(points_detail[j][2]) + "," + str(points_detail[j][1])
#tx
        if i==j:
            gd_distanse_graph[i].append(0)
            gd_time_graph[i].append(0)
            tx_distanse_graph[i].append(0)
            tx_time_graph[i].append(0)
        else:
            gd_results=gd_getDistance_and_time(gd_start,gd_end)
            tx_results=tx_getDistance_and_time(tx_start,tx_end)
            gd_distanse_graph[i].append(gd_results[0])
            gd_time_graph[i].append(gd_results[1])
            tx_distanse_graph[i].append(tx_results[0])
            tx_time_graph[i].append(tx_results[1])
#----------输出行车结果----------
print(gd_distanse_graph)
print(gd_time_graph)
print(tx_distanse_graph)
print(tx_time_graph)

gd_num_vertices = len(gd_time_graph)
tx_num_vertices = len(tx_time_graph)

# ----------计算所有点对之间的最短路径----------
gd_dist = floyd_warshall(gd_time_graph)
tx_dist = floyd_warshall(tx_time_graph)
```

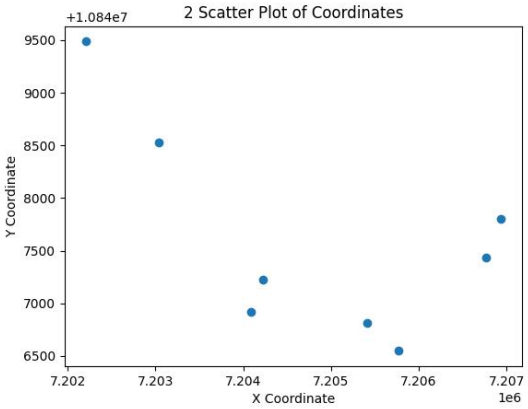
```
# ----------找到一个访问所有点的最短单向路径及其时间----------
gd_min_time, gd_path = find_path(gd_dist, gd_num_vertices)
tx_min_time, tx_path = find_path(tx_dist, tx_num_vertices)
#print(gd_min_time,tx_min_time)
# ----------比较两种地图的导航结果，输出最终结果----------
if (gd_min_time<tx_min_time):

print("-----------------------------------------------------------------------
----------------")
    print("Enable Amap Navigation!!!") #高德
    print("The shortest time:", gd_min_time+sum(dwell_time))
    print("The optimal path is",end=":")
    for i in range(len(gd_path)-1):
        print(points_index[gd_path[i]],end=" -> ")
    print(points_index[gd_path[-1]])
else:

print("-----------------------------------------------------------------------
----------------")
    print("Enable Tencent Maps Navigation!!!")#腾讯
    print("The shortest time:", tx_min_time+sum(dwell_time))
    print("The optimal path is",end=":")
    for i in range(len(tx_path)-1):
        print(points_index[tx_path[i]],end=" -> ")
    print(points_index[tx_path[-1]])
```

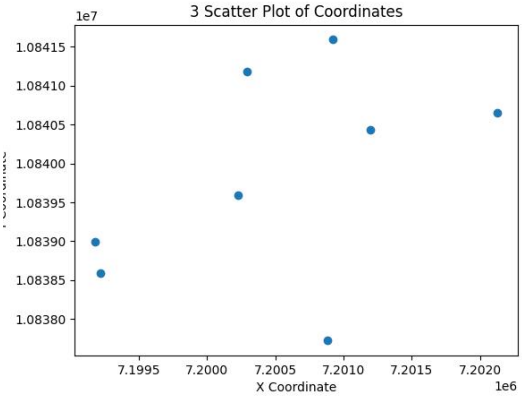
代码 6：Q4_run_script

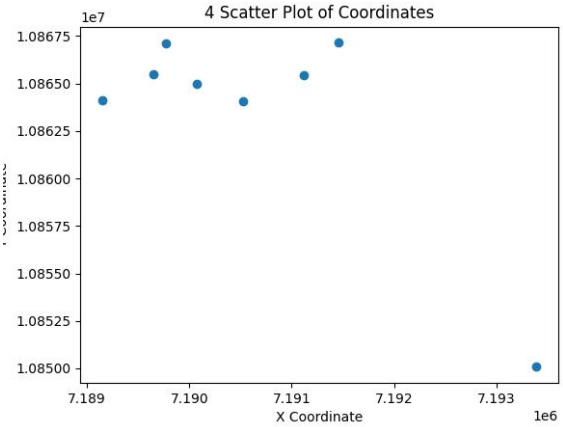
```
@echo off
:loop
python Q4.GD_TX_API-FW-DFS.py >> output.txt
timeout /t 1800
goto loop
```

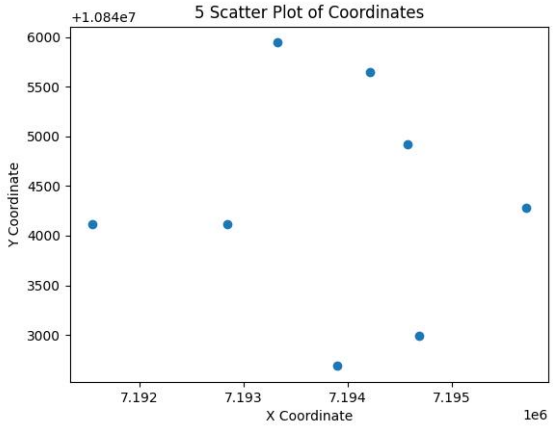
附录三 图片

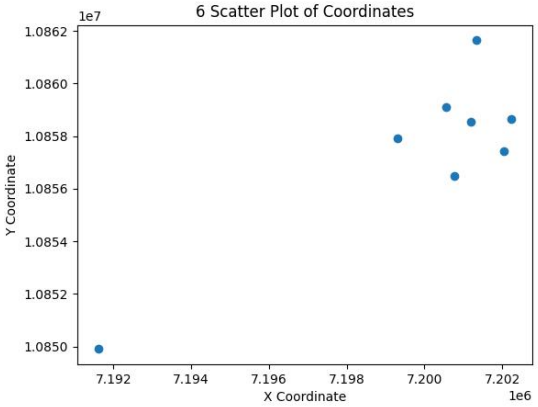

1 Scatter Plot of Coordinates
+1.085e7
4000
3500
3000
2500
2000
1500
1000
500
Y Coordinate
7.190
7.191
7.192
7.193
1e6
X Coordinate

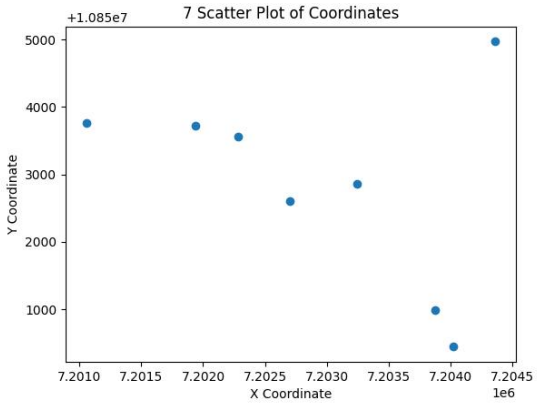

2 Scatter Plot of Coordinates
+1.084e7
9500
9000
8500
8000
7500
7000
6500
Y Coordinate
7.202
7.203
7.204
7.205
7.206
7.207
1e6
X Coordinate

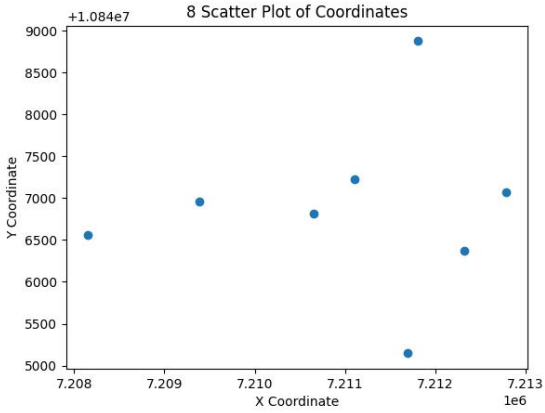

3 Scatter Plot of Coordinates
1e7
1.08415
1.08410
1.08405
1.08400
1.08395
1.08390
1.08385
1.08380
7.1995
7.2000
7.2005
7.2010
7.2015
7.2020
1e6
X Coordinate

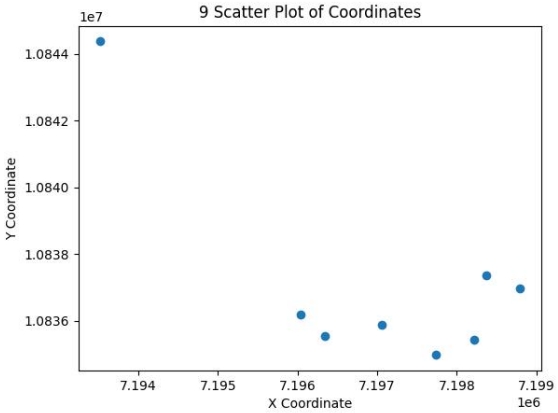

4 Scatter Plot of Coordinates
1e7
1.08675
1.08650
1.08625
1.08600
1.08575
1.08550
1.08525
1.08500
7.189
7.190
7.191
7.192
7.193
1e6
X Coordinate

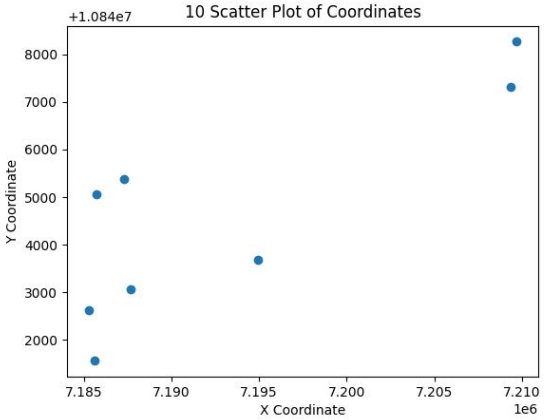

5 Scatter Plot of Coordinates
+1.084e7
6000
5500
5000
4500
4000
3500
3000
Y Coordinate
7.192
7.193
7.194
7.195
1e6
X Coordinate

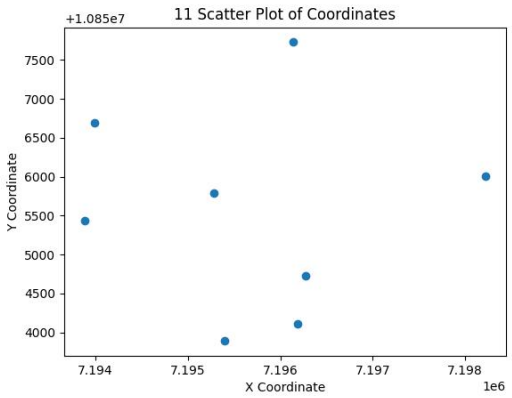

6 Scatter Plot of Coordinates
1e7
1.0862
1.0860
1.0858
1.0856
1.0854
1.0852
1.0850
Y Coordinate
7.192
7.194
7.196
7.198
7.200
7.202
1e6
X Coordinate

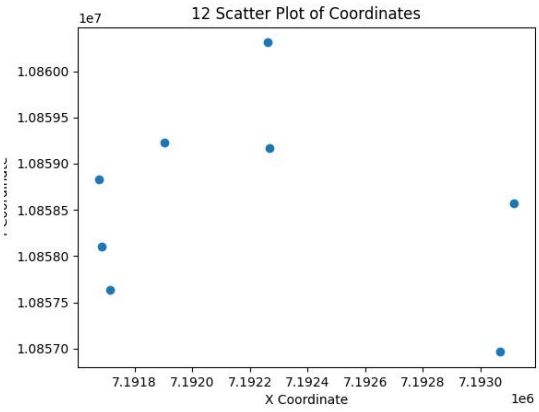

7 Scatter Plot of Coordinates
+1.085e7
Y Coordinate
X Coordinate
1e6

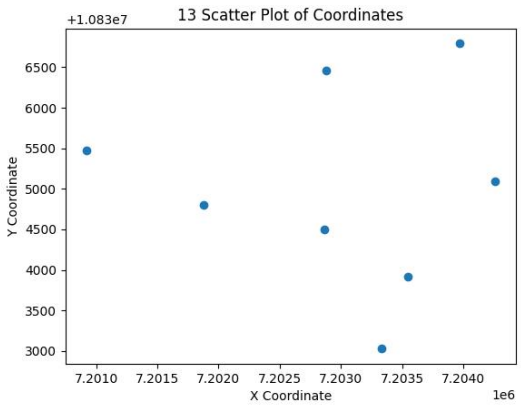

8 Scatter Plot of Coordinates
+1.084e7
Y Coordinate
X Coordinate
1e6

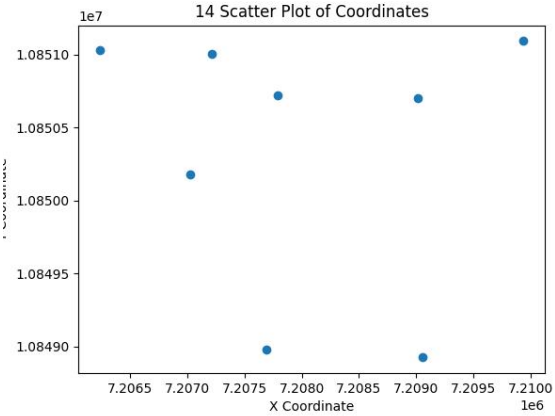

9 Scatter Plot of Coordinates
1e7
Y Coordinate
X Coordinate
1e6

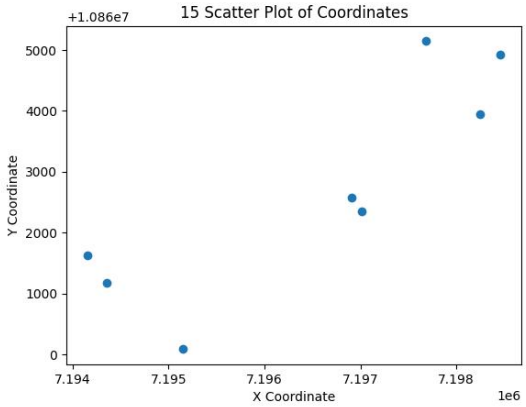

10 Scatter Plot of Coordinates
+1.084e7
Y Coordinate
X Coordinate
1e6

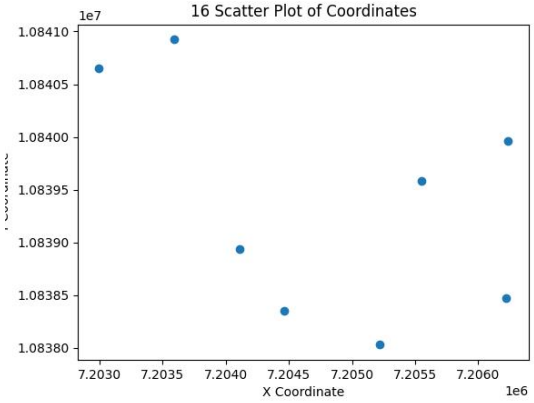

11 Scatter Plot of Coordinates
+1.085e7
Y Coordinate
X Coordinate
1e6

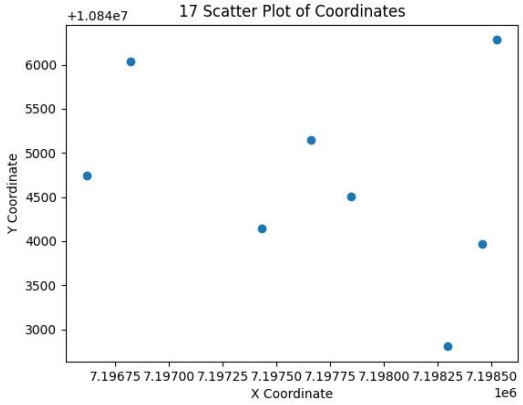

12 Scatter Plot of Coordinates
1e7
X Coordinate
1e6

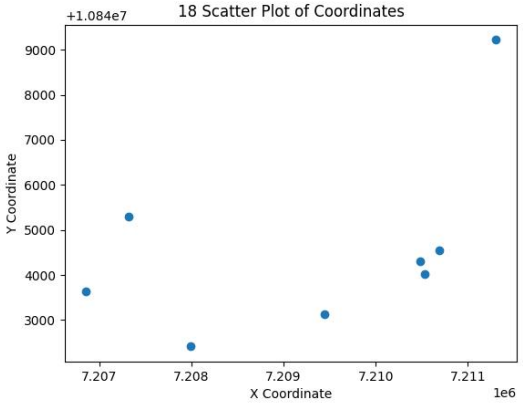

13 Scatter Plot of Coordinates
+1.083e7
Y Coordinate
X Coordinate
1e6

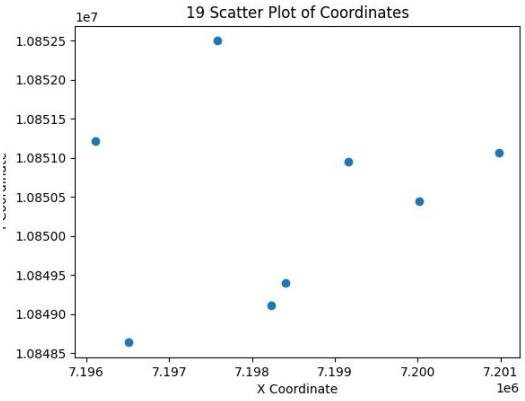

14 Scatter Plot of Coordinates
1e7
X Coordinate
1e6

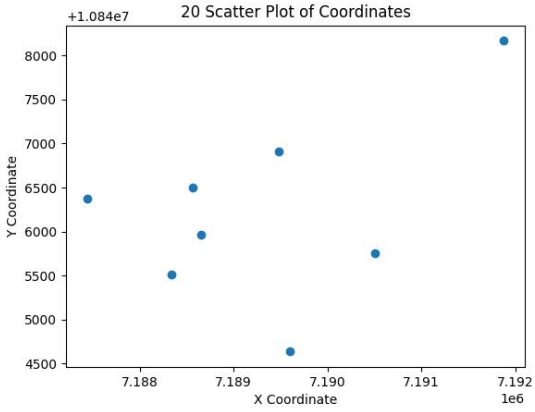

15 Scatter Plot of Coordinates
+1.086e7
Y Coordinate
X Coordinate
1e6

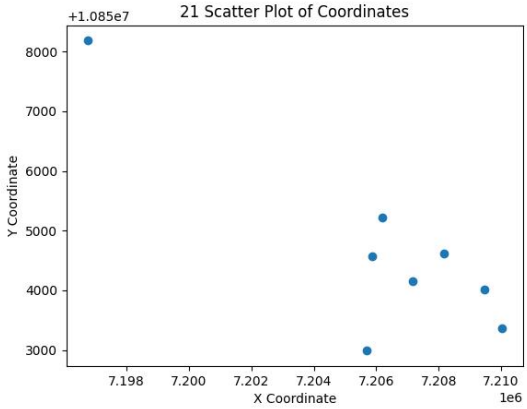

16 Scatter Plot of Coordinates
1e7
X Coordinate
1e6

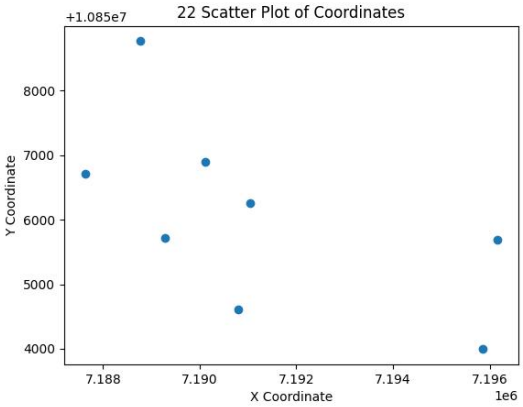

17 Scatter Plot of Coordinates
+1.084e7
Y Coordinate
X Coordinate
1e6

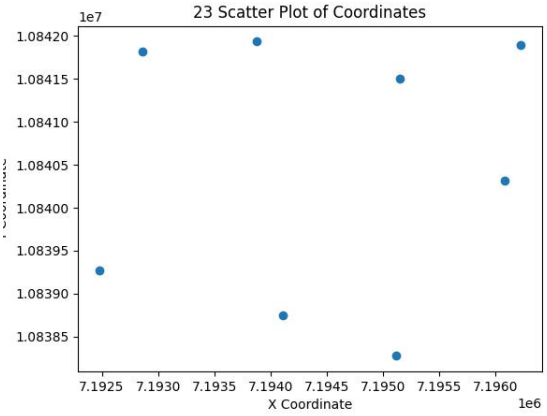

18 Scatter Plot of Coordinates
+1.084e7
Y Coordinate
X Coordinate
1e6

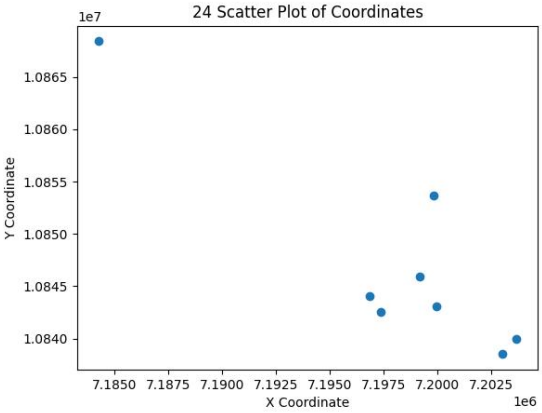

19 Scatter Plot of Coordinates
1e7
X Coordinate
1e6

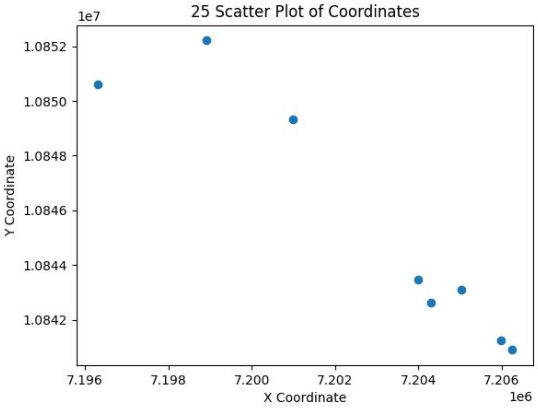

20 Scatter Plot of Coordinates
+1.084e7
Y Coordinate
X Coordinate
1e6

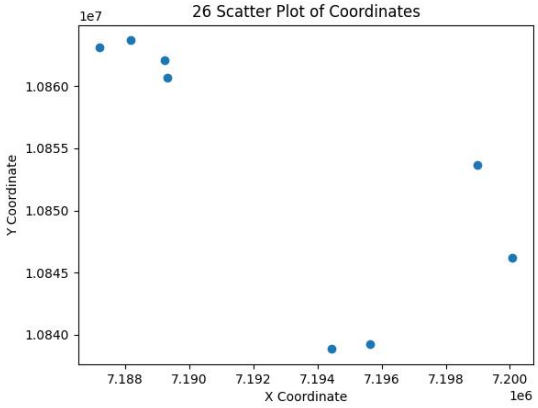

21 Scatter Plot of Coordinates
+1.085e7
Y Coordinate
X Coordinate
1e6

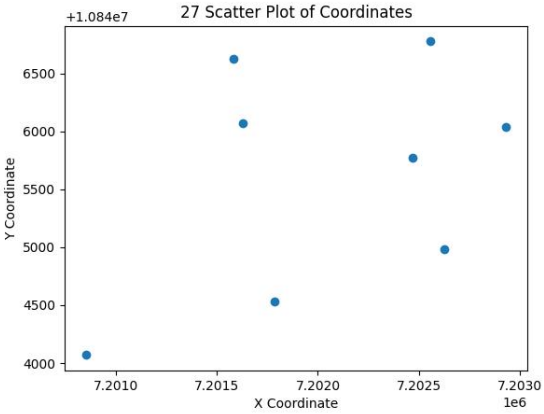

22 Scatter Plot of Coordinates
+1.085e7
Y Coordinate
X Coordinate
1e6

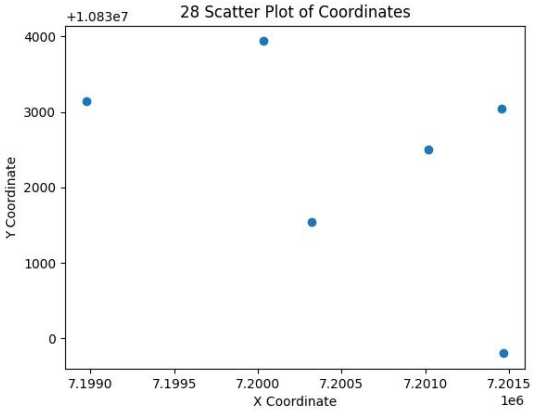












附录四 表格

题 2 时间.xlsx

日期	工作时长/min
第 1 天	407.4170378
第 2 天	394.8883003
第 3 天	419.9862535
第 4 天	442.0714054
第 5 天	401.1866983
第 6 天	448.5523735
第 7 天	421.7530658

第 8 天	409.3453448
第 9 天	414.3415235
第 10 天	445.6728092
第 11 天	417.0665612
第 12 天	407.8053993
第 13 天	421.6897822
第 14 天	399.5338141
第 15 天	415.421369
第 16 天	408.8597763
第 17 天	409.0841704
第 18 天	427.9951717
第 19 天	403.8190881
第 20 天	436.778242
第 21 天	443.8190386
第 22 天	417.8038332
第 23 天	389.1057205
第 24 天	474.5002759
第 25 天	435.8294941
第 26 天	465.8091795
第 27 天	415.7196631
第 28 天	327.1029846

题 3 时间.xlsx

日期	工作时长/min
第 1 天	479.7075739
第 2 天	469.718539
第 3 天	472.4060621
第 4 天	477.0665739
第 5 天	474.274913
第 6 天	483.7492323
第 7 天	447.0962305
第 8 天	454.0171548
第 9 天	472.8354763
第 10 天	477.6457462
第 11 天	469.3831052
第 12 天	474.2774384
第 13 天	483.244774
第 14 天	476.016786
第 15 天	452.7374123
第 16 天	471.7669238
第 17 天	490.2683428
第 18 天	478.9108842
第 19 天	488.737309

第 20 天	477.0802028
第 21 天	481.487291
第 22 天	470.1502653
第 23 天	484.905018
第 24 天	479.4668882
第 25 天	447.7497349

Q4.xlsx

time line	the type of map	the shortest time /min	The optimal path
0:30	Amap	412.3	61 -> 44 -> 115 -> 31 -> 147 -> 100 -> 158 -> 83
1:00	Amap	412.2	83 -> 158 -> 100 -> 147 -> 31 -> 115 -> 61 -> 44
1:30	Tencent Maps	412.0	44 -> 61 -> 115 -> 31 -> 147 -> 100 -> 83 -> 158
2:00	Tencent Maps	412.0	44 -> 61 -> 115 -> 31 -> 147 -> 100 -> 83 -> 158
2:30	Tencent Maps	412.0	44 -> 61 -> 115 -> 31 -> 147 -> 100 -> 83 -> 158
3:00	Tencent Maps	412.0	44 -> 61 -> 115 -> 31 -> 147 -> 100 -> 83 -> 158
3:30	Tencent Maps	412.0	44 -> 61 -> 115 -> 31 -> 147 -> 100 -> 83 -> 158
4:00	Amap	411.9	61 -> 44 -> 115 -> 31 -> 147 -> 100 -> 158 -> 83
4:30	Amap	412.3	83 -> 158 -> 100 -> 147 -> 31 -> 115 -> 44 -> 61
5:00	Amap	412.3	44 -> 61 -> 115 -> 31 -> 147 -> 100 -> 158 -> 83
5:30	Amap	412.1	83 -> 158 -> 100 -> 147 -> 31 -> 115 -> 44 -> 61
6:00	Amap	412.3	44 -> 61 -> 115 -> 31 -> 147 -> 100 -> 158 -> 83
6:30	Amap	412.3	83 -> 158 -> 100 -> 147 -> 31 -> 115 -> 44 -> 61
7:00	Amap	412.2	83 -> 158 -> 100 -> 147 -> 31 -> 115 -> 44 -> 61
7:30	Amap	412.0	83 -> 158 -> 100 -> 147 -> 31 -> 115 -> 44 -> 61
8:00	Amap	412.3	83 -> 158 -> 100 -> 147 -> 31 -> 115 -> 44 -> 61
8:30	Amap	410.3	83 -> 158 -> 100 -> 147 -> 31 -> 115 -> 44 -> 61
9:00	Amap	410.9	83 -> 158 -> 100 -> 147 -> 31 -> 115 -> 44 -> 61
9:30	Amap	411.1	83 -> 158 -> 100 -> 147 -> 31 -> 115 -> 62 -> 44
10:00	Amap	410.4	83 -> 158 -> 100 -> 147 -> 31 -> 115 -> 44 -> 61
10:30	Amap	411.4	83 -> 158 -> 100 -> 147 -> 31 -> 115 -> 61 -> 44
11:00	Amap	410.8	83 -> 158 -> 100 -> 147 -> 31 -> 115 -> 61 -> 44
11:30	Amap	411.0	83 -> 158 -> 100 -> 147 -> 31 -> 115 -> 44 -> 61
12:00	Amap	411.1	83 -> 158 -> 100 -> 147 -> 31 -> 115 -> 61 -> 44
12:30	Amap	411.0	158 -> 83 -> 100 -> 147 -> 31 -> 115 -> 61 -> 44
13:00	Amap	410.9	83 -> 158 -> 100 -> 147 -> 31 -> 115 -> 61 -> 44
13:30	Amap	410.5	83 -> 158 -> 100 -> 147 -> 31 -> 115 -> 61 -> 44
14:00	Amap	411.0	83 -> 158 -> 100 -> 147 -> 31 -> 115 -> 44 -> 61
14:30	Amap	410.9	83 -> 158 -> 100 -> 147 -> 31 -> 115 -> 44 -> 61
15:00	Amap	410.8	83 -> 158 -> 100 -> 147 -> 31 -> 115 -> 44 -> 61
15:30	Amap	410.5	83 -> 158 -> 100 -> 147 -> 31 -> 115 -> 44 -> 61