老外游中国

摘　要

“city 不 city”这一网络流行语在外国网红的推广下引发了广泛关注。随着中国实施过境免签政策，越来越多的外国游客通过网络分享他们在中国的旅行经历。这不仅推动了旅游业的发展，也在国际舞台上生动展示了中国的真实面貌。

针对问题一，经过初步分析，各景点的评分系统最高为 **5 分（BS）**。对附件数据进行了整合，提取了评分为 5 分的景点。然而，校验发现这些数据与实际情况不符。因此，使用 **Selenium** 和 **Xpath** 进行了数据爬取，并对数据进行了检验和修正。修正后的数据显示，全国共有 **1722** 个景点获得了 **BS**。根据统计结果，获得最高评分的前 10 个城市依次为：**烟台、泸州、自贡、玉溪、内江、昭通、益阳、宁德、抚州和德州**。

针对问题二，为对 352 个城市进行综合评价，首先通过 **Requests** 库爬取并收集了相关数据，涵盖**总人口、面积、城市 AQI、公共汽车数量、出租汽车数量和平均气温**等指标。对“小贴士”中的文本进行了**分词和词频统计**，依据词频统计结果评估环境环保、人文底蕴、气候和美食的评分。对于无法获取的数据，采用均值进行替代。接着，利用 **TOPSIS** 方法对各城市进行综合评分，并在文中列出了前 50 个最受外国游客向往的城市。

针对问题三，为在 144 小时内游玩尽可能多的城市，从问题二中排名前 50 的城市中挑选部分城市。通过铁路 12306 获取了两两城市之间的高铁行驶时间与票价，并对数据进行向上取整用以缓冲。将问题转化为 **TSP（旅行商问题）**，首先使用**贪婪算法**确定了从广州出发游览所有城市的最短时间的路径。随后，根据 144 小时的限制，合理制定了每日的游览计划，最终结果为从入境到出境所用时间约为 134 小时，共 **11** 个城市，乘坐高铁费用为 **2570** 元，游览景点费用为 **919** 元，共计 **3489** 元。

针对问题四，在问题三的基础上，考虑门票费用与乘坐高铁费用，尽可能游览更多的城市，采用**遗传算法**以门票费用与乘坐高铁费用为目标进行优化。随后，根据 144 小时的限制，合理制定了每日的游览计划，最终结果为从入境到出境所用时间约为 **134** 小时，共 **11** 个城市，乘坐高铁费用为 **2260** 元，游览景点费用为 **389** 元，共计 **2649** 元。

针对问题五，向山走去，直接选取中国山景中评价最高前 **20** 座山，其中考虑高铁的可达性，最终选取了 12 座山。结合三、四问中 144 小时内旅游的城市并结合爬山运动的特殊性，故模拟游览 9、10、11 和 12 个城市的路径，并使用**加权路径优化算法**获取最优路径，并在 144 小时内合理规划每日行程，最终结果为游览 **10** 个城市，高铁票与门票共花费 **2510 元**，路线为**温州-黄山-南平-九江-萍乡-衡阳-郑州-洛阳-宝鸡-西安-成都**。

由于问题中对许多因素进行了简化，因此在实际制定旅行计划时，需综合考虑多个因素。例如，旅行计划应考虑交通状况、天气变化、景点最佳游览时间、游客偏好、预算限制等实际情况。这样才能确保制定的计划既切实可行，又能最大程度地满足个人需求和期望。

关键词：Selenium　Xpath　Requests　TOPSIS　TSP　贪婪算法　遗传算法

目录

一、 问题重述

近期，“city 不 city”这一网络流行语在外国网红的推动下引起了广泛关注。随着我国过境免签政策的实施，越来越多的外国游客来到中国，通过网络平台分享他们在中国旅行的经历。这不仅促进了中国旅游业的发展，也在国际舞台上展示了一个真实而生动的中国，取得了多重效果。

图 1 免签 144 小时图

假设外国游客入境后能在中国境内逗留 144 小时，且能够从任一城市附近的机场出境。为了使游客能游览更多城市，遵循“每个城市只选择一个评分最高的景点游玩”的原则。现有包含中国（不含港澳台）352 个城市的旅游景点数据集，每个城市的 CSV 文件中包含 100 个景点的信息，包括景点名称、网址、地址、介绍、开放时间、图片网址、景点评分、建议游玩时长、建议游玩季节、门票信息、小贴士等。建立数学模型回答以下问题：

问题 1

352 个城市中所有 35200 个景点评分的最高分（Best Score，简称 BS）是多少？全国有多少个景点获得了这个最高评分（BS）？获评最高评分（BS）景点最多的城市有哪些？请依据这些城市的最高评分（BS）景点数量的多少进行排序，并列出前 10 个城市。

问题 2

假如外国游客遵循“城市最佳景点游览原则”，结合城市规模、环境环保、人文底蕴、交通便利、气候、美食等因素，对 352 个城市进行综合评价。从中选出“最令外国游客向往的 50 个城市”。

问题 3

一名外国游客从广州入境，他希望在 144 小时内游玩尽可能多的城市，并要求综合游玩体验最好。规划他的游玩路线。需要考虑以下要求：

1. 遵循“城市最佳景点游览原则”。
2. 城市之间的交通方式只选择高铁。
3. 只在“最令外国游客向往的 50 个城市”中选择要游玩的城市。

提供具体的游玩路线，包括总花费时间、门票和交通的总费用，以及可以游玩的

景点数量。

问题 4

如果将问题 3 的游览目标改为：既要尽可能多地游览城市，又需要使门票和交通的总费用尽可能少，请重新规划游玩路线。提供包括总费用、总花费时间以及可以游玩的城市数量的具体游玩路线。

问题 5

一名外国游客只想游览中国的山景，他乘飞机入境中国的城市不限。为他选择入境的机场和城市，并个性化定制他的 144 小时旅游路线。需要满足以下要求：

1. 每个城市只游玩一座评分最高的山。

2. 城市之间的交通方式只选择高铁。

3. 旅游城市不局限于“最令外国游客向往的 50 个城市”，游览范围拓展到 352 个城市。

提供具体的游玩路线，包括总花费时间、门票和交通的总费用，以及可以游玩的山景数量。

二、 问题分析

2.1 问题一的分析

从 35200 个景点评分数据中找出最高评分（BS），并统计全国获得该评分的景点数量。接着，需分析每个城市中获得 BS 评分的景点数量，最终列出前 10 个拥有最多 BS 评分景点的城市，并按数量排序。

2.2 问题二的分析

要求综合评估 352 个城市的各项因素，包括城市规模、环境环保、人文底蕴、交通便利、气候和美食等，选出最吸引外国游客的 50 个城市。首先需要收集这些城市的相关数据，然后设定评价权重，使用综合评分方法进行排序，最终确定前 50 个最向往的城市。

2.3 问题三的分析

涉及在 144 小时内规划一条游玩尽可能多城市的路线，并优化体验。需要从“最令外国游客向往的 50 个城市”中筛选出合适的城市，利用高铁作为交通工具，合理规划游玩路线，计算总花费时间和费用，并确保游客能尽可能多地游览景点。

2.4 问题四的分析

为了在 144 小时内尽可能多游览城市，同时最小化门票和交通费用，需要从“最令外国游客向往的 50 个城市”中筛选城市，制定优化路线。关键在于寻找最低成本的高铁票和景点门票，合理规划高效的游玩路线，计算总费用和时间，并确定最大可游玩的城市数量。

2.5 问题五的分析

该任务要求为游客定制 144 小时内的山景旅游路线，确保尽可能多的山景并控制费用。需要从 352 个城市中筛选评分最高的山景，选择适合的入境城市，规划每个城市游玩一座山的路线。使用高铁作为主要交通工具，并计算总花费时间和费用，确保能够游览最多的山景。

三、 模型假设

- 假设数据更新后无误；
- 假设采集的数据合理有效；
- 假设评分为 5 分的景点仅代表网页评分，而非实际评分；
- 假设数据已妥善预处理；
- 假设外国游客为成年人；
- 假设 144 小时内天气均为晴朗天气
- 假设外国游客第一天上午 8 点到达广州且状态良好。
- 假设外国游客能够从任一城市附近的机场出境；
- 假设外国游客游玩两个景点之间需要休息；
- 假设外国游客游玩时间为夏季；
- 假设费用只考虑高铁和景点门票费用；
- 假设优先级时间 > 金钱，城市 > 景点；
- 假设每日只在上午、下午和晚上进行游玩，每晚在旅馆休息；
- 假设高铁时间与票价进行向上取整；
- 假设缓冲时间为 1 个小时；
- 假设游览山景的时间在 4-5 小时；
- 假设每日需要充分休息；
- 假设景点从实际评价中选取；
- 假设外国游客能遵守时间安排；
- 假设缓冲时间能应对实际情况；
- 假设旅行时无突发情况；
- 假设外国游客选择在凌晨爬山；
- 假设外国游客可以在高铁上休息；
- 假设每日行程安排合理。

四、 符号说明

符号	说明
x_{ij}	第 i 个城市的第 j 个指标值
v_{ij}	第 i 个城市的第 j 个指标标准化值
w_j	第 j 个指标权重
$t_{x_i,x_{i+1}}$	第 i 个城市到第 $i+1$ 个城市的高铁乘坐时间
t_{x_n,x_1}	返回到起点的高铁乘坐时间
n	城市的数量
P	乘坐高铁的总费用
T	乘坐高铁的总时间
$t_{i,i+1}$	城市 i 到城市 $i+1$ 的高铁时间
$p_{i,i+1}$	城市 i 到城市 $i+1$ 的高铁费用
m_i	第 i 个景点的门票费用

五、 模型的建立与求解

5.1 问题一的方法与求解

5.1.1 方法

Selenium

Selenium 是一个用于自动化浏览器的工具，它可以模拟用户与网页的交互。其主要包括以下几个方面：

1. 驱动程序：Selenium 通过特定的驱动程序（如 ChromeDriver、GeckoDriver）与浏览器进行通信，控制浏览器的行为。

2. WebDriver API：Selenium 提供了 WebDriver API，通过这些接口可以编写脚本来操控浏览器、查找元素、执行操作等。

3. 浏览器操作：脚本通过 WebDriver 向浏览器发送命令，如点击、输入、滚动等，模拟真实用户操作。

4. 等待机制：Selenium 支持显式和隐式等待，以应对页面加载和元素出现的延迟问题，确保脚本的稳定性。

将其结合在一起，使得 Selenium 能够实现自动化测试和网页数据抓取等功能。

Xpath

Xpath（XML Path Language）是一种用于在 XML 文档中查找和处理数据的语言。它将 XML 文档视为一个树形结构，其中每个元素、属性和文本都被视为节点。它使用路径语法来遍历这棵树，并通过各种条件筛选节点。XPath 支持使用轴和谓词来精确定位节点，从而实现对 XML 数据的高效查询和操作。它广泛用于网页自动化测试中，特别是在 Selenium 中，用于定位网页元素。

在使用 Selenium 进行自动化操作或数据抓取时，XPath 常用于定位网页上的元素。结合这两者，XPath 可以帮助准确地找到要操作的网页元素，而 Selenium 则执行实际的操作，例如点击、输入文本或提取数据。例如，使用 Selenium 和 XPath，可以通过编写简单的 Python 代码打开网页、定位元素并提取其文本内容，最后关闭浏览器。这种结合使得网页自动化和数据抓取变得高效和灵活。

5.1.2 问题求解

经初步分析，所有景点中的最高评分为 5 分（BS）。通过 Excel 进行检索，发现有 2563 个景点的评分为 5 分。然而，根据景点的 URL 发现这些评分存在误差。因此，使用 Selenium 与 Xpath 重新爬取了这 2563 个景点的评分数据，结果显示有 1772 个景点的评分确为 5 分。

此外，由于该平台网页上显示的默认评分为 5 分，这并非实际评分，因此在后续分析中，将对这些数据进行筛选使用。

随后，将相关数据在 Excel 中进行统计分析。分析结果显示，在 352 个城市的

35200 个景点中，最高评分为 5.0，全国共有 1772 个景点获得了这一最高评分。获评最高分的景点最多的城市包括烟台、泸州、自贡、玉溪等。以下是根据拥有最高评分（BS）景点数量排序的前 10 个城市。

表 1 城市最高评分景点数量排序表

城市	最高评分景点数量
烟台	18
泸州	14
自贡	13
玉溪	13
内江	13
昭通	12
益阳	12
宁德	12
抚州	12
德州	12

注：由于网页设计原因，该结果仅供参考。

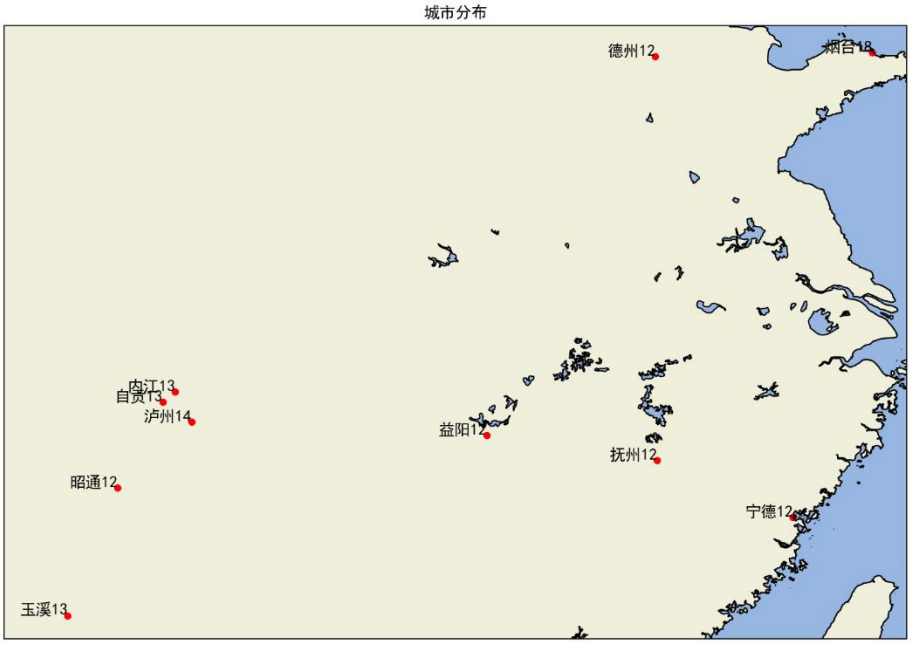

图 2 城市分布图

5.2 问题二方法与求解

5.2.1 方法

Requests

Requests 库的原理基于 Python 的 urllib 和 http.client 模块，但提供了更简洁和友好的 API。其主要流程如下：

1. 构建请求：用户通过方法（如 get、post）提供 URL 和参数，requests 构建 HTTP

请求。

2. 发送请求：库使用 urllib3 处理底层的连接和数据传输，管理连接池和处理重定向。

3. 处理响应：服务器响应数据被接收并解码，requests 解析响应内容、状态码和头信息，封装在响应对象中返回给用户。

4. 异常处理：requests 处理常见的网络错误，如连接超时、HTTP 错误，并提供友好的异常信息。

这一过程让 HTTP 请求变得简单易用，并自动处理许多复杂的底层细节。

TOPSIS

TOPSIS（Technique for Order of Preference by Similarity to Ideal Solution）是一种多准则决策方法。它的核心思想是通过比较各备选方案与理想解和负理想解的相似度来进行排序。具体步骤包括：

1. 标准化决策矩阵：将决策矩阵中的数据进行标准化，以消除不同量纲的影响。常见的标准化方法是向量标准化

2. 计算加权值：将标准化后的数据与准则权重相乘，以反映各准则的重要性。

3. 确定理想解和负理想解：

理想解（Ideal Solution）：每个准则中，选择最优的值。对于收益型准则（值越大越好），理想解是最大值；对于成本型准则（值越小越好），理想解是最小值。

负理想解（Negative Ideal Solution）：每个准则中，选择最差的值。对于收益型准则，负理想解是最小值；对于成本型准则，负理想解是最大值。

4. 计算距离：计算每个备选方案与理想解和负理想解的距离。

5. 计算相对接近度并排序。

5. 2. 2 问题求解

为对 352 个城市进行综合评价，需要结合城市规模、环境、环保、人文底蕴、交通便利，以及气候、美食等因素。

首先需要对相关数据进行收集和爬取：

城市规模：根据每个城市的总人口数、人口密度和面积来衡量，人口越多、面积越大，通常城市规模越大。

基于 Python 中的 Requests 库对 https://www.citypopulation.de/zh/china 进行爬取总人口、面积和人口密度三个指标。

环境环保：通过每个城市的 AQI 值（空气质量指数）来评估，AQI 值越低，表示空气质量越好，环境越环保。

基于 Python 中的 Requests 库对 https://www.air-level.com 进行爬取，统计 AQI 值。

交通便利：根据城市的公共汽车和出租汽车数量，来反映城市的交通设施和便利程度。

在 CSDN 网站上获取了《中国城市统计年鉴》面板数据整理（2000-2022 年）。

气候：通过平均气温来描述气候特征，反映城市的气温状况和气候类型。

在 CSDN 网站上获取了 368 个城市平均气温数据（2001-2022 年）。

美食：通过城市的饮食文化和特色美食来评估。

由于在第一问中发现景点的评分并非实际评分，故采取的方案是，通过 Python 爬虫获取了每个城市按照评价排名的前 10 个景点的“小贴士”，随后进行中文 jieba 分词并词频统计作为该项的评分。

城市评分过程如下：

1. 标准化决策矩阵：

$$\tilde{x}_{ij} = \frac{x_{ij}}{\sqrt{\sum_{i=1}^{n} x_{ij}^2}} \tag{1}$$

2. 标准化加权：

$$v_{ij} = w_j \times x_{ij} \tag{2}$$

设置 AQI 值、气候、平均气温、环境环保、人文底蕴、美食、公共汽车数量、出租汽车数量以及人口密度的权重均为 0.1，总人口与面积的权重为 0.05。

3. 确定理想解与负理想解

其中 AQI 设置为负理想解，平均温度为舒适的气温时为理想解，其余指标均为理想解。

4. 距离计算：

$$\begin{cases} D_i^+ = \sqrt{\sum \left(v_{ij} - \text{理想解} \right)^2} \\ D_i^- = \sqrt{\sum \left(v_{ij} - \text{负理想解} \right)^2} \end{cases} \tag{3}$$

5. 相对接近度计算：

$$C_i = \frac{D_i^-}{D_i^+ + D_i^-} \tag{4}$$

对计算结果进行排序，排名前 50 个城市如下表所示：

表 2 最令外国游客向往的 50 个城市排名表

排名	城市	排名	城市
1	上海	26	儋州
2	北京	27	南昌
3	深圳	28	承德
4	广州	29	百色
5	苏州	30	北海
6	成都	31	陵水
7	南京	32	秦皇岛
8	无锡	33	郑州
9	济南	34	南宁
10	重庆	35	东莞
11	西安	36	万宁
12	呼伦贝尔	37	河源
13	福州	38	江门
14	厦门	39	惠州
15	雅安	40	金华
16	丽水	41	揭阳
17	中山	42	长沙
18	武汉	43	宿州
19	抚州	44	玉林
20	佛山	45	宁波
21	杭州	46	哈尔滨
22	韶关	47	张家口
23	汕头	48	晋中
24	三亚	49	周口
25	珠海	50	嘉峪关

城市的经纬度采集 URL（http://www.jsons.cn/lngcode）

5.3 问题三模型的建立与求解

5.3.1 算法介绍

TSP（旅行商问题）

旅行商问题（TSP, Traveling Salesman Problem）是一个经典的优化问题，描述了一个旅行商需要访问多个城市并最终返回起点的任务。问题的目标是找到一个最短的路径，使得旅行商能访问每个城市恰好一次，并返回到起点城市。TSP 是一个 NP-hard 问题，这意味着随着城市数量的增加，找到最优解的计算复杂度会迅速增长。尽管如此，TSP 在物流、路线规划等实际应用中具有重要意义。

贪婪算法

贪婪算法（Greedy Algorithm）是一种在每一步选择中都采取当前最优的选择，从而希望能够达到全局最优解的算法设计策略。这种算法的核心思想是“局部最优”可以引导我们找到“全局最优”，即在每一步做出当前最好的选择，不必回溯。

为使得综合游玩体验最好，这里缩小城市范围，即仅选取最令外国游客向往的 50 个城市中排名靠前的部分城市进行建模规划。

选取的城市可视化如下：

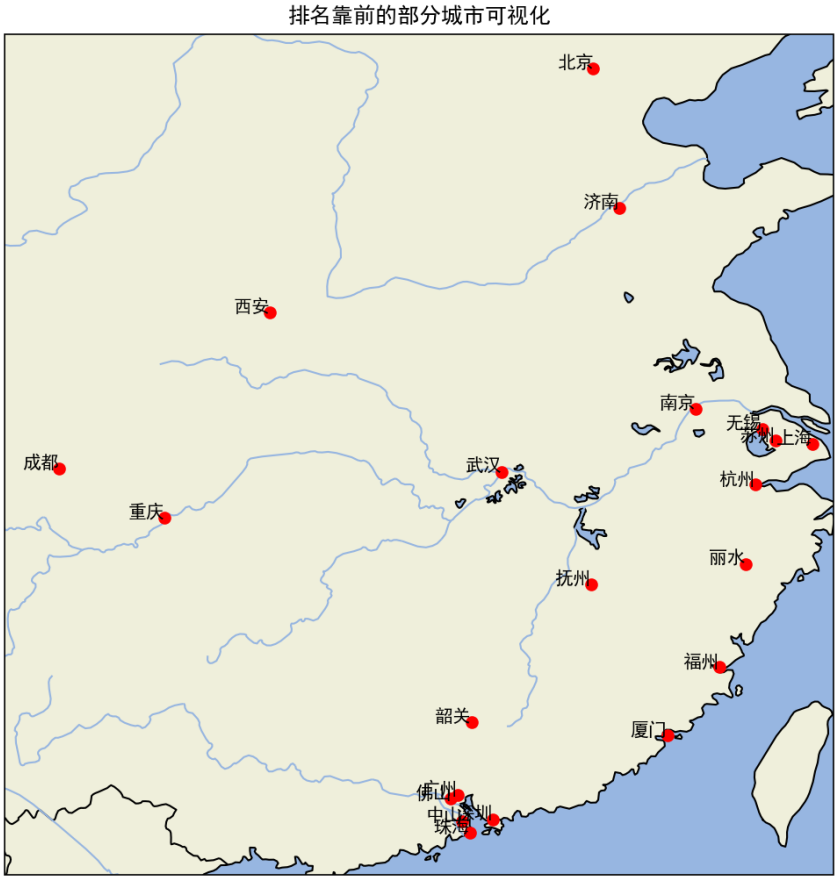

图 3 排名靠前部分城市图

在本问中定义优先级为时间 > 金钱，城市 > 景点，其中乘坐高铁所使用的时间和金钱均向上取整（用于缓冲），其中高铁乘坐时间去相对较快的列次。

通过铁路 12306APP 获取的两两城市之间的高铁时间及票价矩阵如下表所示：

表 3 城市之间的高铁时间及票价表 1

票价 时间	上海	北京	深圳	广州	苏州	成都	南京	无锡	…
上海	0	700,5	900,7	900,7	20,2	700,13	50,4	20,2	…
北京	700,5	0	1100,9	900,11	700,5	800,8	500,5	600,4	…
深圳	900,7	1100,9	0	80,2	900,10	800,8	800,8	900,9	…
广州	900,7	900,11	80,2	0	200,25	700,7	700,7	800,9	…
苏州	20,2	700,5	900,10	200,25	0	1200,13	40,3	11,1	…
成都	700,13	800,8	800,8	700,7	1200,13	0	900,11	700,13	…
南京	50,4	500,5	800,8	700,7	40,3	900,11	0	30,3	…
无锡	20,2	600,4	900,9	800,9	11,1	700,13	30,3	0	…
济南	500,6	200,5	1000,12	900,8	400,5	800,9	100,8	200,9	…

重庆	600,11	900,7	600,7	500,6	900,11	200,2	500,10	600,12	…
西安	700,8	600,6	900,10	800,8	700,7	300,4	600,6	800,9	…
…	…	…	…	…	…	…	…	…	…

使用贪婪算法求解旅行商问题的公式如下：

$$TotalTime = \sum_{i}^{n-1} t_{x_i, x_{i+1}} + t_{x_n, x_1} \tag{5}$$

其中 $t_{x_i, x_{i+1}}$ 表示两两城市的高铁乘坐时间，t_{x_n, x_1} 表示当前城市离起点城市的高铁乘坐时间。

这种贪婪算法的关键在于每一步都选择当前最优的局部解，即选择乘坐高铁时间最近的城市，这样每一步的选择并不全依赖于全局信息。

结果如下：

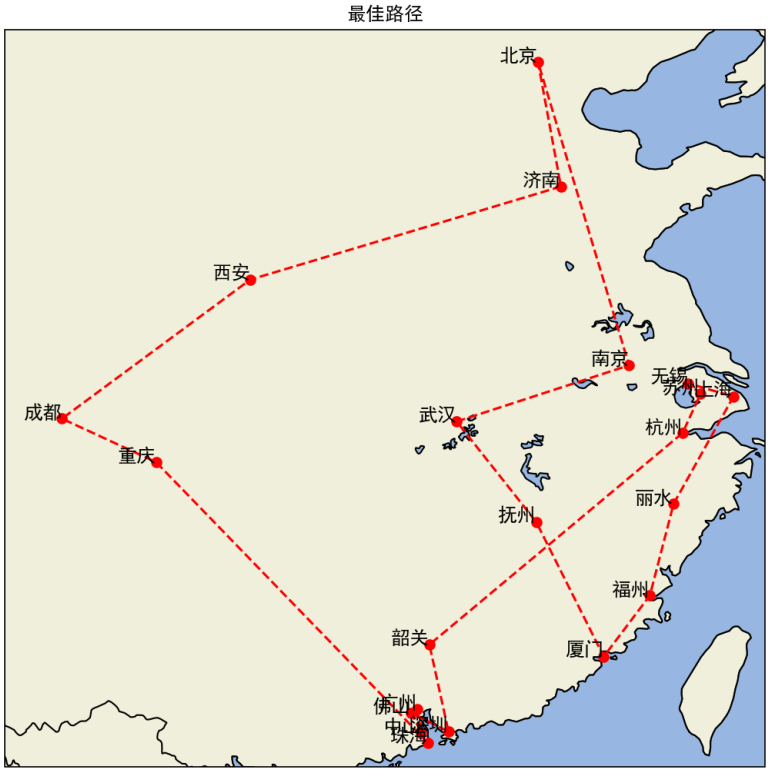

图 4 最佳路径图

注：韶关和厦门之间无高铁可达

基于贪婪算法的 TSP 问题求解结果：

从广州出发，游玩所有城市的总距离为: 61 小时。

最佳路径为: 广州 - 中山 - 佛山 - 深圳 - 韶关 - 杭州 - 苏州 - 无锡 - 上海 - 丽水 - 福州 - 厦门 - 抚州 - 武汉 - 南京 - 北京 - 济南 - 西安 - 成都 - 重庆 - 珠海 - 广州。

基于动态规划的 TSP 问题求解结果：

从广州出发，游玩所有城市的总距离为: 56 小时。

最佳路径为: 广州 - 广州 - 韶关 - 杭州 - 丽水 - 上海 - 无锡 - 苏州 - 南京 - 北京 - 济南 - 重庆 - 成都 - 西安 - 武汉 - 抚州 - 福州 - 厦门 - 深圳 - 佛山 - 珠海 - 中山

由于该问题并非传统的 TSP 问题，可能在规定的 144 小时内无法到达路线较远的城市。因此，相较于动态规划算法，采用贪婪算法逐步求解局部最优路径的实际效果更为理想。

获取最短路径后，动态调整游玩路线。由于附加数据中的 5 分评价可能不准确，因此根据真实排名选择景点。

根据贪婪算法求解路线制定的 144 小时游玩计划如下：

表 4 游玩计划表 1

天数	时间	城市	景点	门票（元）	车票（元）	状态
第一天	8:00 - 9:00	广州				缓冲
	9:00 - 12:00		中山纪念堂	9		游览
	12:00 - 13:00					休息
	13:00 - 14:00					缓冲
	14:00 - 15:00	广州 - 中山			40	转移
	15:00 - 16:00	中山				缓冲
	16:00 - 19:00		幻彩摩天轮	60		游览
	19:00 - 20:00					缓冲
	20:00 - 22:00	中山 - 深圳			400	转移
	22:00 - 7:00	深圳				休息
第二天	7:00 - 9:30					缓冲
	9:30 - 12:00		深圳野生动物园	240		游玩
	12:00 - 13:00					休息
	13:00 - 14:00					缓冲
	14:00 - 16:00	深圳 - 韶关			600	转移
	16:00 - 17:00	韶关				缓冲
	17:00 - 20:00		丹霞山	150		游览
	20:00 - 21:00					缓冲
	21:00 - 5:00	韶关 - 杭州			700	转移
第三天	5:00 - 8:00	杭州				缓冲
	8:00 - 12:00		西湖风景名胜区	0		游览
	12:00 - 13:00					休息
	13:00 - 14:00					缓冲
	14:00 - 16:00	杭州 - 苏州			200	转移
	16:00 - 17:00	苏州				缓冲
	17:00 - 20:00		山塘街	100		游览

	20:00 - 21:00					缓冲
	21:00 - 22:00	苏州 - 无锡			10	转移
	22:00 - 8:00					休息
第四天	8:00 - 9:00					缓冲
	9:00 - 12:00	无锡				游览
	12:00 - 13:00					休息
	13:00 - 14:00					缓冲
	14:00 - 16:00	无锡 - 上海			20	转移
	16:00 - 17:00					缓冲
	17:00 - 21:00	上海	东方明珠	160		游览
	21:00 - 22:00					缓冲
	22:00 - 1:00	上海 - 丽水			200	转移
第五天	1:00 - 8:00					休息
	8:00 - 9:00					缓冲
	9:00 - 12:00	丽水	古堰画乡景区	50		游览
	12:00 - 13:00					休息
	13:00 - 14:00					缓冲
	14:00 - 17:00	丽水 - 福州			200	转移
	17:00 - 18:00					缓冲
	18:00 - 21:00	福州	福州平谭岛	0		游览
	21:00 - 22:00					缓冲
	22:00 - 23:00	福州 - 厦门			200	转移
	23:00 - 8:00					休息
第六天	7:00 - 8:00					缓冲
	8:00 - 11:00	厦门	鼓浪屿	90		游览
	11:00 - 12:00					休息
	12:00 - 13:00					缓冲
	13:00 - 18:00	厦门 - 抚州				转移
	18:00 - 19:00					缓冲
	19:00 - 22:00	抚州	流坑古村	60		游玩
	22:00 -					离开

经统计，从入境到出境所用时间约为 134 小时，共 11 个城市的 11 个景点，乘坐高铁费用为 2570 元，游览景点费用为 919 元，共计 3489 元。

游玩路线可视化如下图所示：

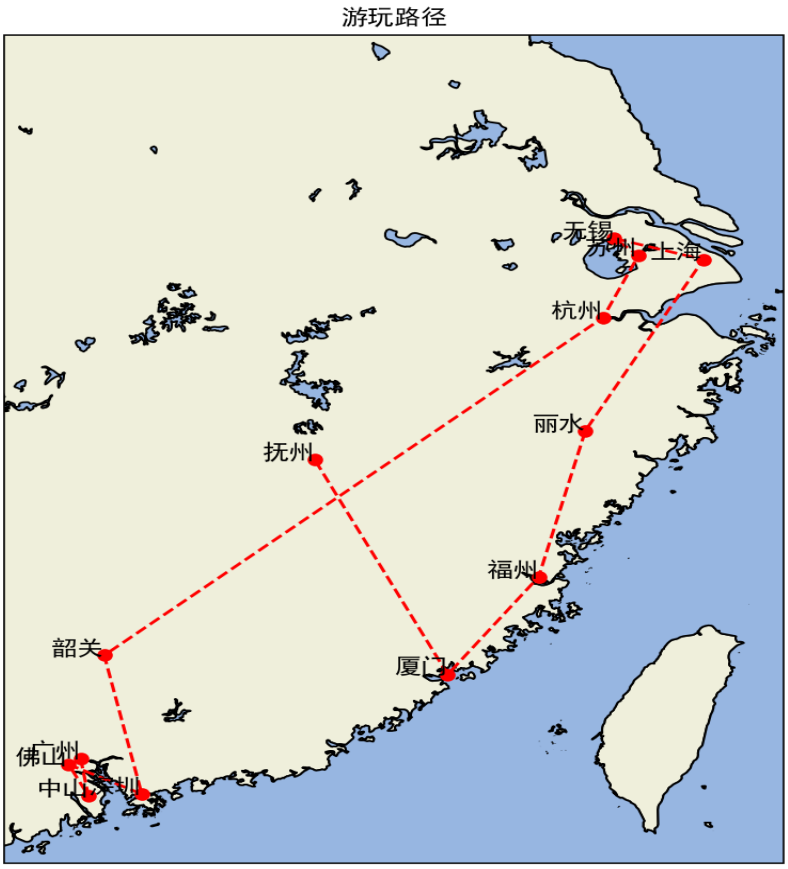

图 5 游玩路径图

5.4 问题四的模型建立与求解

5.4.1 算法介绍

基于遗传算法的多目标旅行商问题，目标为尽可能少的乘坐高铁时间与费用。

多目标旅行商问题

在多目标旅行商问题（MO-TSP）中，需要处理的两个目标分别是：

1. 高铁时间最小化:T

2. 高铁费用最小化:P

遗传算法在 MO-TSP 中的应用

1. 表示解:在旅行商问题中，一个解通常由城市的访问顺序表示。

2. 目标函数:

$$\begin{cases} T = \sum_{i=1}^{n-1} t_{i,i+1} \\ P = \sum_{i=1}^{n-1} p_{i,i+1} \end{cases} \tag{6}$$

其中，$t_{i,i+1}$表示从城市i到城市$i+1$的高铁时间，$p_{i,i+1}$表示对应的高铁费用，n是城市的总数。

3. 适应度函数：由于有多个目标，适应度函数可以是一个权衡函数或者 Pareto 前沿。权衡函数可以将多个目标合并为一个单一目标。

$$F = w_1 T + w_2 P \tag{7}$$

其中，w_1和w_2是权重系数，用于调整时间和费用在总适应度中的相对重要性。

由于 144 小时的限制，故设置时间的权重为 0.6，费用的权重为 0.4。

4. 选择、交叉和变异：遗传算法中的选择、交叉和变异操作需要特别考虑多目标优化的特点。选择操作通常根据每个个体的 Pareto 前沿位置进行；交叉和变异操作需要确保生成的新个体在目标空间中尽可能分布均匀。

5. Pareto 优化：在多目标遗传算法中，Pareto 优化是一种常用的方法。Pareto 前沿表示的是在不降低任何一个目标值的情况下，不能进一步优化其他目标的解集合。遗传算法通过选择那些在 Pareto 前沿上的个体进行繁殖，从而找到多个目标之间的最佳平衡点。

5.4.2 问题求解

使用遗传算法求解多目标旅行商问题并制定行程结果如下表所示：

表 5 游玩计划表

天数	时间	城市	景点	门票（元）	车票（元）	状态
第一天	8:00 - 9:00	广州				缓冲
	9:00 - 12:00		中山纪念堂	9		游览
	12:00 - 13:00					休息
	13:00 - 14:00					缓冲
	14:00 - 16:00	广州 - 珠海			70	转移
	16:00 - 17:00	珠海				缓冲
	17:00 - 20:00		珠海渔女	0		游览
	20:00 - 21:00					缓冲
	22:00 - 22:00	珠海 - 中山			30	转移
	22:00 -	中山				休息
第二天	8:00 - 9:00					缓冲
	9:00 - 12:00		孙中山故居纪念馆	0		游览
	12:00 - 13:00					休息
	13:00 - 14:00					缓冲
	14:00 - 16:00	中山 - 佛山			60	转移
	16:00 - 17:00	佛山				缓冲
	17:00 - 20:00		佛山石湾	0		游览
	20:00 - 21:00					缓冲

	21:00 -	佛山 - 深圳			100	转移
第三天	7:00 - 8:00	深圳				缓冲
	8:00 - 12:00		深圳欢乐海岸	60		游览
	12:00 - 13:00					休息
	13:00 - 14:00					缓冲
	14:00 - 16:00	深圳 - 韶关			600	转移
	16:00 - 17:00	韶关				缓冲
	17:00 - 20:00		丹霞山	150		游览
	20:00 - 21:00					缓冲
	21:00 - 1:00	韶关 - 武汉			400	转移
	1:00 -	武汉				休息
第四天	8:00 - 9:00					缓冲
	9:00 - 12:00		黄鹤楼	70		游览
	12:00 - 13:00					休息
	13:00 - 14:00					缓冲
	14:00 - 18:00	武汉 - 抚州			200	转移
	18:00 - 19:00	抚州				缓冲
	19:00 - 22:00		抚州汝水公园	0		游览
	22:00 - 23:00					缓冲
	23:00 - 3:00	抚州 - 厦门			200	转移
第五天	3:00 - 8:00	厦门				休息
	8:00 - 9:00					缓冲
	9:00 - 12:00		鼓浪屿	50		游览
	12:00 - 13:00					休息
	13:00 - 14:00					缓冲
	14:00 - 17:00	厦门 - 福州			200	转移
	17:00 - 18:00	福州				缓冲
	18:00 - 21:00		福州平潭岛	0		游览
	21:00 - 22:00					缓冲
	22:00 - 1:00	福州 - 丽水			200	转移
	1:00 - 8:00	丽水				休息
第六天	8:00 - 9:00					缓冲
	9:00 - 12:00		古堰画乡景区	50		游览
	12:00 - 13:00					休息
	13:00 - 14:00					缓冲
	14:00 - 17:00	丽水 - 上海			200	转移
	17:00 - 18:00	上海				缓冲
	18:00 - 21:00		外滩	0		游览
	21:00 - 22:00					缓冲
	22:00 -					离开

最佳路径可视化如下：

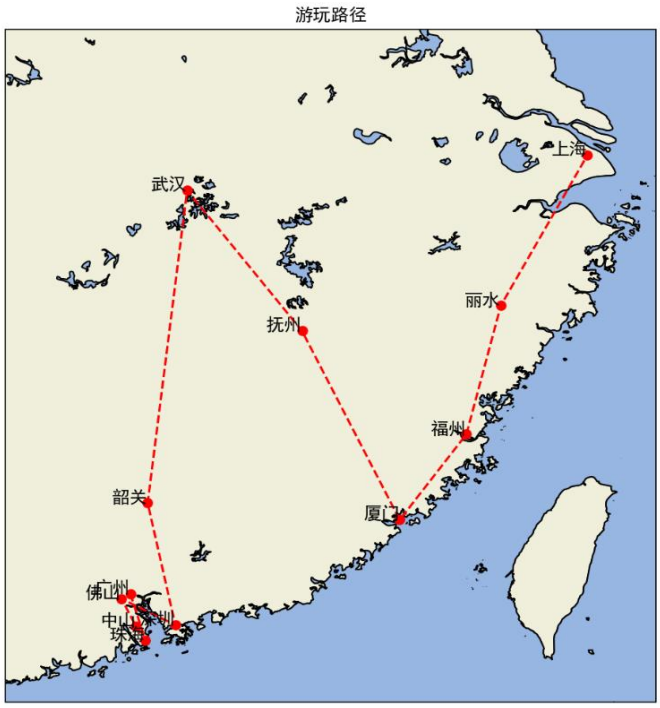

图 6 游玩路径图

经统计，从入境到出境所用时间约为 134 小时，共 11 个城市的 11 个景点，乘坐高铁费用为 2260 元，游览景点费用为 389 元，共计 2649 元。

5.5 问题五的模型建立与求解

针对外国游客的三个要求：

- 每个城市只游玩一座评分最高的山；
- 城市之间的交通方式只选择高铁；
- 旅游城市不局限于“最令外国游客向往的 50 个城市”，游览范围拓展到 352 个城市。

为了优化游玩体验，可以简化选择范围，直接挑选中国境内评价较高的山景，检索结果如下，黄山（黄山市）、泰山（泰安市）、华山（华阴市）、峨眉山（乐山市）、庐山（九江市）、五台山（忻州市）、雁荡山（温州市）、衡山（衡阳市）、嵩山（郑州市）、丹霞山（韶关市）、青城山（成都市）、武当山（十堰市）、老君山（洛阳市）、武功山（萍乡市）、武夷山（南平市）、舟山（舟山市）、钟南山（西安市）、太白山（宝鸡市）、天门山（张家界市）和梵净山（铜仁市）。

其中结合，高铁的可达性与入境出境的选择，最终选择的城市为温州市、黄山市、南平市、九江市、萍乡市、韶关市、衡阳市、郑州市、洛阳市、宝鸡市、西安市和成都市。采用温州作为入境城市，成都作为出境城市。

然后，通过铁路 12306 平台，查询两两城市的票价以及时间，如下表所示：

表 6 城市之间的高铁时间及票价表 2

票价 时间	黄山	九江	温州	衡阳	郑州	韶关	成都	洛阳	…
黄山	0	300,4	9999	500,6	9999	900,11	500,6	500,4	…
九江	9999	0	300,5	90,10	400,5	200,11	500,12	400,6	…
温州	300,4	300,5	0	200,19	600,8	200,23	1100,14	700,11	…
衡阳	9999	90,10	200,19	0	200,11	200,2	9999	200,6	…
郑州	500,6	400,5	600,8	200,11	0	600,6	500,7	70,1	…
韶关	9999	200,11	200,23	200,2	600,6	0	9999	700,7	…
成都	900,11	500,12	1100,14	9999	500,7	9999	0	500,5	…
洛阳	500,6	400,6	700,11	900,6	70,1	700,7	500,5	0	…
萍乡	500,4	200,6	400,5	200,2	500,5	300,3	700,8	9999	…
南平	200,2	200,3	9999	9999	500,7	9999	900,11	700,8	…
西安	700,7	600,7	900,12	700,8	300,2	700,8	300,4	200,2	…
…	…	…	…	…	…	…	…	…	…

其中，由于考虑前两问中实际游览的城市不过 12 个，在结合游览山景是需要大量体力及需要花时间游览以及休息，故在本文中将问题简化为最短路径问题而非旅行商为题，将游览的城市设为 9、10、11 和 12，其中以温州为起点，成都为终点，中间排列组合选取其它城市，并结合加权路径优化算法进行求解。

加权路径优化算法公式如下：

$$Total\,\mathrm{Cos}t=\sum_{i=1}^{n-1}\left(0.5\times t_{X_i,X_{i+1}}+0.4\times p_{X_i,X_{i+1}}\right)+\left(0.5\times t_{X_n,X_1}+0.4\times p_{X_n,X_1}\right)+\sum_{i=1}^{n}0.1\times m_i$$

求解的结果如下表所示：

表 7 游玩路径表

城市数	时间	车票	门票	合计	路径
9	29	1580	930	2510	温州-黄山-南平-九江-衡阳-郑州-洛阳-宝鸡-西安-成都
10	45	1470	1040	2510	温州-黄山-南平-九江-萍乡-衡阳-郑州-洛阳-宝鸡-西安-成都
11	43	1780	1110	2890	温州-黄山-南平-九江-萍乡-韶关-衡阳-郑州-洛阳-宝鸡-西安-成都
12	46	2080	1210	3290	温州-黄山-南平-九江-郑州-洛阳-宝鸡-西安-成都

最后根据求解的结果以及 144 小时的限制，最终选择游览 10 个城市，游玩计划表如下：

表 8 游玩计划表

天数	时间	城市	景点	门票（元）	车票（元）	状态
第一天	8:00 - 9:00	温州				缓冲
	9:00 - 13:00		雁荡山	15		游览
	13:00 - 14:00					休息
	14:00 - 15:00					缓冲
	15:00 - 19:00	温州 - 黄山				转移
	19:00 - 20:00	黄山				缓冲
	20:00 -					休息
第二天	5:00 - 10:00		黄山	190		游览
	10:00 - 11:00					缓冲
	11:00 - 12:00					休息
	12:00 - 13:00					缓冲
	13:00 - 17:00	黄山 - 南平			200	转移
	17:00 - 18:00	南平				缓冲
	18:00 - 23:00		武夷山	140		游览
	23:00 -					休息
第三天	6:00 - 9:00	南平 - 九江			200	转移
	9:00 - 10:00	九江				缓冲
	10:00 - 14:00		庐山	160		游览
	14:00 - 15:00					缓冲
	14:00 - 24:00	九江 - 衡阳			90	转移
第四天	5:00 - 10:00	衡阳	衡山	110		游览
	10:00 - 11:00					缓冲
	11:00 - 12:00					休息
	13:00 - 24:00	衡阳 - 郑州			200	转移
第五天	3:00 - 8:00	郑州	嵩山	80		游览
	8:00 - 9:00					缓冲
	9:00 - 10:00	郑州 - 洛阳			70	转移
	10:00 - 11:00	洛阳				缓冲
	11:00 - 12:00					休息
	12:00 - 17:00		老君山	100		转移
	17:00 - 18:00					缓冲
	18:00 - 24:00	洛阳 - 宝鸡			80	转移
第六天	5:00 - 9:00	宝鸡	太白山	45		游览
	9:00 - 10:00					缓冲
	10:00 - 12:00	宝鸡 - 西安			30	转移
	12:00 - 13:00	西安				缓冲
	13:00 - 17:00		钟南山	100		游览
	17:00 - 18:00					缓冲
	18:00 - 22:00	西安 - 成都		300		转移
	22:00 -	成都				休息

第七天	4:00 - 7:00		青城山	80		游览
	7:00 - 8:00					离开

最终结果为游览 10 个城市，路线为温州-黄山-南平-九江-萍乡-衡阳-郑州-洛阳-宝鸡-西安-成都，高铁票与门票共计 2510 元，时长约 144 小时。

游览结果可视化如下：

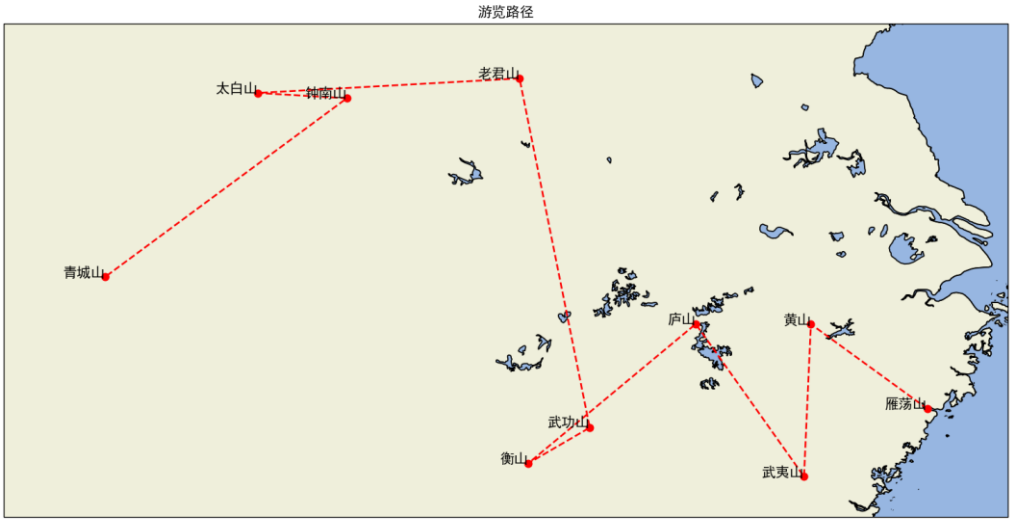

图 7 游玩路径图

六、 模型的评价、改进与推广

6.1 模型的优点

由于在 144 小时的时间限制内无法游览所有景点，因此目标是找到一个局部最优解，尽可能地优化旅行路径。为此，可以使用遗传算法（Genetic Algorithm，GA）来寻找近似最优解。在求解 TSP（旅行商问题）时采用贪婪算法进行求解，使得局部最优。

6.2 模型的缺点

由于该问题并非经典的旅行商问题，并且在模型中对许多实际因素进行了简化，因此这种算法的解法可能难以直接应用于实际旅行规划中。在现实生活中，旅行中涉及的因素，如交通状况、天气变化、景点开放时间以及个人兴趣等，都可能对路径优化产生重大影响。因此，尽管遗传算法在理论上能够提供有效的路径优化方案，但在实际应用时仍需考虑更多复杂因素，以确保制定的旅行计划切实可行并符合实际需求。

6.3 模型的改进

模型的改进主要体现在两个方面。首先，通过结合深度学习方法，模型能够更有效地处理复杂的模式识别和处理旅行商问题。其次，模型在设计时充分考虑了各种不稳定因素，如实时交通状况、天气变化和突发事件等，这使得模型能够在动态和不确定的环境中提供更为可靠和灵活的解决方案。这些改进使得模型不仅能够更准确地预测和优化结果，还能适应实际应用中可能遇到的各种挑战。

6.4 模型的推广

在实际应用中，采用模型进行路线规划时，可以通过实时数据和外部因素的动态调整来优化结果。模型首先根据预设的参数和目标进行初步规划，生成最优或接近最优的路线。随后，通过集成实时信息，如交通状况、天气变化和道路封闭情况等，模型可以实时更新和调整路线，以应对突发事件或环境变化。这种动态调整机制使得规划方案更加灵活和可靠，能够在不断变化的条件下提供最适合的路线选择，从而提高效率和用户满意度。

七、 参考文献

[1] 中国政府网，“中国在这里，欢迎大家来！（最新 24、72、144 小时过境免签名单）”，https://www.gov.cn/zhengce/jiedu/tujie/202407/content_6961526.htm

[2] 朱佳艺,刘从军.基于Selenium的自动化测试框架设计与实现[J].软件导刊,2023,22(05):103-108.

[3] 邱云飞,刘一菲,于智龙,等.求解旅行商问题的GCN-Pointransformer模型[J/OL].计算机科学与探索,1-14[2024-08-18].http://kns.cnki.net/kcms/detail/11.5602.tp.20240624.1713.002.html.

[4] 孙冰,王川,杨强,等.面向多起点均衡多旅行商问题的进化算法[J].计算机工程与设计,2023,44(07):2030-2038.

[5] Denis J ,Petar O ,Karlo L .Data collection automation with the help of Selenium[J].Zbornik radova Medimurskog veleucilista u Cakovcu,2023,14(2):51-57.

[6] Picture 2-tuple linguistic aggregation operators in multiple attribute decision making[J]. Guiwu Wei;Fuad E. Alsaadi;Tasawar Hayat;Ahmed Alsaedi.Soft Computing,2018

[7] 张骏,古凤,卢凤萍.城市旅游空间行为路径分析及优化——以南京市为例[J].地理与地理信息科学,2011,27(01):85-89.

附录

附录 1
介绍：支撑材料的文件列表
问题一 ----附件 --------附件数据 ----5 分景点.xlsx ----可视化.py ----城市分布.png ----数据.csv ----数据合并.py ----数据统计.py ----数据采集.py 问题二 ----50 个城市.png ----城市评分.py ----可视化.py ----数据.xlsx 问题三 ----价格.xlsx ----动态规划.py ----可视化.py ----数据.txt ----时间.xlsx ----景点.xlsx ----最佳路线.png ----游玩路径.png ----游玩路径可视化.py ----结果.py ----贪婪算法.py ----部分城市可视化.png 问题四 ----可视化.py ----游玩路径.png ----遗传算法.py 问题五 ----加权路径优化.py ----可视化.py

----标准化.py
----游览路径.png

附录 2

介绍：代码

```
-----------------------------------------------------------------------------------------------------------
                                        问题一 可视化.py
-----------------------------------------------------------------------------------------------------------
# -*- coding: utf-8 -*-

import cartopy.feature as cfeature
import matplotlib.pyplot as plt
import cartopy.crs as ccrs

plt.rcParams['font.family'] = 'SimHei'
plt.rcParams['axes.unicode_minus'] = False

# 城市及其坐标
cities = {
    '烟台 18': (37.539297, 121.391382),
    '泸州 14': (28.889138, 105.443348),
    '自贡 13': (29.352765, 104.773447),
    '玉溪 13': (24.350461, 102.543907),
    '内江 13': (29.58708, 105.066138),
    '昭通 12': (27.336999, 103.717216),
    '益阳 12': (28.570066, 112.355042),
    '宁德 12': (26.65924, 119.527082),
    '抚州 12': (27.98385, 116.358351),
    '德州 12': (37.453968, 116.307428),
}

# 获取城市的经纬度
lats, lons = zip(*cities.values())

# 创建一个新的图形
fig, ax = plt.subplots(figsize=(10, 10), subplot_kw={'projection': ccrs.PlateCarree()})

# 添加中国地图的特征
ax.add_feature(cfeature.COASTLINE)
ax.add_feature(cfeature.BORDERS)
ax.add_feature(cfeature.LAND, edgecolor='black')
ax.add_feature(cfeature.OCEAN)
ax.add_feature(cfeature.LAKES, edgecolor='black')
```

```
# 设置显示区域以集中在中国
ax.set_extent([73, 135, 18, 54], crs=ccrs.PlateCarree())

# 绘制城市位置
ax.scatter(lons, lats, color='red', s=20, transform=ccrs.PlateCarree())  # 使用 scatter 绘制点
for city, (lat, lon) in cities.items():
    ax.text(lon, lat, city, fontsize=12, ha='right', transform=ccrs.PlateCarree())

# 连接点的顺序
# ax.plot(lons, lats, linestyle='--', marker='o', color='red', transform=ccrs.PlateCarree())

# 添加标题
plt.title('城市分布')

# 显示图形
plt.show()
```

问题一 数据合并.py

```
# -*- coding: utf-8 -*-

import pandas as pd
import warnings
import os

# 忽略警告
warnings.filterwarnings('ignore')
folder_path = "附件"
output_file_path = "数据.csv"

# 创建一个空的 DataFrame 用于存放合并后的数据
all_data = pd.DataFrame()

for filename in os.listdir(folder_path):
    if filename.endswith(".csv"):
        file_path = os.path.join(folder_path, filename)
        df = pd.read_csv(file_path)

        # 筛选出指定的列
        if all(col in df.columns for col in ["名字", "链接", "评分"]):
            df_filtered = df[["名字", "链接", "评分"]]
```

```
            # 添加一列城市数据
            df_filtered["城市"] = filename.replace(".csv", "")

            # 将筛选后的数据追加到总的 DataFrame 中
            all_data = pd.concat([all_data, df_filtered], ignore_index=True)

# 保存所有数据到一个新的 CSV 文件中
all_data.to_csv(output_file_path, index=False)

print("已完成数据合并。")
```

问题一 数据统计.py

```
# -*- coding: utf-8 -*-

import pandas as pd

file_path = "数据.csv"
df = pd.read_csv(file_path)

print(f"数据总量为 {df.shape[0]} 条。")


file_path = "5 分景点.xlsx"
df = pd.read_excel(file_path)

print(f"去重后数据总量为 {df.shape[0]} 条。")
```

问题一 数据采集.py

```
# -*- coding: utf-8 -*-

from selenium.webdriver.common.by import By
from selenium import webdriver
import pandas as pd
import time

# 设置 EdgeDriver
driver = webdriver.Edge()
df = pd.read_excel('景点.xlsx', sheet_name='Sheet4')
```

```
data_url = df['链接']
url = 'http://travel.qunar.com/p-oi38858457-exianlingshengtaiwenhua'

driver = webdriver.Edge()
driver.get(url)
time.sleep(1)
# 提取数据
rating_element = driver.find_element(By.XPATH,

'/html/body/div[3]/div/div[1]/div[2]/div[4]/div/div[2]/div[1]/span[1]')
# /html/body/div[3]/div/div[1]/div[2]/div[4]/div/div[2]/div[1]/span[1]
rating = rating_element.text
print(f'Rating: {rating}')

# 关闭浏览器
driver.quit()
```

问题二 可视化.py

```
# -*- coding: utf-8 -*-

import cartopy.feature as cfeature
import matplotlib.pyplot as plt
import cartopy.crs as ccrs

plt.rcParams['font.family'] = 'SimHei'
plt.rcParams['axes.unicode_minus'] = False

# 城市及其坐标
cities = {
    '上海': (31.231706, 121.472644),
    '北京': (39.904989, 116.405285),
    '深圳': (22.547, 114.085947),
    '广州': (23.125178, 113.280637),
    '苏州': (31.299379, 120.619585),
    '成都': (30.659462, 104.065735),
    '南京': (32.041544, 118.767413),
    '无锡': (31.574729, 120.301663),
    '济南': (36.675807, 117.000923),
    '重庆': (29.533155, 106.504962),
    '西安': (34.263161, 108.948024),
    '呼伦贝尔': (49.215333, 119.758168),
    '福州': (26.075302, 119.306239),
```

```
        '厦门': (24.490474, 118.11022),
        '雅安': (29.987722, 103.001033),
        '丽水': (28.451993, 119.921786),
        '中山': (22.521113, 113.382391),
        '武汉': (30.584355, 114.298572),
        '抚州': (27.98385, 116.358351),
        '佛山': (23.028762, 113.122717),
        '杭州': (30.287459, 120.153576),
        '韶关': (24.801322, 113.591544),
        '汕头': (23.37102, 116.708463),
        '三亚': (18.247872, 109.508268),
        '珠海': (22.255899, 113.552724),
        '儋州': (19.517486, 109.576782),
        '南昌': (28.676493, 115.892151),
        '承德': (40.976204, 117.939152),
        '百色': (23.897742, 106.616285),
        '北海': (21.473343, 109.119254),
        '陵水': (18.505006, 110.037218),
        '秦皇岛': (39.942531, 119.586579),
        '郑州': (34.757975, 113.665412),
        '南宁': (22.82402, 108.320004),
        '东莞': (23.048884, 113.760234),
        '万宁': (18.796216, 110.388793),
        '河源': (23.746266, 114.697802),
        '江门': (22.590431, 113.094942),
        '惠州': (23.079404, 114.412599),
        '金华': (29.089524, 119.649506),
        '揭阳': (23.543778, 116.355733),
        '长沙': (28.19409, 112.982279),
        '宿州': (33.633891, 116.984084),
        '玉林': (22.63136, 110.154393),
        '宁波': (29.868388, 121.549792),
        '哈尔滨': (45.756967, 126.642464),
        '张家口': (40.811901, 114.884091),
        '晋中': (37.696495, 112.736465),
        '周口': (33.620357, 114.649653),
        '嘉峪关': (39.786529, 98.277304)
}
# 获取城市的经纬度
lats, lons = zip(*cities.values())

# 创建一个新的图形
fig, ax = plt.subplots(figsize=(10, 10), subplot_kw={'projection': ccrs.PlateCarree()})
```

```
# 添加中国地图的特征
ax.add_feature(cfeature.COASTLINE)
ax.add_feature(cfeature.BORDERS)
ax.add_feature(cfeature.LAND, edgecolor='black')
ax.add_feature(cfeature.OCEAN)
ax.add_feature(cfeature.LAKES, edgecolor='black')

# 设置显示区域以集中在中国
ax.set_extent([73, 135, 18, 54], crs=ccrs.PlateCarree())

# 绘制城市位置
ax.scatter(lons, lats, color='red', s=20, transform=ccrs.PlateCarree())  # 使用 scatter 绘制点
for city, (lat, lon) in cities.items():
    ax.text(lon, lat, city, fontsize=12, ha='right', transform=ccrs.PlateCarree())

# 连接点的顺序
# ax.plot(lons, lats, linestyle='--', marker='o', color='red', transform=ccrs.PlateCarree())

# 添加标题
plt.title('最令外国游客向往的中国 50 个城市')

# 显示图形
plt.show()
```

问题二 城市评分.py

```
# -*- coding: utf-8 -*-

import pandas as pd
import numpy as np

# 创建数据
data = pd.read_excel('问题二数据.xlsx')
df = pd.DataFrame(data)

# 评估指标和权重
criteria = ['AQI 值', '总人口', '面积', '人口密度', '环境环保', '人文底蕴', '气候', '美食', '公共汽车数量',
            '出租汽车数量', '平均气温']
weights = [0.1, 0.05, 0.05, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1]
```

```
# 标准化数据
df_norm = df.copy()
for column in criteria:
    df_norm[column] = (df[column] - df[column].min()) / (df[column].max() - df[column].min())

# SAW 评分
df_norm['SAW 得分'] = df_norm[criteria].mul(weights).sum(axis=1)


# VIKOR 评分
def vikor(df, criteria, weights):
    # 确保权重长度与指标一致
    if len(criteria) != len(weights):
        raise ValueError("Criteria 和 weights 的长度必须一致")

    norm_df = df.copy()
    for column in criteria:
        norm_df[column] = (df[column] - df[column].min()) / (df[column].max() - df[column].min())

    # 计算最大和最小值
    max_vals = norm_df[criteria].max()
    min_vals = norm_df[criteria].min()

    # 计算 S 和 R
    S = norm_df[criteria].apply(lambda x: np.sum(weights * (x - min_vals) / (max_vals - min_vals)), axis=1)
    R = norm_df[criteria].apply(lambda x: np.max(weights * (x - min_vals) / (max_vals - min_vals)), axis=1)

    # 最优和最劣解
    S_star = S.min()
    S_bar = S.max()
    R_star = R.min()
    R_bar = R.max()

    # 防止除零错误
    S_bar_minus_S_star = S_bar - S_star
    R_bar_minus_R_star = R_bar - R_star

    if S_bar_minus_S_star == 0:
        S_bar_minus_S_star = 1
    if R_bar_minus_R_star == 0:
```

```
        R_bar_minus_R_star = 1

    Q = 0.5 * (S - S_star) / S_bar_minus_S_star + 0.5 * (R - R_star) / R_bar_minus_R_star
    return Q


df_norm['VIKOR 得分'] = vikor(df, criteria, weights)
# 综合评分（均值）
df_norm['综合得分'] = (df_norm['SAW 得分'] + df_norm['VIKOR 得分']) / 2
# 打印结果
print(df_norm[['城市名称', 'SAW 得分', 'VIKOR 得分', '综合得分']])
# df_norm.to_csv('评分结果.csv', index=False)  # 将数据框保存为 CSV 文件
# 排序并获取前 50 个评分最高的城市
top_50_cities = df_norm[['城市名称', '综合得分']].sort_values(by='综合得分', ascending=False).head(50)

# 打印前 50 个城市的名称
print(top_50_cities['城市名称'].tolist())
```

问题三 动态规划.py

```
# -*- coding: utf-8 -*-

import numpy as np

city_names = [
    "上海", "北京", "深圳", "广州", "苏州", "成都", "南京", "无锡", "济南", "重庆",
    "西安", "福州", "厦门", "丽水", "中山", "武汉", "抚州", "佛山", "杭州", "韶关", "珠海"
]
distance_matrix = np.array([
    [0., 5., 7., 7., 2., 13., 4., 2., 6., 11., 8., 5., 6., 3., 9., 4., 10., 12., 3., 16., 10.],
    [5., 0., 9., 11., 5., 8., 5., 4., 5., 7., 6., 9., 11., 6., 11., 11., 14., np.inf, 5., 10., 12.],
    [7., 9., 0., 2., 10., 8., 8., 9., 12., 7., 10., 8., 4., np.inf, 2., 6., 8., 1., 7., 2., 2.],
    [7., 11., 2., 0., 25., 7., 7., 9., 8., 6., 8., 5., 4., 22., 1., 12., 9., 1., 8., 1., 2.],
    [2., 5., 10., 25., 0., 13., 3., 1., 5., 11., 7., 7., 9., 4., np.inf, 5., np.inf, np.inf, 2., 19., np.inf],
    [13., 8., 8., 7., 13., 0., 11., 13., 9., 2., 4., 13., 14., 13., 11., 10., 10., 10., 13., np.inf, 11.],
    [4., 5., 8., 7., 3., 11., 0., 3., 8., 10., 6., 6., np.inf, 4., np.inf, 4., np.inf, np.inf, 2., 8., np.inf],
    [2., 4., 9., 9., 1., 13., 3., 0., 9., 12., 9., 8., 9., 4., np.inf, 5., np.inf, np.inf, 3., 9., np.inf],
```

```
    [6., 5., 12., 8., 5., 9., 8., 9., 0., 7., 5., 7., 9., 7., np.inf, 6., np.inf, np.inf, 5., 10., np.inf],
    [11., 7., 7., 6., 11., 2., 10., 12., 7., 0., 6., 14., 14., 11., 7., 7., 10., 6., 12., 20., 8.],
    [8., 6., 10., 8., 7., 4., 6., 9., 5., 6., 0., 10., 12., 11., 11., 4., 20., 13., 8., np.inf, np.inf],
    [5., 9., 8., 5., 7., 13., 6., 8., 7., 14., 10., 0., 1., 3., np.inf, 6., 3., np.inf, 5., np.inf, np.inf],
    [6., 11., 4., 4., 9., 14., np.inf, 9., 9., 14., 12., 1., 0., np.inf, np.inf, 8., 5., np.inf, 7., 18.,
np.inf],
    [3., 6., np.inf, 22., 4., 13., 4., 4., 7., 11., 11., 3., np.inf, 0., np.inf, 7., np.inf, np.inf, 2.,
2., np.inf],
    [9., 11., 2., 1., np.inf, 11., np.inf, np.inf, np.inf, 7., 11., np.inf, np.inf, np.inf, 0., 5.,
np.inf, 1., 8., 4.,
     1.],
    [4., 11., 6., 12., 5., 10., 4., 5., 6., 7., 4., 6., 8., 7., 5., 0., 4., np.inf, 5., np.inf, 5.],
    [10., 14., 8., 9., np.inf, 10., np.inf, np.inf, np.inf, 10., 20., 3., 5., np.inf, np.inf, 4., 0.,
np.inf, 9.,
     np.inf, np.inf],
    [12., np.inf, 1., 1., np.inf, 10., np.inf, np.inf, np.inf, 6., 13., np.inf, np.inf, np.inf, 1.,
np.inf, np.inf, 0.,
     11., 8., 2.],
    [3., 5., 7., 8., 2., 13., 2., 3., 5., 12., 8., 5., 7., 2., 8., 5., 9., 11., 0., 0., 9.],
    [16., 10., 2., 1., 19., np.inf, 8., 9., 10., 20., np.inf, np.inf, 18., 2., 4., np.inf, np.inf, 8.,
0., 0., 2.],
    [10., 12., 2., 2., np.inf, 11., np.inf, np.inf, np.inf, 8., np.inf, np.inf, np.inf, np.inf, 1.,
5., np.inf, 2., 9.,
     2., 0.]
])

# 替换 inf 为一个很大的数字
inf = 1e9
distance_matrix = np.where(np.isinf(distance_matrix), inf, distance_matrix)


def tsp_dynamic_programming(start):
    n = len(distance_matrix)
    # dp[mask][i] 表示遍历了 mask 所包含的城市，当前位于城市 i 的最短路径长度
    dp = np.full((1 << n, n), inf)
    dp[1 << start][start] = 0

    # 遍历所有可能的状态
    for mask in range(1 << n):
        for u in range(n):
            if mask & (1 << u):
                for v in range(n):
                    if not (mask & (1 << v)):
```

```
                                new_mask = mask | (1 << v)
                                dp[new_mask][v] = min(dp[new_mask][v], dp[mask][u] +
distance_matrix[u][v])

    # 最终路径长度为遍历所有城市后返回到起点的最短路径长度
    end_mask = (1 << n) - 1
    min_cost = inf
    last_city = -1
    for i in range(n):
        if dp[end_mask][i] + distance_matrix[i][start] < min_cost:
            min_cost = dp[end_mask][i] + distance_matrix[i][start]
            last_city = i

    # 回溯路径
    path = []
    mask = end_mask
    current_city = last_city
    while mask:
        path.append(current_city)
        next_city = -1
        for u in range(n):
            if mask & (1 << u) and dp[mask][current_city] == dp[mask ^ (1 <<
current_city)][u] + distance_matrix[u][
                    current_city]:
                    next_city = u
                    break
        mask ^= (1 << current_city)
        current_city = next_city
    path.append(start)
    path.reverse()

    return path, min_cost


# 从广州出发（索引为3）
start_city = 3
path, total_distance = tsp_dynamic_programming(start_city)

print(f"从广州出发，游玩所有城市的总距离为: {total_distance:.2f} 小时")
print("最短路径为:", " -> ".join(f"{city_names[city]}" for city in path))
```

问题三 可视化.py

```
# -*- coding: utf-8 -*-

from scipy.optimize import linear_sum_assignment
from scipy.spatial import distance_matrix
import cartopy.feature as cfeature
import matplotlib.pyplot as plt
import cartopy.crs as ccrs
import numpy as np

plt.rcParams['font.family'] = 'SimHei'
plt.rcParams['axes.unicode_minus'] = False
# 游玩的城市：  ['广州', '中山', '深圳', '佛山', '珠海', '福州', '厦门', '杭州', '苏州', '无锡', '上海', '丽水', '南京', '武汉', '西安', '抚州', '成都', '重庆', '北京', '济南', '韶关']

cities = {
      '广州': (23.125178, 113.280637),
      '中山': (22.521113, 113.382391),
      '佛山': (23.028762, 113.122717),
      '深圳': (22.547, 114.085947),
      '韶关': (24.801322, 113.591544),
      '杭州': (30.287459, 120.153576),
      '苏州': (31.299379, 120.619585),
      '无锡': (31.574729, 120.301663),
      '上海': (31.231706, 121.472644),
      '丽水': (28.451993, 119.921786),
      '福州': (26.075302, 119.306239),
      '厦门': (24.490474, 118.11022),
      '抚州': (27.98385, 116.358351),
      '武汉': (30.584355, 114.298572),
      '南京': (32.041544, 118.767413),
      '北京': (39.904989, 116.405285),
      '济南': (36.675807, 117.000923),
      '西安': (34.263161, 108.948024),
      '成都': (30.659462, 104.065735),
      '重庆': (29.533155, 106.504962),
      '珠海': (22.255899, 113.552724),
      '广州': (23.125178, 113.280637),



}
# 定义中国 50 个城市的经纬度

# 将坐标转换为 numpy 数组
city_names = list(cities.keys())
```

```
coords = np.array(list(cities.values()))

# 使用距离矩阵求解 TSP
dist_matrix = distance_matrix(coords, coords)

# 使用 Cartopy 绘制
fig, ax = plt.subplots(figsize=(10, 10), subplot_kw={'projection': ccrs.PlateCarree()})

# 添加地图功能
ax.add_feature(cfeature.BORDERS)
ax.add_feature(cfeature.COASTLINE)
ax.add_feature(cfeature.LAND)
ax.add_feature(cfeature.OCEAN)
ax.add_feature(cfeature.LAKES, edgecolor='black')
ax.add_feature(cfeature.RIVERS)

# 绘制城市
for city, (lat, lon) in cities.items():
    ax.plot(lon, lat, 'o', color='red', transform=ccrs.PlateCarree())
    ax.text(lon, lat, city, fontsize=10, ha='right', transform=ccrs.PlateCarree())

plt.title('排名靠前的部分城市可视化')
plt.show()
```

问题三 游玩路径可视化.py

```
# -*- coding: utf-8 -*-

import cartopy.feature as cfeature
import matplotlib.pyplot as plt
import cartopy.crs as ccrs

plt.rcParams['font.family'] = 'SimHei'
plt.rcParams['axes.unicode_minus'] = False

# 城市及其坐标
cities = {
    '广州': (23.125178, 113.280637),
    '中山': (22.521113, 113.382391),
    '佛山': (23.028762, 113.122717),
    '深圳': (22.547, 114.085947),
    '韶关': (24.801322, 113.591544),
    '杭州': (30.287459, 120.153576),
```

```
    '苏州': (31.299379, 120.619585),
    '无锡': (31.574729, 120.301663),
    '上海': (31.231706, 121.472644),
    '丽水': (28.451993, 119.921786),
    '福州': (26.075302, 119.306239),
    '厦门': (24.490474, 118.11022),
    '抚州': (27.98385, 116.358351),
}

# 获取城市的经纬度
lats, lons = zip(*cities.values())

# 创建一个新的图形
fig, ax = plt.subplots(figsize=(10, 10), subplot_kw={'projection': ccrs.PlateCarree()})

# 添加中国地图的特征
ax.add_feature(cfeature.COASTLINE)
ax.add_feature(cfeature.BORDERS)
ax.add_feature(cfeature.LAND, edgecolor='black')
ax.add_feature(cfeature.OCEAN)
ax.add_feature(cfeature.LAKES, edgecolor='black')

# 设置显示区域以集中在中国
ax.set_extent([73, 135, 18, 54], crs=ccrs.PlateCarree())

# 绘制城市位置
ax.scatter(lons, lats, color='red', s=20, transform=ccrs.PlateCarree())  # 使用 scatter 绘制点
for city, (lat, lon) in cities.items():
    ax.text(lon, lat, city, fontsize=12, ha='right', transform=ccrs.PlateCarree())

# 连接点的顺序
ax.plot(lons, lats, linestyle='--', marker='o', color='red', transform=ccrs.PlateCarree())

# 添加标题
plt.title('游玩路径')

# 显示图形
plt.show()
```

问题三 结果.py

```
# -*- coding: utf-8 -*-
```

```
import cartopy.feature as cfeature
import matplotlib.pyplot as plt
import cartopy.crs as ccrs

plt.rcParams['font.family'] = 'SimHei'
plt.rcParams['axes.unicode_minus'] = False
# 广州 - 中山 - 佛山 - 深圳 - 韶关 - 杭州 - 苏州 - 无锡 - 上海 - 丽水 - 福州 - 厦门 - 抚州 - 武汉 - 南京 - 北京 - 济南 - 西安 - 成都 - 重庆 - 珠海 - 广州
# 城市及其坐标
cities = {
    '广州': (23.125178, 113.280637),
    '中山': (22.521113, 113.382391),
    '佛山': (23.028762, 113.122717),
    '深圳': (22.547, 114.085947),
    '韶关': (24.801322, 113.591544),
    '杭州': (30.287459, 120.153576),
    '苏州': (31.299379, 120.619585),
    '无锡': (31.574729, 120.301663),
    '上海': (31.231706, 121.472644),
    '丽水': (28.451993, 119.921786),
    '福州': (26.075302, 119.306239),
    '厦门': (24.490474, 118.11022),
    '抚州': (27.98385, 116.358351),
    '武汉': (30.584355, 114.298572),
    '南京': (32.041544, 118.767413),
    '北京': (39.904989, 116.405285),
    '济南': (36.675807, 117.000923),
    '西安': (34.263161, 108.948024),
    '成都': (30.659462, 104.065735),
    '重庆': (29.533155, 106.504962),
    '珠海': (22.255899, 113.552724),
    '广州': (23.125178, 113.280637),
}

# 获取城市的经纬度
lats, lons = zip(*cities.values())

# 创建一个新的图形
fig, ax = plt.subplots(figsize=(10, 10), subplot_kw={'projection': ccrs.PlateCarree()})

# 添加中国地图的特征
ax.add_feature(cfeature.COASTLINE)
```

```
ax.add_feature(cfeature.BORDERS)
ax.add_feature(cfeature.LAND, edgecolor='black')
ax.add_feature(cfeature.OCEAN)
ax.add_feature(cfeature.LAKES, edgecolor='black')

# 设置显示区域以集中在中国
ax.set_extent([73, 135, 18, 54], crs=ccrs.PlateCarree())

# 绘制城市位置
ax.scatter(lons, lats, color='red', s=20, transform=ccrs.PlateCarree())  # 使用 scatter 绘制点
for city, (lat, lon) in cities.items():
    ax.text(lon, lat, city, fontsize=12, ha='right', transform=ccrs.PlateCarree())

# 连接点的顺序
ax.plot(lons, lats, linestyle='--', marker='o', color='red', transform=ccrs.PlateCarree())

# 添加标题
plt.title('最佳路径')

# 显示图形
plt.show()
```

问题三 贪婪算法.py

```
# -*- coding: utf-8 -*-

import numpy as np

city_names = [
    "上海", "北京", "深圳", "广州", "苏州", "成都", "南京", "无锡", "济南", "重庆",
    "西安", "福州", "厦门", "丽水", "中山", "武汉", "抚州", "佛山", "杭州", "韶关", "珠海"
]
distance_matrix = np.array([
    [0., 5., 7., 7., 2., 13., 4., 2., 6., 11., 8., 5., 6., 3., 9., 4., 10., 12., 3., 16., 10.],
    [5., 0., 9., 11., 5., 8., 5., 4., 5., 7., 6., 9., 11., 6., 11., 11., 14., np.inf, 5., 10., 12.],
    [7., 9., 0., 2., 10., 8., 8., 9., 12., 7., 10., 8., 4., np.inf, 2., 6., 8., 1., 7., 2., 2.],
    [7., 11., 2., 0., 25., 7., 7., 9., 8., 6., 8., 5., 4., 22., 1., 12., 9., 1., 8., 1., 2.],
    [2., 5., 10., 25., 0., 13., 3., 1., 5., 11., 7., 7., 9., 4., np.inf, 5., np.inf, np.inf, 2., 19., np.inf],
    [13., 8., 8., 7., 13., 0., 11., 13., 9., 2., 4., 13., 14., 13., 11., 10., 10., 10., 13., np.inf, 11.],
```

```
    [4., 5., 8., 7., 3., 11., 0., 3., 8., 10., 6., 6., np.inf, 4., np.inf, 4., np.inf, np.inf, 2., 8.,
np.inf],
    [2., 4., 9., 9., 1., 13., 3., 0., 9., 12., 9., 8., 9., 4., np.inf, 5., np.inf, np.inf, 3., 9., np.inf],
    [6., 5., 12., 8., 5., 9., 8., 9., 0., 7., 5., 7., 9., 7., np.inf, 6., np.inf, np.inf, 5., 10., np.inf],
    [11., 7., 7., 6., 11., 2., 10., 12., 7., 0., 6., 14., 14., 11., 7., 7., 10., 6., 12., 20., 8.],
    [8., 6., 10., 8., 7., 4., 6., 9., 5., 6., 0., 10., 12., 11., 11., 4., 20., 13., 8., np.inf, np.inf],
    [5., 9., 8., 5., 7., 13., 6., 8., 7., 14., 10., 0., 1., 3., np.inf, 6., 3., np.inf, 5., np.inf, np.inf],
    [6., 11., 4., 4., 9., 14., np.inf, 9., 9., 14., 12., 1., 0., np.inf, np.inf, 8., 5., np.inf, 7., 18.,
np.inf],
    [3., 6., np.inf, 22., 4., 13., 4., 4., 7., 11., 11., 3., np.inf, 0., np.inf, 7., np.inf, np.inf, 2.,
2., np.inf],
    [9., 11., 2., 1., np.inf, 11., np.inf, np.inf, np.inf, 7., 11., np.inf, np.inf, np.inf, 0., 5.,
np.inf, 1., 8., 4.,
     1.],
    [4., 11., 6., 12., 5., 10., 4., 5., 6., 7., 4., 6., 8., 7., 5., 0., 4., np.inf, 5., np.inf, 5.],
    [10., 14., 8., 9., np.inf, 10., np.inf, np.inf, np.inf, 10., 20., 3., 5., np.inf, np.inf, 4., 0.,
np.inf, 9.,
     np.inf, np.inf],
    [12., np.inf, 1., 1., np.inf, 10., np.inf, np.inf, np.inf, 6., 13., np.inf, np.inf, np.inf, 1.,
np.inf, np.inf, 0.,
     11., 8., 2.],
    [3., 5., 7., 8., 2., 13., 2., 3., 5., 12., 8., 5., 7., 2., 8., 5., 9., 11., 0., 0., 9.],
    [16., 10., 2., 1., 19., np.inf, 8., 9., 10., 20., np.inf, np.inf, 18., 2., 4., np.inf, np.inf, 8.,
0., 0., 2.],
    [10., 12., 2., 2., np.inf, 11., np.inf, np.inf, np.inf, 8., np.inf, np.inf, np.inf, np.inf, 1.,
5., np.inf, 2., 9.,
     2., 0.]
])

# 替换 inf 为一个很大的数字
inf = 1e9


distance_matrix = np.where(np.isinf(distance_matrix), inf, distance_matrix)

def greedy_tsp(start):
    n = len(distance_matrix)
    visited = [False] * n
    path = [start]
    total_distance = 0
    visited[start] = True
    current = start

    while len(path) < n:
```

```
            nearest = None
            min_dist = inf
            for i in range(n):
                if not visited[i] and distance_matrix[current][i] < min_dist:
                    nearest = i
                    min_dist = distance_matrix[current][i]
            path.append(nearest)
            visited[nearest] = True
            total_distance += min_dist
            current = nearest

        total_distance += distance_matrix[current][start]
        path.append(start)
        return path, total_distance

# 从广州出发（索引为3）
start_city = 3
path, total_distance = greedy_tsp(start_city)

print(f"从广州出发，游玩所有城市的总距离为: {total_distance:.2f} 小时")
print("最短路径为:", " -> ".join(f"{city_names[city]}" for city in path))
```

问题四 可视化.py

```
# -*- coding: utf-8 -*-

import cartopy.feature as cfeature
import matplotlib.pyplot as plt
import cartopy.crs as ccrs

plt.rcParams['font.family'] = 'SimHei'
plt.rcParams['axes.unicode_minus'] = False

# 城市及其坐标
cities = {
    '广州': (23.125178, 113.280637),
    '珠海': (22.255899, 113.552724),
    '中山': (22.521113, 113.382391),
    '佛山': (23.028762, 113.122717),
    '深圳': (22.547, 114.085947),
    '韶关': (24.801322, 113.591544),
    '武汉': (30.584355, 114.298572),
    '抚州': (27.98385, 116.358351),
```

```
    '厦门': (24.490474, 118.11022),
    '福州': (26.075302, 119.306239),
    '丽水': (28.451993, 119.921786),
    '上海': (31.231706, 121.472644),
}

# 获取城市的经纬度
lats, lons = zip(*cities.values())

# 创建一个新的图形
fig, ax = plt.subplots(figsize=(10, 10), subplot_kw={'projection': ccrs.PlateCarree()})

# 添加中国地图的特征
ax.add_feature(cfeature.COASTLINE)
ax.add_feature(cfeature.BORDERS)
ax.add_feature(cfeature.LAND, edgecolor='black')
ax.add_feature(cfeature.OCEAN)
ax.add_feature(cfeature.LAKES, edgecolor='black')

# 设置显示区域以集中在中国
ax.set_extent([73, 135, 18, 54], crs=ccrs.PlateCarree())

# 绘制城市位置
ax.scatter(lons, lats, color='red', s=20, transform=ccrs.PlateCarree())  # 使用 scatter 绘制点
for city, (lat, lon) in cities.items():
    ax.text(lon, lat, city, fontsize=12, ha='right', transform=ccrs.PlateCarree())

# 连接点的顺序
ax.plot(lons, lats, linestyle='--', marker='o', color='red', transform=ccrs.PlateCarree())

# 添加标题
plt.title('游玩路径')

# 显示图形
plt.show()
```

问题四 遗传算法.py

```
# -*- coding: utf-8 -*-

import numpy as np
import random
```

```
# 城市名称
city_names = [
    "上海", "北京", "深圳", "广州", "苏州", "成都", "南京", "无锡", "济南", "重庆",
    "西安", "福州", "厦门", "丽水", "中山", "武汉", "抚州", "佛山", "杭州", "韶关", "
珠海"
]

# 价格矩阵和时间矩阵（示例数据）

time_matrix = np.array([
    [0., 5., 7., 7., 2., 13., 4., 2., 6., 11., 8., 5., 6., 3., 9., 4., 10., 12., 3., 16., 10.],
    [5., 0., 9., 11., 5., 8., 5., 4., 5., 7., 6., 9., 11., 6., 11., 11., 14., np.inf, 5., 10., 12.],
    [7., 9., 0., 2., 10., 8., 8., 9., 12., 7., 10., 8., 4., np.inf, 2., 6., 8., 1., 7., 2., 2.],
    [7., 11., 2., 0., 25., 7., 7., 9., 8., 6., 8., 5., 4., 22., 1., 12., 9., 1., 8., 1., 2.],
    [2., 5., 10., 25., 0., 13., 3., 1., 5., 11., 7., 7., 9., 4., np.inf, 5., np.inf, np.inf, 2., 19.,
np.inf],
    [13., 8., 8., 7., 13., 0., 11., 13., 9., 2., 4., 13., 14., 13., 11., 10., 10., 10., 13., np.inf,
11.],
    [4., 5., 8., 7., 3., 11., 0., 3., 8., 10., 6., 6., np.inf, 4., np.inf, 4., np.inf, np.inf, 2., 8.,
np.inf],
    [2., 4., 9., 9., 1., 13., 3., 0., 9., 12., 9., 8., 9., 4., np.inf, 5., np.inf, np.inf, 3., 9., np.inf],
    [6., 5., 12., 8., 5., 9., 8., 9., 0., 7., 5., 7., 9., 7., np.inf, 6., np.inf, np.inf, 5., 10., np.inf],
    [11., 7., 7., 6., 11., 2., 10., 12., 7., 0., 6., 14., 14., 11., 7., 7., 10., 6., 12., 20., 8.],
    [8., 6., 10., 8., 7., 4., 6., 9., 5., 6., 0., 10., 12., 11., 11., 4., 20., 13., 8., 9., np.inf],
    [5., 9., 8., 5., 7., 13., 6., 8., 7., 14., 10., 0., 1., 3., np.inf, 6., 3., np.inf, 5., np.inf, np.inf],
    [6., 11., 4., 4., 9., 14., np.inf, 9., 9., 14., 12., 1., 0., np.inf, np.inf, 8., 5., np.inf, 7.,
np.inf, np.inf],
    [3., 6., np.inf, 22., 4., 13., 4., 4., 7., 11., 11., 3., np.inf, 0., np.inf, 7., np.inf, np.inf, 2.,
18., np.inf],
    [9., 11., 2., 1., np.inf, 11., np.inf, np.inf, np.inf, 7., 11., np.inf, np.inf, np.inf, 0., 5.,
np.inf, 1., 8., 2., 1.],
    [4., 11., 6., 12., 5., 10., 4., 5., 6., 7., 4., 6., 8., 7., 5., 0., 4., np.inf, 5., 4., 5.],
    [10., 14., 8., 9., np.inf, 10., np.inf, np.inf, np.inf, 10., 20., 3., 5., np.inf, np.inf, 4., 0.,
np.inf, 9., np.inf, np.inf],
    [12., np.inf, 1., 1., np.inf, 10., np.inf, np.inf, np.inf, 6., 13., np.inf, np.inf, np.inf, 1.,
np.inf, np.inf, 0., 11., np.inf, 2.],
    [3., 5., 7., 8., 2., 13., 2., 3., 5., 12., 8., 5., 7., 2., 8., 5., 9., 11., 0., 8., 9.],
    [16., 10., 2., 1., 19., np.inf, 8., 9., 10., 20., 9., np.inf, np.inf, 18., 2., 4., np.inf, np.inf,
8., 0., 2.],
    [10., 12., 2., 2., np.inf, 11., np.inf, np.inf, np.inf, 8., np.inf, np.inf, np.inf, np.inf, 1.,
5., np.inf, 2., 9., 2., 0.]
])
```

```
price_matrix = np.array([
    [0., 700., 900., 900., 20., 700., 50., 20., 500., 600., 700., 400., 600., 200., 900., 400.,
500., 700., 30., 200., 900.],
    [700., 0., 1100., 900., 700., 800., 500., 600., 200., 900., 600., 900., 1000., 800., 900.,
200., 400., np.inf, 700., 800., 800.],
    [900., 1100., 0., 80., 900., 800., 800., 900., 1000., 600., 900., 400., 300., np.inf, 400.,
500., 400., 100., 800., 600., 200.],
    [900., 900., 80., 0., 200., 700., 700., 800., 900., 500., 800., 500., 400., 200., 40., 200.,
500., 20., 800., 300., 70.],
    [20., 700., 900, 200., 0., 1200., 40., 10., 400., 900., 700., 400., 500., 300., np.inf, 300.,
np.inf, np.inf, 200., 200., np.inf],
    [700., 800., 800., 700., 1200., 0., 900., 700., 800., 200., 300., 1000., 1000., 1000.,
700., 400., 800., 600., 1000., np.inf, 700.],
    [50., 500., 800., 700., 40., 900., 0., 30., 100., 500., 600., 500., np.inf, 250., np.inf,
200., np.inf, np.inf, 200., 1000., np.inf],
    [20., 600., 900., 800., 10., 700., 30., 0., 200., 600., 800., 400., 500., 300., np.inf, 300.,
np.inf, np.inf, 200., 1100., np.inf],
    [500., 200., 1000., 900., 400., 800., 100., 200., 0., 800., 500., 700., 800., 600., np.inf,
600., np.inf, np.inf, 500., 900., np.inf],
    [600., 900., 600., 500., 900., 200., 500., 600., 800., 0., 300., 600., 700., 800., 600.,
300., 500., 600., 600., 700., 600.],
    [700., 600., 900., 800., 700., 300., 600., 800., 500., 300., 0., 900., 1000., 800., 800.,
500., 200., 900., 800., np.inf, np.inf],
    [400., 900., 400., 500., 400., 1000., 500., 400., 700., 600., 900., 0., 200., 200., np.inf,
400., 200., np.inf, 300., np.inf, np.inf],
    [600., 1000., 300., 400., 500., 1000., np.inf, 500., 800., 700., 1000., 200., 0., np.inf,
np.inf, 500., 200., np.inf, 400., 200., np.inf],
    [200., 800., np.inf, 200., 300., 1000., 250., 300., 600., 800., 800., 200., np.inf, 0.,
np.inf, 400., np.inf, np.inf, 100., 200., np.inf],
    [900., 900., 400., 40., np.inf, 700., np.inf, np.inf, np.inf, 600., 800., np.inf, np.inf,
np.inf, 0., 600., np.inf, 60., 800., 400., 30.],
    [400., 200., 500., 200., 300., 400., 200., 300., 600., 300., 500., 400., 500., 400., 600.,
0., 200., np.inf, 400., np.inf, 700.],
    [500., 400., 400., 500., np.inf, 800., np.inf, np.inf, np.inf, 500., 200., 200., 200., np.inf,
np.inf, 200., 0., np.inf, 800., np.inf, np.inf],
    [700., np.inf, 100., 20., np.inf, 600., np.inf, np.inf, np.inf, 600., 900., np.inf, np.inf,
np.inf, 60., np.inf, np.inf, 0., 11., 700., 100.],
    [30., 700., 800., 800., 200., 1000., 200., 200., 500., 600., 800., 300., 400., 100., 800.,
400., 400., 800., 0., 700., 900.],
    [200., 800., 600., 300., 200., np.inf, 1000., 1100., 900., 200., 700, np.inf, np.inf, 200.,
200., 400., np.inf, np.inf, 700., 0., 200.],
    [900., 800., 200., 70., np.inf, 700., np.inf, np.inf, np.inf, 600., np.inf, np.inf, np.inf,
```

```
np.inf, 30., 700., np.inf, 100., 900., 200., 0.]
])
inf = 1e9
# 从广州的索引
start_city = 3


def calculate_path_cost(path, matrix):
    cost = 0
    for i in range(len(path) - 1):
        cost += matrix[path[i], path[i + 1]]
    return cost


def create_initial_population(size, num_cities):
    population = []
    for _ in range(size):
        individual = list(range(num_cities))
        individual.remove(start_city)
        random.shuffle(individual)
        individual.insert(0, start_city)
        population.append(individual)
    return population


def mutate(individual):
    idx1, idx2 = random.sample(range(1, len(individual)), 2)  # 不变的起始城市
    individual[idx1], individual[idx2] = individual[idx2], individual[idx1]


def crossover(parent1, parent2):
    idx1, idx2 = sorted(random.sample(range(1, len(parent1)), 2))  # 不变的起始城市
    child = [-1] * len(parent1)
    child[idx1:idx2 + 1] = parent1[idx1:idx2 + 1]
    fill = [city for city in parent2 if city not in child]
    pos = (idx2 + 1) % len(parent1)
    for city in fill:
        if child[pos] == -1:
            child[pos] = city
        else:
            pos = (pos + 1) % len(parent1)
            child[pos] = city
    return child
```

```
def genetic_algorithm(price_matrix, time_matrix, population_size=50, generations=1000, mutation_rate=0.2):
    num_cities = len(price_matrix)
    population = create_initial_population(population_size, num_cities)

    for generation in range(generations):
        population = sorted(population, key=lambda x: calculate_path_cost(x, time_matrix))
        new_population = population[:2]  # Elitism: carry the best solutions forward

        while len(new_population) < population_size:
            parent1, parent2 = random.choices(population[:10], k=2)  # Tournament selection
            child = crossover(parent1, parent2)
            if random.random() < mutation_rate:
                mutate(child)
            new_population.append(child)

        population = new_population

    best_path = min(population, key=lambda x: calculate_path_cost(x, time_matrix))
    best_time = calculate_path_cost(best_path, time_matrix)
    best_cost = calculate_path_cost(best_path, price_matrix)
    return best_path, best_time, best_cost


best_path, best_time, best_cost = genetic_algorithm(price_matrix, time_matrix)
path_names = [city_names[i] for i in best_path]
print("最佳路径:", " -> ".join(path_names))
print("最短时间:", best_time)
print("最低花费:", best_cost)
```

问题五 加权路径优化.py

```
# -*- coding: utf-8 -*-

import numpy as np
import itertools

# 所有城市
all_cities = [
    "黄山市", "九江市", "温州市", "衡阳市", "郑州市",
```

```
    "韶关市", "成都市", "洛阳市", "萍乡市", "南平市",
    "西安市", "宝鸡市"
]

# 所有城市的价格和时间矩阵
all_price_matrix = np.array([
    [0, 9999, 300, 9999, 500, 9999, 900, 500, 500, 200, 700, 800],
    [9999, 0, 300, 90, 400, 200, 500, 400, 200, 200, 600, 200],
    [300, 300, 0, 200, 600, 200, 1100, 700, 400, 9999, 900, 300],
    [9999, 90, 200, 0, 200, 200, 9999, 900, 200, 9999, 700, 700],
    [500, 400, 600, 200, 0, 600, 500, 70, 500, 500, 300, 200],
    [9999, 200, 200, 200, 600, 0, 9999, 700, 300, 9999, 700, 300],
    [900, 500, 1100, 9999, 500, 9999, 0, 500, 700, 900, 300, 9999],
    [500, 400, 700, 900, 70, 700, 500, 0, 9999, 700, 200, 80],
    [500, 200, 400, 200, 500, 300, 700, 9999, 0, 300, 9999, 9999],
    [200, 200, 9999, 9999, 500, 9999, 900, 700, 300, 0, 800, 900],
    [700, 600, 900, 700, 300, 700, 300, 200, 9999, 800, 0, 30],
    [800, 200, 300, 700, 200, 300, 9999, 80, 9999, 900, 30, 0]
])

all_time_matrix = np.array([
    [0, 9999, 4, 9999, 6, 9999, 11, 6, 4, 2, 7, 9],
    [9999, 0, 5, 10, 5, 11, 12, 6, 6, 3, 7, 22],
    [4, 5, 0, 19, 8, 23, 14, 11, 5, 9999, 12, 33],
    [9999, 10, 19, 0, 11, 2, 9999, 6, 2, 9999, 8, 9],
    [6, 5, 8, 11, 0, 6, 7, 1, 5, 7, 2, 8],
    [9999, 11, 23, 2, 6, 0, 9999, 7, 3, 9999, 8, 28],
    [11, 12, 14, 9999, 7, 9999, 0, 5, 8, 11, 4, 9999],
    [6, 6, 11, 6, 1, 7, 5, 0, 9999, 8, 2, 8],
    [4, 6, 5, 2, 5, 3, 8, 9999, 0, 4, 9999, 9999],
    [2, 3, 9999, 9999, 7, 9999, 11, 8, 4, 0, 9, 10],
    [7, 7, 12, 8, 2, 8, 4, 2, 9999, 9, 0, 2],
    [9, 22, 33, 9, 8, 28, 9999, 8, 9999, 10, 2, 0]
])

# 所有城市的索引映射
city_to_index = {city: idx for idx, city in enumerate(all_cities)}
index_to_city = {idx: city for city, idx in city_to_index.items()}


# 从剩余的 10 个城市中选择 8 个城市
def update_matrices(indices, price_matrix, time_matrix):
    sub_price_matrix = price_matrix[np.ix_(indices, indices)]
    sub_time_matrix = time_matrix[np.ix_(indices, indices)]
```

```
    return sub_price_matrix, sub_time_matrix


def calculate_weighted_score(path, price_matrix, time_matrix, price_weight, time_weight):
    total_price = 0
    total_time = 0
    for i in range(len(path) - 1):
        total_price += price_matrix[path[i], path[i + 1]]
        total_time += time_matrix[path[i], path[i + 1]]
    return price_weight * total_price + time_weight * total_time


def find_optimal_path(start_city, end_city, cities, price_matrix, time_matrix, price_weight, time_weight):
    city_indices = list(range(len(cities)))
    start_index = cities.index(start_city)
    end_index = cities.index(end_city)

    best_score = float('inf')
    best_path = []

    for path in itertools.permutations(city_indices):
        if path[0] == start_index and path[-1] == end_index:
            score = calculate_weighted_score(path, price_matrix, time_matrix, price_weight, time_weight)
            if score < best_score:
                best_score = score
                best_path = path

    return best_path, best_score


# 使用成都市和温州市作为起点和终点
start_city = "成都市"
end_city = "温州市"

remaining_cities = [city for city in all_cities if city != start_city and city != end_city]
best_path = None
best_score = float('inf')

for combination in itertools.combinations(remaining_cities, 8):
    selected_cities = [start_city] + list(combination) + [end_city]
    selected_indices = [city_to_index[city] for city in selected_cities]
```

```
    price_matrix_8, time_matrix_8 = update_matrices(selected_indices, all_price_matrix, all_time_matrix)
    path_indices, score = find_optimal_path(start_city, end_city, selected_cities, price_matrix_8, time_matrix_8, 0.6,
                                            0.4)

    if score < best_score:
        best_score = score
        best_path = [selected_cities[i] for i in path_indices]

print(f"最优路径:", " -> ".join(best_path))
```

问题五 可视化.py

```
# -*- coding: utf-8 -*-

import cartopy.feature as cfeature
import matplotlib.pyplot as plt
import cartopy.crs as ccrs

plt.rcParams['font.family'] = 'SimHei'
plt.rcParams['axes.unicode_minus'] = False

# 城市及其坐标
cities = {
    '雁荡山': (28.000575, 120.672111),
    '黄山': (29.709239, 118.317325),
    '武夷山': (26.635627, 118.178459),
    '庐山': (29.712034, 115.992811),
    '衡山': (26.900358, 112.607693),
    '武功山': (27.622946, 113.852186),
    '老君山': (34.663041, 112.434468),
    '太白山': (34.369315, 107.14487),
    '钟南山': (34.263161, 108.948024),
    '青城山': (30.659462, 104.065735),
}

# 获取城市的经纬度
lats, lons = zip(*cities.values())

# 创建一个新的图形
fig, ax = plt.subplots(figsize=(10, 10), subplot_kw={'projection': ccrs.PlateCarree()})
```

```
# 添加中国地图的特征
ax.add_feature(cfeature.COASTLINE)
ax.add_feature(cfeature.BORDERS)
ax.add_feature(cfeature.LAND, edgecolor='black')
ax.add_feature(cfeature.OCEAN)
ax.add_feature(cfeature.LAKES, edgecolor='black')

# 设置显示区域以集中在中国
ax.set_extent([73, 135, 18, 54], crs=ccrs.PlateCarree())

# 绘制城市位置
ax.scatter(lons, lats, color='black', s=20, transform=ccrs.PlateCarree())  # 使用 scatter 绘制点
for city, (lat, lon) in cities.items():
    ax.text(lon, lat, city, fontsize=12, ha='right', transform=ccrs.PlateCarree())

# 连接点的顺序
ax.plot(lons, lats, linestyle='--', marker='o', color='red', transform=ccrs.PlateCarree())

# 添加标题
plt.title('游览路径')

# 显示图形
plt.show()
```

问题五 标准化.py

```
from sklearn.preprocessing import MinMaxScaler
import numpy as np
import itertools

# 所有城市
all_cities = [
    "黄山市", "九江市", "温州市", "衡阳市", "郑州市",
    "韶关市", "成都市", "洛阳市", "萍乡市", "南平市",
    "西安市", "宝鸡市"
]

# 所有城市的价格和时间矩阵
all_price_matrix = np.array([
    [0, 9999, 300, 9999, 500, 9999, 900, 500, 500, 200, 700, 800],
```

```
    [9999, 0, 300, 90, 400, 200, 500, 400, 200, 200, 600, 200],
    [300, 300, 0, 200, 600, 200, 1100, 700, 400, 9999, 900, 300],
    [9999, 90, 200, 0, 200, 200, 9999, 900, 200, 9999, 700, 700],
    [500, 400, 600, 200, 0, 600, 500, 70, 500, 500, 300, 200],
    [9999, 200, 200, 200, 600, 0, 9999, 700, 300, 9999, 700, 300],
    [900, 500, 1100, 9999, 500, 9999, 0, 500, 700, 900, 300, 9999],
    [500, 400, 700, 900, 70, 700, 500, 0, 9999, 700, 200, 80],
    [500, 200, 400, 200, 500, 300, 700, 9999, 0, 300, 9999, 9999],
    [200, 200, 9999, 9999, 500, 9999, 900, 700, 300, 0, 800, 900],
    [700, 600, 900, 700, 300, 700, 300, 200, 9999, 800, 0, 30],
    [800, 200, 300, 700, 200, 300, 9999, 80, 9999, 900, 30, 0]
])

all_time_matrix = np.array([
    [0, 9999, 4, 9999, 6, 9999, 11, 6, 4, 2, 7, 9],
    [9999, 0, 5, 10, 5, 11, 12, 6, 6, 3, 7, 22],
    [4, 5, 0, 19, 8, 23, 14, 11, 5, 9999, 12, 33],
    [9999, 10, 19, 0, 11, 2, 9999, 6, 2, 9999, 8, 9],
    [6, 5, 8, 11, 0, 6, 7, 1, 5, 7, 2, 8],
    [9999, 11, 23, 2, 6, 0, 9999, 7, 3, 9999, 8, 28],
    [11, 12, 14, 9999, 7, 9999, 0, 5, 8, 11, 4, 9999],
    [6, 6, 11, 6, 1, 7, 5, 0, 9999, 8, 2, 8],
    [4, 6, 5, 2, 5, 3, 8, 9999, 0, 4, 9999, 9999],
    [2, 3, 9999, 9999, 7, 9999, 11, 8, 4, 0, 9, 10],
    [7, 7, 12, 8, 2, 8, 4, 2, 9999, 9, 0, 2],
    [9, 22, 33, 9, 8, 28, 9999, 8, 9999, 10, 2, 0]
])


def normalize_matrix(matrix):
    # 替换 9999 为 NaN
    matrix_with_nan = np.where(matrix == 9999, np.nan, matrix)

    # 创建 MinMaxScaler 实例
    scaler = MinMaxScaler()

    # 只对非 NaN 值进行归一化
    matrix_flat = matrix_with_nan.flatten()
    nan_mask = np.isnan(matrix_flat)

    # 对非 NaN 值进行归一化
    matrix_flat[~nan_mask] = scaler.fit_transform(matrix_flat[~nan_mask].reshape(-1,
1)).flatten()
```

```
    # 将归一化结果重新放回到矩阵中
    matrix_normalized = matrix_flat.reshape(matrix.shape)

    # 重新将 NaN 值设回 9999
    matrix_normalized = np.where(np.isnan(matrix_with_nan), 9999, matrix_normalized)

    return matrix_normalized


# 对价格矩阵和时间矩阵进行归一化处理
normalized_price = normalize_matrix(all_price_matrix)
normalized_time = normalize_matrix(all_time_matrix)

# 所有城市的索引映射
city_to_index = {city: idx for idx, city in enumerate(all_cities)}
index_to_city = {idx: city for city, idx in city_to_index.items()}


def update_matrices(indices, price_matrix, time_matrix):
    sub_price_matrix = price_matrix[np.ix_(indices, indices)]
    sub_time_matrix = time_matrix[np.ix_(indices, indices)]
    return sub_price_matrix, sub_time_matrix


def calculate_weighted_score(path, price_matrix, time_matrix, price_weight, time_weight):
    total_price = 0
    total_time = 0
    for i in range(len(path) - 1):
        total_price += price_matrix[path[i], path[i + 1]]
        total_time += time_matrix[path[i], path[i + 1]]
    return price_weight * total_price + time_weight * total_time


def find_optimal_path(start_city, end_city, cities, price_matrix, time_matrix, price_weight, time_weight):
    city_indices = list(range(len(cities)))
    start_index = cities.index(start_city)
    end_index = cities.index(end_city)

    best_score = float('inf')
    best_path = []
```

```
    for path in itertools.permutations(city_indices):
        if path[0] == start_index and path[-1] == end_index:
            score = calculate_weighted_score(path, price_matrix, time_matrix, price_weight, time_weight)
            if score < best_score:
                best_score = score
                best_path = path

    return best_path, best_score


# 使用成都市和温州市作为起点和终点
start_city = "温州市"
end_city = "成都市"

# 尝试所有从剩余城市中选择 8 个城市的组合
remaining_cities = [city for city in all_cities if city != start_city and city != end_city]
best_path = None
best_score = float('inf')

for combination in itertools.combinations(remaining_cities, 6):
    selected_cities = [start_city] + list(combination) + [end_city]
    selected_indices = [city_to_index[city] for city in selected_cities]

    price_matrix_8, time_matrix_8 = update_matrices(selected_indices, normalized_price, normalized_time)
    path_indices, score = find_optimal_path(start_city, end_city, selected_cities, price_matrix_8, time_matrix_8, 0.6,
                                            0.4)

    if score < best_score:
        best_score = score
        best_path = [selected_cities[i] for i in path_indices]

print("最优路径:", " -> ".join(best_path))
```

代码均符合 PEP8 规范，(>_<)。