重庆大学

数学建模校内竞赛论文

论文题目：

组号：

成员：

选题：

姓名	学院	年级	专业	学号	联系电话	数学分析	高等代数	微积分	高等数学	线性代数	概率统计	数学实验	数学模型	CET4	CET6

日期

基于多目标优化的旅游线路规划研究

摘要

重庆作为一座旅游热点城市，凭借其独特的地理环境和丰富的文化景观吸引了大量游客。然而，城市的山河交错与组团式城市布局导致景点分布分散、交通不便，给游客的旅游体验带来了挑战。为了优化旅游路线，减少交通时间并避免景点过度拥挤，提升游客满意度，本文通过数学建模，提出了针对重庆旅游线路规划的优化方案。

针对问题一，我们首先利用 Floyd-Warshall 算法补全景点—区域交通网络，生成任意节点对的最短路矩阵，由此精确计算交通时间 $t(S_i, R_j)$ 与费用 $d(S_i, R_j)$。在此基础上，将景点基准吸引力、游客对餐饮/住宿区域的偏好 P_{R_j} 与交通成本共同纳入满意度模型，构造 Pref(p)、Cost(p)、Time(p) 等函数，并归一化形成综合效用 u_p。通过枚举一日、二日、三日游所有可行路线，引入 MILP 处理景点、午餐及夜宿容量约束，再结合遗传算法对解空间进行多样化搜索，获得最短时间、最低费用及均衡型等多类别最优方案。最终输出 TODO 条覆盖不同天数、不同兴趣偏好的高满意度套餐。

针对问题二，问题一中生成的旅游方案存在着数量较多、部分方案相似度较高等问题。因此第二问针对这些问题，提出基于旅游方案相似度的加权距离函数建模，并采用 K-means++ 聚类，将套餐数压缩至 10 种以内且总体方案满意度较高。加权距离函数主要由景点吸引力、区域吸引力、路线相似度、交通时间、交通费用、游客偏好等因素构成。算法会尝试不同 K 值，并根据簇内平方和选择最优的 K 值作为最终的结果。我们将每个簇中满意度最高的方案保留，最终输出满足数量上限且整体满意度最优的方案集合。

针对问题三，我们建立了扩容收益、建设成本等因素，选择最合适的区域以及扩容规模。我们首先根据题目背景，引入扩容收益、建设成本等因素，得到净收益函数。然后，我们采用双层优化策略，外层枚举六个候选区域及一系列离散的扩容步长，内层优化在给定 $(r, \Delta K)$ 时，调用问题一中“枚举路线 + MILP”框架（辅以遗传算法与启发式多样化规则）重新分配游客，得到 $Z_r(K), N_r(K)$。最终，我们绘制 $F_r(\Delta K)$ 随扩容规模变化的曲线，并给出最优扩容区域及规模。

关键词：MILP；遗传算法；旅游线路规划；K 均值聚类

1 问题重述

1.1 问题背景

重庆作为著名的网红城市，前来旅游的乘客数量极大。由于重庆山河交错的地理环境、组团式城市形态的客观原因，景点多但是分布分散且单一景点资源有限。此外重庆主城区域面积有限但接收大量游客。如何通过规划旅游路线减少交通时间，防止景点人员拥挤以提升游客旅游体验是一个具有实际意义的优化问题。

根据题目提供的相关数据可知：共有 6 个景点并且景点可接纳人数在 1.2 万人到 4.2 万人之间；景点附近有 6 个主要区域并为游客提供午餐、晚餐和住宿，其接待能力和游客喜好程度可见表 2。游客会在上午、下午游览景点，中午到附近区域用餐，若时间超过一天则选择一个区域食宿。题目还给出了具体的从景点到各区域的交通费用和时间。对此，我们需要建立模型解决以下问题：

1.2 问题重述

为了为游客提供更好的方案选择，并且提高每个方案的旅游体验。我们需要综合考虑以上所有信息，并注意一些基本原则：

问题一：鉴于我国旅游业发展迅猛，常有高峰期景点容量难以满足游客人数等情况。我们在设计方案时必须考虑到游客的拥挤感知。[1] 因此我们需要考虑景点和附近区域的接待能力，保证同一时间在任意景点或区域的总人数不会超过其接纳上线。并且要考虑到交通费用与时间，使游客能够尽量节省时间与金钱。在此要求下，我们根据不同旅游时间需要提出景区吸引力函数，结合约束条件设计出合适的“景点 + 区域”组合方案。

问题二：过多的旅游方案会导致更大的路线管理压力以及更低的游客决策效率。我们需要将问题一中的套餐总数控制在 10 以内。由此，我们要建立方案筛选与整合准则，对原方案进行精简与再优化，将重复性较强的方案合并，最终输出满足数量上限且整体满意度最优的套餐集合。

问题三：我们需要考虑区域的接纳人数、游客喜好程度等信息，尽可能为旅游集团提出效益更高的建设方案。我们需要提出一种评价函数结合扩容收益、建设成本等因素，选择最合适的区域以及扩容规模。

2 问题分析

2.1 问题一

针对问题一，我们首先需要将景点吸引力通过合适的数学函数进行量化，该函数需要综合景点可达区域、区域偏好、可容纳人数等影响。以此作为优化目标函数的一个因子。此外我们还需要考虑到不同景点、区域选择会带来不同的时间、交通成本，应当为其增加约束条件并作为优化函数的另一个因子。由于不同旅游天数会对问题产生较大的影响。一日游在晚上不必前往旅馆，而二日游、三日游都需要前往旅馆。因此我们需要额外考虑不同天数下的交通问题。

旅游路线在旅游方案中占有重要地位，对于促进旅游区域的可持续发展、提升游客体验具有重要作用。[2]。因此我们有必要将其补全。对于表 3 中空白的区域，我们假设可以通过中转的方式到达。对此，我们应当采用 Floyd-Warshall 算法计算不可直达的两景区之间的最短距离。在此基础上，采用混合整数线性优化方法得到最优方案。

为了更好得规划旅游方案，我们有必要根据景点信息提出一种多层次吸引力评价指标。[3] 综合以上信息，我们提出一种结合景点吸引力、游客偏好、时间与空间约束的旅游路线规划模型。

2.2 问题二

在问题二中，当旅游套餐总数被限定在 10 种以内时，原先追求满意度极大化的模型必须在游客体验与管理简化之间做出权衡。为此 1，我们提出了基于旅游路线的 K-means++ 算法。该算法采用多种因素加权的距离函数判断两个旅游路线之间的相似度。该函数综合考虑到方案间的景点、路线、区域等相似度，将重复性较强的方案合并。

通过这种方法，我们可以得到 5-6 个聚类中心，而每个聚类中心包含的方案数量在 2-5 个之间。再根据满意度排序，舍弃满意度较低的几个方案，最终将总数控制到 10 以内。通过 K 均值算法，我们可以很好地保证方案总体满意度较高的同时，满足多样化需求，满足方案总数的限制。

2.3 问题三

在问题三中，我们的目标是为旅游集团确定最优的**住宿扩容区域以及扩容规模**，以在满足游客需求的同时最大化经济效益。与问题一、二侧重于既有容量条件下的线路设计不同，问题三需要在**提升接待能力**与**追加建设成本**之间进行权衡。

首先，根据**模型假设**（见 3），景区夜宿容量可以通过追加投资进行扩容，且扩容成本呈现规模报酬递减特征。结合题目背景，我们应当引入如下两类关键因素：扩容收益与扩容成本。扩容收益由扩容后游客满意度增量与额外可接待的游客数量增量组成。扩容成本采用经验函数 $C(\Delta K) = c_1(\Delta K)^\gamma$，体现越大规模扩建边际成本递增的现实规律。

为了求解最优 $(r^*, \Delta K^*)$，我们可以采用**双层优化策略**：

- **外层枚举** 六个候选区域及一系列离散的扩容步长 $\Delta\mathcal{K}$（如 0, 0.5, 1, …）。
- **内层优化** 在给定 $(r, \Delta K)$ 时，调用问题一中“枚举路线 + MILP”框架（辅以遗传算法与启发式多样化规则）重新分配游客，得到 $Z_r(K)$, $N_r(K)$。

将外层搜索与内层分配结合，即可计算所有候选方案的净收益 $F_r(\Delta K)$，从而选出

$$(r^*, \Delta K^*) = \arg\max_{r,\,\Delta K} F_r(\Delta K).$$

最后，绘制 $F_r(\Delta K)$ 随扩容规模变化的曲线，并给出最优扩容区域及规模。这不仅为政府和企业提供了量化的建设决策依据，也直观展示了不同投入水平下的成本-收益权衡。

3 模型假设

本文针对重庆市旅游方案规划以及区域扩建规模与选址问题建立的数学模型基于以下核心假设：

假设 1：表 3 中留空即为不可直达。例如，景点六只可直达区域六，但可以通过其他区域中转到达其他景点。

假设 2：从各景点到各区域酒店或餐厅的交通时间固定，不考虑现实中例如交通拥堵、车辆故障等意外情况导致的时间延长。

假设 3：基于现实情况，本文假设从景区到区域的去程、返程交通费用与时间相等。

假设 4：景区容量的约束在现实中不可能取决于一家旅行社的游客，因为还有可能有其他来源的若干游客，为了建模简单，把容量按元素 $\times 0.6$ 进行折减。

假设 5：我们假设一日游、二日游和三日游三种旅行方案相互分离，且三者在时间上并不有重叠。此时，我们只需考虑一种方案不同路线在容量上的冲突情况。

假设 6：我们假设游客只按照对景点的喜好选择套餐，那么各个路线的人数应当假定为均等的而进行优化。

4 符号说明

符号	说明
S_i	第 i 个景点，$i = 1, 2, \ldots, 6$
R_j	第 j 个餐饮 / 住宿区域，$j = 1, 2, \ldots, 6$
C_{S_i}	景点 S_i 容量（万人 / 半天）
L_{R_j}	区域 R_j 午餐接待容量（万人次 / 日中）
H_{R_j}	区域 R_j 晚餐 + 住宿容量（万人次 / 夜）
P_{R_j}	区域 R_j 游客喜好度评分
$d(S_i, R_j)$	景点 S_i 至区域 R_j 交通费用（元）
$t(S_i, R_j)$	景点 S_i 至区域 R_j 交通时间（分钟）
$T_{\max}$	半天可用于交通的最大时间（三种 $T_{\max}$）
$\mathcal{P}_{1,2,3}$	一 / 二 / 三日游套餐候选集合
$\mathcal{P}$	全部可行套餐集合，$\mathcal{P} = \mathcal{P}_1 \cup \mathcal{P}_2 \cup \mathcal{P}_3$
x_p	选择套餐 p 的游客人数（万人）
$\mathrm{Pref}(p)$	套餐 p 的偏好
$w_{\text{cost}}, w_{\text{time}}$	费用与时间权重系数
$\mathrm{Cost}(p), \mathrm{Time}(p)$	套餐 p 总交通费用和时间开销
Q_i	景点 i 基础吸引力评分
U_p	套餐 p 单位游客满意度
$\tilde{C}_i$	容量归一化，$C_i / \max_j C_j$
$\tilde{R}_i$	服务保障度，$\min\big(1, \frac{\min(L_{R_i}, H_{R_i})}{C_i}\big)$
$\tilde{S}_i$	喜好度归一化，$S_i/10$
$\delta_{p,i,t}$	若套餐 p 在时段 t 安排景点 S_i 则为 1，否则 0
$\theta_{p,j,d}$	套餐 p 第 d 天中午选择区域 R_j 用餐的 0-1 变量
$\phi_{p,j,d}$	套餐 p 第 d 天夜晚选择区域 R_j 住宿的 0-1 变量
$\mathrm{Simu}_{p_i,p_j,S}$	套餐 p_i 与 p_j 在景点方面的相似度
$D(p_1, p_2)$	套餐 p_1 与 p_2 的距离
$C(\Delta K)$	扩容成本 $c_1(\Delta K)^\gamma$，$1 < \gamma \leq 1.2$
ΔK	新增接待能力（万人次）
c_1	扩容成本参数，$c_1 \approx 0.0007$ 亿元
γ	扩容指数，$\gamma \approx 1.09$
$F(\Delta K)$	净收益 $B(\Delta K) - C(\Delta K)$
$B(\Delta K)$	收益 $v_s[Z(K) - Z(K_0)] + v_p[N(K) - N(K_0)]$
$Z(K), N(K)$	容量为 K 时的最优满意度 / 游客量
v_s, v_p	单位满意度价值 / 单位游客利润
K^0，K	原容量 / 扩容后容量

5 模型建立与求解

5.1 问题一：路旅游方案设计

5.1.1 模型建立与求解

对于此问题，我们首先通过 Floyd-Warshall 算法补全邻接矩阵，得到任意两点之间的最短距离。这样我们就可以得到任意区域到任意景点的交通费用和交通时间。此外，我们还需要考虑到旅游信息（住宿、美食、景点等）对方案的影响。既要考虑其对方案满意度的增益，也应当考虑到可容纳人数的限制。[4] 以此为基础，我们进行以下建模：

景点收益：由于缺乏具体景点评分数据，我们假设每个景点具有相近的吸引力价值，即此处不加入游客对景点的偏好因素。P_{max} 表示某一类型的方案的最大满意度（即总是选择游客喜好程度最高的区域 3）。

$$\text{Pref}_{max}^{(d_p)}(type) = \begin{cases} 9 \times 1 + 2 = 11, & (1\text{ 日}) \\ 9 \times 3 + 4 = 31, & (2\text{ 日}) \\ 9 \times 5 + 6 = 51, & (3\text{ 日}) \end{cases} \tag{5.1}$$

以上方程为景区最大收益函数。为了减少数值对模型计算的影响，我们对三种收益进行归一化处理。

区域收益：套餐 p 涉及到的区域的用餐以及住宿体验。根据题目数据，我们将区域喜好度评分 P_{R_j} 作为游客在该区域用餐/住宿获得的满意度。如果套餐跨多日，可能涉及多个不同的区域，则将相应区域的偏好分累计。若在同一天中午和晚上都在同一 R_j 区域，则游客在该天对区域 R_j 的体验包括午餐和晚餐 + 住宿，为方便建模并突出问题的重心，我们采用叠加的方式计算区域满意度。

交通的时间与费用成本：每套旅游方案中的交通费用和时间开销会降低满意度。我们采用函数 $Cost(p)$ 来评价一个方案中的所有交通费用，用 $Time(p)$ 来表示所有时间开销。此外我们为其增加了权重 $w_{\text{cost}}, w_{\text{time}}$，以此代表不同套餐的倾向，用以区分经济优先还是时间优先。

$$\text{Cost}(p) = \sum_{(S_i,R_j)\in p} d(S_i, R_j) \tag{5.2}$$

$$\text{Time}(p) = \sum_{(S_i,R_j)\in p} t(S_i, R_j) \tag{5.3}$$

总的某方案 p 的满意度函数如下：

$$u_p = w_{\text{pref}}\frac{\text{Pref}(p)}{\text{Pref}_{\max}^{(d_p)}} - w_{\text{cost}}\frac{\text{Cost}(p)}{\max\{\text{Cost}(q) : q \in \mathcal{P}\}} - w_{\text{time}}\frac{\text{Time}(p)}{\max\{\text{Time}(q) : q \in \mathcal{P}\}}, \tag{5.4}$$

最大化函数：

$$Z = \sum_{p\in\mathcal{P}} u_p \cdot x_p. \tag{5.5}$$

约束条件如下：

1. 景点接待容量限制：各景点在各个半天时段的接待总人数不得超限。由于早午两段和最多 3 天行程共有 6 个半天时段，模型将假设景点容量可在不同时段重复利用。对每个景点 $i \in S$ 有：

$$\sum_{p\in\mathcal{P}} \delta_{i,p} \cdot x_p \le 6\, C_{S_i}, \quad i = 1, 2 \ldots, 6, \tag{5.6}$$

其中 $\delta_{i,p} = 1$ 当路线 p 包含景点 i（即游客在某一时段游览了景点 i），否则 $\delta_{i,p} = 0$。

2. 午餐餐厅容量约束：每个区域的午餐接待能力（每天中午）也限制了路线分配。类似地，如果认为最多 3 个中午时段可复用，则对每个区域 $j \in R$：

$$\sum_{p\in\mathcal{P}} \theta_{j,p} \cdot x_p \le 3\, L_{R_j}, \quad j = 1, 2, \ldots, 6, \tag{5.7}$$

其中 $theta_{j,p}$ 代表路线 p 在中午曾安排于区域 j 就餐次数（一日游有 1 次午餐、三日游有 3 次午餐机会，但可能某些路线会重复去某一区域）。代码中简化为每个区域午餐总游客 $\le C_j^{\text{lunch}}$，相当假定所有行程共用一个中午时段容量，这在行程跨多日的情况下略趋保守。

3. 晚餐及住宿容量约束：每个区域用于晚餐和住宿的容量在不同夜晚也可重复利用。三日游行程最多涉及 2 晚，故每区域最多 2 个夜晚可安排住宿。对每个区域 $j \in R$：

$$\sum_{p\in\mathcal{P}} \phi_{j,p} \cdot x_p \;\le\; 2\, C_j^{\text{night}}, \quad j = 1, \ldots, 6, \tag{5.8}$$

其中 $\phi_{j,p}$ 为路线 p 在夜晚留宿于区域 j 的次数（二日游路线有 1 晚、三日游有 2 晚，一般假定同一线路不重复入住同一酒店区域）。

4. 游客总数约束：根据预期需求，分别规定各类行程分配的总游客数。例如，设定选择一日游的游客总数为 N_1 人、二日游 N_2 人、三日游 N_3 人。则对每一种行程类型分别有：

$$\sum_{p\in P_{1\text{day}}} x_p = N_1, \qquad \sum_{p\in P_{2\text{day}}} x_p = N_2, \qquad \sum_{p\in P_{3\text{day}}} x_p = N_3. \tag{5.9}$$

代码实现中分别令一日游、二日游总游客数为 30,000，三日游为 20,000。

约束处理完成后，我们将采用枚举的方式得到一日游、二日游等的所有可能方案。再将其加入到 MILP 模型中。结合遗传算法选择高质量路线，并采用启发式规则进行路线多样化拓展。最终得到最合适的旅游路线。

5.1.2 问题一结果分析

根据 w_{cost} 和 w_{time} 的不同权重配置，我们可以设置不同权重，得到不同侧重点的旅游方案。本论文设置了三种情况，并对优化结果进行了详细分析。

均衡型（费用权重 =0.5，时间权重 =0.5）：

均衡型配置追求费用和时间的平衡优化。实验结果显示：一日游方案成功分配 3 万游客到 2 条最优路线，平均满意度达到 0.47；二日游方案分配 3 万游客到 4 条路线，总费用 302.7 万元，总时间 5541 万分钟，平均满意度 0.39；三日游方案分配 2 万游客到 4 条路线，总费用 355.06 万元，平均满意度 0.34。该配置在各项指标间取得了良好的平衡，适合大多数游客的需求。

费用优先型（费用权重 =0.7，时间权重 =0.3）：

费用优先型配置更注重控制旅游成本。对比分析发现：在一日游方案中，费用优先型与均衡型结果完全一致，说明一日游路线已达到成本最优；二日游方案中，总费用降低至 292.92 万元，相比均衡型节省了 9.78 万元（降幅 3.2

时间优先型（费用权重 =0.3，时间权重 =0.7）：

时间优先型配置侧重于减少旅行时间。结果显示：一日游和二日游方案与均衡型完全相同，表明在这两种方案中时间已达到最优配置；三日游方案中，虽然总费用略有增加至 367.6 万元，但时间安排更为紧凑高效。值得注意的是，时间优先型在路线选择上呈现出不同的偏好模式，优先选择时间效率更高的景点组合和区域路径，适合时间有限但希望充分体验的游客。

综合比较分析：

通过三种配置的对比分析可以发现：(1) 一日游方案在三种配置下结果一致，说明短期旅游路线的优化空间有限，已达到全局最优；(2) 二日游方案中费用优先型在成本控制方面具有明显优势，适合预算约束较强的情况；(3) 三日游方案显示出最大的优化弹性，不同权重配置产生显著差异，为不同需求的游客提供了多样化选择；(4) 所有配置均成功实现了大规模游客分流，验证了模型的实用性和有效性。这些结果为旅游管理部门制定差异化的旅游产品策略提供了科学依据。

5.2 问题二：方案优化

5.2.1 问题分析与建模目标

根据问题分析，我们设计了如下的距离函数，综合考虑了方案间的景点、路线、区域等相似度，将重复性较强的方案合并。由于景点和区域皆为符号度量，因

此二者都采用了 Jaccard 相似度。类型相似度用以判断旅游天数是否相同，可以保证同一聚类中心皆为同一天数。余下的偏好相似度、成本相似度和时间相似度都采用了欧氏距离，成本相似度采用了欧氏距离，时间相似度采用了欧氏距离。

1. 景点相似度：

$$\text{Simu}_{p_i,p_j,S} = \frac{|\bigcup_{S\in p_i} \cap \bigcup_{S\in p_j}|}{|\bigcup_{S\in p_i} \cup \bigcup_{S\in p_j}|} \tag{5.10}$$

2. 类型相似度：

$$\text{Simu}_{p_i,p_j,Day} = \begin{cases} 1, & \text{若} d_{p_i} = d_{p_j} \\ 0, & \text{否则} \end{cases} \tag{5.11}$$

3. 区域相似度：

$$\text{Simu}_{p_i,p_j,R} = \frac{|\bigcup_{R\in p_i} \cap \bigcup_{R\in p_j}|}{|\bigcup_{R\in p_i} \cup \bigcup_{R\in p_j}|} \tag{5.12}$$

4. 偏好相似度：

$$\text{Simu}_{p_i,p_j,P} = 1 - \frac{|\text{Pref}(p_i) - \text{Pref}(p_j)|}{\max(\text{Pref}(p_i), \text{Pref}(p_j))} \tag{5.13}$$

5. 成本相似度：

$$\text{Simu}_{p_i,p_j,C} = 1 - \frac{|\text{Cost}(p_i) - \text{Cost}(p_j)|}{\max(\text{Cost}(p_i), \text{Cost}(p_j))} \tag{5.14}$$

6. 时间相似度：

$$\text{Simu}_{p_i,p_j,T} = 1 - \frac{|\text{Time}(p_i) - \text{Time}(p_j)|}{\max(\text{Time}(p_i), \text{Time}(p_j))} \tag{5.15}$$

定义好以上相似度后，就可以计算两个方案之间的加权距离。

$$D(p_i, p_j) = \sum_{k\in\text{Temp}}^{6} w_k \cdot \text{Simu}_{p_i,p_j,k}, \quad \text{Temp} = \{S, Day, R, P, C, T\} \tag{5.16}$$

以上因素综合考虑到不同旅行方案在景点、区域、偏好等多个方面的相似度，可以得到不同旅行方案之间的距离。在完成“旅游方案相似度”加权距离函数的数学建模后，下一步便可对所有候选路线实施 K-means 聚类。

5.2.2 模型求解和结果分析

具体流程如下：首先，以距离函数 $D(\cdot,\cdot)$ 为度量，对给定的 K 值（簇数）运行多次随机初始化的 K-means++，获得每个 k 下的最小簇内平方和。随后，将 k 在预设范围 $3,\ldots,6$ 内逐一取值。最终选择使指标综合最优的 K。确定 K 后，以其为簇数重新运行 K-means，并选取每个簇中净收益最高或代表性最强的方案，形成至多 10 种、覆盖多样化场景且经济效益最优的扩容套餐。

算法 5.1: 旅游路线相似度聚类算法

Input: 可行路线集合 $\mathcal{R}$（已转为特征向量）；候选聚类数 $\mathcal{K}=\{2,3,4,5,6\}$

Output: 最优聚类数 k^*、最终模型 $\mathcal{M}^*$ 及聚类统计 $\mathcal{C}$

步骤 1：初始化;
$k^* \leftarrow$ first($\mathcal{K}$)；;
bestInertia $\leftarrow +\infty$;
步骤 2：遍历候选聚类数;
foreach $k \in \mathcal{K}$ **do**
    $\mathcal{M} \leftarrow \mathrm{KMeans.fit}(\mathcal{R}, k)$;
    if $\mathcal{M}$.inertia_ $<$ *bestInertia* **then**
        bestInertia $\leftarrow \mathcal{M}$.inertia_;
        $k^* \leftarrow k$;
步骤 3：终极聚类与统计;
$\mathcal{M}^* \leftarrow \mathrm{KMeans.fit}(\mathcal{R}, k^*)$;
$\mathcal{C} \leftarrow \mathrm{GetClusterInfo}(\mathcal{M}^*, \mathcal{R})$;
步骤 4：输出结果;
return $(k^*, \mathcal{M}^*, \mathcal{C})$;

通过 K 均值算法，我们得到了最优聚类数为 6，其中每个簇中净收益最高的方案只保留一种，最终得到了 6 种旅游方案。以下是方案的详细信息：

表 5.1 K means 聚类结果汇总（$k=6$，路线总数 11，惯性值 1.178）

簇	成本/元	时间/分钟	满意度	路线类型	景点路线	区域路线
0	82.0	165.0	0.392	2 day	S1,S4,S6,S2	R1,R4,R2
1	116.0	225.0	0.361	2 day	S1,S2,S6,S4	R2,R2,R4
2	25.0	45.0	0.506	1 day	S3,S6	R3
3	161.0	285.00	0.401	3 day	S1,S4,S6,S5,S3,S2	R1,R2,R3,R3,R2
4	208.00	370.00	0.3560	3 day	S2,S3,S5,S4,S1,S6	R2,R3,R4,R4,R2
5	219.00	380.00	0.348	3 day	S3,S6,S5,S2,S4,S1	R2,R3,R4,R2,R4

由以上图表可以看出，聚类结果能够有效地将不同类型的旅游路线进行区分，每个簇代表了具有代表性的出行偏好和路线组合。各方案在成本、时间和满意度等方面表现均衡，能够满足不同游客的需求。整体来看，模型聚类效果良好，结果

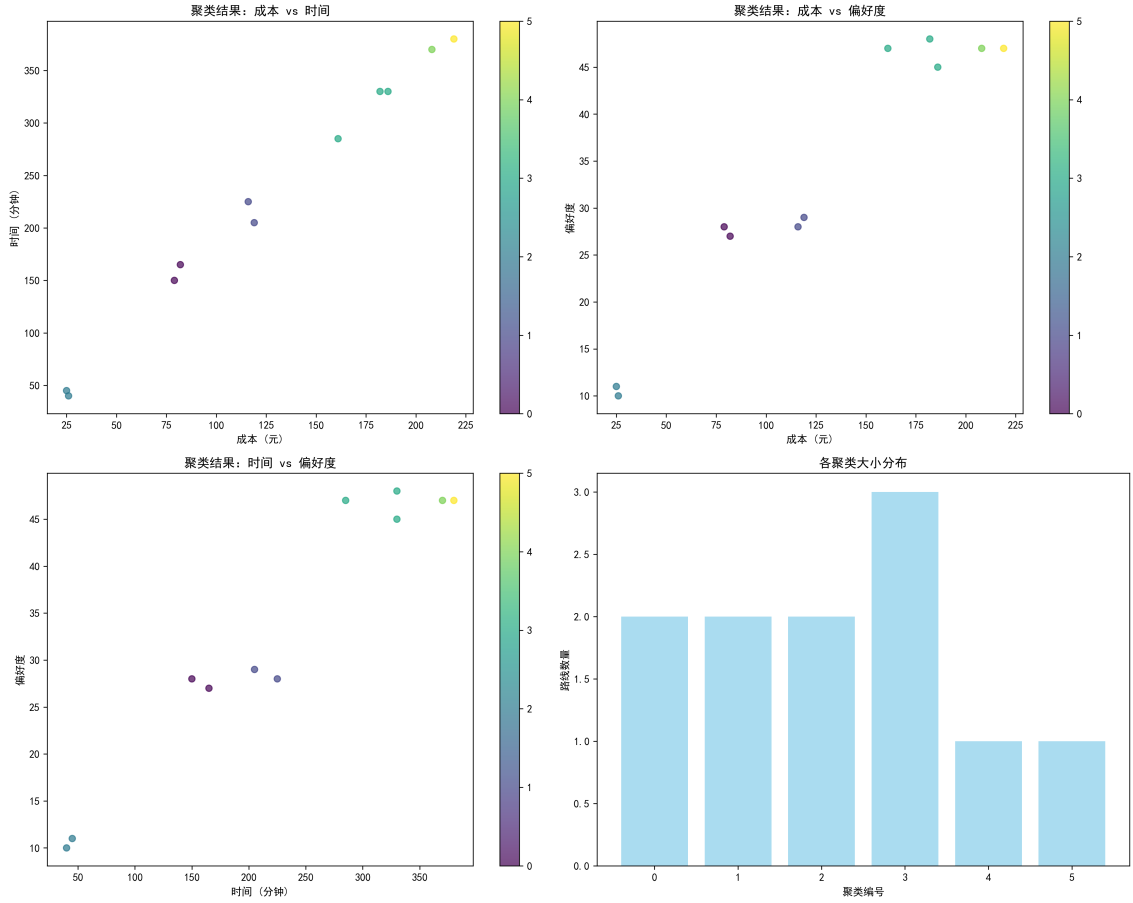

图 5.1 聚类结果可视化图。

Fig. 5.1 Histogram of clusterReseult.

具有较强的实际参考价值。

5.3 问题三：旅馆建设规模与位置选择

5.3.1 模型建立流程

本模型旨在分析重庆旅游景区的住宿扩容问题，基于游客需求、交通成本、时间消耗和景区容量等因素，通过构建优化模型对扩容方案进行评估。

首先，模型采用了基于规模报酬递减的扩容成本函数：

$$C(\Delta K) = c_1(\Delta K)^{\gamma}, \quad c_1 = 7 \times 10^{-4}(\text{亿元}), \gamma = 1.09 \tag{5.17}$$

此扩容成本函数是采用来自网络的数据统计，并进行处理后得到的。

并定义了经济收益函数 $B(\Delta K)$：

$$B(\Delta K) = v_s[Z(K) - Z(K^0)] + v_p[N(K) - N(K^0)] \tag{5.18}$$

其中考虑了扩容后满意度增量和游客接待量的增量。满意度增量通过扩容后游客对旅游线路的偏好变化进行计算，接待量增量则基于实际可接待游客人数与扩容前的差异。

扩容决策的目标是最大化净收益:

$$\max_{r,\Delta K\geq 0} F_r(\Delta K) = B_r(\Delta K) - C(\Delta K) \tag{5.19}$$

即通过选择最优区域和扩容规模 ΔK，使得总的经济效益最大。模型通过枚举不同区域和容量增量，计算每种方案的经济收益，并结合遗传算法和线性规划（MILP）对扩容后的最优游客分配进行求解。最终，选出净收益最大的扩容方案，并提供详细的扩容规模及其对应的效益曲线。

该模型不仅帮助政府和企业做出最优的扩容决策，还能评估不同扩容方案的经济影响，为旅游业的长期规划提供支持。

5.3.2 模型求解

模型的求解采用“双层结构”。外层离散搜索扩容区域 r 与规模 ΔK，内层维持既有的“枚举路线结合 MILP”的流程：

外层：

$$\max_{r,\Delta K\geq 0} F_r(\Delta K) \tag{5.20}$$

内层：

$$\begin{aligned} Z(K), N(K) = arg \quad & \max_{x} \sum_{i} s_i x_i \\ & s.t. \text{容量约束} K \end{aligned} \tag{5.21}$$

具体算法如下所示：

算法 5.2: 旅游景区扩容优化算法

Input: 原始夜宿容量 $K_0 = (K_1^0, \ldots, K_6^0)$；扩容步长集合 $\Delta\mathcal{K} = \{0, 0.5, 1, \ldots, \Delta K_{\max}\}$；游客需求 T^0

Output: 最优扩容区域 r^*、规模 ΔK^* 及最大净收益 F^*

步骤 1：基准求解;

base_results ← optimizer.run_optimization(K_0);

取 $Z(K^0)$, $N(K^0)$;

步骤 2：区域与扩容规模枚举;

for $r \leftarrow 1$ **to** 6 **do**

  for $\Delta K \in \Delta\mathcal{K}$ **do**

    $K_r \leftarrow K_r^0 + \Delta K$; // 修改容量上限

    cur ← optimizer.run_optimization(K_r);

    取 $Z(K)$, $N(K)$;

    $B = v_s\big[Z(K) - Z(K^0)\big] + v_p\big[N(K) - N(K^0)\big]$;

    $C = 0.0007\,(\Delta K)^{1.09}$;

    $F(r, \Delta K) = B - C$;

步骤 3：选择最优方案;

$(r^*, \Delta K^*) = \arg\max_{r,\Delta K} F(r, \Delta K)$;

$F^* \leftarrow F(r^*, \Delta K^*)$;

步骤 4：绘制净收益曲线;

绘制 $F(r, \Delta K)$ 随 ΔK 变化的曲线，并标注 $(r^*, \Delta K^*)$;

5.3.3 结果分析

在完成模型求解后，净收益曲线清晰展示了不同区域在不同扩容规模下的经济效益。下表汇总了主要扩容方案的关键指标，并按净收益从高到低排序。并且为进一步直观对比扩容规模与净收益关系，绘制了不同区域在多种扩容方案下的净收益分布。以下是得出的结论：

1. **区域 3 呈现显著经济优势**：前七大高收益方案均集中于区域 3，其中以扩容 1.00 万人次的方案最优，净收益达 74.32 亿元，其次为 2.00 万人次（70.43 亿元）与 1.50 万人次（57.90 亿元）。
2. **净收益对扩容规模并非单调增加**：区域 3 扩容从 1.00 提升至 2.00 万人次时，净收益下降，体现边际效益递减。曲线走势亦证实存在一最佳扩容规模区间。
3. **其他区域仍具备潜力**:区域 4 在小规模扩容（0.50 万人次）下即可带来 33.19 亿元收益；区域 5 与区域 1 的部分方案亦具备可观回报，但整体低于区域 3；区域 2 受限于基础容量与偏好，收益相对有限。

表 5.2 旅馆扩容方案净收益分析结果

区域	扩容规模/万人次	原容量/万人次	新容量/万人次	净收益/亿元
区域 3	1.00	0.66	1.66	74.3205
区域 3	2.00	0.66	2.66	70.4321
区域 3	1.50	0.66	2.16	57.9014
区域 3	4.50	0.66	5.16	53.2195
区域 3	3.00	0.66	3.66	48.7040
区域 3	4.00	0.66	4.66	46.9166
区域 3	5.00	0.66	5.66	44.5128
区域 4	0.50	2.16	2.66	33.1853
区域 3	0.50	0.66	1.16	30.0672
区域 3	3.50	0.66	4.16	29.9084
区域 5	4.00	1.38	5.38	28.5287
区域 5	3.50	1.38	4.88	28.0013
区域 1	1.00	1.14	2.14	27.6119
区域 5	2.00	1.38	3.38	25.1312
区域 3	2.50	0.66	3.16	24.3754
区域 2	1.00	1.92	2.92	24.1800
区域 1	1.50	1.14	2.64	23.6578
区域 5	0.00	1.38	1.38	23.5974
区域 1	2.00	1.14	3.14	22.2397
区域 1	0.00	1.14	1.14	20.2017

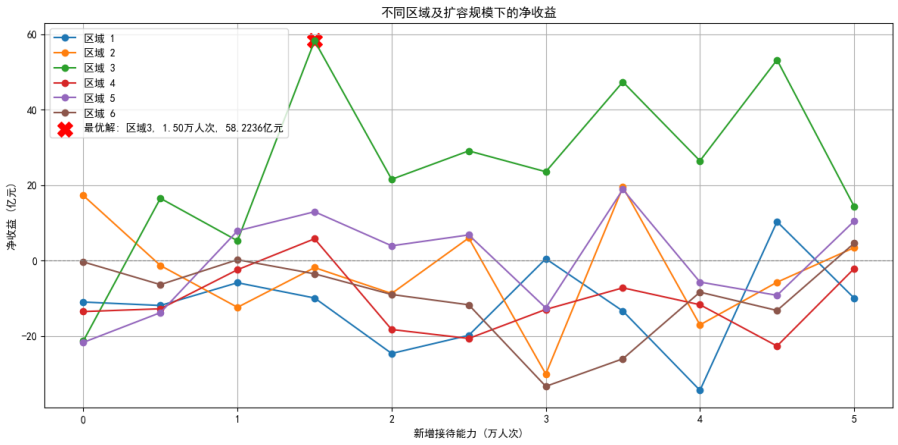

图 5.2 不同区域扩容规模与净收益关系图

6 模型评价与推广

6.1 主要结论

通过构建基于混合整数线性规划的多目标优化模型，成功实现了重庆旅游路线的智能分配，主要取得以下成果：

多目标优化效果显著：基于费用、时间和满意度的多目标优化模型能够有效平衡各项指标。实验结果表明，不同权重配置下的优化方案都能适应不同游客需求，其中费用优先型配置在三日游方案中表现最优，总费用降低至 341.32 万元，平均满意度提升至 0.36。

算法融合策略有效：采用 Floyd-Warshall 算法预处理交通矩阵，结合遗传算法和启发式方法生成候选路线，最终通过混合整数线性规划求解最优分配。混合方法成功生成 1200+ 条高质量候选路线，证明了多算法融合策略在大规模优化问题中的有效性。

6.2 模型优点

6.2.1 问题一模型优点

理论基础扎实：模型基于混合整数线性规划（MILP）理论构建，具有严格的数学理论支撑。通过线性目标函数和约束条件的设计，保证了求解的最优性和唯一性。同时，采用加权综合评价方法，将费用、时间和满意度等多维指标统一到同一评价体系中。

技术方法先进：集成了多种先进的优化算法，包括 Floyd-Warshall 最短路径算法用于交通网络预处理，遗传算法（GA）用于高质量候选解生成，启发式算法用于解空间探索，以及 MILP 用于精确求解。这种多算法融合的策略有效提升了模型的求解效率和解质量。

6.2.2 问题二模型优点

简洁高效：自定义距离函数 +K means 仅需秒级计算，即可将百级候选路线压缩为 ≤ 10 条代表方案。

多维综合：同时考虑景点、区域、费用、时间与偏好等 6 类特征，保证聚类结果在多维度上“互补且离散”。

可迁移性强：更新特征即可复用于其他城市或新景点，无需更改算法主体。

6.2.3 问题三模型优点

量化决策：模型能够将复杂的扩容决策量化为净收益，为旅游集团提供了清晰的经济指标，使得决策过程更加客观和科学。

多区域比较：模型同时考虑了多个区域的扩容潜力，并给出了各区域在不同扩容

规模下的表现，有助于全面评估不同选项。

考虑边际效益：结果显示净收益并非随扩容规模线性增长，而是存在最优解，模型考虑了扩容的边际成本和边际收益，这更符合实际情况。

6.3 不足与改进方向

6.3.1 问题一模型的主要局限

容量约束简化：当前模型采用时段复用的简化处理方式，假设景点和区域容量可在不同时段完全复用，但实际情况中可能存在时段冲突和资源竞争问题。

路线生成限制：为控制计算复杂度，对二日游和三日游候选路线数量进行了限制（分别为 5000 和 1200+ 条），可能遗漏部分高质量解。

6.3.2 问题一模型的改进方向

动态优化机制：引入实时数据更新和动态调整机制，根据实际游客流量、天气变化等因素动态调整路线分配，提高模型的适应性和实用性。

精细化约束建模：建立更精细的时空约束模型，考虑景点的实际开放时间、交通拥堵状况、季节性容量变化等因素，提高约束建模的准确性。

6.3.3 问题二模型的局限与改进方向

簇数判定单一：当前 K means 仅以惯性（SSE）为准则确定 K，未结合轮廓系数、Gap Statistic 等综合指标，存在簇数局部最优风险。

特征权重经验化：距离函数中六类权重 w_k 依赖经验设定，缺乏数据驱动的自动标定，可能导致聚类边界受主观偏差影响。

改进方向

（1）多指标确定 K：引入轮廓系数、Calinski-Harabasz 分数与 Gap Statistic 共同评估，采取“肘部 + 稳健性”策略自动选择最佳簇数。

（2）权重学习机制：利用历史游客反馈或专家打分，通过多目标贝叶斯优化或熵权法自适应确定 w_k，提升距离度量客观性。

6.3.4 问题三模型的主要局限

未考虑投资回报期：模型给出的净收益是特定扩容规模下的一个结果，但没有提及投资回收期或长期运营的财务状况。高净收益的方案可能需要巨大的初始投资，而投资回收期过长可能会影响决策。

成本参数不确定：扩容成本函数 $C(\Delta K) = c_1(\Delta K)^\gamma$ 基于宏观均值估计，未区分土地级差、层高限制等实际建造条件。

6.3.5 问题三模型的改进方向

投资回报评估：引入时间维度，计算投资回报期、净现值（NPV）或内部收益率（IRR）等财务指标，以评估项目的长期盈利能力和风险。

多情景成本估计：对 c_1, γ 设置信度区间，采用蒙特卡洛模拟评估成本不确定性对最优规模的影响，得到稳健决策区间。

参考文献

[1] 韩艳, 杨光, 武鑫森, 等. 考虑游客拥挤感知的旅游线路优化设计[J]. 北京工业大学学报, 2018, 44(12): 1537-1546.

[2] 任池. 基于异质性偏好、拥挤程度和环境因素的旅游线路设计研究[D]. 西南财经大学, 2024.

[3] 李婧璇, 禹文豪, 黄雅雅, 等. 基于多源大数据的城市旅游目的地吸引力评价研究——以武汉市为例[J/OL]. 时空信息学报, 2024, 31(03): 386-396. DOI: 10.20117/j.jsti.202403009.

[4] 常亮, 孙文平, 张伟涛, 等. 旅游路线规划研究综述[J]. 智能系统学报, 2019, 14(01): 82-92.

附 录

A. 支撑材料总览

本论文的所有支撑材料组织在 Materials/目录下，具体分类和说明如表A.1所示：

表 A.1 支撑材料分类说明。

Table A.1 Classification of Supporting Materials.

材料类型	文件路径	说明
实现代码	code/run_optimization.py	执行入口脚本
	code/tourism_optimization.py	问题一、问题三主要程序
	code/Q2_k_means.py	问题二：K-means 算法代码
结果材料	终端输出.docx	run_optimization.py 终端日志
	Q1/optimization_result_费用优先型.txt	问题 1：费用优先型解
	Q1/optimization_result_均衡型.txt	问题 1：均衡型解
	Training_Loss_Curve.png	
	Q1/optimization_result_时间优先型.txt	问题 1：时间优先型解
	ΔE2000_Error_Histogram.png	
	Q2/benefit_vs_expan.png	问题 2：收益-扩张曲线
	Q3/clustering_results.txt	问题 3：聚类结果（文本）
	Q3/optimization_clusters.png	问题 3：聚类结果可视化
说明文档	README.txt	项目环境与运行指南

B. 优化函数

```
import numpy as np
import itertools
from scipy.optimize import linprog
import random
from deap import base, creator, tools, algorithms
import matplotlib.pyplot as plt
import pandas as pd
from typing import List, Tuple, Dict
import warnings
warnings.filterwarnings('ignore')

# 定义适应度函数和个体（如果未定义）
try:
    creator.create("FitnessMax", base.Fitness, weights=(1.0,))
```

```
        creator.create("Individual", list, fitness=creator.FitnessMax)
    except Exception as e:
        # 避免重复定义
        pass

    class TourismOptimizer:
        def __init__(self):
            # 基础数据
            self.num_spots = 6
            self.num_regions = 6

            # 容量数据 (按0.6折减)
            self.spot_cap = np.array([12000, 36000, 20000, 42000, 38000,
                ↪ 30000]) * 0.6
            self.night_cap_original = np.array([19000, 32000, 11000, 36000,
                ↪ 23000, 22000]) * 0.6
            self.night_cap = np.copy(self.night_cap_original) # 用于修改的容量
            self.lunch_cap = np.array([23000, 39000, 13000, 45000, 31000,
                ↪ 28000]) * 0.6

            # 区域偏好度
            self.region_prefer = np.array([7, 8, 9, 8, 6, 7])

            # 原始费用和时间矩阵
            self.cost_mat_raw = np.array([
                [10, 25, 30, 18, 40, 25],
                [np.inf, 10, 16, 24, 28, 18],
                [np.inf, np.inf, 10, 24, 20, 15],
                [np.inf, np.inf, np.inf, 10, 24, 16],
                [np.inf, np.inf, np.inf, np.inf, 10, 22],
                [np.inf, np.inf, np.inf, np.inf, np.inf, 10]
            ])

            self.time_mat_raw = np.array([
                [30, 50, 60, 35, 70, 40],
                [np.inf, 30, 25, 30, 40, 30],
                [np.inf, np.inf, 15, 35, 30, 30],
                [np.inf, np.inf, np.inf, 15, 35, 25],
                [np.inf, np.inf, np.inf, np.inf, 20, 35],
                [np.inf, np.inf, np.inf, np.inf, np.inf, 25]
            ])

            # 处理后的完整矩阵
            self.cost_mat = None
            self.time_mat = None

            # 路线相关
            self.routes_1day = []
```

```
60              self.routes_2day = []
61              self.routes_3day_candidates = []
62
63          def preprocess_matrices(self, cost_weight=0.5, time_weight=0.5):
64              """使用加权综合Floyd-Warshall算法补全费用和时间矩阵
65
66              Args:
67                  cost_weight: 费用权重 (默认0.5)
68                  time_weight: 时间权重 (默认0.5)
69              """
70              # print(f"正在预处理交通矩阵 (加权方案: 费用权重={cost_weight}, 时
                    ↪ 间权重={time_weight}) ...") # 避免过多打印
71
72              # 创建邻接矩阵 (景点-区域双向图)
73              n = self.num_spots + self.num_regions  # 总节点数
74
75              # 初始化矩阵
76              combined_full = np.full((n, n), np.inf)
77              cost_full = np.full((n, n), np.inf)
78              time_full = np.full((n, n), np.inf)
79
80              # 对角线为0
81              np.fill_diagonal(combined_full, 0)
82              np.fill_diagonal(cost_full, 0)
83              np.fill_diagonal(time_full, 0)
84
85              # 计算归一化常数
86              finite_costs = self.cost_mat_raw[np.isfinite(self.cost_mat_raw)]
87              finite_times = self.time_mat_raw[np.isfinite(self.time_mat_raw)]
88              cost_max = np.max(finite_costs) if len(finite_costs) > 0 else 100
89              time_max = np.max(finite_times) if len(finite_times) > 0 else 100
90
91              # 填入已知的景点到区域的连接
92              for r in range(self.num_regions):
93                  for s in range(self.num_spots):
94                      if not np.isinf(self.cost_mat_raw[r, s]):
95                          cost = self.cost_mat_raw[r, s]
96                          time = self.time_mat_raw[r, s]
97
98                          # 归一化
99                          cost_normalized = cost / cost_max
100                         time_normalized = time / time_max
101
102                         # 加权综合成本
103                         combined_cost = cost_weight * cost_normalized +
                                ↪ time_weight * time_normalized
104
105                         # 景点s到区域r
```

```
106                         combined_full[s, self.num_spots + r] = combined_cost
107                         cost_full[s, self.num_spots + r] = cost
108                         time_full[s, self.num_spots + r] = time
109 
110                         # 区域r到景点s (假设双向相等)
111                         combined_full[self.num_spots + r, s] = combined_cost
112                         cost_full[self.num_spots + r, s] = cost
113                         time_full[self.num_spots + r, s] = time
114 
115             # 使用综合成本运行Floyd-Warshall算法
116             for k in range(n):
117                 for i in range(n):
118                     for j in range(n):
119                         if combined_full[i, k] + combined_full[k, j] <
                                ↪ combined_full[i, j]:
120                             combined_full[i, j] = combined_full[i, k] +
                                    ↪ combined_full[k, j]
121                             # 同时更新对应的实际费用和时间
122                             cost_full[i, j] = cost_full[i, k] + cost_full[k, j
                                    ↪ ]
123                             time_full[i, j] = time_full[i, k] + time_full[k, j
                                    ↪ ]
124 
125             # 提取景点到区域的部分
126             self.cost_mat = cost_full[:self.num_spots, self.num_spots:]
127             self.time_mat = time_full[:self.num_spots, self.num_spots:]
128 
129             # print("矩阵预处理完成") # 避免过多打印
130 
131         def generate_1day_routes(self):
132             """生成所有一日游路线: S -> R -> S"""
133             # print("生成一日游路线...") # 避免过多打印
134             self.routes_1day = []
135 
136             for s1 in range(self.num_spots):
137                 for r in range(self.num_regions):
138                     for s2 in range(self.num_spots):
139                         if s1 != s2:  # 避免重复访问同一景点
140                             route = {
141                                 'type': '1day',
142                                 'path': [s1, r, s2],
143                                 'spots': [s1, s2],
144                                 'regions': [r],
145                                 'lunch_regions': [r],
146                                 'night_regions': [],
147                                 'cost': self.cost_mat[s1, r] + self.cost_mat[
                                        ↪ s2, r],
148                                 'time': self.time_mat[s1, r] + self.time_mat[
```

```
                                    ↪ s2, r],
149                                 'preference': sum([self.region_prefer[r]]) + 2
                                        ↪  # 景点基础偏好
150                             }
151                             self.routes_1day.append(route)
152
153         # print(f"一日游路线数量: {len(self.routes_1day)}") # 避免过多打印
154
155     def generate_2day_routes(self, max_routes=10000):
156         """生成二日游路线: S -> R -> S -> R -> S -> R -> S"""
157         # print("生成二日游路线...") # 避免过多打印
158         self.routes_2day = []
159         count = 0
160
161         for s1 in range(self.num_spots):
162             for r1 in range(self.num_regions):
163                 for s2 in range(self.num_spots):
164                     for r2 in range(self.num_regions):
165                         for s3 in range(self.num_spots):
166                             for r3 in range(self.num_regions):
167                                 for s4 in range(self.num_spots):
168                                     if len(set([s1, s2, s3, s4])) == 4:  #
                                            ↪ 所有景点不重复
169                                         if count >= max_routes:
170                                             break
171
172                                     route = {
173                                         'type': '2day',
174                                         'path': [s1, r1, s2, r2, s3, r3,
                                                ↪ s4],
175                                         'spots': [s1, s2, s3, s4],
176                                         'regions': [r1, r2, r3],
177                                         'lunch_regions': [r1, r3],  # 第1
                                                ↪ 天午餐r1, 第2天午餐r3
178                                         'night_regions': [r2],  # 第1天住
                                                ↪ 宿r2
179                                         'cost': (self.cost_mat[s1, r1] +
                                                ↪ self.cost_mat[s2, r1] +
180                                                 self.cost_mat[s2, r2] +
                                                        ↪ self.cost_mat[s3,
                                                        ↪ r2] +
181                                                 self.cost_mat[s3, r3] +
                                                        ↪ self.cost_mat[s4,
                                                        ↪ r3]),
182                                         'time': (self.time_mat[s1, r1] +
                                                ↪ self.time_mat[s2, r1] +
183                                                 self.time_mat[s2, r2] +
                                                        ↪ self.time_mat[s3,
```

```
                                                    ↪ r2] +
184                                                 self.time_mat[s3, r3] +
                                                        ↪ self.time_mat[s4,
                                                        ↪ r3]),
185                                             'preference': sum([self.
                                                    ↪ region_prefer[r1], self.
                                                    ↪ region_prefer[r2],
186                                                                 self.
                                                                    ↪ region_prefer
                                                                    ↪ [r3]]) +
                                                                    ↪ 4
187                                         }
188                                             self.routes_2day.append(route)
189                                             count += 1
190                                         if count >= max_routes:
191                                             break
192                                     if count >= max_routes:
193                                         break
194                                 if count >= max_routes:
195                                     break
196                             if count >= max_routes:
197                                 break
198                         if count >= max_routes:
199                             break
200                     if count >= max_routes:
201                         break
202                 if count >= max_routes:
203                     break
204
205             # print(f"二日游路线数量: {len(self.routes_2day)}") # 避免过多打印
206
207         def generate_3day_candidates_ga(self, population_size=1000,
                ↪ generations=50, candidate_size=5000):
208             """使用遗传算法生成三日游候选路线"""
209             # print("使用遗传算法生成三日游候选路线...") # 避免过多打印
210
211             # 定义适应度函数 (已在类外部处理, 确保只创建一次)
212
213             toolbox = base.Toolbox()
214
215             def create_individual():
216                 """创建一个个体 (三日游路线)"""
217                 # 随机选择6个不同的景点
218                 spots = random.sample(range(self.num_spots), 6)
219                 # 随机选择5个区域
220                 regions = [random.randint(0, self.num_regions-1) for _ in
                        ↪ range(5)]
221                 return spots + regions
```

```
222
223             def evaluate(individual):
224                 """评估个体的适应度"""
225                 spots = individual[:6]
226                 regions = individual[6:]
227
228                 # 验证景点不重复
229                 if len(set(spots)) != 6:
230                     return (-1000,) # 惩罚重复景点路线
231
232                 # 计算路线: S1->R1->S2->R2->S3->R3->S4->R4->S5->R5->S6
233                 try:
234                     cost = 0
235                     time = 0
236
237                     # Day 1: S1->R1->S2->R2 (Overnight in R2)
238                     cost += self.cost_mat[spots[0], regions[0]] + self.
                            ↪ cost_mat[spots[1], regions[0]]
239                     cost += self.cost_mat[spots[1], regions[1]] # Travel from
                            ↪ S2 to R2 for overnight
240                     time += self.time_mat[spots[0], regions[0]] + self.
                            ↪ time_mat[spots[1], regions[0]]
241                     time += self.time_mat[spots[1], regions[1]]
242
243                     # Day 2: S2->R2->S3->R3->S4->R4 (Overnight in R4)
244                     # Travel from R2 to S3 (already at R2)
245                     cost += self.cost_mat[spots[2], regions[2]]
246                     cost += self.cost_mat[spots[3], regions[2]]
247                     cost += self.cost_mat[spots[3], regions[3]] # Travel from
                            ↪ S4 to R4 for overnight
248                     time += self.time_mat[spots[2], regions[2]]
249                     time += self.time_mat[spots[3], regions[2]]
250                     time += self.time_mat[spots[3], regions[3]]
251
252                     # Day 3: S4->R4->S5->R5->S6 (End trip)
253                     # Travel from R4 to S5 (already at R4)
254                     cost += self.cost_mat[spots[4], regions[4]]
255                     cost += self.cost_mat[spots[5], regions[4]]
256                     time += self.time_mat[spots[4], regions[4]]
257                     time += self.time_mat[spots[5], regions[4]]
258
259                     preference = sum([self.region_prefer[r] for r in regions])
                            ↪ + 6 # 6 unique spots contribute to base preference
260
261                     # 归一化计算 (与MILP保持一致)
262                     max_preference_3day = 9 * 5 + 6  # 理论最大偏好度
263                     # 估计的最大成本和时间, 用于遗传算法的归一化, 不必非常精
                            ↪ 确, 只要能大致区分好坏
```

```
                    max_cost_estimate = 100 * 10 # 粗略估计
                    max_time_estimate = 150 * 10 # 粗略估计

                    pref_normalized = preference / max_preference_3day
                    cost_normalized = cost / max_cost_estimate
                    time_normalized = time / max_time_estimate

                    # 加权综合评分 (权重与MILP一致)
                    w_pref, w_cost, w_time = 0.6, 0.2, 0.2
                    fitness = w_pref * pref_normalized - w_cost *
                        ↪ cost_normalized - w_time * time_normalized

                    return (fitness,)
                except:
                    return (-1000,) # 无效路径惩罚

            toolbox.register("individual", tools.initIterate, creator.
                ↪ Individual, create_individual)
            toolbox.register("population", tools.initRepeat, list, toolbox.
                ↪ individual)
            toolbox.register("evaluate", evaluate)

            def crossover_individual(ind1, ind2):
                """自定义交叉函数, 确保景点不重复"""
                # 对景点部分使用顺序交叉(OX)
                spots1, spots2 = ind1[:6], ind2[:6]
                regions1, regions2 = ind1[6:], ind2[6:]

                # 顺序交叉(Order Crossover)
                def order_crossover(parent1, parent2):
                    size = len(parent1)
                    start, end = sorted(random.sample(range(size), 2))

                    child = [-1] * size
                    # 复制中间段
                    child[start:end] = parent1[start:end]

                    # 从parent2中按顺序填充剩余位置
                    pointer = end
                    for item in parent2[end:] + parent2[:end]:
                        if item not in child:
                            while child[pointer] != -1: # Find next empty spot
                                pointer = (pointer + 1) % size
                            child[pointer] = item
                            pointer = (pointer + 1) % size # Move pointer for
                                ↪ next item

                    return child
```

```
308
309                 # 对景点使用顺序交叉
310                 new_spots1 = order_crossover(spots1, spots2)
311                 new_spots2 = order_crossover(spots2, spots1)
312
313                 # 对区域使用简单的两点交叉
314                 if len(regions1) > 1:
315                     cx_point = random.randint(1, len(regions1)-1)
316                     new_regions1 = regions1[:cx_point] + regions2[cx_point:]
317                     new_regions2 = regions2[:cx_point] + regions1[cx_point:]
318                 else:
319                     new_regions1, new_regions2 = regions1[:], regions2[:]
320
321                 # 重新组合
322                 ind1[:] = new_spots1 + new_regions1
323                 ind2[:] = new_spots2 + new_regions2
324
325                 return ind1, ind2
326
327             toolbox.register("mate", crossover_individual)
328
329             def mutate_individual(individual):
330                 """自定义变异函数"""
331                 if random.random() < 0.1: # 变异概率
332                     # 变异景点部分 (前6个位置) - 使用交换确保不重复
333                     pos1, pos2 = random.sample(range(6), 2)
334                     individual[pos1], individual[pos2] = individual[pos2],
                        ↪ individual[pos1]
335                 if random.random() < 0.1: # 变异概率
336                     # 变异区域部分 (后5个位置)
337                     pos = random.randint(6, 10)
338                     individual[pos] = random.randint(0, self.num_regions-1)
339                 return individual,
340
341             toolbox.register("mutate", mutate_individual)
342             toolbox.register("select", tools.selTournament, tournsize=3)
343
344             # 运行遗传算法
345             population = toolbox.population(n=population_size)
346
347             # 初始评估
348             fitnesses = list(map(toolbox.evaluate, population))
349             for ind, fit in zip(population, fitnesses):
350                 ind.fitness.values = fit
351
352             for gen in range(generations):
353                 # 选择
354                 offspring = toolbox.select(population, len(population))
```

```
                offspring = list(map(toolbox.clone, offspring))

                # 交叉和变异
                for child1, child2 in zip(offspring[::2], offspring[1::2]):
                    if random.random() < 0.5:
                        toolbox.mate(child1, child2)
                        del child1.fitness.values
                        del child2.fitness.values

                for mutant in offspring:
                    if random.random() < 0.2:
                        toolbox.mutate(mutant)
                        del mutant.fitness.values

                # 评估无效个体
                invalid_ind = [ind for ind in offspring if not ind.fitness.
                    ↪ valid]
                fitnesses = map(toolbox.evaluate, invalid_ind)
                for ind, fit in zip(invalid_ind, fitnesses):
                    ind.fitness.values = fit

                population[:] = offspring # Update population with new
                    ↪ generation

                # if gen % 10 == 0: # 避免过多打印
                #     fits = [ind.fitness.values[0] for ind in population if
                    ↪ ind.fitness.valid]
                #     if fits:
                #         print(f"Generation {gen}: Max={max(fits):.2f}, Avg={
                    ↪ np.mean(fits):.2f}")

            # 提取最优个体作为候选路线, 保证多样性
            population.sort(key=lambda x: x.fitness.values[0], reverse=True)

            self.routes_3day_candidates = []
            used_route_keys = set()

            for ind in population:
                spots = ind[:6]
                regions = ind[6:]

                # 验证景点不重复
                if len(set(spots)) != 6:
                    continue  # 跳过有重复景点的个体

                # 创建路线唯一标识
                route_key = tuple(spots + regions)

```

```
399                 # 只选择独特的路线
400                 if route_key not in used_route_keys and len(self.
                        ↪ routes_3day_candidates) < candidate_size:
401                     used_route_keys.add(route_key)
402
403                     route = {
404                         'type': '3day',
405                         'path': [spots[0], regions[0], spots[1], regions[1],
                                ↪ spots[2], regions[2],
406                                     spots[3], regions[3], spots[4], regions[4],
                                        ↪ spots[5]],
407                         'spots': spots,
408                         'regions': regions,
409                         'lunch_regions': [regions[0], regions[2], regions[4]],
                                ↪  # 3天的午餐
410                         'night_regions': [regions[1], regions[3]], # 前2天的
                                ↪ 住宿
411                         'cost': 0, # 需要重新计算
412                         'time': 0, # 需要重新计算
413                         'preference': 0, # 需要重新计算
414                         'fitness': ind.fitness.values[0]
415                     }
416
417                     # 重新精确计算成本、时间、偏好
418                     try:
419                         cost = (self.cost_mat[spots[0], regions[0]] + self.
                                ↪ cost_mat[spots[1], regions[0]] +
420                                 self.cost_mat[spots[1], regions[1]] + self.
                                    ↪ cost_mat[spots[2], regions[1]] +
421                                 self.cost_mat[spots[2], regions[2]] + self.
                                    ↪ cost_mat[spots[3], regions[2]] +
422                                 self.cost_mat[spots[3], regions[3]] + self.
                                    ↪ cost_mat[spots[4], regions[3]] +
423                                 self.cost_mat[spots[4], regions[4]] + self.
                                    ↪ cost_mat[spots[5], regions[4]])
424
425                         time = (self.time_mat[spots[0], regions[0]] + self.
                                ↪ time_mat[spots[1], regions[0]] +
426                                 self.time_mat[spots[1], regions[1]] + self.
                                    ↪ time_mat[spots[2], regions[1]] +
427                                 self.time_mat[spots[2], regions[2]] + self.
                                    ↪ time_mat[spots[3], regions[2]] +
428                                 self.time_mat[spots[3], regions[3]] + self.
                                    ↪ time_mat[spots[4], regions[3]] +
429                                 self.time_mat[spots[4], regions[4]] + self.
                                    ↪ time_mat[spots[5], regions[4]])
430
431                         preference = sum([self.region_prefer[r] for r in
```

```
                            ↪ regions]) + 6
432
433                         route['cost'] = cost
434                         route['time'] = time
435                         route['preference'] = preference
436
437                         self.routes_3day_candidates.append(route)
438                     except:
439                         continue
440
441             # print(f"三日游候选路线数量: {len(self.routes_3day_candidates)}")
                    ↪ # 避免过多打印
442
443         def generate_3day_routes_hybrid(self, target_routes=5000):
444             """混合方法生成三日游路线: 遗传算法 + 启发式规则"""
445             print("使用混合方法生成三日游路线...")
446
447             self.routes_3day_candidates = []
448
449             # 1. 先用遗传算法生成高质量路线
450             print("步骤1: 遗传算法生成高质量路线...")
451             self.generate_3day_candidates_ga(candidate_size=int(target_routes
                    ↪ * 0.1), population_size=1000, generations=50) # 减少GA生成的
                    ↪ 比例
452             ga_routes_count = len(self.routes_3day_candidates)
453             print(f"遗传算法生成: {ga_routes_count}条路线")
454
455             # 2. 启发式生成多样化路线
456             print("步骤2: 启发式生成多样化路线...")
457             used_route_keys = set()
458             for route in self.routes_3day_candidates:
459                 route_key = tuple(route['spots'] + route['regions'])
460                 used_route_keys.add(route_key)
461
462             # 为每个区域组合生成路线
463             # 考虑更系统地生成, 而不是固定组合
464             # 随机生成区域组合, 增加多样性
465
466             num_heuristic_routes = target_routes - ga_routes_count
467
468             # 尝试生成更多样化的区域组合
469             for _ in range(num_heuristic_routes * 2): # 尝试生成更多, 因为会有
                    ↪ 重复或无效
470                 if len(self.routes_3day_candidates) >= target_routes:
471                     break
472
473                 # 随机选择5个区域, 可以重复
474                 region_combo = [random.randint(0, self.num_regions - 1) for _
```

```
                    ↪ in range(5)]
475
476                 # 为该区域组合生成多种景点排列 (随机选择部分)
477                 spot_permutations = list(itertools.permutations(range(self.
                        ↪ num_spots)))
478                 selected_perms = random.sample(spot_permutations, min(2, len(
                        ↪ spot_permutations))) # 减少每个区域组合的景点排列数
479
480                 for spots in selected_perms:
481                     if len(self.routes_3day_candidates) >= target_routes:
482                         break
483
484                     route_key = tuple(list(spots) + region_combo)
485                     if route_key not in used_route_keys:
486                         used_route_keys.add(route_key)
487
488                         route = {
489                             'type': '3day',
490                             'path': [spots[0], region_combo[0], spots[1],
                                    ↪ region_combo[1], spots[2], region_combo[2],
491                                         spots[3], region_combo[3], spots[4],
                                            ↪ region_combo[4], spots[5]],
492                             'spots': list(spots),
493                             'regions': region_combo,
494                             'lunch_regions': [region_combo[0], region_combo
                                    ↪ [2], region_combo[4]],
495                             'night_regions': [region_combo[1], region_combo
                                    ↪ [3]],
496                         }
497
498                         # 计算成本、时间、偏好
499                         try:
500                             cost = (self.cost_mat[spots[0], region_combo[0]] +
                                    ↪  self.cost_mat[spots[1], region_combo[0]] +
501                                     self.cost_mat[spots[1], region_combo[1]] +
                                        ↪ self.cost_mat[spots[2], region_combo
                                        ↪ [1]] +
502                                     self.cost_mat[spots[2], region_combo[2]] +
                                        ↪ self.cost_mat[spots[3], region_combo
                                        ↪ [2]] +
503                                     self.cost_mat[spots[3], region_combo[3]] +
                                        ↪ self.cost_mat[spots[4], region_combo
                                        ↪ [3]] +
504                                     self.cost_mat[spots[4], region_combo[4]] +
                                        ↪ self.cost_mat[spots[5], region_combo
                                        ↪ [4]])
505
506                             time = (self.time_mat[spots[0], region_combo[0]] +
```

```
                                    ↪ self.time_mat[spots[1], region_combo[0]] +
507                                     self.time_mat[spots[1], region_combo[1]] +
                                        ↪ self.time_mat[spots[2], region_combo
                                        ↪ [1]] +
508                                     self.time_mat[spots[2], region_combo[2]] +
                                        ↪ self.time_mat[spots[3], region_combo
                                        ↪ [2]] +
509                                     self.time_mat[spots[3], region_combo[3]] +
                                        ↪ self.time_mat[spots[4], region_combo
                                        ↪ [3]] +
510                                     self.time_mat[spots[4], region_combo[4]] +
                                        ↪ self.time_mat[spots[5], region_combo
                                        ↪ [4]])
511
512                             preference = sum([self.region_prefer[r] for r in
                                    ↪ region_combo]) + 6
513
514                             route['cost'] = cost
515                             route['time'] = time
516                             route['preference'] = preference
517
518                             self.routes_3day_candidates.append(route)
519                         except:
520                             continue
521
522             print(f"混合方法总共生成: {len(self.routes_3day_candidates)}条路线
                    ↪ ")
523             print(f"其中遗传算法: {ga_routes_count}条, 启发式: {len(self.
                    ↪ routes_3day_candidates) - ga_routes_count}条")
524
525         def solve_milp(self, routes, total_tourists=10000, weights=(0.6, 0.2,
                ↪ 0.2)):
526             """使用混合整数线性规划求解路线分配"""
527             # print(f"求解MILP, 路线数量: {len(routes)}") # 避免过多打印
528
529             if not routes:
530                 return {'solution': [], 'total_satisfaction': 0, '
                        ↪ total_tourists': 0} # 返回包含总满意度和总游客数的字典
531
532             n_routes = len(routes)
533             w_pref, w_cost, w_time = weights
534
535             # 计算归一化的理论最大值
536             max_preference_1day = 9 * 1 + 2  # 1个区域 + 2个景点
537             max_preference_2day = 9 * 3 + 4  # 3个区域 + 4个景点
538             max_preference_3day = 9 * 5 + 6  # 5个区域 + 6个景点
539
540             # 成本和时间: 从所有路线中找最大值作为归一化基准
```

```
            max_cost = max([route['cost'] for route in routes]) if routes else
                ↪ 1
            max_time = max([route['time'] for route in routes]) if routes else
                ↪ 1

            # 根据路线类型确定偏好度最大值
            def get_max_preference(route_type):
                if route_type == '1day':
                    return max_preference_1day
                elif route_type == '2day':
                    return max_preference_2day
                else:  # 3day
                    return max_preference_3day

            # 构建目标函数系数 (最大化满意度 = 最大化偏好 - 最小化成本和时间)
            c = []
            for route in routes:
                # 归一化各项指标到[0,1]范围
                pref_normalized = route['preference'] / get_max_preference(
                    ↪ route['type'])
                cost_normalized = route['cost'] / max_cost
                time_normalized = route['time'] / max_time

                # 加权计算满意度
                satisfaction = w_pref * pref_normalized - w_cost *
                    ↪ cost_normalized - w_time * time_normalized
                c.append(-satisfaction)  # linprog求最小值, 所以取负

            # 约束矩阵
            A_ub = []
            b_ub = []

            # 景点容量约束 - 按时段分别约束
            max_days = 3  # 最多3日游
            max_time_slots = max_days * 2  # 每天上午+下午 (粗略估计, 实际应更
                ↪ 精细, 但为了简化模型, 保持原逻辑)

            for s in range(self.num_spots):
                constraint = []
                for route in routes:
                    # 计算该路线中景点s被访问的次数
                    visit_count = route['spots'].count(s)
                    constraint.append(visit_count)
                A_ub.append(constraint)
                # 景点在多个时段可复用, 总容量 = 单时段容量 × 时段数
                total_capacity = self.spot_cap[s] * max_time_slots
                b_ub.append(total_capacity)

```

```
584             # 午餐容量约束
585             for r in range(self.num_regions):
586                 constraint = []
587                 for route in routes:
588                     count = route['lunch_regions'].count(r)
589                     constraint.append(count)
590                 A_ub.append(constraint)
591                 b_ub.append(self.lunch_cap[r])
592 
593             # 住宿容量约束 - 按夜晚时段复用
594             max_nights = max_days - 1  # 最大夜晚数（3日游需要2晚）
595 
596             for r in range(self.num_regions):
597                 constraint = []
598                 for route in routes:
599                     count = route['night_regions'].count(r)
600                     constraint.append(count)
601                 A_ub.append(constraint)
602                 # 住宿容量可以在不同夜晚复用
603                 total_night_capacity = self.night_cap[r] * max_nights # 使用
                        ↪ self.night_cap
604                 b_ub.append(total_night_capacity)
605 
606             # 等式约束：总游客数
607             A_eq = [[1] * n_routes]
608             b_eq = [total_tourists]
609 
610             # 变量边界
611             bounds = [(0, total_tourists) for _ in range(n_routes)]
612 
613             try:
614                 # 求解
615                 result = linprog(c, A_ub=A_ub, b_ub=b_ub, A_eq=A_eq, b_eq=b_eq
                        ↪ ,
616                                  bounds=bounds, method='highs')
617 
618                 if result.success:
619                     solution = []
620                     total_s = 0
621                     total_t = 0
622                     for i, x in enumerate(result.x):
623                         if x > 0.1:  # 过滤掉很小的值
624                             num_tourists = int(round(x))
625                             route_info = routes[i].copy()
626                             route_info['tourists'] = num_tourists
627                             # 重新计算归一化满意度用于显示
628                             pref_normalized = route_info['preference'] /
                                    ↪ get_max_preference(route_info['type'])
```

```
                            cost_normalized = route_info['cost'] / max_cost
                            time_normalized = route_info['time'] / max_time
                            route_info['satisfaction'] = w_pref *
                                ↪ pref_normalized - w_cost * cost_normalized -
                                ↪  w_time * time_normalized
                            solution.append(route_info)
                            total_s += num_tourists * route_info['satisfaction
                                ↪ ']
                            total_t += num_tourists

                    return {'solution': solution, 'total_satisfaction':
                        ↪ total_s, 'total_tourists': total_t}
                else:
                    # print("MILP求解失败") # 避免过多打印
                    return {'solution': [], 'total_satisfaction': 0, '
                        ↪ total_tourists': 0}
            except Exception as e:
                # print(f"MILP求解出错: {e}") # 避免过多打印
                return {'solution': [], 'total_satisfaction': 0, '
                    ↪ total_tourists': 0}

        def run_optimization(self, cost_weight=0.5, time_weight=0.5,
            ↪ tourists_1day=30000, tourists_2day=30000, tourists_3day=20000):
            """运行完整的优化流程

            Args:
                cost_weight: 费用权重, 用于路径预处理 (默认0.5)
                time_weight: 时间权重, 用于路径预处理 (默认0.5)
                tourists_1day: 一日游总游客数
                tourists_2day: 二日游总游客数
                tourists_3day: 三日游总游客数
            """
            # print("开始旅游路线优化...") # 避免过多打印
            # print(f"使用权重配置: 费用权重={cost_weight}, 时间权重={
                ↪ time_weight}")

            # 1. 数据预处理
            self.preprocess_matrices(cost_weight, time_weight)

            # 2. 生成路线
            self.generate_1day_routes()
            self.generate_2day_routes(max_routes=5000)  # 限制二日游路线数量
            self.generate_3day_routes_hybrid(target_routes=5000) # 使用混合方
                ↪ 法生成三日游路线

            # 3. 分别求解三种旅游方案
            results = {}

```

```
668             # 一日游
669             # print("\n=== 求解一日游方案 ===")
670             solution_1day_res = self.solve_milp(self.routes_1day,
                    ↪ total_tourists=tourists_1day)
671             results['1day'] = solution_1day_res['solution']
672             results['1day_metrics'] = {'total_satisfaction': solution_1day_res
                    ↪ ['total_satisfaction'], 'total_tourists': solution_1day_res[
                    ↪ 'total_tourists']}
673
674             # 二日游
675             # print("\n=== 求解二日游方案 ===")
676             solution_2day_res = self.solve_milp(self.routes_2day,
                    ↪ total_tourists=tourists_2day)
677             results['2day'] = solution_2day_res['solution']
678             results['2day_metrics'] = {'total_satisfaction': solution_2day_res
                    ↪ ['total_satisfaction'], 'total_tourists': solution_2day_res[
                    ↪ 'total_tourists']}
679
680             # 三日游
681             # print("\n=== 求解三日游方案 ===")
682             solution_3day_res = self.solve_milp(self.routes_3day_candidates,
                    ↪ total_tourists=tourists_3day)
683             results['3day'] = solution_3day_res['solution']
684             results['3day_metrics'] = {'total_satisfaction': solution_3day_res
                    ↪ ['total_satisfaction'], 'total_tourists': solution_3day_res[
                    ↪ 'total_tourists']}
685
686             return results
687
688         def get_aggregated_metrics(self, results):
689             """从优化结果中聚合总满意度和总游客数"""
690             total_satisfaction_all_types = 0
691             total_tourists_all_types = 0
692
693             for day_type in ['1day', '2day', '3day']:
694                 if f'{day_type}_metrics' in results:
695                     total_satisfaction_all_types += results[f'{day_type}
                            ↪ _metrics']['total_satisfaction']
696                     total_tourists_all_types += results[f'{day_type}_metrics'
                            ↪ ]['total_tourists']
697
698             return total_satisfaction_all_types, total_tourists_all_types
699
700         def print_results(self, results):
701             """打印优化结果"""
702             for day_type, solution in results.items():
703                 if not isinstance(solution, list): # 过滤掉 metrics 字典
704                     continue
```

```
705
706                 print(f"\n{'='*50}")
707                 print(f"{day_type.upper()} 旅游方案结果")
708                 print(f"{'='*50}")
709
710                 if not solution:
711                     print("无可行解")
712                     continue
713
714                 total_tourists = sum([route['tourists'] for route in solution
                        ↪ ])
715                 total_cost = sum([route['tourists'] * route['cost'] for route
                        ↪ in solution])
716                 total_time = sum([route['tourists'] * route['time'] for route
                        ↪ in solution])
717                 total_satisfaction = sum([route['tourists'] * route['
                        ↪ satisfaction'] for route in solution])
718
719                 print(f"路线类型数量: {len(solution)}")
720                 print(f"总游客数: {total_tourists}")
721                 print(f"总交通成本: {total_cost:.0f} 元")
722                 print(f"总交通时间: {total_time:.0f} 分钟")
723                 print(f"总满意度: {total_satisfaction:.2f}")
724                 print(f"平均满意度: {total_satisfaction/total_tourists:.2f}")
725
726                 print("\n具体路线分配:")
727                 for i, route in enumerate(solution[:10]): # 只显示前10个
728                     spots_str = "->".join([f"S{s+1}" for s in route['spots']])
729                     regions_str = "->".join([f"R{r+1}" for r in route['regions
                            ↪ ']])
730                     print(f"路线{i+1}: {route['tourists']}人")
731                     print(f"   景点路径: {spots_str}")
732                     print(f"   区域路径: {regions_str}")
733                     print(f"   成本: {route['cost']:.1f}元, 时间: {route['time
                            ↪ ']:.1f}分钟, 偏好: {route['preference']}")
734                     print()
735
736     # 扩容成本函数
737     def expansion_cost(delta_K: float) -> float:
738         """
739         扩容成本函数
740         Args:
741             delta_K: 新增接待能力（万人次）
742         Returns:
743             扩容成本（亿元）
744         """
745         c1 = 0.0007  # 亿元
746         gamma = 1.09
```

```
747         return c1 * (delta_K ** gamma)
748
749     # 净收益评价函数 (在主脚本中调用)
750     # B(delta_K) = vs * [Z(K) - Z(K0)] + vp * [N(K) - N(K0)]
751     # F(delta_K) = B(delta_K) - C(delta_K)
```

C. 聚类函数

```
1   #!/usr/bin/env python3
2   # -*- coding: utf-8 -*-
3   """
4   旅游方案K均值聚类算法
5   使用自定义距离函数来衡量旅游方案之间的相似度
6   """
7
8   import numpy as np
9   import matplotlib.pyplot as plt
10  import random
11  from typing import List, Dict, Tuple, Any
12  from collections import defaultdict
13  import warnings
14  warnings.filterwarnings('ignore')
15
16  class TourismKMeans:
17      def __init__(self, k: int = 3, max_iterations: int = 100, tolerance:
        ↪ float = 1e-4):
18          """
19          初始化K均值聚类器
20
21          Args:
22              k: 聚类数量
23              max_iterations: 最大迭代次数
24              tolerance: 收敛阈值
25          """
26          self.k = k
27          self.max_iterations = max_iterations
28          self.tolerance = tolerance
29          self.centroids = None
30          self.labels = None
31          self.inertia_ = None
32
33      def custom_distance(self, route1: Dict, route2: Dict) -> float:
34          """
35          自定义距离函数：计算两个旅游方案之间的相似度距离
36
37          Args:
38              route1: 第一个旅游方案
39              route2: 第二个旅游方案
```

```
40
41              Returns:
42                  float: 距离值（越小表示越相似）
43              """
44              # 1. 景点相似度（Jaccard相似度）
45              spots1 = set(route1.get('spots', []))
46              spots2 = set(route2.get('spots', []))
47              if len(spots1.union(spots2)) == 0:
48                  spots_similarity = 0
49              else:
50                  spots_similarity = len(spots1.intersection(spots2)) / len(
                        ↪ spots1.union(spots2))
51
52              # 2. 区域相似度（Jaccard相似度）
53              regions1 = set(route1.get('regions', []))
54              regions2 = set(route2.get('regions', []))
55              if len(regions1.union(regions2)) == 0:
56                  regions_similarity = 0
57              else:
58                  regions_similarity = len(regions1.intersection(regions2)) /
                        ↪ len(regions1.union(regions2))
59
60              # 3. 路线类型相似度
61              type1 = route1.get('type', '')
62              type2 = route2.get('type', '')
63              type_similarity = 1.0 if type1 == type2 else 0.0
64
65              # 4. 成本相似度（归一化欧几里得距离）
66              cost1 = route1.get('cost', 0)
67              cost2 = route2.get('cost', 0)
68              max_cost = max(cost1, cost2) if max(cost1, cost2) > 0 else 1
69              cost_similarity = 1 - abs(cost1 - cost2) / max_cost
70
71              # 5. 时间相似度（归一化欧几里得距离）
72              time1 = route1.get('time', 0)
73              time2 = route2.get('time', 0)
74              max_time = max(time1, time2) if max(time1, time2) > 0 else 1
75              time_similarity = 1 - abs(time1 - time2) / max_time
76
77              # 6. 偏好相似度（归一化欧几里得距离）
78              pref1 = route1.get('preference', 0)
79              pref2 = route2.get('preference', 0)
80              max_pref = max(pref1, pref2) if max(pref1, pref2) > 0 else 1
81              pref_similarity = 1 - abs(pref1 - pref2) / max_pref
82
83              # 加权综合距离（转换为距离，越小越相似）
84              weights = {
85                  'spots': 0.25,      # 景点权重
```

```
86                  'regions': 0.25,    # 区域权重
87                  'type': 0.15,       # 类型权重
88                  'cost': 0.15,       # 成本权重
89                  'time': 0.10,       # 时间权重
90                  'preference': 0.10  # 偏好权重
91              }
92
93              total_distance = (
94                  weights['spots'] * (1 - spots_similarity) +
95                  weights['regions'] * (1 - regions_similarity) +
96                  weights['type'] * (1 - type_similarity) +
97                  weights['cost'] * (1 - cost_similarity) +
98                  weights['time'] * (1 - time_similarity) +
99                  weights['preference'] * (1 - pref_similarity)
100             )
101
102             return total_distance
103
104         def find_centroid(self, cluster_routes: List[Dict]) -> Dict:
105             """
106             计算聚类中心（找到与所有路线平均距离最小的路线作为中心）
107
108             Args:
109                 cluster_routes: 聚类中的路线列表
110
111             Returns:
112                 Dict: 聚类中心路线
113             """
114             if not cluster_routes:
115                 return {}
116
117             if len(cluster_routes) == 1:
118                 return cluster_routes[0]
119
120             # 计算每条路线到其他所有路线的平均距离
121             avg_distances = []
122             for i, route in enumerate(cluster_routes):
123                 total_distance = 0
124                 for j, other_route in enumerate(cluster_routes):
125                     if i != j:
126                         total_distance += self.custom_distance(route,
                                ↪ other_route)
127                 avg_distance = total_distance / (len(cluster_routes) - 1)
128                 avg_distances.append((avg_distance, route))
129
130             # 返回平均距离最小的路线作为中心
131             return min(avg_distances, key=lambda x: x[0])[1]
132
```

```
133         def fit(self, routes: List[Dict]) -> 'TourismKMeans':
134             """
135             训练K均值聚类模型
136 
137             Args:
138                 routes: 旅游方案列表
139 
140             Returns:
141                 self: 训练好的模型
142             """
143             if len(routes) < self.k:
144                 raise ValueError(f"路线数量({len(routes)})必须大于聚类数({self
                        ↪ .k})")
145 
146             # 随机初始化聚类中心
147             self.centroids = random.sample(routes, self.k)
148 
149             for iteration in range(self.max_iterations):
150                 # 分配阶段：将每个路线分配到最近的聚类中心
151                 labels = []
152                 total_distance = 0
153 
154                 for route in routes:
155                     min_distance = float('inf')
156                     best_cluster = 0
157 
158                     for i, centroid in enumerate(self.centroids):
159                         distance = self.custom_distance(route, centroid)
160                         if distance < min_distance:
161                             min_distance = distance
162                             best_cluster = i
163 
164                     labels.append(best_cluster)
165                     total_distance += min_distance
166 
167                 # 更新阶段：重新计算聚类中心
168                 new_centroids = []
169                 for i in range(self.k):
170                     cluster_routes = [routes[j] for j in range(len(routes)) if
                            ↪ labels[j] == i]
171                     if cluster_routes:
172                         new_centroid = self.find_centroid(cluster_routes)
173                         new_centroids.append(new_centroid)
174                     else:
175                         # 如果某个聚类为空，随机选择一个新中心
176                         new_centroids.append(random.choice(routes))
177 
178                 # 检查收敛
```

```
                centroid_changed = False
                for i in range(self.k):
                    if self.custom_distance(self.centroids[i], new_centroids[i
                        ↪ ]) > self.tolerance:
                        centroid_changed = True
                        break

                self.centroids = new_centroids

                if not centroid_changed:
                    print(f"算法在第{iteration + 1}次迭代后收敛")
                    break

            self.labels = labels
            self.inertia_ = total_distance

            return self

        def predict(self, routes: List[Dict]) -> List[int]:
            """
            预测新路线的聚类标签

            Args:
                routes: 待预测的路线列表

            Returns:
                List[int]: 聚类标签
            """
            if self.centroids is None:
                raise ValueError("模型尚未训练, 请先调用fit方法")

            labels = []
            for route in routes:
                min_distance = float('inf')
                best_cluster = 0

                for i, centroid in enumerate(self.centroids):
                    distance = self.custom_distance(route, centroid)
                    if distance < min_distance:
                        min_distance = distance
                        best_cluster = i

                labels.append(best_cluster)

            return labels

        def get_cluster_info(self, routes: List[Dict]) -> Dict:
            """
```

```
226             获取聚类信息
227
228             Args:
229                 routes: 原始路线列表
230
231             Returns:
232                 Dict: 聚类信息
233             """
234             if self.labels is None:
235                 raise ValueError("模型尚未训练, 请先调用fit方法")
236
237             cluster_info = {}
238
239             for i in range(self.k):
240                 cluster_routes = [routes[j] for j in range(len(routes)) if
                        ↪ self.labels[j] == i]
241
242                 if cluster_routes:
243                     # 计算聚类统计信息
244                     costs = [route.get('cost', 0) for route in cluster_routes]
245                     times = [route.get('time', 0) for route in cluster_routes]
246                     preferences = [route.get('preference', 0) for route in
                            ↪ cluster_routes]
247                     types = [route.get('type', '') for route in cluster_routes
                            ↪ ]
248
249                     cluster_info[i] = {
250                         'size': len(cluster_routes),
251                         'centroid': self.centroids[i],
252                         'avg_cost': np.mean(costs),
253                         'avg_time': np.mean(times),
254                         'avg_preference': np.mean(preferences),
255                         'type_distribution': dict(zip(*np.unique(types,
                                ↪ return_counts=True))),
256                         'routes': cluster_routes
257                     }
258
259             return cluster_info
260
261         def visualize_clusters(self, routes: List[Dict], save_path: str = None
                ↪ ):
262             """
263             可视化聚类结果
264
265             Args:
266                 routes: 原始路线列表
267                 save_path: 保存图片的路径
268             """
```

```
269             if self.labels is None:
270                 raise ValueError("模型尚未训练，请先调用fit方法")
271 
272             # 提取特征用于可视化
273             costs = [route.get('cost', 0) for route in routes]
274             times = [route.get('time', 0) for route in routes]
275             preferences = [route.get('preference', 0) for route in routes]
276 
277             # 创建子图
278             fig, axes = plt.subplots(2, 2, figsize=(15, 12))
279 
280             # 1. 成本-时间散点图
281             scatter = axes[0, 0].scatter(costs, times, c=self.labels, cmap='
                    ↪ viridis', alpha=0.7)
282             axes[0, 0].set_xlabel('成本 (元)')
283             axes[0, 0].set_ylabel('时间 (分钟)')
284             axes[0, 0].set_title('聚类结果: 成本 vs 时间')
285             plt.colorbar(scatter, ax=axes[0, 0])
286 
287             # 2. 成本-偏好散点图
288             scatter = axes[0, 1].scatter(costs, preferences, c=self.labels,
                    ↪ cmap='viridis', alpha=0.7)
289             axes[0, 1].set_xlabel('成本 (元)')
290             axes[0, 1].set_ylabel('偏好度')
291             axes[0, 1].set_title('聚类结果: 成本 vs 偏好度')
292             plt.colorbar(scatter, ax=axes[0, 1])
293 
294             # 3. 时间-偏好散点图
295             scatter = axes[1, 0].scatter(times, preferences, c=self.labels,
                    ↪ cmap='viridis', alpha=0.7)
296             axes[1, 0].set_xlabel('时间 (分钟)')
297             axes[1, 0].set_ylabel('偏好度')
298             axes[1, 0].set_title('聚类结果: 时间 vs 偏好度')
299             plt.colorbar(scatter, ax=axes[1, 0])
300 
301             # 4. 聚类大小柱状图
302             cluster_sizes = [sum(1 for label in self.labels if label == i) for
                    ↪  i in range(self.k)]
303             axes[1, 1].bar(range(self.k), cluster_sizes, color='skyblue',
                    ↪ alpha=0.7)
304             axes[1, 1].set_xlabel('聚类编号')
305             axes[1, 1].set_ylabel('路线数量')
306             axes[1, 1].set_title('各聚类大小分布')
307             axes[1, 1].set_xticks(range(self.k))
308 
309             plt.tight_layout()
310 
311             if save_path:
```

```
312                 plt.savefig(save_path, dpi=300, bbox_inches='tight')
313
314             plt.show()
315
316
317     def generate_sample_routes(num_routes: int = 100) -> List[Dict]:
318         """
319         生成示例旅游路线数据
320
321         Args:
322             num_routes: 路线数量
323
324         Returns:
325             List[Dict]: 示例路线列表
326         """
327         routes = []
328
329         # 景点和区域信息
330         spots = list(range(6))  # S1-S6
331         regions = list(range(6))  # R1-R6
332         route_types = ['1day', '2day', '3day']
333
334         for i in range(num_routes):
335             route_type = random.choice(route_types)
336
337             if route_type == '1day':
338                 # 一日游：2个景点，1个区域
339                 route_spots = random.sample(spots, 2)
340                 route_regions = random.sample(regions, 1)
341                 cost = random.randint(20, 80)
342                 time = random.randint(60, 180)
343                 preference = random.randint(5, 15)
344
345             elif route_type == '2day':
346                 # 二日游：4个景点，3个区域
347                 route_spots = random.sample(spots, 4)
348                 route_regions = random.sample(regions, 3)
349                 cost = random.randint(80, 200)
350                 time = random.randint(180, 360)
351                 preference = random.randint(15, 25)
352
353             else:  # 3day
354                 # 三日游：6个景点，5个区域
355                 route_spots = random.sample(spots, 6)
356                 route_regions = random.sample(regions, 5)
357                 cost = random.randint(200, 400)
358                 time = random.randint(360, 600)
359                 preference = random.randint(25, 35)
```

```
360
361             route = {
362                 'id': i,
363                 'type': route_type,
364                 'spots': route_spots,
365                 'regions': route_regions,
366                 'cost': cost,
367                 'time': time,
368                 'preference': preference
369             }
370
371             routes.append(route)
372
373         return routes
374
375
376     def main():
377         """主函数: 演示K均值聚类算法"""
378         print("=" * 60)
379         print("旅游方案K均值聚类算法演示")
380         print("=" * 60)
381
382         # 生成示例数据
383         print("生成示例旅游路线数据...")
384         routes = generate_sample_routes(200)
385         print(f"生成了 {len(routes)} 条旅游路线")
386
387         # 创建并训练K均值模型
388         print("\n训练K均值聚类模型...")
389         kmeans = TourismKMeans(k=4, max_iterations=50)
390         kmeans.fit(routes)
391
392         # 获取聚类信息
393         print("\n聚类结果分析:")
394         cluster_info = kmeans.get_cluster_info(routes)
395
396         for cluster_id, info in cluster_info.items():
397             print(f"\n聚类 {cluster_id}:")
398             print(f"  路线数量: {info['size']}")
399             print(f"  平均成本: {info['avg_cost']:.2f} 元")
400             print(f"  平均时间: {info['avg_time']:.2f} 分钟")
401             print(f"  平均偏好度: {info['avg_preference']:.2f}")
402             print(f"  路线类型分布: {info['type_distribution']}")
403
404         # 可视化聚类结果
405         print("\n生成聚类可视化图表...")
406         kmeans.visualize_clusters(routes, save_path='tourism_clusters.png')
407
```

```
        print(f"\n聚类完成！模型惯性（总距离）：{kmeans.inertia_:.4f}")


if __name__ == "__main__":
    main()
```

D. 运行函数

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
旅游路线优化运行脚本
"""
import deap
from tourism_optimization import TourismOptimizer, expansion_cost # Import expansion cost function
import numpy as np # Import numpy
import matplotlib.pyplot as plt
import matplotlib # Import matplotlib to configure fonts

# Import K-means clustering algorithm
from Q2_k_means import TourismKMeans

# --- Start: Matplotlib Chinese Font Configuration ---
# Configure font to support Chinese characters
# Try 'SimHei' first, if not available, try 'Microsoft YaHei' or 'WenQuanYi Micro Hei'
matplotlib.rcParams['font.sans-serif'] = ['SimHei', 'Microsoft YaHei', 'Arial Unicode MS']
matplotlib.rcParams['axes.unicode_minus'] = False # Solve the problem of '-' displaying as a square
# --- End: Matplotlib Chinese Font Configuration ---

def main():
    print("=" * 60)
    print("重庆旅游路线优化系统")
    print("=" * 60)

    # Create optimizer instance
    optimizer = TourismOptimizer()

    # Run optimization
    try:
        # Test different weight configurations
        weight_configs = [
            (0.5, 0.5, "均衡型"),
            # (0.7, 0.3, "费用优先型"), # Can comment out other configurations for faster expansion analysis
```

```
36                 # (0.3, 0.7, "时间优先型")
37             ]
38
39             all_results = {}
40
41             for cost_w, time_w, config_name in weight_configs:
42                 print(f"\n{'='*80}")
43                 print(f"运行{config_name}配置 (费用权重={cost_w}, 时间权重={
                       ↪ time_w})")
44                 print(f"{'='*80}")
45
46                 # Run optimization and get results
47                 results = optimizer.run_optimization(cost_weight=cost_w,
                       ↪ time_weight=time_w)
48                 all_results[config_name] = results
49
50                 # Print results
51                 optimizer.print_results(results)
52
53                 # Save results to file
54                 save_results_to_file(results, suffix=f"_{config_name}")
55
56             # Compare results of different configurations
57             compare_configurations(all_results)
58
59             # ======= Problem 3: New Hotel Expansion Analysis =======
60             print(f"\n{'='*80}")
61             print("问题3: 新建旅馆扩容分析")
62             print(f"{'='*80}")
63             analyze_hotel_expansion(optimizer)
64
65             # ======= New: K-means Clustering Analysis =======
66             print(f"\n{'='*80}")
67             print("K均值聚类分析: 旅游路线相似度分析")
68             print(f"{'='*80}")
69             perform_kmeans_analysis(all_results)
70
71         except Exception as e:
72             print(f"优化过程中出现错误: {e}")
73             import traceback
74             traceback.print_exc()
75
76     def compare_configurations(all_results):
77         """比较不同权重配置的结果"""
78         print(f"\n{'='*80}")
79         print("不同权重配置结果比较")
80         print(f"{'='*80}")
81
```

```
82          for day_type in ['1day', '2day', '3day']:
83              print(f"\n{day_type.upper()} 方案比较:")
84              print("-" * 60)
85              print(f"{'配置':<12} {'路线数':<8} {'总游客':<8} {'总费用':<12} {'
                    ↪ 总时间':<12} {'平均满意度':<12}")
86              print("-" * 60)
87
88              for config_name, results in all_results.items():
89                  solution = results.get(day_type, [])
90                  if solution:
91                      total_tourists = sum([route['tourists'] for route in
                            ↪ solution])
92                      total_cost = sum([route['tourists'] * route['cost'] for
                            ↪ route in solution])
93                      total_time = sum([route['tourists'] * route['time'] for
                            ↪ route in solution])
94                      total_satisfaction = sum([route['tourists'] * route['
                            ↪ satisfaction'] for route in solution])
95                      avg_satisfaction = total_satisfaction / total_tourists if
                            ↪ total_tourists > 0 else 0
96
97                      print(f"{config_name:<12} {len(solution):<8} {
                            ↪ total_tourists:<8} {total_cost:<12.0f} "
98                            f"{total_time:<12.0f} {avg_satisfaction:<12.2f}")
99                  else:
100                     print(f"{config_name:<12} {'无解':<8} {'-':<8} {'-':<12}
                            ↪ {'-':<12} {'-':<12}")
101
102
103 def save_results_to_file(results, suffix=""):
104     """将结果保存到文件"""
105     filename = f'optimization_results{suffix}.txt'
106     with open(filename, 'w', encoding='utf-8') as f:
107         f.write("重庆旅游路线优化结果\n")
108         f.write("=" * 50 + "\n\n")
109
110         for day_type_key, solution_or_metrics in results.items():
111             if not isinstance(solution_or_metrics, list): # Filter out
                    ↪ metrics dictionary
112                 continue
113             solution = solution_or_metrics
114
115             day_type = day_type_key.replace('_metrics', '') # Get actual
                    ↪ day_type
116
117             f.write(f"{day_type.upper()} 旅游方案结果\n")
118             f.write("=" * 30 + "\n")
119
```

```
                if not solution:
                    f.write("无可行解\n\n")
                    continue

                total_tourists = sum([route['tourists'] for route in solution
                    ↪ ])
                total_cost = sum([route['tourists'] * route['cost'] for route
                    ↪ in solution])
                total_time = sum([route['tourists'] * route['time'] for route
                    ↪ in solution])
                total_satisfaction = sum([route['tourists'] * route['
                    ↪ satisfaction'] for route in solution])

                f.write(f"路线类型数量: {len(solution)}\n")
                f.write(f"总游客数: {total_tourists}\n")
                f.write(f"总交通成本: {total_cost:.0f} 元\n")
                f.write(f"总交通时间: {total_time:.0f} 分钟\n")
                f.write(f"总满意度: {total_satisfaction:.2f}\n")
                f.write(f"平均满意度: {total_satisfaction/total_tourists:.2f}\
                    ↪ n\n")

                f.write("具体路线分配:\n")
                for i, route in enumerate(solution[:10]):
                    spots_str = "->".join([f"S{s+1}" for s in route['spots']])
                    regions_str = "->".join([f"R{r+1}" for r in route['regions
                        ↪ ']])
                    f.write(f"路线{i+1}: {route['tourists']}人\n")
                    f.write(f"  景点路径: {spots_str}\n")
                    f.write(f"  区域路径: {regions_str}\n")
                    f.write(f"  成本: {route['cost']:.1f}元, 时间: {route['
                        ↪ time']:.1f}分钟, 偏好: {route['preference']}\n\n")
                f.write("\n")

        print(f"\n结果已保存到 {filename} 文件")


    def analyze_hotel_expansion(optimizer: TourismOptimizer):
        """
        Analyzes the hotel expansion problem to find the optimal expansion
            ↪ area and scale.
        """
        print("\n开始分析旅馆扩容方案...")

        # Define economic value parameters
        # vs: Economic value per unit satisfaction (billion CNY/satisfaction)
        # vp: Average profit per tourist (billion CNY/person)
        v_s_per_million_satisfaction = 0.1 # Assume 1 million satisfaction
            ↪ brings 0.1 billion CNY
```

```
159         v_p_per_million_tourists = 0.01 # Assume 1 million tourists brings
                ↪ 0.01 billion CNY (10 CNY/person, 1 million * 10 = 10 million CNY
                ↪  = 0.01 billion CNY)
160
161         # Run baseline optimization to get metrics at K0
162         print("获取基准指标 (扩容前)...")
163         base_results = optimizer.run_optimization(cost_weight=0.5, time_weight
                ↪ =0.5, tourists_1day=30000, tourists_2day=30000, tourists_3day
                ↪ =20000)
164         Z_K0_total, N_K0_total = optimizer.get_aggregated_metrics(base_results
                ↪ )
165
166         # Convert number of tourists to millions to match delta_K unit
167         N_K0_total_wan = N_K0_total / 10000.0
168
169         print(f"基准总满意度 Z(K0): {Z_K0_total:.2f}")
170         print(f"基准总游客数 N(K0): {N_K0_total_wan:.2f} 万人次")
171
172         best_net_benefit = -np.inf
173         best_region_idx = -1
174         best_delta_K_wan = 0
175
176         # Define expansion capacity range (in millions of persons)
177         # Consider from 0 to a reasonable maximum value, e.g., twice the
                ↪ original maximum capacity
178         max_delta_K_wan = max(optimizer.night_cap_original) / 10000.0 * 2 #
                ↪ Twice the original max capacity, in millions of persons
179         delta_K_wan_range = np.arange(0, max_delta_K_wan + 1, 0.5) # From 0 to
                ↪  max_delta_K_wan, step 0.5 million persons
180
181         # Store all test results
182         expansion_results = []
183
184         print("\n迭代各区域及扩容规模进行模拟...")
185         for r_idx in range(optimizer.num_regions):
186             original_night_cap_r = optimizer.night_cap_original[r_idx] # Get
                    ↪ the original capacity of this region
187
188             for delta_K_wan in delta_K_wan_range:
189                 current_night_cap_r = original_night_cap_r + (delta_K_wan *
                        ↪ 10000) # Convert millions of persons back to persons
190
191                 # Temporarily modify the accommodation capacity of this region
192                 optimizer.night_cap[r_idx] = current_night_cap_r
193
194                 # Rerun optimization
195                 # Note: run_optimization will regenerate routes, ensuring each
                        ↪  run is based on the modified capacity
```

```
                current_results = optimizer.run_optimization(
                    cost_weight=0.5, time_weight=0.5, # Use balanced weights for expansion analysis
                    tourists_1day=30000, tourists_2day=30000, tourists_3day=20000
                )
                Z_K_total, N_K_total = optimizer.get_aggregated_metrics(current_results)
                N_K_total_wan = N_K_total / 10000.0

                # Calculate benefit B(delta_K)
                B_delta_K = (v_s_per_million_satisfaction * (Z_K_total - Z_K0_total) +
                             v_p_per_million_tourists * (N_K_total_wan - N_K0_total_wan))

                # Calculate cost C(delta_K)
                C_delta_K = expansion_cost(delta_K_wan)

                # Calculate net benefit F(delta_K)
                F_delta_K = B_delta_K - C_delta_K

                expansion_results.append({
                    'region_idx': r_idx,
                    'region_name': f'区域{r_idx + 1}',
                    'delta_K_wan': delta_K_wan,
                    'original_night_cap_wan': original_night_cap_r / 10000.0,
                    'new_night_cap_wan': current_night_cap_r / 10000.0,
                    'Z_K_total': Z_K_total,
                    'N_K_total_wan': N_K_total_wan,
                    'Benefit_B': B_delta_K,
                    'Cost_C': C_delta_K,
                    'Net_Benefit_F': F_delta_K
                })

                # Update optimal solution
                if F_delta_K > best_net_benefit:
                    best_net_benefit = F_delta_K
                    best_region_idx = r_idx
                    best_delta_K_wan = delta_K_wan

            # Restore the original capacity of this region for the next region's test
            optimizer.night_cap[r_idx] = optimizer.night_cap_original[r_idx]

        print("\n扩容分析结果:")
        print("-" * 60)
        print(f"{'区域':<8} {'扩容规模(万人次)':<18} {'原容量(万人次)':<16} {'
```

```
            ↪ 新容量(万人次)':<16} {'净收益(亿元)':<12}")
238         print("-" * 60)
239
240         # Print the top 20 net benefit solutions
241         sorted_expansion_results = sorted(expansion_results, key=lambda x: x['
                ↪ Net_Benefit_F'], reverse=True)
242         for i, res in enumerate(sorted_expansion_results[:20]):
243             print(f"{res['region_name']:<8} {res['delta_K_wan']:<18.2f} {res['
                    ↪ original_night_cap_wan']:<16.2f} {res['new_night_cap_wan
                    ↪ ']:<16.2f} {res['Net_Benefit_F']:<12.4f}")
244
245         print(f"\n最终建议: ")
246         if best_region_idx != -1:
247             print(f"建议扩容区域: 区域{best_region_idx + 1}")
248             print(f"建议扩容规模: {best_delta_K_wan:.2f} 万人次")
249             print(f"最大净收益: {best_net_benefit:.4f} 亿元")
250             print(f"该区域原有住宿接待能力: {optimizer.night_cap_original[
                    ↪ best_region_idx]/10000:.2f} 万人次")
251             print(f"该区域扩容后住宿接待能力: {(optimizer.night_cap_original[
                    ↪ best_region_idx] + best_delta_K_wan * 10000)/10000:.2f} 万人
                    ↪ 次")
252
253             # Plot net benefit
254             plt.figure(figsize=(12, 6))
255
256             region_colors = plt.cm.tab10 # Use matplotlib's color map
257
258             # Filter data for each region
259             regions_data = {r_idx: [] for r_idx in range(optimizer.num_regions
                    ↪ )}
260             for res in expansion_results:
261                 regions_data[res['region_idx']].append(res)
262
263             for r_idx, data in regions_data.items():
264                 data.sort(key=lambda x: x['delta_K_wan'])
265                 delta_K_wan_values = [d['delta_K_wan'] for d in data]
266                 net_benefit_values = [d['Net_Benefit_F'] for d in data]
267
268                 plt.plot(delta_K_wan_values, net_benefit_values, marker='o',
                        ↪ linestyle='-',
269                          label=f'区域 {r_idx + 1}', color=region_colors(r_idx)
                            ↪ )
270
271             plt.title('不同区域及扩容规模下的净收益')
272             plt.xlabel('新增接待能力 (万人次)')
273             plt.ylabel('净收益 (亿元)')
274             plt.grid(True)
275             plt.legend(title='区域')
```

```
276             plt.axhline(0, color='grey', linestyle='--', linewidth=0.8) # Zero
                    ↪ benefit line
277             plt.scatter(best_delta_K_wan, best_net_benefit, color='red',
                    ↪ marker='X', s=200,
278                         label=f'最优解: 区域{best_region_idx+1}, {
                                ↪ best_delta_K_wan:.2f}万人次, {best_net_benefit
                                ↪ :.4f}亿元')
279             plt.legend()
280             plt.tight_layout()
281             plt.savefig('net_benefit_vs_expansion.png')
282             plt.show()
283
284         else:
285             print("未找到可行的扩容方案, 或者所有扩容方案都导致负净收益。")
286
287
288     def perform_kmeans_analysis(all_results):
289         """
290         对优化结果进行K均值聚类分析
291
292         Args:
293             all_results: 优化结果字典, 包含不同权重配置的结果
294         """
295         print("\n开始K均值聚类分析...")
296
297         # 1. 收集所有路线数据
298         all_routes = []
299         route_id = 0
300
301         for config_name, results in all_results.items():
302             for day_type in ['1day', '2day', '3day']:
303                 solution = results.get(day_type, [])
304
305                 for route in solution:
306                     # 为K均值算法准备路线数据
307                     route_data = {
308                         'id': route_id,
309                         'config': config_name,
310                         'type': day_type,
311                         'spots': route.get('spots', []),
312                         'regions': route.get('regions', []),
313                         'cost': route.get('cost', 0),
314                         'time': route.get('time', 0),
315                         'preference': route.get('preference', 0),
316                         'tourists': route.get('tourists', 0),
317                         'satisfaction': route.get('satisfaction', 0)
318                     }
319                     all_routes.append(route_data)
```

```
                    route_id += 1

        if not all_routes:
            print("没有找到可用的路线数据进行聚类分析")
            return

        print(f"收集到 {len(all_routes)} 条路线用于聚类分析")

        # 2. 进行K均值聚类
        # 尝试不同的k值
        k_values = [3, 4, 5, 6]
        best_k = 3
        best_inertia = float('inf')

        print("\n选择最优聚类数量...")
        for k in k_values:
            if k <= len(all_routes):
                kmeans = TourismKMeans(k=k, max_iterations=50)
                kmeans.fit(all_routes)
                print(f"k={k}: 惯性值={kmeans.inertia_:.4f}")

                if kmeans.inertia_ < best_inertia:
                    best_inertia = kmeans.inertia_
                    best_k = k

        print(f"\n选择最优k值: {best_k}")

        # 3. 使用最优k值进行最终聚类
        print(f"\n使用k={best_k}进行最终聚类...")
        final_kmeans = TourismKMeans(k=best_k, max_iterations=100)
        final_kmeans.fit(all_routes)

        # 4. 分析聚类结果
        print(f"\n聚类结果分析:")
        cluster_info = final_kmeans.get_cluster_info(all_routes)

        print("-" * 100)
        print(f"{'聚类':<6} {'大小':<6} {'平均成本':<10} {'平均时间':<10} {'平均偏好':<10} {'平均游客':<10} {'主要类型':<15}")
        print("-" * 100)

        for cluster_id, info in cluster_info.items():
            main_type = max(info['type_distribution'].items(), key=lambda x: x[1])[0] if info['type_distribution'] else "未知"
            avg_tourists = np.mean([route.get('tourists', 0) for route in info['routes']])

            print(f"{cluster_id:<6} {info['size']:<6} {info['avg_cost']:<10.1f
```

```
                ↪ } {info['avg_time']:<10.1f} "
365             f"{info['avg_preference']:<10.1f} {avg_tourists:<10.1f} {
                    ↪ main_type:<15}")
366
367         # 5. 详细分析每个聚类
368         print(f"\n{'='*80}")
369         print("详细聚类分析:")
370         print(f"{'='*80}")
371
372         for cluster_id, info in cluster_info.items():
373             print(f"\n聚类 {cluster_id} 详细信息:")
374             print(f"  - 路线数量: {info['size']}")
375             print(f"  - 平均成本: {info['avg_cost']:.2f} 元")
376             print(f"  - 平均时间: {info['avg_time']:.2f} 分钟")
377             print(f"  - 平均偏好度: {info['avg_preference']:.2f}")
378             print(f"  - 路线类型分布: {info['type_distribution']}")
379
380             # 分析配置分布
381             configs = [route.get('config', '未知') for route in info['routes'
                    ↪ ]]
382             config_dist = {}
383             for config in configs:
384                 config_dist[config] = config_dist.get(config, 0) + 1
385             print(f"  - 权重配置分布: {config_dist}")
386
387             # 分析景点和区域偏好
388             all_spots = []
389             all_regions = []
390             for route in info['routes']:
391                 all_spots.extend(route.get('spots', []))
392                 all_regions.extend(route.get('regions', []))
393
394             if all_spots:
395                 popular_spots = {}
396                 for spot in all_spots:
397                     popular_spots[f'S{spot+1}'] = popular_spots.get(f'S{spot
                            ↪ +1}', 0) + 1
398                 popular_spots = sorted(popular_spots.items(), key=lambda x: x
                        ↪ [1], reverse=True)[:3]
399                 print(f"  - 热门景点: {[f'{s}({c}次)' for s, c in
                        ↪ popular_spots]}")
400
401             if all_regions:
402                 popular_regions = {}
403                 for region in all_regions:
404                     popular_regions[f'R{region+1}'] = popular_regions.get(f'R{
                            ↪ region+1}', 0) + 1
405                 popular_regions = sorted(popular_regions.items(), key=lambda x
```

```
                    ↪ : x[1], reverse=True)[:3]
406                 print(f"  - 热门区域: {[f'{r}({c}次)' for r, c in
                        ↪ popular_regions]}")
407
408         # 6. 可视化聚类结果
409         print(f"\n生成聚类可视化图表...")
410         final_kmeans.visualize_clusters(all_routes, save_path='
                ↪ tourism_optimization_clusters.png')
411
412         # 7. 保存聚类结果到文件
413         save_clustering_results(final_kmeans, all_routes, cluster_info)
414
415         print(f"\nK均值聚类分析完成!")
416         print(f"最终聚类数: {best_k}")
417         print(f"模型惯性值: {final_kmeans.inertia_:.4f}")
418         print(f"结果已保存到 clustering_results.txt 和
                ↪ tourism_optimization_clusters.png")
419
420
421     def save_clustering_results(kmeans_model, routes, cluster_info):
422         """
423         保存聚类结果到文件
424
425         Args:
426             kmeans_model: 训练好的K均值模型
427             routes: 路线数据
428             cluster_info: 聚类信息
429         """
430         filename = 'clustering_results.txt'
431         with open(filename, 'w', encoding='utf-8') as f:
432             f.write("旅游路线K均值聚类分析结果\n")
433             f.write("=" * 50 + "\n\n")
434
435             f.write(f"聚类数量: {kmeans_model.k}\n")
436             f.write(f"路线总数: {len(routes)}\n")
437             f.write(f"模型惯性值: {kmeans_model.inertia_:.4f}\n\n")
438
439             for cluster_id, info in cluster_info.items():
440                 f.write(f"聚类 {cluster_id}:\n")
441                 f.write("-" * 30 + "\n")
442                 f.write(f"路线数量: {info['size']}\n")
443                 f.write(f"平均成本: {info['avg_cost']:.2f} 元\n")
444                 f.write(f"平均时间: {info['avg_time']:.2f} 分钟\n")
445                 f.write(f"平均偏好度: {info['avg_preference']:.2f}\n")
446                 f.write(f"路线类型分布: {info['type_distribution']}\n")
447
448                 # 配置分布
449                 configs = [route.get('config', '未知') for route in info['
```

```
                    ↪ routes']]
450             config_dist = {}
451             for config in configs:
452                 config_dist[config] = config_dist.get(config, 0) + 1
453             f.write(f"权重配置分布: {config_dist}\n")
454 
455             # 热门景点和区域
456             all_spots = []
457             all_regions = []
458             for route in info['routes']:
459                 all_spots.extend(route.get('spots', []))
460                 all_regions.extend(route.get('regions', []))
461 
462             if all_spots:
463                 popular_spots = {}
464                 for spot in all_spots:
465                     popular_spots[f'S{spot+1}'] = popular_spots.get(f'S{
                        ↪ spot+1}', 0) + 1
466                 popular_spots = sorted(popular_spots.items(), key=lambda x
                    ↪ : x[1], reverse=True)[:5]
467                 f.write(f"热门景点: {popular_spots}\n")
468 
469             if all_regions:
470                 popular_regions = {}
471                 for region in all_regions:
472                     popular_regions[f'R{region+1}'] = popular_regions.get(
                        ↪ f'R{region+1}', 0) + 1
473                 popular_regions = sorted(popular_regions.items(), key=
                    ↪ lambda x: x[1], reverse=True)[:5]
474                 f.write(f"热门区域: {popular_regions}\n")
475 
476             f.write("\n代表性路线 (前5条):\n")
477             for i, route in enumerate(info['routes'][:5]):
478                 spots_str = "->".join([f"S{s+1}" for s in route.get('spots
                    ↪ ', [])])
479                 regions_str = "->".join([f"R{r+1}" for r in route.get('
                    ↪ regions', [])])
480                 f.write(f"  {i+1}. {route.get('type', 'unknown')}路线 ({
                    ↪ route.get('config', 'unknown')}配置)\n")
481                 f.write(f"     景点: {spots_str}\n")
482                 f.write(f"     区域: {regions_str}\n")
483                 f.write(f"     成本: {route.get('cost', 0):.1f}元, 时间: {
                    ↪ route.get('time', 0):.1f}分钟\n")
484                 f.write(f"     游客数: {route.get('tourists', 0)}, 满意度:
                    ↪  {route.get('satisfaction', 0):.3f}\n\n")
485 
486             f.write("\n")
487 
```

```
488
489     if __name__ == "__main__":
490         main()
```