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

摘 要

针对问题一，首先将题目中

针对问题二，我们通过训练一个

针对问题三，建立了

关键词：MILP；遗传算法；旅游线路规划；

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：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，我们要建立套餐筛选与整合准则，对原方案进行精简与再优化，输出满足数量上限且整体效益最优的套餐集合。

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，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。

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。比如，从景点 TODO 到区域 TODO，以其为中转，再到景区 TODO。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。

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

在第二问中，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，我们首先将模型提升为混合整数规划，在决策层面新增 0-1 变量 y_p，通过约束 $\sum y_p \leq 10$ 与“大 M”逻辑联结保证仅有至多 10 条路线被激活；这一严格控制可直接给出受限情形下的全局最优，但求解规模膨胀、计算代价高。为兼顾可行性，我们在算法层面先对可行路线做预筛：依据单位满意度、覆盖度与相似度剔除明显次优或冗余方案，仅保留贡献高且互补性强的候选集进入优化；随后采用两阶段策略，先在精简集合上求线性松弛或 MILP 近优解，再按贡献度排序保留前 10 条并用贪心回填剩余游客。最终形成的综合流程是：生成全部可行路线并时长过滤，预筛精简候选，加入套餐数上限求解 MILP，如遇大规模求解超时则回退启发式结果。该折中方案既在合理计算时间内满足“套餐 ≤10”这一运营约束，又保证总体满意度与资源利用接近原始最优水平，实现了模型对管理需求的灵活适应。

2.3 问题三

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 TODO。

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）
y_p	0-1 决策变量，$y_p = 1$ 表示采用套餐 p
$w_{\text{cost}}, w_{\text{time}}$	费用与时间权重系数
$\mathrm{Cost}(p)$	套餐 p 总交通费用
$\mathrm{Time}(p)$	套餐 p 总交通时间
Q_i	景点 i 基础吸引力评分
U_p	套餐 p 单位游客满意度
A_i	景点 i 吸引力
$\tilde{C}_i$	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
$C(\Delta K)$	扩容成本 $c_1(\Delta K)^\gamma$, $1 < \gamma \le 1.2$
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。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。

$$Cost(p) = \tag{5.2}$$

$$Time(p) = \tag{5.3}$$

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：**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}TODO，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。

均衡型：

经济型：

时间节约型：

5.2 问题二：方案优化

5.2.1 问题分析与建模目标

当要求旅游套餐种类总数不超过 10 种时，需要对上述模型进行适应性修改。原模型在追求最大满意度下，可能启用许多不同路线方案来绕开局部容量瓶颈，从而方案种类较多。如果限制套餐类型数 ≤ 10，模型必须在满意度与管理简化之间折中。我们做出了如下调整：

加入套餐数量约束: 增加约束 $\sum_{p \in P} y_p \leq 10$，限制被选用的不同路线方案数不超过 10 种。其中 y_p 为 0-1 变量，表示方案 p 是否被采用。并引入逻辑约束 $x_p \leq N_{\text{total}}; y_p$（或大 M 法）将 x_p 与 y_p 关联。这样仅当 $y_p = 1$ 时，x_p 可取正值，否则该路线不分配游客。此调整实质上将线性规划扩展为混合整数规划，有助于直接在求解过程中控制方案种类数上限。

精选候选路线集: 在模型求解前预先筛选和精简路线集合 P，只保留若干“优选”套餐候选。比如，根据单条路线的满意度系数 u_p 对候选方案排序，选取满意度高且覆盖面广的前若干条路线进入优化。原模型生成的路线非常多（一日游 $6 \times 5 \times 6 = 180$ 种，二日游上限 10^4 种，三日游经遗传算法 + 启发式生成 5000 种)，有许多方案实际贡献的游客数很小甚至为 0。我们可剔除明显次优或冗余的方案，例如去除行程过长费用偏高而偏好分较低的组合，或者对行程相似且收益接近的方案只保留其中一种。这相当于在不增加过多成本的前提下人工设定候选集精简至 ≤ 10 条路线，然后在此基础上重新优化分配。这种路线筛选（如按满意度排序取 Top 10）和聚类归并的方法可极大减少方案种类，同时尽量保证总满意度接近原最优值。

优先级调度与容量放宽: 在仅允许提供有限几种套餐时，可考虑适当调整对容量约束的处理，确保主要景点和区域得到利用。例如，将重点偏好的景点设计为* 固定线路 **，优先占用一定游客量，然后对剩余游客采用次优线路。在代码实现层面，可以对生成的候选路线按照偏好得分或单位满意度收益排序，优先选取少数几条

高效路线分配游客直到触及某一容量约束，再引入下一优方案。这种贪心分配过程可以模拟人工规划几种典型套餐的思路：例如优先设计几条涵盖热门景区且总体花费合理的线路，再视剩余接待资源补充其他路线，从而将总套餐数控制在上限之内。

综上，当限制套餐总数不超过 10 种时，模型需要从“穷尽所有可能组合求全局最优”转变为“在有限方案中求近优”。这要求在模型中增加 ** 组合选择的决策变量和约束 **，或采用启发式策略筛选方案。通过上述调整，既能满足管理方便的要求，又尽可能兼顾游客满意度和资源利用的均衡。模型的结构与求解思路具有一定灵活性：无论是通过加入 y_p 变量实现严格的种类约束，还是通过缩减候选路线集来隐含满足要求，都体现了对原优化模型的适应性改进。

5.2.2 模型求解和结果分析

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：

$$C(\Delta K) = c_1(\Delta K)^{\gamma}, \quad c_1 = 7 \times 10^{-4}\text{(亿元)}, \gamma = 1.09 \tag{5.10}$$

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 $B(\Delta K)$：

$$B(\Delta K) = v_s[Z(K) - Z(K^0)] + v_p[N(K) - N(K^0)] \tag{5.11}$$

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:

$$\max_{r,\Delta K \geq 0} F_r(\Delta K) = B_r(\Delta K) - C(\Delta K) \tag{5.12}$$

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：

$$\underset{r,\Delta K\geq 0}{max}\ F_r(\Delta K) \tag{5.13}$$

内层：

$$\begin{aligned} Z(K), N(K) = arg \quad & \underset{x}{max} \sum_i s_i x_i \\ & s.t. \text{容量约束} K \end{aligned} \tag{5.14}$$

具体算法如下所示：

算法 5.1: 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

6 模型评价与推广

6.1 主要结论

本文针对 LED 显示器颜色转换与校正问题，建立了基于 CIE Lab 色彩空间和感知色差理论的数学模型，采用多种优化算法实现了高精度的颜色处理。主要研究结论如下：

（1）BT.2020 到 sRGB 颜色空间转换模型

构建了基于 ΔE_{00} 感知误差最小化的优化模型，通过差分进化算法求解最优线性映射矩阵。在 50 次独立实验中，平均 ΔE_{00} 损失值为 0.0744，远低于人眼可察觉阈值；色域面积差异控制在 0.001 以内，映射后色度三角与标准 sRGB 色域几乎完全重合。

（2）多通道颜色空间转换神经网络模型

设计了 ColorNet 神经网络架构，采用混合损失函数成功解决 4 通道到 5 通道的颜色转换问题。混合损失函数结合 MSE 数值精度与 ΔE_{00} 感知准确性，优先保证视觉效果；验证集上 ΔE_{00} 误差主要集中在较低范围。

（3）LED 显示器颜色校正优化模型

建立了结合伽马校正与线性矩阵变换的综合校正模型，采用差分进化与 L-BFGS-B 混合优化策略。三种基色图像平均改善幅度达 95.6%，校正后平均色差降至 0.095；100% 像素达到 $\Delta E < 1.0$ 的优秀标准；校正矩阵行列式值约 0.10，保证了数值稳定性。

6.2 模型优点

（1）理论基础扎实：基于 CIE Lab 色彩空间和国际标准色差公式，确保了颜色处理的科学性和准确性。

（2）技术方法先进：采用差分进化算法、神经网络和混合优化策略，有效处理非线性、维度不匹配等复杂问题。

（3）实用价值突出：校正流程简洁高效，数值稳定性良好，实验验证充分，适合实际工程应用。

6.3 不足与改进方向

（1）主要局限

线性映射矩阵可能无法充分捕捉复杂的非线性颜色响应关系；神经网络模型使用模拟数据训练，与真实设备数据可能存在差异；对环境光照、设备老化等外在因素考虑有限。

（2）改进方向

探索非线性映射方法，结合多模态数据融合，开发实时自适应校正算法；将模型应用于 HDR 显示、VR/AR 设备等专业领域；推动建立跨平台颜色校正标准，促进技术普及应用。

总之，本文为 LED 显示器颜色处理提供了完整的理论框架和实用解决方案，在颜色空间转换、多通道映射和颜色校正等关键环节均实现了技术突破，为高质量显示技术发展奠定了基础。

参考文献

[1] Poynton C. Digital video and hd: algorithms and interfaces[J]. Elsevier, 2012.

[2] Sugawara M, Choi S Y, Wood D. Ultra-high-definition television (rec. itu-r bt.2020): A generational leap in the evolution of television [standards in a nutshell][J/OL]. IEEE Signal Processing Magazine, 2014, 31(3): 170-174. DOI: 10.1109/MSP.2014.2302331.

[3] Fairman H S, Brill M H, Hemmendinger H. How the cie 1931 color-matching functions were derived from wright-guild data[J]. Color Research & Application: Endorsed by Inter-Society Color Council, The Colour Group (Great Britain), Canadian Society for Color, Color Science Association of Japan, Dutch Society for the Study of Color, The Swedish Colour Centre Foundation, Colour Society of Australia, Centre Français de la Couleur, 1997, 22(1): 11-23.

[4] Hunter R S. Photoelectric color difference meter[J]. Journal of the Optical Society of America, 1958, 48(12): 985-995.

[5] 郑元林刘士伟. 最新色差公式:CIEDE2000[J]. 印刷质量与标准化, 2004(07): 34-37.

[6] Gonzalez R C, Woods R E. Gamma correction in digital image processing[J]. Digital Image Processing, 2018: 156-189.

[7] Price K, Storn R M, Lampinen J A. Differential evolution: a handbook for global permutation-based combinatorial optimization[J]. Springer Science & Business Media, 2013.

[8] 王广志. 基于 ColorNet 架构的比色传感器阵列用于猪肉新鲜度检测[D]. 东北电力大学, 2024.

附 录

A. 支撑材料总览

本论文的所有支撑材料组织在 MathModel_Code/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	B/p1/p1.py	问题 1：BT.2020 到 sRGB 色域映射优化
	B/p2/p2.py	问题 2：四通道到五通道神经网络转换
	B/p3/p3.py	问题 3：LED 显示器颜色校正算法
原始数据	data/origin/xlsx/B 题附件：RGB 数值.xlsx	题目提供的原始 RGB 数值
	data/preprocess/RedPicture.xlsx	预处理后的红色基图数据
	data/preprocess/GreenPicture.xlsx	预处理后的绿色基图数据
	data/preprocess/BluePicture.xlsx	预处理后的蓝色基图数据
结果图像	results/p1/DE2000.png	问题 1：50 次优化 ΔE_{00} 分布
	results/p1/色度.png	问题 1：CIE1931 色度图对比
	results/p1/面积 Loss.png	问题 1：色域面积差异分析
	results/p2/Training_Loss_Curve.png	问题 2：神经网络训练损失曲线
	results/p2/ΔE2000_Error_Histogram.png	问题 2：色差误差分布直方图
	results/p2/CDF.png	问题 2：误差累积分布函数
	results/p2/Sample.png	问题 2：样本预测效果展示
	results/p2/色度图.png	问题 2：多基色色域可视化
	results/p3/{R,G,B}.pdf	问题 3：RGB 三原色校正对比图
环境配置	env.txt	Python 依赖包列表
	mathmodel_env.yaml	Conda 环境配置文件
说明文档	README.md	项目使用说明和运行指南

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 = []
            self.routes_2day = []
            self.routes_3day_candidates = []

        def preprocess_matrices(self, cost_weight=0.5, time_weight=0.5):
            """使用加权综合Floyd-Warshall算法补全费用和时间矩阵

            Args:
                cost_weight: 费用权重 (默认0.5)
                time_weight: 时间权重 (默认0.5)
            """
            # 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

            # 创建邻接矩阵 (景点-区域双向图)
            n = self.num_spots + self.num_regions  # 总节点数

            # 初始化矩阵
            combined_full = np.full((n, n), np.inf)
            cost_full = np.full((n, n), np.inf)
            time_full = np.full((n, n), np.inf)

            # 对角线为0
            np.fill_diagonal(combined_full, 0)
            np.fill_diagonal(cost_full, 0)
            np.fill_diagonal(time_full, 0)

            # 计算归一化常数
            finite_costs = self.cost_mat_raw[np.isfinite(self.cost_mat_raw)]
            finite_times = self.time_mat_raw[np.isfinite(self.time_mat_raw)]
            cost_max = np.max(finite_costs) if len(finite_costs) > 0 else 100
            time_max = np.max(finite_times) if len(finite_times) > 0 else 100

            # 填入已知的景点到区域的连接
            for r in range(self.num_regions):
                for s in range(self.num_spots):
                    if not np.isinf(self.cost_mat_raw[r, s]):
                        cost = self.cost_mat_raw[r, s]
                        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
```

```
                            route = {
                                'type': '1day',
                                'path': [s1, r, s2],
                                'spots': [s1, s2],
                                'regions': [r],
                                'lunch_regions': [r],
                                'night_regions': [],
                                'cost': self.cost_mat[s1, r] + self.cost_mat[
                                    ↪ s2, r],
                                'time': self.time_mat[s1, r] + self.time_mat[
                                    ↪ s2, r],
                                'preference': sum([self.region_prefer[r]]) + 2
                                    ↪  # 景点基础偏好
                            }
                            self.routes_1day.append(route)

        # print(f"一日游路线数量: {len(self.routes_1day)}") # 避免过多打印

    def generate_2day_routes(self, max_routes=10000):
        """生成二日游路线: S -> R -> S -> R -> S -> R -> S"""
        # print("生成二日游路线...") # 避免过多打印
        self.routes_2day = []
        count = 0

        for s1 in range(self.num_spots):
            for r1 in range(self.num_regions):
                for s2 in range(self.num_spots):
                    for r2 in range(self.num_regions):
                        for s3 in range(self.num_spots):
                            for r3 in range(self.num_regions):
                                for s4 in range(self.num_spots):
                                    if len(set([s1, s2, s3, s4])) == 4:  #
                                        ↪ 所有景点不重复
                                        if count >= max_routes:
                                            break

                                        route = {
                                            'type': '2day',
                                            'path': [s1, r1, s2, r2, s3,
                                                ↪ r3, s4],
                                            'spots': [s1, s2, s3, s4],
                                            'regions': [r1, r2, r3],
                                            'lunch_regions': [r1, r3],  #
                                                ↪ 第1天午餐r1, 第2天午餐r3
                                            'night_regions': [r2],  # 第1
                                                ↪ 天住宿r2
                                            '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]]
```

```
                    time += self.time_mat[spots[3], regions[2]]
                    time += self.time_mat[spots[3], regions[3]]

                    # Day 3: S4->R4->S5->R5->S6 (End trip)
                    # Travel from R4 to S5 (already at R4)
                    cost += self.cost_mat[spots[4], regions[4]]
                    cost += self.cost_mat[spots[5], regions[4]]
                    time += self.time_mat[spots[4], regions[4]]
                    time += self.time_mat[spots[5], regions[4]]

                    preference = sum([self.region_prefer[r] for r in regions])
                        ↪ + 6 # 6 unique spots contribute to base preference

                    # 归一化计算 (与MILP保持一致)
                    max_preference_3day = 9 * 5 + 6  # 理论最大偏好度
                    # 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)
```

```
292                     start, end = sorted(random.sample(range(size), 2))
293
294                     child = [-1] * size
295                     # 复制中间段
296                     child[start:end] = parent1[start:end]
297
298                     # 从parent2中按顺序填充剩余位置
299                     pointer = end
300                     for item in parent2[end:] + parent2[:end]:
301                         if item not in child:
302                             while child[pointer] != -1: # Find next empty spot
303                                 pointer = (pointer + 1) % size
304                             child[pointer] = item
305                             pointer = (pointer + 1) % size # Move pointer for
                                ↪ next item
306
307                     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)
```

```
                    individual[pos] = random.randint(0, self.num_regions-1)
                return individual,

            toolbox.register("mutate", mutate_individual)
            toolbox.register("select", tools.selTournament, tournsize=3)

            # 运行遗传算法
            population = toolbox.population(n=population_size)

            # 初始评估
            fitnesses = list(map(toolbox.evaluate, population))
            for ind, fit in zip(population, fitnesses):
                ind.fitness.values = fit

            for gen in range(generations):
                # 选择
                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}")

```

```
382             # 提取最优个体作为候选路线，保证多样性
383             population.sort(key=lambda x: x.fitness.values[0], reverse=True)
384 
385             self.routes_3day_candidates = []
386             used_route_keys = set()
387 
388             for ind in population:
389                 spots = ind[:6]
390                 regions = ind[6:]
391 
392                 # 验证景点不重复
393                 if len(set(spots)) != 6:
394                     continue  # 跳过有重复景点的个体
395 
396                 # 创建路线唯一标识
397                 route_key = tuple(spots + regions)
398 
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
```

```
                        try:
                            cost = (self.cost_mat[spots[0], region_combo[0]] +
                                ↪  self.cost_mat[spots[1], region_combo[0]] +
                                self.cost_mat[spots[1], region_combo[1]] +
                                    ↪ self.cost_mat[spots[2], region_combo
                                    ↪ [1]] +
                                self.cost_mat[spots[2], region_combo[2]] +
                                    ↪ self.cost_mat[spots[3], region_combo
                                    ↪ [2]] +
                                self.cost_mat[spots[3], region_combo[3]] +
                                    ↪ self.cost_mat[spots[4], region_combo
                                    ↪ [3]] +
                                self.cost_mat[spots[4], region_combo[4]] +
                                    ↪ self.cost_mat[spots[5], region_combo
                                    ↪ [4]])

                            time = (self.time_mat[spots[0], region_combo[0]] +
                                ↪  self.time_mat[spots[1], region_combo[0]] +
                                self.time_mat[spots[1], region_combo[1]] +
                                    ↪ self.time_mat[spots[2], region_combo
                                    ↪ [1]] +
                                self.time_mat[spots[2], region_combo[2]] +
                                    ↪ self.time_mat[spots[3], region_combo
                                    ↪ [2]] +
                                self.time_mat[spots[3], region_combo[3]] +
                                    ↪ self.time_mat[spots[4], region_combo
                                    ↪ [3]] +
                                self.time_mat[spots[4], region_combo[4]] +
                                    ↪ self.time_mat[spots[5], region_combo
                                    ↪ [4]])

                            preference = sum([self.region_prefer[r] for r in
                                ↪ region_combo]) + 6

                            route['cost'] = cost
                            route['time'] = time
                            route['preference'] = preference

                            self.routes_3day_candidates.append(route)
                        except:
                            continue

            print(f"混合方法总共生成: {len(self.routes_3day_candidates)}条路线
                ↪ ")
            print(f"其中遗传算法: {ga_routes_count}条, 启发式: {len(self.
                ↪ routes_3day_candidates) - ga_routes_count}条")

        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
541             max_cost = max([route['cost'] for route in routes]) if routes else
                    ↪ 1
542             max_time = max([route['time'] for route in routes]) if routes else
                    ↪ 1
543
544             # 根据路线类型确定偏好度最大值
545             def get_max_preference(route_type):
546                 if route_type == '1day':
547                     return max_preference_1day
548                 elif route_type == '2day':
549                     return max_preference_2day
550                 else:  # 3day
551                     return max_preference_3day
552
553             # 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)
554             c = []
555             for route in routes:
556                 # 归一化各项指标到[0,1]范围
557                 pref_normalized = route['preference'] / get_max_preference(
                        ↪ route['type'])
558                 cost_normalized = route['cost'] / max_cost
559                 time_normalized = route['time'] / max_time
560
561                 # 加权计算满意度
562                 satisfaction = w_pref * pref_normalized - w_cost *
                        ↪ cost_normalized - w_time * time_normalized
563                 c.append(-satisfaction)  # linprog求最小值, 所以取负
564
565             # 约束矩阵
566             A_ub = []
567             b_ub = []
```

```
568
569             # 景点容量约束 - 按时段分别约束
570             max_days = 3  # 最多3日游
571             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)
572
573             for s in range(self.num_spots):
574                 constraint = []
575                 for route in routes:
576                     # 计算该路线中景点s被访问的次数
577                     visit_count = route['spots'].count(s)
578                     constraint.append(visit_count)
579                 A_ub.append(constraint)
580                 # 景点在多个时段可复用, 总容量 = 单时段容量 × 时段数
581                 total_capacity = self.spot_cap[s] * max_time_slots
582                 b_ub.append(total_capacity)
583
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:
```

```
                # 求解
                result = linprog(c, A_ub=A_ub, b_ub=b_ub, A_eq=A_eq, b_eq=b_eq,
                    bounds=bounds, method='highs')

                if result.success:
                    solution = []
                    total_s = 0
                    total_t = 0
                    for i, x in enumerate(result.x):
                        if x > 0.1:  # 过滤掉很小的值
                            num_tourists = int(round(x))
                            route_info = routes[i].copy()
                            route_info['tourists'] = num_tourists
                            # 重新计算归一化满意度用于显示
                            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
```

```
653             """
654             # print("开始旅游路线优化...") # 避免过多打印
655             # print(f"使用权重配置: 费用权重={cost_weight}, 时间权重={
                    ↪ time_weight}")
656 
657             # 1. 数据预处理
658             self.preprocess_matrices(cost_weight, time_weight)
659 
660             # 2. 生成路线
661             self.generate_1day_routes()
662             self.generate_2day_routes(max_routes=5000)  # 限制二日游路线数量
663             self.generate_3day_routes_hybrid(target_routes=5000) # 使用混合方
                    ↪ 法生成三日游路线
664 
665             # 3. 分别求解三种旅游方案
666             results = {}
667 
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")
```

```
                    print(f"   景点路径: {spots_str}")
                    print(f"   区域路径: {regions_str}")
                    print(f"   成本: {route['cost']:.1f}元, 时间: {route['time']:.1f}分钟, 偏好: {route['preference']}")
                    print()

    # 扩容成本函数
    def expansion_cost(delta_K: float) -> float:
        """
        扩容成本函数
        Args:
            delta_K: 新增接待能力（万人次）
        Returns:
            扩容成本（亿元）
        """
        c1 = 0.0007  # 亿元
        gamma = 1.09
        return c1 * (delta_K ** gamma)

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

C. 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

# --- 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():
```

```
20          print("=" * 60)
21          print("重庆旅游路线优化系统")
22          print("=" * 60)
23
24          # Create optimizer instance
25          optimizer = TourismOptimizer()
26
27          # Run optimization
28          try:
29              # Test different weight configurations
30              weight_configs = [
31                  (0.5, 0.5, "均衡型"),
32                  # (0.7, 0.3, "费用优先型"), # Can comment out other
                        ↪ configurations for faster expansion analysis
33                  # (0.3, 0.7, "时间优先型")
34              ]
35
36              all_results = {}
37
38              for cost_w, time_w, config_name in weight_configs:
39                  print(f"\n{'='*80}")
40                  print(f"运行{config_name}配置 (费用权重={cost_w}, 时间权重={
                        ↪ time_w})")
41                  print(f"{'='*80}")
42
43                  # Run optimization and get results
44                  results = optimizer.run_optimization(cost_weight=cost_w,
                        ↪ time_weight=time_w)
45                  all_results[config_name] = results
46
47                  # Print results
48                  optimizer.print_results(results)
49
50                  # Save results to file
51                  save_results_to_file(results, suffix=f"_{config_name}")
52
53              # Compare results of different configurations
54              compare_configurations(all_results)
55
56              # ======= Problem 3: New Hotel Expansion Analysis =======
57              print(f"\n{'='*80}")
58              print("问题3: 新建旅馆扩容分析")
59              print(f"{'='*80}")
60              analyze_hotel_expansion(optimizer)
61
62          except Exception as e:
63              print(f"优化过程中出现错误: {e}")
64              import traceback
```

```
65              traceback.print_exc()
66
67      def compare_configurations(all_results):
68          """比较不同权重配置的结果"""
69          print(f"\n{'='*80}")
70          print("不同权重配置结果比较")
71          print(f"{'='*80}")
72
73          for day_type in ['1day', '2day', '3day']:
74              print(f"\n{day_type.upper()} 方案比较:")
75              print("-" * 60)
76              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}")
77              print("-" * 60)
78
79              for config_name, results in all_results.items():
80                  solution = results.get(day_type, [])
81                  if solution:
82                      total_tourists = sum([route['tourists'] for route in
                            ↪ solution])
83                      total_cost = sum([route['tourists'] * route['cost'] for
                            ↪ route in solution])
84                      total_time = sum([route['tourists'] * route['time'] for
                            ↪ route in solution])
85                      total_satisfaction = sum([route['tourists'] * route['
                            ↪ satisfaction'] for route in solution])
86                      avg_satisfaction = total_satisfaction / total_tourists if
                            ↪ total_tourists > 0 else 0
87
88                      print(f"{config_name:<12} {len(solution):<8} {
                            ↪ total_tourists:<8} {total_cost:<12.0f} "
89                            f"{total_time:<12.0f} {avg_satisfaction:<12.2f}")
90                  else:
91                      print(f"{config_name:<12} {'无解':<8} {'-':<8} {'-':<12}
                            ↪ {'-':<12} {'-':<12}")
92
93
94      def save_results_to_file(results, suffix=""):
95          """将结果保存到文件"""
96          filename = f'optimization_results{suffix}.txt'
97          with open(filename, 'w', encoding='utf-8') as f:
98              f.write("重庆旅游路线优化结果\n")
99              f.write("=" * 50 + "\n\n")
100
101             for day_type_key, solution_or_metrics in results.items():
102                 if not isinstance(solution_or_metrics, list): # Filter out
                        ↪ metrics dictionary
103                     continue
```

```
104                 solution = solution_or_metrics
105
106                 day_type = day_type_key.replace('_metrics', '') # Get actual
                        ↪ day_type
107
108                 f.write(f"{day_type.upper()} 旅游方案结果\n")
109                 f.write("=" * 30 + "\n")
110
111                 if not solution:
112                     f.write("无可行解\n\n")
113                     continue
114
115                 total_tourists = sum([route['tourists'] for route in solution
                        ↪ ])
116                 total_cost = sum([route['tourists'] * route['cost'] for route
                        ↪ in solution])
117                 total_time = sum([route['tourists'] * route['time'] for route
                        ↪ in solution])
118                 total_satisfaction = sum([route['tourists'] * route['
                        ↪ satisfaction'] for route in solution])
119
120                 f.write(f"路线类型数量: {len(solution)}\n")
121                 f.write(f"总游客数: {total_tourists}\n")
122                 f.write(f"总交通成本: {total_cost:.0f} 元\n")
123                 f.write(f"总交通时间: {total_time:.0f} 分钟\n")
124                 f.write(f"总满意度: {total_satisfaction:.2f}\n")
125                 f.write(f"平均满意度: {total_satisfaction/total_tourists:.2f}\
                        ↪ n\n")
126
127                 f.write("具体路线分配:\n")
128                 for i, route in enumerate(solution[:10]):
129                     spots_str = "->".join([f"S{s+1}" for s in route['spots']])
130                     regions_str = "->".join([f"R{r+1}" for r in route['regions
                            ↪ ']])
131                     f.write(f"路线{i+1}: {route['tourists']}人\n")
132                     f.write(f"  景点路径: {spots_str}\n")
133                     f.write(f"  区域路径: {regions_str}\n")
134                     f.write(f"  成本: {route['cost']:.1f}元, 时间: {route['
                            ↪ time']:.1f}分钟, 偏好: {route['preference']}\n\n")
135                 f.write("\n")
136
137         print(f"\n结果已保存到 {filename} 文件")
138
139
140     def analyze_hotel_expansion(optimizer: TourismOptimizer):
141         """
142         Analyzes the hotel expansion problem to find the optimal expansion
                ↪ area and scale.
```

```
143         """
144         print("\n开始分析旅馆扩容方案...")
145
146         # Define economic value parameters
147         # vs: Economic value per unit satisfaction (billion CNY/satisfaction)
148         # vp: Average profit per tourist (billion CNY/person)
149         v_s_per_million_satisfaction = 0.1 # Assume 1 million satisfaction
                ↪ brings 0.1 billion CNY
150         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)
151
152         # Run baseline optimization to get metrics at K0
153         print("获取基准指标 (扩容前)...")
154         base_results = optimizer.run_optimization(cost_weight=0.5, time_weight
                ↪ =0.5, tourists_1day=30000, tourists_2day=30000, tourists_3day
                ↪ =20000)
155         Z_K0_total, N_K0_total = optimizer.get_aggregated_metrics(base_results
                ↪ )
156
157         # Convert number of tourists to millions to match delta_K unit
158         N_K0_total_wan = N_K0_total / 10000.0
159
160         print(f"基准总满意度 Z(K0): {Z_K0_total:.2f}")
161         print(f"基准总游客数 N(K0): {N_K0_total_wan:.2f} 万人次")
162
163         best_net_benefit = -np.inf
164         best_region_idx = -1
165         best_delta_K_wan = 0
166
167         # Define expansion capacity range (in millions of persons)
168         # Consider from 0 to a reasonable maximum value, e.g., twice the
                ↪ original maximum capacity
169         max_delta_K_wan = max(optimizer.night_cap_original) / 10000.0 * 2 #
                ↪ Twice the original max capacity, in millions of persons
170         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
171
172         # Store all test results
173         expansion_results = []
174
175         print("\n迭代各区域及扩容规模进行模拟...")
176         for r_idx in range(optimizer.num_regions):
177             original_night_cap_r = optimizer.night_cap_original[r_idx] # Get
                    ↪ the original capacity of this region
178
179             for delta_K_wan in delta_K_wan_range:
180                 current_night_cap_r = original_night_cap_r + (delta_K_wan *
```

```
                    ↪ 10000) # Convert millions of persons back to persons
181
182                 # Temporarily modify the accommodation capacity of this region
183                 optimizer.night_cap[r_idx] = current_night_cap_r
184
185                 # Rerun optimization
186                 # Note: run_optimization will regenerate routes, ensuring each
                        ↪ run is based on the modified capacity
187                 current_results = optimizer.run_optimization(
188                     cost_weight=0.5, time_weight=0.5, # Use balanced weights
                            ↪ for expansion analysis
189                     tourists_1day=30000, tourists_2day=30000, tourists_3day
                            ↪ =20000
190                 )
191                 Z_K_total, N_K_total = optimizer.get_aggregated_metrics(
                        ↪ current_results)
192                 N_K_total_wan = N_K_total / 10000.0
193
194                 # Calculate benefit B(delta_K)
195                 B_delta_K = (v_s_per_million_satisfaction * (Z_K_total -
                        ↪ Z_K0_total) +
196                              v_p_per_million_tourists * (N_K_total_wan -
                                 ↪ N_K0_total_wan))
197
198                 # Calculate cost C(delta_K)
199                 C_delta_K = expansion_cost(delta_K_wan)
200
201                 # Calculate net benefit F(delta_K)
202                 F_delta_K = B_delta_K - C_delta_K
203
204                 expansion_results.append({
205                     'region_idx': r_idx,
206                     'region_name': f'区域{r_idx + 1}',
207                     'delta_K_wan': delta_K_wan,
208                     'original_night_cap_wan': original_night_cap_r / 10000.0,
209                     'new_night_cap_wan': current_night_cap_r / 10000.0,
210                     'Z_K_total': Z_K_total,
211                     'N_K_total_wan': N_K_total_wan,
212                     'Benefit_B': B_delta_K,
213                     'Cost_C': C_delta_K,
214                     'Net_Benefit_F': F_delta_K
215                 })
216
217                 # Update optimal solution
218                 if F_delta_K > best_net_benefit:
219                     best_net_benefit = F_delta_K
220                     best_region_idx = r_idx
221                     best_delta_K_wan = delta_K_wan
```

```
222
223             # Restore the original capacity of this region for the next region
                    ↪ 's test
224             optimizer.night_cap[r_idx] = optimizer.night_cap_original[r_idx]
225
226         print("\n扩容分析结果:")
227         print("-" * 60)
228         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}")
229         print("-" * 60)
230
231         # Print the top 20 net benefit solutions
232         sorted_expansion_results = sorted(expansion_results, key=lambda x: x['
                ↪ Net_Benefit_F'], reverse=True)
233         for i, res in enumerate(sorted_expansion_results[:20]):
234             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}")
235
236         print(f"\n最终建议: ")
237         if best_region_idx != -1:
238             print(f"建议扩容区域: 区域{best_region_idx + 1}")
239             print(f"建议扩容规模: {best_delta_K_wan:.2f} 万人次")
240             print(f"最大净收益: {best_net_benefit:.4f} 亿元")
241             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")
242             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")
243
244             # Plot net benefit
245             plt.figure(figsize=(12, 6))
246
247             region_colors = plt.cm.tab10 # Use matplotlib's color map
248
249             # Filter data for each region
250             regions_data = {r_idx: [] for r_idx in range(optimizer.num_regions
                    ↪ )}
251             for res in expansion_results:
252                 regions_data[res['region_idx']].append(res)
253
254             for r_idx, data in regions_data.items():
255                 data.sort(key=lambda x: x['delta_K_wan'])
256                 delta_K_wan_values = [d['delta_K_wan'] for d in data]
257                 net_benefit_values = [d['Net_Benefit_F'] for d in data]
258
259                 plt.plot(delta_K_wan_values, net_benefit_values, marker='o',
                        ↪ linestyle='-',
```

```
260                             label=f'区域 {r_idx + 1}', color=region_colors(r_idx)
                                    ↪ )
261
262             plt.title('不同区域及扩容规模下的净收益')
263             plt.xlabel('新增接待能力 (万人次)')
264             plt.ylabel('净收益 (亿元)')
265             plt.grid(True)
266             plt.legend(title='区域')
267             plt.axhline(0, color='grey', linestyle='--', linewidth=0.8) # Zero
                    ↪  benefit line
268             plt.scatter(best_delta_K_wan, best_net_benefit, color='red',
                    ↪ marker='X', s=200,
269                         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')
270             plt.legend()
271             plt.tight_layout()
272             plt.savefig('net_benefit_vs_expansion.png')
273             plt.show()
274
275         else:
276             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。")
277
278
279     if __name__ == "__main__":
280         main()
```