

重慶大學

## 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  $\text{Pref}(p)$ 、 $\text{Cost}(p)$ 、 $\text{Time}(p)$  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

随着中国乡村振兴战略的深入推进，农业产业结构优化和可持续发展成为乡村经济振兴的重要课题。华北山区某乡村耕地资源有限且分散，现有 1201 亩露天耕地和 12 亩大棚。受气候限制，大多数耕地每年仅能种植一季作物，且不同地块适宜种植的作物类型不同。此外，农作物种植需遵循轮作规则和豆类种植要求，同时需兼顾田间管理的便利性。

而 2023 年的种植数据为优化未来种植策略提供了依据。未来，农作物的预期销量、亩产量、成本和价格可能受市场、气候等因素影响而波动。例如，小麦、玉米需求可能增长，蔬菜价格可能上涨，而食用菌价格可能下降。我们需要综合考虑耕地资源约束、农作物生长规律、市场需求变化和经济效益最大化等因素，建立模型解决以下问题：

### 1.2 问题重述

为了优化农作物种植结构，提高土地利用效率并实现经济效益最大化，我们需要综合考虑耕地资源、作物生长规律和市场因素，并遵循以下基本原则：

**问题一：**在假设未来农作物预期销售量、种植成本、亩产量和销售价格保持稳定的前提下，我们需要制定 2024-2030 年的最优种植方案。当产量超过预期销售量时，该问题需考虑超产部分完全滞销造成浪费，超产部分可按 2023 年销售价格的 50% 降价出售两种不同的销售情景。

**问题二：**考虑到实际农业生产中存在诸多不确定性因素，我们需要建立鲁棒性优化模型，在考虑考虑市场需求、产量波动、成本上涨及价格变动等不确定性因素的情况下，重新制定出 2024-2030 年的最优种植方案。

**问题三：**在问题二的基础上，需要进一步考虑农作物之间的可替代性和互补性，以及销售量、价格和成本之间的相关性。我们需要建立更符合现实情况的综合评价模型，通过模拟分析得到最优种植策略，并与问题二的结果进行对比分析，为实际决策提供更全面的参考依据。

## 2 问题分析

### 2.1 问题一

针对问题一，我们需要根据乡村的耕地类型及作物特性，设计一个优化的农作物种植方案。首先，考虑到不同地块的种植要求，例如平旱地、梯田和山坡地只能种植一季粮食类作物，而水浇地可以种水稻或两季蔬菜。每种作物的种植面积必须满足一定的最小面积要求，并且同一地块不能连续重茬种植相同作物，这些都需要在模型中加入约束条件。

此外，根据题目要求和查阅的论文可知，乡村的露天地块每年必须种植至少 401（ $1201/3$ ，此处向上取整）亩豆类作物，大棚需要种植至少 4 亩豆类作物，这些限制也应纳入模型中进行优化。作物的销售量与价格存在不确定性，若超出销售量则会面临滞销或降价出售的风险。因此，我们在模型中需要考虑两种情景：一是滞销，二是超出部分按降价销售。

最终，优化目标是最大化乡村的农作物总收益，综合考虑作物的预期销售量、单价、产量和种植成本。在此过程中，还要考虑耕地面积的限制，确保每块地的种植面积不超过其可用面积。通过这些约束和目标，我们将能够得出一个最优的种植方案，确保乡村的农业生产既能满足市场需求，又能提高经济效。

### 2.2 问题二

在问题二中，根据经验，小麦和玉米的未来销售量预计将有 5% 到 10% 的年增长率，而其他作物的销售量相对于 2023 年则会 5% 的波动。为了反映这些不确定性，我们将对每种作物的预期销售量进行随机模拟，考虑每个作物的年增长率，并将这种变化应用于模型中。这样能够准确捕捉到销售量的波动性，并对市场需求的不确定性做出调整。

农作物的亩产量也往往受到气候等因素的影响，预计每年会有 10% 的变化。这个不确定性在模型中也需要考虑，确保我们在设定作物种植方案时能够考虑到气候变化可能带来的影响。作物的种植成本随着时间推移而逐年增长，预计年增长幅度约为 5%。因此，优化模型中将包括种植成本的动态增长机制，并在决策过程中考虑这一成本的增加。对于不同类型的作物，特别是粮食类作物、蔬菜类作物和食用菌类作物，销售价格也存在不同程度的波动。粮食类作物的销售价格相对稳定，而蔬菜类作物销售价格每年有大约 5% 的增长，食用菌价格则会逐年下降，羊肚菌的销售价格每年下降幅度达到 5

基于这些不确定性和约束条件，我们将利用蒙特卡洛-SAA 方法对不同场景下

的种植方案进行求解，最终得出 2024 至 2030 年期间的最优种植方案。这一方案不仅能够确保乡村农业生产满足市场需求，还能最大化经济效益，为乡村发展提供坚实的支持。

### 2.3 问题三

第三问的核心在于考虑农作物间的替代性和互补性，并将其与价格、销售量和种植成本的相关性结合起来。这不仅要求我们在制定种植策略时考虑单一作物的利润最大化，还需要综合考虑作物之间的市场联动效应。具体而言，相同类型的作物可能在市场上存在替代关系，即一类作物的销售量增加可能会导致另一类作物的需求下降。与此同时，作物间也可能存在互补性，例如轮作和间作会提升土壤质量或作物的单产，从而促进整个种植体系的收益增长。

在这类复杂的环境下，传统的单一作物优化模型已经不能满足需求，因此需要引入基于价格和需求反向调节的模型。通过引入价格弹性系数和需求弹性系数，可以模拟作物市场中的供需变化，调整作物的价格和需求，从而更真实地反映实际市场的波动。同时，作物间的互补效应，如轮作的产量提升，也需要在模型中体现，以便根据土壤改良、作物间的协同效应优化种植决策。

此外，尽管引入了更为复杂的替代性和互补性关系，但问题的求解仍需保持在线性化范围内。通过使用适当的线性化技术（如大  $M$  法），我们可以将这些非线性关系转化为线性形式，从而使用混合整数线性规划（MILP）方法来求解。最终目标是通过模拟不同的市场情景和作物种植策略，找到 2024 至 2030 年间最优的种植方案，不仅要在收益上达到最大化，还要确保作物种植的可持续性和经济性。

### 3 模型假设

本文针对农作物种植优化方案以及作物间的替代性、互补性以及价格、销售量、成本等因素之间的相关性问题的，建立的数学模型基于以下核心假设：

**假设 1：** 各类作物的需求量受市场供需的影响，且价格弹性较大。具体而言，蔬菜类作物的价格会随着供给量变化而调整，而粮食类作物的价格则保持相对刚性。

**假设 2：** 作物间的替代性仅限于同类作物。例如，豆类作物与豆类作物之间具有替代性，但与其他作物（如粮食作物、蔬菜等）不具备直接的替代性。

**假设 3：** 作物间的互补性仅限于有明确互补效应的作物对。例如，豆类作物与某些粮食作物或蔬菜作物的种植具有互补关系，互补关系通过提高作物产量和单价来体现。

**假设 4：** 作物的市场价格和销量之间存在价格弹性，蔬菜类作物的需求会受到价格波动的影响，而粮食类作物的需求对价格的波动较为迟钝，因此假设粮食类作物价格保持不变。

**假设 5：** 作物的产量在每年内保持线性增长，且轮作和互补关系所带来的增产效果是固定的。具体而言，轮作带来的增产系数为 10%，并且根据种植作物的不同，作物的价格和产量也会进行调整。

**假设 6：** 有效需求与供给量之间存在双向联动，即供给量的变化影响价格，价格变化又会反作用于需求。每个作物的产量不应超过其有效需求，且需求量随着价格调整而变化。

## 4 符号说明

符号	说明
$t$	年份索引, $t = 2024, \dots, 2030$
$m$	作物类别索引, $m = 1, 2, \dots, 41$
$i$	地块编号索引, $i = 1, 2, \dots, 34$
$s$	季节索引, $s = 1, 2$
$x_{i,m,t,s}$	第 $i$ 块地第 $t$ 年第 $s$ 季节种植第 $m$ 类作物的面积
$z_{i,m,t}$	二元变量, 第 $i$ 块地第 $t$ 年种植第 $m$ 类作物时为 1, 否则为 0
$P_{m,s}$	第 $m$ 类作物在季节 $s$ 的单价 (元/亩)
$Y_{m,s}$	第 $m$ 类作物在季节 $s$ 的亩产量 (单位产量)
$C_{m,s}$	第 $m$ 类作物在季节 $s$ 的种植成本 (元/亩)
$V_b$	豆类作物集合
$A_{fall}$	第 $i$ 块地的最大可用种植面积 (亩)
$N_{\max}$	每块地种植作物种类的最大数量限制
$\delta$	作物最小种植面积限制 (亩)
$\omega$	场景索引, $\omega = 1, 2, \dots, S$
$S$	场景总数
$\delta_{m,t}^D$	第 $m$ 类作物第 $t$ 年需求的随机增量
$\delta_{m,t}^Y$	第 $m$ 类作物第 $t$ 年亩产量的随机增量
$\delta_{m,t}^C$	第 $m$ 类作物第 $t$ 年种植成本的随机增量
$\delta_{m,t}^P$	第 $m$ 类作物第 $t$ 年单价的随机增量
$D_{m,t}^{(\omega)}$	第 $\omega$ 场景下第 $m$ 类作物第 $t$ 年的需求量 (亩)
$Y_{m,t}^{(\omega)}$	第 $\omega$ 场景下第 $m$ 类作物第 $t$ 年的亩产量
$C_{m,t}^{(\omega)}$	第 $\omega$ 场景下第 $m$ 类作物第 $t$ 年的种植成本
$P_{m,t}^{(\omega)}$	第 $\omega$ 场景下第 $m$ 类作物第 $t$ 年的单价
$\tilde{P}_{m,t}$	考虑替代性和互补性调整后的作物单价
$Y_{m,t}^{\text{eff}}$	考虑轮作增益后的有效亩产量
$\tilde{D}_{m,t}$	考虑价格弹性调节后的有效需求量
$\eta_m$	第 $m$ 类作物的价格弹性系数
$\alpha_{k \rightarrow m}$	作物 $k$ 对作物 $m$ 的替代性系数
$\beta_{k \rightarrow m}$	作物 $k$ 对作物 $m$ 的互补性系数
$Q_{k,t}$	作物 $k$ 在第 $t$ 年的供给量
$Q_k^*$	作物 $k$ 的基准供给量
$\theta_m^{\text{rotation}}$	作物 $m$ 的轮作增益系数
$z_{i,m,t-1}$	第 $i$ 块地第 $t-1$ 年是否种植作物 $m$ 的二元变量
$Z$	目标函数, 总收益 (最大化农业经济效益)

## 5 模型建立与求解

### 5.1 问题一：农作物种植优化方案设计

#### 5.1.1 模型建立与求解

对于此问题，我们首先需要通过设定合理的决策变量，来确定每个地块在每个年份和季节种植的作物面积。每个地块的种植方案需考虑土壤轮作、作物间的轮换以及豆类作物的种植要求。我们还需要加入作物销售限制、耕地面积限制等约束条件，以确保种植计划合理、可行。根据这些条件，我们建立了以下的数学模型：

**目标函数：**为了最大化农业经济效益，我们的目标函数是将所有作物的总收益最大化。作物的收益由其亩产量、市场价格和种植面积决定，同时减去种植成本。公式如下：

$$\text{Maximize } Z = \sum_{t=2024}^{2030} \sum_{m=1}^{41} \sum_{i=1}^{34} \sum_{s=1}^2 (P_{m,s} \cdot Y_{m,s} \cdot x_{i,m,t,s} - C_{m,s} \cdot x_{i,m,t,s}) \quad (5.1)$$

其中， $P_{m,s}$  为第  $m$  类作物在季节  $s$  的单价； $Y_{m,s}$  为第  $m$  类作物在季节  $s$  的亩产量； $C_{m,s}$  为第  $m$  类作物在季节  $s$  的种植成本； $x_{i,m,t,s}$  为第  $i$  块地在第  $t$  年第  $s$  季节种植第  $m$  类作物的面积。

**豆类作物种植面积要求：**根据题目要求，乡村露天地块每年必须种植至少 401 亩豆类作物，大棚需要种植至少 4 亩豆类作物。我们定义豆类作物集合  $V_b$ ，并添加以下约束：

$$\sum_{m \in V_b} \sum_{i=1}^{34} x_{i,m,t} = \frac{1}{3} \cdot 1201 \quad \forall t \quad (5.2)$$

$$\sum_{m \in V_b} \sum_{j=1}^{16} x_{j,m,t} + \sum_{m \in V_b} \sum_{k=1}^4 x_{k,m,t} = 4 \quad \forall t \quad (5.3)$$

**作物轮作约束：**同一块地不能连续重茬种植相同作物。为此，引入二元变量  $z_{i,m,t}$ ，当第  $i$  块地在第  $t$  年种植第  $m$  类作物时， $z_{i,m,t} = 1$ ，否则  $z_{i,m,t} = 0$ 。因此，约束条件为：

$$z_{i,m,t} + z_{i,m,t+1} \leq 1 \quad \forall i, \forall m, \forall t \quad (5.4)$$

**耕地面积限制：**每年各地块的种植总面积不能超过其可用面积。设  $A_{f_{all}}$  为第



$i$  块地的最大种植面积，约束条件为：

$$\sum_{m=1}^{41} x_{i,m,t} \leq A_{fall} \quad \forall i, \forall t \quad (5.5)$$

**作业便利性约束：**每块地的作物种植数有最大数量限制，即每个地块种植的作物种类数量不能超过  $N_{\max}$ 。公式为：

$$\sum_{m=1}^{41} z_{i,m,t} \leq N_{\max} \quad \forall i, \forall t \quad (5.6)$$

### 求解过程

#### 输入数据：

- 作物的预期销售量  $D_{m,s}$ 。
- 每种作物的单价  $P_{m,s}$ ，亩产量  $Y_{m,s}$ ，种植成本  $C_{m,s}$ 。
- 各地块的可用面积  $A_{fall}$ ，以及每块地的作物最小种植面积限制  $\delta$ 。

**模型求解方法：**采用混合整数线性规划（MILP）方法求解此优化问题。具体步骤如下：

- 使用求解器（如 CPLEX 或 Gurobi）解决模型中的线性约束和目标函数。
- 对于每个年度  $t$ ，按照最大化收益的目标，计算每个地块的作物种植面积分配。
- 考虑作物的销售限制、作物轮作约束以及豆类作物种植面积要求，确保在求解过程中遵循所有约束。

#### 结果输出：

## 5.2 问题二：农作物种植优化方案设计

### 5.2.1 模型建立与求解

对于此问题，我们首先需要通过设定合理的决策变量，来确定每个地块在每个年份和季节种植的作物面积。每个地块的种植方案需考虑作物的销售限制、亩产量、种植成本、销售价格的变化，并考虑气候等不确定因素的影响。我们还需要加入作物销售限制、耕地面积限制等约束条件，以确保种植计划合理、可行。根据这些条件，我们建立了以下的数学模型：

**目标函数：**为了最大化农业经济效益，我们的目标函数是将所有作物的总收益最大化。作物的收益由其亩产量、市场价格和种植面积决定，同时减去种植成本。公式如下：

$$\text{Maximize } Z = \sum_{t=2024}^{2030} \sum_{m=1}^{41} \sum_{i=1}^{34} \sum_{s=1}^2 (P_{m,s} \cdot Y_{m,s} \cdot x_{i,m,t,s} - C_{m,s} \cdot x_{i,m,t,s}) \quad (5.7)$$

其中,  $P_{m,s}$  为第  $m$  类作物在季节  $s$  的单价;  $Y_{m,s}$  为第  $m$  类作物在季节  $s$  的亩产量;  $C_{m,s}$  为第  $m$  类作物在季节  $s$  的种植成本;  $x_{i,m,t,s}$  为第  $i$  块地在第  $t$  年第  $s$  季节种植第  $m$  类作物的面积。

**随机因素:** 为考虑市场、气候等不确定性因素的影响, 我们引入随机符号以模拟作物的销售量、亩产量、种植成本和价格的波动。随机增量符号定义如下:

- $\delta_{m,t}^D$ : 需求 (销量) 变化, 遵循  $\mathcal{U}(0.05, 0.10)$  对于小麦和玉米, 其他作物为  $\mathcal{U}(-0.05, 0.05)$ ;
- $\delta_{m,t}^Y$ : 亩产量变化, 遵循  $\mathcal{U}(-0.10, 0.10)$ ;
- $\delta_{m,t}^C$ : 成本变化, 遵循  $\mathcal{U}(-0.02, 0.02)$ ;
- $\delta_{m,t}^P$ : 单价变化, 粮食类为  $\mathcal{U}(-0.02, 0.02)$ , 蔬菜类作物有 5

**需求与收益计算:** 每个场景下的需求、单产、成本和价格分别由随机增量计算得出, 具体为:

- $D_{m,t}^{(\omega)}$ : 第  $\omega$  场景下的需求;
- $Y_{m,t}^{(\omega)}$ : 第  $\omega$  场景下的亩产量;
- $C_{m,t}^{(\omega)}$ : 第  $\omega$  场景下的成本;
- $P_{m,t}^{(\omega)}$ : 第  $\omega$  场景下的单价。

**约束条件:** 在优化过程中, 我们需要满足以下约束条件:

1. \*\* 产量与销量的约束: \*\* 作物的产量不应超过需求量, 具体为:

$$Y_{m,t}^{(\omega)} \sum_i x_{i,m,t,s} \leq D_{m,t}^{(\omega)} \quad \forall m, t, s, \forall \omega \in \Omega \quad (5.8)$$

2. \*\* 耕地面积约束: \*\* 每块地的种植总面积不能超过其可用面积, 设第  $i$  块地的最大可用面积为  $A_{fall}$ , 约束条件为:

$$\sum_{m=1}^{41} x_{i,m,t} \leq A_{fall} \quad \forall i, \forall t \quad (5.9)$$

3. \*\* 作物种植数量限制: \*\* 每块地的种植作物数量有最大数量限制, 具体为:

$$\sum_{m=1}^{41} z_{i,m,t} \leq N_{\max} \quad \forall i, \forall t \quad (5.10)$$

### 求解过程

#### 输入数据:

- 作物的预期销售量  $D_{m,s}$ ;
- 每种作物的单价  $P_{m,s}$ , 亩产量  $Y_{m,s}$ , 种植成本  $C_{m,s}$ ;
- 各地块的可用面积  $A_{fall}$ , 以及每块地的作物最小种植面积限制  $\delta$ 。

**模型求解方法:** 采用蒙特卡洛采样法 (SAA) 与混合整数线性规划 (MILP) 方法求解此优化问题。具体步骤如下:

- 使用求解器 (如 Gurobi 或 CPLEX) 解决模型中的线性约束和目标函数。
- 对于每个年度  $t$ , 按照最大化收益的目标, 计算每个地块的作物种植面积分配。
- 考虑作物的销售限制、作物轮作约束以及豆类作物种植面积要求, 确保在求解过程中遵循所有约束。

**结果输出:** 求解器返回最优的种植方案, 输出每个地块在每年各季节的种植作物和面积。该结果可以填入对应的模板文件中。

通过此优化模型和求解过程, 我们可以得到 2024 至 2030 年间的最优种植方案, 确保农作物种植的经济效益最大化, 同时满足所有农业生产和管理要求。

### 5.2.2 算法伪代码流程

以下为模型求解的伪代码流程:

---

#### 算法 5.1: 农作物种植优化模型求解伪代码流程

---

**Input:** 作物的预期销售量  $D_{m,s}$ 、单价  $P_{m,s}$ 、亩产量  $Y_{m,s}$ 、种植成本  $C_{m,s}$ 、可用面积  $A_{fall}$ 、最小种植面积  $\delta$

**Data:** 场景数  $S$ , 风险系数  $\lambda$

- 1 **步骤 1: 场景数据生成**
  - 2 **for**  $\omega = 1$  **to**  $S$  **do**
  - 3     生成随机增量  $\delta_{m,t}^D$ 、 $\delta_{m,t}^Y$ 、 $\delta_{m,t}^C$ 、 $\delta_{m,t}^P$
  - 4     计算派生参数  $D_{m,t}^{(\omega)}$ 、 $Y_{m,t}^{(\omega)}$ 、 $C_{m,t}^{(\omega)}$ 、 $P_{m,t}^{(\omega)}$
  - 5 **步骤 2: 逐年逐地块优化**
  - 6 **for**  $t = 2024$  **to**  $2030$  **do**
  - 7     **for**  $i = 1$  **to**  $34$  **do**
  - 8         计算各作物种植面积  $x_{i,m,t,s}$  以最大化收益
  - 9         计算并更新作物种植成本与销售收益
  - 10         验证各项约束条件 (如面积限制、轮作约束等)
  - 11 **步骤 3: 求解优化模型**
  - 12 求解并获得最优解  $x^*$
  - 13 **步骤 4: 输出结果**
  - 14 输出每年各地块的作物种植方案
-

### 5.3 问题三：农作物种植优化策略（考虑价格弹性、替代性与互补性）

#### 5.3.1 模型建立与求解

在第三问中，我们需要基于第二问的模型，加入考虑农作物间的价格弹性、替代性和互补性等因素，来设计 2024-2030 年间的最优种植策略。根据实际市场情况，作物的价格、销量、种植成本之间存在相互影响，尤其是作物间的替代性和互补性需要特别考虑。本模型的目标是最大化农作物种植的经济效益，同时考虑供需关系对价格和销量的影响。

**目标函数：**与第二问相似，我们的目标函数依然是最大化农作物的收益，但是这次需要对价格和销量进行调整。通过引入价格和需求弹性系数，将作物的价格和需求纳入优化目标。具体公式如下：

$$\text{Maximize } Z = \sum_{t=2024}^{2030} \sum_{m=1}^{41} \sum_{i=1}^{34} \sum_{s=1}^2 \left( \tilde{P}_{m,t} \cdot Y_{m,t}^{\text{eff}} \cdot x_{i,m,t,s} - C_{m,t} \cdot x_{i,m,t,s} \right) \quad (5.11)$$

其中， $\tilde{P}_{m,t}$  为考虑替代性和互补性后的调整价格， $Y_{m,t}^{\text{eff}}$  为考虑轮作增益后的有效亩产量， $x_{i,m,t,s}$  为第  $i$  块地在第  $t$  年第  $s$  季节种植第  $m$  类作物的面积。

**有效需求：**在此模型中，我们将有效需求（ $\tilde{D}_{m,t}$ ）替代原始的需求量（ $D_{m,t}$ ），以便反映作物价格与销量之间的关系。有效需求是通过价格弹性和市场供需关系进行调节的，公式如下：

$$\tilde{D}_{m,t} = D_{m,t} \left( 1 + \eta_m \cdot \frac{\tilde{P}_{m,t} - P_{m,t}}{P_{m,t}} \right) \quad (5.12)$$

**作物替代性与互补性：**我们引入了替代性系数和互补性系数来调整作物之间的供需关系。作物的需求会受到同类作物增加（替代性）或互补作物增加（互补性）的影响。替代性和互补性会影响作物的价格和需求，公式如下：

$$\tilde{P}_{m,t} = P_{m,t} \left( 1 - \sum_{k \in \text{Subs}(m)} \alpha_{k \rightarrow m} \frac{Q_{k,t}}{Q_k^*} + \sum_{k \in \text{Comp}(m)} \eta_{k \rightarrow m} \frac{Q_{k,t}}{Q_k^*} \right) \quad (5.13)$$

$$\tilde{D}_{m,t} = \tilde{D}_{m,t} - \sum_{k \in \text{Subs}(m)} \alpha_{k \rightarrow m} Q_{k,t} + \sum_{k \in \text{Comp}(m)} \eta_{k \rightarrow m} Q_{k,t} \quad (5.14)$$

**轮作增益：**为了简化模型，我们假设轮作能够增加单产（例如，豆类与玉米的轮作可增加 10

$$Y_{i,m,t}^{\text{eff}} = Y_{m,t} \cdot (1 + \theta_m^{\text{rotation}} \cdot z_{i,m,t-1}) \quad (5.15)$$

其中,  $z_{i,m,t}$  为二元变量, 表示第  $i$  块地在第  $t$  年是否种植第  $m$  类作物。

### 5.3.2 求解过程

#### 输入数据:

- 作物的预期销售量  $D_{m,t}$ 。
- 每种作物的单价  $P_{m,t}$ , 亩产量  $Y_{m,t}$ , 种植成本  $C_{m,t}$ 。
- 作物替代性系数  $\alpha_{k \rightarrow j}$  和互补性系数  $\beta_{k \rightarrow j}$ 。
- 各地块的可用面积  $A_{fall}$ , 以及每块地的作物最小种植面积限制  $\delta$ 。
- 每种作物的轮作增益系数  $\theta_m^{\text{rotation}}$ 。

**模型求解方法:** 我们采用混合整数线性规划 (MILP) 方法来求解该问题, 具体步骤如下:

- 对于每个年度  $t$ , 利用供给和价格弹性关系, 计算调整后的价格  $\tilde{P}_{m,t}$  和有效需求  $\tilde{D}_{m,t}$ 。
- 引入替代性和互补性调整项, 根据作物间的相互关系调整作物的市场需求。
- 考虑轮作增益, 计算每个地块的有效单产  $Y_{m,t}^{\text{eff}}$ 。
- 使用求解器 (如 Gurobi 或 CPLEX) 求解优化问题, 得到每个地块的最优种植面积和作物分配方案。

**结果输出:** 求解器返回最优的种植方案, 输出每个地块在每年各季节的种植作物和面积, 并提供每种作物的调整后价格和有效需求。通过分析这些结果, 可以得出最优的农作物种植策略, 以最大化 2024 至 2030 年间的经济效益。

通过该优化模型, 我们能够考虑作物间的价格弹性、替代性与互补性以及轮作效应, 从而得到更加精细化和可行的农作物种植策略, 进一步提升农业生产的效益并减少市场风险。

### 5.3.3 结果分析

## 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

**多重实际约束考虑：**模型通过引入土壤轮作、作物销售限制、耕地面积限制等多项约束，确保了种植计划的可行性和合理性。在理论上，这种方法符合农业生产中的多重约束条件，能够反映现实中的复杂限制，确保每个决策变量的有效性。

**MILP 求解方法优势：**MILP 方法在理论和计算上具有良好的解决能力，尤其适合处理复杂的多维决策问题。通过采用该方法，可以在保证模型准确性的同时，迅速找到全局最优解，具有较强的计算效率和扩展性。

#### 6.2.2 问题二模型优点

**考虑不确定性：**通过引入蒙特卡洛采样（SAA）方法，模型能够有效地处理市场需求、单产、价格波动和种植成本的随机性。这使得模型能够更准确地反映未来农业生产中的不确定性，从而提供更为稳健的种植方案。

**优化收益和风险：**模型通过引入风险系数  $\lambda$ ，使得优化目标不仅仅是追求最大化的收益，还兼顾了风险管理。通过最小化收益的波动（方差），模型能够在收益和风险之间找到平衡，适应不同的风险偏好。

**灵活性强：**模型能够根据不同的场景生成需求、单产、价格和成本的随机增量，使得可以对不同的市场或环境变化进行适应性分析。用户可以根据需要选择不同的场景数量以及风险偏好，调整优化结果。

### 6.2.3 问题三模型优点

**替代性与互补性考虑：**本模型在作物种植优化过程中引入了作物之间的替代性和互补性理论。这种替代性和互补性假设在农业经济学中有着广泛的应用，能够反映不同作物间的市场联动效应，提高了模型的经济性和准确性。**轮作与增产效应线性化：**轮作和互补增产效应通过 Big-M 方法进行线性化处理，使得本模型依然能够通过 MILP 方法求解。该理论方法在保持模型计算可行性的同时，能够准确反映农业生产中的增产效应和土壤改良效应，提高了模型的现实适应性。**价格与需求精细调整：**模型通过调整价格系数和有效需求关系，精确模拟了作物间的市场联动效应。在理论上，这种方法能够准确反映价格波动对作物销售的影响，并使得优化方案更符合市场实际需求。

## 6.3 不足与改进方向

### 6.3.1 问题一模型的主要局限

**不考虑市场需求波动：**销售量和价格假设稳定，未考虑到市场供需变化对收益的影响。

### 6.3.2 问题一模型的改进方向

增加市场需求变化和价格波动的动态因素，模拟市场不确定性，改进预期销售量和单价的估计。

### 6.3.3 问题二模型的主要局限

**计算复杂度较高：**随机模拟和多场景求解增加了模型的计算复杂度，尤其是在考虑多个场景的情况下，求解过程可能会变得非常耗时。

### 6.3.4 问题二模型的改进方向

**优化计算方法：**可以采用启发式算法或近似算法（如遗传算法或模拟退火算法）来降低计算复杂度，同时保证模型的求解效率。

### 6.3.5 问题三模型的主要局限

**轮作增益的估算不准确：**轮作增益系数假设较为简单，可能忽略了一些农业生态因素的复杂性，导致增益估算存在误差。

### 6.3.6 问题三模型的改进方向

**加强数据校准与灵敏度分析：**加强市场数据和模型假设的校准，并进行灵敏度分析，评估模型对参数变化的敏感性，提高模型的稳健性和可信度。

## 参考文献



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）
说明文档	Q3/optimization_clusters.png	问题 3：聚类结果可视化
	README.txt	项目环境与运行指南

B. 优化函数

```
1 import numpy as np
2 import itertools
3 from scipy.optimize import linprog
4 import random
5 from deap import base, creator, tools, algorithms
6 import matplotlib.pyplot as plt
7 import pandas as pd
8 from typing import List, Tuple, Dict
9 import warnings
10 warnings.filterwarnings('ignore')
11
12 # 定义适应度函数和个体（如果未定义）
13 try:
14     creator.create("FitnessMax", base.Fitness, weights=(1.0,))
```

```

15         creator.create("Individual", list, fitness=creator.FitnessMax)
16     except Exception as e:
17         # 避免重复定义
18         pass
19
20     class TourismOptimizer:
21     def __init__(self):
22         # 基础数据
23         self.num_spots = 6
24         self.num_regions = 6
25
26         # 容量数据 (按0.6折减)
27         self.spot_cap = np.array([12000, 36000, 20000, 42000, 38000,
28             ↪ 30000]) * 0.6
29         self.night_cap_original = np.array([19000, 32000, 11000, 36000,
30             ↪ 23000, 22000]) * 0.6
31         self.night_cap = np.copy(self.night_cap_original) # 用于修改的容量
32         self.lunch_cap = np.array([23000, 39000, 13000, 45000, 31000,
33             ↪ 28000]) * 0.6
34
35         # 区域偏好度
36         self.region_prefer = np.array([7, 8, 9, 8, 6, 7])
37
38         # 原始费用和时间矩阵
39         self.cost_mat_raw = np.array([
40             [10, 25, 30, 18, 40, 25],
41             [np.inf, 10, 16, 24, 28, 18],
42             [np.inf, np.inf, 10, 24, 20, 15],
43             [np.inf, np.inf, np.inf, 10, 24, 16],
44             [np.inf, np.inf, np.inf, np.inf, 10, 22],
45             [np.inf, np.inf, np.inf, np.inf, np.inf, 10]
46         ])
47
48         self.time_mat_raw = np.array([
49             [30, 50, 60, 35, 70, 40],
50             [np.inf, 30, 25, 30, 40, 30],
51             [np.inf, np.inf, 15, 35, 30, 30],
52             [np.inf, np.inf, np.inf, 15, 35, 25],
53             [np.inf, np.inf, np.inf, np.inf, 20, 35],
54             [np.inf, np.inf, np.inf, np.inf, np.inf, 25]
55         ])
56
57         # 处理后的完整矩阵
58         self.cost_mat = None
59         self.time_mat = None
60
61         # 路线相关
62         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
71         # ↪ 间权重={time_weight}) ...") # 避免过多打印
72
73         # 创建邻接矩阵 (景点-区域双向图)
74         n = self.num_spots + self.num_regions # 总节点数
75
76         # 初始化矩阵
77         combined_full = np.full((n, n), np.inf)
78         cost_full = np.full((n, n), np.inf)
79         time_full = np.full((n, n), np.inf)
80
81         # 对角线为0
82         np.fill_diagonal(combined_full, 0)
83         np.fill_diagonal(cost_full, 0)
84         np.fill_diagonal(time_full, 0)
85
86         # 计算归一化常数
87         finite_costs = self.cost_mat_raw[np.isfinite(self.cost_mat_raw)]
88         finite_times = self.time_mat_raw[np.isfinite(self.time_mat_raw)]
89         cost_max = np.max(finite_costs) if len(finite_costs) > 0 else 100
90         time_max = np.max(finite_times) if len(finite_times) > 0 else 100
91
92         # 填入已知的景点到区域的连接
93         for r in range(self.num_regions):
94             for s in range(self.num_spots):
95                 if not np.isinf(self.cost_mat_raw[r, s]):
96                     cost = self.cost_mat_raw[r, s]
97                     time = self.time_mat_raw[r, s]
98
99                     # 归一化
100                     cost_normalized = cost / cost_max
101                     time_normalized = time / time_max
102
103                     # 加权综合成本
104                     combined_cost = cost_weight * cost_normalized +
105                     # ↪ time_weight * time_normalized
106
107                     # 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] <
120                     ⇨ combined_full[i, j]:
121                     combined_full[i, j] = combined_full[i, k] +
122                         ⇨ combined_full[k, j]
123                     # 同时更新对应的实际费用和时间
124                     cost_full[i, j] = cost_full[i, k] + cost_full[k, j]
125                     ⇨ ]
126                     time_full[i, j] = time_full[i, k] + time_full[k, j]
127                     ⇨ ]
128
129     # 提取景点到区域的部分
130     self.cost_mat = cost_full[:self.num_spots, self.num_spots:]
131     self.time_mat = time_full[:self.num_spots, self.num_spots:]
132
133     # print("矩阵预处理完成") # 避免过多打印
134
135     def generate_1day_routes(self):
136         """生成所有一日游路线: S -> R -> S"""
137         # print("生成一日游路线...") # 避免过多打印
138         self.routes_1day = []
139
140         for s1 in range(self.num_spots):
141             for r in range(self.num_regions):
142                 for s2 in range(self.num_spots):
143                     if s1 != s2: # 避免重复访问同一景点
144                         route = {
145                             'type': '1day',
146                             'path': [s1, r, s2],
147                             'spots': [s1, s2],
148                             'regions': [r],
149                             'lunch_regions': [r],
150                             'night_regions': [],
151                             'cost': self.cost_mat[s1, r] + self.cost_mat[
152                                 ⇨ s2, r],
153                             'time': self.time_mat[s1, r] + self.time_mat[

```

```

149         ↪ s2, r],
        '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: #
169                                         ↪ 所有景点不重复
170                                         if count >= max_routes:
171                                             break
172
173                                     route = {
174                                         'type': '2day',
175                                         'path': [s1, r1, s2, r2, s3, r3,
176                                             ↪ s4],
177                                         'spots': [s1, s2, s3, s4],
178                                         'regions': [r1, r2, r3],
179                                         'lunch_regions': [r1, r3], # 第1
180                                             ↪ 天午餐r1, 第2天午餐r3
181                                         'night_regions': [r2], # 第1天住
182                                             ↪ 宿r2
183                                         'cost': (self.cost_mat[s1, r1] +
184                                             ↪ self.cost_mat[s2, r1] +
185                                                 self.cost_mat[s2, r2] +
186                                                 ↪ self.cost_mat[s3,
187                                                     ↪ r2] +
188                                                 self.cost_mat[s3, r3] +
189                                                 ↪ self.cost_mat[s4,
190                                                     ↪ r3]),
191                                         'time': (self.time_mat[s1, r1] +
192                                             ↪ self.time_mat[s2, r1] +
193                                                 self.time_mat[s2, r2] +
194                                                 ↪ self.time_mat[s3,

```

```

184         ↪ r2] +
        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.
239                 ↪ cost_mat[spots[1], regions[0]]
240             cost += self.cost_mat[spots[1], regions[1]] # Travel from
241                 ↪ S2 to R2 for overnight
242             time += self.time_mat[spots[0], regions[0]] + self.
243                 ↪ time_mat[spots[1], regions[0]]
244             time += self.time_mat[spots[1], regions[1]]
245
246             # Day 2: S2->R2->S3->R3->S4->R4 (Overnight in R4)
247             # Travel from R2 to S3 (already at R2)
248             cost += self.cost_mat[spots[2], regions[2]]
249             cost += self.cost_mat[spots[3], regions[2]]
250             cost += self.cost_mat[spots[3], regions[3]] # Travel from
251                 ↪ S4 to R4 for overnight
252             time += self.time_mat[spots[2], regions[2]]
253             time += self.time_mat[spots[3], regions[2]]
254             time += self.time_mat[spots[3], regions[3]]
255
256             # Day 3: S4->R4->S5->R5->S6 (End trip)
257             # Travel from R4 to S5 (already at R4)
258             cost += self.cost_mat[spots[4], regions[4]]
259             cost += self.cost_mat[spots[5], regions[4]]
260             time += self.time_mat[spots[4], regions[4]]
261             time += self.time_mat[spots[5], regions[4]]
262
263             preference = sum([self.region_prefer[r] for r in regions])
264                 ↪ + 6 # 6 unique spots contribute to base preference
265
266             # 归一化计算 (与MILP保持一致)
267             max_preference_3day = 9 * 5 + 6 # 理论最大偏好度
268             # 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
269                 ↪ 确, 只要能大致区分好坏

```

```

264         max_cost_estimate = 100 * 10 # 粗略估计
265         max_time_estimate = 150 * 10 # 粗略估计
266
267         pref_normalized = preference / max_preference_3day
268         cost_normalized = cost / max_cost_estimate
269         time_normalized = time / max_time_estimate
270
271         # 加权综合评分 (权重与MILP一致)
272         w_pref, w_cost, w_time = 0.6, 0.2, 0.2
273         fitness = w_pref * pref_normalized - w_cost *
                ↪ cost_normalized - w_time * time_normalized
274
275         return (fitness,)
276     except:
277         return (-1000,) # 无效路径惩罚
278
279     toolbox.register("individual", tools.initIterate, creator.
                ↪ Individual, create_individual)
280     toolbox.register("population", tools.initRepeat, list, toolbox.
                ↪ individual)
281     toolbox.register("evaluate", evaluate)
282
283     def crossover_individual(ind1, ind2):
284         """自定义交叉函数, 确保景点不重复"""
285         # 对景点部分使用顺序交叉(OX)
286         spots1, spots2 = ind1[:6], ind2[:6]
287         regions1, regions2 = ind1[6:], ind2[6:]
288
289         # 顺序交叉(Order Crossover)
290         def order_crossover(parent1, parent2):
291             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)
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))

```

```

355         offspring = list(map(toolbox.clone, offspring))
356
357     # 交叉和变异
358     for child1, child2 in zip(offspring[::2], offspring[1::2]):
359         if random.random() < 0.5:
360             toolbox.mate(child1, child2)
361             del child1.fitness.values
362             del child2.fitness.values
363
364     for mutant in offspring:
365         if random.random() < 0.2:
366             toolbox.mutate(mutant)
367             del mutant.fitness.values
368
369     # 评估无效个体
370     invalid_ind = [ind for ind in offspring if not ind.fitness.
371                    ↪ valid]
372     fitnesses = map(toolbox.evaluate, invalid_ind)
373     for ind, fit in zip(invalid_ind, fitnesses):
374         ind.fitness.values = fit
375
376     population[:] = offspring # Update population with new
377                               ↪ generation
378
379     # if gen % 10 == 0: # 避免过多打印
380     #     fits = [ind.fitness.values[0] for ind in population if
381     #             ↪ ind.fitness.valid]
382     #     if fits:
383     #         print(f"Generation {gen}: Max={max(fits):.2f}, Avg={
384     #             ↪ np.mean(fits):.2f}")
385
386     # 提取最优个体作为候选路线, 保证多样性
387     population.sort(key=lambda x: x.fitness.values[0], reverse=True)
388
389     self.routes_3day_candidates = []
390     used_route_keys = set()
391
392     for ind in population:
393         spots = ind[:6]
394         regions = ind[6:]
395
396         # 验证景点不重复
397         if len(set(spots)) != 6:
398             continue # 跳过有重复景点的个体
399
400         # 创建路线唯一标识
401         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)}")
442     ↪ # 避免过多打印
443
444 def generate_3day_routes_hybrid(self, target_routes=5000):
445     """混合方法生成三日游路线: 遗传算法 + 启发式规则"""
446     print("使用混合方法生成三日游路线...")
447
448     self.routes_3day_candidates = []
449
450     # 1. 先用遗传算法生成高质量路线
451     print("步骤1: 遗传算法生成高质量路线...")
452     self.generate_3day_candidates_ga(candidate_size=int(target_routes
453         ↪ * 0.1), population_size=1000, generations=50) # 减少GA生成的
454     ↪ 比例
455
456     ga_routes_count = len(self.routes_3day_candidates)
457     print(f"遗传算法生成: {ga_routes_count}条路线")
458
459     # 2. 启发式生成多样化路线
460     print("步骤2: 启发式生成多样化路线...")
461     used_route_keys = set()
462     for route in self.routes_3day_candidates:
463         route_key = tuple(route['spots'] + route['regions'])
464         used_route_keys.add(route_key)
465
466     # 为每个区域组合生成路线
467     # 考虑更系统地生成, 而不是固定组合
468     # 随机生成区域组合, 增加多样性
469
470     num_heuristic_routes = target_routes - ga_routes_count
471
472     # 尝试生成更多样化的区域组合
473     for _ in range(num_heuristic_routes * 2): # 尝试生成更多, 因为会有
474         ↪ 重复或无效
475         if len(self.routes_3day_candidates) >= target_routes:
476             break
477
478     # 随机选择5个区域, 可以重复
479     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]] +

```

```

507         ↪ self.time_mat[spots[1], region_combo[0]] +
        self.time_mat[spots[1], region_combo[1]] +
        ↪ self.time_mat[spots[2], region_combo
508         ↪ [1]] +
        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

```

```

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
604                 ↪ self.night_cap
605             b_ub.append(total_night_capacity)
606
607         # 等式约束: 总游客数
608         A_eq = [[1] * n_routes]
609         b_eq = [total_tourists]
610
611         # 变量边界
612         bounds = [(0, total_tourists) for _ in range(n_routes)]
613
614         try:
615             # 求解
616             result = linprog(c, A_ub=A_ub, b_ub=b_ub, A_eq=A_eq, b_eq=b_eq
617                 ↪ ,
618                               bounds=bounds, method='highs')
619
620             if result.success:
621                 solution = []
622                 total_s = 0
623                 total_t = 0
624                 for i, x in enumerate(result.x):
625                     if x > 0.1: # 过滤掉很小的值
626                         num_tourists = int(round(x))
627                         route_info = routes[i].copy()
628                         route_info['tourists'] = num_tourists
629                         # 重新计算归一化满意度用于显示
630                         pref_normalized = route_info['preference'] /
631                             ↪ get_max_preference(route_info['type'])

```



```

629         cost_normalized = route_info['cost'] / max_cost
630         time_normalized = route_info['time'] / max_time
631         route_info['satisfaction'] = w_pref *
            ↪ pref_normalized - w_cost * cost_normalized -
            ↪ w_time * time_normalized
632         solution.append(route_info)
633         total_s += num_tourists * route_info['satisfaction
            ↪ ']
634         total_t += num_tourists
635
636         return {'solution': solution, 'total_satisfaction':
            ↪ total_s, 'total_tourists': total_t}
637     else:
638         # print("MLP求解失败") # 避免过多打印
639         return {'solution': [], 'total_satisfaction': 0, '
            ↪ total_tourists': 0}
640     except Exception as e:
641         # print(f"MLP求解出错: {e}") # 避免过多打印
642         return {'solution': [], 'total_satisfaction': 0, '
            ↪ total_tourists': 0}
643
644     def run_optimization(self, cost_weight=0.5, time_weight=0.5,
        ↪ tourists_1day=30000, tourists_2day=30000, tourists_3day=20000):
645         """运行完整的优化流程
646
647         Args:
648             cost_weight: 费用权重, 用于路径预处理 (默认0.5)
649             time_weight: 时间权重, 用于路径预处理 (默认0.5)
650             tourists_1day: 一日游总游客数
651             tourists_2day: 二日游总游客数
652             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                               ↪ ])
716         total_cost = sum([route['tourists'] * route['cost'] for route
717                           ↪ in solution])
718         total_time = sum([route['tourists'] * route['time'] for route
719                           ↪ in solution])
720         total_satisfaction = sum([route['tourists'] * route['
721                                   ↪ satisfaction'] for route in solution])
722
723         print(f"路线类型数量: {len(solution)}")
724         print(f"总游客数: {total_tourists}")
725         print(f"总交通成本: {total_cost:.0f} 元")
726         print(f"总交通时间: {total_time:.0f} 分钟")
727         print(f"总满意度: {total_satisfaction:.2f}")
728         print(f"平均满意度: {total_satisfaction/total_tourists:.2f}")
729
730         print("\n具体路线分配:")
731         for i, route in enumerate(solution[:10]): # 只显示前10个
732             spots_str = "->".join([f"S{s+1}" for s in route['spots']])
733             regions_str = "->".join([f"R{r+1}" for r in route['regions
734                                     ↪ ']])
735             print(f"路线{i+1}: {route['tourists']}人")
736             print(f"    景点路径: {spots_str}")
737             print(f"    区域路径: {regions_str}")
738             print(f"    成本: {route['cost']:.1f}元, 时间: {route['time
739                                     ↪ ']:.1f}分钟, 偏好: {route['preference']}")
740             print()
741
742         # 扩容成本函数
743         def expansion_cost(delta_K: float) -> float:
744             """
745             扩容成本函数
746             Args:
747                 delta_K: 新增接待能力 (万人次)
748             Returns:
749                 扩容成本 (亿元)
750             """
751             c1 = 0.0007 # 亿元
752             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     #  $B(\delta_K) = v_s * [Z(K) - Z(K_0)] + v_p * [N(K) - N(K_0)]$ 
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:
18         ↪ float = 1e-4):
19         """
20         初始化K均值聚类器
21
22         Args:
23             k: 聚类数量
24             max_iterations: 最大迭代次数
25             tolerance: 收敛阈值
26
27         """
28         self.k = k
29         self.max_iterations = max_iterations
30         self.tolerance = tolerance
31         self.centroids = None
32         self.labels = None
33         self.inertia_ = None
34
35     def custom_distance(self, route1: Dict, route2: Dict) -> float:
36         """
37         自定义距离函数：计算两个旅游方案之间的相似度距离
38
39         Args:
40             route1: 第一个旅游方案
41             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(
51             ↪ spots1.union(spots2))
52
53     # 2. 区域相似度（Jaccard相似度）
54     regions1 = set(route1.get('regions', []))
55     regions2 = set(route2.get('regions', []))
56     if len(regions1.union(regions2)) == 0:
57         regions_similarity = 0
58     else:
59         regions_similarity = len(regions1.intersection(regions2)) /
60             ↪ len(regions1.union(regions2))
61
62     # 3. 路线类型相似度
63     type1 = route1.get('type', '')
64     type2 = route2.get('type', '')
65     type_similarity = 1.0 if type1 == type2 else 0.0
66
67     # 4. 成本相似度（归一化欧几里得距离）
68     cost1 = route1.get('cost', 0)
69     cost2 = route2.get('cost', 0)
70     max_cost = max(cost1, cost2) if max(cost1, cost2) > 0 else 1
71     cost_similarity = 1 - abs(cost1 - cost2) / max_cost
72
73     # 5. 时间相似度（归一化欧几里得距离）
74     time1 = route1.get('time', 0)
75     time2 = route2.get('time', 0)
76     max_time = max(time1, time2) if max(time1, time2) > 0 else 1
77     time_similarity = 1 - abs(time1 - time2) / max_time
78
79     # 6. 偏好相似度（归一化欧几里得距离）
80     pref1 = route1.get('preference', 0)
81     pref2 = route2.get('preference', 0)
82     max_pref = max(pref1, pref2) if max(pref1, pref2) > 0 else 1
83     pref_similarity = 1 - abs(pref1 - pref2) / max_pref
84
85     # 加权综合距离（转换为距离，越小越相似）
86     weights = {
87         '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,
127                                                         ↪ other_route)
128             avg_distance = total_distance / (len(cluster_routes) - 1)
129             avg_distances.append((avg_distance, route))
130
131     # 返回平均距离最小的路线作为中心
132     return min(avg_distances, key=lambda x: x[0])[1]

```

```

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
145                 ↪ .k})")
146
147         # 随机初始化聚类中心
148         self.centroids = random.sample(routes, self.k)
149
150         for iteration in range(self.max_iterations):
151             # 分配阶段：将每个路线分配到最近的聚类中心
152             labels = []
153             total_distance = 0
154
155             for route in routes:
156                 min_distance = float('inf')
157                 best_cluster = 0
158
159                 for i, centroid in enumerate(self.centroids):
160                     distance = self.custom_distance(route, centroid)
161                     if distance < min_distance:
162                         min_distance = distance
163                         best_cluster = i
164
165                 labels.append(best_cluster)
166                 total_distance += min_distance
167
168             # 更新阶段：重新计算聚类中心
169             new_centroids = []
170             for i in range(self.k):
171                 cluster_routes = [routes[j] for j in range(len(routes)) if
172                     ↪ labels[j] == i]
173                 if cluster_routes:
174                     new_centroid = self.find_centroid(cluster_routes)
175                     new_centroids.append(new_centroid)
176                 else:
177                     # 如果某个聚类为空，随机选择一个新中心
178                     new_centroids.append(random.choice(routes))
179
180             # 检查收敛

```

```

179         centroid_changed = False
180         for i in range(self.k):
181             if self.custom_distance(self.centroids[i], new_centroids[i
182                                     ↵ ])) > self.tolerance:
183                 centroid_changed = True
184                 break
185
186         self.centroids = new_centroids
187
188         if not centroid_changed:
189             print(f"算法在第{iteration + 1}次迭代后收敛")
190             break
191
192         self.labels = labels
193         self.inertia_ = total_distance
194
195         return self
196
197     def predict(self, routes: List[Dict]) -> List[int]:
198         """
199         预测新路线的聚类标签
200
201         Args:
202             routes: 待预测的路线列表
203
204         Returns:
205             List[int]: 聚类标签
206         """
207         if self.centroids is None:
208             raise ValueError("模型尚未训练，请先调用fit方法")
209
210         labels = []
211         for route in routes:
212             min_distance = float('inf')
213             best_cluster = 0
214
215             for i, centroid in enumerate(self.centroids):
216                 distance = self.custom_distance(route, centroid)
217                 if distance < min_distance:
218                     min_distance = distance
219                     best_cluster = i
220
221             labels.append(best_cluster)
222
223         return labels
224
225     def get_cluster_info(self, routes: List[Dict]) -> Dict:
226         """

```



```

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
241                             ↪ self.labels[j] == i]
242
243         if cluster_routes:
244             # 计算聚类统计信息
245             costs = [route.get('cost', 0) for route in cluster_routes]
246             times = [route.get('time', 0) for route in cluster_routes]
247             preferences = [route.get('preference', 0) for route in
248                             ↪ cluster_routes]
249             types = [route.get('type', '') for route in cluster_routes
250                     ↪ ]
251
252             cluster_info[i] = {
253                 'size': len(cluster_routes),
254                 'centroid': self.centroids[i],
255                 'avg_cost': np.mean(costs),
256                 'avg_time': np.mean(times),
257                 'avg_preference': np.mean(preferences),
258                 'type_distribution': dict(zip(*np.unique(types,
259                     ↪ return_counts=True))),
260                 'routes': cluster_routes
261             }
262
263     return cluster_info
264
265 def visualize_clusters(self, routes: List[Dict], save_path: str = None
266     ↪ ):
267     """
268     可视化聚类结果
269
270     Args:
271         routes: 原始路线列表
272         save_path: 保存图片的路径
273     """

```

```

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_iter=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

```

```

408     print(f"\n聚类完成！模型惯性（总距离）：{kmeans.inertia_:.4f}")
409
410
411     if __name__ == "__main__":
412         main()

```

## D. 运行函数

```

1     #!/usr/bin/env python3
2     # -*- coding: utf-8 -*-
3     """
4     旅游路线优化运行脚本
5     """
6     import deap
7     from tourism_optimization import TourismOptimizer, expansion_cost # Import
        ↳ expansion cost function
8     import numpy as np # Import numpy
9     import matplotlib.pyplot as plt
10    import matplotlib # Import matplotlib to configure fonts
11
12    # Import K-means clustering algorithm
13    from Q2_k_means import TourismKMeans
14
15    # --- Start: Matplotlib Chinese Font Configuration ---
16    # Configure font to support Chinese characters
17    # Try 'SimHei' first, if not available, try 'Microsoft YaHei' or '
        ↳ WenQuanYi Micro Hei'
18    matplotlib.rcParams['font.sans-serif'] = ['SimHei', 'Microsoft YaHei', '
        ↳ Arial Unicode MS']
19    matplotlib.rcParams['axes.unicode_minus'] = False # Solve the problem of
        ↳ '-' displaying as a square
20    # --- End: Matplotlib Chinese Font Configuration ---
21
22    def main():
23        print("=" * 60)
24        print("重庆旅游路线优化系统")
25        print("=" * 60)
26
27        # Create optimizer instance
28        optimizer = TourismOptimizer()
29
30        # Run optimization
31        try:
32            # Test different weight configurations
33            weight_configs = [
34                (0.5, 0.5, "均衡型"),
35                # (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={
44             ↪ time_w})")
45         print(f"{' '*80}")
46
47         # Run optimization and get results
48         results = optimizer.run_optimization(cost_weight=cost_w,
49             ↪ time_weight=time_w)
50         all_results[config_name] = results
51
52         # Print results
53         optimizer.print_results(results)
54
55         # Save results to file
56         save_results_to_file(results, suffix=f"_{config_name}")
57
58     # Compare results of different configurations
59     compare_configurations(all_results)
60
61     # ===== Problem 3: New Hotel Expansion Analysis =====
62     print(f"\n{' '*80}")
63     print("问题3: 新建旅馆扩容分析")
64     print(f"{' '*80}")
65     analyze_hotel_expansion(optimizer)
66
67     # ===== New: K-means Clustering Analysis =====
68     print(f"\n{' '*80}")
69     print("K均值聚类分析: 旅游路线相似度分析")
70     print(f"{' '*80}")
71     perform_kmeans_analysis(all_results)
72
73 except Exception as e:
74     print(f"优化过程中出现错误: {e}")
75     import traceback
76     traceback.print_exc()
77
78 def compare_configurations(all_results):
79     """比较不同权重配置的结果"""
80     print(f"\n{' '*80}")
81     print("不同权重配置结果比较")
82     print(f"{' '*80}")

```

```

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} "  
                f"{total_time:<12.0f} {avg_satisfaction:<12.2f}")
98
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

```

```

120         if not solution:
121             f.write("无可行解\n\n")
122             continue
123
124         total_tourists = sum([route['tourists'] for route in solution
125                               ↪ ])
126         total_cost = sum([route['tourists'] * route['cost'] for route
127                           ↪ in solution])
128         total_time = sum([route['tourists'] * route['time'] for route
129                           ↪ in solution])
130         total_satisfaction = sum([route['tourists'] * route['
131                                   ↪ satisfaction'] for route in solution])
132
133         f.write(f"路线类型数量: {len(solution)}\n")
134         f.write(f"总游客数: {total_tourists}\n")
135         f.write(f"总交通成本: {total_cost:.0f} 元\n")
136         f.write(f"总交通时间: {total_time:.0f} 分钟\n")
137         f.write(f"总满意度: {total_satisfaction:.2f}\n")
138         f.write(f"平均满意度: {total_satisfaction/total_tourists:.2f}\
139                 ↪ n\n")
140
141         f.write("具体路线分配:\n")
142         for i, route in enumerate(solution[:10]):
143             spots_str = "->".join([f"S{s+1}" for s in route['spots']])
144             regions_str = "->".join([f"R{r+1}" for r in route['regions
145                                     ↪ ']])
146             f.write(f"路线{i+1}: {route['tourists']} 人\n")
147             f.write(f"  景点路径: {spots_str}\n")
148             f.write(f"  区域路径: {regions_str}\n")
149             f.write(f"  成本: {route['cost']:.1f}元, 时间: {route['
150                   ↪ time']:.1f}分钟, 偏好: {route['preference']}\n\n")
151         f.write("\n")
152
153     print(f"\n结果已保存到 {filename} 文件")
154
155 def analyze_hotel_expansion(optimizer: TourismOptimizer):
156     """
157     Analyzes the hotel expansion problem to find the optimal expansion
158     ↪ area and scale.
159     """
160     print("\n开始分析旅馆扩容方案...")
161
162     # Define economic value parameters
163     # vs: Economic value per unit satisfaction (billion CNY/satisfaction)
164     # vp: Average profit per tourist (billion CNY/person)
165     v_s_per_million_satisfaction = 0.1 # Assume 1 million satisfaction
166     ↪ 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

```

```

196         current_results = optimizer.run_optimization(
197             cost_weight=0.5, time_weight=0.5, # Use balanced weights
198             ↪ for expansion analysis
199             tourists_1day=30000, tourists_2day=30000, tourists_3day
200             ↪ =20000
201         )
202     Z_K_total, N_K_total = optimizer.get_aggregated_metrics(
203         ↪ current_results)
204     N_K_total_wan = N_K_total / 10000.0
205
206     # Calculate benefit B(delta_K)
207     B_delta_K = (v_s_per_million_satisfaction * (Z_K_total -
208         ↪ Z_K0_total) +
209         v_p_per_million_tourists * (N_K_total_wan -
210             ↪ N_K0_total_wan))
211
212     # Calculate cost C(delta_K)
213     C_delta_K = expansion_cost(delta_K_wan)
214
215     # Calculate net benefit F(delta_K)
216     F_delta_K = B_delta_K - C_delta_K
217
218     expansion_results.append({
219         'region_idx': r_idx,
220         'region_name': f'区域{r_idx + 1}',
221         'delta_K_wan': delta_K_wan,
222         'original_night_cap_wan': original_night_cap_r / 10000.0,
223         'new_night_cap_wan': current_night_cap_r / 10000.0,
224         'Z_K_total': Z_K_total,
225         'N_K_total_wan': N_K_total_wan,
226         'Benefit_B': B_delta_K,
227         'Cost_C': C_delta_K,
228         'Net_Benefit_F': F_delta_K
229     })
230
231     # Update optimal solution
232     if F_delta_K > best_net_benefit:
233         best_net_benefit = F_delta_K
234         best_region_idx = r_idx
235         best_delta_K_wan = delta_K_wan
236
237     # Restore the original capacity of this region for the next region
238     ↪ 's test
239     optimizer.night_cap[r_idx] = optimizer.night_cap_original[r_idx]
240
241     print("\n扩容分析结果:")
242     print("-" * 60)
243     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)

```

```

320         route_id += 1
321
322     if not all_routes:
323         print("没有找到可用的路线数据进行聚类分析")
324         return
325
326     print(f"收集到 {len(all_routes)} 条路线用于聚类分析")
327
328     # 2. 进行K均值聚类
329     # 尝试不同的k值
330     k_values = [3, 4, 5, 6]
331     best_k = 3
332     best_inertia = float('inf')
333
334     print("\n选择最优聚类数量...")
335     for k in k_values:
336         if k <= len(all_routes):
337             kmeans = TourismKMeans(k=k, max_iterations=50)
338             kmeans.fit(all_routes)
339             print(f"k={k}: 惯性值={kmeans.inertia_:.4f}")
340
341             if kmeans.inertia_ < best_inertia:
342                 best_inertia = kmeans.inertia_
343                 best_k = k
344
345     print(f"\n选择最优k值: {best_k}")
346
347     # 3. 使用最优k值进行最终聚类
348     print(f"\n使用k={best_k}进行最终聚类...")
349     final_kmeans = TourismKMeans(k=best_k, max_iterations=100)
350     final_kmeans.fit(all_routes)
351
352     # 4. 分析聚类结果
353     print(f"\n聚类结果分析:")
354     cluster_info = final_kmeans.get_cluster_info(all_routes)
355
356     print("-" * 100)
357     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}")
358     print("-" * 100)
359
360     for cluster_id, info in cluster_info.items():
361         main_type = max(info['type_distribution'].items(), key=lambda x: x
    ↪ [1])[0] if info['type_distribution'] else "未知"
362         avg_tourists = np.mean([route.get('tourists', 0) for route in info
    ↪ ['routes']])
363
364         print(f"{'cluster_id':<6} {'info['size']':<6} {'info['avg_cost']':<10.1f

```

```

365         ↪ } {info['avg_time']:<10.1f} "
           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{
466                 ↪ spot+1}', 0) + 1
467         popular_spots = sorted(popular_spots.items(), key=lambda x
468             ↪ : x[1], reverse=True)[:5]
469         f.write(f"热门景点: {popular_spots}\n")
470
471     if all_regions:
472         popular_regions = {}
473         for region in all_regions:
474             popular_regions[f'R{region+1}'] = popular_regions.get(
475                 ↪ f'R{region+1}', 0) + 1
476         popular_regions = sorted(popular_regions.items(), key=
477             ↪ lambda x: x[1], reverse=True)[:5]
478         f.write(f"热门区域: {popular_regions}\n")
479
480     f.write("\n代表性路线 (前5条):\n")
481     for i, route in enumerate(info['routes'][:5]):
482         spots_str = "->".join([f"S{s+1}" for s in route.get('spots
483             ↪ ', [])])
484         regions_str = "->".join([f"R{r+1}" for r in route.get('
485             ↪ regions', [])])
486         f.write(f"    {i+1}. {route.get('type', 'unknown')}路线 ({
487             ↪ route.get('config', 'unknown')}配置)\n")
488         f.write(f"        景点: {spots_str}\n")
489         f.write(f"        区域: {regions_str}\n")
490         f.write(f"        成本: {route.get('cost', 0):.1f}元, 时间: {
491             ↪ route.get('time', 0):.1f}分钟\n")
492         f.write(f"        游客数: {route.get('tourists', 0)}, 满意度:
493             ↪ {route.get('satisfaction', 0):.3f}\n\n")
494
495     f.write("\n")

```



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