

从三维数据到调度方案：面向土方挖掘的调度方案规划算法

摘要

在国家对大江大河治理和高速公路网络建设日益重视的背景下，本文深入探讨了河道整治与高速公路修建过程中面临的复杂工程问题。针对这些问题，本文提出了一种创新的解决方案，即利用数学建模与三维重建技术，对河道开挖与高速公路边坡修建进行精确建模和优化调度。

在任务一中，我们基于航道挖掘的勘探图数据，系统开展了三维建模与高程分析，通过数据驱动的方式解析了勘探图中的关键图层信息，重点提取了边坡设计所需的坡比参数。以航道线高程为基准高度，构建了多层级几何解析模型，通过空间坐标系转换与级联计算，精确推导了边坡线在立体空间中的 Z 轴坐标。为验证模型的科学性，我们还开展了多维度可视化分析，通过基于 **Delaunay 三角剖分** 的边坡示意图展示空间结构，二维高程等高线图刻画高程分布规律，形成了完整的空间认知体系。

在任务二中，聚焦高速公路边坡三维建模，以桂林外环高速 B 段边坡为对象。通过简化模型，将路段中心线近似为直线降低复杂度；构建三维空间直角坐标系，明确各轴定义及对应关系；从 CAD 图纸提取二维数据，利用参照点和工程参数，通过特定映射关系将二维坐标转换为三维坐标；采用 Delaunay 三角剖分进行空间插值，生成连续曲面，满足边坡工程分析需求。

基于任务一和任务二的结果，我们在任务三中，首先通过读取挖高数据和截面面积数据，计算了土方挖掘的粗略体积，之后通过二重积分与立方堆积体积之差计算出了回填土方体积。在方案规划中，我们首先规范了优化问题的标准形式，之后将问题拆解为 CVRP 问题（带容量约束的车辆路径问题）和聚类参数与车辆数选择问题两个子问题。在处理 CVRP 问题时，我们采用**蚁群算法**对于每个类别的待挖掘区域进行最优方案求解。对于聚类参数选择和车辆数量处理这一问题，我们采用**粒子群算法**，将构造好的 CVRP 问题的求解器作为粒子群算法的适应度计算接口，进而实现全局优化，确定了对应的聚类选择参数和车辆数量。实验结果表明，当挖掘机数量为 27 辆，装卸点数为 14，河道开挖得到最优解，时间和成本分别为 447.66 小时和 471,984 元。当挖掘机数量为 25 辆，装卸点数为 10，边坡修建得到最优解，时间和成本分别为 259.36 小时和 210,420.63 元。

最后，我们还对本问题进行了进一步探索。我们从计算机视觉的角度出发，探讨了端到端三维重建模型对于快速获取三维勘探数据的重要意义。我们复现了这一领域的前沿工作——VGGT^[1] (Visual Geometry Grounded Transformer)，并对真实的高速公路图片进行了三维数据解析重建。

注：您可以访问 <https://immc25140846.github.io/Diggingplan-page/> 来查看更多有关本研究的可视化和完整代码，页面正在更新中。为保证同行评议的双盲性，该 github 账号为全新账号。

关键词：三维重建 调度规划 CVRP 问题 蚁群算法 粒子群算法

目录

1 引言	3
1.1 问题背景	3
1.2 问题重述	3
1.3 研究概述	3
2 模型准备	4
2.1 假设与解释	4
2.2 符号说明	5
3 任务一：对河道挖掘方案的三维建模	5
3.1 概述	5
3.2 河道边坡高程计算模型	6
4 任务二：对高速公路边坡修建的三维建模	8
4.1 概述	8
4.2 高速公路边坡高程计算模型	9
5 任务三：挖方体积计算与最优调度路径规划	11
5.1 概述	11
5.2 挖方体积计算	11
5.3 调度方案优化问题的标准化	13
5.4 问题分解	14
5.5 基于蚁群算法的 CVRP 问题求解	16
5.6 基于粒子群算法的全局优化问题求解	18
5.7 实验结果与可视化	19
6 拓展：从图片到三维模型	20
7 总结	22
8 附录	22

1 引言

1.1 问题背景

近年来，随着国家对大江大河治理和高速公路网络建设的战略部署，河道整治与高速公路修建已成为现代基础设施建设的核心任务。这类工程规模庞大、技术复杂，尤其在破土环节中面临地质条件多变、施工精度要求高、环境影响深远等多重挑战。传统施工方法依赖人工经验与分散式管理，难以实现高效协同与动态优化，导致资源浪费、工期延误及安全隐患频发。例如，河道整治需统筹污染源控制、生态修复与水质监测，而传统手段在系统性治理与实时响应能力上存在明显短板；高速公路施工在应对冻土、岩溶等复杂地质时，仍依赖经验判断与粗放式作业模式，难以兼顾效率与安全性。在此背景下，如何通过先进技术手段提升施工智能化水平，成为破解工程难题的关键突破口。人工智能、物联网、大数据等新兴技术的引入，为工程设计、施工过程监控及资源调度提供了全新解决方案，推动基础设施建设从“经验驱动”向“数据驱动”转型。

1.2 问题重述

任务一：我们需要正确读取勘探图的关键信息，根据航道线高程-7.61m 这一基准高度，利用几何解析方法计算边坡线的 z 轴信息。

任务二：我们需要将任务一的方法进行迁移，读取高速公路山体的勘探图数据，进而构建山体的三维模型。

任务三：我们需要基于任务一和任务二得出的三维信息，计算土方挖掘的体积，并使用优化算法规划最优调度路径。

1.3 研究概述

本研究进行了以下工作：

1. 在任务一中，本研究基于航道挖掘的勘探图数据，系统开展了三维建模与高程分析：首先，通过数据驱动的方式解析了勘探图中的关键图层信息，重点提取了边坡设计所需的坡比参数。以航道线高程-7.61m 为基准高度，构建了多层次几何解析模型，通过空间坐标系转换与级联计算，精确推导了边坡线在立体空间中的 Z 轴坐标。该过程实现了从二维设计参数到三维空间形态的关键转化，为河道开挖提供了精确的空间定位基准。为验证模型的科学性，研究开展了多维度可视化分析，通过基于 Delaunay 三角剖分的边坡示意图展示空间结构，二维高程等高线图刻画高程分布规律，形成了完整的空间认知体系。

2. 在任务二中，研究基于桂林外环高速公路 B 段的设计图纸，通过提取边坡横断面数据并建立三维直角坐标系，将二维 CAD 图纸坐标转换为三维空间坐标。通过设定参照点实现精确映射，最终生成离散三维点集。利用 Delaunay 三角剖分对点集进行二维投影与三维

插值，构建连续边坡曲面，并通过三维坡面图与等高线图直观展示建模结果。该模型在保证精度的同时简化了计算流程，为边坡工程分析提供了可靠的空间数据支撑。

3. 在任务三中，我们首先通过读取勘探图的设计，基于立方采集这一基本假设确定了土方体积和回填体积。之后我们设计了基于蚁群算法和粒子群算法的最优调度路径规划算法，设计了一个平衡效率和成本的调度与安排的方案，并以多种形式对方案进行了可视化。

4、基于上述任务，我们从计算机视觉的角度出发，构建了一个仅需数张照片就能生成对应真实三维场景的模型。这一模型对于辅助勘探具有重要的应用价值和科学意义。

2 模型准备

2.1 假设与解释

为了对问题进行适当简化，我们做出以下假设：

- **假设 1:** 本文所提到的挖掘机指的是挖掘机和运输车的组合，默认其为共同移动，共享容量。
→ **解释:** 为了简化问题的处理，我们不考虑挖掘机一直挖掘，运输车反复运输的情形。
- **假设 2:** 每一台挖掘机必须从装卸点出发，当容量达到上限时，或完成挖掘任务时，必须返回出发的装卸点。
→ **解释:** 为了简化问题的处理，我们不考虑挖掘机更换装卸点的情况，在实际任务中，这也有利于将整个挖掘任务分解给若干个施工小队。
- **假设 3:** 装卸点足以承载分配给这一装卸点的挖掘机同时装卸。
→ **解释:** 为了方便时间的计算，我们不考虑挖掘机进入不了装卸点，从而产生等待时间的情形。
- **假设 4:** 每个待挖掘地点只由一辆挖掘机挖掘。
→ **解释:** 为了便于土方挖掘量的计算，我们假设每个待挖掘地点只由一辆挖掘机挖掘，同时，我们对每一个待挖掘地点的分割比较精细，对应的挖方体积也相对较小。因此，多辆挖掘机同时挖掘一个地点也并不必要。
- **假设 5:** 每个待挖掘地点只被访问一次。
→ **解释:** 这一条假设规避了一辆挖掘机的剩余容量小于下一个待挖掘地点的挖方体积，挖掘机先访问一次装到容量上限，之后再访问一次，完成挖掘任务的情形。这一假设是5.4中将问题分解成 CVRP 子问题的重要前提。
- **假设 6:** 不考虑不同的装卸点的挖掘机互相调度的情况。
→ **解释:** 为了简化计算，我们不考虑挖掘机更换服务区域的情形，也不考虑某一台挖掘机完成所在区域任务后被调度到其他区域的情形。实验表明，各个区域完成挖掘任务的时间基本相同，表明该假设产生的影响可以忽略不计。
- **假设 7:** 忽略边坡放坡与土方松散系数。
→ **解释:** 采用 $V_i = A_i \cdot |h_i|$ 的柱体模型，回避复杂三维几何计算，强化模型可操作性。

- **假设 8：**任务二中路段的中心线近似处理为一条直线。

→ **解释：**通过识别勘测图上各横断面对应的桩号，本研究的建模对象确定为桂林外环高速公路 B 段的某一特定边坡。我们通过桂林外环 B 段总体平面图对其几何形态进行了分析。结果表明，在可接受的误差范围内，该路段的中心线可近似处理为一条直线。

2.2 符号说明

表 1: 本文中使用的数学符号说明

符号	解释
V	顶点序列
m_i	顶点 v_i 所在区域的坡比
z_{base}	基准高程
d_i	顶点到内层边坡线的水平投影距离
P	边坡线的统一顶点集合
$Triangle$	Delaunay 三角剖分算法生成的平面三角网格
$P_{ref,2D(i)}$	二维 CAD 截面图参考点
$P_{ref,3D(i)}$	三维基准点
P_{2D}	图纸上横断面上的一点
E_b	平衡优化指数
W_t, w_c	时间权重系数，成本权重系数
T	完成任务所用时间
C	完成任务所费成本

* 部分变量未在此列出，将在各章节中详述。

3 任务一：对河道挖掘方案的三维建模

3.1 概述

任务一要求我们基于提供的勘探图数据，构建河道开挖的三维模型。首先，我们需要分析和提取勘探图上与边坡线相关的信息，定位坡顶线，坡底线和变坡线的二维坐标信息，根据提取出的坡比和俯视图的平面几何关系，利用水平距离差计算出高程差，完成对各边坡线 z 轴高程的计算，接着利用 Delaunay 三角剖分生成边坡表面，绘制连续曲面，直观呈现地形起伏。

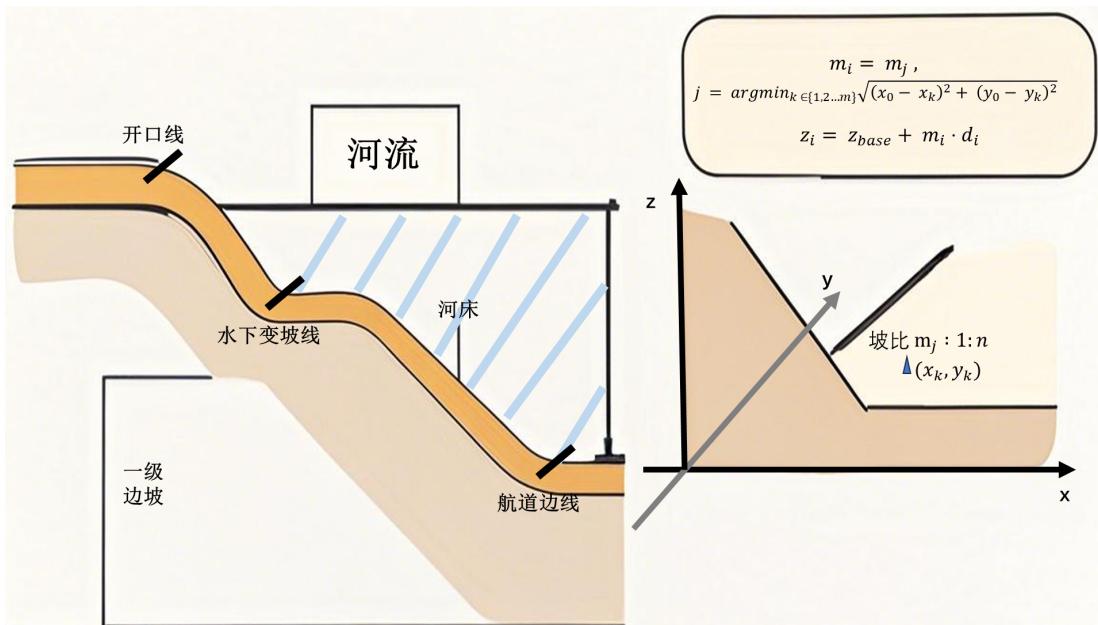


图 1: 任务一示意图

3.2 河道边坡高程计算模型

3.2.1 数据提取和分析

我们通过 CAD 软件对提供的 dxf 文件进行分析，识别出以下与边坡高程计算相关的关键要素，如图 1 左侧所示：

- 航道边线：作为航道边坡的坡底线，用于确定边坡起点的空间定位。航道边线的高程已知， $z = -7.61m$ 。
- 水下变坡线：表征水下地形坡度突变的分界曲线，其两侧地形坡度存在显著差异，需分别计算两侧高程。
- 压顶线：位于边坡顶部的防护结构边界线，辅助界定边坡陆域起点的空间位置。
- 开口线：航道开挖工程在地表或水面的最外侧边界线，直接作为边坡陆域起点的定位基准。
- 坡比标注文本：标注边坡陡峭程度的核心参数，定义为垂直高度与水平投影距离的比值（通常表示为 $1:n$ ），是边坡高程计算的关键输入信息。

由于曲线难以直接用坐标表示，上述所有曲线通过有序顶点集合表示为分段线段：将曲线离散为顶点序列 $V = \{v_1, v_2, \dots, v_n\}$ ，相邻顶点连线形成 $n - 1$ 条线段逼近原曲线，顶点序保留走向与拓扑关系。每个文字标注均对应图纸中的唯一插入坐标，用于文字信息与标注对象的空间关联匹配。

3.2.2 模型建立

分块区域并确定坡比。如图 1 右侧所示，为了将文字标注信息对应边坡区域相关联，我们采用了最近邻搜索，具体来说，针对每条由有序顶点集合表示的边坡线段（如开口线、

压顶线分段线段），对于某一个顶点 v_i , $i = 1, 2 \dots n$, 顶点坐标为 (x_i, y_i) , 做两条辅助线 $y = y_i + \Delta y, y = y_i - \Delta y, \Delta y \rightarrow 0$, 考虑该线段以及两条辅助线与内层边坡线围成的区域，区域内的坡比视为相同值，取值为最接近顶点的文字标注，即对于顶点 $v_i(x_0, y_0)$, 其所在分块区域的坡比 m_i 定义为：

$$m_i = m_j \quad , \quad j = \operatorname{argmin}_{k \in \{1, 2, \dots, m\}} \sqrt{(x_0 - x_k)^2 + (y_0 - y_k)^2} \quad (1)$$

式中： (x_k, y_k) 为第 k 个文字标注的插入坐标， m_k 为其对应的坡比参数， m 为文字标注的总数。通过最小化二维欧氏距离确定最近邻标注点，从而将坡比参数关联至对应分块区域。

计算顶点高程。对于某个顶点 (x_i, y_i) , 该顶点高程等于与内层边坡线（如航道边线、水下变坡线等基准线）的水平距离乘以对应分块区域的坡比，并叠加内层边坡线的已知高程。具体而言，设内层基准线的已知高程为 z_{base} （如航道边线高程 $z = -7.61 \text{ m}$ ），顶点所在分块区域的坡比为 m_i （定义为垂直高度与水平投影距离的比值，即 $m_i = \Delta z / \Delta d$ ），顶点到内层边坡线的水平投影距离为 d_i ，则顶点高程 z_i 可表示为：

$$z_i = z_{\text{base}} + m_i \cdot d_i \quad (2)$$

3.2.3 坡面可视化

在获取所有边坡线顶点的三维坐标 (x_i, y_i, z_i) 后，采用 Delaunay 三角剖分算法对离散顶点进行曲面重构，生成连续的边坡表面。Delaunay 三角剖分通过最大化三角形的最小内角，确保生成的三角网格无重叠且满足空圆特性，能够准确反映地形的起伏特征。具体步骤如下：

- 构建三维顶点集：将开口线、压顶线、水下变坡线、航道边线等边坡线的离散顶点坐标（含计算得到的高程值 z_i ）整合为统一的顶点集合 $P = \{(x_j, y_j, z_j) | j = 1, 2, \dots, N\}$ ，其中 N 为顶点总数。
- 二维投影三角剖分：由于 Delaunay 三角剖分通常在二维平面上进行，首先提取所有顶点的 x, y 坐标，生成平面三角网格 $\text{Triangle} = \{\triangle(v_a, v_b, v_c) | v_a, v_b, v_c \in P\}$ ，每个三角形面片对应边坡表面的一个局部平面。
- 三维曲面构建：对于每个三角形面片 $\triangle(v_a, v_b, v_c)$ ，利用顶点的高程值 z_a, z_b, z_c 定义其空间位置。通过线性插值或双线性插值方法，确保相邻三角形面片在公共边上的连续性，最终形成覆盖整个边坡区域的连续曲面。

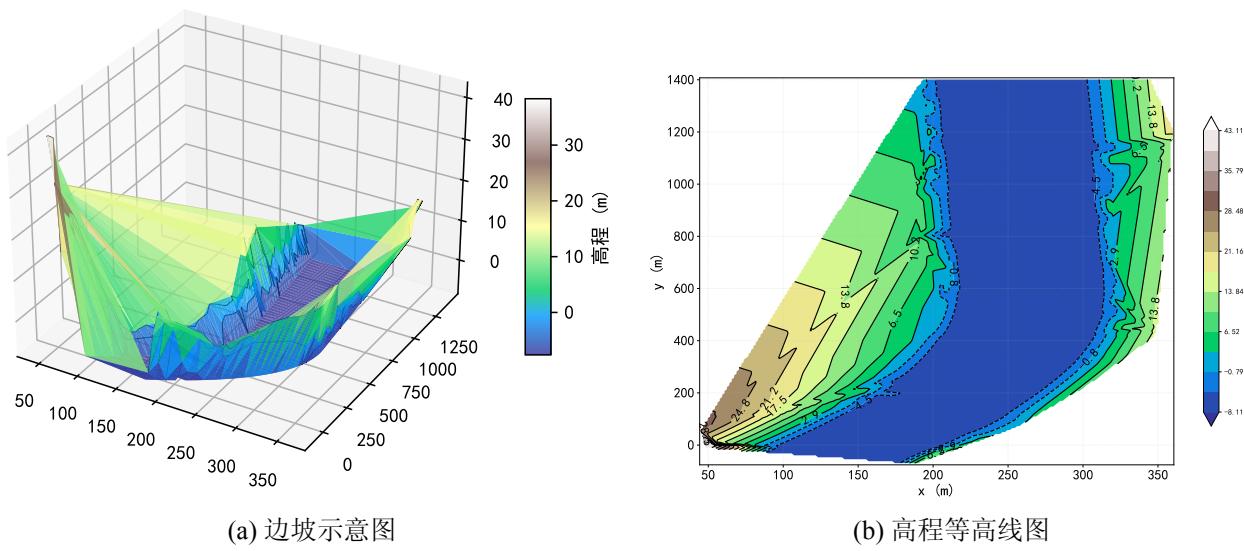


图 2: 河道边坡三维建模可视化

4 任务二：对高速公路边坡修建的三维建模

4.1 概述

高速公路边坡三维建模核心是将 CAD 图纸二维数据精准转化为三维模型，支撑工程分析。针对桂林外环高速 B 段边坡，首先简化模型，基于路段短且关注横向形态的特点，将中心线近似为直线以降低建模复杂度，再进行数据转换，将 CAD 图纸二维坐标系数据，通过参照点与工程参数，映射到三维空间直角坐标系；最后进行模型构建，对转换后的散乱点集，采用 Delaunay 三角剖分等空间插值方法生成连续曲面，并可视化呈现坡面与等高线图，满足工程应用需求。

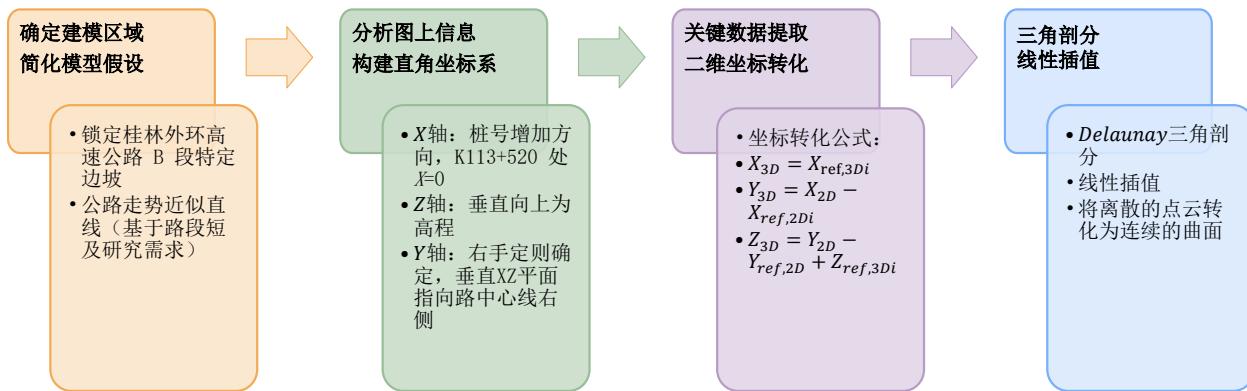


图 3: 任务二工作流程

4.2 高速公路边坡高程计算模型

4.2.1 空间坐标系构建

为了定量描述边坡的空间形态，我们建立了一个三维空间直角坐标系 O_{XYZ} 。坐标系的定义遵循土木工程领域的通用惯例，并与实际工程测量相对应：

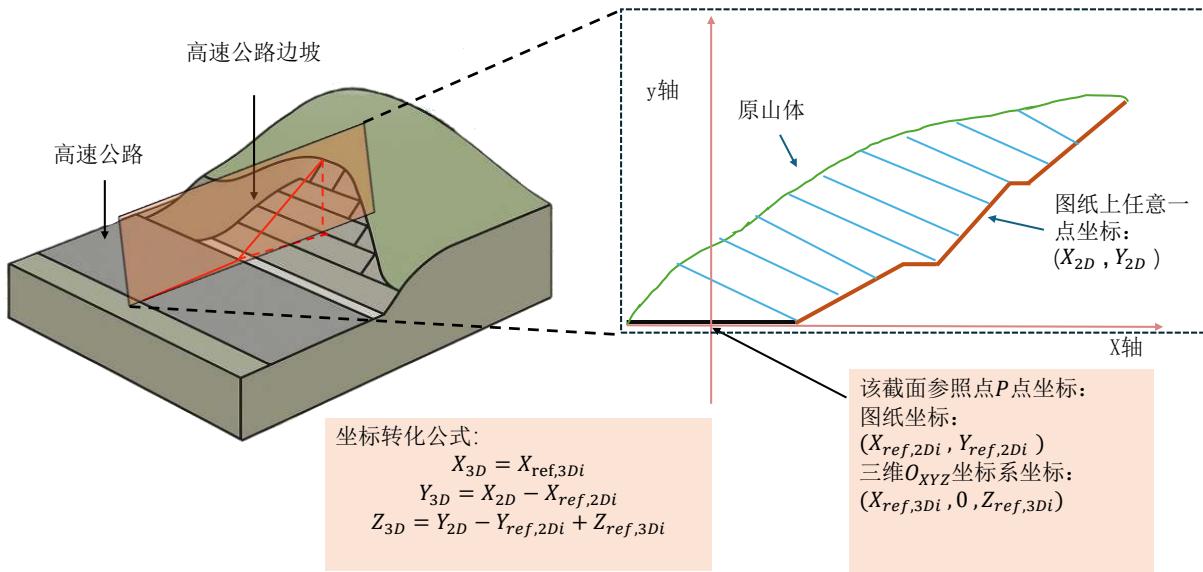


图 4: 任务二示意图

- X 轴: 沿高速公路中心线的走向 (即桩号增加的方向) 为 X 轴正方向。选取目标路段的起始桩号 $K113 + 520$ 对应的中心线位置作为 $X = 0$ 处, 即 $X = 0$ 对应于桩号 $K113 + 520$ 处。后续桩号对应的 X 坐标按实际里程线性增加。
- Z 轴: 以垂直于水平面向上为 Z 轴正方向, 代表高程。
- Y 轴: 根据右手定则确定 Y 轴方向, 垂直于 XZ 平面, 指向道路中心线的右侧

4.2.2 数据提取与坐标转换

原始数据来源于 CAD 工程设计图纸。为便于处理, 将数据存储为 JSON 格式, 便于后续解析和处理。JSON 文件中存储的坐标是基于 CAD 图纸的二维绘图坐标系, 记为 (x_{2D}, y_{2D}) 。为了构建三维模型, 需要将这些二维坐标点映射转换到 4.1.3 节定义的全局三维空间坐标系 O_{XYZ} 中, 记为 (X_{3D}, Y_{3D}, Z_{3D}) 。转换过程依赖于图纸中的“参照点” (截面图中轴线与路面的交点) 和已知的实际工程参数。

在图纸上存在多张截面图, 而对于每一个横断面, 存在唯一的参照点 $P_{ref,i}$ 。

不妨设在图纸中参考点坐标:

$$P_{ref,2D(i)} = (x_{ref,2D(i)}, y_{ref,2D(i)}) \quad (3)$$

这些参照点对应着三维空间中具有已知坐标的基准点

$$P_{\text{ref},3D(i)} = (X_{\text{ref},3D(i)}, 0, Z_{\text{ref},3D(i)}) \quad (4)$$

其中， $X_{\text{ref},3D(i)}$ 由该断面对应的桩号确定， $Y_{\text{ref},3D(i)}$ 为 0， $Z_{\text{ref},3D(i)}$ 为该参照点的实际高程（由图上的文字标注 H_s 得到）。

对于图纸上任意一个属于第 i 个横断面的点 $P_{2D} = (x_{2D}, y_{2D})$ ，其对应的三维坐标 (X_{3D}, Y_{3D}, Z_{3D}) 可通过以下映射关系计算得出：

- X 坐标：该点的三维 X 坐标由其所属的横断面决定，即等于该断面参考点的三维 X 坐标：

$$X_{3D} = X_{\text{ref},3D(i)} \quad (5)$$

- Y 坐标：该点的三维 Y 坐标由其在二维图纸上相对于参考点的水平偏移量决定：

$$Y_{3D} = x_{2D} - x_{\text{ref},2D(i)} \quad (6)$$

- Z 坐标：该点的三维 Z 坐标由其在二维图纸上相对于参考点的垂直偏移量，加上参考点的实际高程决定：

$$Z_{3D} = (y_{2D} - y_{\text{ref},2D(i)}) + Z_{\text{ref},3D(i)} \quad (7)$$

通过对所有相关的二维截面图上的点执行上述坐标转换，我们得到边坡的顶点集合 $P = \{(x_j, y_j, z_j) \mid j = 1, 2, \dots, M\}$ ，其中 M 为转换后的点的总数。

4.2.3 坡面可视化

与 3.2.3 一样，采用 Delaunay 三角剖分对点集 P 进行三维曲面构建，最终形成覆盖整个边坡区域的连续曲面，如图 5a，以及等高线图，如图 5b。

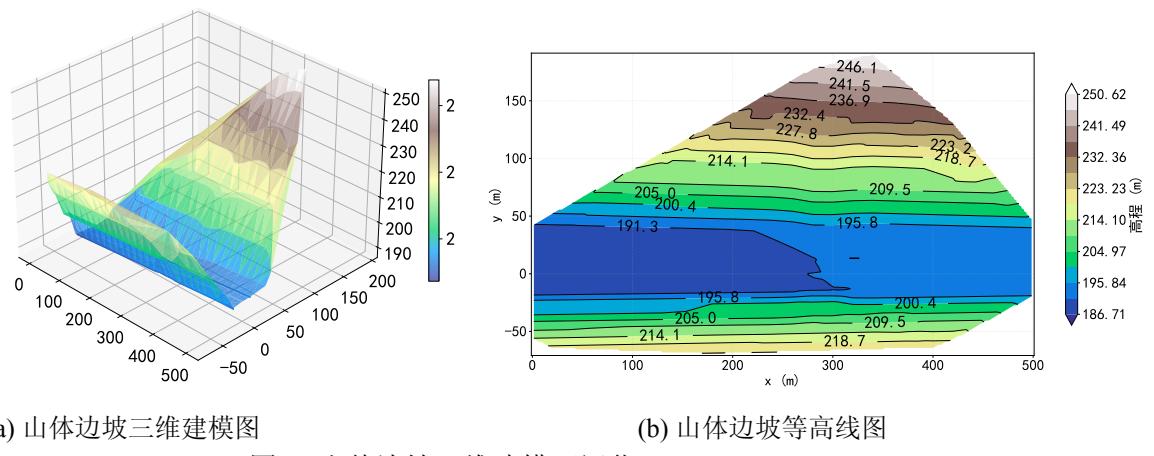


图 5: 山体边坡三维建模可视化

5 任务三：挖方体积计算与最优调度路径规划

5.1 概述

在本章中，我们首先计算了两个任务中的粗略体积，之后通过二重积分得到的体积与立方体积作差得到了回填体积。之后我们构建了一个基于双层优化方法的挖方方案调度规划模型。具体而言，我们首先将该优化问题标准化，然后将其拆解为 CVRP 问题和聚类参数和车辆数选择这两个子问题。对于 CVRP 问题，我们使用蚁群算法求解，对于聚类参数和车辆数选择问题，我们使用粒子群算法求解。

在本章中，如表3所示，我们从多方来源确定了若干常数的值。

表 2: 挖掘土方作业有关数据

符号	解释	单位	数值	来源
c_h	工作所需人员时薪	元	30	职友集 ^[2]
V_h	卡特彼勒 CAT®305.5 液压挖掘机每小时挖掘体积	m^3/h	66	中国路面机械网 ^[3]
c_f	同型号液压挖掘机工作燃料单价	元/升	6.55	国家发改委 ^[4]
V_c	卡特彼勒 CAT®770G 车斗容量	m^3	120	中国路面机械网 ^[5]
n_h	同型号液压挖掘机工作所需人员数量	/	3	中国路面机械网 ^[3]
c_d	同型号液压挖掘机工作租金	元/小时	60	中国路面机械网 ^[3]
n_f	同型号液压挖掘机工作燃料消耗	升/小时	3.48	中国路面机械网 ^[3]
v_d	同型号液压挖掘机行进速度	米/小时	3×10^5	中国路面机械网 ^[3]

5.2 挖方体积计算

针对任务一的挖方体积计算，我们提取了各个开挖船位小方格四个顶点的二维坐标，通过文字标注的插入坐标与各船位进行对应，每个开挖船位小方格为平面四边形，其体积计算公式如下：

1. 四边形面积计算（鞋带公式）：

$$A_i = \frac{1}{2} \left| \sum_{j=1}^4 (x_j y_{j+1} - x_{j+1} y_j) \right| \quad (8)$$

其中：

- (x_j, y_j) 为四边形顶点坐标， $j = 1, 2, 3, 4$ 按顺时针或逆时针排列
- $(x_5, y_5) = (x_1, y_1)$ 构成闭合多边形

2. 挖方高度计算。挖方高度按水位-基准面关系分两段计算：

- 高水位段 ($z_{\text{water},i} > z_{\text{base}}$)。穿透水层至基准面下，附加硬底差： $h_i = \underbrace{(z_{\text{base}} - z_{\text{water},i})}_{\text{水层穿透}} + \underbrace{d_{\text{hard},i}}_{\text{底层处理}}$

• 低水位段 ($z_{\text{water},i} \leq z_{\text{base}}$)。直接挖至基准面，扣除硬底差： $h_i = \underbrace{(z_{\text{base}} - z_{\text{water},i})}_{\text{基面高差}} - \underbrace{d_{\text{hard},i}}_{\text{防超挖}}$

$$h_i = \begin{cases} (z_{\text{base}} - z_{\text{water},i}) + d_{\text{hard},i}, & z_{\text{water},i} > z_{\text{base}} \\ (z_{\text{base}} - z_{\text{water},i}) - d_{\text{hard},i}, & \text{otherwise} \end{cases} \quad (9)$$

其中：

- $z_{\text{base}} = 7.61$ 为基准高程
- $z_{\text{water},i}$ 为第 i 个船位标注水位
- $d_{\text{hard},i}$ 为第 i 个船位硬底差

3. 下挖与回填体积分项计算：

根据挖方高度符号，分别计算各船位的挖方体积与回填体积：

$$V_{\text{digging},i} = \begin{cases} A_i \cdot h_i, & h_i \geq 0 \\ 0, & h_i < 0 \end{cases} \quad (10)$$

$$V_{\text{backfill},i} = \begin{cases} 0, & h_i \geq 0 \\ A_i \cdot |h_i|, & h_i < 0 \end{cases} \quad (11)$$

4. 总挖方体积计算：

累加所有船位体积，得到总挖方体积 V_{digging} 和总回填体积 V_{backfill} ：

$$V_{\text{digging}} = \sum_{i=1}^n V_{\text{digging},i} \quad (12)$$

$$V_{\text{backfill}} = \sum_{i=1}^n V_{\text{backfill},i} \quad (13)$$

针对第二问的挖方体积计算，与第一问相同，采用对开挖区域的离散化求和来近似挖方体积。首先，在工程区域的 XY 水平投影面上定义一个规则的矩形网格。设网格在 X 和 Y 方向的步长分别为 Δx 和 Δy ，则每个网格单元的底面积为 $\Delta A = \Delta x \cdot \Delta y$ 。利用前述插值方法，可以获得每个网格节点 (X_i, Y_j) 处的挖方深度 h_{ij} ，该深度定义为原始地表高程与设计高程之差。当 $h_{ij} > 0$ 时，表示该处需要开挖。假定每个网格单元内的挖方深度可由其中心的深度代表，则该单元对应的微元挖方体积 V_{ij} 近似为 $h_{ij} \cdot \Delta A$ 。通过对所有位于有效插值区域内且 $h_{ij} > 0$ 的网格单元的微元体积进行求和，即可得到项目区域的总挖方体积 $V = \sum_i \sum_j V_{ij}$ 。

此方法的精度与网格划分的精细程度相关，网格越密，估算结果越接近真实体积。而在实际工程中，往往采用先粗挖后细挖的策略，存在挖方过度再回填的过程。所以我们通过调节网格划分的精细程度，计算二者差值，模拟计算实际挖掘中的挖掘过度的回填土方量。

表 3: 体积计算 (单位: 立方米)

挖掘阶段	河道开挖	高速公路修建
粗挖阶段	435271.156	387780.518
回填阶段	4009.356	3420.630

5.3 调度方案优化问题的标准化

在求解最优调度路径之前，我们首先需要对该问题进行标准的数学形式描述。

优化目标：我们以时间短，开销少作为优化目标。定义平衡优化指数 E_b :

$$E_b = w_t \times T + w_c \times C \quad (14)$$

其中， w_t 为时间权重系数， w_c 为成本权重系数。其中， $w_t + w_c = 1$ ，通过调整权重系数，可以控制模型更倾向于优化时间还是开销。

优化变量：本文综合考虑了以下变量，装卸点的数量 n_l ，装卸点的位置 $(\mathbf{x}_{n_l}, \mathbf{y}_{n_l})$ ，挖掘机的数量 n_d ，挖掘机的行进路径 \mathbf{I} 。其中 \mathbf{x}_{n_l} , \mathbf{y}_{n_l} 是向量形式，代表所有装卸点的横纵坐标， \mathbf{I} 是一个 $n_d \times N_s$ 的矩阵，其中 N_s 是所有挖掘机中经过的挖掘点数量的最大值， \mathbf{I} 表示的含义是每一台挖掘机依次经过的挖掘点，每个挖掘点有一个唯一索引。对于每一个方案 i ，其优化变量可以描述为：

$$\mathbf{X}_i = [n_{l_i}, (\mathbf{x}_{n_l}, \mathbf{y}_{n_l})_i, n_{d_i}, \mathbf{I}_i] \quad (15)$$

约束条件：在这里，我们先不给出全局的约束条件，我们将会在 5.4 节问题分解中详细讨论。

标准格式：基于上述列出的优化目标和变量，我们可以写出本问题的标准优化形式。

$$\begin{aligned} & \min E_b(\mathbf{X}) \\ & \text{s.t. constraints} \end{aligned} \quad (16)$$

$E_b(\mathbf{X})$ 指的是对一个方案的优化变量计算其平衡优化指数。

$$E_b(\mathbf{X}) = w_t \times T(\mathbf{X}) + w_c \times C(\mathbf{X}) \quad (17)$$

$$C(\mathbf{X}) = n_d \times T(\mathbf{X}) \times c_d + n_d \times n_h \times T(\mathbf{X}) \times c_h + n_d \times n_f \times T(\mathbf{X}) \times c_f \quad (18)$$

简单来说，总成本等于挖掘机的租赁成本 + 用人成本 + 燃料成本，式 18 给出的是一个

粗略的表达形式，实际的成本计算由仿真过程逐渐累加得到。 $T(\mathbf{X})$ 由仿真过程得到，定义为所有车结束挖掘任务并回到装卸点对应的结束时间戳减去开始时间戳。

5.4 问题分解

我们分析得到，如果同时考虑上述所有的变量，将会使得求解的复杂度极高，时间成本将会难以接受。同时，变量之间也存在着复杂的耦合关系，也不利于中间变量的计算。我们发现，基于我们在2.1节提出的假设，在给定一个装卸点以及其“服务”的待挖掘地点的前提下，这个问题恰好是一个标准的 CVRP 问题（带容量约束的车辆路径问题）。

CVRP 问题最早由 Dantzig 和 Ramser 提出^[6]，可形式化定义如下：给定一个配送中心（仓库）、一组客户以及一个车队。每个客户都有特定的货物需求，每辆车有最大载货容量限制。目标是在满足所有客户需求且不违反车辆容量约束的前提下，设计一系列从配送中心出发并最终返回的车辆行驶路线，使得总运输成本（如行驶距离、时间或费用）最小。具体来说，给定一个图：

$$G(V, E) \quad V = \{0, 1, 2, \dots, n\} \quad (19)$$

其中， V 表示顶点集合，0 为配送中心，其余为客户端（本问题中为待挖掘地点）， E 表示边集合，每条边 $(i, j) \in E$ 对应一个非负权 c_{ij} （成本）。每辆车的最大载货量为 M ，客户 i 的需求为 q_i 。目标是在满足约束条件的情况下，返回一个点集 V' ，包含每辆车经过的顶点索引，使得方案总成本（包括时间和其他成本）最小。

在假定每一个分解出来的子问题（CVRP 问题）都可以解决的前提下，我们的优化问题变为确定装卸点的位置选择及每一个装卸点的服务对象（即有哪些待挖掘地点分配给这个装卸点对应的队伍）与分配车辆数量的问题。不难想到，这个问题可行解的数量为：

$$\sum_{l_{min}}^{l_{max}} \sum_{d=d_{min}}^{d_{max}} \frac{n_s!}{n_1! n_2! \dots n_l!} \times \frac{n_d!}{n_1! n_2! \dots n_l!} \quad (20)$$

其中， n_s 代表挖掘点的总数， n_l 代表装卸点的总数， n_d 代表挖掘机的总数， l_{min}, l_{max} 分别代表装卸点数量的上下限， d_{min}, d_{max} 代表挖掘机数量的上下限。这是一个难以接受的搜索空间。

我们想到，现实工程作业中，任务点一定会充分考虑空间位置关系，一个工作小队被分配到的任务一定是空间距离较近的。因此，我们采用 K-means 算法来分配装卸点位置和每一个装卸点的“服务”区域。然而，在指定全局随机种子后，K-means 算法对于同一个类别数和数据分布情况产生的聚类结果就是固定的。也就是说，当指定了装卸点数量和全局的挖掘点分布时，装卸点的位置就是确定的。这就极大地忽略了装卸点位置对于模型结果的影响，因此我们在执行 K-means 算法时，引入了一个挖高系数作用在挖掘高度上。

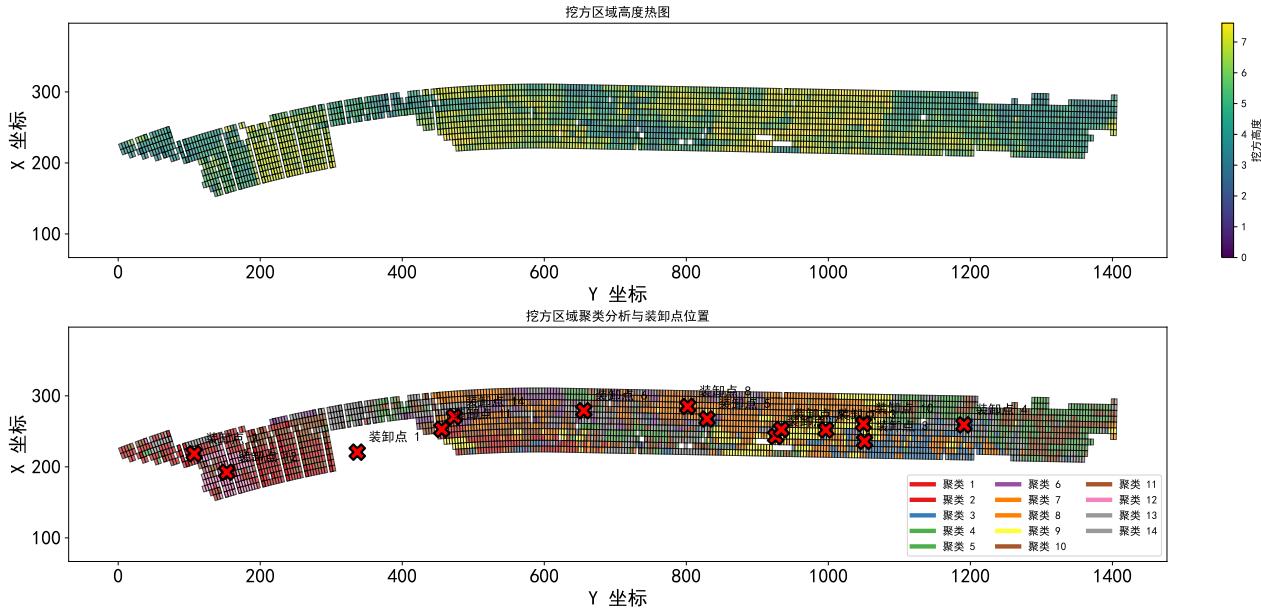


图 6: 在给定的聚类数量和挖高系数下的装卸点位置，及每个装卸点分配的任务。

$$\mathbf{X}_{\text{means}} = [\mathbf{X}, \mathbf{Y}, \mathbf{H}]$$

$$\mathbf{X}_{\text{scale}} = \text{Standardization}(\mathbf{X}_{\text{means}})$$

$$\mathbf{X}'_{\text{scale}} = [\mathbf{X}_{\text{scale}}, \mathbf{Y}_{\text{scale}}, e_{\text{height}} \times \mathbf{H}_{\text{scale}}] \quad (21)$$

$$(\mathbf{x}_{n_l}, \mathbf{y}_{n_l}) = K\text{means}(\mathbf{X}'_{\text{scale}})$$

挖高系数的物理意义在于，当 $e_{\text{height}} \rightarrow 0$ 时，代表聚类算法完全将空间位置作为分类依据运行；当 $e_{\text{height}} \rightarrow \infty$ 时，代表聚类算法完全将单个挖掘点的挖掘时间作为分类依据运行。考虑挖掘机分配问题，我们则简单地使用按照挖掘任务的总体积，按比例进行分配，这样做的好处是，我们的优化变量就无需考虑挖掘机的分配方案，而只需要讨论挖掘机的数量，在模型分析中我们会进一步讨论这种分配方式的合理性。这样我们就把式20的可行解数量，大幅减少到了：

$$(l_{\max} - l_{\min}) \times (d_{\max} - d_{\min}) \times \text{Num}(e_{\text{height}}) \quad (22)$$

其中 $\text{Num}(e_{\text{height}})$ 表示在给定分辨率下浮点数 e_{height} 的可行数量，分辨率指的是 e_{height} 变化时的最小步长。

经过上述问题分解后，我们可以改写式15为

$$\mathbf{X}_i = [n_{l_i}, e_{\text{height}}, n_{d_i}, \mathbf{I}_i] \quad (23)$$

进一步修改式16，给出优化问题的最终形式。

$$\begin{aligned}
 & \min E_b(\mathbf{X}) \\
 & \text{s.t.} \\
 & a. \sum_{k=1}^{n'_d} \sum_{j=1}^{n'_l} x_{0jk} = \sum_{k=1}^{n'_d} \sum_{j=1}^{n'_l} x_{j0k} = n'_d \\
 & b. \sum_{i=1}^{n'_l} x_{ijk} = \sum_{i=1}^{n'_l} x_{jik} \quad \forall j = 1, 2, \dots, N, \forall k \in m \\
 & c. \sum_{i=0}^{n'_l} \sum_{j=0}^{n'_l} x_{ijk} m_j \leq V_c \quad \forall k \in m \\
 & d. \sum_{k=1}^{n'_d} \sum_{i=0}^{n'_l} x_{ijk} = 1 \quad \forall j = 1, 2, \dots, N \\
 & e. n'_d \geq \left\lceil \frac{\sum_{j=1}^{n'_l} m_j}{V_c} \right\rceil \quad K = 1, 2, \dots, N
 \end{aligned} \tag{24}$$

其中， n'_d 和 n'_l 分配到这一个装卸点的挖掘机数量和挖掘点数量：

$$x_{ijk} = \begin{cases} 1, & \text{表示挖掘机 } k \text{ 从挖掘点 } i \text{ 行驶到挖掘点 } j \\ 0, & \text{其他} \end{cases} \tag{25}$$

约束依次代表：

- a. 配送中心约束：所有车辆均由配送中心出发，完成所有的配送任务后返回配送中心。
- b. 客户点流量平衡：进出车辆数相等。
- c. 重量约束：每辆车的装载量不能超过其最大载重量限制。
- d. 客户点服务约束：每个挖掘点被访问 1 次。
- e. 所需车辆数约束

总而言之，我们将全局优化问题，拆解成先通过控制挖高系数和聚类数控制装卸点的位置和被分配的任务，之后对于每一个分配点对应的子任务，将其转化为 CVRP 问题求解。

5.5 基于蚁群算法的 CVRP 问题求解

CVRP 问题是一个公认的 NP 难问题^[7]，使用暴力求解的方式将会使得时间成本难以接受。因此我们使用蚁群算法进行求解，蚁群算法（Ant Colony Optimization, ACO）是一种受自然界中蚂蚁群体觅食行为启发的元启发式优化算法，由 Dorigo 等人于 1991 年首次提出^[8]。该算法通过模拟蚂蚁在路径搜索过程中利用信息素（pheromone）进行间接通信的机制，构建了一种基于种群的分布式概率搜索策略。其核心原理在于，蚂蚁在移动过程中释

放信息素以标记路径，后续蚂蚁倾向于选择信息素浓度较高的路径，从而形成正反馈机制，促使群体智能逐渐收敛于最优解。算法通过结合启发式信息（如路径长度）和信息素浓度，以概率方式迭代更新候选解，平衡了全局探索与局部开发能力。

Algorithm 1 蚁群算法求解CVRP问题伪代码

```

输入： 车辆数量、客户数量、车辆容量等参数
输出： 最优路径、最优时间、最优花费、最优加权值
初始化信息素矩阵 初始最佳时间、最佳花费、最佳加权值和最佳路径
for 迭代次数 in 1 到最大迭代次数 do
    初始化蚂蚁群
    for 每只蚂蚁 in 蚂蚁群 do
        所有车辆从仓库出发
        while 未访问所有客户 do
            当前节点 = 当前路径的末尾节点 找出未访问且可载重的客户
            if 无可用客户 then
                切换车辆（如有）或返回仓库
            else
                计算转移概率 随机选择下一个节点 更新路径和已访问客户
            end if
        end while
        计算总时间、总花费和总加权值 更新全局最佳解
    end for
    信息素挥发 根据蚂蚁路径更新信息素
end for
返回： 最优路径、最优时间、最优花费、最优加权值

```

算法 1 给出了蚁群算法求解 CVRP 问题的伪代码，简单来说，算法包括以下的关键步骤。

- **路径构建概率：** 蚂蚁 k 在节点 i 选择节点 j 的概率由信息素 τ_{ij} 和启发式信息 η_{ij} 共同决定：

$$p_{ij}^k = \frac{[\tau_{ij}]^\alpha \cdot [\eta_{ij}]^\beta}{\sum_{l \in \mathcal{N}_i^k} [\tau_{il}]^\alpha \cdot [\eta_{il}]^\beta}, \quad j \in \mathcal{N}_i^k \quad (26)$$

其中 α 控制信息素权重， β 控制启发式信息权重， \mathcal{N}_i^k 为可行节点集。

- **信息素更新：** 包含挥发和增强两个过程：

$$\tau_{ij} \leftarrow (1 - \rho)\tau_{ij} + \Delta\tau_{ij} \quad (27)$$

$$\Delta\tau_{ij} = \sum_{k=1}^m \Delta\tau_{ij}^k, \quad \Delta\tau_{ij}^k = \begin{cases} Q/L_k & \text{若边}(i, j) \in \text{路径 } k \\ 0 & \text{否则} \end{cases} \quad (28)$$

其中 $\rho \in (0, 1)$ 为挥发系数， Q 为常数， L_k 为优化目标。

在这里我们不再赘述蚁群算法这一已经非常完善的算法，而是强调一些实现上的细节。

在更新当前装卸点对应 CVRP 的总时间时，我们选取当前车队耗费时间最长的总用时。蚁群算法的优化目标为最小化加权值，时间和花费的权重与式17中的相同。在构建信息素更新策略时，我们采用精英蚂蚁策略，即每次只有目标值最优的个体释放信息素。代码的具体实现请详见附录8或查看我们公开的代码仓库。

5.6 基于粒子群算法的全局优化问题求解

尽管在5.4节问题分解中，我们将确定装卸点位置和数量以及挖掘机总数问题的可行解数量简化成了式22的形式。但由于每一个解都要进行装卸点数量次的蚁群算法求解，遍历这一暴力方法在时间上依旧是不经济的。对此我们采用粒子群算法这一启发式算法。

粒子群算法（Particle Swarm Optimization, PSO）是一种基于群体智能的元启发式优化算法，由 Kennedy 和 Eberhart 于 1995 年提出^[9]，其灵感来源于鸟群或鱼群的社会行为协作机制。该算法将搜索空间中的每个候选解视为一个“粒子”，粒子通过跟踪个体历史最优位置（pbest）和群体全局最优位置（gbest）动态调整自身速度与方向，模拟生物群体在觅食过程中的信息共享与协作行为。

Algorithm 2 粒子群优化算法伪代码

```
初始化粒子群，每个粒子的初始位置、速度和个体最优位置、适应度  
初始化全局最优位置和适应度  
for 每次迭代 do  
    更新算法参数（如惯性权重等）  
    for 每个粒子 do  
        生成两个随机向量 计算认知项和社交项 更新粒子速度 更新粒子位  
        置 对整数维度取整 边界处理  
    end for  
    并行评估所有粒子的适应度  
    for 每个粒子 do  
        根据适应度更新粒子个体最优位置和适应度  
        根据适应度更新全局最优位置和适应度  
    end for  
    记录本次迭代全局最优适应度  
    剪枝表现不佳的粒子 检查早停条件，若满足则提前停止  
end for  
输出最优位置和最优适应度
```

算法 2 给出粒子群算法求解全局优化问题的伪代码，简单来说，算法包括以下的关键步骤。

- **速度更新：**粒子 i 在维度 d 的速度更新由惯性项、个体认知项和群体协作项组成：

$$v_{id}^{t+1} = \omega v_{id}^t + c_1 r_1 (pbest_{id} - x_{id}^t) + c_2 r_2 (gbest_d - x_{id}^t) \quad (29)$$

其中 ω 为惯性权重， c_1, c_2 为加速常数， $r_1, r_2 \sim U(0, 1)$ 。

- **位置更新：**基于速度调整粒子位置：

$$x_{id}^{t+1} = x_{id}^t + v_{id}^{t+1} \quad (30)$$

- **最优解更新：**比较适应度值并更新个体和群体最优：

$$pbest_i = \begin{cases} x_i^{t+1} & \text{若 } f(x_i^{t+1}) < f(pbest_i) \\ pbest_i & \text{否则} \end{cases} \quad (31)$$

$$gbest = \arg \min_{pbest_i} f(pbest_i) \quad (32)$$

在这里我们也不再赘述粒子群算法这一已经非常完善的算法，而是强调一些实现上的细节。

在算法运行的过程中，我们对 w 实行二次递减策略，驱使模型在初期保持探索，而到后期则专注于精细搜索。我们对 c_1 实行逐渐减小的策略，对 c_2 实行逐渐增大的策略，使模型初期注重个体探索 (c_1 大)，后期注重群体收敛 (c_2 大)。同时，为了加快算法运行速度，我们并行计算每一个粒子的适应度，适应度的计算由上一节 CVRP 问题的求解算法给出，时间为每一个装卸点对应任务时间的最大值，成本为累加。

5.7 实验结果与可视化

由于篇幅限制，我们这里只展示部分装卸点对应的最优路径，完整可视化请详见附录8。

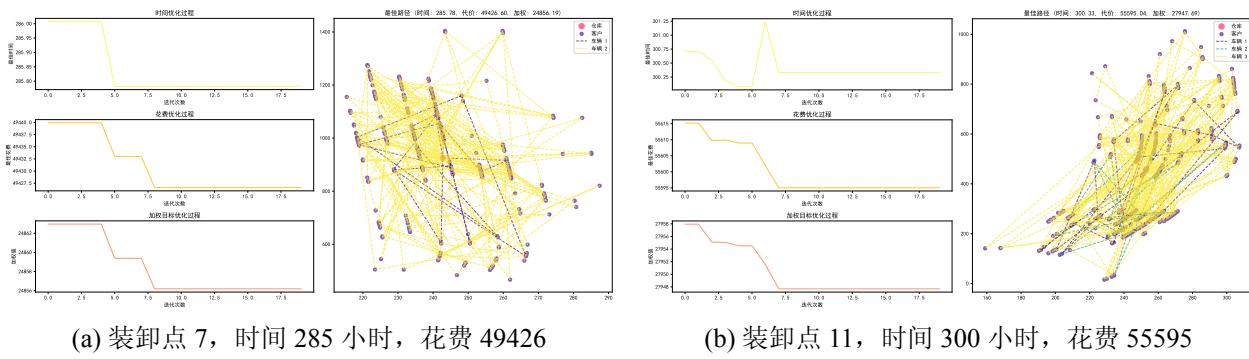


图 7: 对河道开挖部分装卸点的调度路径可视化

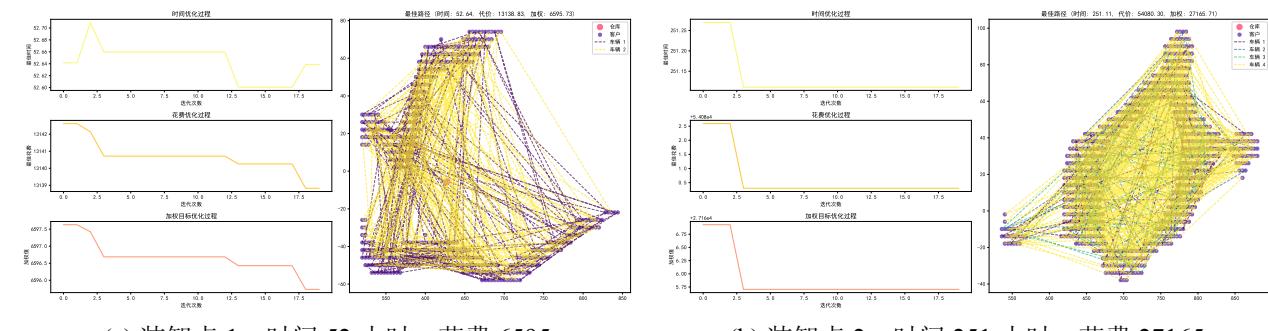


图 8: 对边坡修建部分装卸点的调度路径可视化

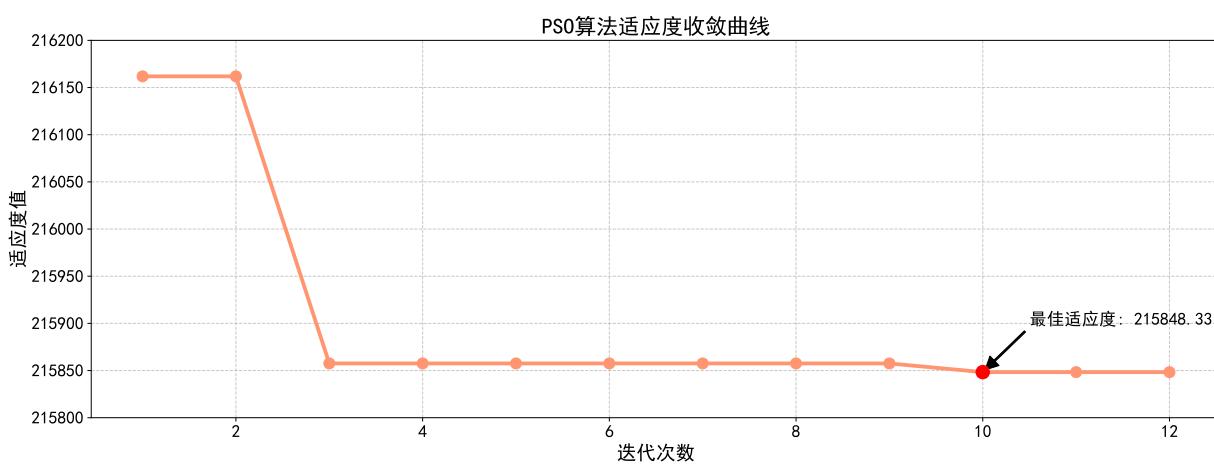


图 9: 粒子群算法的收敛曲线

表 4: 两项任务的求解结果

任务	时间 (小时)	总成本	挖掘机数	装卸点数	挖高系数
河道开挖	447.66	471,984.45	27	14	5.39
边坡修建	259.36	210,420.63	25	10	3.60

表5展示了两项任务的完整结果，值得注意的是，这里的时间指的是连续的小时数。

6 拓展：从图片到三维模型

基于上述的任务，可以发现，准确的勘探图是建模合适方案的重要条件。传统的勘探方法时间和金钱成本都十分高昂，是否有一个合适的方法能够迅速低成本地获取三维数据呢？

计算机视觉为解决这个问题提供了新的视角，如图所示，RGB 数据为三维数据的获取提供了 xy 轴上的数据，高效的深度估计算法，如 Metric3D^[10], DepthAnything^[11-12], Depth-pro^[13] 等，可以获取 z 轴上深度（像素到相机平面上的距离）的信息。理论上我们只需要将每一个 xy 轴的像素向 z 轴偏移深度数值，就可以得到这张图片对应的三维数据了。

然而，这样的方式并不可行，原因在于，图片和估计的深度往往都是相对距离或相对深度（部分单目深度估计模型是可以估计绝对深度的，如 Depthpro^[13]），我们并不能获取到相对距离关系和相对深度关系与真实世界之间的关系。正如在地图上，只有知道比例尺才能计算真实距离一样，在三维建模这一任务中，我们也需要为相对数据找到一个比例尺——相机参数。在获取到这一关键信息之后，我们就能从若干张图片，甚至是一张图片中获取完整的三维信息。



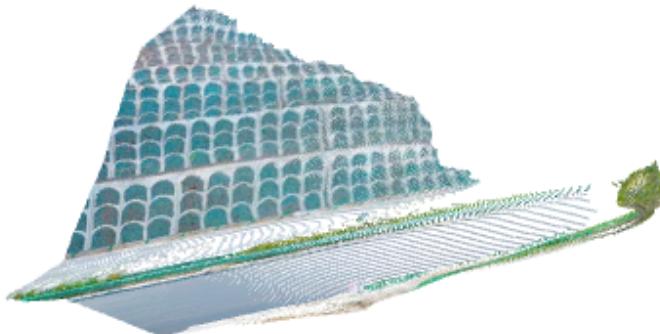
(a) 真实图片



(b) 生成的三维模型（和原图保持相同角度）



(c) 真实图片



(d) 生成的三维模型（旋转一个角度）

图 10: VGGT^[1]生成的三维模型，包含不同角度的，生成的三维模型可以自由旋转

VGGT^[1] (Visual Geometry Grounded Transformer) 是这一领域的前沿工作，其能够直接从单视角、多视角甚至数百视角的图像输入中，端到端预测场景的所有关键三维属性（包括相机内外参、深度图、点图及三维点轨迹），且无需依赖传统几何后处理优化（如集束调整）。与传统视觉几何方法不同，VGGT^[1]摒弃了针对三维任务的特殊网络设计，如图所示，VGGT 仅通过交替式帧内与全局注意力机制，在公开三维标注数据上训练即实现几何理解能力。

我们复现了 VGGT^[1]的工作，并使用了多个高速公路的真实图片进行了测试。如图10所示，模型可以在非常短的时间内，完成三维模型的重建。这样的三维重建模型对于快速获取勘探数据具有重要意义，极大地节约了勘探的时间和金钱成本。然而，VGGT^[1]也有一定的不足，其对鱼眼/全景图像支持较差，大旋转的图像性能下降明显，非刚性大变形场景仍然容易失败。我们还将一个视频作为输入，输出一个更大的三维模型，具体请见附录8。

7 总结

本文针对土方挖掘调度难题，构建了从三维建模（河道与公路边坡）、土方量计算到优化调度（结合蚁群与粒子群算法）的完整方案。其优点在于综合创新地结合了三维重建与先进优化算法、有效的问题分解策略、考虑空间与工作量的参数化聚类方法、兼顾时间与成本的多目标优化以及清晰的可视化结果。然而，模型也存在简化假设（如忽略边坡精确计算、土方松散系数、部分几何线性近似及 CVRP 简化）带来的局限性。未来改进方向包括：进一步细化模型假设与计算精度，提升算法性能与效率，融合计算机视觉等新型数据采集技术以增强数据获取能力，发展动态实时调度功能，并进行不确定性分析以提高方案鲁棒性。总体而言，该模型框架具有显著的创新性和实用价值，通过上述改进，其在实际工程中的应用前景将更为广阔。

8 附录

参考文献

- [1] WANG J, CHEN M, KARAEV N, et al. Vggt: Visual geometry grounded transformer[J]. arXiv preprint arXiv:2503.11651, 2025.
- [2] 职友集. 挖掘机司机就业前景[EB/OL]. 2025. <https://www.jobui.com/salary/quanguo-wajuejisiji/>.
- [3] 卡特彼勒中国. 卡特彼勒 CAT®305.5 液压挖掘机参数配置[EB/OL]. 2025. <https://zj.lmjx.net/wajueji/caterpillar/3055e2/param/>.
- [4] 中华人民共和国国家发展和改革委员会. 2025 年 4 月 17 日国内成品油价格按机制调整[EB/OL]. 2025. https://www.ndrc.gov.cn/xwdt/xwfb/202504/t20250417_1397240.html.
- [5] 卡特彼勒中国. 卡特彼勒 CAT®770G 非公路卡车参数配置[EB/OL]. 2025. <https://zj.lmjx.net/kuangyongkache/caterpillar/770g/param/>.
- [6] CRISPIN A, SYRICHAS A. Quantum annealing algorithm for vehicle scheduling[C]//2013 IEEE International Conference on Systems, Man, and Cybernetics. 2013: 3523-3528.
- [7] LETCHFORD A N, SALAZAR-GONZÁLEZ J J. The capacitated vehicle routing problem: Stronger bounds in pseudo-polynomial time[J]. European Journal of Operational Research, 2019, 272(1): 24-31.
- [8] COLORNI A, DORIGO M, MANIEZZO V, et al. Distributed optimization by ant colonies [C]//Proceedings of the first European conference on artificial life: vol. 142. 1991: 134-142.
- [9] KENNEDY J, EBERHART R. Particle swarm optimization[C]//Proceedings of ICNN'95-international conference on neural networks: vol. 4. 1995: 1942-1948.

- [10] YIN W, ZHANG C, CHEN H, et al. Metric3d: Towards zero-shot metric 3d prediction from a single image[C]//Proceedings of the IEEE/CVF International Conference on Computer Vision. 2023: 9043-9053.
- [11] YANG L, KANG B, HUANG Z, et al. Depth anything: Unleashing the power of large-scale unlabeled data[C]//Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition. 2024: 10371-10381.
- [12] YANG L, KANG B, HUANG Z, et al. Depth anything v2[J]. Advances in Neural Information Processing Systems, 2024, 37: 21875-21911.
- [13] BOCHKOVSKII A, DELAUNOY A, GERMAIN H, et al. Depth pro: Sharp monocular metric depth in less than a second[J]. arXiv preprint arXiv:2410.02073, 2024.

附录一：附录结构说明

表 5: 附录结构说明

附录索引	解释
附录 2-1	高速公路边坡挖方体积可视化
附录 2-2	河道开挖最优路径完整可视化
附录 2-3	高速公路边坡修建最优路径完整可视化
附录 2-4	从视频到三维建模
附录 3	运行代码所需要的 python 依赖
附录 4	第一问源代码
附录 5	第二问源代码
附录 6	第三问源代码

附录二：完整可视化

高速公路边坡挖方体积可视化

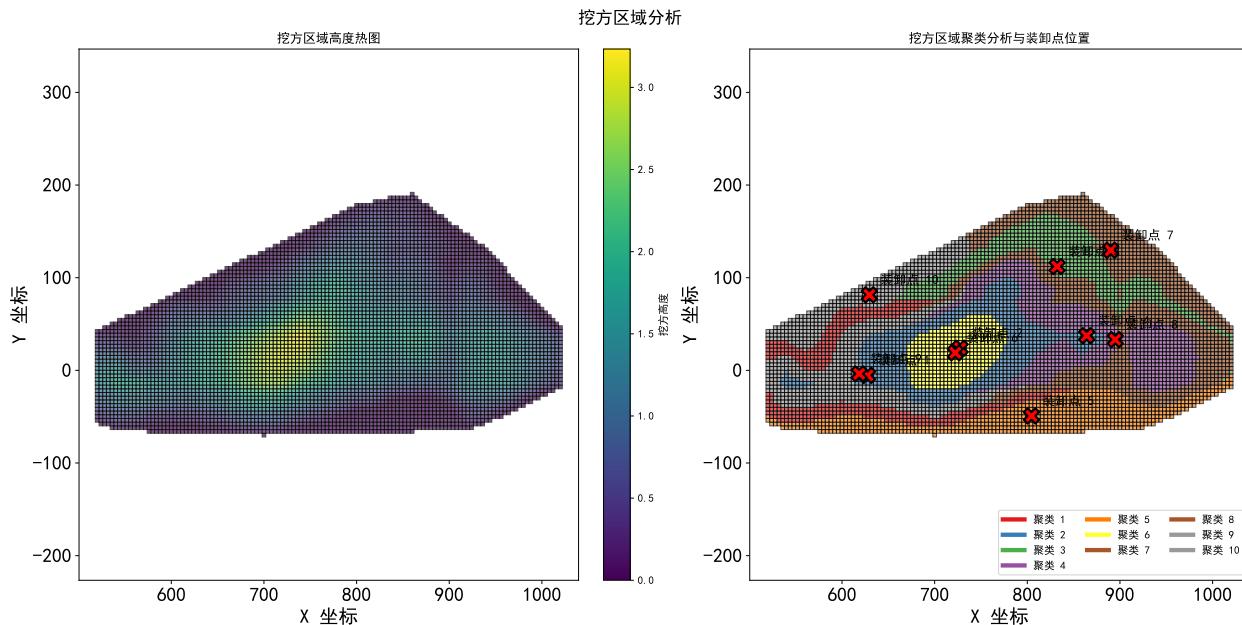


图 11: 高速公路边坡挖方体积可视化

河道开挖最优路径完整可视化

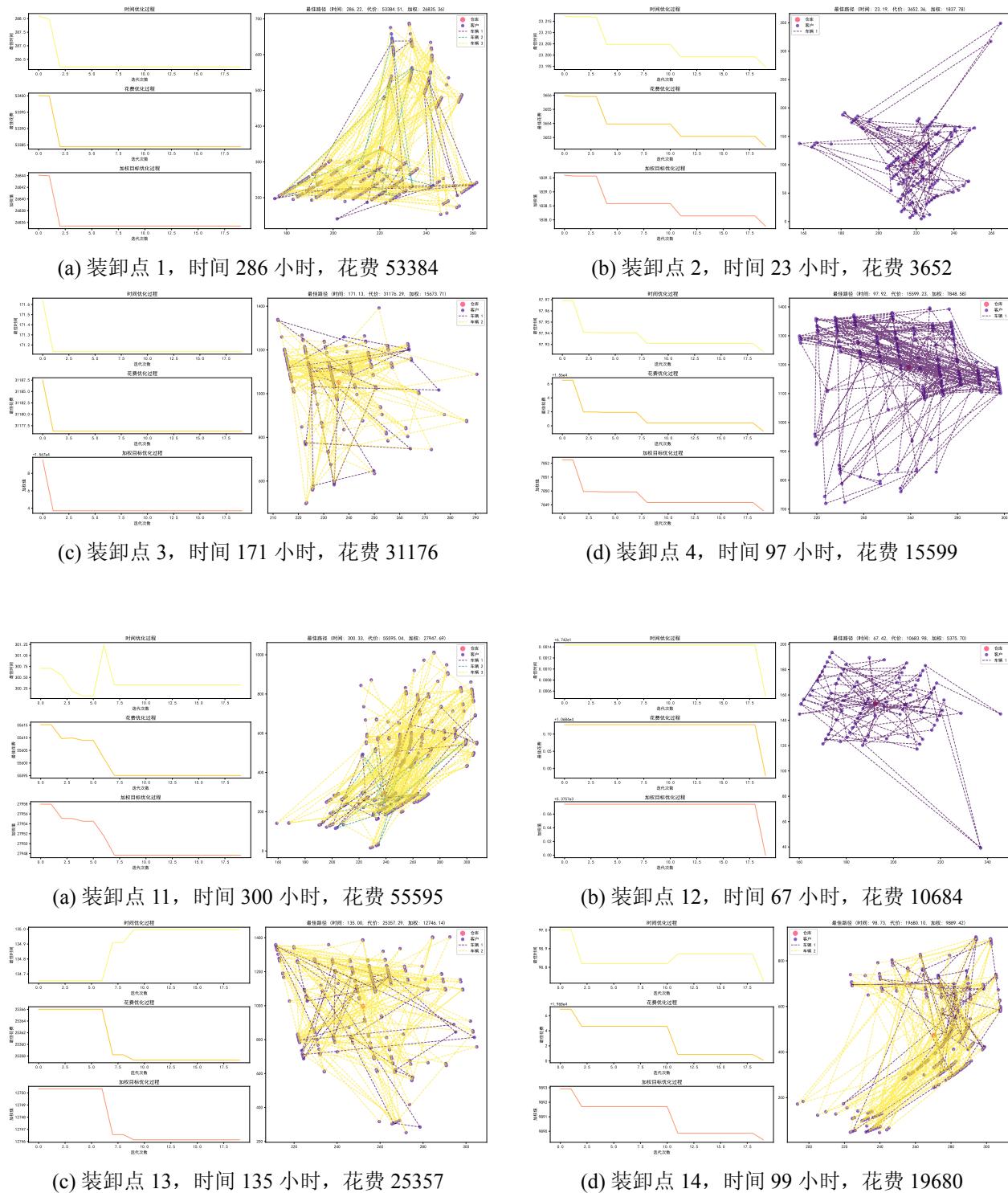
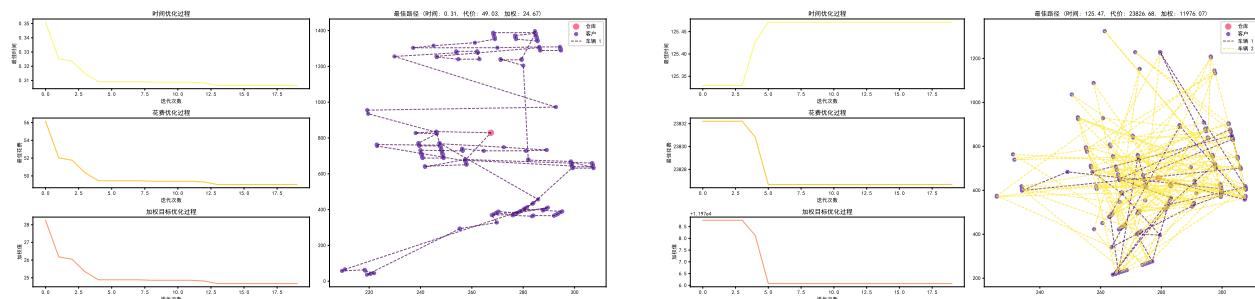
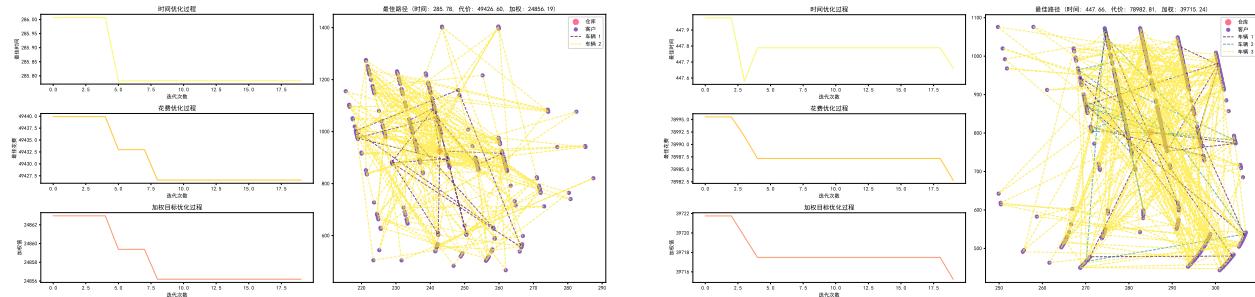


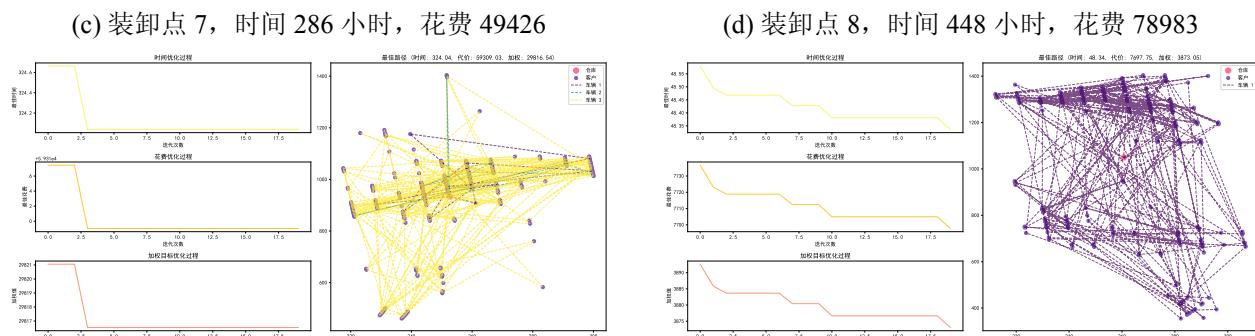
图 14: 河道开挖最优路径完整可视化



(a) 装卸点 5, 时间 0.31 小时, 花费 49



(b) 装卸点 6, 时间 125 小时, 花费 23827

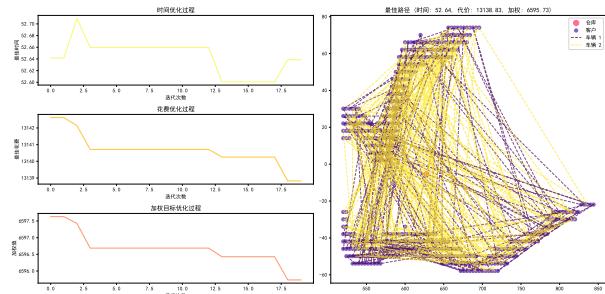


(c) 装卸点 7, 时间 286 小时, 花费 49426

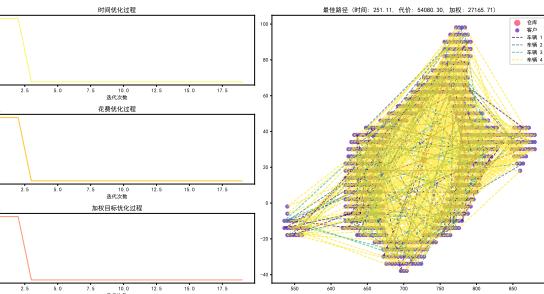
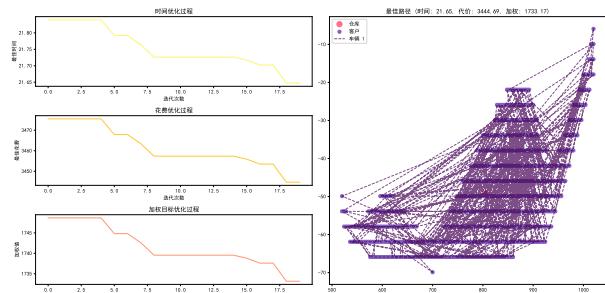


(d) 装卸点 8, 时间 448 小时, 花费 78983

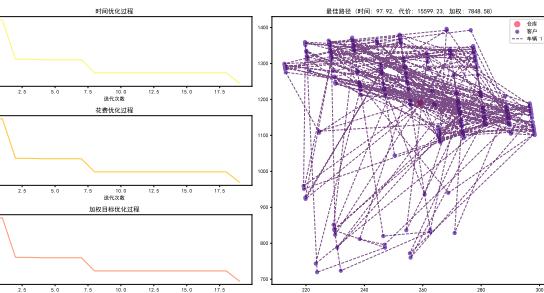
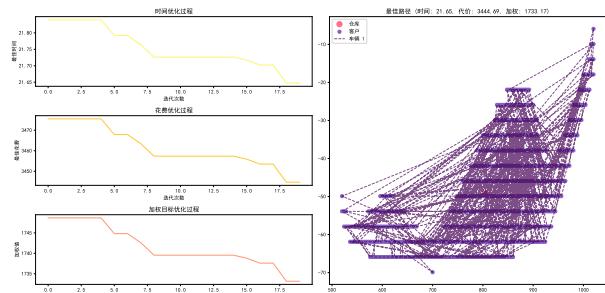
高速公路边坡修建最优路径完整可视化



(a) 装卸点 1, 时间 52 小时, 花费 13138



(b) 装卸点 2, 时间 251 小时, 花费 54080



(c) 装卸点 3, 时间 22 小时, 花费 1733

(d) 装卸点 4, 时间 98 小时, 花费 15599

从视频到三维建模



图 16: 从视频生成的三维模型

图16是从一个 100 帧，17s 的稀疏视频生成的三位模型，在更多图片的输入的情况下，模型的细节明显改善。

附录三：运行代码所需要的 python 依赖

Listing 1: requirements.txt

```
numpy  
scipy  
matplotlib  
pandas  
json  
joblib  
typing  
yaml  
logging
```

附录四：第一问源代码

Listing 2: P1_river_slope.py

```
import json
import matplotlib.pyplot as plt
import re
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
from scipy.interpolate import LinearNDInterpolator
import numpy as np
from scipy.spatial import Delaunay
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d.art3d import Poly3DCollection
from scipy.interpolate import griddata
from matplotlib import cm

def calculate_distance(point1, point2):
    return ((point1[0] - point2[0]) ** 2 +
            (point1[1] - point2[1]) ** 2) ** 0.5

def calculate_x(y, point1, point2):
    return (y - point1[1]) * (point2[0] - point1[0]) /
           (point2[1] - point1[1]) + point1[0]

def calculate_z(y, point1, point2):
    return (y - point1[1]) * (point2[2] - point1[2]) /
           (point2[1] - point1[1]) + point1[2]

def find_x(y, points):
    if(points[0][1] > y):
        return points[0][0]
    for index, point in enumerate(points):
        if(point[1] > y):
            return calculate_x(y, points[index - 1], point)
    return points[-1][0]

def find_z(y, points):
    if(points[0][1] > y):
        return points[0][2]
    for index, point in enumerate(points):
        if(point[1] > y):
            return calculate_z(y, points[index - 1], point)
    return points[-1][2]

def find_closest_point(point, points, middle_lines):
    is_left = point[0] < find_x(point[1], middle_lines)
```

```

new_points = [p for p in points if (p[0] > point[0]
if is_left else p[0] < point[0])]
#寻找距离最近的点，返回4维point坐标
if len(new_points) == 0:
    return (point[0], point[1], 0, 0)
closest_point = min(new_points, key=lambda
p: calculate_distance(point, p))
return (point[0], point[1], closest_point[2], closest_point[3])
def get_slope():
    # 读取 JSON 文件
    with open('dxf_layers_data.json', 'r', encoding='utf-8') as json_file:
        layer_entities = json.load(json_file)
    # 指定要遍历的图层名称
    specified_layers = ["开口线", "水下变坡线", "压顶线",
    "ZH-25- 施工图"]
    # 初始化最小和最大坐标值
    x_min = float('inf')
    x_max = float('-inf')
    y_min = float('inf')
    y_max = float('-inf')
    # 提取指定图层所有线的起点和终点坐标，并更新最小和最大坐标值
    lines = []
    middle_lines=[]
    sidelines_1 = []
    sidelines_2 = []
    insertion_points = []
    for layer_name, layer in layer_entities.items():
        if layer_name == "文字标注":
            for label in layer['labels']:
                insertion_points.append([label
                ['insertion_point'][0], label['insertion_point'][1],
                label['text'].split(":")[0],
                re.search(r'^[-+]?\\d+\\.?\\d*', label['text'].split(":")[1]).group(0)])
        if layer_name == "90m-520m半径方案 925":
            for label in layer['labels']:
                insertion_points.append([label['insertion_point'][0],
                label['insertion_point'][1],
                label['text'].split(":")[0],
                re.search(r'^[-+]?\\d+\\.?\\d*', label['text'].split(":")[1]).group(0)])

```

```

if layer_name == "8500~13000 推荐方案":
    for label in layer[ 'labels' ]:
        insertion_points.append([label[ 'insertion_point' ][0] ,
                                label[ 'insertion_point' ][1] ,
                                label[ 'text' ].split(":")[0] ,
                                re.search(r'^[-+]?\\d+\\.?\\d*', 
                                          label[ 'text' ].split(":")[1]).group(0)])
if layer_name in specified_layers:
    for line in layer[ 'lines' ]:
        start = line[ 'start' ]
        end = line[ 'end' ]
        lines.append((start, end))
if layer_name == "ZH-25- 施工图":
    middle_lines.append(((start[0]+end[0])/2,
                          (start[1]+end[1])/2))
    sidelines_1.append((start[0], start[1]))
    sidelines_2.append((end[0], end[1]))
    x_min = min(x_min, start[0], end[0])
    x_max = max(x_max, start[0], end[0])
    y_min = min(y_min, start[1], end[1])
    y_max = max(y_max, start[1], end[1])
# 提取指定图层所有多段线的数据，并更新最小和最大坐标值
lwpolylines = {
    "开口线": [] ,
    "水下变坡线": [] ,
    "压顶线": []}
for layer_name, layer in layer_entities.items():
    if layer_name in lwpolylines.keys():
        for lwpolyline in layer.get('lwpolylines', []):
            points = lwpolyline[ 'points' ]
            for point in points:
                lwpolylines[layer_name].append((point[0], point[1]))
                x_min = min(x_min, point[0])
                x_max = max(x_max, point[0])
                y_min = min(y_min, point[1])
                y_max = max(y_max, point[1])
middle_lines = sorted(middle_lines,
key=lambda point: point[1]) # 按照y坐标排序
sidelines_1 = sorted(sidelines_1,
key=lambda point: point[1]) # 按照y坐标排序
sidelines_2 = sorted(sidelines_2,
key=lambda point: point[1]) # 按照y坐标排序

```

```
#对每个图层的点分为左右两侧按y轴进行排序，分别存储左右点
for layer_name, points in lwpolylines.items():
    left_points = []
    right_points = []
    for point in points:
        if point[0] < find_x(point[1], middle_lines):
            left_points.append(point)
        else:
            right_points.append(point)
    left_points.sort(key=lambda point: point[1])
    right_points.sort(key=lambda point: point[1])
    lwpolylines[layer_name] = [left_points, right_points]

closest_point = {
    "开口线左侧": [],
    "开口线右侧": [],
    "水下变坡线左侧": [],
    "水下变坡线右侧": [],
    "压顶线左侧": [],
    "压顶线右侧": []}

for layer_name, points in lwpolylines.items():
    for point in points[0]:
        if layer_name == "开口线":
            closest_point[layer_name+"左侧"].append(
                find_closest_point(point, insertion_points,
                                    middle_lines))
        elif layer_name == "水下变坡线":
            closest_point[layer_name+"左侧"].append(
                find_closest_point(point, insertion_points,
                                    middle_lines))
        elif layer_name == "压顶线":
            closest_point[layer_name+"左侧"].append(
                find_closest_point(point, insertion_points,
                                    middle_lines))

    for point in points[1]:
        if layer_name == "开口线":
            closest_point[layer_name+"右侧"].append(
                find_closest_point(point, insertion_points,
                                    middle_lines))
        elif layer_name == "水下变坡线":
            closest_point[layer_name+"右侧"].append(
                find_closest_point(point, insertion_points,
```

```
, middle_lines))
elif layer_name == "压顶线":
    closest_point[layer_name+"右侧"].append
    (find_closest_point(point, insertion_points,
    middle_lines))

base_z = -7.61
new_polylines = {
    "开口线左侧": [],
    "开口线右侧": [],
    "水下变坡线左侧": [],
    "水下变坡线右侧": [],
    "压顶线左侧": [],
    "压顶线右侧": [],
}
for layer_name, points in closest_point.items():
    if "水下变坡线" in layer_name or "压顶线" in layer_name:
        for point in points:
            x = find_x(point[1], sidelines_1 if point[0]
            < find_x(point[1], middle_lines) else sidelines_2)
            new_polylines[layer_name].append([point[0],
            point[1], base_z + abs(point[0] - x)/float(point[3])
            if point[3] != 0 else base_z])
for layer_name, points in closest_point.items():
    if layer_name == "开口线左侧":
        for point in points:
            x = find_x(point[1], new_polylines["水下变坡线左侧"])
            z = find_z(point[1], new_polylines["水下变坡线左侧"])
            if x != new_polylines["水下变坡线左侧"][0][0] and
            x != new_polylines["水下变坡线左侧"][-1][0]:
                new_polylines[layer_name].append([point[0]
                , point[1], z + abs(point[0] - x)/float(point[3])
                if point[3] != 0 else z])
        else:
            x = find_x(point[1], sidelines_1)
            new_polylines[layer_name].append([point[0],
            point[1], base_z + abs(point[0] - x)/float(point[3])
            if point[3] != 0 else base_z])
    elif layer_name == "开口线右侧":
        for point in points:
            x = find_x(point[1], new_polylines["水下变坡线右侧"])
            z = find_z(point[1], new_polylines["水下变坡线右侧"])
```

```

        if x != new_polylines[”水下变坡线右侧”][0][0]
        and x != new_polylines[”水下变坡线右侧”][-1][0]:
            new_polylines[layer_name].append([point[0], point[1],
z + abs(point[0] - x)/float(point[3])
                if point[3] != 0 else z])
        else:
            x = find_x(point[1], sidelines_2)
            new_polylines[layer_name].append([point[0],
point[1], base_z + abs(point[0] - x)/float(point[3])
                if point[3] != 0 else base_z])

results={}
results[”new_polylines”]=new_polylines
results[”middle_lines”]=middle_lines
results[”sidelines_1”]=sidelines_1
results[”sidelines_2”]=sidelines_2
results[”base_z”]=base_z
results[”x_min”]=x_min
results[”x_max”]=x_max
results[”y_min”]=y_min
results[”y_max”]=y_max
return results

if __name__ == ”__main__":
    results = get_slope()
    # 添加坐标偏移常量
    X_OFFSET = 486400
    Y_OFFSET = 2427000
    # 在绘图前调整坐标的函数
    def adjust_coordinates(x, y):
        return x - X_OFFSET, y - Y_OFFSET
    # 在绘制三维图之前调整所有点的坐标
    adjusted_new_polylines = {}
    for layer_name, points in results[”new_polylines”].items():
        adjusted_points = []
        for point in points:
            adjusted_x, adjusted_y = adjust_coordinates(point[0],
point[1])
            adjusted_points.append([adjusted_x, adjusted_y, point[2]])
        adjusted_new_polylines[layer_name] = adjusted_points

adjusted_sidelines_1 = []
for point in results[”sidelines_1”]:

```

```

adjusted_x, adjusted_y = adjust_coordinates(point[0], point[1])
adjusted_sidelines_1.append((adjusted_x, adjusted_y))

adjusted_sidelines_2 = []
for point in results["sidelines_2"]:
    adjusted_x, adjusted_y = adjust_coordinates(point[0], point[1])
    adjusted_sidelines_2.append((adjusted_x, adjusted_y))

adjusted_middle_lines = []
for point in results["middle_lines"]:
    adjusted_x, adjusted_y = adjust_coordinates(point[0], point[1])
    adjusted_middle_lines.append((adjusted_x, adjusted_y))

plt.rcParams['font.sans-serif'] = ['SimHei']
plt.rcParams['axes.unicode_minus'] = False
plt.rcParams['figure.dpi'] = 300
plt.rcParams['savefig.dpi'] = 300
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
#三维线图
for layer_name, points in adjusted_new_polylines.items():
    xs = [point[0] for point in points]
    ys = [point[1] for point in points]
    zs = [point[2] for point in points]
    ax.plot(xs, ys, zs, label=layer_name)

x = [point[0] for point in adjusted_sidelines_1]
y = [point[1] for point in adjusted_sidelines_1]
ax.plot(x, y, results["base_z"], label="航道线左侧")

x = [point[0] for point in adjusted_sidelines_2]
y = [point[1] for point in adjusted_sidelines_2]
ax.plot(x, y, results["base_z"], label="航道线右侧")

x = [point[0] for point in adjusted_middle_lines]
y = [point[1] for point in adjusted_middle_lines]
ax.plot(x, y, results["base_z"], label="航道中线")

ax.legend()
plt.savefig('三维线图.pdf', format='pdf', bbox_inches='tight')
#plt.show()

```

```

# 合并所有点
all_points = []
for layer_name, points in adjusted_new_polylines.items():
    all_points.extend([(p[0], p[1], p[2]) for p in points])
for point in adjusted_sidelines_1:
    all_points.append((point[0], point[1], results["base_z"]))
for point in adjusted_sidelines_2:
    all_points.append((point[0], point[1], results["base_z"]))
for point in adjusted_middle_lines:
    all_points.append((point[0], point[1], results["base_z"]))

points_2d = np.array([(p[0], p[1]) for p in all_points])
tri = Delaunay(points_2d)

# 绘制坡面
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
surf = ax.plot_trisurf(
    points_2d[:, 0], points_2d[:, 1], np.array([p[2] for p
        in all_points]),
    triangles=tri.simplices,
    cmap='terrain', alpha=0.8
)

colorbar = fig.colorbar(surf, ax=ax, shrink=0.5, aspect=10,
label='高程(m)')

for layer_name, points in adjusted_new_polylines.items():
    xs = [p[0] for p in points]
    ys = [p[1] for p in points]
    zs = [p[2] for p in points]
    ax.plot(xs, ys, zs, color='black', linewidth=0.5)

plt.title("三维坡面图")
plt.savefig('三维坡面图.pdf', format='pdf', bbox_inches='tight')

plt.figure(figsize=(10, 6))
for layer_name, points in adjusted_new_polylines.items():
    # 提取XY坐标和高程Z值
    x = [p[0] for p in points]
    y = [p[1] for p in points]
    z = [p[2] for p in points]
    scatter = plt.scatter(x, y, c=z, cmap='viridis', s=5,

```

```

label=layer_name)

# 添加航道线
plt.plot([p[0] for p in adjusted_sidelines_1], [p[1] for p
in adjusted_sidelines_1], 'k--', lw=1, label="航道边线")
plt.plot([p[0] for p in adjusted_sidelines_2], [p[1] for p
in adjusted_sidelines_2], 'k--', lw=1)
plt.plot([p[0] for p in adjusted_middle_lines], [p[1] for p
in adjusted_middle_lines], 'r-', lw=1, label="航道中线")

plt.colorbar(scatter, label='高程(m)')
plt.title("二维平面投影 (颜色表示高程)")
plt.legend()

adjusted_x_min = results['x_min'] - X_OFFSET
adjusted_x_max = results['x_max'] - X_OFFSET
adjusted_y_min = results['y_min'] - Y_OFFSET
adjusted_y_max = results['y_max'] - Y_OFFSET
plt.xlim(adjusted_x_min, adjusted_x_max)
plt.ylim(adjusted_y_min, adjusted_y_max)
plt.savefig('二维平面投影图.pdf', format='pdf', bbox_inches='tight')

x = np.array([p[0] for p in all_points])
y = np.array([p[1] for p in all_points])
z = np.array([p[2] for p in all_points])
xi = np.linspace(min(x)-1, max(x)+1, 200)
yi = np.linspace(min(y)-1, max(y)+1, 200)
X, Y = np.meshgrid(xi, yi)
Z = griddata((x, y), z, (X, Y), method='linear')
plt.figure(figsize=(12, 8))
levels = np.linspace(z.min()-0.5, z.max()+0.5, 15)
# 根据实际高程范围设定
contour = plt.contourf(X, Y, Z, levels=levels,
cmap=cm.terrain, extend='both')
plt.colorbar(contour, label='高程(m)', shrink=0.8)
C = plt.contour(X, Y, Z, levels=levels, colors='black',
linewidths=1.5)
plt.clabel(C, inline=True, fontsize=16, fmt='%.1f')
# plt.title("高程等高线图")
plt.xlabel('x(m)', fontsize=20)
plt.ylabel('y(m)', fontsize=20)
plt.grid(True, linestyle=':', alpha=0.3)

```

```

plt.xlim(adjusted_x_min, adjusted_x_max)
plt.ylim(adjusted_y_min, adjusted_y_max)
plt.xticks(fontsize=18)
plt.yticks(fontsize=18)

plt.tight_layout()
plt.savefig('高程等高线图.pdf', format='pdf', bbox_inches='tight')
# plt.show()

```

Listing 3: P1_river_earth

```

import json
import matplotlib.pyplot as plt
import re
from P1_river_slope import get_slope
import numpy as np
from scipy.spatial import Delaunay
X_OFFSET = 486400
Y_OFFSET = 2427000
def get_earthwork():
    # 读取 JSON 文件
    with open("dxf_layers_data.json", 'r', encoding='utf-8') as json_file:
        layer_entities = json.load(json_file)
    lwpolylines = []
    texts = {
        "insertion_point": [],
        "水位": [],
        "硬底差": [],
        "square_point": [],
        "z": [],
        "height": []}
    for layer_name, layer in layer_entities.items():
        if layer_name == "船位":
            for lwpolyline in layer["lwpolylines"]:
                lwpolylines.append(lwpolyline["points"])
        elif layer_name == "开挖船位文字标注":
            for mtext in layer["mtext"]:
                texts["insertion_point"]
                .append(mtext["insertion_point"])
                text = mtext["text"]
                # 匹配“水位”后和“m”前的浮点数

```

```
match = re.search(r"水位 ([+-]?\d+(?:\.\d+)?)", m, text)
if match:
    texts["水位"].append(float(match.group(1)))
else:
    texts["水位"].append(0)
#匹配"差"后和"m"前的浮点数
match = re.search(r"差 ([+-]?\d+(?:\.\d+)?)", m, text)
if match:
    texts["硬底差"].append(float(match.group(1)))
else:
    texts["硬底差"].append(0)

base_z = 7.61
square = []
true_square = []
for polyline in lwpolylines:
    x = [point[0] for point in polyline]
    y = [point[1] for point in polyline]
    square.append((min(x), min(y), max(x), max(y)))
#画长方形需要5个点
    x.append(x[0])
    y.append(y[0])
    true_square.append([x, y])
#根据insertion_point将texts中的浮点数和square匹配
for i in range(len(texts["insertion_point"])):
    for j in range(len(square)):
        if texts["insertion_point"][i][0] >=
            square[j][0] and texts["insertion_point"][i][0] <=
            square[j][2] and texts["insertion_point"][i][1] >=
            square[j][1] and texts["insertion_point"][i][1] <=
            square[j][3]:
            if texts["水位"][i] > base_z:
                texts["height"].append(base_z - texts["水位"]
                    [i] + texts["硬底差"][i])
            else:
                texts["height"].append(base_z - texts["水位"]
                    [i] - texts["硬底差"][i])
            texts["square_point"].append(true_square[j])
            texts["z"].append(-base_z + texts["height"][-1])
            break
return texts
```

```
if __name__ == "__main__":
    plt.rcParams['font.sans-serif'] = ['SimHei']
    plt.rcParams['axes.unicode_minus'] = False

results = get_slope()

def adjust_coordinates(x, y):
    return x - X_OFFSET, y - Y_OFFSET

adjusted_new_polylines = {}
for layer_name, points in results["new_polylines"].items():
    adjusted_points = []
    for point in points:
        adjusted_x, adjusted_y = adjust_coordinates(point[0], point[1])
        adjusted_points.append([adjusted_x, adjusted_y, point[2]])
    adjusted_new_polylines[layer_name] = adjusted_points

adjusted_sidelines_1 = []
for point in results["sidelines_1"]:
    adjusted_x, adjusted_y =
        adjust_coordinates(point[0], point[1])
    adjusted_sidelines_1.append((adjusted_x, adjusted_y))

adjusted_sidelines_2 = []
for point in results["sidelines_2"]:
    adjusted_x, adjusted_y =
        adjust_coordinates(point[0], point[1])
    adjusted_sidelines_2.append((adjusted_x, adjusted_y))

adjusted_middle_lines = []
for point in results["middle_lines"]:
    adjusted_x, adjusted_y =
        adjust_coordinates(point[0], point[1])
    adjusted_middle_lines.append((adjusted_x, adjusted_y))

texts = get_earthwork()

fig = plt.figure(figsize=(15, 12))
ax = fig.add_subplot(111, projection='3d')
```

1. 绘制方格挖方区域

```

for i in range(len(texts["square_point"])):
    x = [texts["square_point"][i][0] - X_OFFSET,
          texts["square_point"][i][2] - X_OFFSET,
          texts["square_point"][i][2] - X_OFFSET,
          texts["square_point"][i][0] - X_OFFSET,
          texts["square_point"][i][0] - X_OFFSET]
    y = [texts["square_point"][i][1] - Y_OFFSET,
          texts["square_point"][i][1] - Y_OFFSET,
          texts["square_point"][i][3] - Y_OFFSET,
          texts["square_point"][i][3] - Y_OFFSET,
          texts["square_point"][i][1] - Y_OFFSET]
    z = [texts["z"][i]] * 5
    ax.plot(x, y, z, linewidth=1.5, color='blue')

```

2. 绘制三角剖分坡面

```

all_points = []
for layer_name, points in adjusted_new_polylines.items():
    all_points.extend([(p[0], p[1], p[2]) for p in points])
for point in adjusted_sidelines_1:
    all_points.append((point[0], point[1], results["base_z"]))
for point in adjusted_sidelines_2:
    all_points.append((point[0], point[1], results["base_z"]))
for point in adjusted_middle_lines:
    all_points.append((point[0], point[1], results["base_z"]))

points_2d = np.array([(p[0], p[1]) for p in all_points])
tri = Delaunay(points_2d)

```

```

surf = ax.plot_trisurf(
    points_2d[:, 0], points_2d[:, 1],
    np.array([p[2] for p in all_points]),
    triangles=tri.simplices,
    cmap='terrain', alpha=0.6
)
colorbar = fig.colorbar(surf, ax=ax, shrink=0.5,
                        aspect=10, label='高程(m)')

```

3. 绘制三维线框

```

for layer_name, points in adjusted_new_polylines.items():
    xs = [point[0] for point in points]
    ys = [point[1] for point in points]
    zs = [point[2] for point in points]

```

```

        ax.plot(xs, ys, zs, label=layer_name, linewidth=1.5)
x = [point[0] for point in adjusted_sidelines_1]
y = [point[1] for point in adjusted_sidelines_1]
ax.plot(x, y, results["base_z"], label="航道线左侧",
linewidth=1.5)

x = [point[0] for point in adjusted_sidelines_2]
y = [point[1] for point in adjusted_sidelines_2]
ax.plot(x, y, results["base_z"], label="航道线右侧",
linewidth=1.5)

x = [point[0] for point in adjusted_middle_lines]
y = [point[1] for point in adjusted_middle_lines]
ax.plot(x, y, results["base_z"], label="航道中线",
linewidth=1.5)

ax.legend(fontsize=10, loc='best')

ax.set_xlabel('X', fontsize=12)
ax.set_ylabel('Y', fontsize=12)
ax.set_zlabel('Z', fontsize=12)
plt.title('三维挖方综合示意图', fontsize=14)
ax.view_init(elev=30, azim=45)
plt.tight_layout()
plt.show()
plt.savefig("3d_plot_combined.png", dpi=300, bbox_inches='tight')

volume = 0
for i in range(len(texts["square_point"])):
    square_area = (texts["square_point"][i][2] -
    texts["square_point"][i][0]) * (texts["square_point"][i][3] -
    texts["square_point"][i][1])
    volume += square_area * texts["height"][i]
print("挖方体积为：", volume)

```

附录五：第二问源代码

Listing 4: P2_road_slope

```
import json
```

```

import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
import numpy as np
from scipy.spatial import Delaunay
from mpl_toolkits.mplot3d.art3d import Poly3DCollection

# 等高线图绘制
import numpy as np
import matplotlib.pyplot as plt
from matplotlib import cm
from scipy.interpolate import griddata

def get_road_slope():
    points={}
    plt.rcParams[ 'font.sans-serif' ] = [ 'SimHei' ]
    plt.rcParams[ 'axes.unicode_minus' ] = False
    def calculate_distance(point1, point2):
        return ((point1[0] - point2[0]) ** 2 +
        (point1[1] - point2[1]) ** 2) ** 0.5
    # 读取 JSON 文件
    with open('gradient_layers_data.json', 'r', encoding='utf-8') as json_file:
        layer_entities = json.load(json_file)

    axis_lines = layer_entities[ "横_轴线" ][ "lines" ]
    axis_x_coordinates = [line[ "start" ][0] for line in axis_lines]
    design_lines = layer_entities[ "横_设计线" ][ "lwpolylines" ]

    reference_points = []
    start_index = 0
    ans = 0
    for axis_x in axis_x_coordinates:
        for i in range(start_index, len(design_lines)):
            # 从 start_index 开始遍历
            polyline = design_lines[ i ]
            point = polyline[ "points" ][0]
            if abs( point[0] - axis_x ) <= 0.05:
                ans += 1
                if ans == 2: # 只有出现两次才会记录
                    reference_points.append( point )
                    start_index = i + 1 # 下次从下一个 折线开始
                    ans = 0

```

```
break  
# 将参考点映射到真实世界坐标  
real_world_coordinates = []  
z_values = [  
    188.325, 188.539, 188.753, 188.968, 189.182, 189.396,  
    189.61, 189.824,  
    190.039, 190.253, 190.467, 190.681, 190.896, 191.11,  
    191.324, 191.538,  
    191.752, 191.967, 192.181, 192.395, 192.609, 192.823,  
    193.038, 193.252,  
    193.466, 193.68  
]  
# 读图一个一个列出来的z坐标  
x_start = 520 # x 坐标起始值  
x_increment = 20 # 每个点 x 坐标的增加的量  
for i, point in enumerate(reference_points):  
    real_x = x_start + i * x_increment  
    real_y = 0  
    real_z = z_values[i]  
    real_world_coordinates.append((real_x, real_y, real_z))  
  
An = [3, 2, 2, 2, 2, 2, 1, 1, 1, 1, 2, 2, 2, 2]  
converted_points = []  
for polyline in design_lines:  
    for vertex in polyline["points"]:  
        x = vertex[0]  
        y = vertex[1]  
        # 计算可能的参考点索引范围  
        column = int((x - 4556) / 258)  
        if column < 0 or column >= len(An):  
            continue  
  
        sum_indices = sum(An[:column + 1])  
        possible_indices = range(sum_indices - An[column],  
                                 sum_indices)  
        # 找到对应的参考点  
        for idx in possible_indices:  
            if idx < 0 or idx >= len(reference_points):  
                continue  
            ref_point = reference_points[idx]  
            if y > ref_point[1] - 3:  
                x0, y0, z0 = real_world_coordinates[idx]  
                x1 = x0
```

```
y1 = x - ref_point[0]
z1 = y - ref_point[1] + z0
converted_points.append((x1, y1, z1))
break

# 提取三维坐标
converted_x = [p[0] for p in converted_points]
converted_y = [p[1] for p in converted_points]
converted_z = [p[2] for p in converted_points]

points["x"] = converted_x
points["y"] = converted_y
points["z"] = converted_z
points["base_z"] = min(z_values)
points["converted_points"] = converted_points
return points

if __name__ == "__main__":
    points = get_road_slope()
    converted_x = points["x"]
    converted_y = points["y"]
    converted_z = points["z"]
    base_z = points["base_z"]
    converted_points = points["converted_points"]

    fig = plt.figure(figsize=(10, 6))
    ax = fig.add_subplot(111, projection='3d')
    ax.scatter(converted_x, converted_y, converted_z, color='red',
               label='转化后的点',
               marker='o')
    ax.plot_trisurf(converted_x, converted_y, converted_z,
                     color='blue', alpha=0.5,
                     edgecolor='none')

    ax.set_xlabel('X\u2083坐标(\u5e02\u573a\u5e02)')
    ax.set_ylabel('Y\u2083坐标(\u5e02\u573a\u5e02)')
    ax.set_zlabel('Z\u2083坐标(\u5e02\u573a\u5e02)')
    ax.set_title('横\u2083设计线顶点在真实世界坐标中的分布及拟合曲面')
    ax.legend()
    ax.grid(True)
    plt.show()

points_2d = np.array([(p[0], p[1]) for p in converted_points])
```

```
tri = Delaunay(points_2d)
fig = plt.figure(figsize=(10, 6))
ax = fig.add_subplot(111, projection='3d')
surf = ax.plot_trisurf(
    points_2d[:, 0],
    points_2d[:, 1],
    [p[2] for p in converted_points],
    triangles=tri.simplices,
    cmap='terrain',
    alpha=0.7,
    edgecolor='none',
)

min_z = min([p[2] for p in converted_points]) + 5
y_min_indices = []
y_max_indices = []
y_coords = np.array([p[1] for p in converted_points])
x_coords = np.array([p[0] for p in converted_points])
unique_x = np.unique(x_coords)
for x_val in unique_x:
    x_mask = (x_coords == x_val)
    if np.sum(x_mask) > 0:
        y_values = y_coords[x_mask]
        min_y_idx = np.where((x_coords == x_val) & (y_coords == np.min(y_values)))
        [0][0]
        max_y_idx = np.where((x_coords == x_val) & (y_coords == np.max(y_values)))
        [0][0]
        y_min_indices.append(min_y_idx)
        y_max_indices.append(max_y_idx)
for idx_list, is_min in [(y_min_indices, True), (y_max_indices, False)]:
    for i in range(len(idx_list) - 1):
        idx1, idx2 = idx_list[i], idx_list[i + 1]
        p1 = converted_points[idx1]
        p2 = converted_points[idx2]
        p3 = (p2[0], p2[1], min_z)
        p4 = (p1[0], p1[1], min_z)

        segments = 50
        z_min = min_z
```

```

z_max_1 = p1[2]
z_max_2 = p2[2]
for j in range(segments):
    z1_bottom = z_min + (z_max_1 - z_min) * j / segments
    z1_top = z_min + (z_max_1 - z_min) * (j + 1) / segments
    z2_bottom = z_min + (z_max_2 - z_min) * j / segments
    z2_top = z_min + (z_max_2 - z_min) * (j + 1) / segments
v1 = (p1[0], p1[1], z1_top)
v2 = (p2[0], p2[1], z2_top)
v3 = (p2[0], p2[1], z2_bottom)
v4 = (p1[0], p1[1], z1_bottom)
verts = [v1, v2, v3, v4]
poly = Poly3DCollection([verts], alpha=0.7)

avg_z = (z1_top + z2_top + z1_bottom + z2_bottom) / 4
norm_z = (avg_z - min([p[2] for p in converted_points])) /
/ (max([p[2] for p in converted_points]) - min([p[2]
for p in converted_points]))
color = plt.cm.terrain(norm_z)
poly.set_color(color)
ax.add_collection3d(poly)

bottom_points = []
for idx in y_min_indices:
    p = converted_points[idx]
    bottom_points.append((p[0], p[1], min_z))
for idx in reversed(y_max_indices):
    p = converted_points[idx]
    bottom_points.append((p[0], p[1], min_z))
if len(bottom_points) > 2:
    poly = Poly3DCollection([bottom_points], alpha=0.7)
    color = plt.cm.terrain(0.0)
    poly.set_color(color)
    ax.add_collection3d(poly)

ax.set_xlabel('X坐标(m)')
ax.set_ylabel('Y坐标(m)')
ax.set_zlabel('高程(m)')
ax.set_title('三维坡面模型与设计线框')
fig.colorbar(surf, ax=ax, shrink=0.5, label='高程(米)')
ax.view_init(elev=30, azim=-45)
ax.set_zlim(min_z, max([p[2] for p in converted_points]) + 5)
plt.tight_layout()

```

```

plt.show()

x = np.array(converted_x)
y = np.array(converted_y)
z = np.array(converted_z)
xi = np.linspace(min(x)-1, max(x)+1, 200)
yi = np.linspace(min(y)-1, max(y)+1, 200)
X, Y = np.meshgrid(xi, yi)
Z = griddata((x, y), z, (X, Y), method='linear')
plt.figure(figsize=(12, 8))

levels = np.linspace(z.min()-0.5, z.max()+0.5, 15)
contour = plt.contourf(X, Y, Z, levels=levels,
cmap=cm.terrain, extend='both')
plt.colorbar(contour, label='高程(m)', shrink=0.8)
C = plt.contour(X, Y, Z, levels=levels, colors='black',
, linewidths=0.5)
plt.clabel(C, inline=True, fontsize=8, fmt='%.1f')
plt.title('横_设计线高程等高线图')
plt.xlabel('x(m)')
plt.ylabel('y(m)')
plt.grid(True, linestyle=':', alpha=0.3)
plt.tight_layout()
plt.show()

```

Listing 5: P2_road_earth.py

```

import json
import numpy as np
from scipy.interpolate import interp1d
import numpy as np
from scipy.interpolate import griddata
import matplotlib.pyplot as plt
plt.rcParams['font.sans-serif'] = ['SimHei']
plt.rcParams['axes.unicode_minus'] = False

def get_road_earth():
    # 读取 JSON 文件
    with open('gradient_layers_data.json', 'r', encoding='utf-8') as json_file:
        layer_entities = json.load(json_file)
        axis_lines = layer_entities["横_轴线"]["lines"]

```

```
axis_x_coordinates = [line["start"][0] for line in axis_lines]
design_lines = layer_entities["横_设计线"]["lwpolylines"]
ground_lines = layer_entities["横_原地线"]["lwpolylines"]

reference_points = []
start_index = 0
ans = 0
for axis_x in axis_x_coordinates:
    for i in range(start_index, len(design_lines)):
        polyline = design_lines[i]
        point = polyline["points"][0]
        if abs(point[0] - axis_x) <= 0.05:
            ans += 1
            if ans == 2:
                reference_points.append(point)
                start_index = i + 1
                ans = 0
            break
# 将参考点映射到真实世界坐标
real_world_coordinates = []
z_values = [
    188.325, 188.539, 188.753, 188.968,
    189.182, 189.396, 189.61, 189.824,
    190.039, 190.253, 190.467, 190.681,
    190.896, 191.11, 191.324, 191.538,
    191.752, 191.967, 192.181, 192.395,
    192.609, 192.823, 193.038, 193.252,
    193.466, 193.68
] # 真实世界的 z 坐标
x_start = 520
x_increment = 20
for i, point in enumerate(reference_points):
    real_x = x_start + i * x_increment
    real_y = 0
    real_z = z_values[i]
    real_world_coordinates.append((real_x, real_y, real_z))

An = [3, 2, 2, 2, 2, 2, 1, 1, 1, 1, 2, 2, 2, 2]
design_points_by_reference =
{i: [] for i in range(len(reference_points))}

ground_points_by_reference =
{i: [] for i in range(len(reference_points))}
```

```
for polyline in design_lines:
    for vertex in polyline["points"]:
        x = vertex[0]
        y = vertex[1]
        column = int((x - 4556) / 258)
        if column < 0 or column >= len(An):
            continue
        sum_indices = sum(An[:column + 1])
        possible_indices = range(sum_indices - An[column],
        , sum_indices)
        for idx in possible_indices:
            if idx < 0 or idx >= len(reference_points):
                continue
            ref_point = reference_points[idx]
            if y > ref_point[1] - 3:
                design_points_by_reference[idx].append(vertex)
                break
for polyline in ground_lines:
    for vertex in polyline["points"]:
        x = vertex[0]
        y = vertex[1]
        column = int((x - 4556) / 258)
        if column < 0 or column >= len(An):
            continue
        sum_indices = sum(An[:column + 1])
        possible_indices = range(sum_indices - An[column],
        , sum_indices)
        for idx in possible_indices:
            if idx < 0 or idx >= len(reference_points):
                continue
            ref_point = reference_points[idx]
            if y > ref_point[1] - 3:
                ground_points_by_reference[idx].append(vertex)
                break
results = {}
for ref_idx, ref_point in enumerate(reference_points):
    design_points = design_points_by_reference[ref_idx]
    design_points = sorted(design_points,
    key=lambda p: (p[0], -p[1]))
    unique_design_points = []
    prev_x = None
    for point in design_points:
```

```

        if prev_x is None or point[0] != prev_x:
            unique_design_points.append(point)
            prev_x = point[0]

design_x = [p[0] for p in unique_design_points]
design_y = [p[1] for p in unique_design_points]
design_interp = interp1d(design_x, design_y, kind='linear',
fill_value="extrapolate")
ground_points = ground_points_by_reference[ref_idx]
ground_points = sorted(ground_points, key=lambda p: p[0])
ground_x = [p[0] for p in ground_points]
ground_y = [p[1] for p in ground_points]
x_min, x_max = min(design_x), max(design_x)
x_values = np.linspace(x_min, x_max, 100)
delta_y = []
for x in x_values:
    design_y_value = design_interp(x)
    closest_idx = np.argmin([abs(gx - x) for gx in ground_x])
    ground_y_value = ground_y[closest_idx]
    delta_y_value = ground_y_value - design_y_value

    x0, y0, z0 = real_world_coordinates[ref_idx]
    x_real = x0
    y_real = x - ref_point[0]

    delta_y.append({
        "x_real": x_real,
        "y": y_real,
        "delta_y": delta_y_value
    })
results[f"参考点{ref_idx+1}"] = delta_y
all_points = []
for ref_data in results.values():
    for point in ref_data:
        all_points.append((point["x_real"],
                           point["y"], point["delta_y"]))

all_points = np.array(all_points)
x_real_values = all_points[:, 0]
y_real_values = all_points[:, 1]
delta_y_values = all_points[:, 2]

```

```

x_min, x_max = x_real_values.min(), x_real_values.max()
y_min, y_max = y_real_values.min(), y_real_values.max()
x_step = 4.0
y_step = 4.0
x_grid = np.arange(x_min, x_max + x_step, x_step)
y_grid = np.arange(y_min, y_max + y_step, y_step)
x_mesh, y_mesh = np.meshgrid(x_grid, y_grid)
delta_y_mesh = griddata(
    (x_real_values, y_real_values), # 原始点的 (x, y)
    delta_y_values, # 原始点的 delta_y
    (x_mesh, y_mesh), # 插值网格点
    method='linear' # 线性插值
)

original_grid_area = 1.0 * 1.0
new_grid_area = x_step * y_step
area_ratio = original_grid_area / new_grid_area
delta_y_mesh = delta_y_mesh * area_ratio

interpolated_results = []
nan_count = 0
valid_count = 0
for i in range(x_mesh.shape[0]):
    for j in range(x_mesh.shape[1]):
        x_real = x_mesh[i, j]
        y_real = y_mesh[i, j]
        delta_y = delta_y_mesh[i, j]
        if not np.isnan(delta_y): # 过滤掉插值结果中的 NaN 值
            interpolated_results.append([x_real, y_real, delta_y])
            valid_count += 1
        else:
            nan_count += 1
interpolated_results = np.array(interpolated_results)
return interpolated_results

```

Listing 6: P2_caculate.py

```

import json
import numpy as np
from scipy.interpolate import interp1d

with open('gradient_layers_data.json', 'r', encoding='utf-8')

```

```
as json_file:
    layer_entities = json.load(json_file)
axis_lines = layer_entities["横轴线"]["lines"]
axis_x_coordinates = [line["start"][0] for line in axis_lines]
design_lines = layer_entities["横设计线"]["lwpolylines"]
ground_lines = layer_entities["横原地线"]["lwpolylines"]

reference_points = []
start_index = 0
ans = 0
for axis_x in axis_x_coordinates:
    for i in range(start_index, len(design_lines)):
        polyline = design_lines[i]
        point = polyline["points"][0]
        if abs(point[0] - axis_x) <= 0.05:
            ans += 1
            if ans == 2:
                reference_points.append(point)
                start_index = i + 1
                ans = 0
                break
real_world_coordinates = []
z_values = [
    188.325, 188.539, 188.753, 188.968, 189.182,
    189.396, 189.61, 189.824,
    190.039, 190.253, 190.467, 190.681, 190.896,
    191.11, 191.324, 191.538,
    191.752, 191.967, 192.181, 192.395, 192.609,
    192.823, 193.038, 193.252,
    193.466, 193.68
]
x_start = 0
x_increment = 20
for i, point in enumerate(reference_points):
    real_x = x_start + i * x_increment
    real_y = 0
    real_z = z_values[i]
    real_world_coordinates.append((real_x, real_y, real_z))
An = [3, 2, 2, 2, 2, 2, 1, 1, 1, 1, 2, 2, 2, 2]
design_points_by_reference =
{i: [] for i in range(len(reference_points))}
ground_points_by_reference =
```

```

{i: [] for i in range(len(reference_points))}

for polyline in design_lines:
    for vertex in polyline["points"]:
        x = vertex[0]
        y = vertex[1]
        column = int((x - 4556) / 258)
        if column < 0 or column >= len(An):
            continue
        sum_indices = sum(An[:column + 1])
        possible_indices = range(sum_indices - An[column], sum_indices)
        for idx in possible_indices:
            if idx < 0 or idx >= len(reference_points):
                continue
            ref_point = reference_points[idx]
            if y > ref_point[1] - 3:
                design_points_by_reference[idx].append(vertex)
                break

for polyline in ground_lines:
    for vertex in polyline["points"]:
        x = vertex[0]
        y = vertex[1]
        column = int((x - 4556) / 258)
        if column < 0 or column >= len(An):
            continue
        sum_indices = sum(An[:column + 1])
        possible_indices = range(sum_indices - An[column], sum_indices)
        for idx in possible_indices:
            if idx < 0 or idx >= len(reference_points):
                continue
            ref_point = reference_points[idx]
            if y > ref_point[1] - 3:
                ground_points_by_reference[idx].append(vertex)
                break

results = {}
for ref_idx, ref_point in enumerate(reference_points):
    design_points = design_points_by_reference[ref_idx]
    design_points = sorted(design_points,
                           key=lambda p: (p[0], -p[1]))
    unique_design_points = []
    prev_x = None
    for point in design_points:
        if prev_x is None or point[0] != prev_x:

```

```
unique_design_points.append(point)
prev_x = point[0]
design_x = [p[0] for p in unique_design_points]
design_y = [p[1] for p in unique_design_points]
design_interp = interp1d(design_x, design_y, kind='linear',
fill_value="extrapolate")
ground_points = ground_points_by_reference[ref_idx]
ground_points = sorted(ground_points, key=lambda p: p[0])
ground_x = [p[0] for p in ground_points]
ground_y = [p[1] for p in ground_points]
x_min, x_max = min(design_x), max(design_x)
x_values = np.linspace(x_min, x_max, 100)
delta_y = []
for x in x_values:
    design_y_value = design_interp(x)
    closest_idx = np.argmin([abs(gx - x) for gx in ground_x])
    ground_y_value = ground_y[closest_idx]
    delta_y_value = ground_y_value - design_y_value
    x0, y0, z0 = real_world_coordinates[ref_idx]
    x_real = x0
    y_real = x - ref_point[0]
    delta_y.append({
        "x_real": x_real,
        "y": y_real,
        "delta_y": delta_y_value
    })
results[f"参考点{ref_idx+1}"] = delta_y

import numpy as np
from scipy.interpolate import griddata
import matplotlib.pyplot as plt
plt.rcParams['font.sans-serif'] = ['SimHei']
plt.rcParams['axes.unicode_minus'] = False

all_points = []
for ref_data in results.values():
    for point in ref_data:
        all_points.append((point["x_real"],
                           point["y"], point["delta_y"]))

all_points = np.array(all_points)
x_real_values = all_points[:, 0]
```

```
y_real_values = all_points[:, 1]
delta_y_values = all_points[:, 2]
x_min, x_max = x_real_values.min(), x_real_values.max()
y_min, y_max = y_real_values.min(), y_real_values.max()
x_step = 1.0
y_step = 1.0
x_grid = np.arange(x_min, x_max + x_step, x_step)
y_grid = np.arange(y_min, y_max + y_step, y_step)
x_mesh, y_mesh = np.meshgrid(x_grid, y_grid)
delta_y_mesh = griddata(
    (x_real_values, y_real_values),
    delta_y_values,
    (x_mesh, y_mesh),
    method='linear')

interpolated_results = []
nan_count = 0
valid_count = 0
for i in range(x_mesh.shape[0]):
    for j in range(x_mesh.shape[1]):
        x_real = x_mesh[i, j]
        y_real = y_mesh[i, j]
        delta_y = delta_y_mesh[i, j]
        if not np.isnan(delta_y):
            interpolated_results.append({
                "x_real": x_real,
                "y": y_real,
                "delta_y": delta_y
            })
            valid_count += 1
        else:
            nan_count += 1
fig = plt.figure(figsize=(10, 6))
ax = fig.add_subplot(111, projection='3d')
sc = ax.scatter(
    x_real_values, y_real_values, delta_y_values,
    c=delta_y_values, cmap='viridis', marker='o'
)
cbar = plt.colorbar(sc, ax=ax, shrink=0.5, aspect=10)
cbar.set_label('挖高 (delta_y)')
ax.set_xlabel('X 坐标 (真实世界)')
ax.set_ylabel('Y 坐标 (真实世界)')
```

```

ax.set_zlabel('挖高 $\Delta y$ (delta_y)')
ax.set_title('真实点云及挖高分布')
plt.show()

plt.figure(figsize=(10, 6))
plt.contourf(x_mesh, y_mesh, delta_y_mesh,
levels=100, cmap='viridis')
plt.colorbar(label='挖高 $\Delta y$ (delta_y)')
plt.xlabel('X $\Delta$ 坐标(真实世界)')
plt.ylabel('Y $\Delta$ 坐标(真实世界)')
plt.title('挖高分布的二维俯视图')
plt.show()

total_excavation_volume = 0.0
for i in range(delta_y_mesh.shape[0]):
    for j in range(delta_y_mesh.shape[1]):
        excavation_height = delta_y_mesh[i, j]
        if not np.isnan(excavation_height)
            and excavation_height > 0:
            cell_area = x_step * y_step
            cell_volume = excavation_height * cell_area
            total_excavation_volume += cell_volum
print(f"\n估算的挖方总体积为:{total_excavation_volume:.3f}立方米")

```

附录六：第三问源代码

Listing 7: P_ant.py

```

import numpy as np
import matplotlib.pyplot as plt
import matplotlib.cm as cm
import logging
from logger_setup import setup_logger

plt.rcParams['font.sans-serif'] = ['SimHei']
plt.rcParams['axes.unicode_minus'] = False

logger = setup_logger()

```

```

class Ant:
    def __init__(self, num_vehicles, num_customers,
                 vehicle_capacity, alpha, beta,
                 rho, Q, times, prices, demands, prices_digging,
                 times_digging, pheromone,
                 time_weight=0.5, cost_weight=0.5, cluster_center=[0, 0]):
        self.routes = []
        for _ in range(num_vehicles): # 每辆车的路径
            self.visited = set() # 已访问客户
            self.loads = [0] * num_vehicles # 每辆车的当前载重
            self.times = [0] * num_vehicles # 每辆车的行驶时间
            self.costs = [0] * num_vehicles # 每辆车的行驶费用
            self.total_time = 0 # 总时间
            self.total_cost = 0 # 总代价
            self.total_weighted = 0 # 加权总目标值
            self.current_vehicle = 0 # 当前使用的车辆
            self.num_vehicles = num_vehicles
            self.num_customers = num_customers
            self.vehicle_capacity = vehicle_capacity
            self.alpha = alpha
            self.beta = beta
            self.rho = rho
            self.Q = Q
            self.time_weight = time_weight # 时间权重
            self.cost_weight = cost_weight # 花费权重
            self.times_matrix = times
            self.times_digging = times_digging
            self.prices_matrix = prices
            self.prices_digging = prices_digging
            self.pheromone = pheromone
            self.demands = demands
            self.cluster_center = cluster_center
    def build_route(self):
        # 所有车辆从仓库出发
        for v in range(self.num_vehicles):
            self.routes[v] = [0]
        self.visited = set()
        self.loads = [0] * self.num_vehicles
        self.current_vehicle = 0
        while len(self.visited) < self.num_customers:
            next_node = self.select_next_node()
            if next_node == 0: # 需要返回仓库补充

```

```

        self.routes[self.current_vehicle].append(0)
        self.loads[self.current_vehicle] = 0
        # 如果还有未使用的车辆，切换到新车辆
        if self.loads.count(0) > 1 and self.current_vehicle <
           self.num_vehicles - 1:
            unused_vehicles = [v for v in
                               range(self.num_vehicles) if
                               len(self.routes[v]) == 1]
            if unused_vehicles:
                self.current_vehicle = unused_vehicles[0]
        else:
            self.routes[self.current_vehicle].append(next_node)
            self.visited.add(next_node)
            self.loads[self.current_vehicle] +=
              self.demands[next_node-1]
        # 所有车辆最后返回仓库
        for v in range(self.num_vehicles):
            if self.routes[v][-1] != 0:
                self.routes[v].append(0)
        self.calculate_cost_and_time()
    def select_next_node(self):
        current_node = self.routes[self.current_vehicle][-1]
        # 找出尚未访问且当前车辆能载重的客户
        candidates = [j for j in range(1, self.num_customers+1)
                      if j not in self.visited and
                         self.demands[j-1] +
                         self.loads[self.current_vehicle]
                         <=
                         self.vehicle_capacity]
        if not candidates:
            # 如果有未使用的车辆且还有未访问的客户，考虑使用新车辆
            if len(self.visited) < self.num_customers
               and self.current_vehicle
               <
               self.num_vehicles - 1:
                unused_vehicles = [v for v in range(self.num_vehicles)
                                  if len(self.routes[v]) == 1]
                if unused_vehicles:
                    self.current_vehicle = unused_vehicles[0]
            return self.select_next_node()
        return 0 # 必须返回仓库

```

```

# 计算转移概率 - 同时考虑时间和花费
probabilities = []
total = 0
for j in candidates:
    pheromone_val = self.pheromone[current_node][j]
    # 考虑时间和花费的综合启发式信息
    time_heuristic = 1 / (self.times_matrix[current_node][j] +
                           self.times_digging[j] + 1e-10)
    cost_heuristic = 1 / (self.prices_matrix[current_node][j] +
                           self.prices_digging[j] + 1e-10)
    heuristic_val = self.time_weight * time_heuristic +
                    self.cost_weight *
                    cost_heuristic
    probabilities.append((pheromone_val**self.alpha) *
                          (heuristic_val**self.beta))
    total += probabilities[-1]
if total == 0:
    return np.random.choice(candidates)
probabilities = [p/total for p in probabilities]
return np.random.choice(candidates, p=probabilities)

def calculate_cost_and_time(self):
    self.total_cost = 0
    for v in range(self.num_vehicles):
        self.times[v] = 0
        self.costs[v] = 0
        route = self.routes[v]
        for i in range(len(route)-1):
            self.times[v] +=
                self.times_matrix[route[i]][route[i+1]]
            self.times[v] +=
                self.times_digging[route[i+1]]
            self.costs[v] +=
                self.prices_matrix[route[i]][route[i+1]]
            self.costs[v] +=
                self.prices_digging[route[i+1]]
    # 成本需要累加
    self.total_cost += self.costs[v]
    self.total_time = max(self.times) if self.times else 0
    self.total_weighted = self.time_weight * self.total_time +
                         self.cost_weight *
                         self.total_cost

```

```

def create_problem_data(num_customers, customer_coords, demands,
v, price_hour,
digging_v, area, cluster_center=[0,0]):
    """ 创建问题数据
参数：
    num_customers: 客户数量（挖方区域数量）
    customer_coords: 客户坐标数组，形状为(num_customers, 2)
    demands: 每个客户的需求量（挖方高度）
    v: 车辆行驶速度，单位为米/小时
    price_hour: 每小时运营成本，单位为元/小时
    digging_v: 挖掘速度，单位为立方米/小时
    area: 每个挖方区域的面积，用于计算挖掘体积
    cluster_center: 装卸点（仓库）坐标，默认为[0,0]
返回：
    customer_coords: 客户坐标
    demands: 客户需求量
    times: 行驶时间矩阵，表示从点i到点j所需的时间
    prices: 行驶成本矩阵，表示从点i到点j所需的成本
    times_digging: 挖掘时间数组，表示在每个点挖掘所需的时间
    prices_digging: 挖掘成本数组，表示在每个点挖掘所需的成本
"""
# 计算时间矩阵 - 从点i到点j的行驶时间
times = np.zeros((num_customers+1, num_customers+1))
for i in range(num_customers+1):
    for j in range(num_customers+1):
        if i == j:
            times[i,j] = 0 # 相同点之间的时间为0
        else:
            # 索引0表示装卸点（仓库），其他索引需要-1来匹配customer_coords
            x1, y1 = (cluster_center[0], cluster_center[1]) if i == 0 else
            customer_coords[i-1]
            x2, y2 = (cluster_center[0], cluster_center[1]) if j == 0 else
            customer_coords[j-1]
            # 计算欧氏距离并除以速度得到时间
            times[i,j] = np.sqrt((x1-x2)**2 + (y1-y2)**2) / v
# 计算行驶成本矩阵 - 时间乘以每小时成本
prices = times * price_hour
# 计算挖掘时间 - 体积(高度*面积)除以挖掘速度
times_digging = np.insert((demands*area)/digging_v, 0, 0)

```

```
# 在索引0处插入0（仓库不需要
# 挖掘）
times_digging = np.maximum(times_digging, 1e-6)
# 将负值或零值替换为一个很小的正数，避免
# 计算问题
# 计算挖掘成本 - 挖掘时间乘以每小时成本
prices_digging = np.insert(times_digging * price_hour, 0, 0)
# 在索引0处插入0
prices_digging = np.maximum(prices_digging, 1e-6)
# 确保没有负值
return customer_coords, demands, times, prices,
       times_digging, prices_digging
def start_simulation(max_iterations, num_ants,
                     num_vehicles, num_customers,
                     vehicle_capacity,
                     alpha, beta, rho, Q, customer_coords,
                     demands, times, prices,
                     times_digging, prices_digging,
                     time_weight=0.5, cost_weight=0.5,
                     if_show=True, cluster_center=
                     [0,0], id=0, cluster=0):
    """运行蚁群算法模拟，同时优化时间和花费
参数：
```

max_iterations: 最大迭代次数
num_ants: 蚂蚁数量
num_vehicles: 车辆数量
num_customers: 客户数量（挖方区域数量）
vehicle_capacity: 车辆容量
alpha: 信息素重要程度参数
beta: 启发式信息重要程度参数
rho: 信息素蒸发系数
Q: 信息素增强系数
customer_coords: 客户坐标数组，形状为(num_customers, 2)
demands: 客户需求量（挖方高度）
times: 行驶时间矩阵，表示从点i到点j所需的时间
prices: 行驶成本矩阵，表示从点i到点j所需的成本
times_digging: 挖掘时间数组，表示在每个点挖掘所需的时间
prices_digging: 挖掘成本数组，表示在每个点挖掘所需的成本
time_weight: 时间权重，默认为0.5
cost_weight: 成本权重，默认为0.5
if_show: 是否显示可视化结果，默认为True

返回：

无直接返回值，但会打印迭代过程中的最佳解，并可选择性地可视化结果

```

"""
# 初始化信息素矩阵
pheromone = np.ones((num_customers+1, num_customers+1))

# 蚁群算法主循环
best_time = float('inf')
best_cost = float('inf')
best_weighted = float('inf')
best_routes = None
history_time = []
history_cost = []
history_weighted = []

for iteration in range(max_iterations):
    ants = [Ant(num_vehicles, num_customers,
                vehicle_capacity, alpha, beta, rho,
                Q, times, prices, demands, prices_digging,
                times_digging, pheromone,
                time_weight, cost_weight, cluster_center)
            for _ in range(num_ants)]
    # 所有蚂蚁构建路径
    for ant in ants:
        ant.build_route()

    # 更新基于加权目标的最优解
    if ant.total_weighted < best_weighted:
        best_weighted = ant.total_weighted
        best_time = ant.total_time
        best_cost = ant.total_cost
        best_routes = [route.copy() for route in ant.routes]

    # 信息素挥发
    pheromone = pheromone * (1 - rho)

    # 信息素更新（精英蚂蚁策略）
    for ant in ants:
        # 根据加权目标更新信息素
        for v in range(num_vehicles):
            route = ant.routes[v]
            for i in range(len(route) - 1):
                from_node = route[i]
                to_node = route[i + 1]
                # 使用加权目标作为信息素增强的依据
                pheromone[from_node][to_node] +=
                    Q / ant.total_weighted
                history_time.append(best_time)

```

```
        history_cost.append(best_cost)
        history_weighted.append(best_weighted)
        logger.info(f"粒子{id}迭代{iteration+1},
最佳时间:{best_time:.2f},最佳代价:
{best_cost:.2f},加权值:{best_weighted:.2f}")
    if if_show:
        visualize_results(history_time, history_cost,
                           history_weighted, best_time,
                           best_cost, best_weighted, best_routes, customer_coords,
                           num_vehicles, cluster_center, id, cluster)
    return best_routes, best_time, best_cost, best_weighted
def visualize_results(history_time, history_cost,
                      history_weighted, best_time,
                      best_cost, best_weighted, best_routes, customer_coords,
                      num_vehicles, cluster_center, id, cluster):
    """ 可视化结果 """
    # 参数：
    history_time: 列表，每次迭代的最佳时间记录
    history_cost: 列表，每次迭代的最佳花费记录
    history_weighted: 列表，每次迭代的最佳加权值记录
    best_time: 浮点数，最终最佳时间
    best_cost: 浮点数，最终最佳花费
    best_routes: 列表，最佳路径方案
    customer_coords: 数组，客户坐标
    num_vehicles: 整数，车辆数量
    """
    plt.figure(figsize=(16, 8))
    plt.subplot(3, 2, 1)
    plt.plot(history_time, '#f9f871')
    plt.title('时间优化过程')
    plt.xlabel('迭代次数')
    plt.ylabel('最佳时间')
    plt.subplot(3, 2, 3)
    plt.plot(history_cost, '#ffc735')
    plt.title('花费优化过程')
    plt.xlabel('迭代次数')
    plt.ylabel('最佳花费')
    plt.subplot(3, 2, 5)
    plt.plot(history_weighted, '#ff9671')
    plt.title('加权目标优化过程')
    plt.xlabel('迭代次数')
    plt.ylabel('加权值')
```

```

plt.subplot(1, 2, 2)
plt.scatter([cluster_center[0]], [cluster_center[1]],
c='#FF6F91', s=100, label='仓库')
plt.scatter(customer_coords[:, 0], customer_coords[:, 1],
c='#845ec2', label='客户')

colors = cm.viridis(np.linspace(0, 1, num_vehicles))
for v in range(num_vehicles):
    if not best_routes[v] or len(best_routes[v]) <= 1:
        continue
    current_route = []
    for node in best_routes[v]:
        current_route.append((cluster_center[0], cluster_center[1]))
        if node == 0 else customer_coords[node - 1])
    x, y = zip(*current_route)
    plt.plot(x, y, '--', color=colors[v], alpha=0.7,
    label=f'车辆 {v+1}')
plt.title(f'最佳路径 (时间: {best_time:.2f}, 代价: {best_cost:.2f},\n加权: {best_weighted:.2f})')
plt.legend()
plt.tight_layout()
plt.savefig(f'最佳路径_时间{best_time:.2f}_\n代价{best_cost:.2f}_加权{best_weighted:.2f}_聚类{cluster}.pdf',
format='pdf', bbox_inches='tight')
# plt.show()

logger.info(f"\n粒子{id} 聚类{cluster} 各车辆路径详情：")
for v in range(num_vehicles):
    if best_routes[v]:
        logger.info(f"粒子{id} 车辆 {v+1} 路径: {best_routes[v]}")

def run_for_one_region(num_customers, vehicle_capacity, num_vehicles,
max_iterations,
num_ants,
alpha, beta, rho, customer_coords, demands_height,
time_weight=0.5, cost_weight=0.5, v=10, price_hour=100, digging_v=5,
area=16, if_show=True, cluster_center=[0, 0], id=0, cluster=0):
    """
    运行蚁群算法，对一个区域进行优化，同时考虑时间和花费
    参数：
    - num_customers: 客户数量（需要服务的点位数量）
    - vehicle_capacity: 车辆容量（每辆车最大装载量）
    - num_vehicles: 车辆数量（可用于配送的车辆总数）
    """

```

运行蚁群算法，对一个区域进行优化，同时考虑时间和花费

参数：

- num_customers: 客户数量（需要服务的点位数量）
- vehicle_capacity: 车辆容量（每辆车最大装载量）
- num_vehicles: 车辆数量（可用于配送的车辆总数）

```

- max_iterations: 最大迭代次数（算法运行的最大循环次数）
- num_ants: 蚂蚁数量（参与寻路的蚂蚁个体数）
- alpha: 信息素重要程度参数（控制信息素对蚂蚁决策的影响）
- beta: 启发式信息重要程度参数（控制距离等启发式因素对决策的影响）
- rho: 信息素蒸发系数（每次迭代后信息素衰减的比例）
- Q: 信息素增加强度系数（蚂蚁完成路径后释放信息素的量）
- customer_coords: 客户坐标列表（每个客户点的位置坐标）
- demands_height: 需求高度列表（每个挖方点的高度值）
- time_weight: 时间权重（0-1之间，优化目标中时间因素的权重）
- cost_weight: 花费权重（0-1之间，优化目标中成本因素的权重）
- v: 车辆行驶速度（单位：米/小时，影响行驶时间计算）
- price_hour: 每小时运营成本（单位：元/小时，影响总成本计算）
- digging_v: 挖掘速度（单位：立方米/小时，影响挖掘时间计算）
- area: 挖掘面积（单位：平方米，用于计算每个点的挖掘体积）
- if_show: 是否显示结果（控制是否展示优化过程和结果图表）
- cluster_center: 聚类中心坐标（当前区域的中心点坐标，默认为原点[0,0]）
"""

# 创建问题数据
customer_coords, demands, times, prices,
times_digging,
prices_digging = create_problem_data(num_customers,
customer_coords, demands_height,
v, price_hour,
digging_v, area, cluster_center)
# 运行蚁群算法
best_routes, best_time, best_cost, best_weighted = start_simulation(
    max_iterations, num_ants, num_vehicles, num_customers,
    vehicle_capacity, alpha, beta, rho, Q, customer_coords,
    demands, times, prices, times_digging, prices_digging,
    time_weight, cost_weight, if_show, cluster_center, id, cluster)
return best_routes, best_time, best_cost, best_weighted

```

Listing 8: P3_best_dispatch_river.py

```

from P1_river_earth import *
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.patches import Rectangle
from matplotlib.colors import Normalize
from sklearn.cluster import KMeans
from sklearn.preprocessing import StandardScaler

```

```

import pandas as pd
import yaml
from P3_ant import run_for_one_region
import logging
from logger_setup import setup_logger

plt.rcParams['font.sans-serif'] = ['SimHei']
plt.rcParams['axes.unicode_minus'] = False

logger = setup_logger()
def cluster_earthwork(earth_result, n_clusters=10,
height_threshold=0.5):
    """
    对挖方区域数据进行聚类分析，确定最佳装卸点位置
    参数：
        n_clusters: 聚类数量，即希望得到的装卸点数量
    返回：
        cluster_centers: 聚类中心点坐标，可作为装卸点位置
        labels: 每个挖方区域所属的聚类
        earth_points: 挖方区域的中心点坐标和高度数据
    """
    centers_x = []
    centers_y = []
    heights = []
    for i in range(len(earth_result["square_point"])):
        x_coords = [x - X_OFFSET for x
                    in earth_result["square_point"][i][0]]
        y_coords = [y - Y_OFFSET for y
                    in earth_result["square_point"][i][1]]
        center_x = sum(x_coords) / len(x_coords)
        center_y = sum(y_coords) / len(y_coords)
        centers_x.append(center_x)
        centers_y.append(center_y)
        heights.append(earth_result["height"][i])
    X = np.column_stack((centers_x, centers_y, heights))
    scaler = StandardScaler()
    X_scaled = scaler.fit_transform(X)
    X_scaled[:, 2] = X_scaled[:, 2] * height_threshold
    kmeans = KMeans(n_clusters=n_clusters, random_state=42)
    labels = kmeans.fit_predict(X_scaled)
    cluster_centers_scaled = kmeans.cluster_centers_
    cluster_centers = scaler.inverse_transform(cluster_centers_scaled)

```

```
earth_points = pd.DataFrame({  
    'x': centers_x,  
    'y': centers_y,  
    'height': heights,  
    'cluster': labels  
})  
return cluster_centers, labels, earth_points  
def vis_earthwork(n_clusters=7, show_centers=True,  
height_threshold=0.5, if_show=True):  
    """  
    可视化挖方区域，将挖高设色和聚类结果分开显示  
    """  
    earth_result = get_earthwork()  
    n_clusters=int(n_clusters)  
    cluster_centers, labels, earth_points =  
        cluster_earthwork(earth_result, n_clusters, height_threshold)  
    if if_show:  
        fig, (ax1, ax2) = plt.subplots(2, 1, figsize=(16, 8),  
            constrained_layout=True)  
        cmap = plt.cm.viridis  
        norm = Normalize(vmin=min(earth_result["height"]),  
            vmax=max(earth_result["height"]))  
        cluster_colors = plt.cm.Set1(np.linspace(0, 1, n_clusters))  
        for i in range(len(earth_result["square_point"])):)  
            x_coords = [x - X_OFFSET for  
                x in earth_result["square_point"][i][0]]  
            y_coords = [y - Y_OFFSET for  
                y in earth_result["square_point"][i][1]]  
            height = max(0, earth_result["height"][i])  
            fill_color = cmap(norm(height))  
            polygon_xy = list(zip(y_coords, x_coords))  
            polygon = plt.Polygon(polygon_xy, closed=True,  
                facecolor=fill_color, edgecolor='black',  
                alpha=0.7, linewidth=0.8)  
            ax1.add_patch(polygon)  
  
            norm_colorbar = Normalize(vmin=0,  
                vmax=max(earth_result["height"]))  
            cbar1 = plt.colorbar(plt.cm.ScalarMappable  
                (norm=norm_colorbar, cmap=cmap),  
                ax=ax1)
```

```
cbar1.set_label('挖方高度')

ax1.set_title('挖方区域高度热图')
ax1.set_xlabel('Y下坐标', fontsize=18)
ax1.set_ylabel('X下坐标', fontsize=18)
ax1.tick_params(axis='both', labelsize=18)
ax1.axis('equal')
for i in range(len(earth_result["square_point"])):
    x_coords = [x - X_OFFSET for x
                 in earth_result["square_point"][i][0]]
    y_coords = [y - Y_OFFSET for y
                 in earth_result["square_point"][i][1]]
    fill_color = cluster_colors[labels[i]]
    polygon_xy = list(zip(y_coords, x_coords))
    polygon = plt.Polygon(polygon_xy, closed=True,
                          facecolor=fill_color, edgecolor='black',
                          alpha=0.7, linewidth=0.8)
    ax2.add_patch(polygon)
if show_centers:
    for i, center in enumerate(cluster_centers):
        ax2.scatter(center[1], center[0], s=200, c='red',
                    marker='X',
                    edgecolor='black', linewidth=2)
        ax2.annotate(f'装卸点{i+1}', (center[1], center[0]),
                     xytext=(10, 10), textcoords='offset points',
                     fontsize=12, fontweight='bold')
ax2.set_title('挖方区域聚类分析与装卸点位置')
ax2.set_xlabel('Y下坐标', fontsize=18)
ax2.set_ylabel('X下坐标', fontsize=18)
ax2.tick_params(axis='both', labelsize=18)
handles = [plt.Line2D([0], [0], color=cluster_colors[i], lw=4,
                      label=f'聚类{i+1}') for i in
            range(n_clusters)]
ax2.legend(handles=handles, loc='lower right', ncol=3
ax2.axis('equal')

plt.suptitle('挖方区域分析', fontsize=16)
plt.savefig('挖方区域分析11.pdf', format='pdf',
            bbox_inches='tight')

return cluster_centers, earth_points
def analyze_loading_points(n_clusters=7, height_threshold=0.5,
                           if_show=True):
```

""" 分析装卸点的详细信息 """

```
cluster_centers, earth_points = vis_earthwork(n_clusters=n_clusters,
height_threshold=height_threshold, if_show=if_show)
print("\n装卸点详细信息:")
print("-" * 50)
for i, center in enumerate(cluster_centers):
    cluster_data = earth_points[earth_points['cluster'] == i]
    total_height = cluster_data['height'].sum()
    avg_height = cluster_data['height'].mean()
    point_count = len(cluster_data)

    print(f"\n装卸点 {i+1}:")
    print(f"坐标 : X={center[0]:.2f}, Y={center[1]:.2f}")
    print(f"覆盖挖方区域数量 : {point_count}")
    print(f"平均挖方高度 : {avg_height:.2f}")
    print(f"总体挖方高度 : {total_height:.2f}")

return cluster_centers, earth_points
```

def run_for_simulation_river(n_clusters, height_threshold, if_show, num_vehicles, split_mode, vehicle_capacity, max_iterations, num_ants, alpha, beta, rho, Q, time_weight, cost_weight, v, price_hour, digging_v, id):

"""

运行蚁群算法，对所有挖方区域进行优化配送路径规划

参数：

- n_clusters: 聚类数量（装卸点数量）
- height_threshold: 挖方高度阈值，用于筛选有效挖方点
- if_show: 是否显示可视化结果
- num_vehicles: 总车辆数量
- split_mode: 车辆分配模式，可选”ratio”（按比例分配）或”random”（随机分配）
- vehicle_capacity: 每辆车的装载容量
- max_iterations: 蚁群算法最大迭代次数
- num_ants: 蚂蚁数量
- alpha: 信息素重要程度参数
- beta: 启发式信息重要程度参数
- rho: 信息素蒸发系数
- Q: 信息素增加强度系数
- time_weight: 时间权重（优化目标中时间因素的权重）
- cost_weight: 成本权重（优化目标中成本因素的权重）
- v: 车辆行驶速度（米/小时）
- price_hour: 每小时运营成本（元/小时）

- digging_v: 挖掘速度（立方米/小时）
- id: 粒子编号

返回：

- 包含优化结果的字典，包括各区域的时间、成本、加权得分和路径

```
"""
results={
    "time":[] ,
    "cost":[] ,
    "weighted":[] ,
    "routes":[]
}
cluster_centers , earth_points = vis_earthwork(n_clusters=n_clusters ,
height_threshold=height_threshold , if_show=if_show)
total_vehicles = int(num_vehicles)
cluster_heights = []
for i in range(len(cluster_centers)):
    cluster_data = earth_points[earth_points['cluster'] == i]
    total_height = cluster_data['height'].sum()
    cluster_heights.append(total_height)
allocation_modes = {
    "随机分配": [],
    "按比例分配": []
}
remaining_vehicles = total_vehicles - len(cluster_centers)
random_allocation = [1] * len(cluster_centers)
if remaining_vehicles > 0:
    random_indices = np.random.choice(len(cluster_centers) ,
    remaining_vehicles ,
    replace=True)
    for idx in random_indices:
        random_allocation[idx] += 1
allocation_modes["随机分配"] = random_allocation
proportional_allocation = [1] * len(cluster_centers)
remaining_vehicles = total_vehicles - len(cluster_centers)
if remaining_vehicles > 0 and sum(cluster_heights) > 0:
    height_proportions = [h / sum(cluster_heights) for h in
    cluster_heights]
    vehicle_distribution = [int(p * remaining_vehicles) for p in
    height_proportions]

while sum(vehicle_distribution) < remaining_vehicles:
```

```

errors = [(p * remaining_vehicles) - v for p, v in
          zip(height_proportions,
              vehicle_distribution)]
idx = errors.index(max(errors))
vehicle_distribution[idx] += 1
for i in range(len(cluster_centers)):
    proportional_allocation[i] += vehicle_distribution[i]
allocation_modes["按比例分配"] = proportional_allocation

logger.info(f"粒子{id}车辆分配方案:")
logger.info("-" * 50)
for mode, allocation in allocation_modes.items():
    logger.info(f"\n{mode}:")
    for i, num in enumerate(allocation):
        logger.info(f"\t粒子{id}聚类{i+1}:{num}辆车")
for i in range(len(cluster_centers)):
    cluster_data = earth_points[earth_points['cluster'] == i]
    if split_mode == "ratio":
        vehicles_for_cluster = allocation_modes["按比例分配"][i]
    elif split_mode == "random":
        vehicles_for_cluster = allocation_modes["随机分配"][i]
    logger.info(f"\n正在为粒子{id}聚类{i+1}运行算法"
               f"(分配{vehicles_for_cluster}辆车)...")
best_routes, best_time, best_cost,
best_weighted=run_for_one_region(num_customers=len(cluster_data),
                                   vehicle_capacity=vehicle_capacity,
                                   num_vehicles=vehicles_for_cluster,
                                   max_iterations=max_iterations,
                                   num_ants=num_ants,
                                   alpha=alpha,
                                   beta=beta,
                                   rho=rho,
                                   Q=Q,
                                   customer_coords=
                                   cluster_data[['x', 'y']].values,
                                   demands_height=cluster_data['height'].values,
                                   time_weight=time_weight,
                                   cost_weight=cost_weight,
                                   if_show=if_show,
                                   v=v,
                                   price_hour=price_hour,
                                   )

```

```

        digging_v=digging_v ,
        cluster_center=cluster_centers[ i ] ,
        id=id ,
        cluster=i+1)
    results [ "time" ].append( best_time )
    results [ "cost" ].append( best_cost )
    results [ "weighted" ].append( best_weighted )
    results [ "routes" ].append( best_routes )
    return results
def load_config(cfg_path):
    with open(cfg_path, 'r') as file:
        cfg = yaml.safe_load(file)
    return cfg
if __name__ == "__main__":
    cfg = load_config('config.yaml')
    run_for_simulation_river(cfg)

```

Listing 9: P3best_dispatchroad.py

```

from P2_road_earth import *
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.patches import Rectangle
from matplotlib.colors import Normalize
from sklearn.cluster import KMeans
from sklearn.preprocessing import StandardScaler
import pandas as pd
import yaml
from P3_ant import run_for_one_region
import logging
from logger_setup import setup_logger

plt.rcParams[ 'font.sans-serif' ] = [ 'SimHei' ]
plt.rcParams[ 'axes.unicode_minus' ] = False

logger = setup_logger()
X_OFFSET=0
Y_OFFSET=0
def cluster_earthwork(earth_result, n_clusters=10,
height_threshold=0.5):
    """

```

对挖方区域数据进行聚类分析，确定最佳装卸点位置

参数：

`earth_result`: 三列数组，包含x坐标、y坐标和挖高
`n_clusters`: 聚类数量，即希望得到的装卸点数量
`height_threshold`: 高度阈值权重

返回：

`cluster_centers`: 聚类中心点坐标，可作为装卸点位置
`labels`: 每个挖方区域所属的聚类
`earth_points`: 挖方区域的中心点坐标和高度数据

”””

```
centers_x = earth_result[:, 0] - X_OFFSET
centers_y = earth_result[:, 1] - Y_OFFSET
heights = earth_result[:, 2]
X = np.column_stack((centers_x, centers_y, heights))
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)
X_scaled[:, 2] = X_scaled[:, 2] * height_threshold
kmeans = KMeans(n_clusters=n_clusters, random_state=42)
labels = kmeans.fit_predict(X_scaled)
cluster_centers_scaled = kmeans.cluster_centers_
cluster_centers = scaler.inverse_transform(cluster_centers_scaled)
earth_points = pd.DataFrame({
    'x': centers_x,
    'y': centers_y,
    'height': heights,
    'cluster': labels
})
return cluster_centers, labels, earth_points
```

`def vis_earthwork(n_clusters=7, show_centers=True,`

`height_threshold=0.5, if_show=True):`

”””

可视化挖方区域，将挖高设色和聚类结果分开显示

”””

```
earth_result = get_road_earth()
n_clusters=int(n_clusters)
cluster_centers, labels, earth_points =
cluster_earthwork(earth_result,
n_clusters, height_threshold)
if if_show:
    fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(16, 8),
    constrained_layout=True)
    cmap = plt.cm.viridis
    norm = Normalize(vmin=min(earth_points['height']),
```

```
vmax=max(earth_points[ 'height' ]))  
cluster_colors = plt.cm.Set1(np.linspace(0, 1, n_clusters))  
for _, row in earth_points.iterrows():  
    x, y = row[ 'x' ], row[ 'y' ]  
    square_vertices = [  
        (x - 2.0, y - 2.0),  
        (x + 2.0, y - 2.0),  
        (x + 2.0, y + 2.0),  
        (x - 2.0, y + 2.0)  
    ]  
    height = max(0, row[ 'height' ])  
    fill_color = cmap(norm(height))  
    polygon = plt.Polygon(square_vertices, closed=True,  
                          facecolor=fill_color, edgecolor='black',  
                          alpha=0.7, linewidth=0.5)  
    ax1.add_patch(polygon)  
norm_colorbar = Normalize(vmin=0, vmax=max(earth_points[ 'height' ]))  
cbar1 = plt.colorbar(plt.cm.ScalarMappable(norm=norm_colorbar,  
                                         cmap=cmap),  
                     ax=ax1)  
cbar1.set_label('挖方高度')  
  
ax1.set_title('挖方区域高度热图')  
ax1.set_xlabel('XU坐标', fontsize=18)  
ax1.set_ylabel('YU坐标', fontsize=18)  
ax1.tick_params(axis='both', labelsize=18)  
ax1.axis('equal')  
for _, row in earth_points.iterrows():  
    x, y = row[ 'x' ], row[ 'y' ]  
    cluster_id = int(row[ 'cluster' ])  
    square_vertices = [  
        (x - 2.0, y - 2.0),  
        (x + 2.0, y - 2.0),  
        (x + 2.0, y + 2.0),  
        (x - 2.0, y + 2.0)  
    ]  
    fill_color = cluster_colors[cluster_id]  
    polygon = plt.Polygon(square_vertices, closed=True,  
                          facecolor=fill_color, edgecolor='black',  
                          alpha=0.7, linewidth=0.5)  
    ax2.add_patch(polygon)  
if show_centers:
```

```

        for i, center in enumerate(cluster_centers):
            ax2.scatter(center[0], center[1], s=200, c='red',
                        marker='X',
                        edgecolor='black', linewidth=2)
            ax2.annotate(f'装卸点{i+1}', (center[0], center[1]),
                        xytext=(10, 10), textcoords='offset points',
                        fontsize=12, fontweight='bold')
        ax2.set_title('挖方区域聚类分析与装卸点位置')
        ax2.set_xlabel('X坐标', fontsize=18)
        ax2.set_ylabel('Y坐标', fontsize=18)
        ax2.tick_params(axis='both', labelsize=18)
        handles = [plt.Line2D([0], [0], color=cluster_colors[i], lw=4,
                             label=f'聚类{i+1}') for i in
                   range(n_clusters)]
        ax2.legend(handles=handles, loc='lower right', ncol=3)
        ax2.axis('equal')

    plt.suptitle('挖方区域分析', fontsize=16)
    plt.savefig('挖方区域分析11.pdf', format='pdf',
                bbox_inches='tight')
    plt.show()

    return cluster_centers, earth_points
def analyze_loading_points(n_clusters=7, height_threshold=0.5,
                           if_show=True):
    """分析装卸点的详细信息"""
    cluster_centers, earth_points = vis_earthwork(n_clusters=n_clusters,
                                                   height_threshold=height_threshold,
                                                   if_show=if_show)
    print("\n装卸点详细信息:")
    print("-" * 50)
    for i, center in enumerate(cluster_centers):
        cluster_data = earth_points[earth_points['cluster'] == i]
        total_height = cluster_data['height'].sum()
        avg_height = cluster_data['height'].mean()
        point_count = len(cluster_data)
        print(f"\n装卸点{i+1}:")
        print(f"  坐标:X={center[0]:.2f}, Y={center[1]:.2f}")
        print(f"  覆盖挖方区域数量:{point_count}")
        print(f"  平均挖方高度:{avg_height:.2f}")
        print(f"  总体挖方高度:{total_height:.2f}")

    return cluster_centers, earth_points
def run_for_simulation_road(n_clusters, height_threshold, if_show,
                           num_vehicles,

```

```

split_mode, vehicle_capacity, max_iterations,
        num_ants, alpha, beta, rho, Q, time_weight,
        cost_weight, v,
        price_hour, digging_v, id):
"""

```

运行蚁群算法，对所有挖方区域进行优化配送路径规划

参数：

- n_clusters: 聚类数量（装卸点数量）
- height_threshold: 挖方高度阈值，用于筛选有效挖方点
- if_show: 是否显示可视化结果
- num_vehicles: 总车辆数量
- split_mode: 车辆分配模式，可选”ratio”（按比例分配）或”random”（随机分配）
- vehicle_capacity: 每辆车的装载容量
- max_iterations: 蚁群算法最大迭代次数
- num_ants: 蚂蚁数量
- alpha: 信息素重要程度参数
- beta: 启发式信息重要程度参数
- rho: 信息素蒸发系数
- Q: 信息素增加强度系数
- time_weight: 时间权重（优化目标中时间因素的权重）
- cost_weight: 成本权重（优化目标中成本因素的权重）
- v: 车辆行驶速度（米/小时）
- price_hour: 每小时运营成本（元/小时）
- digging_v: 挖掘速度（立方米/小时）
- id: 粒子编号

返回：

- 包含优化结果的字典，包括各区域的时间、成本、加权得分和路径

"""

```

results={
    "time":[] ,
    "cost":[] ,
    "weighted":[] ,
    "routes":[]
}
cluster_centers, earth_points = vis_earthwork(n_clusters=n_clusters,
height_threshold=height_threshold, if_show=if_show)
total_vehicles = int(num_vehicles)
cluster_heights = []
for i in range(len(cluster_centers)):
    cluster_data = earth_points[earth_points['cluster'] == i]

```

```

total_height = cluster_data[ 'height' ].sum()
cluster_heights.append(total_height)

allocation_modes = {
    "随机分配": [],
    "按比例分配": []
}

remaining_vehicles = total_vehicles - len(cluster_centers)
random_allocation = [1] * len(cluster_centers)

if remaining_vehicles > 0:
    random_indices = np.random.choice(len(cluster_centers),
        remaining_vehicles,
        replace=True)
    for idx in random_indices:
        random_allocation[idx] += 1

allocation_modes["随机分配"] = random_allocation

proportional_allocation = [1] * len(cluster_centers)
remaining_vehicles = total_vehicles - len(cluster_centers)

if remaining_vehicles > 0 and sum(cluster_heights) > 0:
    height_proportions = [h / sum(cluster_heights) for h in
        cluster_heights]
    vehicle_distribution = [int(p * remaining_vehicles) for p in
        height_proportions]

    while sum(vehicle_distribution) < remaining_vehicles:
        errors = [(p * remaining_vehicles) - v for p, v in
            zip(height_proportions,
            vehicle_distribution)]
        idx = errors.index(max(errors))
        vehicle_distribution[idx] += 1

    for i in range(len(cluster_centers)):
        proportional_allocation[i] += vehicle_distribution[i]

allocation_modes["按比例分配"] = proportional_allocation

logger.info(f"粒子{id}车辆分配方案:")
logger.info("-" * 50)

for mode, allocation in allocation_modes.items():
    logger.info(f"\n{mode}:")
    for i, num in enumerate(allocation):
        logger.info(f" 粒子{id}聚类{i+1}:{num}辆车")

for i in range(len(cluster_centers)):
    cluster_data = earth_points[earth_points['cluster'] == i]

    if split_mode == "ratio":
        vehicles_for_cluster = allocation_modes["按比例分配"][i]

```

```
    elif split_mode == "random":
        vehicles_for_cluster = allocation_modes["随机分配"][i]

        logger.info(f"\n正在为粒子 {id} 聚类 {i+1} 运行算法 (分配
{vehicles_for_cluster} 辆
车) ...")

        best_routes, best_time, best_cost,
        best_weighted=run_for_one_region(num_customers=len(cluster_data),
                                          vehicle_capacity=vehicle_capacity,
                                          num_vehicles=vehicles_for_cluster,
                                          max_iterations=max_iterations,
                                          num_ants=num_ants,
                                          alpha=alpha,
                                          beta=beta,
                                          rho=rho,
                                          Q=Q,
                                          customer_coords
                                          =cluster_data[['x', 'y']].values,
                                          demands_height=cluster_data['height'].values,
                                          time_weight=time_weight,
                                          cost_weight=cost_weight,
                                          if_show=if_show,
                                          v=v,
                                          price_hour=price_hour,
                                          digging_v=digging_v,
                                          cluster_center=cluster_centers[i],
                                          id=id)

        results["time"].append(best_time)
        results["cost"].append(best_cost)
        results["weighted"].append(best_weighted)
        results["routes"].append(best_routes)

    return results

def load_config(cfg_path):
    with open(cfg_path, 'r') as file:
        cfg = yaml.safe_load(file)

    return cfg

if __name__ == "__main__":
    cfg = load_config('config.yaml')
    run_for_simulation_road(cfg)
```

Listing 10: P3_fitness_calculator.py

```

import numpy as np
from P3_best_dispatch_road import run_for_simulation_road
from P3_best_dispatch_river import run_for_simulation_river

def calculate_fitness(position, cfg, id):
    """
    计算给定位置的适应度值
    参数：
        position: 解空间中的位置向量
    返回：
        float: 适应度值
    """
    if cfg['problem_type'] == "river":
        results=run_for_simulation_river(n_clusters=position[1],
        height_threshold=position[2], if_show=cfg['if_show'],
        num_vehicles=position[0], split_mode
        =cfg['split_mode'], vehicle_capacity=cfg['vehicle_capacity'],
        max_iterations=cfg['max_ite
        rations'], num_ants=cfg['num_ants'], alpha=cfg['alpha'], beta=cfg['beta'],
        rho=cfg['rho'], Q
        =cfg['Q'], time_weight=cfg['time_weight'], cost_weight=cfg['cost_weight'],
        v=cfg['v'], pric
        e_hour=cfg['price_hour'], digging_v=cfg['digging_v'], id=id)

    elif cfg['problem_type'] == "road":
        results=run_for_simulation_road(n_clusters=position[1],
        height_threshold=position[2], if_show=cfg['if_show'], num_vehicles=position[0],
        split_mode=cfg['split_mode'], vehicle_capacity
        =cfg['vehicle_capacity'], max_iterations=cfg['max_iterations'], num_ants=cfg['num_ants'])

    # 计算适应度值
    costs = sum(results['cost'])
    time = max(results['time'])
    fitness = cfg['cost_weight']*costs + cfg['time_weight']*time
    return fitness

```

Listing 11: P3_particle_parr.py

```

import numpy as np
import matplotlib.pyplot as plt
from typing import Callable, Tuple, List, Optional
# 导入适应度计算接口

```

```

from P3_fitness_calculator import calculate_fitness
import yaml
# 导入并行处理模块
import multiprocessing
from concurrent.futures import ProcessPoolExecutor, as_completed
# 导入日志模块
# 使用中文字体
plt.rcParams['font.sans-serif'] = ['SimHei'] # 用来正常显示中文标签
plt.rcParams['axes.unicode_minus'] = False # 用来正常显示负号
# 从particle_parr.py中移除setup_logger函数，改为导入
from logger_setup import setup_logger
logger = setup_logger()
# 定义用于并行计算适应度的函数
def evaluate_fitness(particle_position, cfg, id):
    """ 并行计算适应度的函数 """
    return calculate_fitness(particle_position, cfg, id)
class Particle:
    """ 粒子类，代表搜索空间中的一个候选解 """
    def __init__(self, dim: int, bounds: List[Tuple[float, float]], initial_position: Optional[np.ndarray] = None, integer_dims: List[int] = None):
        """
        初始化一个粒子
        参数：
            dim: 问题的维度
            bounds: 每个维度的取值范围，格式为 [(min_1, max_1), (min_2, max_2), ...]
            initial_position: 可选的初始位置，如果提供则使用此位置，否则随机初始化
            integer_dims: 需要取整数值的维度列表，例如 [0, 1] 表示第0维和第1维需要是整数
        """
        # 存储哪些维度需要是整数
        self.integer_dims = [] if integer_dims is None else integer_dims
        # 初始化位置，如果提供了初始位置则使用，否则随机初始化
        if initial_position is not None:
            self.position = initial_position.copy()
        else:
            self.position = np.array([np.random.uniform(low, high) for low, high in bounds])
        # 将指定维度转为整数

```

```

    for dim_idx in self.integer_dims:
        self.position[dim_idx] = int(self.position[dim_idx])
    # 随机初始化速度（位置范围的10%作为速度范围）
    self.velocity = np.array([np.random.uniform(-0.1*(high-low),
                                                0.1*(high-low))
    for low, high in bounds])
    # 粒子的当前适应度值
    self.fitness = float('inf') # 初始化为无穷大（假设是最小化问题）
    # 粒子历史最佳位置
    self.best_position = self.position.copy()
    # 确保 best_position 中的整数维度也是整数类型
    for dim_idx in self.integer_dims:
        self.best_position[dim_idx] = int(self.best_position
                                          [dim_idx])
    # 粒子历史最佳适应度
    self.best_fitness = float('inf')
    # 用于剪枝的属性
    self.poor_performance_count = 0 # 连续表现不佳的次数

class ParticleSwarmOptimization:
    """粒子群优化算法"""
    def __init__(self,
                 dim: int,
                 bounds: List[Tuple[float, float]],
                 num_particles: int = 30,
                 max_iter: int = 100,
                 w: float = 0.7, # 惯性权重
                 c1: float = 1.5, # 认知系数
                 c2: float = 1.5, # 社会系数
                 minimize: bool = True,
                 initial_values=None,
                 integer_dims=None,
                 cfg=None,
                 early_stop: bool = False, # 是否启用早停机制
                 early_stop_iter: int = 20, # 连续多少次迭代无改善时停止
                 early_stop_tol: float = 1e-6, # 改善的最小阈值
                 dynamic_params: bool = False, # 是否启用参数动态调整
                 w_strategy: str = "linear", # 惯性权重调整策略：linear
                                              , nonlinear,
                                              chaotic
                 w_min: float = 0.4, # 惯性权重最小值

```

```

w_max: float = 0.9,          # 惯性权重最大值
c_strategy: str = "constant", # 学习因子调整策略：
constant, adaptive
pruning: bool = False,       # 是否启用剪枝
pruning_start: int = 20,      # 从第几次迭代开始剪枝
pruning_threshold: float = 0.5, # 剪枝阈值（相对于
全局最佳的差距比例）
pruning_count: int = 10,      # 连续多少次表现不佳时剪枝
min_particles: int = 5,       # 保留的最小粒子数量
parallel: bool = True,       # 是否启用并行计算
n_processes: int = None):   # 并行进程数量，None表示
使用CPU核心数
"""

```

初始化PSO算法

参数：

dim: 搜索空间的维度
 bounds: 每个维度的取值范围
 num_particles: 粒子数量
 max_iter: 最大迭代次数
 w: 惯性权重
 c1: 认知系数（个体学习因子）
 c2: 社会系数（群体学习因子）
 minimize: 是否为最小化问题（True为最小化，False为最大化）
 initial_values: 每个维度的初始值，如果提供，第一个粒子会使用这些值初始化
 integer_dims: 需要取整数值的维度列表，例如[0, 1]表示第0维和第1维需要是整数
 early_stop: 是否启用早停机制
 early_stop_iter: 早停条件，连续多少次迭代无改善时停止
 early_stop_tol: 早停条件，改善小于此值视为无改善
 dynamic_params: 是否启用参数动态调整
 w_strategy: 惯性权重调整策略
 w_min: 惯性权重最小值（用于线性和非线性策略）
 w_max: 惯性权重最大值（用于线性和非线性策略）
 c_strategy: 学习因子调整策略
 pruning: 是否启用剪枝功能
 pruning_start: 从第几次迭代开始剪枝
 pruning_threshold: 剪枝阈值（相对于全局最佳的差距比例）
 pruning_count: 连续多少次表现不佳时剪枝
 min_particles: 保留的最小粒子数量
 parallel: 是否启用并行计算
 n_processes: 并行进程数量，None表示使用CPU核心数

```
"""
# 移除了 objective_func 参数，将从外部接口获取适应度
self.dim = dim
self.bounds = bounds
self.num_particles = num_particles
self.max_iter = max_iter
self.w = w
self.c1 = c1
self.c2 = c2
self.minimize = minimize
self.integer_dims = [] if integer_dims is None else integer_dims
self.cfg = cfg

# 早停相关参数
self.early_stop = early_stop
self.early_stop_iter = early_stop_iter
self.early_stop_tol = early_stop_tol
# 参数动态调整相关参数
self.dynamic_params = dynamic_params
self.w_strategy = w_strategy
self.w_min = w_min
self.w_max = w_max
self.c_strategy = c_strategy
# 如果启用动态参数，初始化惯性权重为最大值
if self.dynamic_params and self.w_strategy in
[ "linear", "nonlinear" ]:
    self.w = self.w_max
# 混沌映射参数（用于chaotic策略）
self.chaotic_z = 0.7 # 初始值在(0,1)之间
# 初始化粒子群
self.particles = []
# 如果提供了初始值，第一个粒子使用初始值，其余随机初始化
if initial_values is not None:
    initial_position = np.array(initial_values)
    # 确保初始位置在边界内
    for i in range(dim):
        low, high = bounds[i]
        initial_position[i] = max(low, min(high,
        initial_position[i]))
    # 如果是整数维度，则取整
    if i in self.integer_dims:
        initial_position[i] = int(initial_position[i])
```

```
# 第一个粒子使用初始值
    self.particles.append(Particle(dim, bounds, initial_position,
                                    self.integer_dims))
# 其余粒子随机初始化
    for _ in range(1, num_particles):
        self.particles.append(Particle(dim, bounds, None,
                                        self.integer_dims))
else:
    # 所有粒子随机初始化
    self.particles = [Particle(dim, bounds, None,
                               self.integer_dims) for _ in
                      range(num_particles)]
# 初始化全局最佳
    self.global_best_position = None
    self.global_best_fitness = float('inf') if minimize
    else float('-inf')
# 记录每次迭代的最佳适应度，用于后续分析
    self.fitness_history = []
# 剪枝相关参数
    self.pruning = pruning
    self.pruning_start = pruning_start
    self.pruning_threshold = pruning_threshold
    self.pruning_count = pruning_count
    self.min_particles = min_particles
# 并行计算相关参数
    self.parallel = parallel
    self.n_processes = n_processes if n_processes is not None
    else multiprocessing.cpu_count()
def update_params(self, iter_num):
    """
    根据当前迭代次数更新算法参数
    参数：
        iter_num：当前迭代次数
    """
    if not self.dynamic_params:
        return

    # 更新惯性权重
    if self.w_strategy == "linear":
        # 线性递减惯性权重
        self.w = self.w_max - (self.w_max - self.w_min) *
            iter_num / self.max_iter
```

```

    elif self.w_strategy == "nonlinear":
        # 非线性递减惯性权重（二次递减）
        self.w = self.w_min + (self.w_max - self.w_min) *
            ((self.max_iter -
            iter_num) / self.max_iter) ** 2
    elif self.w_strategy == "chaotic":
        # 使用 Logistic 映射生成混沌序列
        self.chaotic_z = 4 * self.chaotic_z * (1 - self.chaotic_z)
        self.w = self.w_min + (self.w_max - self.w_min) *
            self.chaotic_z
    # 更新学习因子
    if self.c_strategy == "adaptive":
        # 自适应调整学习因子
        progress_ratio = iter_num / self.max_iter
        # 随着迭代进行，认知系数减小，社会系数增大
        self.c1 = 2.5 - 2 * progress_ratio
        self.c2 = 0.5 + 2 * progress_ratio
def prune_particles(self, iter_num):
    """
剪枝表现不佳的粒子
参数：
    iter_num: 当前迭代次数
    """
    if not self.pruning or iter_num < self.pruning_start or
        len(self.particles) <=
        self.min_particles:
        return

    # 计算适应度差距阈值
    if self.minimize:
        # 最小化问题
        base_fitness = self.global_best_fitness
        threshold = base_fitness * (1 + self.pruning_threshold)
        # 检查并更新每个粒子的表现计数
        for particle in self.particles:
            if particle.fitness > threshold:
                particle.poor_performance_count += 1
            else:
                particle.poor_performance_count = 0
    else:
        # 最大化问题
        base_fitness = self.global_best_fitness

```

```

threshold = base_fitness * (1 - self.pruning_threshold)
# 检查并更新每个粒子的表现计数
for particle in self.particles:
    if particle.fitness < threshold:
        particle.poor_performance_count += 1
    else:
        particle.poor_performance_count = 0
# 移除表现不佳的粒子
if len(self.particles) > self.min_particles:
    # 筛选出需要保留的粒子
    good_particles = [p for p in self.particles if
                      p.poor_performance_count <
                      self.pruning_count]
    # 确保至少保留 min_particles 个粒子
    if len(good_particles) >= self.min_particles:
        pruned_count = len(self.particles) - len(good_particles)
        self.particles = good_particles
        if pruned_count > 0:
            return pruned_count # 返回剪枝的粒子数量
return 0
def optimize(self, verbose: bool = True) -> Tuple[np.ndarray, float]:
    """
    执行粒子群优化
    参数：
        verbose: 是否打印迭代过程信息
    返回：
        Tuple[np.ndarray, float]: (最优解位置, 最优适应度值)
    """
    # 初始化：评估所有粒子的初始适应度
    if self.parallel:
        # 并行评估初始适应度
        with ProcessPoolExecutor(max_workers=self.n_processes) as
executor:
            # 提交所有粒子的适应度计算任务
            futures = [executor.submit(evaluate_fitness,
                           particle.position, self.cfg, id)
                       for id, particle in enumerate(self.particles)]
        # 收集结果
        for i, future in enumerate(as_completed(futures)):
            self.particles[i].fitness = future.result()
        # 更新粒子的个体最佳
            self.particles[i].best_fitness =

```

```
        self.particles[i].fitness
        self.particles[i].best_position =
        self.particles[i].position.copy()
    # 确保 best_position 中的整数维度是整数类型
    for dim_idx in self.integer_dims:
        self.particles[i].best_position[dim_idx] =
            int(self.particles[i].best_position[dim_idx]))
# 更新全局最佳
if (self.minimize and
    self.particles[i].fitness < self.global_best_fitness) or \
    (not self.minimize and
    self.particles[i].fitness > self.global_best_fitness):
    self.global_best_fitness
    = self.particles[i].fitness
    self.global_best_position
    = self.particles[i].position.copy()
# 确保 global_best_position 中的整数维度是整数类型
for dim_idx in self.integer_dims:
    self.global_best_position[dim_idx]
    = int(self.global_best_position[dim_idx]))

else:
    # 串行评估初始适应度（保持原有代码）
    for particle in self.particles:
        # 使用外部接口计算适应度
        particle.fitness = calculate_fitness(particle.position,
                                              self.cfg)
    # 更新粒子的个体最佳
    particle.best_fitness = particle.fitness
    particle.best_position = particle.position.copy()
    # 确保 best_position 中的整数维度是整数类型
    for dim_idx in self.integer_dims:
        particle.best_position[dim_idx] =
            int(particle.best_position[dim_idx]))
    # 更新全局最佳
    if (self.minimize and particle.fitness
        < self.global_best_fitness) or \
        (not self.minimize and particle.fitness >
        self.global_best_fitness):
        self.global_best_fitness = particle.fitness
        self.global_best_position = particle.position.copy()
    # 确保 global_best_position 中的整数维度是整数类型
    for dim_idx in self.integer_dims:
```

```
        self.global_best_position[dim_idx]
        = int(self.global_best_position[dim_idx])

# 早停相关变量
no_improve_count = 0
last_best_fitness = self.global_best_fitness

# 主循环
for iter_num in range(self.max_iter):
    # 动态更新算法参数
    self.update_params(iter_num)
    # 更新粒子的速度和位置
    for particle in self.particles:
        # 速度更新公式
        r1 = np.random.random(self.dim)
        r2 = np.random.random(self.dim)

        cognitive_velocity = self.c1 * r1 *
        (particle.best_position - particle.position)
        social_velocity = self.c2 * r2 *
        (self.global_best_position - particle.position)
        particle.velocity = self.w * particle.velocity +
        cognitive_velocity +
        social_velocity
        # 位置更新
        particle.position = particle.position +
        particle.velocity
    # 整数维度取整
    for dim_idx in self.integer_dims:
        particle.position[dim_idx] =
            int(particle.position[dim_idx])
    # 边界处理
    for i in range(self.dim):
        low, high = self.bounds[i]
        if particle.position[i] < low:
            particle.position[i] = low
            particle.velocity[i] *= -0.5 # 反弹
        elif particle.position[i] > high:
            particle.position[i] = high
            particle.velocity[i] *= -0.5 # 反弹
    # 确保整数维度在边界处理后仍然是整数
    if i in self.integer_dims:
        particle.position[i] = int(particle.position[i])
# 并行评估所有粒子的适应度
```

```
if self.parallel:
    positions = [particle.position for particle in
    self.particles]

with ProcessPoolExecutor(max_workers=self.n_processes)
as executor:
    # 提交所有粒子的适应度计算任务
    futures = [executor.submit(evaluate_fitness, pos,
    self.cfg, id) for
    id, pos in enumerate(positions)]
    # 收集结果并更新粒子适应度
    for i, future in enumerate(as_completed(futures)):
        fitness = future.result()
        self.particles[i].fitness = fitness
    # 更新粒子的个体最佳
    if (self.minimize and
    fitness < self.particles[i].best_fitness) or \
    (not self.minimize
    and fitness > self.particles[i].best_fitness):
        self.particles[i].best_fitness = fitness
        self.particles[i].best_position =
        self.particles[i].position.copy()
    # 确保 best_position 中的整数维度是整数类型
    for dim_idx in self.integer_dims:
        self.particles[i].best_position[dim_idx] =
        int(self.particles[i].best_position[dim_idx]))
    # 更新全局最佳
    if (self.minimize and
    fitness < self.global_best_fitness) or \
    (not self.minimize and fitness >
    self.global_best_fitness):
        self.global_best_fitness = fitness
        self.global_best_position =
        self.particles[i].position.copy()
    # 确保 global_best_position 中的
    # 整数维度是整数类型
    for dim_idx in self.integer_dims:
        self.global_best_position[dim_idx]
        = int(self.global_best_position[dim_idx]))
else:
    # 串行评估适应度（保持原有代码）
    for particle in self.particles:
```

```
# 评估新位置，使用外部接口计算适应度
particle.fitness = calculate_fitness
(particle.position, self.cfg)
# 更新粒子的个体最佳
if (self.minimize and particle.fitness
< particle.best_fitness) or
\
(not self.minimize and
particle.fitness > particle.best_fitness):
    particle.best_fitness = particle.fitness
    particle.best_position =
    particle.position.copy()
# 确保 best_position 中的整数维度是整数类型
for dim_idx in self.integer_dims:
    particle.best_position[dim_idx] =
        int(particle.best_position[dim_idx]))
# 更新全局最佳
if (self.minimize and particle.fitness
< self.global_best_fitness)
or \
(not self.minimize and
particle.fitness > self.global_best_fitness):
    self.global_best_fitness = particle.fitness
    self.global_best_position =
    particle.position.copy()
# 确保 global_best_position 中的整数维度是整数类型
for dim_idx in self.integer_dims:
    self.global_best_position[dim_idx] =
        int(self.global_best_position[dim_idx]))
# 记录本次迭代的最佳适应度
self.fitness_history.append(self.global_best_fitness)
# 剪枝表现不佳的粒子
pruned_count = self.prune_particles(iter_num)
# 检查是否需要早停
if self.early_stop:
    # 计算改善程度
    if self.minimize:
        improvement = last_best_fitness
        - self.global_best_fitness
    else:
        improvement = self.global_best_fitness -
        last_best_fitness
```

```

# 判断是否有显著改善
if improvement > self.early_stop_tol:
    no_improve_count = 0 # 重置计数器
    last_best_fitness = self.global_best_fitness
else:
    no_improve_count += 1
# 达到早停条件
if no_improve_count >= self.early_stop_iter:
    if verbose:
        logger.info(f"早停触发！连续{self.early_stop_iter}次迭代无显著改善，在第{iter_num+1}次迭代停止")
        break
# 打印进度信息
if verbose and (iter_num + 1) % 1 == 0:
    if self.dynamic_params:
        logger.info(f"迭代{iter_num+1}/{self.max_iter}, 当前最佳适应度:{self.global_best_fitness}, 最佳位置:{self.global_best_position}, w={self.w:.4f}, c1={self.c1:.4f}, c2={self.c2:.4f}, 剩余粒子:{len(self.particles)})")
    else:
        logger.info(f"迭代{iter_num+1}/{self.max_iter}, 当前最佳适应度:{self.global_best_fitness}, 最佳位置:{self.global_best_position}, 剩余粒子:{len(self.particles)})")
    if verbose:
        logger.info(f"优化完成！最佳位置:{self.global_best_position}, 最佳适应度:{self.global_best_fitness}")
return self.global_best_position, self.global_best_fitness
def plot_convergence(self) -> None:
    """绘制收敛曲线"""
    plt.figure(figsize=(10, 6))
    plt.plot(range(1, len(self.fitness_history) + 1), self.fitness_history, marker='o', linestyle='-', markersize=2)
    plt.title('PSO 算法收敛曲线')
    plt.xlabel('迭代次数')
    plt.ylabel('全局最佳适应度')
    plt.grid(True)
    plt.show()

```

```
def load_config(cfg_path):
    with open(cfg_path, 'r', encoding='utf-8') as file:
        cfg = yaml.safe_load(file)
    return cfg

# 示例使用
def start_simulation():
    # 设置日志
    cfg = load_config('config.yaml')
    logger.info("加载配置文件成功")
    # 定义问题参数
    dim = 3
    bounds = [(cfg['vehicles_lower_bounds'], cfg['vehicles_upper_bounds']),
              (cfg['n_clusters_lower_bounds'], cfg['n_clusters_upper_bounds']),
              (cfg['height_threshold_lower_bounds'], cfg['height_threshold_upper_bounds'])]
    logger.info(f"问题维度:{dim}, 搜索范围:{bounds}")
    initial_values = [cfg['num_vehicles'], cfg['n_clusters'], cfg['height_threshold']]
    logger.info(f"初始值:{initial_values}")

    # 指定第0维和第1维（车辆数量和聚类数）为整数维度
    integer_dims = [0, 1]
    # 从配置文件获取早停参数，如果不存在则使用默认值
    early_stop = cfg.get('early_stop', False)
    early_stop_iter = cfg.get('early_stop_iter', 20)
    early_stop_tol = cfg.get('early_stop_tol', 1e-6)
    # 从配置文件获取动态参数设置，如果不存在则使用默认值
    dynamic_params = cfg.get('dynamic_params', False)
    w_strategy = cfg.get('w_strategy', 'linear')
    w_min = cfg.get('w_min', 0.4)
    w_max = cfg.get('w_max', 0.9)
    c_strategy = cfg.get('c_strategy', 'constant')
    # 从配置文件获取剪枝参数
    pruning = cfg.get('pruning', False)
    pruning_start = cfg.get('pruning_start', 20)
    pruning_threshold = cfg.get('pruning_threshold', 0.5)
    pruning_count = cfg.get('pruning_count', 10)
    min_particles = cfg.get('min_particles', 5)
    # 从配置文件获取并行处理参数
    parallel = cfg.get('parallel', True)
    n_processes = cfg.get('n_processes', None)

    logger.info("PSO参数配置:")
    logger.info(f"粒子数:{cfg['num_particles']},")
```

```

    最大迭代次数 : cfg['max_iter'])
    logger.info(f"是否最小化问题 : cfg['minimize']",
    w: cfg['w'], c1: cfg['c1'], c2:
    cfg['c2'])")
    logger.info(f"早停 : early_stop, 早停迭代 : early_stop_iter,
    早停阈值 :
    early_stop_tol")
    logger.info(f"动态参数 : dynamic_params, w策略 : w_strategy,
    w范围 : [w_min],
    {w_max}], c策略 : c_strategy")
    logger.info(f"剪枝 : pruning, 开始剪枝迭代 : pruning_start,
    剪枝阈值 :
    pruning_threshold")
    logger.info(f"最小粒子数 : min_particles, 并行计算 : parallel,
    进程数 :
    n_processes")

pso = ParticleSwarmOptimization(
    dim=dim,
    bounds=bounds,
    num_particles=cfg['num_particles'],
    max_iter=cfg['max_iter'],
    minimize=cfg['minimize'],
    w=cfg['w'],
    c1=cfg['c1'],
    c2=cfg['c2'],
    initial_values=initial_values,
    integer_dims=integer_dims, # 添加整数维度参数
    cfg=cfg,
    early_stop=early_stop,
    early_stop_iter=early_stop_iter,
    early_stop_tol=early_stop_tol,
    dynamic_params=dynamic_params,
    w_strategy=w_strategy,
    w_min=w_min,
    w_max=w_max,
    c_strategy=c_strategy,
    pruning=pruning,
    pruning_start=pruning_start,
    pruning_threshold=pruning_threshold,
    pruning_count=pruning_count,
    min_particles=min_particles,
)

```

```
    parallel=parallel ,  
    n_processes=n_processes  
)  
logger.info("开始PSO优化...")  
# 运行优化  
best_position, best_fitness = pso.optimize(verbose=True)  
logger.info(f"优化结束.最佳位置:{best_position},  
最佳适应度:{best_fitness}")  
# 绘制收敛曲线  
pso.plot_convergence()  
logger.info("收敛曲线已生成")  
return best_position, best_fitness  
if __name__ == "__main__":  
    try:  
        best_position, best_fitness = start_simulation()  
        logger.info(f"程序成功完成.最终结果:位置={best_position},  
适应度={best_fitness}")  
    except Exception as e:  
        logger.exception(f"程序执行出错:{str(e)}")
```