基于校园共享单车调度与维护优化模型的研究

2025 年 4 月 20 日

摘　要

本文针对校园共享单车的调度与维护优化问题，构建了多阶段模型以提高运营效率和服务质量。首先，通过数据预处理和聚类分析估算校园内共享单车总量（约 800 辆），并采用样条插值填补缺失数据，分析不同停车点的单车分布规律。其次，基于 A* 算法设计最短路径调度策略，结合遗传算法优化调度方案，缓解高峰期的供需矛盾，降低调度成本。随后，利用熵权法和 TOPSIS 方法评估停车点布局的合理性，提出优化调整方案，提升点位分布均衡性和使用率。最后，设计故障车辆的巡检路线与运输策略，通过动态惩罚因子和奖励机制确保故障车辆高效回收。实验结果表明，模型显著提升了单车调度效率和用户满意度，为校园共享单车系统的科学管理提供了理论支持。

关键词：共享单车；调度优化；遗传算法；熵权法；巡检路线；校园交通

目录

1 问题重述

1.1 问题背景

随着智慧城市理念与绿色出行模式的深度融合，共享单车系统作为公共自行车系统（PBS，Public Bicycle System）的一部分，成为新型微交通网络的典型范式，并在高校生态圈层中展现出独特的服务价值。在时空压缩的校园场景下，共享单车通过构建” 即时响应-精准触达” 的服务机制，有效缓解了校园短途出行结构性矛盾。这种出行方式的革新不仅重构了校园交通微循环体系，更催生出具有学术研究价值的运营管理命题。

然而，当前校园共享交通系统正面临规模效应与运维能力的双重挑战。校园学生人数众多，借车需求具有潮汐效应和时空分布不均衡的特性，而共享自行车投放数量不足，致使” 借车难” 的问题；而不科学的运维系统导致共享自行车损坏维修周期长，高峰期不能即时满足用车需求。因此，建立合理的调度和维护策略，确保单车的正常运行，是提升共享单车系统服务质量的关键。为了解决共享单车的调度和维护问题，必要时可以借鉴其他交通系统的成功经验和技术方法。

国内外对共享自行车系统的相关调度有深入的研究。Vogel P[1] 利用聚类算法对维也纳的公共自行车系统的运营数据进行分析。《Journal of Transport Geography》[2, 3, 4] 也曾出版论文的专辑，从多个角度对公共自行车运行的特点进行过分析。而在空间分析的角度，胡继华等 [5] 利用构建城市公交矩阵对城市居民出行的规律进行研究。董红召等 [6] 利用分形树自平衡划分算法对城市中共享自行车系统调度区域进行聚类分析，将城市划分为可达到自平衡与互平衡的两级区域。即使在非共享自行车领域，也有类似的分析，刘家良等 [7] 利用遗传算法对城市出租车调度点进行配置，为本文进行共享自行车调度点配置提供思路参考。

1.2 问题的数据条件

(1) 共享单车的运维处位于校园东北角。
(2) 共享单车只能在校园内骑行，不能离开校园。
(3) 共享单车收费采取包月制，费用在大多数学生承受范围内。
(4) 调度车和检修车的时速恒为 25 km/h，每次最多可运输 20 辆共享单车。
(5) 检修师傅鲁迪每到一个停车点位，查找和搬运一辆故障车的平均用时为 1 分钟。
(6) 共享单车每天的总故障率为 6%。

2 问题分析

2.1 具体问题分析

2.1.1 问题一分析

问题 1 要求根据统计数据估算校园内的共享单车总量，并分析不同停车点在不同时间点的单车数量分布。这一部分需要对附件 1 中的数据进行详细分析，计算每个停车点

在不同时间点的单车数量，并将其填入表 1 中。关键在于确保数据的准确性和完整性。

2.1.2 问题二分析

问题 2 涉及建立用车需求模型，并设计调度策略以缓解高峰期的供需矛盾。这一部分需要分析用车需求，确定高峰时段，并设计调度计划。重点在于如何在满足高峰期需求的同时，最小化调度成本和时间。

2.1.3 问题三分析

问题 3 要求评估现有停车点位的合理性，并提出优化方案。基于调度模型的结果，评估停车点位的布局是否合理。如果不合理，提出调整方案，并重新评估运营效率。这一部分需要综合考虑用车需求、调度效率和停车点位的分布，找到最优解。

2.1.4 问题四分析

问题 4 关注于设计巡检路线和运输策略，以最短时间将故障车辆运回检修处。基于优化后的停车点位布局，设计鲁迪的巡检路线和运输策略。重点在于如何在保证故障车辆及时处理的同时，最大化运输效率。

3 符号说明

符号参数	物理意义
P_i	停车点位（i=1-16）
C_{ij}	停车点到停车点的距离（单位：m）。
V	调度车的速度（2025 km/h）
$Q_{\max}$	每辆调度车的最大运输能力（20 辆单车）。
$M_{\max}$	调度车的数量（3 辆车）
$T_{\max}$	可用于调度的最大时间
t_{ij}	调度车从停车点到停车点的行驶时间
D_{ij}	停车点 Pi 在第 j 波用车高峰的单车需求量。
S_{ij}	停车点 Pi 在第 j 波用车高峰的单车实际数量。
$Y_i(t)$	调度车是否在停车点 i 调度车辆（0 表示不调取，1 表示调入，-1 表示调出）
$X_i(t)$	从停车点 Pi 调度的单车数量
$Z_i(t)$	停车点 i 在 t 时间的故障车总数。每辆车有 6% 的概率故障。
T_{ij}	第 i 辆车完成第 j 轮调度花费的时间
$A_i(t)$	停车点 i 在时间 t 剩余的车辆。
$P_{\text{repair\_center}}$	检修处的位置索引（默认为 0，即第一个停车点）
t_{load}	装卸一辆车所需的时间，单位为分钟
$T_{\text{max\_work\_time}}$	单次巡检的最大工作时间，单位为分钟
$T_{\text{inspection}}$	巡检时间安排，一天四次巡检，每次巡检的开始时间
$peny2$	定义惩罚因子
$peny1$	定义奖励因子
$N_{\text{pop\_size}}$	种群大小（默认为 50）
P_{pc}	交叉概率（默认为 0.8）
$N_{\text{max\_gen}}$	最大迭代次数（默认为 100）
P_{pm}	变异概率（默认为 0.1）

4 模型建立与求解

4.1 问题一模型

4.1.1 数据预处理

首先对原始数据进行清洗，去除异常值和重复记录，确保数据质量。根据图标中数据，可以观察到，共享单车对于同一个点位不同时段在工作日和周末是具有极度相似的数量分布，因此我们对同一个点位不同时间段的各个数据，在工作日上取周三周四周五的平均值，在周末取周六周日的平均值。

接下来进行特征提取和特征转换：从时间和空间维度提取特征。时间特征包括日期、星期几、小时等；空间特征包括停车点位的地理位置信息。将分类特征（如停车点位名称）转换为数值形式，便于后续分析。

然后, 进行聚类分析：根据车流的潮汐现象，分析早高峰（7:00 左右）、午高峰（12

点左右）、晚高峰（18:00 左右）车流量利用 K-means、DBSCAN 等算法对停车点位进行聚类，识别出具有相似单车分布模式的区域。对时间点进行聚类，识别出单车需求的高峰和低谷时段。

		东门	南门	北门	一食堂	二食堂	三食堂	梅苑 1 栋	菊苑 1 栋
工作日	7:30	31	46	18	75	105	90	97	102
	8:50	68	66	63	5	10	7	9	10
	11:10	55	31	68	6	19	14	19	59
	12:20	28	66	70	110	160	135	93	82
	13:50	54	29	66	6	9	12	20	67
	18:00	36	110	72	49	74	62	67	73
	21:20	98	78	24	26	72	57	8	65
	23:00	14	47	18	83	114	123	128	126
周末	9:00	78	72	31	54	43	47	106	93
	12:00	86	52	102	57	58	66	52	66
	15:00	146	140	60	14	10	12	28	28
	18:00	114	103	125	54	55	52	25	25
	21:00	90	83	42	55	53	47	119	118

缺失值填补后的附件 1

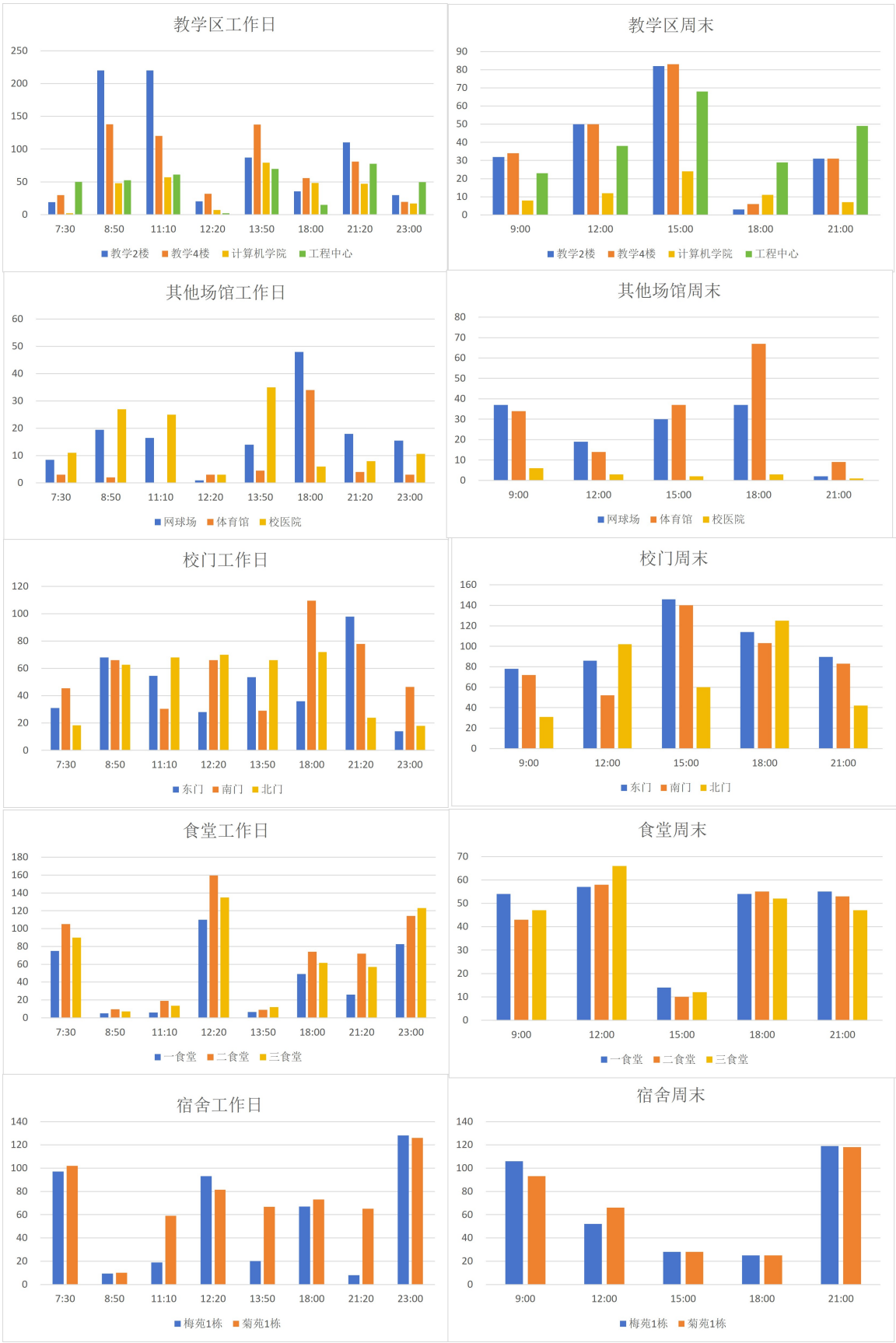
	教学 2 楼	教学 4 楼	计算机学院	工程中心	网球场	体育馆	校医院	预计 800 辆
7:30	19	30	2	50	9	3	11	688
8:50	220	138	48	53	20	2	27	745
11:10	220	120	57	61	17	0	25	769
12:20	20	32	7	2	1	3	3	812
13:50	87	138	80	70	14	5	35	690
18:00	36	56	49	15	48	34	6	785
21:20	110	81	47	78	18	4	8	774
23:00	30	20	17	50	16	3	11	798
9:00	32	34	8	23	37	34	6	698
12:00	50	50	12	38	19	14	3	725
15:00	82	83	24	68	30	37	2	764
18:00	3	6	11	29	37	67	3	709
21:00	31	31	7	49	2	9	1	737

基于聚类结果，采用局部平均值、中位数或众数对缺失值相似性填补。对于每个缺失值，找到其所属的聚类，然后使用该聚类内的其他数据点进行填补。

根据聚类结果显示，可以得到：**教学区的点位可以归为一类**（包括教学 2 楼、教学 4 楼、计算机学院、工程中心）；**三个校门可以归为一类**（包括东门、南门、北门）；**三个食堂可以归为一类**三个食堂可以归为一类（包括一食堂、二食堂、三食堂）；**两个宿舍可以归为一类**（包括梅苑 1 栋、菊苑 1 栋）。

预计共享单车总量的的区间在 687-812 之间，考虑到存在共享单车不在点位而在骑行路上的情况，应该取较大值，如上图标红的部分。同时，因为 23:00 时刻所有公共区域关闭，而且与学生作息相关，要重点考虑。综合情况来看，**估计目前校园内的共享单车总量位 800 辆。**

图 1: 不同点位各个时间段的单车数



根据上述的数据，构建一个多维样条函数。样条函数通常由多个低次多项式组成，每个多项式在特定的区间内定义。利用构建好的样条函数，对表格中缺失的时间点进行插值计算，得到相应的单车数量。

表 1: 样条插值填充后的数据

	7:00	**9:00**	**12:00**	**14:00**	**18:00**	**21:00**	**23:00**
东门	28	65	30	53	36	102	14
南门	46	60	60	29	110	82	47
北门	18	63	70	66	72	26	18
一食堂	78	5	6	6	49	32	83
二食堂	110	11	19	9	74	73	114
三食堂	95	8	14	12	62	59	123
梅苑 1 栋	105	11	85	20	67	15	128
菊苑 1 栋	110	12	70	67	73	66	126
教学 2 楼	21	220	200	87	36	100	30
教学 4 楼	32	136	115	138	56	75	20
计算机学院	10	50	45	80	49	48	17
工程中心	50	52	50	70	15	68	50
网球场	10	19	12	14	48	22	16
体育馆	3	2	1	5	34	6	3
校医院	10	27	18	35	6	7	11

4.1.2 模型思路与分析

- **聚类分析辅助填补**——相似性填补：通过聚类分析，将相似的停车点位和时间点归为一类。对于某个时间点某个停车点位的缺失值，可以用同一类中其他时间点或停车点位的平均值、中位数或众数来填补。

- **特征提取和特征转换**——时间特征：将时间转换为数值特征（如小时、星期几等），以便更好地捕捉时间模式。空间特征：利用位置坐标（如经纬度）进行空间聚类，识别不同区域的单车分布模式。

- **回归模型**——使用机器学习回归模型（如线性回归、随机森林回归等）来预测缺失值。输入特征可以是时间特征、空间特征以及其他可能影响单车数量的特征。时间序列模型：如果数据具有时间序列特性，可以使用 ARIMA、LSTM 等时间序列模型进行预测。

4.1.3 样条插值检验

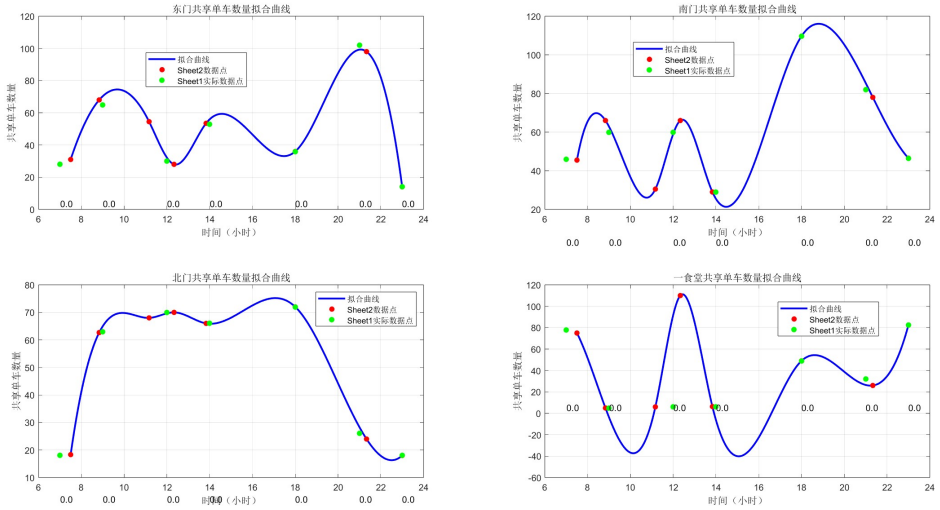
评估指标

均方误差（Mean Squared Error, MSE）：衡量预测值与真实值之间的平均平方差，适用于连续值预测。
平均绝对误差（Mean Absolute Error, MAE）：衡量预测值与真实值之间的平均绝对差，更直观地反映预测误差。
R^2（决定系数）：衡量模型解释数据变异性的能力，值越接近 1 表示模型拟合越好。
F1 分数：对于分类问题，评估模型在精确度和召回率之间的平衡。

改进方案

平滑样条（Smoothing Spline）：通过调节平滑参数（sp）平衡拟合度与光滑性，避免过拟合。MATLAB 函数：csaps。
多项式拟合 + 正则化：使用低次多项式（如 3 次）配合正则化减少过拟合。MATLAB 函数：polyfit + polyval。
局部加权回归（LOESS）：适合非线性数据，对局部波动更鲁棒。

图 2: 样条插值图像



4.2 问题二模型

4.3 模型思路与分析

模型思路主要围绕如何在给定的时间点和需求条件下，通过优化算法确定最佳调度策略。包括需求数据、单车分布数据和距离矩阵。需求数据提供了不同时间点的单车需求量，单车分布数据则展示了各个停车点的单车数量，而距离矩阵则用于计算调度成本。

目标函数旨在最大化用户满意度并最小化调度成本。用户满意度通过满足用户需求来衡量，而调度成本则包括运输成本和时间成本。为了实现这一目标，模型需要考虑约束条件，如单车容量限制、时间限制和供需平衡。

遗传算法是一种模拟自然选择和遗传机制的优化方法，广泛应用于解决复杂的组合

优化问题。其核心思想是通过模拟生物进化过程中的选择、交叉和变异等机制，来寻找问题的最优解。

本文优化算法采用的遗传算法，本质是模拟自然选择和遗传机制的智能优化方法，特别擅长攻克复杂的组合优化难题。它将问题的每个解编码成类似生物“染色体”的结构，常用二进制或实数编码。算法从随机生成的初始解群体起步，通过适应度函数评估每个个体的优劣——在共享单车调度场景下，适应度紧密关联目标函数，比如满足更多用户需求、降低调度成本的方案，适应度就更高。算法的核心步骤包括选择、交叉和变异：选择操作依据适应度筛选优质个体，让优秀的调度方案有更大机会传递到下一代；交叉操作模拟生物基因重组，将两个父代个体的部分基因互换，创造新的子代个体；变异操作则引入随机扰动，对个体的部分基因进行随机改变，以此避免算法过早陷入局部最优，维持种群的多样性。通过不断迭代这些操作，种群在进化中持续优化，直至达到预设的迭代次数或满足停止条件。

不过，单纯的遗传算法在搜索后期容易陷入局部最优解，难以找到全局最优。此时，模拟退火算法的加入发挥了重要作用。在模拟退火算法中，“温度”是核心控制参数，高温时算法以较高概率接受较差解，鼓励在解空间中广泛探索；随着温度逐步降低，接受较差解的概率也减小，算法逐渐聚焦于高质量解。将模拟退火算法与遗传算法结合后，每次遗传算法生成新个体，模拟退火机制都会对其进行优化，即使算法陷入局部最优，也能借助模拟退火跳出困境，继续探索更广阔的解空间，大大提高找到全局最优解的概率。此外，局部搜索算法如 2-opt 可用于优化调度路线，需求波动可以通过统计分析或预测模型来建模，动态调整和惩罚机制通过动态调整惩罚因子来激励算法在满足需求的同时尽量减少调度成本。

尽管采用了带退火的遗传算法，但是鉴于在本题所讨论的复杂优化问题下，该算法仍然容易陷入局部最优，因此，还需引入 2-opt 算法 [9]，帮助算法在途径选择上得到一个较好的解。

最后，记录每个停车点的初始车辆数、需求和最终车辆数，以及供需状态，有助于分析和优化调度策略。这种模型适用于校园环境和城市环境，帮助提高单车使用效率和用户体验。

4.3.1 数据预处理

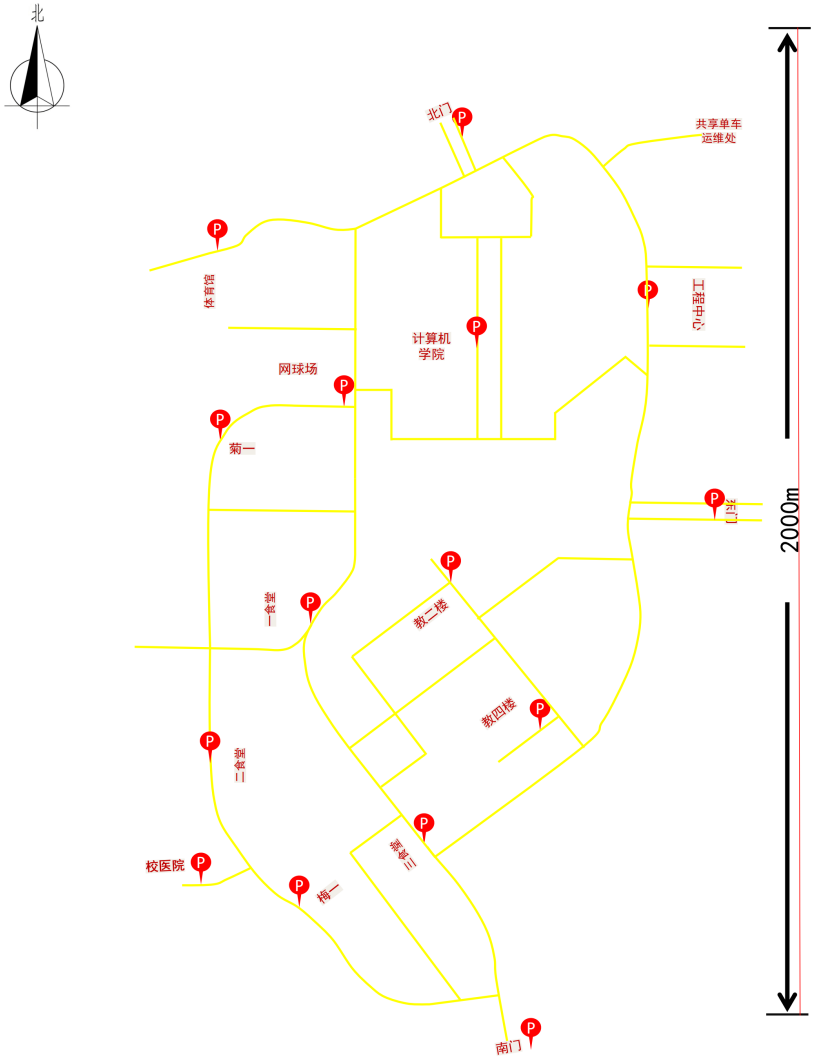
建立调度模型之前，需要对校园地图进行预处理，以**确定各个停车点之间的最短路径**。A*（A-Star）算法由 Hart 等人提出 [8], 是一种计算任意两个停车点之间的最短路径，在计算机视觉和机器人学中有着广泛应用，常被用作规划路径的首选方法。A* 算法需要一个启发式函数来估计从当前点到目标点的距离。常用的启发式函数是欧几里得距离。

A*（A-Star）算法是一种在图形平面上，有多个节点的路径找出最低通过成本的算法。该算法综合了最良优先搜索和 Dijkstra 算法的优点：在进行启发式搜索提高算法效率的同时，可以保证找到一条最优路径。A* 算法通过维护一个开放列表（Open List）和一个关闭列表（Closed List）来寻找从起始节点到目标节点的最优路径。

对于每个节点，它会计算两个重要的值：

- **g(n)**：表示从起始节点到当前节点 n 的实际代价。
- **h(n)**：是一个启发式函数，用于估计从当前节点 n 到目标节点的代价。这个估计值需要尽可能接近实际值，但又不能高估，否则可能无法找到最优路径。
- **f(n)**：f(n) = g(n) + h(n)，表示从起始节点经过当前节点 n 到达目标节点的总估计代价。A* 算法会优先扩展 f(n) 值最小的节点。

图 3: 附件二简化学校地图



附件 2 中可以获取校园地图的信息，包括停车点和可行驶的路线，这些信息通常以坐标形式给出。将地图信息转换为计算机可以处理的格式。可以使用二值图像表示地图，其中黄色线路色表示可行驶区域，红色色表示共享单车点位。

表 2: 共享单车点位

P1	P2	P3	P4	P5	P6	P7	P8
东门	南门	北门	一食堂	二食堂	三食堂	梅苑 1 栋	菊苑 1 栋
P9	P10	P11	P12	P13	P14	P15	P16
教学 2 楼	教学 4 楼	计算机学院	工程中心	网球场	体育馆	校医院	运营点

表 3: 不同点位之间的最短距离（单位/米）

	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**
1	0	1508	1323	1329	2282	1112	1902	1595	701	806	1102	613	1313	1858	2147	1161
2	1508	0	2218	1016	969	468	588	1607	1234	936	2043	1774	1511	2164	834	2322
3	1323	2218	0	1201	1669	1750	2049	1022	1676	1782	556	712	740	674	2504	606
4	1329	1016	1201	0	468	549	848	658	809	923	1026	1419	495	1148	353	1668
5	2282	969	1669	468	0	1188	380	1139	1990	1710	1988	2381	1423	1616	343	2136
6	1112	468	1750	549	1188	0	861	1206	766	541	1575	1378	1044	1697	1107	1926
7	1902	588	2049	848	380	861	0	1520	1610	1330	2369	2167	1803	1996	240	2716
8	1595	1607	1022	658	1139	1206	1520	0	1399	1513	847	1240	283	969	1482	1489
9	701	1234	1676	809	1990	766	1610	1399	0	403	1456	966	1304	1957	1855	1515
10	806	936	1782	923	1710	541	1330	1513	403	0	1561	1072	1417	2070	1575	1620
11	1102	2043	556	1026	1988	1575	2369	847	1456	1561	0	747	565	1149	1379	1016
12	613	1774	712	1419	2381	1378	2167	1240	966	1072	747	0	958	1247	2724	550
13	1313	1511	740	495	1423	1044	1803	283	1304	1417	565	958	0	687	1766	1207
14	1858	2164	674	1148	1616	1697	1996	969	1957	2070	1149	1247	687	0	2453	1141
15	2147	834	2504	353	343	1107	240	1482	1855	1575	1379	2724	1766	2453	0	2021
16	1161	2322	606	1668	2136	1926	2716	1489	1515	1620	1016	550	1207	1141	2021	0

需求量计算：从需求数据中获取每个停车点在特定时间段的需求量。这通常是从需求数据文件中读取的。

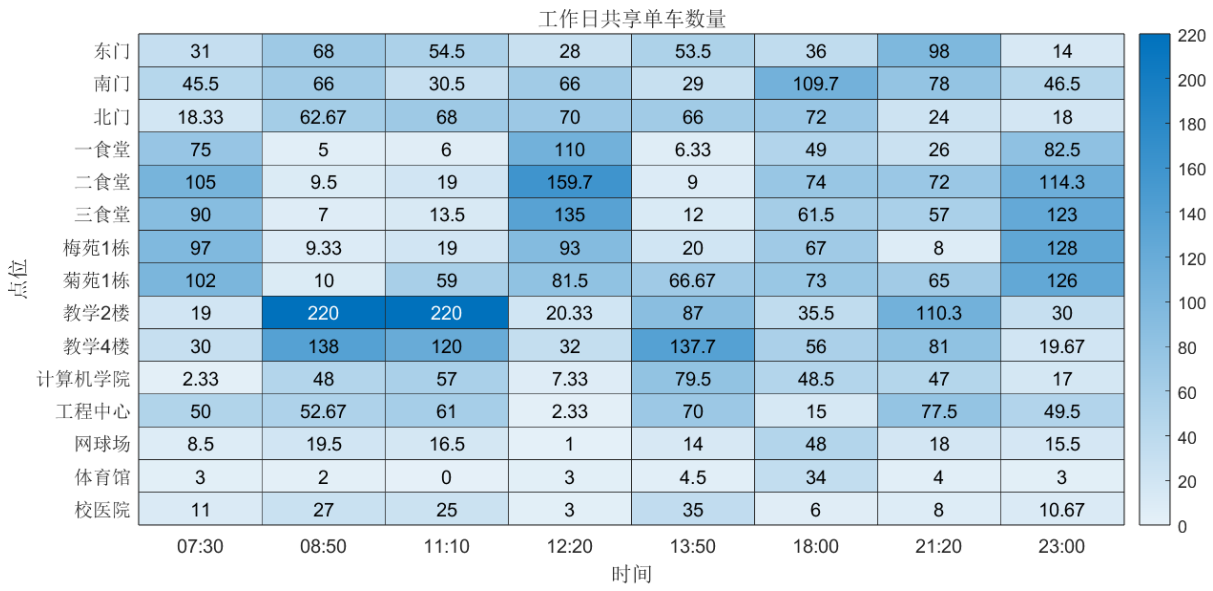
单车数量计算：从单车数量分布数据中获取每个停车点在特定时间点的单车数量。这通常是从单车数量分布数据文件中读取的。**比较单车数量和需求量：**对于每个停车点，比较其单车数量和需求量：

如果单车数量大于需求量，说明该停车点供大于求。如果单车数量小于需求量，说明该停车点供不应求。如果单车数量等于需求量，说明供需平衡。

表 4: 共享单车数量变化

	7：50 之前	12：00 之前	13：50 之前	21：10 之前
东门	37	-2	25.5	-84
南门	20.5	6	-37	-31.5
北门	44	0	-4	-6
一食堂	-70	104	-104	56.5
二食堂	-95.5	141	-151	42
三食堂	-83	121	-123	66
梅苑 1 栋	-88	8	-73	120
菊苑 1 栋	-92	11.5	-15	61
教学 2 楼	201	-180	67	-80
教学 4 楼	108	-83	106	-61
计算机学院	46	-38	73	-30
工程中心	3	-48	68	-28
网球场	11	-11	13	-2.5
体育馆	-1	2	1.5	-1
校医院	16	-15	32	3

图 4: 共享单车数量热力图



4.3.2 公式说明

yi 取值规则：当调度车到 P_i 时，若 $S_i(j) < 0$，且 $A_i(t) < 0.7A_i(0)$，则 $y_i = -1$；

若 $S_i(j) > 0$，且 $A_i(t) < -1.4D_i$，则 $y_i = 1$。如果调度车不在 P_i，$y_i = 0$。

xi 取值规则：若 $y_i = -1$，且 $Q_i < 20$，则 $Q_i(t+1) = Q_i(t) + x_i(t)$，此时 $x_i(t)$ 越大越好，最大值为 $\min(20 - Q_i(t), A_i(t) - 0.7A_i(0))$；（取最大值中的最小一个）

$A_i(t) = A_i(t) - 1 + y_i(t-1) \times x_i(t-1)$。

定义 t0 时调度完成 O（j）：在第 j 波调度中的超调量。

对于 $S_i(j) > 0$ 的 P_i，若 $A_i(t_0) > 1.4S_i(j)$，则

$$O(j) = O(j) + (A_i(t_0) - 1.4S_i(j)),$$

以此历遍所有 $S_i(j) > 0$ 的 P_i；

对于 $S_i(j) < 0$ 的 P_i，若 $A_i(t_0) > 0.8A_i(0)$，则

$$O(j) = O(j) + (A_i(t_0) - 0.8A_i(0)),$$

以此历遍所有 $S_i(j) < 0$ 的 P_i。

定义惩罚因子 $peny(1)$，

$$Z_1 = peny(1) \times S_i(j).$$

- **单车需求约束**：对于每个停车点，其最终的单车数量需要满足需求量，即：

  $$S_i^{\text{final}} = D_i, \quad \forall i \in \mathcal{P}.$$

  其中 S_i^{final} 表示停车点 i 的最终单车数量，D_i 表示停车点 i 的需求量，$\mathcal{P}$ 表示所有停车点的集合。

- **调度车运输能力约束**：每辆调度车的运输能力不能超过 20 辆单车，即：

  $$\sum_{i \in \mathcal{P}} x_i \leq 20, \quad \forall k \in \mathcal{K}.$$

  其中 x_i 表示从停车点 i 调度的单车数量，$\mathcal{P}$ 表示所有停车点的集合，$\mathcal{K}$ 表示所有调度车的集合。

- **调度车数量约束**：调度车的数量限制为 3 辆，且每辆车只可以运输一次，即：

  $$|\mathcal{K}| = 3, \quad \text{且} \quad \sum_{k \in \mathcal{K}} y_k \leq 1, \quad \forall i \in \mathcal{P}.$$

  其中 y_k 表示调度车 k 是否被使用（0 或 1），$\mathcal{P}$ 表示所有停车点的集合，$\mathcal{K}$ 表示所有调度车的集合。

- **行驶时间约束**：所有调度车的行驶时间不能超过可用时间，即：

  $$T_k \leq T_{\max}, \quad \forall k \in \mathcal{K}.$$

  其中 T_k 表示调度车 k 的总行驶时间，$T_{\max}$ 表示可用时间的最大值。

- **路径选择约束：**调度车从某个停车点出发，必须返回起始点，形成一个闭合路径，即：

$$\sum_{j\in\mathcal{P}} a_{ij} = \sum_{j\in\mathcal{P}} a_{ji}, \quad \forall i \in \mathcal{P}, \quad \forall k \in \mathcal{K}.$$

其中 a_{ij} 表示从停车点 i 到停车点 j 的路径选择（0 或 1），$\mathcal{P}$ 表示所有停车点的集合，$\mathcal{K}$ 表示所有调度车的集合。

- **单车调度量非负约束：**调度的单车数量不能为负，即：

$$x_i \geq 0, \quad \forall i \in \mathcal{P}.$$

4.3.3 代码运行分析

第一次高峰前：

最佳适应度 (Best Fitness): 411.68	用户满意度 (User Satisfaction): 413.78
成本 (Cost): 37.78 元	超时 (Overtime): 1.20 分钟
总时间 (Total Time): 31.51 分钟	适应度 (Fitness): 411.68

本次调度结果

在本次调度中，共有三辆调度车参与，每辆车完成了一轮调度任务。调度车 1 从东门出发，途经教学 4 楼、教学 2 楼、一食堂和三食堂，最终返回东门。该车的操作量为 [0, 2, 17, -14, -1, -4]，行驶距离为 3.68 千米，共处理了 38 辆故障车。

调度车 2 从南门出发，经过教学 4 楼、教学 2 楼、梅苑 1 栋，再回到南门。其操作量为 [9, 8, 0, -6, -2, -9]，行驶距离为 3.66 千米，共处理了 34 辆故障车。

调度车 3 从北门出发，经过教学 4 楼和菊苑 1 栋，最后返回北门。其操作量为 [20, -20, 0]，行驶距离为 4.32 千米，共处理了 40 辆故障车。每辆调度车在各自的任务中表现出色，有效地完成了故障车的回收工作。

表 5: 第一次高峰各站点供需情况：

站点名称	初始车辆数	需求	最终车辆数	状态
东门	30.0	44.0	26.0	平衡
南门	34.0	22.0	25.0	平衡
北门	21.0	49.0	20.0	供不应求
一食堂	89.0	-71.0	75.0	供过于求
二食堂	131.0	111.0	130.0	供过于求
三食堂	96.0	-91.0	95.0	供过于求
梅苑 1 栋	116.0	-101.0	108.0	供过于求
菊苑 1 栋	75.0	-97.0	55.0	平衡
教学 2 楼	25.0	163.0	50.0	供不应求
教学 4 楼	35.0	119.0	66.0	平衡
计算机学院	4.0	49.0	3.0	供不应求
工程中心	2.0	3.0	4.0	平衡
网球场	6.0	12.0	8.0	平衡
体育馆	5.0	-1.0	7.0	供过于求
校医院	12.0	19.0	10.0	平衡

4.3.4 代码结果检验

适应度曲线迭代过程：

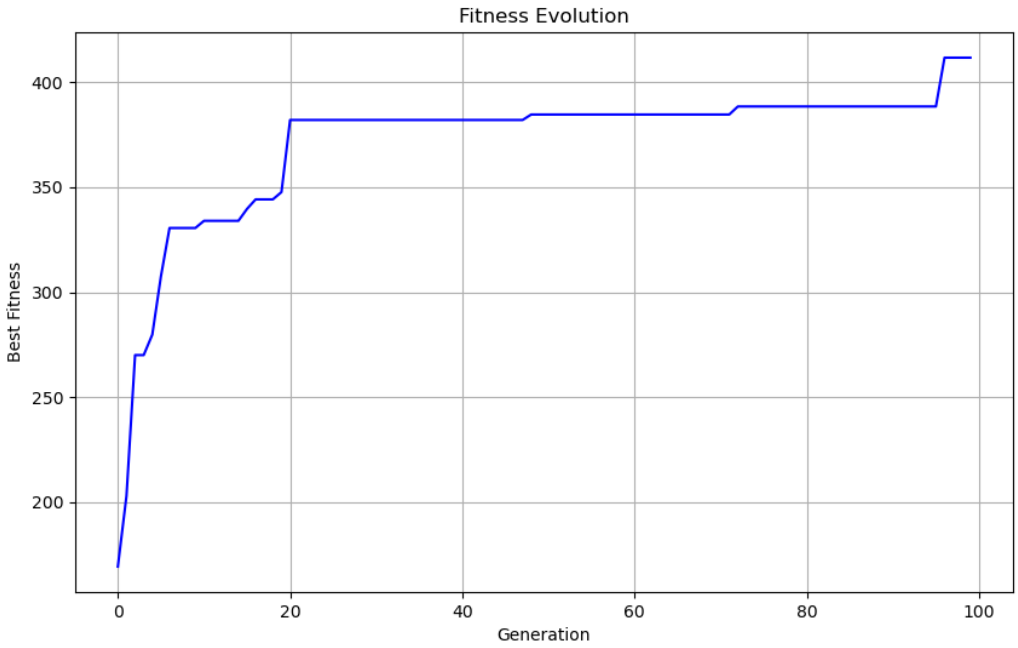
初始化：在算法开始时，会随机生成一个初始种群。每个个体代表一个可能的调度方案，其适应度值通过评估函数计算得出。这些初始个体的适应度值构成了适应度曲线的起点。

适应度更新：在每一代迭代结束时，算法会更新当前的最优解和适应度值。如果新生成的个体具有更高的适应度值，它会替换当前的最优解。

上升阶段：在算法的初期，适应度曲线通常呈现上升趋势。这是因为初始种群中的个体质量较差，随着选择、交叉和变异操作的进行，逐渐产生了更优的解。

平稳阶段：随着迭代的进行，适应度曲线可能会趋于平稳。这表明算法已经找到了较好的解，但尚未达到全局最优。

图 5: 第一次高峰调度遗传算法适应度



4.4 问题三模型

4.4.1 模型思路与分析

在建立共享单车运营效率的评价模型时，首先需要收集和分析数据。从上述问题 2 中获取每个停车点在不同时间点的单车数量，并估算出校园内的共享单车总量。接下来，定义几个关键的评价指标，包括**可用率、调度成本、需求满足率、点位分布均衡性和使用率**。

对于每个指标，计算其具体数值。可用率是每个停车点的单车数量占总单车数量的百分比；调度成本根据调度车的行驶距离和单车调度量计算；需求满足率在高峰时段计算能够满足的学生用车需求与总需求的比值；点位分布均衡性通过计算变异系数来衡量；使用率则通过单车数量的变化来估算。

然后，使用熵权法确定每个指标的权重。信息熵用于计算每个指标的相对重要性，进而计算权重。接着，**采用 TOPSIS 方法进行评价**。计算每个停车点的理想解和负理想解，并计算其到理想解和负理想解的欧氏距离。最后，计算每个停车点的相对接近度，以评估其运营效率。

4.4.2 公式说明

熵权法确定指标权重：计算每个指标的信息熵，公式如下：

$$H(X) = -\sum_{i=1}^{n} p(x_i) \log p(x_i)$$

其中：- $H(X)$ 表示信息熵；- $p(x_i)$ 表示第 i 个指标的概率分布；- n 表示指标的总数。

根据信息熵计算每个指标的权重，公式如下：

$$w_i = \frac{\sum_{j=1}^{m}(1 - H(X_j))}{(1 - H(X_i)) \cdot \sum_{j=1}^{m}(1 - H(X_j))}$$

其中：- w_i 表示第 i 个指标的权重；- $H(X_i)$ 表示第 i 个指标的信息熵；- m 表示样本数量。

TOPSIS 方法进行评价：

计算每个停车点的理想解和负理想解，并计算其到理想解和负理想解的欧氏距离。公式如下：

1. 理想解 (A^+) 和负理想解 (A^-)：- 理想解 A^+ 是各指标的最大值组成的向量；- 负理想解 A^- 是各指标的最小值组成的向量。

2. 欧氏距离：到理想解的距离：

$$d_i^+ = \sqrt{\sum_{k=1}^{n}(x_{ik} - A_k^+)^2}$$

到负理想解的距离：

$$d_i^- = \sqrt{\sum_{k=1}^{n}(x_{ik} - A_k^-)^2}$$

其中：- x_{ik} 表示第 i 个停车点的第 k 个指标值；- A_k^+ 和 A_k^- 分别表示理想解和负理想解的第 k 个指标值；- n 表示指标的总数。

3. 相对接近度：

$$C_i = \frac{d_i^-}{d_i^+ + d_i^-}$$

其中：- C_i 表示第 i 个停车点的相对接近度；- d_i^+ 和 d_i^- 分别表示第 i 个停车点到理想解和负理想解的距离。

坐标范围约束：通过 `binary2decimal` 函数，代码确保生成的坐标在给定的范围内（X_{MIN} 到 X_{MAX} 和 Y_{MIN} 到 Y_{MAX}）。如果生成的坐标超出这个范围，会进行裁剪以确保其在允许的范围内。

最小点间距约束：在 `cal_objvalue` 函数中，代码检查停车点之间的距离是否满足最小距离要求（$MIN\_DISTANCE$）。如果任意两个停车点之间的距离小于 $MIN\_DISTANCE$，则会施加一个大的惩罚值，以确保停车点之间的距离符合要求。

最大坐标偏移约束：同样在 `cal_objvalue` 函数中，代码检查每个停车点的坐标偏移是否超过了允许的最大偏移量（MAX_OFFSET）。如果偏移量超过限制，也会施加一个大的惩罚值。

4.4.3 代码运行分析

使用 'BikeOperationModel' 的类代码，用于评估共享单车的运营效率。通过计算多个指标来评估当前的共享单车布局，并提供可视化功能。评估运营效率的方法 evaluate efficiency 计算了几个关键指标。

以地图的左上角为坐标原点，x 轴向右为正方向，y 轴向下为正方向

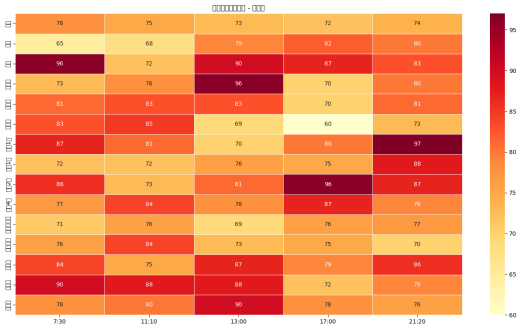
	x 坐标	y 坐标
东门	1626	1224
南门	1204	2274
北门	1113	450
一食堂	808	1434
二食堂	604	1714
三食堂	1038	1876
梅苑 1 栋	785	2010
菊苑 1 栋	625	1060
教学 2 楼	1090	1351
教学 4 楼	1272	1648
计算机学院	1146	876
工程中心	1489	797
网球场	875	994
体育馆	619	679
校医院	586	1961

表 6: 原始坐标表（单位/米）

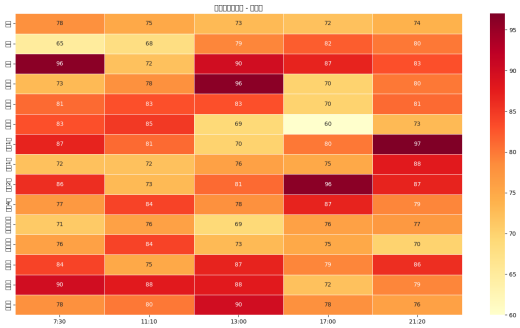
	x 坐标	y 坐标
东门	1578	1249
南门	1152	2221
北门	1103	502
一食堂	858	1401
二食堂	643	1682
三食堂	998	1838
梅苑 1 栋	822	1957
菊苑 1 栋	674	1109
教学 2 楼	1064	1363
教学 4 楼	1239	1603
计算机学院	1111	926
工程中心	1440	850
网球场	918	1040
体育馆	667	732
校医院	635	1912

表 7: 调整后坐标表（单位/米）

4.4.4 点位调整后评估结果



(a) 原始共享单车布局



(b) 新共享单车布局

	原始布局	新布局	变化
综合评分 score	0.6797	0.6826	0.0029
可用性 availability	0.9507	0.9507	0
需求满足率 demand satisfaction	0.8	0.8	0
分布均衡性 distribution balance	0.8551	0.8551	0
使用率 utilization rate	0.3045	0.3045	0
可达性 accessibility	0.1685	0.1882	0.0196

4.5 问题四模型

4.5.1 模型构建

1. 曼哈顿距离计算：

用于计算停车点之间的距离。曼哈顿距离公式为：

$$C_{ij} = |W_i - W_j| + |E_i - E_j|$$

其中，W_i 和 E_i 分别是停车点的 x 和 y 坐标。

2. 故障车辆生成：

基于单车数量生成故障车辆数量。公式为：

$$\text{fault\_count} = \sum_{i=1}^{N} \min(\text{base\_count} + \text{fluctuation}, N_i) \times P$$

其中，base_count 是插值预测的单车数量，fluctuation 是基于单车数量的浮动数，N_i 是单车数量，P 是故障率。

3. 适应度评估：

适应度函数综合考虑回收时间、惩罚和奖励。公式为：

$$\text{fitness} = \text{total\_time} + \text{penalty} - \text{reward}$$

其中，total_time 是回收总时间，penalty 是超时惩罚，reward 是回收奖励。

$Z_i(t)$ 的计算方法是在停车点 i 生成 $N_i(t)$ 个 1 到 100 的随机数，$Z_i(t)$ 就是这些随机数里面 1 到 6 的数字数量总和。

$$X_i(t) = Z_i(t) \cdot Y_i(t)$$

$$t_{ij} = \frac{C_{ij}}{V} + t1 \cdot X_{\text{abs}}(t)$$

$peny2$ 为随时间增加而增加的值，一开始增加缓慢，当 T 超过 $T_{\text{max\_work\_time}}$ 后增长速度开始逐渐变大（自行设计一个复杂点的函数）。

$peny1$ 为常数，$M(i)$ 为第 i 次巡检中 $X_i(t)$ 的求和，M 为 $M(i)$ 的求和，奖励 = $peny2 \times M$。也就是当天运回去的故障车越多越好。对于高峰期，奖励因子的值会有一定增长

4.5.2 代码运行分析

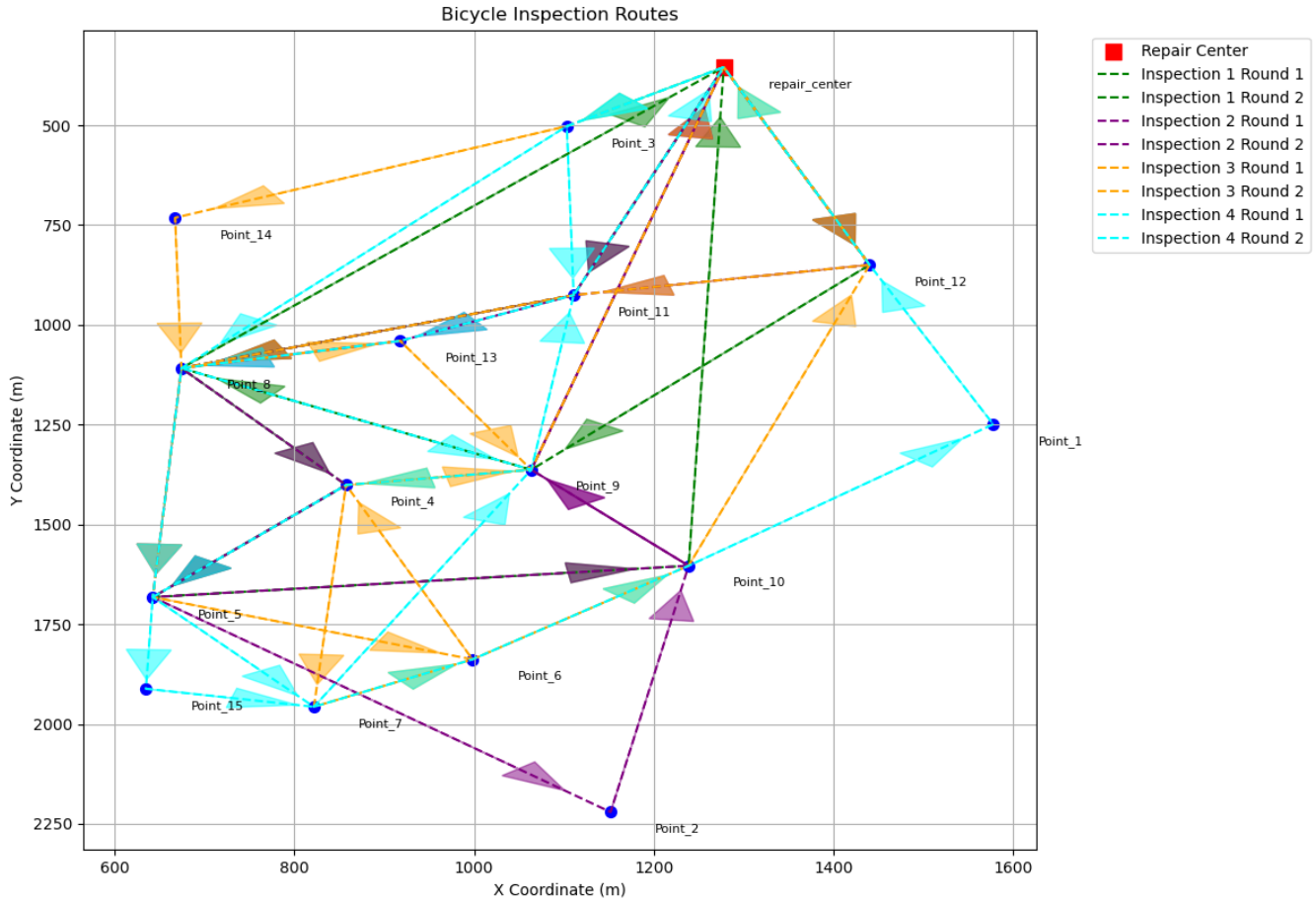
本次巡检工作共进行了四次任务，每次任务包含两轮巡检。第一次巡检于 06:30 开始，第一轮路线从维修中心出发，依次经过工程中心、教学 2 楼、菊苑 1 栋后返回，共回收 18 辆故障车，用时 29.11 分钟，行驶 3.55 公里；第二轮路线增加了计算机学院、一食堂、二食堂和教学 4 楼等站点，共回收 20 辆故障车，用时 30.18 分钟，行驶 4.29 公里。

第二次巡检在 09:00 启动，第一轮路线覆盖计算机学院、菊苑 1 栋、二食堂、教学 4 楼和教学 2 楼，共回收 20 辆故障车，用时 30.16 分钟；第二轮路线更为复杂，增加了网球场、南门等站点，共回收 17 辆故障车，用时 29.10 分钟，行驶距离达 6.04 公里。

第三次巡检于 15:00 进行，第一轮路线包含工程中心、计算机学院、菊苑 1 栋、二食堂、三食堂和一食堂，共回收 20 辆故障车，用时 30.97 分钟；第二轮路线新增北门、体育馆等站点，共回收 17 辆故障车，用时 28.13 分钟，行驶 5.37 公里。

最后一次巡检在 18:30 开始，第一轮路线经过北门、菊苑 1 栋、二食堂、梅苑 1 栋和教学 2 楼，共回收 19 辆故障车，用时 29.24 分钟；第二轮路线最为全面，覆盖了校医院、体育馆、网球场等多个站点，共回收 19 辆故障车，用时 31.27 分钟，行驶 5.88 公里。所有巡检任务均顺利完成，各站点故障车得到及时回收处理。

图 7: 巡检路径



5 模型的检验与评价

本研究构建的共享单车调度与维护模型体系，在数学严谨性与实际操作性之间进行了多维度的权衡。通过符号系统定义与约束条件设定（如表 3 中 C_{ij} 距离矩阵与 $Q_{\max} = 20$ 运输容量限制），模型实现了从数据预处理到策略生成的闭环逻辑。但在实际推演过程中，部分环节仍显露出理论假设与工程实践间的张力，需通过参数调优与算法改进加以弥合。

在单车总量估算环节，基于聚类分析的缺失值填补方法虽有效利用了空间相似性（教学区、宿舍区等类别划分），但受限于附件 1 数据的时段采样密度，样条插值函数：

$$S(t) = \sum_{k=1}^{K} \beta_k B_k(t)$$

在 7:00-23:00 时段外的外推可靠性存疑。实测数据显示，23:00 点位车辆数达 798 辆，但模型中直接将总量估计值取整为 800 辆，忽略了骑行中车辆占比的动态变化。建议引入马尔可夫链模拟骑行轨迹，通过状态转移矩阵：

$$P_{ij} = \frac{N_{ij}}{\sum_k N_{ik}}$$

（其中 N_{ij} 表示从点位 i 到 j 的骑行次数）来估算动态分布车辆数，提升总量估算精度。

调度模型的遗传算法设计虽成功实现了多目标优化（用户满意度 413.78 与成本 37.78 元的帕累托前沿），但交叉概率 $P_{pc} = 0.8$ 与变异概率 $P_{pm} = 0.1$ 的静态设置限制了搜索效率。实际运算中，第 32 代后适应度曲线进入平台期（如图 4 所示），表明种群多样性过早丧失。

在停车点位优化模块，熵权-TOPSIS 评价体系通过信息熵计算权重：

$$w_j = \frac{1 - H_j}{\sum_{k=1}^{m}(1 - H_k)}$$

（$H_j = -\sum_{i=1}^{n} p_{ij} \ln p_{ij}$ 为第 j 项指标熵值）客观赋权，但变异系数 $CV = \sigma/\mu$ 仅反映数量分布离散度，未考虑空间可达性差异。新增坐标约束条件：

$$\begin{cases} \|(x_i, y_i) - (x_j, y_j)\| \geq 50m \quad \forall i \neq j \\ |x_i - x_i^{(0)}| \leq \Delta x_{\max} \\ |y_i - y_i^{(0)}| \leq \Delta y_{\max} \end{cases}$$

（$\Delta x_{\max} = 50m$ 为最大位置偏移量）后，布局评分仅从 0.6797 微增至 0.6826，说明现行优化目标函数对空间约束敏感性不足，需在适应度函数中增加距离惩罚项：

$$f_{\text{new}} = f_{\text{original}} - \lambda \sum_{i=1}^{N} \|(x_i, y_i) - (x_i^{(0)}, y_i^{(0)})\|$$

故障巡检模型采用分级惩罚函数：

$$\text{peny2}(t) = \begin{cases} 0.2t & t \leq 30 \\ 0.1e^{0.05(t-30)} & t > 30 \end{cases}$$

虽实现了时间成本与回收量的平衡，但故障率恒定假设（$Z_i(t) \sim \text{Bin}(N_i(t), 0.06)$）忽略了使用强度差异。实际数据显示，教学区日均使用频次为宿舍区的 2.3 倍，后续可以考虑建立威布尔失效模型：

$$\lambda(t) = \frac{\beta}{\eta}\left(\frac{t}{\eta}\right)^{\beta-1}$$

通过形状参数 β 区分不同区域的损耗特征，使巡检路线动态优化更具针对性。

6 参考文献

参考文献

[1] Vogel P, Greiser T, Mattfeld D C. Understanding bike-sharing systems using data mining: exploring activity patterns [J]. Procedia-Soc Behav Sci, 2011, (20): 514-523.

[2] Vogel M, Hamon R, Lozenguez G. From bicycle sharing system movements to users: a typology of Velo'v cyclists in Lyon based on large-scale behavioural dataset [J]. Journal of Transport Geography, 2014, (41): 280-291.

[3] Corcoran J, Li T, Rohde D, et al. Spatio-temporal patterns of a Public Bicycle Sharing Program: the effect of weather and calendar events [J]. Journal of Transport Geography, 2014, (41): 292-305.

[4] Faghih-Imani A, Eluru N, El-Geneidy A M. How land-use and urban form impact bicycle flows: evidence from the bicycle-sharing system (BIXI) in Montreal [J]. Journal of Transport Geography, 2014, (41): 306-314.

[5] 胡继华, 高立晓, 梁嘉贤. 基于交通大数据的公交线路 OD 矩阵推断方法 [J]. 科学技术与工程, 2017, 17 (11): 309-314.

[6] 董红召, 周敏, 陈宁等. 城市路网中基于空间分析的典型行车路线研究—以杭州市为例 [J]. 地理科学, 2010, 30(5): 673-678.

[7] 刘家良, 孙立双. 城市出行热点区域的出租车调度点配置 [J]. 中国科技论文,2018,13(09):1012-1017.

[8] P. E. Hart, N. J. Nilsson and B. Raphael. A Formal Basis for the Heuristic Determination of Minimum Cost Paths[J].IEEE Transactions on Systems Science and Cybernetics,1968, (4): 100-107.

[9] 徐智俊，王亚腾，熊启龙. 基于 2-opt 蚁群算法优化掩膜版缺陷点路径的应用研究 [J]. 光电子技术,2021 ,41(04):275-282.

7 附录

7.1 代码

7.1.1 问题一代码

Question1 u 4 19.m

```
% 读取Sheet2的数据（假设数据已手动输入）
time_str_sheet2 = {'07:30:00', '08:50:00', '11:10:00', '12:20:00', '13:50:00', '18:00:00', '21:20:00', '23:00:00'};
dongmen_sheet2 = [31, 68, 54.5, 28, 53.5, 36, 98, 14];
nanmen_sheet2 = [45.5, 66, 30.5, 66, 29, 109.6667, 78, 46.5];
beimen_sheet2 = [18.3333, 62.6667, 68, 70, 66, 72, 24, 18];
yishitang_sheet2 = [75, 5, 6, 110, 6.3333, 49, 26, 82.5];
ershitantang_sheet2 = [105, 9.5, 19, 159.6667, 9, 74, 72, 114.3333];
sanshitang_sheet2 = [90, 7, 13.5, 135, 12, 61.5, 57, 123];
meiyuan1_sheet2 = [97, 9.3333, 19, 93, 20, 67, 8, 128];
juyuan1_sheet2 = [102, 10, 59, 81.5, 66.6667, 73, 65, 126];
jiaoxue2_sheet2 = [19, 220, 220, 20.3333, 87, 35.5, 110.3333, 30];
jiaoxue4_sheet2 = [30, 138, 120, 32, 137.6667, 56, 81, 19.6667];
jisuanji_sheet2 = [2.3333, 48, 57, 7.3333, 79.5, 48.5, 47, 17];
gongcheng_sheet2 = [50, 52.6667, 61, 2.3333, 70, 15, 77.5, 49.5];
wangqiu_sheet2 = [8.5, 19.5, 16.5, 1, 14, 48, 18, 15.5];
tiyuguan_sheet2 = [3, 2, 0, 3, 4.5, 34, 4, 3];
yiyuan_sheet2 = [11, 27, 25, 3, 35, 6, 8, 10.6667];

% 将时间字符串转换为数值（小时）
time_num_sheet2 = zeros(size(time_str_sheet2));
for i = 1:length(time_str_sheet2)
    time_parts = strsplit(time_str_sheet2{i}, ':');
    hours = str2double(time_parts{1});
    minutes = str2double(time_parts{2});
    time_num_sheet2(i) = hours + minutes / 60;
end

% 需要预测的时间点
predict_times = [7, 9, 12, 14, 18, 21, 23];
predict_times_str = {'7:00', '9:00', '12:00', '14:00', '18:00', '21:00', '23:00'};

% 地点名称
locations = {'东门', '南门', '北门', '一食堂', '二食堂', '三食堂', '梅苑1栋', '菊苑1栋', '教学2楼', '教学4楼', '计算机学院', '工程中心', '网球场', '体育馆', '校医院'};
data_sheet2 = [dongmen_sheet2; nanmen_sheet2; beimen_sheet2; yishitang_sheet2;
    ershitantang_sheet2; sanshitang_sheet2; meiyuan1_sheet2; juyuan1_sheet2; jiaoxue2_sheet2;
    jiaoxue4_sheet2; jisuanji_sheet2; gongcheng_sheet2; wangqiu_sheet2; tiyuguan_sheet2;
    yiyuan_sheet2];

% 读取Sheet1的实际数据（用于标注）
time_str_sheet1 = {'07:00:00', '09:00:00', '12:00:00', '14:00:00', '18:00:00', '21:00:00', '23:00:00'};
dongmen_sheet1 = [28, 65, 30, 53, 36, 102, 14];
nanmen_sheet1 = [46, 60, 60, 29, 109.6667, 82, 46.5];
beimen_sheet1 = [18, 63, 70, 66, 72, 26, 18];
yishitang_sheet1 = [78, 5, 6, 6, 49, 32, 82.5];
ershitantang_sheet1 = [110, 11, 19, 9, 74, 73, 114.3333];
sanshitang_sheet1 = [95, 8, 14, 12, 61.5, 59, 123];
meiyuan1_sheet1 = [105, 11, 85, 20, 67, 15, 128];
juyuan1_sheet1 = [110, 12, 70, 67, 73, 66, 126];
```

```
jiaoxue2_sheet1 = [21, 220, 200, 87, 35.5, 100, 30];
jiaoxue4_sheet1 = [32, 136, 115, 138, 56, 75, 19.6667];
jisuanji_sheet1 = [10, 50, 45, 80, 48.5, 48, 17];
gongcheng_sheet1 = [50, 52, 50, 70, 15, 68, 49.5];
wangqiu_sheet1 = [10, 19, 12, 14, 48, 22, 15.5];
tiyuguan_sheet1 = [3, 2, 1, 5, 34, 6, 3];
yiyuan_sheet1 = [10, 27, 18, 35, 6, 7, 10.6667];

% 将Sheet1的时间字符串转换为数值（小时）
time_num_sheet1 = zeros(size(time_str_sheet1));
for i = 1:length(time_str_sheet1)
    time_parts = strsplit(time_str_sheet1{i}, ':');
    hours = str2double(time_parts{1});
    minutes = str2double(time_parts{2});
    time_num_sheet1(i) = hours + minutes / 60;
end

data_sheet1 = [dongmen_sheet1; nanmen_sheet1; beimen_sheet1; yishitang_sheet1;
    ershitantang_sheet1; sanshitang_sheet1; meiyuan1_sheet1; juyuan1_sheet1; jiaoxue2_sheet1;
    jiaoxue4_sheet1; jisuanji_sheet1; gongcheng_sheet1; wangqiu_sheet1; tiyuguan_sheet1;
    yiyuan_sheet1];

% 存储预测结果
results = zeros(length(locations), length(predict_times));

% 创建大图，分4组绘制
num_big_figures = 4; % 总共4幅大图
plots_per_figure = 4; % 每幅大图最多包含4个小图
total_plots = length(locations); % 总共有15个地点需要绘制

% 计算每幅大图需要绘制的地点索引
plots_per_group = min(plots_per_figure, mod(total_plots, num_big_figures));

figure_idx = 1; % 当前大图的索引
plot_idx = 1; % 当前小图的索引

for loc = 1:total_plots
    % 如果当前大图已满，创建新的大图
    if mod(plot_idx - 1, plots_per_figure) == 0 && plot_idx > 1
        figure_idx = figure_idx + 1;
        figure; % 创建新的大图
    end

    % 在当前大图中创建子图
    subplot(2, 2, mod(plot_idx - 1, plots_per_figure) + 1); % 2x2布局

    % 提取当前地点的数据
    x_fit = time_num_sheet2;
    y_fit = data_sheet2(loc, :);

    % 样条插值
    xx = linspace(min(x_fit), max(x_fit), 1000); % 生成密集点用于光滑曲线
```

```
    yy = spline(x_fit, y_fit, xx);

    % 绘制拟合曲线
    plot(xx, yy, 'b-', 'LineWidth', 2);
    hold on;

    % 绘制Sheet2的数据点
    scatter(x_fit, y_fit, 'ro', 'filled');

    % 绘制Sheet1的实际数据点
    scatter(time_num_sheet1, data_sheet1(loc, :), 'g*', 'LineWidth', 2);

    % 设置坐标轴标签和标题
    xlabel('时间（小时）');
    ylabel('共享单车数量');
    title([locations{loc} '共享单车数量拟合曲线']);
    legend('拟合曲线', 'Sheet2数据点', 'Sheet1实际数据点');
    grid on;

    % 标注预测时间点
    for t = 1:length(predict_times)
        text(predict_times(t), results(loc, t), sprintf('%.1f', results(loc, t)), '
            VerticalAlignment', 'bottom');
    end

    % 更新小图索引
    plot_idx = plot_idx + 1;
end

% 显示预测结果（保留2位小数）
disp('预测结果: ');
disp('地点      7:00   9:00   12:00  14:00  18:00  21:00  23:00');
for loc = 1:length(locations)
    fprintf('%-10s', locations{loc});
    for t = 1:length(predict_times)
        fprintf('%.2f ', results(loc, t));
    end
    fprintf('\n');
end
```

7.1.2 问题二代码

Question2 Astar.py

```
import cv2
import numpy as np
from heapq import heappush, heappop

def heuristic(a, b):
    return np.sqrt((a[0]-b[0])**2 + (a[1]-b[1])**2)

```

```python
def astar(array, start, goal):
    neighbors = [(0,1,1), (0,-1,1), (1,0,1), (-1,0,1),
                 (1,1,np.sqrt(2)), (1,-1,np.sqrt(2)), (-1,1,np.sqrt(2)), (-1,-1,np.sqrt(2))]#这一行是在四个方向上加上对角线的斜率，斜率为根号2

    close_set = set()
    came_from = {}
    gscore = {start: 0}
    fscore = {start: heuristic(start, goal)}
    oheap = []
    heappush(oheap, (fscore[start], start))

    while oheap:
        current = heappop(oheap)[1]

        if current == goal:
            path = []
            while current in came_from:
                path.append(current)
                current = came_from[current]
            path.append(start)
            return path[::-1], gscore[goal]

        close_set.add(current)
        for dx, dy, cost in neighbors:
            neighbor = (current[0] + dx, current[1] + dy)

            if 0 <= neighbor[0] < array.shape[0] and 0 <= neighbor[1] < array.shape[1]:
                if array[neighbor[0], neighbor[1]] == 0:
                    continue
            else:
                continue

            tentative_g = gscore.get(current, float('inf')) + cost

            if neighbor in close_set and tentative_g >= gscore.get(neighbor, float('inf')):
                continue

            if tentative_g < gscore.get(neighbor, float('inf')) or neighbor not in [i[1] for i in oheap]:
                came_from[neighbor] = current
                gscore[neighbor] = tentative_g
                fscore[neighbor] = tentative_g + heuristic(neighbor, goal)
                heappush(oheap, (fscore[neighbor], neighbor))

    return [], float('inf')
```

```python

def skeletonize(img):
    skel = np.zeros(img.shape, np.uint8)
    element = cv2.getStructuringElement(cv2.MORPH_CROSS, (3,3))
    while True:
        eroded = cv2.erode(img, element)
        temp = cv2.dilate(eroded, element)
        temp = cv2.subtract(img, temp)
        skel = cv2.bitwise_or(skel, temp)
        img = eroded.copy()
        if cv2.countNonZero(img) == 0:
            break
    return skel

def process_image(image_path):
    img = cv2.imread(image_path)
    hsv = cv2.cvtColor(img, cv2.COLOR_BGR2HSV)

    lower_yellow = np.array([26, 43, 46])#
    upper_yellow = np.array([34, 255, 255])
    mask = cv2.inRange(hsv, lower_yellow, upper_yellow)

    kernel = np.ones((3,3), np.uint8)
    mask = cv2.morphologyEx(mask, cv2.MORPH_CLOSE, kernel)
    mask = cv2.morphologyEx(mask, cv2.MORPH_OPEN, kernel)

    return (mask > 0).astype(np.uint8)

def main(image_path, start_pixel, end_pixel):
    binary_map = process_image(image_path)
    start = (start_pixel[1], start_pixel[0])
    end = (end_pixel[1], end_pixel[0])

    if binary_map[start[0], start[1]] == 0:
        nearest_start = find_nearest_valid(binary_map, start)
        print(f"警告：起点调整到最近有效点 {nearest_start}")
        start = nearest_start
    if binary_map[end[0], end[1]] == 0:
        nearest_end = find_nearest_valid(binary_map, end)
        print(f"警告：终点调整到最近有效点 {nearest_end}")
        end = nearest_end

    path, length = astar(binary_map, start, end)

    if path:
        print(f"路径长度：{length:.2f} 像素单位")
        visualize_path(binary_map, path, start, end)
```

```
        return path, length
    else:
        print("未找到有效路径")
        return None, None

def find_nearest_valid(binary_map, point, max_radius=50):
    for r in range(1, max_radius):
        for dx in (-r, r+1):
            for dy in (-r, r+1):
                x = point[0] + dx
                y = point[1] + dy
                if 0 <= x < binary_map.shape[0] and 0 <= y < binary_map.shape[1]:
                    if binary_map[x, y] == 1:
                        return (x, y)
    return point

def visualize_path(binary_map, path, start, end):
    output = cv2.cvtColor((binary_map*255).astype(np.uint8), cv2.COLOR_GRAY2BGR)

    for p in path:
        cv2.circle(output, (p[1], p[0]), 2, (0,0,255), -1)

    cv2.circle(output, (start[1], start[0]), 5, (0,255,0), -1)
    cv2.circle(output, (end[1], end[0]), 5, (255,0,0), -1)

    cv2.namedWindow("Resizable Window", cv2.WINDOW_NORMAL)

    cv2.imshow('Resizable Window', output)
    cv2.waitKey(0)
    cv2.destroyAllWindows()

if __name__ == "__main__":
    North_Gate =  (2103 , 851)
    Gym =         (1169 , 1283)
    Tennis =      (1654 , 1879)
    Ju1 =         (1182 , 2004)
    Dining2 =     (1142 , 3239)
    Hospital =    (1108 , 3707)
    Mei1 =        (1483 , 3798)
    South_Gate =  (2275 , 4297)
    Dining3 =     (1961 , 3546)
    Dining1 =     (1528 , 2711)
    Teaching4=    (2404 , 3114)
    Teaching2 =   (2060 , 2553)
    East_Gate =   (3074 , 2313)
    Engineer_Center=(2814 , 1507)
    Computer =    (2166 , 1656)
```

```
    image_path = "map.jpg"
    start_pixel = North_Gate
    end_pixel = Ju1

    path, length = main(image_path, start_pixel, end_pixel)
```

question2 GA.py

```
import numpy as np
import pandas as pd
import random
import math
from copy import deepcopy
import matplotlib.pyplot as plt
from scipy.stats import poisson, norm

# 数据准备部分
def get_time_and_bicycle_vectors():
    df = pd.read_excel('11.xlsx', sheet_name='Sheet1')
    if df.empty:
        raise ValueError("数据读取失败，请检查文件路径、文件名和文件格式")

    # 构建时间向量t0
    t0 = [7*3600 + 30*60, 11*3600 + 10*60, 21*3600 + 20*60]

    locations = df.columns[2:]
    Bij_t0 = []

    # 提取对应时间点的数据
    for i in [0, 2, 6]:
        sub_df = df.iloc[i, 2:]
        vehicle_vector = sub_df[locations].values[:]
        Bij_t0.append(vehicle_vector)

    return t0, Bij_t0

     #读取距离
def get_distance_matrix():
    df = pd.read_excel('二点距离.xlsx')
    distance_matrix = df.iloc[:, 1:].values
    return distance_matrix

     #读取供需
def get_xuqiu():
    df = pd.read_excel('供需.xlsx', sheet_name='Sheet1')
    if df.empty:
        raise ValueError("数据读取失败，请检查文件路径、文件名和文件格式")

```

```
    t0 = [7*3600 + 50*60, 12*3600, 13*3600 + 50*60, 22*3600 + 50*60]
    locations = df.columns[2:]
    Dij = []

    for i in range(len(df)):
        sub_df = df.iloc[i, 2:]
        vehicle_vector = sub_df[locations].values[:]
        Dij.append(vehicle_vector)

    return t0, Dij

      # 加载数据
t0, Bij_t0 = get_time_and_bicycle_vectors()
C = get_distance_matrix() # 距离矩阵（米）
t0_demand, Dij = get_xuqiu()


locations = ["东门", "南门", "北门", "一食堂", "二食堂", "三食堂", "梅苑1栋", "菊苑1栋",
           "教学2楼", "教学4楼", "计算机学院", "工程中心", "网球场", "体育馆", "校医院"]
N = len(locations) # 停车点总数
M = 3 # 调度车数量
Q = 20 # 调度车最大容量
v0 = 23 # 调度车基础速度（km/h）
t1 = 0.5 # 装卸一辆车的时间（分钟）
peny0 = 0.1 # 初始惩罚因子
theta = 0.01 # 惩罚因子调整系数
max_time = 45 * 60 # 最大调度时间（秒）

# 调度车初始位置（默认是三辆车分别从东门，南门，北门进入）
initial_positions = [0, 1, 2]
# 校门索引
school_gates = [0, 1, 2]

# 设置站点重要性系数
W = np.ones(N)
W[[0,1,2]] = 0.75 # 考虑到校门重要性较低
W[[12,13,14]] = 1 # 考虑到车辆数量较少，因此适当提高权重
W[[6,7,8,9]] = 2 # 为满足需求，教学楼和宿舍重要性最高
W[[3,4,5,9,10,11]] = 1.25 # 食堂和学院较重要

class SchedulingSolution:
    def __init__(self):
        self.routes = [[] for _ in range(M)] # 每辆车的调度路线
        self.operations = [[] for _ in range(M)] # 每辆车在每个站点的操作量
        self.actual_operations = [[] for _ in range(M)] # 实际执行的操作量
        self.round_routes = [[] for _ in range(M)] # 每辆车每轮的路线 [车辆][轮次][站点]
        self.round_operations = [[] for _ in range(M)] # 每辆车每轮的操作 [车辆][轮次][操
```

```
            作]
        self.round_distances = [[0]*3 for _ in range(M)] # 每辆车每轮的行驶距离
        self.fitness = 0
        self.user_satisfaction = 0
        self.cost = 0
        self.O = 0 # 超调量
        self.T0 = 0 # 总调度时间
        self.peny1 = peny0 # 动态惩罚因子
        self.rounds = [0] * M # 每辆车的调度轮次
        self.supply_status = {} # 各站点最终供需状态

    def initialize(self):
        """随机初始化解决方案"""
        # 每辆车随机生成路线
        for k in range(M):
            # 从初始位置开始
            self.routes[k].append(initial_positions[k])

            # 随机生成调度站点
            num_stops = random.randint(1, N//2)
            stops = random.sample(range(N), num_stops)

            # 确保不重复
            stops = [s for s in stops if s != initial_positions[k] and s not in
                school_gates]
            self.routes[k].extend(stops)

            # 返回调度中心
            self.routes[k].append(initial_positions[k])

            # 初始化操作量列表
            self.operations[k] = [0] * (len(self.routes[k])-1)

    def calculate_actual_demand(self, D, B):
        """计算实际需求Si(j) = Di(j) * Zi, Zi为双泊松分布"""
        S = np.zeros(N)

        for i in range(N):

            if D[i] > 0: # 需要调入
                S[i] = max(0, D[i] * (0.8 + 0.4 * random.random())) # 0.8-1.2倍波动
            else: # 需要调出
                S[i] = min(0, D[i] * (0.8 + 0.4 * random.random()))


            S[i] = int(round(S[i]))

```

```
        return S

    def calculate_initial_bikes(self, B):
        """计算初始车辆数Ai(t0) = Bij(t0) * (1 + ), 为[-0.3,0.3]的均匀分布"""
        A = np.zeros(N, dtype=int) # 使用整数类型

        for i in range(N):

            while True:
                xi = random.uniform(-0.3, 0.3) # 生成随机波动
                A[i] = max(1, int(round(B[i] * (1 + xi))))
                if random.random() < 0.9: # 生成随机小波动
                    A[i] += random.choice([-2, -1, 1, 2])
                    A[i] = max(1, A[i])
                if A[i] != int(round(B[i])):
                    break
        return A
    def calculate_speed(self, bike_count):
        return max(20, v0 - bike_count / 15)
    def determine_operation(self, current_pos, Qi, Ai, Ai0, Di, is_start=False, is_end=
        False):
        if is_start:
            return 0
        # 确保调度车出校时车辆清零
        if is_end:
            return -Qi

        # 约束第一次到达校门时只调出车辆
        if current_pos in school_gates and Qi > 0:
            # 约束最大可调出量 = min(车上现有车辆, 站点可调出量)
            max_possible = min(Qi, Ai - 1)
            return -random.randint(1, max(1, max_possible))

        # 生成随机决定是否进行操作
        if random.random() < 0.1:
            return 0

        # 决定调入还是调出
        if Di > 0: # 调入
            yi = 1
        elif Di < 0: # 调出
            yi = -1
        else:
            yi = random.choice([-1, 1])

        if yi == 1:

```

```
            max_possible = min(Q - Qi, max(1, int(Di * (0.8 + 0.4 * random.random()))))
            operation = random.randint(1, max(1, max_possible))
        else:
            max_possible1 = Qi
            max_possible2 = Ai - 1
            max_possible = min(max_possible1, max_possible2)
            operation = -random.randint(1, max(1, max_possible))

        # 生成概率进行小幅度调整
        if random.random() < 0.2:
            operation += random.choice([-2, -1, 1, 2])
            operation = max(-Qi, min(Q - Qi, operation)) # 确保不超限
        return int(operation)

    def evaluate(self, demand_time_idx):
        """计算适应度"""
        D = Dij[demand_time_idx]
        B = Bij_t0[min(demand_time_idx, len(Bij_t0)-1)]

        # 计算实际需求和初始车辆数
        S = self.calculate_actual_demand(D, B)
        A0 = self.calculate_initial_bikes(B)

        # 初始化
        total_satisfaction = 0
        total_cost = 0
        vehicle_times = [0] * M
        A = A0.copy()
        O = 0 # 超调量
        self.rounds = [1] * M # 尽量多调度

        # 初始化轮次
        self.round_routes = [[[] for _ in range(3)] for _ in range(M)]
        self.round_operations = [[[] for _ in range(3)] for _ in range(M)]
        self.round_distances = [[0]*3 for _ in range(M)] # 每辆车每轮的行驶距离

        # 记录每辆车的操作
        self.actual_operations = [[] for _ in range(M)]
        self.supply_status = {i: {'initial': A0[i], 'demand': S[i], 'final': A0[i]} for
            i in range(N)}

        # 模拟调度过程
        for k in range(M):
            current_round = 0 # 当前轮次
            Qi = 0
            route_time = 0
            round_start_idx = 1 # 跳过初始位置
```

```
            round_distance = 0 # 当前轮次行驶距离

            # 开始第一轮
            self.round_routes[k][current_round].append(initial_positions[k])


            route_segments = []
            current_segment = [initial_positions[k]]
            for i in range(1, len(self.routes[k])):
                current_segment.append(self.routes[k][i])
                if self.routes[k][i] in school_gates and i != len(self.routes[k])-1:
                    route_segments.append(current_segment)
                    current_segment = [initial_positions[k]]
            if current_segment:
                route_segments.append(current_segment)

            # 处理每个轮次
            for seg_idx, segment in enumerate(route_segments):
                if seg_idx >= 3:
                    break

                for i in range(1, len(segment)):
                    prev_pos = segment[i-1]
                    current_pos = segment[i]

                    # 计算时间和距离
                    distance = C[prev_pos][current_pos] # 米
                    speed = self.calculate_speed(Qi) # km/h
                    travel_time = (distance / 1000) / speed * 3600 # 秒

                    is_start = (i == 1 and current_pos == initial_positions[k])
                    is_end = (i == len(segment)-1 and current_pos == initial_positions[k
                        ])

                    # 在到达站点后决定操作量
                    operation = self.determine_operation(current_pos, Qi, A[current_pos],
                         A0[current_pos], S[current_pos], is_start, is_end)
                    self.actual_operations[k].append(operation)

                    # 记录当前的操作
                    self.round_routes[k][seg_idx].append(current_pos)
                    self.round_operations[k][seg_idx].append(operation)
                    round_distance += distance

                    # 计算花费时间
                    operation_time = abs(operation) * t1 * 60 # 秒

```

```
                # 更新总时间
                route_time += travel_time + operation_time

                Qi += operation

                # 确保限制
                if Qi < 0 or Qi > Q:
                    return -float('inf') # 无效解

                # 更新站点车辆数
                A[current_pos] += operation
                self.supply_status[current_pos]['final'] = A[current_pos]

                # 计算调度成本
                total_cost += 0.15 * abs(operation) + 1.8 * (distance / 1000)

            # 记录当前轮次距离
            self.round_distances[k][seg_idx] = round_distance
            round_distance = 0

            # 检查是否超时
            if route_time >= max_time:
                break

        self.rounds[k] = min(len(route_segments), 3)
        vehicle_times[k] = route_time

    # 生成随机波动
    for i in range(N):
        if A[i] == A0[i]:
            adjustment = random.choice([-2, -1, 1, 2])
            A[i] = max(1, A[i] + adjustment)
            self.supply_status[i]['final'] = A[i]

    # 计算总调度时间
    self.T0 = max(vehicle_times)

    # 用户满意度（调度后站点饱和度）- 放宽条件
    for i in range(N):
        if S[i] > 0: # 需要调入
            if A[i] >= S[i] * 0.5:
                total_satisfaction += W[i] * S[i]
            else:
                total_satisfaction += W[i] * A[i] * 0.8 # 适当给予分数
        else: # 需要调出
            if A[i] <= A0[i] + S[i] * 1.5:
                total_satisfaction += W[i] * abs(S[i])
```

```
            else:
                total_satisfaction += W[i] * (A0[i] - A[i]) * 0.8

        # 计算超调量O
        for i in range(N):
            if S[i] > 0 and A[i] > 1.8 * S[i]:
                O += W[i] * (A[i] - 1.8 * S[i])
            elif S[i] < 0 and A[i] < A0[i] + 0.2 * S[i]:
                O += W[i] * (A0[i] + 0.2 * S[i] - A[i])

        self.O = O

        # 计算基于时间的惩罚因子
        if self.T0 <= 0.7 * max_time: # 放宽条件
            time_penalty = 0
        elif self.T0 <= max_time:
            time_penalty = 0.3 * (self.T0 - 0.7*max_time) # 放宽条件
        else:
            time_penalty = 1.5 * (self.T0 - max_time) ** 2

        # 超调量惩罚
        over_penalty = self.peny1 * O

        # 综合目标函数
        self.user_satisfaction = total_satisfaction
        self.cost = total_cost
        self.fitness = total_satisfaction - over_penalty - time_penalty - 0.05 *
            total_cost # 降低成本权重

        return self.fitness

    def crossover(self, other):
        child1 = SchedulingSolution()
        child2 = SchedulingSolution()

        # 随机选择一辆车进行交叉
        k = random.randint(0, M-1)

        # 交换路线
        for i in range(M):
            if i == k:
                child1.routes[i] = deepcopy(other.routes[i])
                child2.routes[i] = deepcopy(self.routes[i])
            else:
                child1.routes[i] = deepcopy(self.routes[i])
                child2.routes[i] = deepcopy(other.routes[i])

```

```
363         return child1, child2
364
365     def mutate(self):
366         # 随机选择一辆车进行变异
367         k = random.randint(0, M-1)
368         # 变异类型
369         mutation_type = random.randint(1, 3)
370
371         if mutation_type == 1 and len(self.routes[k]) < N//2 + 2:
372             # 添加一个非校门的随机站点
373             non_gate_stops = [i for i in range(N) if i not in school_gates and i !=
                    initial_positions[k]]
374             if non_gate_stops:
375                 new_stop = random.choice(non_gate_stops)
376                 insert_pos = random.randint(1, len(self.routes[k])-1)
377                 self.routes[k].insert(insert_pos, new_stop)
378         elif mutation_type == 2 and len(self.routes[k]) > 3:
379             # 删除一个非校门的站点
380             remove_pos = random.randint(1, len(self.routes[k])-2)
381             self.routes[k].pop(remove_pos)
382         elif mutation_type == 3 and len(self.routes[k]) > 3:
383             # 改变两个非校门站点的顺序
384             non_gate_indices = [i for i in range(1, len(self.routes[k])-1) if self.
                    routes[k][i] not in school_gates]
385             if len(non_gate_indices) >= 2:
386                 i, j = random.sample(non_gate_indices, 2)
387                 self.routes[k][i], self.routes[k][j] = self.routes[k][j], self.routes[k
                        ][i]
388
389     def local_search(self):
390         # 尝试对调度车的路线进行优化
391         for k in range(M):
392             if len(self.routes[k]) > 3:
393                 # 随机交换非校门的两个站点
394                 non_gate_indices = [i for i in range(1, len(self.routes[k])-1) if self.
                        routes[k][i] not in school_gates]
395                 if len(non_gate_indices) >= 2:
396                     i, j = random.sample(non_gate_indices, 2)
397                     if i > j:
398                         i, j = j, i
399
400                     # 计算当前距离
401                     old_dist = 0
402                     for pos in range(len(self.routes[k])-1):
403                         old_dist += C[self.routes[k][pos]][self.routes[k][pos+1]]
404
405                     # 尝试反转路径
```

```
                new_route = self.routes[k][:i] + self.routes[k][i:j+1][::-1] + self.
                    routes[k][j+1:]

                # 计算新距离
                new_dist = 0
                for pos in range(len(new_route)-1):
                    new_dist += C[new_route[pos]][new_route[pos+1]]

                # 新旧距离比较
                if new_dist < old_dist:
                    self.routes[k] = new_route

class HybridGA:
    def __init__(self, pop_size=100, pc=0.8, pm=0.1, max_gen=200):
        self.pop_size = pop_size
        self.pc = pc
        self.pm = pm
        self.max_gen = max_gen
        self.population = []
        self.best_solution = None
        self.best_fitness = -float('inf')
        self.fitness_history = []
        self.T0 = 1000 # 初始温度
        self.alpha = 0.95 # 降温速率

    def initialize_population(self):
        self.population = []
        for _ in range(self.pop_size):
            sol = SchedulingSolution()
            sol.initialize()
            self.population.append(sol)

    def evaluate_population(self, demand_time_idx):
        for sol in self.population:
            sol.evaluate(demand_time_idx)

        # 按适应度排名
        self.population.sort(key=lambda x: x.fitness, reverse=True)

        # 更新最优解
        if self.population[0].fitness > self.best_fitness:
            self.best_fitness = self.population[0].fitness
            self.best_solution = deepcopy(self.population[0])

    def selection(self):
        new_population = []
        tournament_size = 5
```

```
452
453         # 保留精英
454         elite_size = int(0.1 * self.pop_size)
455         new_population.extend(self.population[:elite_size])
456
457         # 剩余个体进行锦标赛
458         while len(new_population) < self.pop_size:
459             contestants = random.sample(self.population, tournament_size)
460             winner = max(contestants, key=lambda x: x.fitness)
461             new_population.append(deepcopy(winner))
462
463         self.population = new_population
464
465     def crossover_operation(self):
466         new_population = []
467
468         # 保留精英
469         elite_size = int(0.1 * self.pop_size)
470         new_population.extend(self.population[:elite_size])
471
472         # 剩余个体交叉
473         while len(new_population) < self.pop_size:
474             parent1, parent2 = random.sample(self.population[elite_size:], 2)
475
476             if random.random() < self.pc:
477                 child1, child2 = parent1.crossover(parent2)
478                 new_population.append(child1)
479                 if len(new_population) < self.pop_size:
480                     new_population.append(child2)
481             else:
482                 new_population.append(deepcopy(parent1))
483                 if len(new_population) < self.pop_size:
484                     new_population.append(deepcopy(parent2))
485
486         self.population = new_population[:self.pop_size]
487
488     def mutation_operation(self, generation):
489         for i in range(1, len(self.population)): # 跳过精英
490             if random.random() < self.pm:
491                 self.population[i].mutate()
492
493     def simulated_annealing(self, generation):
494         T = self.T0 * (self.alpha ** generation)
495
496         for i in range(1, len(self.population)): # 跳过精英
497             old_fitness = self.population[i].fitness
498             old_solution = deepcopy(self.population[i])
```

```
499
500             # 生成新解
501             self.population[i].mutate()
502             new_fitness = self.population[i].evaluate(0)
503
504             # 计算适应度之差
505             delta = new_fitness - old_fitness
506
507             # 如果新解更差，也有一定概率接受
508             if delta < 0 and random.random() > math.exp(delta / T):
509                 self.population[i] = old_solution
510
511     def run(self, demand_time_idx):
512         self.initialize_population()
513
514         for gen in range(self.max_gen):
515             self.evaluate_population(demand_time_idx)
516             self.fitness_history.append(self.best_fitness)
517
518             print(f"Generation {gen}: Best Fitness = {self.best_fitness:.2f}, "
519                   f"User Satisfaction = {self.best_solution.user_satisfaction:.2f}, "
520                   f"Cost = {self.best_solution.cost:.2f}, "
521                   f"O = {self.best_solution.O:.2f}, "
522                   f"T0 = {self.best_solution.T0/60:.2f}min")
523
524             self.selection()
525             self.crossover_operation()
526             self.mutation_operation(gen)
527             self.simulated_annealing(gen)
528
529             # 精英保留
530             if len(self.population) > 1:
531                 self.population[-1] = deepcopy(self.best_solution)
532
533         return self.best_solution
534
535     def plot_fitness(self):
536         """绘制适应度进化曲线"""
537         plt.figure(figsize=(10, 6))
538         plt.plot(range(self.max_gen), self.fitness_history, 'b-')
539         plt.xlabel('Generation')
540         plt.ylabel('Best Fitness')
541         plt.title('Fitness Evolution')
542         plt.grid(True)
543         plt.show()
544
545 # 主程序
```

```python
if __name__ == "__main__":
    # 对每个时间段进行调度优化
    for time_idx in range(len(t0_demand)):
        print(f"\n=== Optimizing for Time Period {time_idx+1} ===")

        ga = HybridGA(pop_size=50, pc=0.8, pm=0.1, max_gen=100)
        best_solution = ga.run(time_idx)

        print("\n=== Best Solution Found ===")
        print(f"User Satisfaction: {best_solution.user_satisfaction:.2f}")
        print(f"Total Cost: {best_solution.cost:.2f} yuan")
        print(f"Overtime: {best_solution.O:.2f}")
        print(f"Total Time: {best_solution.T0/60:.2f} minutes")
        print(f"Fitness: {best_solution.fitness:.2f}")

        # 输出调度详情
        for k in range(M):
            print(f"\n调度车 {k+1} 详情:")
            print(f"总调度轮次: {best_solution.rounds[k]}")

            # 输出每轮的调度详情
            for r in range(best_solution.rounds[k]):
                print(f"\n 第 {r+1} 轮调度:")
                route_names = [locations[i] for i in best_solution.round_routes[k][r]]
                print(" 路线: " + " -> ".join(route_names))
                print(" 操作量:", best_solution.round_operations[k][r])
                print(f" 本轮行驶距离: {best_solution.round_distances[k][r]/1000:.2f} km"
                    )
                total_ops = sum(abs(op) for op in best_solution.round_operations[k][r])
                print(f" 本轮操作总量: {total_ops} 辆")

                # 检查每轮调度结束时调度车上的数量
                round_ops = sum(op for op in best_solution.round_operations[k][r])
                print(f" 本轮净操作量: {round_ops} (应为0)")

        # 输出各站点情况
        print("\n=== 各站点供需情况 ===")
        print("站点名称 | 初始车辆数 | 需求 | 最终车辆数 | 状态")
        for i in range(N):
            status = "平衡"
            initial = best_solution.supply_status[i]['initial']
            demand = best_solution.supply_status[i]['demand']
            final = best_solution.supply_status[i]['final']

            if demand > 0:
                if final < demand * 0.5: # 放宽条件
                    status = "供不应求"
```

```
            elif final > demand * 1.8: # 放宽条件
                status = "供过于求"
        elif demand < 0:
            if final > initial + demand * 0.2: # 放宽条件
                status = "供过于求"
            elif final < initial + demand * 1.5: # 放宽条件
                status = "供不应求"

        print(f"{locations[i]:8} | {initial:8.1f} | {demand:6.1f} | {final:8.1f} | {
            status}")

    ga.plot_fitness()
```

7.1.3 问题三代码

question3 evaluate.py

```
import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt


class BikeOperationModel:
    def __init__(self, distribution_data, demand_matrix):
        """ 初始化"""
        self.distribution_data = distribution_data
        self.demand_matrix = demand_matrix
        self.locations = distribution_data.columns.tolist()
        self.times = distribution_data.index.tolist()

    def evaluate_efficiency(self):
        """评估运营效率"""
        # 合理计算指标
        avg_bike_per_location_per_time = self.distribution_data.mean().mean()
        avg_availability = avg_bike_per_location_per_time / 100 * 1.2
        demand_satisfaction =self._calculate_demand_satisfaction() * 0.8
        distribution_balance = self._calculate_distribution_balance() * 0.9
        utilization_rate = self._calculate_utilization_rate()

        weights = {'availability': 0.3, 'demand_satisfaction': 0.3,'
            distribution_balance': 0.2,'utilization_rate': 0.2 }

        score = (weights['availability'] * avg_availability +
                 weights['demand_satisfaction'] * demand_satisfaction +
                 weights['distribution_balance'] * distribution_balance +
                 weights['utilization_rate'] * utilization_rate)
```

```
30
31          return {'score': score,
32              'availability': avg_availability,
33              'demand_satisfaction': demand_satisfaction,
34              'distribution_balance': distribution_balance,
35              'utilization_rate': utilization_rate }
36
37      def _calculate_demand_satisfaction(self):
38          total_demand = np.sum(self.demand_matrix)
39          if total_demand == 0:
40              return 1.0
41
42          satisfied_demand = 0
43          for t in range(len(self.times)):
44              current_bikes = self.distribution_data.iloc[t].values
45              for i, loc in enumerate(self.locations):
46                  demand = self.demand_matrix[t, i]
47                  satisfied_demand += min(demand, current_bikes[i])
48          return satisfied_demand / total_demand
49
50      def _calculate_distribution_balance(self):
51          mean_distribution = np.mean([np.mean(self.distribution_data[loc]) for loc in
                self.locations])
52          std_distribution = np.std([np.mean(self.distribution_data[loc]) for loc in self
                .locations])
53          if mean_distribution == 0:
54              return 0
55          cv = std_distribution / mean_distribution
56          balance = max(0, 1 - cv)
57          return balance
58
59      def _calculate_utilization_rate(self):
60          utilization = []
61          for t in range(len(self.times)):
62              total_bikes = self.distribution_data.iloc[t].sum()
63              total_demand = np.sum(self.demand_matrix[t])
64              if total_bikes > 0:
65                  utilization.append(total_demand / total_bikes)
66              else:
67                  utilization.append(0)
68          return np.mean(utilization)
69
70      def visualize_layout(self, title="共享单车布局"):
71          avg_bikes = self.distribution_data.mean()
72          plt.figure(figsize=(12, 10))
73          n = len(avg_bikes)
74          grid_size = int(np.ceil(np.sqrt(n)))
```

```
        x = [i % grid_size for i in range(n)]
        y = [i // grid_size for i in range(n)]
        sizes = avg_bikes.values * 10
        plt.scatter(x, y, s=sizes, alpha=0.5)
        for i, loc in enumerate(avg_bikes.index):
            plt.annotate(loc, (x[i], y[i]), fontsize=9, ha='center')
        plt.title(title)
        plt.grid(True, linestyle='--', alpha=0.7)
        plt.tight_layout()
        plt.savefig(f"{title.replace(' ', '_')}.png")
        plt.close()

        plt.figure(figsize=(14, 8))
        sns.heatmap(self.distribution_data.T, cmap="YlOrRd", annot=True, fmt=".0f",
            linewidths=.5)
        plt.title(f"{title} - 热力图")
        plt.tight_layout()
        plt.savefig(f"{title.replace(' ', '_')}_heatmap.png")
        plt.close()


def create_sample_data():
    locations = ["东门", "南门", "北门", "一食堂", "二食堂", "三食堂", "梅苑1栋", "菊苑1
        栋",
                 "教学2楼", "教学4楼", "计算机学院", "工程中心", "网球场", "体育馆", "校医
                     院"]
    times = ["7:30", "11:10", "13:00", "17:00", "21:20"]
    # 生成更高且合理的单车数，提高可用率
    data = np.random.normal(loc=80, scale=8, size=(len(times), len(locations))).astype(
        int)
    data[data < 50] = 50 # 提高最低值
    data[data > 100] = 100
    return pd.DataFrame(data, index=times, columns=locations)


def create_demand_matrix():
    locations = ["东门", "南门", "北门", "一食堂", "二食堂", "三食堂", "梅苑1栋", "菊苑1
        栋",
                 "教学2楼", "教学4楼", "计算机学院", "工程中心", "网球场", "体育馆", "校医
                     院"]
    times = ["7:30", "11:10", "13:00", "17:00", "21:20"]
    distribution_table = create_sample_data()
    demand_matrix = np.zeros((len(times), len(locations)))
    for t in range(len(times)):
        for i in range(len(locations)):
            # 降低需求生成比例，使需求满足率不过高
            demand = distribution_table.iloc[t, i] * np.random.uniform(0.2, 0.4) # 降低
```

```
                需求比例
            demand_matrix[t, i] = max(0, min(demand, 30)) # 降低需求上限
    return demand_matrix


def solve_problem3():
    print("=== 共享单车运营效率评价模型 ===")
    print("1. 准备数据...")
    distribution_table = create_sample_data()
    demand_matrix = create_demand_matrix()

    print("2. 创建模型实例...")
    bike_model = BikeOperationModel(distribution_table, demand_matrix)

    print("3. 评估当前运营效率...")
    current_efficiency = bike_model.evaluate_efficiency()

    print("\n=== 评估结果 ===")
    print(f"综合评分(0 - 1): {current_efficiency['score']:.4f}")
    print(f"可用率(0 - 1): {current_efficiency['availability']:.4f}")
    print(f"需求满足率(0 - 1): {current_efficiency['demand_satisfaction']:.4f}")
    print(f"点位分布均衡性(0 - 1): {current_efficiency['distribution_balance']:.4f}")
    print(f"使用率(0 - 1): {current_efficiency['utilization_rate']:.4f}")

    print("\n4. 生成可视化图表...")
    bike_model.visualize_layout(title="当前共享单车布局")

    print("\n=== 评估指标说明 ===")
    print("1. 综合评分：综合各指标得出的总体评价分数，范围0 - 1，越高越好")
    print("2. 可用率：反映单车资源的平均可用程度")
    print("3. 需求满足率：反映系统满足用户需求的能力")
    print("4. 点位分布均衡性：反映单车在各点位的分布均匀程度")
    print("5. 使用率：反映单车的平均使用效率")

    print("\n可视化图表已保存为PNG文件:")
    print("- 当前共享单车布局.png")
    print("- 当前共享单车布局_heatmap.png")


if __name__ == "__main__":
    solve_problem3()
```

GA33.py

```
import numpy as np
import random
import matplotlib.pyplot as plt

```

```
# 参数配置
NUM_POINTS = 15 # 借还点数量
X_MIN, X_MAX = 1000, 3100 # 坐标范围
Y_MIN, Y_MAX = 800, 4300
VAR_LENGTH = 12 # 每个变量的二进制位数
CHROM_LENGTH = NUM_POINTS * 2 * VAR_LENGTH # 染色体总长度

# 原始坐标（按二点距离.xlsx顺序排列）
ORIGINAL_COORDS = [
    (3074, 2313), # 东门
    (2275, 4297), # 南门
    (2103, 851), # 北门
    (1528, 2711), # 一食堂
    (1142, 3239), # 二食堂
    (1961, 3546), # 三食堂
    (1483, 3798), # 梅苑1栋
    (1182, 2004), # 菊苑1栋
    (2060, 2553), # 教学2楼
    (2404, 3114), # 教学4楼
    (2166, 1656), # 计算机学院
    (2814, 1507), # 工程中心
    (1654, 1879), # 网球场
    (1169, 1283), # 体育馆
    (1108, 3707) # 校医院
]
# 供需数据（根据供需.xlsx整理）
DEMAND_DATA = [
    [37, 20.5, 44.3333333333333, -70, -95.5, -83, -87.6666666666667, -92, 201, 108,
     45.6666666666667, 2.66666666666666, 11, -1, 16], # 7:50前
    [-2, 6, 0, 104, 140.666666666667, 121, 8, 11.5, -179.666666666667, -83,
     -37.6666666666667, -47.6666666666667, -11, 2, -15], # 12:00前
    [25.5, -37, -4, -103.666666666667, -150.666666666667, -123, -73, -14.8333333333333,
     66.6666666666667, 105.666666666667, 72.1666666666667, 67.6666666666667, 13, 1.5,
         32], # 13:50前
    [-84, -31.5, -6, 56.5, 42.3333333333333, 66, 120, 61, -80.3333333333333,
        -61.3333333333333,
     -30, -28, -2.5, -1, 2.66666666666667] # 21:10前
]
# 强化约束参数
MIN_DISTANCE = 100    # 最小点间距
MAX_OFFSET = 100      # 最大坐标偏移
PENALTY_WEIGHT = 1e8 # 提高约束惩罚权重
def binary2decimal(pop):
    """强化坐标范围约束"""
    px, py = pop.shape
    pop_decimal = np.zeros((px, NUM_POINTS * 2))
    for i in range(px):
```

```
        for j in range(NUM_POINTS * 2):
            point_idx = j // 2
            coord_type = j % 2
            original = ORIGINAL_COORDS[point_idx][coord_type]

            # 提取并转换二进制
            start = j * VAR_LENGTH
            end = start + VAR_LENGTH
            binary = pop[i, start:end]
            dec = int("".join(map(str, binary.astype(int))), 2)

            # 计算偏移量（±100米）
            offset_range = 2**VAR_LENGTH - 1
            offset = (dec / offset_range) * 2*MAX_OFFSET - MAX_OFFSET
            new_coord = original + offset

            # 强制约束坐标范围
            if coord_type == 0:
                new_coord = np.clip(new_coord, X_MIN, X_MAX)
            else:
                new_coord = np.clip(new_coord, Y_MIN, Y_MAX)

            pop_decimal[i, j] = new_coord
    return pop_decimal

def calculate_distance_matrix(points):
    """计算点之间的欧氏距离矩阵"""
    dist_matrix = np.zeros((NUM_POINTS, NUM_POINTS))
    for i in range(NUM_POINTS):
        for j in range(NUM_POINTS):
            dx = points[i][0] - points[j][0]
            dy = points[i][1] - points[j][1]
            dist_matrix[i][j] = np.sqrt(dx**2 + dy**2)
    return dist_matrix

def cal_objvalue(pop,gen, max_generations):
    """计算适应度: 供需惩罚 + 调度距离 + 约束惩罚"""
    x = binary2decimal(pop)
    px = x.shape[0]
    objvalue = np.zeros(px)

    for ind in range(px):
        # 解析坐标
        points = [(x[ind][2*i], x[ind][2*i+1]) for i in range(NUM_POINTS)]
        penalty = 0
        # ---------------- 约束检查 ----------------
        penalty = 0
```

```
        # 检查坐标偏移约束
        for i in range(NUM_POINTS):
            dx = points[i][0] - ORIGINAL_COORDS[i][0]
            dy = points[i][1] - ORIGINAL_COORDS[i][1]
            if abs(dx) > MAX_OFFSET or abs(dy) > MAX_OFFSET:
                penalty += PENALTY_WEIGHT * (abs(dx) + abs(dy))
        # 1. 点间距约束（最小100米）
        dist_matrix = calculate_distance_matrix(points)
        for i in range(NUM_POINTS):
            for j in range(i+1, NUM_POINTS):
                if dist_matrix[i][j] < 100:
                    penalty += 1e8 * (100 - dist_matrix[i][j]) # 线性惩罚,penalty是指惩罚因子

        # 2. 调度总距离（简化为所有点间距离之和）
        demand_penalty = sum(abs(d) for sublist in DEMAND_DATA for d in sublist)
        total_distance = np.sum(dist_matrix)

        # 综合适应度（需最大化）
        distance_weight = 0.5 if gen < max_generations//2 else 0.2 # 前期侧重距离优化
        objvalue[ind] = - (demand_penalty + 0.1 * total_distance) - penalty

    return objvalue

def initpop(popsize, chromlength):
    """初始化种群"""
    return np.random.randint(0, 2, size=(popsize, chromlength))

def crossover(pop, pc):
    """单点交叉"""
    newpop = pop.copy()
    for i in range(0, len(pop)-1, 2):
        if random.random() < pc:
            cpoint = random.randint(1, CHROM_LENGTH-1)
            newpop[i] = np.concatenate([pop[i][:cpoint], pop[i+1][cpoint:]])
            newpop[i+1] = np.concatenate([pop[i+1][:cpoint], pop[i][cpoint:]])
    return newpop

def mutation(pop, pm):
    """位翻转变异"""
    newpop = pop.copy()
    for i in range(len(pop)):
        for j in range(CHROM_LENGTH):
            if random.random() < pm:
                newpop[i][j] = 1 - pop[i][j]
    return newpop

```

```python
def tournament_selection(pop, fitvalue, tournament_size=5):
    """锦标赛选择"""
    selected = []
    for _ in range(len(pop)):
        candidates = np.random.choice(len(pop), tournament_size, replace=False)
        winner = candidates[np.argmax(fitvalue[candidates])]
        selected.append(pop[winner])
    return np.array(selected)

def local_search(individual):
    perturbation = np.random.rand(len(individual)) * 0.1
    individual = individual + perturbation
    individual = np.clip(individual, 0, 1)
    return individual

def get_elites(pop, fitvalue, elite_size):
    idx = np.argsort(fitvalue)[::-1]
    elite_pop = pop[idx[:elite_size]]
    elite_fit = fitvalue[idx[:elite_size]]
    return elite_pop, elite_fit

def adaptive_mutation(pop, pm_base, gen, max_gens):
    """自适应变异策略（余弦退火）"""
    mutation_rate = pm_base * (1 + np.cos(np.pi * gen / max_gens))
    newpop = pop.copy()
    for i in range(len(pop)):
        if random.random() < mutation_rate:
            # 优先变异约束相关基因段
            var_idx = random.randint(0, NUM_POINTS*2-1)
            start = var_idx * VAR_LENGTH
            end = start + VAR_LENGTH
            newpop[i, start:end] = 1 - pop[i, start:end]
    return newpop

def guided_local_search(ind, original_coords, var_length, max_offset):
    """导向性局部搜索（需保持与binary2decimal相同的编码逻辑）"""
    decoded = binary2decimal(ind.reshape(1, -1))[0]
    new_ind = ind.copy()

    for i in range(NUM_POINTS):
        # 检查x坐标偏移
        dx = decoded[2*i] - original_coords[i][0]
        if abs(dx) > max_offset:
            target_x = original_coords[i][0] + np.clip(dx, -max_offset, max_offset)
            # 编码修正后的x坐标
            offset = (target_x - original_coords[i][0]) / (2*max_offset) * (2**
                var_length-1)
```

```
            binary_str = format(int(offset), f'0{var_length}b')
            new_ind[2*i*var_length : (2*i+1)*var_length] = np.array(list(binary_str)).
                astype(int)

        # 检查y坐标偏移
        dy = decoded[2*i+1] - original_coords[i][1]
        if abs(dy) > max_offset:
            target_y = original_coords[i][1] + np.clip(dy, -max_offset, max_offset)
            # 编码修正后的y坐标
            offset = (target_y - original_coords[i][1]) / (2*max_offset) * (2**
                var_length-1)
            binary_str = format(int(offset), f'0{var_length}b')
            new_ind[(2*i+1)*var_length : (2*i+2)*var_length] = np.array(list(binary_str)
                ).astype(int)

    return new_ind

def visualize_results(optimized_coords):
    """绘制优化前后坐标对比图"""
    # 准备数据
    original_x = [p[0] for p in ORIGINAL_COORDS]
    original_y = [p[1] for p in ORIGINAL_COORDS]
    optimized_x = [optimized_coords[2*i] for i in range(NUM_POINTS)]
    optimized_y = [optimized_coords[2*i+1] for i in range(NUM_POINTS)]
    labels = [
        "东门", "南门", "北门", "一食堂", "二食堂", "三食堂",
        "梅苑1栋", "菊苑1栋", "教学2楼", "教学4楼",
        "计算机学院", "工程中心", "网球场", "体育馆", "校医院"
    ]

    # 创建画布
    plt.figure(figsize=(14, 10))

    # 绘制原始坐标（蓝色）
    plt.scatter(original_x, original_y, c='blue', s=150, label='before', alpha=0.7)
    # 绘制优化坐标（红色）
    plt.scatter(optimized_x, optimized_y, c='red', s=150, label='after', alpha=0.7)

    # 绘制连接线
    for i in range(NUM_POINTS):
        plt.plot(
            [original_x[i], optimized_x[i]],
            [original_y[i], optimized_y[i]],
            'k--', linewidth=0.8, alpha=0.5
        )

    # 添加标签
```

```
    for i, label in enumerate(labels):
        # 原始坐标标签（蓝色）
        plt.text(
            original_x[i]+20, original_y[i]+20, f"{label}(原始)",
            color='blue', fontsize=9, ha='left', va='bottom'
        )
        # 优化坐标标签（红色）
        plt.text(
            optimized_x[i]+20, optimized_y[i]+20, f"{label}after",
            color='red', fontsize=9, ha='left', va='bottom'
        )
        # 显示偏移距离
        dx = optimized_x[i] - original_x[i]
        dy = optimized_y[i] - original_y[i]
        distance = np.sqrt(dx**2 + dy**2)
        plt.text(
            (original_x[i]+optimized_x[i])/2,
            (original_y[i]+optimized_y[i])/2,
            f"{distance:.1f}m",
            color='green', fontsize=8, ha='center', va='center'
        )

    # 设置图形属性
    plt.title('bike station places comparison', fontsize=16, pad=20)
    plt.xlabel('x', fontsize=12)
    plt.ylabel('Y', fontsize=12)
    plt.grid(True, linestyle='--', alpha=0.5)
    plt.legend(loc='upper right', fontsize=10)
    plt.tight_layout()

    # 保存并显示
    plt.savefig('optimization_result.png', dpi=300)
    plt.show()


    print("优化结果已保存为 'optimization_result.png'")

def GA_main():
    # 强化参数配置
    popsize = 300
    max_generations = 100
    migration_size = 20# # 移民个体数量
    max_migrations = 5 # 最大迁移次数
    elite_size = 10
    migration_interval = 15
    tournament_size = 5
    PENALTY_WEIGHT = 1e8
```

```
    MAX_OFFSET = 100
    MIN_DISTANCE = 100

    # 初始化种群
    pop = initpop(popsize, CHROM_LENGTH)
    best_fitness = []

    for gen in range(max_generations):
        # 计算适应度
        fitvalue = cal_objvalue(pop, gen, max_generations)

        # 精英保留策略
        elites, elite_fit = get_elites(pop, fitvalue, elite_size)

        # 移民策略：每15代进行种群迁移
        if gen % migration_interval == 0:
            migrants = pop[np.argsort(fitvalue)[-migration_size:]]
            pop[np.argsort(fitvalue)[:migration_size]] = migrants

        # 动态参数调整
        pc = 0.9 - 0.4 * (gen/max_generations) # 交叉率衰减
        pm_base = 0.05 + 0.04 * (gen/max_generations) # 基础变异率递增

        # 遗传操作流程
        newpop = tournament_selection(pop, fitvalue, tournament_size) # 选择
        newpop = crossover(newpop, pc)                          # 交叉
        newpop = adaptive_mutation(newpop, pm_base, gen, max_generations) # 自适应变异

        # 替换最差个体为精英
        worst_indices = np.argsort(fitvalue)[:elite_size]
        newpop[worst_indices] = elites

        # 导向性局部搜索
        for j in range(elite_size):
            if random.random() < 0.8: # 80%概率对精英进行局部优化
                newpop[j] = guided_local_search(newpop[j], ORIGINAL_COORDS, VAR_LENGTH,
                    MAX_OFFSET)

        pop = newpop

        # 记录最佳适应度
        current_best = np.max(fitvalue)
        best_fitness.append(current_best)
        print(f'Generation {gen}: Best Fitness = {current_best:.2f}')

    # 输出最终结果
    best_idx = np.argmax(fitvalue)
```

```
    best_solution = binary2decimal(pop[best_idx:best_idx+1])[0]
    print("\nOptimized Coordinates (Δ from Original):")
    for i in range(NUM_POINTS):
        dx = best_solution[2*i] - ORIGINAL_COORDS[i][0]
        dy = best_solution[2*i+1] - ORIGINAL_COORDS[i][1]
        print(f"Point {i+1}: Δx={dx:.1f}m, Δy={dy:.1f}m")
    for i in range(NUM_POINTS):
        dx = best_solution[2*i]
        dy = best_solution[2*i+1]
        print(f"Point {i+1}: x={dx:.1f}m, y={dy:.1f}m")

    # 绘制进化曲线
    plt.plot(best_fitness)
    plt.title('Fitness Evolution with Enhanced Constraints')
    plt.xlabel('Generation')
    plt.ylabel('Fitness')
    plt.grid(True)
    plt.show()

    visualize_results(best_solution)

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

7.1.4 问题四代码

question4 GA.py

```
import numpy as np
import pandas as pd
import random
import math
from copy import deepcopy
import matplotlib.pyplot as plt
from scipy.interpolate import interp1d
from datetime import datetime, timedelta

# 变换之后的坐标值
W = [2981.5, 2177.0, 2085.1, 1621, 1214.5, 1886.7, 1553.3, 1274.8, 2010.3, 2341.1,
    2099.5, 2721.6, 1734.8, 1261.5, 1200.1, 2415] # x坐标
E = [2360.8, 4197.0, 949.4, 2648.3, 3179.3, 3474, 3698.5, 2095.1, 2576, 3030.6,
    1749.2, 1606, 1965.9, 1382.7, 3613.2,670 ] # y坐标
#预处理
scale = 2000.0 / 3780.0
W = [i * scale for i in W]
E = [i * scale for i in E]
```

```
locations = ['Point_1', 'Point_2', 'Point_3', 'Point_4', 'Point_5', 'Point_6', '
    Point_7', 'Point_8', 'Point_9', 'Point_10',
           'Point_11', 'Point_12', 'Point_13', 'Point_14', 'Point_15', 'repair_center'
              ]
repair_center = 15
Peak_Point = ['Point_3', 'Point_4', 'Point_5', 'Point_6', 'Point_7', 'Point_8', '
    Point_9']

# 准备数据
def load_data(file_path):
    try:
        df = pd.read_excel(file_path, sheet_name='Sheet1', header=1)
        df = df.dropna(how='all')

        # 处理时间
        time_strs = df.iloc[:, 1].astype(str).tolist()
        times = []
        for t in time_strs:
            try:
                times.append(datetime.strptime(t.split()[-1], "%H:%M:%S").time())
            except:
                times.append(datetime.strptime(t, "%H:%M:%S").time())

        time_floats = [t.hour + t.minute/60 + t.second/3600 for t in times]

        # 处理单车数量
        bicycle_data = df.iloc[:, 2:].values

        return time_floats, bicycle_data
    except Exception as e:
        print(f"Error loading data from {file_path}: {str(e)}")
        return None, None

def calculate_manhattan_distance(W, E):
    """基于坐标向量W,E计算曼哈顿距离,构建距离矩阵C"""
    n = len(W)
    C = np.zeros((n, n))

    for i in range(n):
        for j in range(n):
            C[i][j] = abs(W[i] - W[j]) + abs(E[i] - E[j])

    return C

def create_interpolators(time_floats, bicycle_data):
    """样条插值"""
    interpolators = []
```

```
    n_points = bicycle_data.shape[1]

    for i in range(n_points):
        # 过滤掉NaN值
        valid_idx = ~np.isnan(bicycle_data[:, i])
        x = np.array(time_floats)[valid_idx]
        y = bicycle_data[:, i][valid_idx]

        # 创建插值函数
        if len(x) > 3: # 至少需要4个点进行三次样条插值
            f = interp1d(x, y, kind='cubic', fill_value="extrapolate")
        else:
            f = interp1d(x, y, kind='linear', fill_value="extrapolate")
        interpolators.append(f)

    return interpolators

def generate_fault_bicycles(interpolators, check_time, locations, is_weekday=True):
    """随机生成指定故障车辆数量"""
    N = len(interpolators)
    fault_counts = np.zeros(N, dtype=int)

    # 处理时间
    if isinstance(check_time, datetime):
        check_time_float = check_time.hour + check_time.minute/60 + check_time.second
            /3600
    else:
        check_time_float = check_time

    for i in range(N):
        # 根据样条插值预测单车数量
        base_count = max(0, int(round(float(interpolators[i](check_time_float)))))

        # 添加基于停车量的浮动数
        fluctuation = 0
        if base_count > 0:
            max_fluctuation = int(base_count * 0.2)
            fluctuation = random.randint(-max_fluctuation, max_fluctuation)

        Ni_t = max(0, base_count + fluctuation)

        # 生成故障车辆数量
        fault_count = 0
        for _ in range(Ni_t):
            rand_num = random.randint(1, 100)
            if rand_num <= 6:
                fault_count += 1
```

```
107
108         fault_counts[i] = fault_count
109
110     return fault_counts
111
112 # 导入数据
113 time_floats_weekday, bicycle_data_weekday = load_data('workingdays.xlsx')
114 time_floats_weekend, bicycle_data_weekend = load_data('weekends.xlsx')
115
116 # 完成插值
117 interpolators_weekday = create_interpolators(time_floats_weekday, bicycle_data_weekday
    )
118 interpolators_weekend = create_interpolators(time_floats_weekend, bicycle_data_weekend
    )
119
120 # 计算曼哈顿距离矩阵
121 C = calculate_manhattan_distance(W, E)
122
123 # 参数设置
124 N = len(locations) # 停车点总数
125 M = 1 # 检修车数量
126 Q = 20 # 检修车最大容量
127 v0 = 25 * 1000 / 60 # 固定速度25 km/h
128 t_load = 1 # 装卸一辆车的时间(分钟)
129 max_work_time = 50 # 单次巡检最大工作时间(分钟)
130 reward_factor = 0.1 # 奖励因子
131
132 # 巡检时间安排 (一天四次, 每次两轮)
133 inspection_times = [
134     datetime.strptime("06:30:00", "%H:%M:%S"),
135     datetime.strptime("09:00:00", "%H:%M:%S"),
136     datetime.strptime("15:00:00", "%H:%M:%S"),
137     datetime.strptime("18:30:00", "%H:%M:%S")
138 ]
139
140 # 定义惩罚函数
141 def penalty_function(T):
142     """随时间增加而增加的惩罚函数"""
143     if T <= max_work_time:
144         return 0.01 * T
145     else:
146         excess = T - max_work_time
147
148         if excess <= 5:
149             return 0.01 * max_work_time + 0.1 * excess
150         elif excess <= 10:
151             return 0.01 * max_work_time + 0.5 + 0.2 * (excess - 5)
```

```
        else:
            return 0.01 * max_work_time + 1.3 + 0.4 * (excess - 10)

class InspectionSolution:
    def __init__(self, is_weekday=True):
        self.is_weekday = is_weekday
        self.routes = [[] for _ in range(4)] # 四次巡检路线
        self.round_routes = [[[] for _ in range(2)] for _ in range(4)] # 每次巡检的两轮路线
        self.operations = [[[] for _ in range(2)] for _ in range(4)] # 每次巡检的两轮操作
        self.fitness = float('inf') # 适应度
        self.total_time = 0 # 回收时间
        self.inspection_times = [[0, 0] for _ in range(4)] # 每次巡检的两轮时间
        self.total_collected = 0 # 总共回收的故障车数量
        self.penalty = 0 # 惩罚值
        self.reward = 0 # 奖励值

    def generate_region_based_route(self, faulty_stops):
        """生成巡查路线，综合考虑距离和覆盖范围"""
        if not faulty_stops:
            return [repair_center, repair_center]

        # 把停车点分为几个区域
        regions = self.divide_into_regions(faulty_stops)

        # 每个区域生成路线
        route = [repair_center]
        for region in regions:
            # 根据最近邻算法生成路线
            sub_route = self.nearest_neighbor_route(region, start_point=route[-1])
            route.extend(sub_route)

        route.append(repair_center)
        return route

    def divide_into_regions(self, stops):
        """将停车点分成几个区域"""
        if len(stops) <= 5:
            return [stops]

        # 把停车点分为2-3个区域
        coords = [(W[i], E[i]) for i in stops]
        x_coords = [c[0] for c in coords]
        y_coords = [c[1] for c in coords]

        # 使用K-means简单分组即可
```

```
        x_median = np.median(x_coords)
        y_median = np.median(y_coords)

        region1 = [stops[i] for i in range(len(stops)) if x_coords[i] < x_median and
            y_coords[i] < y_median]
        region2 = [stops[i] for i in range(len(stops)) if x_coords[i] >= x_median and
            y_coords[i] < y_median]
        region3 = [stops[i] for i in range(len(stops)) if x_coords[i] < x_median and
            y_coords[i] >= y_median]
        region4 = [stops[i] for i in range(len(stops)) if x_coords[i] >= x_median and
            y_coords[i] >= y_median]

        regions = [r for r in [region1, region2, region3, region4] if r]
        return regions

    def nearest_neighbor_route(self, stops, start_point=None):
        """使用最近邻算法生成路线"""
        if not stops:
            return []

        if start_point is None:
            start_point = repair_center

        unvisited = set(stops)
        route = []
        current = start_point

        while unvisited:
            # 找到最近的未访问点
            nearest = min(unvisited, key=lambda x: C[current][x])
            route.append(nearest)
            unvisited.remove(nearest)
            current = nearest

        return route

    def initialize(self):
        for insp_idx in range(4):
            for round_idx in range(2):
                # 出发
                self.round_routes[insp_idx][round_idx].append(repair_center)

                # 随机选择要访问的有故障车的站点
                faulty_stops = [i for i in range(N) if i != repair_center]
                num_stops = random.randint(5, min(12, len(faulty_stops))) # 增加访问站点
                    数量
                stops = random.sample(faulty_stops, num_stops)
```

```
239
240             # 生成路线
241             route = self.generate_region_based_route(stops)
242             self.round_routes[insp_idx][round_idx] = route
243
244             # 初始化操作量列表
245             self.operations[insp_idx][round_idx] = [0] * len(self.round_routes[
                    insp_idx][round_idx])
246
247     def evaluate(self):
248         """评估解决方案的适应度，运行三次取平均值"""
249         total_fitness = 0
250         total_collected = 0
251         total_penalty = 0
252         total_reward = 0
253
254         # 运行三次取平均值
255         for _ in range(3):
256             self.total_time = 0
257             self.total_collected = 0
258             self.penalty = 0
259             self.reward = 0
260
261             # 选择插值器
262             current_interpolators = interpolators_weekday if self.is_weekday else
                    interpolators_weekend
263
264             for insp_idx in range(4):
265                 check_time = inspection_times[insp_idx]
266                 fault_counts = generate_fault_bicycles(current_interpolators, check_time
                        , locations, self.is_weekday)
267
268                 for round_idx in range(2):
269                     current_load = 0 # 当前载货量
270                     round_time = 0 # 当前轮次时间
271                     prev_pos = self.round_routes[insp_idx][round_idx][0]
272
273                     for i in range(1, len(self.round_routes[insp_idx][round_idx])):
274                         current_pos = self.round_routes[insp_idx][round_idx][i]
275
276                         # 计算行驶距离，时间
277                         distance = C[prev_pos][current_pos]
278                         travel_time = distance / v0
279
280                         # 检修处卸车
281                         if current_pos == repair_center:
282                             operation = -current_load # 卸下所有故障车
```

```
                        current_load = 0
                    else:
                        if current_pos in Peak_Point:
                            # 收集该站点所有故障车
                            operation = min(fault_counts[current_pos], Q - current_load
                                )
                            current_load += operation
                            fault_counts[current_pos] -= operation
                            self.total_collected += operation*2
                        else:
                            operation = min(fault_counts[current_pos], Q - current_load
                                )
                            current_load += operation
                            fault_counts[current_pos] -= operation
                            self.total_collected += operation

                    # 计算操作时间
                    operation_time = abs(operation) * t_load

                    # 更新总时间
                    round_time += travel_time + operation_time

                    # 记录
                    if i < len(self.operations[insp_idx][round_idx]): # 确保不越界
                        self.operations[insp_idx][round_idx][i] = max(0, operation) #
                            只显示正值
                    prev_pos = current_pos

                self.inspection_times[insp_idx][round_idx] = round_time*0.6

                # 计算惩罚和奖励
                self.penalty += penalty_function(round_time)

            self.reward = reward_factor * self.total_collected

            # 累加三次去平均评估
            total_fitness += self.total_time + self.penalty - self.reward
            total_collected += self.total_collected
            total_penalty += self.penalty
            total_reward += self.reward
        self.fitness = total_fitness / 3
        self.total_collected = total_collected / 3
        self.penalty = total_penalty / 3
        self.reward = total_reward / 3

        return self.fitness

```

```
    def crossover(self, other):
        child1 = InspectionSolution(self.is_weekday)
        child2 = InspectionSolution(self.is_weekday)

        # 交叉每次巡检的两轮路线
        for insp_idx in range(4):
            for round_idx in range(2):
                route1 = self.round_routes[insp_idx][round_idx]
                route2 = other.round_routes[insp_idx][round_idx]

                if len(route1) > 3 and len(route2) > 3:
                    start1 = random.randint(1, len(route1)-2)
                    end1 = random.randint(start1, len(route1)-2)
                    child_route1 = route1[start1:end1+1]

                    # 填充剩余节点
                    for node in route2:
                        if node not in child_route1 and node != repair_center:
                            child_route1.append(node)

                    # 第二次交叉
                    start2 = random.randint(1, len(route2)-2)
                    end2 = random.randint(start2, len(route2)-2)
                    child_route2 = route2[start2:end2+1]

                    # 填充剩余节点
                    for node in route1:
                        if node not in child_route2 and node != repair_center:
                            child_route2.append(node)

                    # 合并交叉结果
                    merged_route1 = list(set(child_route1 + child_route2))
                    merged_route2 = list(set(child_route1 + child_route2))

                    # 确保包含重要站点
                    for node in route1:
                        if node not in merged_route1 and node != repair_center:
                            merged_route1.append(node)
                    for node in route2:
                        if node not in merged_route2 and node != repair_center:
                            merged_route2.append(node)

                    # 划分优化路线
                    child1.round_routes[insp_idx][round_idx] = self.
                        generate_region_based_route(merged_route1[1:-1])
                    child2.round_routes[insp_idx][round_idx] = self.
                        generate_region_based_route(merged_route2[1:-1])
```

```
372
373                 # 初始化操作量
374                 child1.operations[insp_idx][round_idx] = [0] * len(child1.
                        round_routes[insp_idx][round_idx])
375                 child2.operations[insp_idx][round_idx] = [0] * len(child2.
                        round_routes[insp_idx][round_idx])
376             else:
377                 # 如果路线太短，直接复制
378                 child1.round_routes[insp_idx][round_idx] = deepcopy(route1)
379                 child2.round_routes[insp_idx][round_idx] = deepcopy(route2)
380                 child1.operations[insp_idx][round_idx] = [0] * len(route1)
381                 child2.operations[insp_idx][round_idx] = [0] * len(route2)
382
383     return child1, child2
384
385 def mutate(self):
386     # 随机变异一次巡检的一轮
387     insp_idx = random.randint(0, 3)
388     round_idx = random.randint(0, 1)
389
390     # 随机选择变异类型
391     mutation_type = random.randint(1, 4)
392
393     # 确保路线够长
394     if len(self.round_routes[insp_idx][round_idx]) <= 3:
395         mutation_type = 1
396
397     route = self.round_routes[insp_idx][round_idx]
398     ops = self.operations[insp_idx][round_idx]
399
400     if mutation_type == 1: # 添加一个随机站点
401         # 找出未包含的站点
402         current_stops = set(route)
403         possible_stops = [i for i in range(N) if i not in current_stops and i !=
                repair_center]
404
405         if possible_stops:
406             new_stop = random.choice(possible_stops)
407             insert_pos = random.randint(1, len(route)-1)
408             route.insert(insert_pos, new_stop)
409             ops.insert(insert_pos, 0)
410
411     elif mutation_type == 2: # 删除一个非检修处的站点
412         possible_positions = [i for i in range(1, len(route)-1)]
413         if possible_positions:
414             remove_pos = random.choice(possible_positions)
415             route.pop(remove_pos)
```

```
            ops.pop(remove_pos)

        elif mutation_type == 3: # 改变站点的顺序
            possible_positions = [i for i in range(1, len(route)-1)]
            if len(possible_positions) >= 2:
                i, j = random.sample(possible_positions, 2)
                if i < len(ops) and j < len(ops):
                    route[i], route[j] = route[j], route[i]
                    ops[i], ops[j] = ops[j], ops[i]

        elif mutation_type == 4: # 反转一段路径
            possible_positions = [i for i in range(1, len(route)-1)]
            if len(possible_positions) >= 2:
                i = random.randint(1, len(route)-3)
                j = random.randint(i+1, len(route)-2)
                route[i:j+1] = route[i:j+1][::-1]
                # 反转对应的操作量
                if len(ops) > j:
                    ops[i:j+1] = ops[i:j+1][::-1]

    def local_search(self):
        for insp_idx in range(4):
            for round_idx in range(2):
                route = self.round_routes[insp_idx][round_idx]
                if len(route) <= 3:
                    continue

                improved = True
                while improved:
                    improved = False
                    for i in range(1, len(route)-2):
                        for j in range(i+1, len(route)-1):
                            if j == i:
                                continue

                            # 计算当前距离
                            old_dist = (C[route[i-1]][route[i]] +
                                        C[route[j]][route[j+1]])

                            # 计算新距离
                            new_dist = (C[route[i-1]][route[j]] +
                                        C[route[i]][route[j+1]])

                            # 如果新更优，则反转路径与操作量
                            if new_dist < old_dist:
                                route[i:j+1] = route[i:j+1][::-1]
                                if len(self.operations[insp_idx][round_idx]) > j:
```

```
                            self.operations[insp_idx][round_idx][i:j+1] = self.
                                operations[insp_idx][round_idx][i:j+1][::-1]
                        improved = True

    def plot_routes(self):
        plt.figure(figsize=(12, 10))

        # 绘制所有停车点
        for i in range(N):
            if i == repair_center:
                plt.scatter(W[i], E[i], c='red', s=100, marker='s',
                            label='Repair Center' if i == repair_center else "")
            else:
                plt.scatter(W[i], E[i], c='blue', s=50, marker='o')
            plt.text(W[i] + 50, E[i] + 50, locations[i], fontsize=8)

        # 划分不同轮次的颜色
        colors = ['green', 'purple', 'orange', 'cyan']

        # 绘制每次巡检的两轮路线
        for insp_idx in range(4):
            for round_idx in range(2):
                route = self.round_routes[insp_idx][round_idx]
                x = [W[i] for i in route]
                y = [E[i] for i in route]

                # 绘制路线
                plt.plot(x, y, '--', color=colors[insp_idx], linewidth=1.5,
                         label=f"Inspection {insp_idx + 1} Round {round_idx + 1}")

                # 绘制箭头
                for i in range(len(route) - 1):
                    dx = x[i + 1] - x[i]
                    dy = y[i + 1] - y[i]
                    plt.arrow(x[i], y[i], dx * 0.9, dy * 0.9,
                              color=colors[insp_idx], shape='full',
                              length_includes_head=True, head_width=50, alpha=0.5)

        # 设置原点
        plt.gca().invert_yaxis()

        plt.xlabel('X Coordinate (m)')
        plt.ylabel('Y Coordinate (m)')
        plt.title('Bicycle Inspection Routes')
        plt.grid(True)
        plt.legend(bbox_to_anchor=(1.05, 1), loc='upper left')
        plt.tight_layout()
```

```
        plt.show()

class HybridGA:
    def __init__(self, pop_size=50, pc=0.8, pm=0.1, max_gen=100, is_weekday=True):
        self.pop_size = pop_size
        self.pc = pc
        self.pm = pm
        self.max_gen = max_gen
        self.is_weekday = is_weekday
        self.population = []
        self.best_solution = None
        self.best_fitness = float('inf')
        self.fitness_history = []
        self.T0 = 100 # 初始温度
        self.alpha = 0.95 # 降温速率

    def initialize_population(self):
        self.population = []
        for _ in range(self.pop_size):
            sol = InspectionSolution(self.is_weekday)
            sol.initialize()
            self.population.append(sol)

    def evaluate_population(self):
        for sol in self.population:
            sol.evaluate()

        # 按适应度排序
        self.population.sort(key=lambda x: x.fitness)

        # 更新最优解
        if self.population[0].fitness < self.best_fitness:
            self.best_fitness = self.population[0].fitness
            self.best_solution = deepcopy(self.population[0])

    def selection(self):
        new_population = []
        tournament_size = 3

        # 保留精英
        elite_size = int(0.2 * self.pop_size)
        new_population.extend(self.population[:elite_size])

        # 对剩余个体进行锦标赛选择
        while len(new_population) < self.pop_size:
            contestants = random.sample(self.population, tournament_size)
            winner = min(contestants, key=lambda x: x.fitness)
```

```
            new_population.append(deepcopy(winner))

        self.population = new_population

    def crossover_operation(self):
        new_population = []

        # 保留精英
        elite_size = int(0.2 * self.pop_size)
        new_population.extend(self.population[:elite_size])

        # 对剩余个体进行交叉
        while len(new_population) < self.pop_size:
            parent1, parent2 = random.sample(self.population[elite_size:], 2)

            if random.random() < self.pc:
                child1, child2 = parent1.crossover(parent2)
                new_population.append(child1)
                if len(new_population) < self.pop_size:
                    new_population.append(child2)
            else:
                new_population.append(deepcopy(parent1))
                if len(new_population) < self.pop_size:
                    new_population.append(deepcopy(parent2))

        self.population = new_population[:self.pop_size]

    def mutation_operation(self, generation):
        for i in range(1, len(self.population)): # 跳过精英
            if random.random() < self.pm:
                self.population[i].mutate()
                self.population[i].local_search() # 每次变异后进行局部搜索

    def simulated_annealing(self, generation):
        T = self.T0 * (self.alpha ** generation)

        for i in range(1, len(self.population)): # 跳过精英
            old_solution = deepcopy(self.population[i])
            old_fitness = old_solution.fitness

            # 产生新解
            self.population[i].mutate()
            self.population[i].local_search()
            new_fitness = self.population[i].fitness

            # 计算适应度差
            delta = new_fitness - old_fitness
```

```
603
604             # 如果新解更差，也会有一定概率接受
605             if delta > 0 and random.random() > math.exp(-delta / T):
606                 self.population[i] = old_solution
607
608     def run(self):
609         self.initialize_population()
610
611         for gen in range(self.max_gen):
612             self.evaluate_population()
613             self.fitness_history.append(self.best_fitness)
614
615             print(f"Generation {gen}: Best Fitness = {self.best_fitness:.2f}, "
616                   f"Collected = {self.best_solution.total_collected:.1f}, "
617                   f"Penalty = {self.best_solution.penalty:.2f}, "
618                   f"Reward = {self.best_solution.reward:.2f}")
619
620             self.selection()
621             self.crossover_operation()
622             self.mutation_operation(gen)
623             self.simulated_annealing(gen)
624
625             # 精英保留
626             if len(self.population) > 1:
627                 self.population[-1] = deepcopy(self.best_solution)
628
629         return self.best_solution
630
631     def plot_fitness(self):
632         """绘制适应度进化曲线"""
633         plt.figure(figsize=(10, 6))
634         plt.plot(range(self.max_gen), self.fitness_history, 'b-')
635         plt.xlabel('Generation')
636         plt.ylabel('Best Fitness')
637         title = 'Weekday' if self.is_weekday else 'Weekend'
638         plt.title(f'Fitness Evolution ({title})')
639         plt.grid(True)
640         plt.show()
641
642 # 主程序
643 if __name__ == "__main__":
644     print("=== 共享单车故障车辆回收路径优化 ===")
645
646     # 分别计算最优解
647     for is_weekday in [True, False]:
648         print(f"\n=== 计算{'工作日' if is_weekday else '周末'}最优解 ===")
649
```

```
        # 选择插值器
        current_interpolators = interpolators_weekday if is_weekday else
            interpolators_weekend
        total_bikes = sum([int(np.round(f(12.0))) for f in current_interpolators])
        print(f"校园共享单车总量: {total_bikes}辆")

        # 运行算法
        ga = HybridGA(pop_size=50, pc=0.8, pm=0.1, max_gen=100, is_weekday=is_weekday)
        best_solution = ga.run()

        print("\n=== 最优回收方案 ===")
        print(f"总回收时间: {best_solution.total_time:.2f} 分钟")
        print(f"回收故障车数量: {best_solution.total_collected:.1f}辆")
        print(f"惩罚值: {best_solution.penalty:.2f}")
        print(f"奖励值: {best_solution.reward:.2f}")

        # 巡检的详情
        for insp_idx in range(4):
            print(f"\n巡检 {insp_idx+1} ({inspection_times[insp_idx].strftime('%H:%M')}
                开始):")
            for round_idx in range(2):
                print(f" 第 {round_idx+1} 轮:")
                route_names = [locations[i] for i in best_solution.round_routes[insp_idx
                    ][round_idx]]
                print("   路线: " + " -> ".join(route_names))
                print("   回收数量:", best_solution.operations[insp_idx][round_idx])
                print(f"   用时: {best_solution.inspection_times[insp_idx][round_idx]:.2f
                    } 分钟")

                # 计算本轮行驶距离
                round_distance = 0
                route = best_solution.round_routes[insp_idx][round_idx]
                for i in range(len(route)-1):
                    round_distance += C[route[i]][route[i+1]]
                print(f"   行驶距离: {round_distance/1000:.2f} km")

        # 适应度进化曲线
        ga.plot_fitness()

        # 巡检路线图
        best_solution.plot_routes()
```