《高级算法设计与分析》课程报告

无人机配送路径规划问题

姓　名：司若愚　　学　号：2023282210298

学　院：国家网络安全学院　专　业：电子信息

课　程：高级算法设计与分析　教　师：林海

二〇二四年六月

一、 作业要求：

无人机可以快速解决最后 10 公里的配送，本作业要求设计一个算法，在一个区域内实现无人机配送的路径规划。在此区域中，共有 j 个配送中心，任意一个配送中心有用户所需要的商品，其数量无限，同时任一配送中心的无人机数量无限。该区域同时有 k 个卸货点（无人机只需要将货物放到相应的卸货点即可），假设每个卸货点会随机生成订单，一个订单只有一个商品，但这些订单有优先级别，分为三个优先级别（用户下订单时，会选择优先级别，优先级别高的付费高）：一般、较紧急、紧急（分别对应用时 180、90、30 分钟）。

目标：在规定的一段时间内（如一天），最小化所有无人机的总配送路径长度，同时满足所有订单的优先级别要求。

- 系统需要做出决策：

  1. 哪些配送中心出动多少无人机完成哪些订单；
  2. 每个无人机的路径规划，即先完成那个订单，再完成哪个订单，...，最后返回原来的配送中心；

- 前提条件：

  1. 无人机一次最多只能携带 n 个物品；
  2. 无人机一次飞行最远路程为 20 公里（无人机送完货后需要返回配送点）；
  3. 无人机的速度为 60 公里/小时；
  4. 配送中心的无人机数量无限；
  5. 任意一个配送中心都能满足用户的订货需求；

二、 假设与参数配置

本报告设置大小 10*10（公里）的区域空间，基于坐标系标记配送中心和卸货点的位置，具体参数配置详情如下：

```
# 常量定义
J = 3  # 配送中心数量
K = J * 3  # 卸货点数量
N = 6  # 每次最多携带物品
T = 10  # 时间间隔（分钟）
P = [180, 90, 30]  # 优先级
PW = [0.1, 0.2, 0.7]  # 优先级权重
W = 10  # 送货区域大小（公里）
MAX_DISTANCE = 20  # 每次飞行最远距离（公里）
MAX_ORDER_NUM = N  # 每次最大订单生成数
TOTAL_TIME = 60  # 总时间（分钟）
```

为什么要提出假设？基于题设进行假设，在合理范围内补足一些额外信息，可以显著降低建模难度。

题设给出的前提条件，可以做出如下假设：

1. 首先，无条件假设所有坐标（包括配送中心、卸货点坐标或订单坐标）在可行域内均随机生成，所有数据均视为平均水平。
2. 理想配送空间是以单个配送中心为圆心、半径为 10 公里的圆形区域，适合构建平面极坐标系，但极坐标系的计算相对复杂，尤其引入多配送中心时，计算量很大，因此考虑划分 10*10 的正方形区域（100 平方公里），构建平面直角坐标系。

   其次，无人机（UAV）一次性最大航程 20 公里，速度 60 公里/小时（1 公里/分钟），其最大航程刚好满足往返 10 公里直线距离，在此前提下，圆形区域全覆盖所需配送中心最小数为 1（当且仅当配送中心位于圆心），而方形区域全覆盖所需配送中心的最小数为 3（显然成立，证明不做赘述），故设置配送中心数 J=3（随机情况下认为 3 个配送中心可全覆盖方形区域）。

   特别的，结合最高优先级 30 分钟时限和 UAV 最大航行时长 20 分钟，可以认为单线任务完成率很高，单线任务即单 UAV（或配送中心、出发地）单目的地（或卸货点）路线任务，一种简单的一对一关系。这对选择算法的基础策略有很大帮助。
3. 配送中心的 UAV 数量无限，任一配送中心均可满足用户订单需求。结合

订单的优先级信息，可解读所有订单本质上无差别，重点关注其数量和优先级，同时，不会出现无人机短缺的情况，无人机载荷量与每个卸货点最大生成订单数的比例（UAV 无限，所以不会特别高）、配送中心与卸货点的数量比不影响定性分析。

取卸货点数量是配送中心数量的3倍(减小计算量,通常认为是远大于)，即 K=3*J，同时，假定无人机载荷 N 与每个卸货点每周期可生成的最大订单数 MAX_ORDER_NUM 相同，取 N = MAX_ORDER_NUM = 6（大于等于 2 即可）。

4. 优先级包括：一般（3 小时）、较紧急（1.5 小时）、紧急（0.5 小时），等价于 180 分钟（或公里）、90 分钟、30 分钟。不妨直接定义优先级为 P = [180, 90, 30]，这样设置有利于周期性更新订单的优先级，假定周期为 T，则下周期只需将每个订单的优先级减去 T 即可。在生成订单时，取对应的概率/权重为 PW = [0.1, 0.2, 0.7]，即认为人们更倾向于选择紧急订单，这也有利于检验系统对任务优先级的响应灵敏度。

   周期的设置需要关注系统可接受的最大检测时宽，紧急订单时限 30 分钟，无人机最大航行时长 20 分钟，这表明系统最久在 30-20=10 分钟内必须检测一次，再久可能会错失订单。因此设置周期 T = 10（分钟），同时取时间总长 TOTAL_TIME = 60（分钟），系统将更新状态 6 次（生成订单+调度）。

此外，为输出决策过程和最佳方案，本报告还额外设置了一些辅助变量，用以存储记录未交付的订单、可选方案和规避已交付的订单。具体参数配置如下：

```
ORDERS = []  # 订单列表
ORDER_ID_COUNTER = 0  # 订单 ID 计数器
DRONE_ID_COUNTER = 0  # 无人机 ID 计数器
DELIVERED = []  # 规避池
CHOICES = []  # 决策池
LOG = []  # 决策记录
```

三、 算法构思与实现

1. 算法构思

本报告提出一种基于遗传算法（GA）+贪心算法的混合算法实现 UAV 系统调度。其中遗传算法部分用于模拟单 UAV 单卸货点（单线）任务调度，贪心算法部分用于合并调度路径、优化调度结果。

GA 算法中规模化的种群可以较快速的模拟出单线任务的最佳路线，但显然单线结果只是基础结果，仅仅依赖多个相同单线上的订单远不能达到预期，其总路程代价还有相当的优化空间。本报告选择一种合并最少载荷路线的贪心算法用于减少 UAV 开销。

2. 建模

该模块生成坐标空间、订单、UAV 路径方案，并构建控制逻辑。

坐标空间包括：配送中心坐标（centers）、卸货点或订单坐标（droppoints）以及全体坐标构成的距离矩阵（distance_matrix）。所有坐标均为二元组，距离矩阵（A）为二维数组，元素 A_{ij} 的值表示第 i 个坐标与第 j 个坐标间的距离。

订单的属性结构为：{'id', 'location', 'priority'}，分别表示订单 ID、订单坐标和优先级。

路径方案的树形结构为：{'center_location',' drone_id' ,' order_id' , 'route', 'distance'}，分别表示始发地（配送中心坐标）、无人机 ID、携带的订单 ID（可能多个）、派送路径（格式为[(x1,y1),(x2,y2),…]）以及路程代价。注意，单线任务下路径包含三个坐标“vc→vd→vc”（vc 表示配送中心坐标，vd 表示卸货点坐标），于是路程代价公式为：

$$D = 2 * d(vc,vd)$$

即配送中心与卸货点距离的 2 倍，其中 d(vc,vd)可通过查询距离矩阵获得。

多线任务的派送路径包含多个卸货点，为“vc→vd1→vd2→…→vdn→vc”（vdi，i=1,2,3…,n，表示多个卸货点坐标），对应的路程代价为：

$$D = d(vc,vd1) + d(vd1,vd2) + \cdots + d(vdn,vc)$$

实现函数如图 1。

```python
2 个用法
def genPos(n, exclude=None):
    """生成n个不重复的坐标点"""
    ps = set()
    while len(ps) < n:
        item = (random.randint( a: 0, W), random.randint( a: 0, W))
        if not (exclude and item in exclude):
            ps.add(item)
    return list(ps)


# 生成送货中心和卸货点坐标
1 个用法
def generate_locations(num_centers, num_droppoints):
    # 生成不包含重复元素的centers坐标
    centers = genPos(J)
    droppoints = genPos(K, exclude=centers)
    return centers, droppoints
```

图 1（a） 随机生成不重复的配送中心和卸货点坐标

```python
# 随机生成订单
1 个用法
def generate_orders(droppoints, max_orders):
    global ORDER_ID_COUNTER
    orders = []
    for dp in droppoints:
        num_orders = random.randint( a: 0, max_orders)
        for _ in range(num_orders):
            priority = random.choices(P, PW)[0]
            ORDER_ID_COUNTER += 1
            orders.append({'id': ORDER_ID_COUNTER, 'location': dp, 'priority': priority})
    # 按优先级排序订单
    return sorted(orders, key=lambda x: x['priority'])
```

图 1（b） 随机生成订单

```python
# 计算距离矩阵
1 个用法
def create_distance_matrix(locations):
    n = len(locations)
    distance_matrix = np.zeros((n, n))
    for i in range(n):
        for j in range(n):
            if i != j:
                distance_matrix[i][j] = np.linalg.norm(np.array(locations[i]) - np.array(locations[j]))
    return distance_matrix
```

图 1（c） 计算距离矩阵

系统控制逻辑的基本框架伪代码如下：

```
逻辑控制函数 :
    初始化配送中心和卸货点的坐标
    打印配送中心和卸货点的坐标
--------------------------------------
    对于 current_time 从 0 到 TOTAL_TIME，步长为 T:
        更新订单优先级并生成新订单，按优先级排序
        ------------------------------
        更新配送中心和订单位置的距离矩阵
        ------------------------------
        对于每个配送中心：
            使用遗传算法规划最佳路径方案①
            将①中所有可选调度方案加入决策池
            ---------------------------
            使用贪心算法优化路径方案②
            记录②中所有可选调度方案
            ---------------------------
        系统从决策池选择①和②中的最佳方案
        移除冲突方案，更新剩余订单列表和已配送订单列表
        ------------------------------
        紧急调度优先级低于 20 的订单
        ------------------------------
        打印当前周期和累计配送距离
        打印剩余订单
--------------------------------------
函数结束
```

逻辑详情见本章第 2~3 小节。

3. 基于遗传算法的单线任务调度

前文提到，遗传算法处理单线任务的优势在于规模化种群通过模拟生物遗传规律多次迭代选择出最佳个体（最佳配送方案）。

本报告对种群个体的定义为：订单集的一个随机排列（一维数组 A 表示，元素为订单索引号 k，长度为订单数 num_orders），元素 A[i]表示由 i 号配送中心调度无人机派送索引号 k=A[i]的订单；种群的定义为：若干个订单排列（二维数组，行数为种群大小 pop_size，列数为订单数 num_orders），最佳个体通过计算对比所有个体的元素 A[i]所对应的适应度值（UAV 距离代价）获得，适应度值最小者即为最佳个体。

例如，假设 pop_size=3，num_orders=5，则种群可能如下：

```
[[2, 0, 4, 1, 3],
 [1, 3, 2, 4, 0],
 [0, 2, 1, 3, 4]]
```

种群操作包括选择（selection）、交叉（crossover）、变异（mutate）。

- 选择（selection）操作：从种群中选择出适应度较高的一半个体，作为下一代种群的基础。

  先计算每个个体的适应度值，生成适应度值与个体的配对列表。按照适应度值对配对列表进行排序，适应度值越小表示个体越优。最后返回适应度值最好的前半部分个体。

- 交叉（crossover）操作：通过两个父代个体生成新的子代个体，继承父代的部分基因，同时保持基因序列的完整性。

  随机选择两个位置，定义父代基因片段（订单序列）的交叉区间。将父代 1 的基因片段复制到子代的对应位置。依次从父代 2 中选择不在子代中的基因，按顺序填补子代的剩余位置。

- 变异（mutate）操作：通过随机交换个体中的基因，增加种群的多样性，避免陷入局部最优解。

  遍历个体的每个基因位置，根据变异率决定是否进行变异。如果决定变异，随机选择另一个基因位置，与当前基因位置的基因进行交换。

遗传算法的逻辑是通过种群遗传迭代，寻找在特定限制条件（如单线、最大携带数量、最大飞行距离）下的最佳订单配送路径。每个配送中心面向全体或部分卸货点（小概率会超出 UAV 最长飞行距离）会产生若干个最佳个体（最短单线路径），同一卸货点产生的订单要尽可能集中在同一 UAV 派送，可局部最小化 UAV 数量。所有配送中心产生的最佳个体，都会被投入决策池中候选，具体的选择策略由系统来决定，详情见本章第 4 小节。

其伪代码如下：

```
函数 genetic_algorithm(num_orders, distance_matrix, max_carry,
max_distance, pop_size=100, generations=200, mutation_rate=0.01):
    ----------------------------------
    初始化种群 population
    初始化 best_individual 和 best_fitness 为当前最佳个体和适应度值
    ----------------------------------
    对于 generation 从 1 到 generations:
        选择操作：选择适应度值最好的个体组成新的种群
        交叉操作：两两配对进行交叉生成新的个体
        变异操作：对新生成的个体进行变异
        更新种群：将新生成的个体作为当前种群
        ----------------------------------
        计算当前种群中的最佳个体及其适应度值
        如果当前最佳适应度值优于之前的最佳适应度值，则更新 best_individual 和
best_fitness
    ----------------------------------
    返回 best_individual 和 best_fitness
```

实现函数如图 2。

```python
# 适应度函数
5 个用法
def fitness(individual, distance_matrix, max_carry, max_distance):
    total_distance = 0
    segment_distance = 0
    carried_items = 0
    for i in range(1, len(individual)):
        carried_items += 1
        segment_distance += distance_matrix[individual[i - 1]][individual[i]]

        if carried_items > max_carry or segment_distance > max_distance:
            return 1e6  # 超出携带量或最大飞行距离，适应度值设为无穷大
        else:
            total_distance += distance_matrix[individual[i - 1]][individual[i]]

    return total_distance
```

图 2（a） 受最大载荷、最远航程约束的适应度函数

```python
# 初始化种群
1 个用法
def init_population(pop_size, num_orders):
    return [np.random.permutation(num_orders).tolist() for _ in range(pop_size)]
```

图 2（b） 初始化种群

```python
# 选择操作
1 个用法
def selection(population, distance_matrix, max_carry, max_distance):
    population_fitness = [(ind, fitness(ind, distance_matrix, max_carry, max_distance)) for ind in population]
    population_fitness.sort(key=lambda x: x[1])
    return [ind for ind, _ in population_fitness[:len(population) // 2]]


# 交叉操作
2 个用法
def crossover(parent1, parent2):
    size = len(parent1)
    start, end = sorted(np.random.choice(range(size), size: 2, replace=False))
    child = [-1] * size
    child[start:end] = parent1[start:end]
    pointer = 0
    for item in parent2:
        if item not in child:
            while child[pointer] != -1:
                pointer += 1
            child[pointer] = item
    return child


# 变异操作
2 个用法
def mutate(individual, mutation_rate):
    for i in range(len(individual)):
        if np.random.rand() < mutation_rate:
            swap_idx = np.random.randint( low: 0, len(individual))
            individual[i], individual[swap_idx] = individual[swap_idx], individual[i]
```

图 2（c） 选择、交叉、变异操作

```python
# 遗传算法
1 个用法
def genetic_algorithm(num_orders, distance_matrix, max_carry, max_distance, pop_size=100, generations=200,
                      mutation_rate=0.01):
    # 初始化种群
    population = init_population(pop_size, num_orders)
    best_individual = None
    best_fitness = float('inf')
    # while best_fitness > max_distance:
    # 计算初始种群中的最佳个体和其适应度值
    best_individual = min(population, key=lambda ind: fitness(ind, distance_matrix, max_carry, max_distance))
    best_fitness = fitness(best_individual, distance_matrix, max_carry, max_distance)

    # 迭代遗传算法
    for _ in range(generations):
        # 选择操作
        population = selection(population, distance_matrix, max_carry, max_distance)
        next_population = []

        # 交叉和变异操作
        pop_size = len(population)  # 重新计算选择后的种群大小
        if pop_size % 2 != 0:  # 确保种群大小为偶数
            population.append(population[0])
            pop_size += 1

        for i in range(0, pop_size, 2):
            parent1, parent2 = population[i], population[i + 1]
            child1, child2 = crossover(parent1, parent2), crossover(parent2, parent1)
            mutate(child1, mutation_rate)
            mutate(child2, mutation_rate)
            next_population.extend([child1, child2])

        # 更新种群
        population = next_population

        # 计算当前种群中的最佳个体和其适应度值
        current_best = min(population, key=lambda ind: fitness(ind, distance_matrix, max_carry, max_distance))
        current_fitness = fitness(current_best, distance_matrix, max_carry, max_distance)

        # 更新最佳个体和其适应度值
        if current_fitness < best_fitness:
            best_individual, best_fitness = current_best, current_fitness

    return best_individual, best_fitness
```

图 2（d） 遗传算法主体函数

4. 基于贪心算法的多线任务优化

第一个优化思路是通过贪心算法合并路径。

本报告贪心算法的核心思想是，通过合并单线路径成为回路，来减少 UAV 群的数量。而本小节贪心算法合并操作建立在本章第 2 小节遗传算法产生的若干局部最佳方案上。以下示例可描述其合并思路。

坐标为(10,7)的配送中心的 GA 算法调度方案如下：

```
无人机[28]派送订单：[45, 46, 47]，路径：[(10, 7), (7, 5), (10, 7)]，路程：7.21 km
无人机[32]派送订单：[43, 44]，路径：[(10, 7), (9, 9), (10, 7)]，路程：4.47 km
```

经贪心算法优化后的调度方案如下：

```
无人机[34]派送订单：[43, 44, 45, 46, 47]，路径：[(10, 7), (9, 9), (7, 5), (10, 7)]，路程：10.31 km
```

这个逻辑是显而易见的，如果两个单线任务载荷订单总数没有超过 UAV 载荷上限：3 + 2 = 5 ≤ 6，且合并后的路程代价（当然，需要不超过 UAV 最远航程）比原来两个路程代价之和小：10.31 < 7.21 + 4.47 = 11.68，则可合并到同一路径方案。

但仍有两个问题有待解决，从哪些方案开始合并、合并到什么时候停止。回归到本节开头的核心思想，为最小化 UAV 数量（一个 UAV 就代表一条路径方案），找到集合中载荷量最小的两个 UAV 尝试合并，合并成功生成新 UAV，移除这两个 UAV，重复合并步骤；合成失败或集合中 UAV 数量降至 1 则终止合并。

上述控制逻辑伪代码如下：

```
函数 合并无人机路径集合(无人机列表, 所有地点, 距离矩阵):
    ----------------------------------------
    当 无人机数量 >= 2 时:
        选择订单数最少的无人机 min1
        找到 min1 路径中最后一个地点的索引 idx1
        从列表中移除 min1
        ------------------------------------
        选择剩余无人机中订单数最少的无人机 min2
        找到 min2 路径中第一个地点的索引 idx2
        从列表中移除 min2
        ------------------------------------
        尝试合并 min1 和 min2 (使用 idx1 和 idx2 之间的距离)
        如果合并成功:
            将新无人机加入列表
        否则:
            将 min1 和 min2 加回列表
            退出循环
    ----------------------------------------
```

合并完成后得到最终的优化方案集合。

但这是否意味着可以放心丢弃 GA 算法得到的基础方案，而全部替换为优化后的方案呢？答案是否定的。

第二个优化思路是延迟丢弃 GA 算法方案，同贪心算法方案一同投入决策池。原因是可能会出现这种情况：两个配送中心 A 和 B，各自均有 GA 算法基础方案集合（A’、B’）和贪心算法优化方案集合（A*、B*）。

假定 B’中某方案 b’优于 A’中某方案 a’，同时 b’参与合并到 B*中的某方案 b*，而 A*中某方案 a*（注意 a’和 a*无关）优于 b*，即 b’≤a’，b* ≥ a*（代价小者为优）。如若在合并完成后立即丢弃基础方案 b’，最终系统决策时会选择 a*和 a’，但显然理想情况下应选择 a*和 b’。合理的操作是将基础方案和优化方案都投入决策池。

上述功能代码实现如图 3。

```python
# 合并两条路径
1 个用法
def merge_drone(d1, d2, dis):
    global DRONE_ID_COUNTER
    # 如果两个无人机的订单数相加不超过N，且总距离不超过MAX_DISTANCE，则合并
    orders = d1.orders + d2.orders
    half = (d1.distance + d2.distance) / 2
    new_dis = half + dis
    if len(orders) <= N and new_dis <= MAX_DISTANCE and dis < half:
        DRONE_ID_COUNTER += 1
        return Drone(DRONE_ID_COUNTER, orders, d1.route + d2.route, new_dis)
    else:
        return None


# 合并路径集合
1 个用法
def merge_drones(drones, all_locations, distance_matrix):
    # 如果drones数量不小于2，找出其中订单数最少的两个尝试合并
    while len(drones) >= 2:
        min1 = min(drones, key=lambda x: len(x.orders))
        idx1 = all_locations.index(min1.route[-1])
        drones.remove(min1)
        min2 = min(drones, key=lambda x: len(x.orders))
        idx2 = all_locations.index(min2.route[0])
        drones.remove(min2)
        merged = merge_drone(min1, min2, distance_matrix[idx1][idx2])
        if merged:
            drones.append(merged)
        else:
            drones.append(min1)
            drones.append(min2)
            break
```

图 3 贪心算法主体函数

5. 系统决策与调度

前文第 2~4 小节描述的是每个配送中心各自的最佳方案集合（包括 GA 方案和贪心方案），但最终还需要系统从全体配送中心全体方案中做出最后一步最佳决策。这层决策需要用到辅助变量决策池（CHOICES）、规避池（DELIVERED）和决策记录（LOG）。选择策略伪代码如下：

```
将 CHOICES 按距离排序
将 CHOICES 按订单数量排序（降序）
----------------------------------------
当 CHOICES 不为空时：
    选择最佳方案 best_choice
    从 CHOICES 中移除 best_choice
    累加最佳方案的距离到 total_dis
    将最佳方案的订单加入已配送订单 DELIVERED
    记录最佳方案到 LOG
    从 CHOICES 中移除与已配送订单有交集的方案
```

```
-----------------------------------------
更新 ORDERS，移除已配送订单
-----------------------------------------
对于剩余订单中的每个订单：
    如果订单的优先级 <= 20：
        找到最近的配送中心 min_center
        调度无人机派送该订单
        从 ORDERS 中移除该订单
        累加紧急调度的距离到 total_dis
-----------------------------------------
打印当前周期的总距离 total_dis
累加当前周期距离到 TOTAL_DISTANCE
打印累计距离 TOTAL_DISTANCE
```

决策的核心思想为：将全体决策方案按（载荷订单数，路程代价）复合排序，优先订单数多的靠前、其次代价小的靠前。每次调度认定决策池（CHOICES）的头部方案为最优方案（记作 R*），取出并将载荷订单号计入规避池集合（DELIVERED）。

可以肯定的是，选出 R*后决策池中载荷量与 R*相同、且所载订单集合与规避池集合有交集的方案路径代价一定比 R*大，可直接移除。重复上述调度步骤，直到剩余订单（或决策池）为空。最终可以得到每周期内的全体最佳方案集合{R*}。

事实上，报告的参数配置可保证优先级为 30 的订单不会滞留到下一周期（上百次模拟结果未出现），原因是订单每次更新都会按优先级顺序排序，即优先级低的订单总会优先得到派送。但为避免极端情况，得到{R*}后，系统检查剩余订单中是否滞留有优先级低于 20 的订单，有则立即加派 UAV 派送。可能会有优先级为 90、180 的订单滞留到后续周期，但基本都可在 2 个周期内得到处理。

实现代码见图 4。

```python
print(f"【系统调度最佳方案】")
# print('CHOICES', CHOICES)
while CHOICES:
    # 选择最优方案
    best_choice = CHOICES.pop(0)
    total_dis += best_choice[3]
    # 描述最佳方案
    print(
        f"    分配中心{best_choice[-1]}调度无人机[{best_choice[0]}]派送订单：{best_choice[1]}, 路径：{best_choi
    best_orders = best_choice[1]
    LOG.append(best_choice)  # 记录决策
    DELIVERED += best_orders  # 记录已配送订单
    # 移除最佳方案的互斥方案
    # 即移除CHOICES中订单对应的集合和DELIVERED中订单对应的集合有交集的元素
    for choice in CHOICES[::-1]:
        if set(choice[1]) & set(DELIVERED):
            CHOICES.remove(choice)
# 更新订单
ORDERS = [order for order in ORDERS if order['id'] not in DELIVERED]
# 紧急调度
```

图 4（a） 系统调度最佳方案{R*}

```python
# 紧急调度
# 如果发现剩余订单有优先级低于20的，立即找到最近的配送中心加派无人机
for order in ORDERS:
    if order['priority'] <= 20:
        min_dis = float('inf')
        min_center = None
        for center in centers:
            dis = distance_matrix[all_locations.index(center)][all_locations.index(order['location'])]
            if dis < min_dis:
                min_dis = dis
                min_center = center
        DRONE_ID_COUNTER += 1
        DELIVERED.append(order['id'])
        total_dis += min_dis * 2
        print(f"【紧急调度】")
        print(
            f"    分配中心{min_center}调度无人机[{DRONE_ID_COUNTER}]派送订单：{order['id']}, 路径：[{min_center}, {order
        ORDERS.remove(order)
print(f"   *当前周期共计：{total_dis:.2f} km")
TOTAL_DISTANCE += total_dis
print(f"   *累计（0~{current_time + T}分钟）共计：{TOTAL_DISTANCE:.2f} km")
# 描述剩余订单
print(f"【剩余订单】")
for order in ORDERS:
    print(
        f"    订单[{order['id']}]：卸货点{order['location']}, 优先级{order['priority']}")
```

图 4（b） 紧急调度（订单优先级低于 20）

四、 结果分析

报告代码见同目录。代码通过 python 实现，环境 python3.7。

运行实验，取 5 次模拟结果，分别存入 5 个 txt 文件。部分结果如图 5，全体结果见同目录。

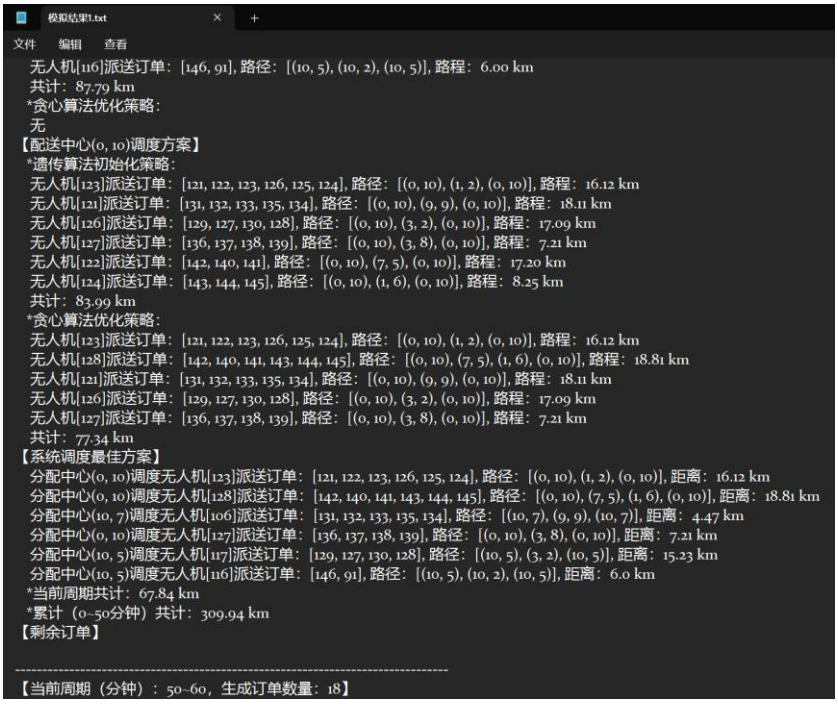

图 5 部分模拟结果

本报告针对优化方案给出评价公式：

$$\rho = 1 - D_{\{R^*\}} / (\Sigma_C D_{GA} / C)$$

其中，ρ 为优化率，$D_{\{R^*\}}$ 表示系统调度得到的全局最佳方案集合的总路程代价，$\Sigma_C D_{GA}$ 表示全体周期全体配送中心通过 GA 方案（或单线调度）得到全体基本方案的路程代价之和，$\Sigma_C D_{GA}/C$ 表示取平均值（C 为配送中心数量，随机前提下认为所有配送中心平均分配了订单）。

在给定参数下，五次模拟结果的数据如表 1。

表 1 模拟结果数据分析（C=3）

表头序号	*1*	*2*	*3*	*4*	*5*
$D_{\{R^*\}}$	357.9	377.91	294.48	411.33	318.57
$\Sigma_C D_{GA}$	1464.39	1387.51	1198.8	1694.91	1329.27
$\Sigma_C D_{GA}/C$	488.13	462.5	399.6	564.97	443.09
ρ	0.267	0.183	0.263	0.272	0.281

实验结果表明，五次模拟结果的优化率分别为 26.7%，18.3%，26.3%，27.2%，28.1%，去掉最优与最差结果求平均 $\bar{\rho} = 0.267$，即相比单线调度策略，本报告优化后的调度策略平均可降低路程代价 26.7%左右。