无人机配送路径规划问题

1. 问题分析：

多配送中心带容量约束的无人机路径问题（MDCVRP）是经典VRP问题的扩展，它研究的是有多个配送中心中带容量限制的无人机同时对若干个卸货点进行服务，每个卸货点都有一定的货物需求。所要确定的是卸货点由哪个配送中心服务，并对服务时的无人机路径进行优化，以便达到路径最短的目标。

除此之外，该问题还是一个带时间窗约束的无人机路径规划问题（VRPTW），VRPTW 不仅考虑经典VRP问题的所有约束，还需要考虑时间窗约束，也就是每个卸货点对应一个时间窗 $[e_i, l_i]$，其中 e_i 和 l_i 分别代表该点的最早到达时间和最晚到达时间。卸货点 $i \in V$ 的需求必须要在其时间窗内被送达。

2. 场景设计

单向：送货；
多配送中心：存在多个配送中心和卸货点；
无人机封闭：完成配送任务的无人机需回到配送中心；
无人机充足：不限制无人机数量，即配送无人机需求均能满足；
时间窗限制：卸货点的需求必须要在其时间窗内被送达。

3. 要求

优化目标：最小化无人机总行驶距离
约束条件：单一无人机行驶距离约束，重量约束，时间窗约束；
已知信息：配送中心位置、卸货点位置、卸货点需求、无人机最大载重、无人机最大行驶距离、无人机速度

4. 数学模型

4.1 符号说明

j表示配送中心数量，k表示卸货点数量;

n表示使用的无人机数，m表示无人机集合（m = 1, 2, 3. . . n）；

D表示车辆最大形式距离；

d_{ij}表示两个节点的距离；

q_k表示某一轮第k个卸货点的需求

Q表示无人机的最大载重

决策变量：

$$x_{ijn} = \begin{cases} 0, \text{其他} \\ 1, \text{表示无人机} n \text{从} i \text{行驶到} j \end{cases}$$

4.2 建立模型

目标函数：行驶的总路程最小

$MinD = \sum_{i=1}^{j} \sum_{j=1}^{k} \sum_{n=1}^{m} d_{ij} x_{ijn}$

约束：

1. 配送中心约束：所有无人机均从配送中心出发，完成所有的配送任务后返回配送中心。

   $\sum_{i=1}^{j} \sum_{j=1}^{k} \sum_{n=1}^{m} x_{ijn} = \sum_{i=1}^{j} \sum_{j=1}^{k} \sum_{n=1}^{m} x_{jin}$

2. 卸货点流量平衡：进出无人机数相等

   $\sum_{j=1}^{k} x_{ijn} = \sum_{j=1}^{k} x_{jin} (n \in m)$

3. 重量约束：每台无人机的装载量不能超过其最大载重量限制

   $\sum_{i=1}^{j} \sum_{j=1}^{k} x_{ijn} q_j \leq Q$

4. 卸货点服务约束：每个卸货点在一轮内只被服务1次

   $\sum_{n=1}^{m} \sum_{i=1}^{j} x_{ijn} = 1$

5. 无人机行驶距离约束：每台无人机配送不超过最大配送距离

   $\sum_{i=1}^{j} \sum_{j=1}^{k} x_{ijn} dij \leq D (n \in m)$

6. 时间窗约束：在某一轮产生的订单在一段时间后必须要在其时间窗内被送达

   $Tq_k + T_w \in [e_i, l_i]$

5.代码设计

5.1 整体框架设计

带时间窗的有容量约束多配送中心VRP问题（MDCVRPTW）是一个NPC问题，直接求得其最优解很有难度。经过思考，我决定将该问题拆分为两个阶段进行：①首先先将卸货点分配给配送中心，转化为单配送中心（CVRPTW）问题；②在对每一个配送中心的路径进行优化。

代码的核心框架逻辑如下：

1. 读取超参数并初始化相应数据
2. 根据距离将卸货点分配给最近的配送中心，将MDCVRPTW问题变成CVRPTW问题
3. 开始时间模拟，每一轮配送中心生成合法订单
4. 根据订单的时间窗约束分配当前需要送的订单给已经分配好的配送中心
5. 按照贪心的方式尽可能分配更少的无人机，使无人机尽可能装满
6. 使用遗传算法计算并生成合法最短路径
7. 结束时间模拟，输出最终结果

5.2 读取超参数并初始化相应数据

本次要求自己生成数据，考虑大作业要求生成编写超参数config文件

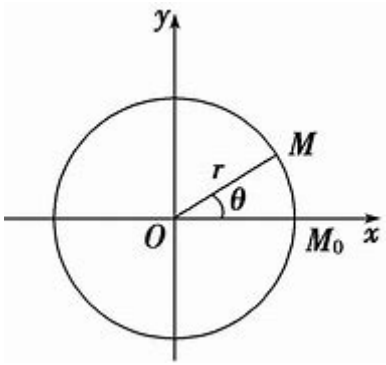
```
drop_points_num: 15         #卸货点个数
depots_num: 5               #配送中心个数
drone_speed: 1              #无人机速度，60km/h=1km/min
max_distance: 4             #配送中心/卸货点之间的最大距离
one_round_order_time: 15    #生成订单一轮的时间
simulation_time: 720        #模拟720min(12hour)
max_order_num: 10           #一次最多生成10个订单
```

初始节点生成算法：

核心思路为按照距离和角度产生大致均匀分布的点，且每个点之间存在一定扰动，按数目返回相应的距离或角度列表。

```
def generate_spaced_distances(num_points: int, max_distance: float,
reverse=False) -> list[float]:
    uniform_points = np.linspace(0, max_distance, num_points)

    jitter = np.random.uniform(-0.5, 0.5, num_points) * (max_distance /
num_points)
    jittered_points = uniform_points + jitter

    jittered_points = np.clip(jittered_points, 0, max_distance)

    result = jittered_points.tolist()

    if reverse:
        return result[::-1]
    else:
        return result
```

之后使用类似极坐标的方式将随机生成的列表排布到二维平面中，r为随机生成的距离大小，θ代表随机生成的角度大小



5.3 根据距离将卸货点分配给最近的配送中心

经过考虑，使用最简单的最近距离分配方式，将卸货点分配给配送中心，转化为单配送中心（CVRPTW）问题，也就是说每一个卸货点的生成订单都会分配给相应的配送中心。

这样做的前提是每一个配送中心无人机数量是无限的，那么有

$$\sum_{n=1}^{m} d_i > \sum_{n=1}^{m} d_{min}$$

因此，这种分配算法能够优于将卸货点随机分配配送中心等这类随机算法

分配卸货点给最近的卸货中心的核心逻辑如下：

1. 计算当前卸货点与周围卸货中心的距离
2. 将最近的配送中心分配给该卸货点

```
def distribute_depot(drop_points: list[DropPoint], depots: list[DePot]):
    for drop_point in drop_points:
        ...
        for depot in depots:
            distance = calculate_distance([drop_point.x, drop_point.y],
[depot.x, depot.y])
        ...
                drop_point.depot = depot
```

5.4 每一轮配送中心生成合法订单

每一轮生成合法订单的算法核心逻辑如下：

1. 随机生成超参数中规定个订单, 每个生成订单含优先级对应时间窗和相应阈值，阈值等于时间窗大小
2. 对于当前生成的订单进行检查， 由于当前卸货点与周围卸货中心的距离的限制，如果直接送都会超过当前优先级的阈值，就认为其是一个非法订单
3. 将合法订单加入订单队列

```
def generate_new_orders(current_orders_num: int, drop_points: list[DropPoint],
current_time: int, max_orders: int) -> list[Order]:
    ...
    for i in range(len(drop_points)):
        for _ in range(random.randint(0, max_orders)):
            ...
            priority = random.choice([0, 1, 2])
            if priority == 0:
                time_window = (current_time, current_time + 180)
                threshold = 180

            elif priority == 1:
                time_window = (current_time, current_time + 90)
                threshold = 90

            else:
                time_window = (current_time, current_time + 30)
                threshold = 30
    ...
            if calculate_distance([destination.x, destination.y],
[drop_points[i].x, drop_points[i].y]) / 1 <= threshold:
                orders.append(Order(order_id,destination , demand, time_window,
priority))

    return orders
```

5.5 订单调度

每一轮订单调度的算法核心逻辑如下，

1. 将当前订单队列中所有订单按照优先级排序
2. 查询订单队列，将能在当前时间窗进行发送的订单全部加入发送队列（无人机数量无限）
3. 按照之前分配好的配送中心与卸货点的关系将订单分配给相应的配送中心

按照超参数中的设置，图中存在15个卸货点和5个配送中心，平均1个配送中心会接受3个卸货点的订单

```
def check_time_due(orders: list[Order], drone_speed: int,
dealing_window:Tuple[float, float] ) -> list[Order]:
    ...
    for order in orders:
        ...
        transport_time = calculate_distance(depot, [order.destination.x,
order.destination.y]) / drone_speed
        if order.time_window[1] -  transport_time <= dealing_window[1]:
            due_orders.append(order)
    ...
    logger.info("Due orders delivering...")
    logger.info(f"Due orders: {len(due_orders)}")
    return due_orders
```

5.6 分配无人机

在分配完当前轮需要传输的订单后，对于无人机数目进行匹配，由于无人机数目是无限的，因此能够想到的最简单的匹配的算法如下：

1. 对于配送中心i对应的j个卸货点，派出k类无人机进行服务（$j = k$），也就是说，每一类无人机
2. 对于订单队列中某一卸货点的订单O_j，使用$\sum O_j/C_k$的无人机进行运送，其中，C_k代表规定的无人机容量，这里为25

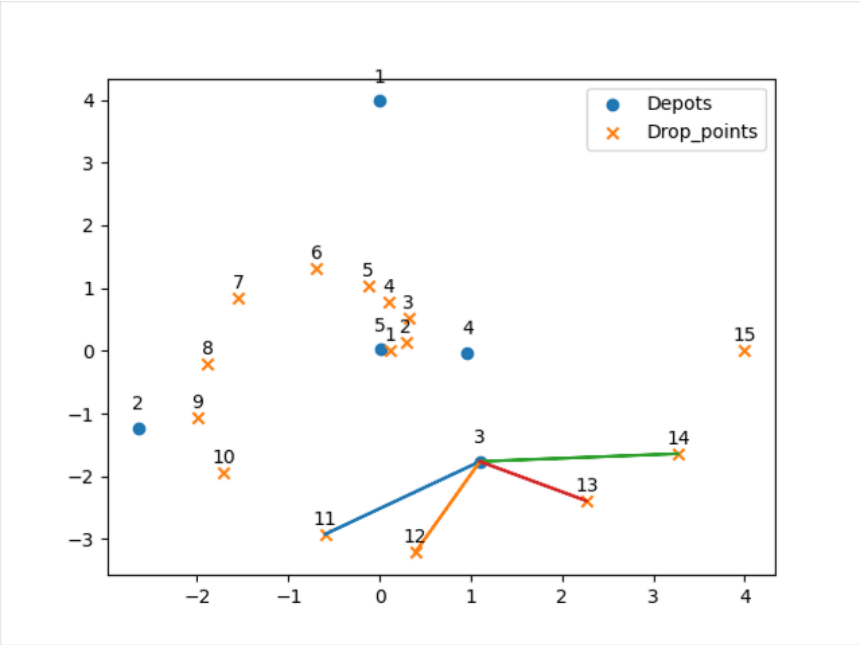
```
def greed_drone_num(orders:list[Order], drone_id: int) -> list[Drone]:
    ...
    drone_capacity = 25

    num_drones_needed = math.ceil(sum([order.demand for order in orders]) /
drone_capacity)
    for i in range(num_drones_needed):
        allocated_drones = initial_drones(num_drones_needed, drone_capacity, 1)

    ...
    return allocated_drones
```

这种算法会形成类似于下图的分配方式，且这种算法显然可行，如图所示：在某一轮次中，11，12，13，14号卸货点被分配给了3号配送中心，配送中心会不断发送4类无人机去单一目的地并直接返回。

但这种算法会出现无人机空载的情况，从而浪费大量无人机。原因是在某一轮次分配给配送中心的订单数为卸货点订单数之和

$$O = \sum_{j \in i} O_j$$

而每一卸货点的订单数不同，很可能存在O_j/C_k的情况。



5.7 优化

事实上，5.6中算法就已经可以解决问题。但是，该算法会浪费大量无人机，因此也会增加无人机行驶总路程，这与优化目标不符。因此需要进一步优化。

5.7.1 优化策略一：分配更少的无人机

考虑到5.6中算法会造成大量的无人机空载，优化算法首先可做的是分配更少的无人机，分配更少的无人机算法核心逻辑如下：

1. 遍历当前无人机队列，检查一台无人机是否能在不违反容量和时间的约束的情况下接受另一台无人机的订单
2. 若可以就将被合并无人机弹出无人机队列
3. 持续遍历直到无人机数目达到最小

分配更少无人机的核心代码如下：

```
def drones_merge(drones: list[Drone], depot: Tuple[float, float],
current_time:float) -> list[Drone]:
    ...
    while aggregatation_flag:
        ...
        for i in range(len(drones)):
            for j in range(i+1, len(drones)):
                if check_drone_capacity(drones[i], drones[j]) and
check_drone_time(drones[i], drones[j], depot, current_time):
                    ...
                    drones.pop(j)
        ...

    return drones
```

5.7.2 优化策略二：使用遗传算法计算并生成合法最短路径

如果使用5.6中算法，配送中心执行的路径是一定的，无需使用算法寻找最短路径。但在分配更少无人机后，对于某一台无人机而言，执行路径变成了一个多节点回路，因此需要求解一个旅行商问题，这同样是一个NPC问题，当然，我是将最大的NPC问题（MDCVRPTW）多配送中心带容量和时间窗约束的无人机路径问题转换为了

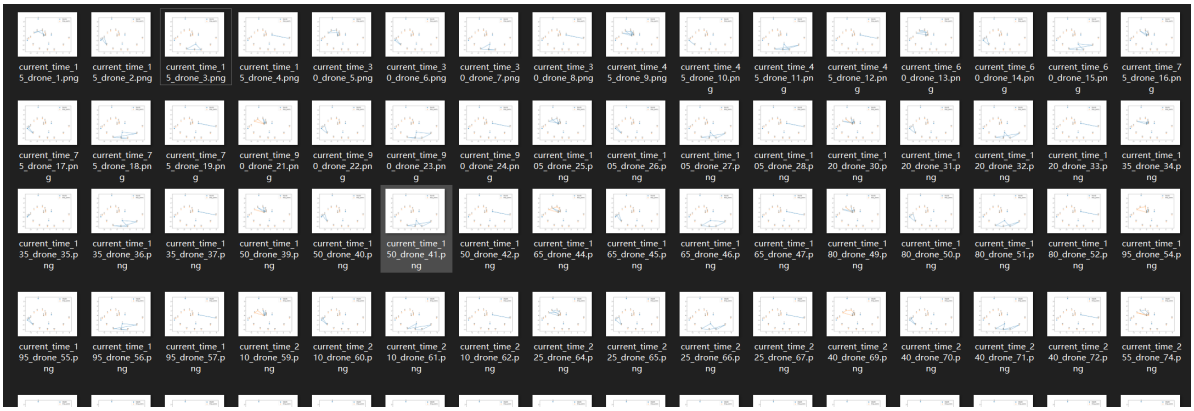
三个贪心问题（分配卸货点 + 分配订单 + 减少无人机）和最后的旅行商问题。这里我使用遗传算法对于某一无人机D_n需要去的节点进行最短合法路径寻找，遗传算法的核心逻辑如下：

1. 初始化种群，这里设置种群数目为100
2. 设置适应度函数为无人机行驶最短路径和
3. 选取一半好的种群作为下一代
4. 适度变异，这里设置变异率为0.1
5. 检查生成的解是否合法，如果路径长度超过无人机的最大行驶距离20km，则不会将解加入下一代
6. 迭代，这里设置为100次

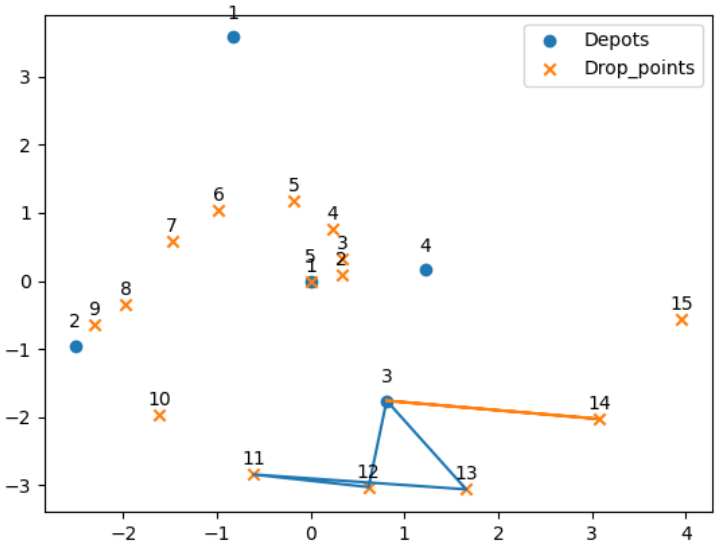
```
def gene_algo(drone: Drone, depot: Tuple[float, float]):

    initial_population_num = 100
    generation_num = 100
    ...

    for _ in range(generation_num):
        ...
        selected_fitness_distances = []
        for route in selected_routes:
            ...
            fitness_distance += calculate_distance([route[-1].x, route[-1].y],
depot)
            if fitness_distance <= 20:
                selected_fitness_distances.append(fitness_distance)

        optimal_route = selected_routes[0]

        ...
        selected_routes = selected_routes[: math.ceil(len(selected_routes) / 2)]
        ...

        for i in range(len(selected_routes)):
            mutation_rate = 0.1
            ...

    drone.route = optimal_route
```

5.8 结果分析

在5.2中设置的超参数下模拟12小时（720min），每一轮订单生成、分配的间隔为15min，因此，每15min生成一轮结果图，一张图代表某一时刻某一配送中心无人机的运送路径，最终生成的结果如下图，一共使用了238台无人机。



单一图片结果如下，如图所示，在585min（9.75hour）时，3号配送中心发送了2台满载无人机，分别服务14号卸货点和11，12，13号卸货点。



对于单一无人机的路程输出，设置日志追踪进行提醒，确保生成的结果都是合法解。

```
if round_distance > 20:
    logger.warning(f"Drone {drone.drone_id} out of gas")
```

模拟12小时的日志输出结果如下，如图所示，日志输出了每一台无人机的行驶距离、编号和最终的无人机总行驶距离，且没有出现警告，表明成功对于该问题进行了求解。

```
INFO     Drone 235 Round distance: 4.2190895127835395 km                    draw_solution.py:70
INFO     allocate due orders to depot...                                    main.py:126
INFO     start from depot 2...                                              main.py:127
INFO     allocate path for drone 236                                        main.py:137
[06/28/24 00:18:49]
INFO     Drone 236 Round distance: 3.8379989314115344 km                    draw_solution.py:70
INFO     allocate due orders to depot...                                    main.py:126
INFO     start from depot 3...                                              main.py:127
INFO     allocate path for drone 237                                        main.py:137
INFO     Drone 237 Round distance: 8.839978648603498 km                     draw_solution.py:70
INFO     allocate due orders to depot...                                    main.py:126
INFO     start from depot 4...                                              main.py:127
INFO     allocate path for drone 238                                        main.py:137
INFO     Drone 238 Round distance: 5.670160265557832 km                     draw_solution.py:70
INFO     The total distance covered by all drones is 1315.1550201474643 km  main.py:150
```

对于优化前的算法，模拟12小时的日志输出结果如下，从图中可以看出，优化前算法使用了682台无人机，远远超过了优化后算法的238台，表明确实存在大量的空载无人机。总路程为1855.74km，也大于优化后算法的路程1315.16km，相比优化目标最短路程而言，优化后算法比优化前提升了**141%**。

```
INFO     allocate path for drone 677                                                    main.py:138
INFO     allocate path for drone 678                                                    main.py:138
INFO     allocate path for drone 679                                                    main.py:138
INFO     Drone 677 Round distance: 0.935971015921489 km                 draw_solution.py:70
INFO     Drone 678 Round distance: 1.6652521823911566 km                draw_solution.py:70
INFO     Drone 679 Round distance: 4.836641640611576 km                 draw_solution.py:70
INFO     allocate due orders to depot...                                        main.py:126
INFO     start from depot 4...                                                  main.py:127
INFO     allocate path for drone 680                                            main.py:138
INFO     allocate path for drone 681                                            main.py:138
INFO     allocate path for drone 682                                            main.py:138
INFO     Drone 680 Round distance: 5.958332978392782 km                 draw_solution.py:70
INFO     Drone 681 Round distance: 5.563329080260405 km                 draw_solution.py:70
INFO     Drone 682 Round distance: 5.394403976582259 km                 draw_solution.py:70
INFO     The total distance covered by all drones is 1855.7378967161053 km      main.py:151
```

	无人机数目/台	总配送路程/公里
优化前算法	682	1855.74
优化后算法	238	1315.16