高级算法大作业

2023282210265 朱德范

问题描述

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，实现一定区域的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。图和订单数据（订单生成）自行生成。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：3小时内配送到即可；
- 较紧急：1.5小时内配送到；
- 紧急：0.5小时内配送到。

  将时间离散化，也就是每隔t分钟，所有的卸货点会生成订单（0-m个订单），同时每隔t分钟，系统要做成决策，包括：

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；

注意：系统做决策时，可以不对当前的某些订单进行配送，因为当前某些订单可能紧急程度不高，可以累积后和后面的订单一起配送。
目标：一段时间内（如一天），所有无人机的总配送路径最短
约束条件：满足订单的优先级别要求
假设条件：

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。

配送地图生成

地图生成分为配送中心与卸货点生成，根据问题描述并结合现实需要，配送地图需满足以下条件：

1. 配送中心之间的距离应该适中，不应该过远或过近；
2. 卸货点到最近配送中心的距离不大于10km。
   结合上述条件，采用配送中心与卸货点完全随机生成。
   参数设置：地图大小设置为40×40，配送中心数量为5个，围绕每个配送中心周围生成10个卸货点

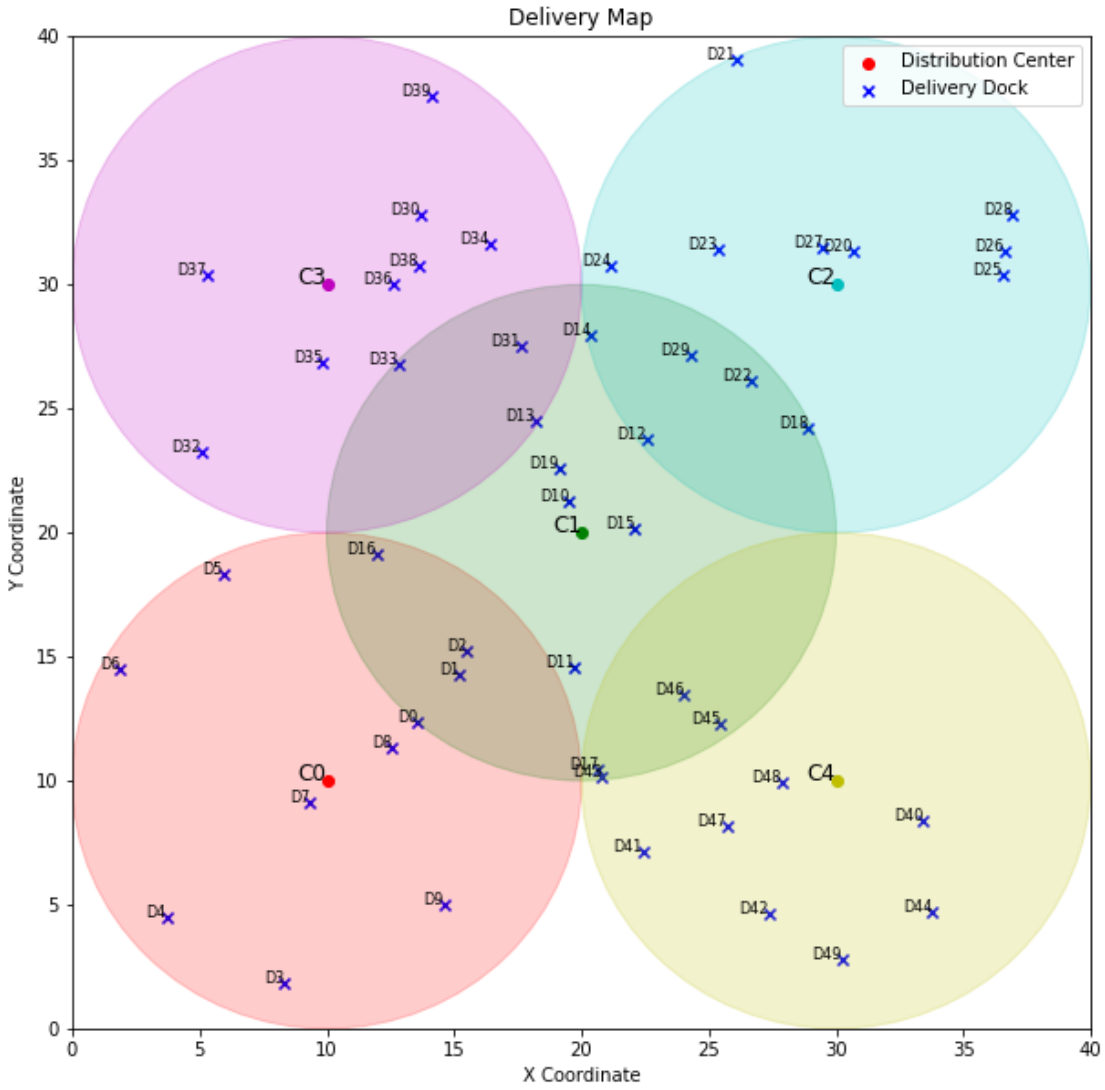
- 完全随机生成

```
# 在指定范围内生成配货点，确保两两之间的距离适中
def generate_distribution_centers(n, w, u, range_size):
    distribution_centers = []
    while len(distribution_centers) < n:
        new_point = (random.uniform(0, range_size), random.uniform(0,
range_size))
        if all(u ≥distance(new_point, point) ≥ w for point in
distribution_centers):
            distribution_centers.append(new_point)
    return distribution_centers

# 围绕每个配货点生成卸货点
def generate_delivery_docks(distribution_centers, k, dock_range):
    delivery_docks = []
    for center in distribution_centers:
        for _ in range(k):
            angle = random.uniform(0, 2 * math.pi)
            radius = random.uniform(0, dock_range)
            new_dock = (center[0] + radius * math.cos(angle), center[1] +
radius * math.sin(angle))
            # 确保卸货点在有效范围内
            if 0 ≤ new_dock[0] ≤ range_size and 0 ≤ new_dock[1] ≤
range_size:
                delivery_docks.append(new_dock)
    return delivery_docks
```

这样简单的完全随机生成由于限制条件的存在与地图大小的限制，在配送中心生成时会出现无法完成生成的情况，限于死循环。对此，采用简单直接的配送中心定点来解决。

我们将配送中心定到地图上的点中，让卸货点围绕这选定的五个点生成，最终得到地图结果如下：



图中C_i表示配送中心，D_i表示卸货点，颜色不同的圆形为以配送中心为圆心10km距离的范围。

订单生成

订单生成主要采用每个卸货点随机生成订单的方式，这也符合现实情况。

```
# 订单生成
def generate_orders():
    orders = []
    for dock_id in range(num_docks):
        num_orders = random.randint(0, max_orders_per_interval)
        for _ in range(num_orders):
            priority = random.choice(list(priority_levels.keys()))
```

```
            orders.append((dock_id, priority))
    return orders
```

下图为生成的部分订单：

```
生成的订单如下：
卸货点ID: 0, 优先级: 紧急
卸货点ID: 0, 优先级: 紧急
卸货点ID: 0, 优先级: 一般
卸货点ID: 0, 优先级: 一般
卸货点ID: 0, 优先级: 较紧急
卸货点ID: 1, 优先级: 紧急
卸货点ID: 1, 优先级: 一般
卸货点ID: 1, 优先级: 一般
卸货点ID: 44, 优先级: 一般
卸货点ID: 45, 优先级: 较紧急
卸货点ID: 45, 优先级: 紧急
卸货点ID: 45, 优先级: 紧急
卸货点ID: 45, 优先级: 紧急
卸货点ID: 45, 优先级: 较紧急
卸货点ID: 46, 优先级: 一般
卸货点ID: 46, 优先级: 一般
```

路径规划

该任务的无人机路径规划主要目的是根据订单的优先级和无人机的最大飞行距离及携带物品数量，规划无人机从配送中心出发执行订单的路径。

- 初始化和排序
  - 按优先级排序订单：首先，根据订单的优先级对订单列表进行排序。优先级高的订单排在前面，优先级低的订单排在后面。这一步确保了紧急订单优先被处理。
- 初始化变量
  - 总距离：初始化一个变量用于记录无人机飞行的总距离。
  - 当前无人机位置：初始化为配送中心的位置，因为无人机从配送中心出发。
  - 路径列表：用于存储每次无人机出发执行的订单及其路径距离。
- 循环处理所有订单
  - 在有未处理的订单时，进入一个循环进行订单处理。
- 初始化本次出发的变量
  - 本次出发的订单列表：初始化为空列表，用于记录本次出发要处理的订单。
  - 本次出发的路径距离：初始化为0，用于记录本次出发的总路径距离。
  - 当前载货量：初始化为0，用于记录无人机当前携带的货物数量。
- 为本次出发选择订单
  - 在还有未处理的订单且当前载货量未达到最大载货量时，进行以下步骤：
    - 获取下一个订单：从排序后的订单列表中取出第一个订单，即优先级最高的订单。
    - 计算距离：计算当前无人机位置到该订单对应的卸货点的距离，以及从该卸货点回到配送中心的距离。
    - 计算总距离：计算本次出发的总路径距离，包括从当前地点到下一个卸货点的距离以及从下一个卸货点回到配送中心的距离。
    - 判断距离限制：如果本次出发的总路径距离不超过无人机的最大飞行距离，则将该订单添加到本次出发的订单列表中，并更新路径距离和无人机当前位置；否则，结束本次出发选择订单的过程。

- 回到配送中心
  - 无人机完成本次出发的所有订单后，回到配送中心，并更新本次出发的总路径距离。
  - 记录本次出发的路径：将本次出发的订单列表和路径距离添加到路径列表中。
  - 重置当前位置：将当前无人机的位置重置为配送中心，以便为下一次出发做准备。
- 返回结果
  - 返回路径和总距离：最后，算法返回包含每次出发的订单列表和路径距离的路径列表，以及无人机飞行的总距离。

下面结合代码进行说明：

1. 参数和数据结构

```
# 配送中心的位置
center: 配送中心的坐标（例如 (x, y)）
# 订单列表
orders: 每个订单包含卸货点ID和优先级（例如 [(dock_id, priority), ...]）
# 优先级字典
priority_levels: 一个字典，键是优先级名称（例如 '一般', '较紧急', '紧急'），值是处理这些订单所需的时间（小时）
# 最大携带物品数量
max_cargo: 无人机最大携带物品数量
# 无人机最大飞行距离
max_distance: 无人机最大飞行距离（公里）
# 配送中心和卸货点的坐标列表
delivery_docks: 卸货点的坐标列表（例如 [(x, y), ...]）
```

2. 路径规划函数 `plan_path`

--按优先级排序订单

```
orders = sorted(orders, key=lambda x: priority_levels[x[1]])  # 按优先级排序
```

根据优先级对订单进行排序，优先级高的订单排在前面。使用优先级字典`priority_levels`中的值作为排序的依据。

3. 初始化变量

```
total_distance = 0
current_location = center
path = []
```

- `total_distance`：累计的总距离，初始化为0。
- `current_location`：当前无人机的位置，初始化为配送中心的位置。

- `path`：存储每次出发执行的订单和对应的路径距离。

4. 处理订单

--初始化本次出发的变量

```
while orders:
trip_orders = [] # 本次出发执行的订单列表
trip_distance = 0 # 本次出发的路径距离
current_cargo = 0 # 本次出发携带的货物数量
```

--添加订单到本次出发中

```
while orders and current_cargo < max_cargo:
    next_order = orders[0]
    next_location = delivery_docks[next_order[0]]
    dist_to_next = distance(current_location, next_location)
    dist_back_to_center = distance(next_location, center)
    round_trip_dist = trip_distance + dist_to_next + dist_back_to_center
    if round_trip_dist ≤ max_distance:
        trip_orders.append(next_order)
        trip_distance += dist_to_next
        current_location = next_location
        current_cargo += 1
        orders.pop(0)
    else:
        break
```

在还有未处理的订单并且当前无人机携带的货物数量小于最大携带数量时：

- 获取下一个订单和卸货点的位置。
- 计算当前地点到下一个卸货点的距离以及从下一个卸货点回到配送中心的距离。
- 计算本次出发的总距离，如果总距离不超过无人机的最大飞行距离，则将订单添加到本次出发的订单列表中，并更新距离和位置；否则，结束本次出发。

--回到配送中心

```
trip_distance += distance(current_location, center)  # 回到配送中心
total_distance += trip_distance
path.append((trip_orders, trip_distance))
current_location = center
```

- 无人机回到配送中心，更新本次出发的总距离。
- 将本次出发的订单和路径距离添加到 `path` 列表中。
- 重置当前位置为配送中心。

这个贪心算法通过按优先级排序订单，并在每次出发时尽可能多地携带订单来最大化无人机的使用效率，同时确保总飞行距离不超过无人机的最大飞行距离。该算法适用于在有约束条件的情况下（如飞行距离和携带货物数量）规划高效的无人机配送路径。

决策系统

这个 `decision_system` 函数旨在将生成的订单分配给各个配送中心，并使用贪心算法为每个配送中心规划最佳路径。以下是该函数的详细解释：

1. 初始化

```
total_distance = 0
center_orders = defaultdict(list)  # 每个配送中心分配的订单
```

- **total_distance**: 用于记录所有配送中心的累计配送距离。
- **center_orders**: 使用 `defaultdict` 创建一个字典，以存储每个配送中心分配到的订单列表。

2. 分配订单到最近的配送中心

```
for order in orders:
    # 找到距离该订单最近的配送中心
    closest_center = min(distribution_centers, key=lambda center:
distance(center, delivery_docks[order[0]]))
    center_orders[closest_center].append(order)
```

- 遍历所有订单，对于每个订单，通过计算订单的卸货点到所有配送中心的距离，找到最近的配送中心，并将订单分配给该配送中心。

3. 为每个配送中心规划路径

```
all_paths = defaultdict(list)  # 用于存储每个配送中心的路径
for center, orders in center_orders.items():
    path, dist = plan_path(center, orders)
    total_distance += dist
    all_paths[center].extend(path)
```

- 遍历每个配送中心及其分配的订单，调用 `plan_path` 函数为每个配送中心规划路径。
- **plan_path(center, orders)**: 这是之前定义的贪心算法函数，返回该配送中心的路径和总距离。
- 将该配送中心的配送距离累加到 `total_distance` 中，并将路径信息添加到 `all_paths` 字典中。

该 `decision_system` 函数的核心思想是先将所有订单分配给距离最近的配送中心，然后利用贪心算法为每个配送中心规划最佳路径。通过这种方式，可以确保订单尽量分配给最近的配送中心，从而减少配送距离，并在满足无人机飞行距离和载货量约束的前提下，尽可能多地完成配送任务。

决策优化

优化方案一：

上述方案并没有考虑到“系统做决策时，可以不对当前的某些订单进行配送，因为当前某些订单可能紧急程度不高，可以累积后和后面的订单一起配送”。由于本题的条件中没有限制无人机的数量，所以我们可以让某时刻订单足够多，这样路径规划时会产生更多的满载无人机出发，从而优化最短路径。

采用最简单的订单累计方案：根据订单的紧急程度进行优先级配送，同时累积优先级较低的订单以优化配送路径和效率。订单的累积和执行是根据其优先级和剩余时间来决定的。

- 订单累积条件：
  - 一个订单会被累积的条件是：订单的优先级别要求的配送时间（以分钟为单位）减去当前时间间隔已经过去的时间大于一个时间间隔（即 `time_interval`）。
  - 例如，如果一个订单的优先级是“一般”（3小时内配送），那么它有 180 分钟的配送时间。如果当前时间是第一个时间间隔（0分钟），那么此订单还剩 180 - 0 = 180 分钟；如果当前时间是第二个时间间隔（10分钟），此订单还剩 180 - 10 = 170 分钟。只要剩余时间大于 `time_interval`，这个订单就可以被累积。
- 订单执行条件：
  - 一个订单会被立即执行的条件是：订单的优先级别要求的配送时间（以分钟为单位）减去当前时间间隔已经过去的时间小于或等于一个时间间隔（即 `time_interval`）。
  - 例如，如果一个订单的优先级是“紧急”（0.5小时内配送），那么它有 30 分钟的配送时间。如果当前时间是第一个时间间隔（0分钟），那么此订单还剩 30 - 0 = 30 分钟；如果当前时间是第二个时间间隔（10分钟），此订单还剩 30 - 10 = 20 分钟。只要剩余时间小于或等于 `time_interval`，这个订单就需要立即执行。

对应的决策修改如下：

```
def decision_system(all_orders, current_time):
    total_distance = 0
    center_orders = defaultdict(list)  # 每个配送中心分配的订单
    pending_orders = []  # 累积的未配送订单
    for order in all_orders:
        dock_id, priority = order
        time_left = priority_levels[priority] * 60 - current_time
        if time_left > time_interval:  # 优先级较低，可以累积
            pending_orders.append(order)
        else:
            closest_center = min(distribution_centers, key=lambda center:
```

```
distance(center, delivery_docks[dock_id]))
            center_orders[closest_center].append(order)

    all_paths = defaultdict(list)  # 用于存储每个配送中心的路径
    for center, orders in center_orders.items():
        path, dist = plan_path(center, orders)
        total_distance += dist
        all_paths[center].extend(path)
    return total_distance, all_paths, pending_orders
```

优化方案二：

存在一种情况，某个卸货点只有一单要送，但需要单独出发一台无人机，这种情况下如果该订单可以累积到下一时间段，那么我们可以等该卸货点的其它订单一起运输。

为了避免无人机为单个订单单独出发的情况，我们可以引入一个延迟处理机制，即对于某些优先级较低的订单，如果当前的订单量不足以充分利用无人机的运力，则可以将这些订单累积到下一时间段。具体的做法是，对于每个配送中心，将优先级较低且数量不足的订单暂时不进行配送，而是累积到下一时间段，直到订单量达到一定数量或者某些订单的优先级提高到需要立即处理的程度。

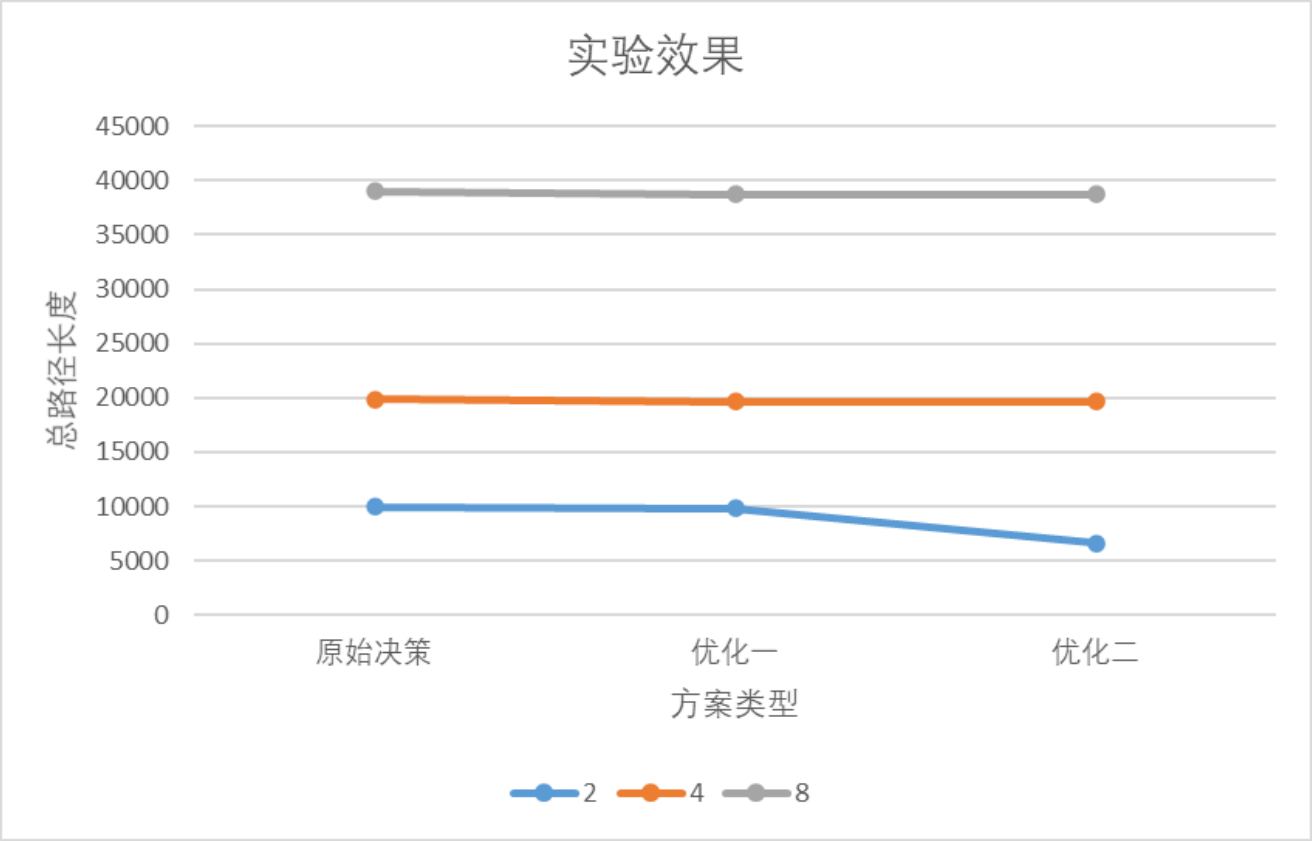
对应的决策修改如下：

```
def decision_system(orders, accumulated_orders):
    total_distance = 0
    center_orders = defaultdict(list)  # 每个配送中心分配的订单
    for order in orders:
        # 找到距离该订单最近的配送中心
        closest_center = min(distribution_centers, key=lambda center:
distance(center, delivery_docks[order[0]]))
        center_orders[closest_center].append(order)
    # 合并累积的订单
    for center in center_orders.keys():
        center_orders[center].extend(accumulated_orders[center])
        accumulated_orders[center] = []
    all_paths = defaultdict(list)  # 用于存储每个配送中心的路径
    for center, orders in center_orders.items():
        # 先处理紧急订单
        urgent_orders = [order for order in orders if
priority_levels[order[1]] ≤ 1.5]
        non_urgent_orders = [order for order in orders if
priority_levels[order[1]] > 1.5]
        # 路径规划
        if urgent_orders:
            path, dist = plan_path(center, urgent_orders)
            total_distance += dist
            all_paths[center].extend(path)
```

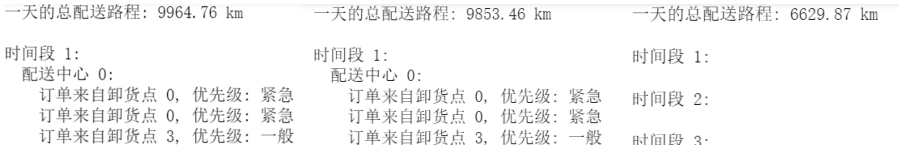
```
            # 累积非紧急订单
            accumulated_orders[center].extend(non_urgent_orders)
            # 如果累积的订单达到一定数量或者某些订单的优先级提高到需要立即处理的程度
            if len(accumulated_orders[center]) ≥ max_cargo or
    any(priority_levels[order[1]] ≤ 1.5 for order in
    accumulated_orders[center]):
                path, dist = plan_path(center, accumulated_orders[center])
                total_distance += dist
                all_paths[center].extend(path)
                accumulated_orders[center] = []  # 清空累积的订单
        return total_distance, center_orders, all_paths, accumulated_orders
```

实验结果

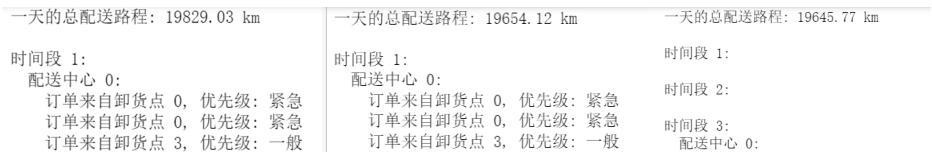
我们假定一个订单生成间隔（time_interval）为10分钟，不同方案与模拟时长都采用相同的地图与相同的各个时间段订单。



在2小时的模拟时长下：



在4小时的模拟时长下：



在8小时的模拟时长下：

```
一天的总配送路程：39049.97 km

时间段 1：
  配送中心 0：
    订单来自卸货点 0，优先级：紧急
    订单来自卸货点 0，优先级：紧急
    订单来自卸货点 3，优先级：一般
```

```
一天的总配送路程：38729.81 km

时间段 1：
  配送中心 0：
    订单来自卸货点 0，优先级：紧急
    订单来自卸货点 0，优先级：紧急
    订单来自卸货点 3，优先级：一般
```

```
一天的总配送路程：38721.46 km

时间段 1：

时间段 2：
```

结果分析

- 原始决策
  原始决策是不考虑订单可以累计的情况，每个时间间隔都立即处理所有订单。这种方式下，无人机可能会为了送一个订单而单独出发，导致总配送距离较长。
- 优化一
  优化一的策略是根据订单的优先级和剩余时间来决定是否将订单累计。高优先级订单会优先处理，低优先级订单会等待下一时间段。这种方法相比原始决策稍有改进，在2小时、4小时和8小时模拟时长下的总配送距离分别减少了1.1%、0.9%和0.8%。
- 优化二
  优化二采用延迟处理机制，对于某些优先级较低且数量不足的订单，选择将其累积到下一时间段，直到订单量达到一定数量或某些订单的优先级提高到需要立即处理的程度。这种策略显著减少了无人机为了单个订单单独出发的情况，在2小时、4小时和8小时模拟时长下的总配送距离分别减少了33.4%、0.1%和0.1%。

1. **效率提升显著**：优化二在短时间（2小时）内显著减少了总配送距离，表明延迟处理机制在短时间模拟下非常有效。
2. **逐步优化**：在较长时间（4小时和8小时）内，优化二和优化一的效果相似，但仍略优于原始决策。这说明在更长时间内订单的累积效果逐渐显现。
3. **优化一与优化二的差异**：优化二比优化一更加智能和灵活地处理订单，特别是在短时间内效果更为显著。优化二在模拟时长为2小时时效果最好，但在较长时间内效果与优化一接近，说明随着时间的增加，订单量的积累使得优化策略差异减少。

根据实验结果，可以考虑在实际应用中采用优化二的延迟处理机制，特别是在短时间配送任务中，以显著减少总配送距离，提高配送效率。在较长时间的配送任务中，虽然优化一和优化二的效果相似，但优化二仍然具备一定优势，可以更好地处理波动的订单需求。