武漢大學

《高级算法设计与分析》实验报告

研 究 生 姓 名 ：赵思敏

学 号 ：2023202210051

指 导 教 师 姓 名 ：林海

专 业 名 称 ：网络空间安全

二〇二四年六月

目录

第 1 章　问题分析

无人机可以快速解决最后 10 公里的配送，本次实验要求设计一个算法，实现区域内无人机配送的路径规划。

1.1　配送中心及其职能

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。

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。

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。

1.2　卸货点及其职能

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），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。

我们将时间离散化，也就是每隔 t 分钟，所有的卸货点会生成 0~m 个订单。

1.3　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；

1.4 算法目标

算法的目标是在满足订单的优先级别要求的约束条件下，一段时间内（如一天）所有无人机的总配送路径最短。

也就是通过算法我们可以得到每个 t 时间间隔的无人机调度路线以及配送路径长度，从而得到一段时间内所有无人机的最短总配送路径。

第 2 章　问题背景

本实验需要解决的问题可以视为一种特殊的车辆路径问题（Vehicle Routing Problem, VRP），这一章我们从 VRP 问题出发，介绍 VRP 问题及其演化问题等背景知识，以帮助我们更好地理解本次实验需要解决的无人机路径规划问题。

2.1　VRP 问题及其演化

车辆路径问题（Vehicle Routing Problem，VRP）是运筹学和组合优化领域的经典问题，自 1959 年由 Dantzig 和 Ramser 首次提出以来，已成为物流和配送领域的重要研究课题。VRP 的基本形式可以描述为：在一个配送中心和若干客户需求点的情况下，规划多辆车辆的配送路线，使得所有客户的需求都能被满足，且总行驶距离最短。

在企业联盟和拥有多个车场的大规模物流运输企业逐渐兴起的社会背景下，VRP 有多个变种，以适应不同的实际应用场景和复杂约束条件，常见的变种包括：

1. CVRP（Capacitated Vehicle Routing Problem）：引入了车辆的载重量限制，需要在满足客户需求的同时，不超过车辆的最大载重。
2. VRPTW（Vehicle Routing Problem with Time Windows）：增加了时间窗口的约束，即每个客户需求点必须在规定的时间段内被访问。
3. MDVRP（Multi-Depot Vehicle Routing Problem）：考虑了多个配送中心，每个中心都可以提供配送服务，需要优化多个中心的整体配送路径。
4. MDVRPTW（Multi-Depot Vehicle Routing Problem with Time Windows）：结合了多个配送中心和时间窗口的约束。

2.2　MDVRPTW 问题描述与数学建模

MDVRPTW 的问题定义如下：

- 多个配送中心：区域内有多个配送中心，每个中心都有独立的库存和车辆队伍。
- 客户需求：客户需求点具有各自的需求量，车辆需要从某个配送中心出发，满足客户的需求，然后返回到同一个配送中心。
- 时间窗口：每个客户需求点都有指定的时间窗口，车辆必须在该时间窗口内完成配送。

- 目标：最小化所有车辆的总行驶距离，同时满足所有客户的需求和时间窗口约束。

MDVRPTW 在实际应用中具有广泛的应用场景，如城市物流配送、快递服务和供应链管理等。

假设用户编号为 1，2，…，N，车场编号为 N+1，N+2，…，N+M，定义变量如下：

$$x_{ij}^{mk}=\begin{cases}1 & \text{车场 } m \text{ 的车辆 } k \text{ 从用户 } i \text{ 行驶到用户 } j\\ 0 & \text{否则}\end{cases} \tag{1}$$

d_{ij}表示从用户 i 到用户 j 的运输成本，它的含义可以是距离、费用、时间等，s_i表示车辆到达用户 i 的时间，p_E表示在ET_i之前到达用户 i 等待的单位时间成本，p_L表示在LT_i之后到达用户 i 的单位时间成本。若车辆在ET_i之前到达用户 i，则增加机会成本$p_E \times (s_i - ET_i)$；若车辆在ET_i之后到达用户 i，则增加罚金成本$p_L \times (LT_i - s_i)$。

可用数学模型表示如下：

$$minZ = min \sum_{i=1}^{N+MN+M} \sum_{j=1}^{N} \sum_{m=1}^{M} \sum_{k=1}^{K_m} d_{ij} x_{ij}^{mk} + p_E \sum_{i=1}^{N} max(ET_i - s_i, 0) + p_L \sum_{i=1}^{N} max(s_i - LT_i, 0) \tag{2}$$

s.t.

$$\sum_{j=1}^{V} \sum_{k=1}^{K_m} x_{ij}^{mk} \leqslant k_m, i = m \in \{N+1, N+2, \cdots, N+M\} \tag{3}$$

$$\sum_{j=1}^{N} x_{ij}^{mk} = \sum_{j=1}^{N} x_{ij}^{mk} \leqslant 1, i = m \in \{N+1, N+2, \cdots, N+M\}, k \in \{1,2,\cdots,K_m\} \tag{4}$$

$$\sum_{j=1}^{N+M} \sum_{m=1}^{M} \sum_{k=1}^{K_m} x_{ij}^{mk} = 1, i \in \{1,2,\cdots,N\} \tag{5}$$

$$\sum_{i=1}^{N+M} \sum_{m=1}^{M} \sum_{k=1}^{K_m} x_{ij}^{mk} = 1, j \in \{1,2,\cdots,N\} \tag{6}$$

$$\sum_{i=1}^{N} g_i \sum_{j=1}^{N+M} x_{ij}^{mk} \leqslant q, m \in \{N+1, N+2, \cdots, N+M\}, k \in \{1,2,\cdots,K_m\} \tag{7}$$

$$\sum_{j=N+1}^{N+M} x_{ij}^{mk} = \sum_{j=N+1}^{N+M} x_{ji}^{mk} = 0, i = m \in \{N+1, N+2, \cdots, N+M\}, k \in \{1,2,\cdots,K_m\} \tag{8}$$

式(2)表示目标函数，即最小成本；式(3)表示各车场派出的车辆数不能超过该车场的车辆数；式(4)确保车辆都是从各自的车场出发，并回到原车场；式(5)、(6)保证每个用户只能被一辆车服务一次；式(7)表示车辆承载的货物量之和不能大于车辆的容量；式(8)表示车辆不能从车场到车场。

2.3 MDVRPTW 问题研究现状

为解决 MDVRPTW 问题，研究者们提出了许多启发式算法和元启发式算法。这些算法通过近似最优解或优化路径来提高求解效率。以下是几种常用的求解方法：

（1）启发式算法（Heuristic Algorithms）：

- 最近邻算法（Nearest Neighbor Algorithm）：通过从当前节点选择最近的未访问节点来构建路径，简单易行，但易陷入局部最优。
- 插入法（Insertion Methods）：逐步将客户点插入到现有路径中，通过最小化增加的路径成本来决定插入位置。

（2）元启发式算法（Metaheuristic Algorithms）：

- 遗传算法（Genetic Algorithm, GA）：模拟自然选择和遗传变异，通过选择、交叉和变异操作生成优化解，对初始解的质量依赖较大。
- 禁忌搜索（Tabu Search, TS）：利用禁忌表避免搜索过程中的循环，通过邻域搜索找到最优解，适用于多种复杂优化问题。
- 模拟退火（Simulated Annealing, SA）：模拟物理退火过程，通过接受劣解的概率逐步降低来跳出局部最优，适用于大规模问题。
- 蚁群算法（Ant Colony Optimization, ACO）：模拟蚂蚁觅食行为，通过信息素引导搜索路径，适用于离散优化问题。
- 粒子群优化（Particle Swarm Optimization, PSO）：模拟鸟群觅食行为，通过个体间的信息共享和全局搜索找到最优解。

（3）混合算法（Hybrid Algorithms）：

- 结合多种算法的优点，设计出混合优化策略。例如，将遗传算法与局部搜索算法相结合，提高解的质量和收敛速度。
- 启发式与元启发式结合：如先用最近邻算法构造初始解，再通过禁忌搜索或模拟退火进行优化。

（4）精确算法（Exact Algorithms）：

对于小规模问题，精确算法如分支定界法（Branch and Bound, B&B）和动态规划（Dynamic Programming, DP）可以找到最优解，但计算复杂度较高，不适用于大规模问题。

第 3 章　解决方案

本实验研究的无人机配送路径规划问题是 MDVRPTW 问题的具体应用变种，针对无人机在城市配送中的应用场景，考虑了以下具体条件：

- 配送中心数量（j）：区域内有多个配送中心，每个配送中心都能提供充足的商品和无人机，且配送中心的无人机数量无限。
- 卸货点数量（k）：区域内有多个卸货点，每个卸货点会随机生成 0~m 个订单，每个订单包含一个商品。
- 订单优先级：订单分为三类优先级：一般、较紧急和紧急，分别要求在 3 小时、1.5 小时和 0.5 小时内完成配送。
- 时间离散化：将时间离散化为固定时间段（t 分钟），每隔 t 分钟，系统会生成新的订单，并进行配送决策。
- 无人机限制：每次配送的无人机有最大载物量和最大飞行距离限制，并且需要在规定时间内完成订单。

本次实验的主要目标是设计高效的调度和路径优化算法，在满足所有订单优先级要求的前提下，使总配送路径最短。具体而言，面临以下挑战：

（1）动态订单生成：每隔 t 分钟生成的新订单，系统需要实时进行调度决策，动态调整配送计划。

（2）多中心协调：需要在多个配送中心之间协调无人机的调度，确保整体配送路径的最优化。

（3）优先级处理：不同订单有不同的优先级，系统需要根据优先级合理安排配送顺序，保证紧急订单优先完成。

（4）路径优化：无人机路径规划需要考虑载物量和飞行距离的限制，规划最优路径以最小化总行驶距离。

3.1　动态订单生成

每隔 t 分钟生成新订单，模拟实际配送过程中订单动态产生的场景。每次生成的订单数量在 0 到 m 之间随机变化，确保订单数量的随机性和多样性。

每个订单主要包括的属性有：订单 ID、生成时间、卸货点 ID、优先级（紧急、较紧急、一般）。其中订单 ID 为全局唯一标识符，生成时间用于计算订单的配送时限，优先级决定订单的配送优先级别。

对于订单优先级管理，我使用优先级队列（小顶堆）管理待处理订单，根据优先级和生成时间进行排序。系统在每个时间段从优先级队列中提取订单进行调度决策。

3.2 多中心协调

对于每个新生成的订单，根据订单的地理位置和紧急程度，将订单分配到最适合的配送中心。使用距离最近原则，将订单分配到距离最近的配送中心，确保配送路径的总距离最小化。

每个配送中心独立进行路径规划和优化，确保在本中心内实现最优配送路径。使用改进的最近邻算法生成初始路径，再使用遗传算法进行优化。

3.3 优先级处理

订单分为三种优先级：紧急（0.5 小时内配送）、较紧急（1.5 小时内配送）、一般（3 小时内配送）。系统根据订单的优先级进行分类，分别加入对应的优先级队列中。

在每个决策时刻，系统优先处理紧急订单，再处理较紧急订单，最后处理一般订单。在调度过程中，确保紧急订单优先完成，避免超时。

3.4 路径优化

使用改进的最近邻算法进行初始路径规划：从配送中心出发，选择最近的未配送订单进行配送，直至无人机达到最大容量或飞行距离限制。

使用遗传算法对初始路径进行优化：

- 生成初始种群：对初始路径进行随机打乱，生成一定数量的初始解。
- 评估适应度：根据总行驶距离和订单的配送时限，评估每个路径的适应度。
- 选择、交叉、变异：通过选择适应度高的路径进行交叉、变异，生成新的路径。
- 多代进化：通过多代进化，逐步优化路径，最终选择最优路径。

通过上述实验方案，可以动态生成订单、协调多个配送中心、处理订单优先级，并优化无人机的配送路径，实现最小化总行驶距离的目标。

第 4 章　具体实施过程

4.1　参数设置与初始配置

我们设置了实验所需的参数。这些参数包括配送中心数量、卸货点数量、决策时间段、无人机最大容量、订单生成最大数量、最大飞行距离、无人机速度和订单优先级限制等。此外，我们还定义了地图的最大坐标和全局订单 ID，并生成配送中心和卸货点位置，从而模拟真实的无人机配送场景。

```
# 参数设置
j = 3  # 配送中心数量
k = 20  # 卸货点数量
t = 10  # 决策时间段（分钟）
max_capacity = 5  # 无人机最大容量
m = 3  # 每次生成订单的最大数量
max_distance = 20  # 公里
drone_speed = 60 / 60  # 公里/分钟
priority_limits = {'urgent': 0.5 * 60, 'semi-urgent': 1.5 * 60, 'normal': 3 * 60}  # 分钟
max_coord = 10  # 地图最大坐标
order_id = 0  # 全局订单 ID
# 配送中心和卸货点坐标
distribution_centers = [(random.uniform(0, max_coord), random.uniform(0, max_coord)) for _ in range(j)]
delivery_points = [(random.uniform(0, max_coord), random.uniform(0, max_coord)) for _ in range(k)]
```

4.2　订单生成

在每个时间段 t 分钟生成新的订单，随机分配到各个卸货点，并为每个订单分配一个优先级。这一部分实现了动态订单生成的功能，模拟了实际中订单的随机到达。

```
def generate_orders(current_time):
    global order_id
    orders = []
    for point in range(k):
        for _ in range(random.randint(0, m)):
            order = {
                'id': order_id,
                'time': current_time,
                'point': point,
                'priority': random.choice(['urgent', 'semi-urgent', 'normal']),
            }
            orders.append(order)
            order_id += 1
    return orders
```

4.3 最近邻算法

使用改进的最近邻算法初步生成配送路径，确保在无人机容量和飞行距离限制内，选择最短路径进行配送。

最近邻算法的基本思想是从配送中心出发，每次选择最近的未访问过的卸货点，直到达到无人机的容量或飞行距离限制。下面将详细介绍算法的实现步骤及其功能。

（1）函数定义与初始化

```
def nearest_neighbor_algorithm(center, orders):
    routes = []
    ……
    return routes
```

函数 nearest_neighbor_algorithm 接收两个参数：配送中心的坐标 center 和订单列表 orders。初始化一个空列表 routes，用于存储生成的所有配送路径。

主循环结束时，返回生成的所有路径 routes。

（2）主循环

```
    while orders:
        route = []
        current_point = center
        current_distance = 0
        current_time = 0
        current_capacity = 0
```

在主循环中，首先检查是否还有未处理的订单。然后，初始化一个空列表 route 来存储当前路径上的订单。初始化 current_point 为配送中心的坐标，current_distance 为 0，current_time 为 0，current_capacity 为 0。这些变量分别表示当前点的坐标、当前路径的总距离、当前时间和当前无人机的载物量。

（3）路径生成内循环

```
        while orders and current_capacity < max_capacity and current_distance < max_distance / 2:
            next_order = None
            min_distance = float('inf')
            for order in orders:
                distance = calculate_distance(current_point, delivery_points[order['point']])
                delivery_time = current_time + distance / drone_speed

                if (current_capacity + 1) <= max_capacity and (current_distance + distance) <=
max_distance / 2:
                    if distance < min_distance and delivery_time <= priority_limits[order['priority']]:
                        next_order = order
```

```
                    min_distance = distance
            if next_order is None:
                break
            route.append(next_order)
            orders.remove(next_order)
            current_distance += min_distance
            current_time += min_distance / drone_speed
            current_capacity += 1
            current_point = delivery_points[next_order['point']]
        routes.append(route)
```

在路径生成的内循环中，检查是否还有未处理的订单，并确保无人机未达到容量和飞行距离的限制。初始化 next_order 为 None，min_distance 为无穷大，这些变量分别表示下一个订单和最小距离。

在 for 循环中，遍历所有未处理的订单，计算当前点到订单卸货点的距离 distance 和到达时间 delivery_time。检查无人机是否能够在容量和距离限制内接收该订单，并且该订单的到达时间是否在其优先级时间限制内。如果符合条件且距离小于 min_distance，则更新 next_order 和 min_distance。如果找到了 next_order，将其添加到当前路径 route 中，并从订单列表中移除。然后更新当前路径的总距离、当前时间、无人机载物量和当前点坐标。

内循环结束时，将当前路径 route 添加到所有路径列表 routes 中。

4.4 遗传算法优化路径

使用遗传算法对初始路径进行多代优化，通过选择、交叉和变异操作，逐步提高路径的优化程度，确保总行驶距离最小化。

（1）初始化参数

```
def genetic_algorithm(initial_routes):
    population_size = 50
    generations = 100
    mutation_rate = 0.1
```

定义遗传算法的参数，包括种群大小、迭代次数和变异率。population_size 是每一代包含的路径个数，generations 是遗传算法运行的迭代次数，mutation_rate 是变异率，即路径变异的概率。

（2）个体表示与适应度函数

每个个体表示一条路径（由订单组成的序列），适应度函数用于评估每条路径的优劣。

```
    def calculate_delivery_time(current_position, order):
        distance = calculate_distance(current_position, delivery_points[order['point']])
        travel_time = distance / drone_speed
        delivery_time = current_time + travel_time
        if order['priority'] == 'normal':
            time_limit = 180
        elif order['priority'] == 'semi-urgent':
            time_limit = 90
        elif order['priority'] == 'urgent':
            time_limit = 30
        return delivery_time
```

calculate_delivery_time 函数计算从当前位置到某个订单点的送货时间，并返回预计的送货时间。

```
    def fitness(route):
        total_distance = 0
        penalty = 0
        for i in range(len(route) - 1):
            total_distance += calculate_distance(delivery_points[route[i]['point']],
delivery_points[route[i + 1]['point']])
        for order in route:
            delivery_time = calculate_delivery_time(delivery_points[0], order)
            if delivery_time > priority_limits[order['priority']]:
                penalty += 1000
        fitness_value = total_distance - penalty
        return fitness_value
```

fitness 函数计算路径的适应度值，适应度值越高表示路径越优。适应度值由总距离和惩罚项组成，如果订单未在规定时间内送达，则增加惩罚。

（3）初始化种群

```
    def initial_population():
        population = []
        for _ in range(population_size):
            route = initial_routes[:]
            random.shuffle(route)
            population.append(route)
        return population
```

initial_population 函数用于生成初始种群，通过随机打乱初始路径生成多个不同的个体。

（4）选择

选择操作用于从种群中选择适应度较高的个体作为父代，以生成下一代。

```
    def select_parents(population):
        fitnesses = [fitness(route) for route in population]
        total_fitness = sum(fitnesses)
        if total_fitness == 0:  # 添加检查以避免除以零
            return population[0], population[1]  # 或者选择其他适当的处理方式
        probabilities = [fit / total_fitness for fit in fitnesses]
        parents = []
        for _ in range(2):
            r = random.random()
            cumulative_probability = 0
            for i, probability in enumerate(probabilities):
                cumulative_probability += probability
                if r < cumulative_probability:
                    parents.append(population[i])
                    break
        return parents
```

select_parents 函数通过轮盘赌选择方法选择父代个体，选择概率与适应度值成正比。在无人机路径规划问题中，适应度值越高表示路径越短、订单按时送达的可能性越高，因此被选择作为父代的可能性越大。

（5）交叉

交叉操作用于生成新个体（子代），通过组合父代的基因（路径段）生成新的路径。

```
    def crossover(parent1, parent2):
        if len(parent1) < 3 or len(parent2) < 3:
            # 如果任一路线长度小于 3，不进行交叉或以特殊方式处理
            return parent1[:], parent2[:]
        crossover_point1 = random.randint(1, len(parent1) - 2)
        crossover_point2 = random.randint(1, len(parent2) - 2)
        # 确保交叉点不同
        if crossover_point1 == crossover_point2:
            crossover_point2 = (crossover_point2 + 1) % len(parent2)
        child1 = parent1[:crossover_point1] + [gene for gene in parent2 if  gene not in
parent1[:crossover_point1]]
        child2 = parent2[:crossover_point2] + [gene for gene in parent1 if  gene not in
parent2[:crossover_point2]]
        return child1, child2
```

crossover 函数实现交叉操作，通过在父代路径中选择交叉点生成两个子代路径。在无人机路径规划问题中，交叉操作通过组合父代的路径段生成新的路径，可能会发现更优的路径组合。

（6）变异

变异操作用于引入新的基因（路径段），通过随机改变个体的一部分来增加种群的多样性。

```
def mutate(route):
    for i in range(len(route)):
        if random.random() < mutation_rate:
            j = random.randint(0, len(route) - 1)
            route[i], route[j] = route[j], route[i]
    return route
```

mutate 函数实现变异操作，通过交换路径中的随机两个订单实现变异。在无人机路径规划问题中，变异操作通过随机改变路径中的订单顺序来增加路径的多样性，有助于跳出局部最优。

（7）主循环

```
population = initial_population()
for _ in range(generations):
    new_population = []
    while len(new_population) < population_size:
        parent1, parent2 = select_parents(population)
        child1, child2 = crossover(parent1, parent2)
        child1 = mutate(child1)
        child2 = mutate(child2)
        new_population.extend([child1, child2])
    population = sorted(new_population, key=lambda route: fitness(route))[:population_size]
best_route = min(population, key=lambda route: fitness(route))
return best_route
```

initial_population 生成初始种群。在每一代中，使用 select_parents 选择父代，使用 crossover 生成子代，并使用 mutate 进行变异。new_population 生成新的种群，并根据适应度值排序，选择最优的个体保留到下一代。最终，返回适应度值最高的路径作为优化后的最佳路径。

4.5 无人机任务分配

根据优化后的路径分配无人机执行配送任务，确保在无人机容量和时间限制内完成订单配送。函数 assign_drone(center, route) 主要实现了根据给定的配送路径 route，在指

定的配送中心 center 调度无人机执行订单配送任务，并根据约束条件（载货量、时间优先级）进行相应的判断和处理。这是整个路径规划系统中重要的一部分，负责实际的订单配送和路径执行管理。

```
def assign_drone(center, route):
    current_point = center
    current_time = 0
    current_capacity = 0
    for order in route:
        distance_to_order = calculate_distance(current_point, delivery_points[order['point']])
        travel_time = distance_to_order / drone_speed
        current_time += travel_time
        if (current_capacity + 1) > max_capacity:
            print(f"Drone capacity exceeded. Unable to deliver order at point {order['point']}")
            break
        if current_time - order['time'] > priority_limits[order['priority']]:
            print(f"Failed to deliver order at point {order['point']} within the required time")
        else:
            print(f"Order at point {order['point']} delivered at time {current_time:.2f} minutes")
        current_point = delivery_points[order['point']]
        current_capacity += 1
    distance_to_center = calculate_distance(current_point, center)
    return_time = distance_to_center / drone_speed
    current_time += return_time
    print(f"Drone returned to center at time {current_time:.2f} minutes")
```

首先初始化变量，current_point 指当前无人机所在位置初始化为配送中心的位置 center。current_time 指当前时间初始化为 0 分钟。current_capacity 指当前无人机载货量初始化为 0。

然后遍历订单路线，对于路径 route 中的每个订单 order：计算当前位置 current_point 到订单所在位置 delivery_points[order['point']] 的距离；根据无人机速度 drone_speed 计算从当前位置飞到订单位置所需的时间 travel_time；更新当前时间 current_time，加上飞行时间 travel_time。

接着进行容量限制检查和优先级限制检查，并更新当前位置和载货量：检查当前无人机的载货量 current_capacity 是否超过最大限制 max_capacity；检查当前时间 current_time 是否超过订单要求的最迟送达时间 priority_limits[order['priority']]；将当前位置更新为已配送完订单的位置 delivery_points[order['point']]，增加当前载货量 current_capacity。

当然也不能忘了无人机返回配送中心的花费：计算当前位置 current_point 到配送中心 center 的距离；根据无人机速度 drone_speed 计算返回时间 return_time；更新当前时间 current_time，加上返回时间 return_time。

最后返回函数执行完成后的当前时间 current_time。

该函数根据优化后的路径逐个分配订单给无人机，检查容量和时间限制，确保合理分配无人机资源。

4.6 实验过程模拟

主循环部分通过时间段循环、订单生成、优先级处理、路径规划优化、任务分配执行和结果输出等步骤，实现了动态的无人机配送路径规划系统。每个步骤都围绕着订单的生成、处理和优化路径的计算展开，确保了系统在多中心协调下以最优的方式完成配送任务，并且在每个时间段内动态调整和优化。

（1）初始化变量

```
orders_dict = {}
orders_heap = []
cumulative_total_distance = 0
for current_time in range(0, 24 * 60, t):
```

orders_dict 用于存储所有生成的订单信息，以订单 ID 为键；orders_heap 使用堆数据结构存储待处理的订单，按照订单优先级和生成时间排序；cumulative_total_distance 用于累积总配送距离，用于统计每个时间段内的总配送距离。

循环每隔 t 分钟（时间段）进行一次决策和调度。

（2）生成新订单

```
    new_orders = generate_orders(current_time)
    for order in new_orders:
        orders_dict[order['id']] = order
        priority=0 if order['priority'] == 'urgent' else 1 if order['priority'] == 'semi-urgent' else 2
        priority=3 - priority
        heapq.heappush(orders_heap, (priority, order['time'], order['id']))
```

调用 generate_orders(current_time) 函数生成在当前时间段 current_time 内的新订单。将新订单加入 orders_dict 字典，并将订单信息推入 orders_heap 堆中，按订单的优先级和生成时间进行堆排序。

（3）处理过期订单

```
    pending_orders = []
    while orders_heap:
```

```
        priority, order_time, order_id = heapq.heappop(orders_heap)
        if current_time - order_time <= priority_limits[orders_dict[order_id]['priority']]:
            order = orders_dict[order_id]
            pending_orders.append(order)
        else:
            print(f"Order at point {order['point']} expired and was removed from the queue")
```

从 orders_heap 中弹出订单，检查订单是否在指定的时间内未过期，如果未过期则添加到 pending_orders 中，否则打印订单过期信息。

（4）订单分配给配送中心

```
    center_orders = defaultdict(list)
    for order in pending_orders:
        closest_center = min(distribution_centers, key=lambda center: calculate_distance(center,
delivery_points[order['point']]))
        center_orders[closest_center].append(order)
```

使用 center_orders 字典，按照每个订单最近的配送中心进行分组。使用 min 函数和 calculate_distance 函数找到每个订单最近的配送中心，并将订单添加到对应配送中心的列表中。

（5）路径规划和调度

```
    center_routes_dict = defaultdict(list)
    current_total_distance = 0
    for center_idx, (center, orders) in enumerate(center_orders.items()):
        initial_routes = nearest_neighbor_algorithm(center, orders)
        optimized_routes = [genetic_algorithm(route) for route in initial_routes]
        center_routes_dict[center] = optimized_routes
        for route in optimized_routes:
            assign_drone(center, route)
            for order in route:
                pending_orders.remove(order)
            current_total_distance += calculate_total_distance(center, route)
```

对每个配送中心的订单集合，首先使用最近邻算法 nearest_neighbor_algorithm 进行初始路径规划。然后使用遗传算法 genetic_algorithm 进行路径优化，得到最优路径集合，并存储在 center_routes_dict 中。对于每个配送中心的每条优化路径 route，调用 assign_drone(center, route)函数进行无人机任务分配和执行，计算当前路径的总配送距离。

（6）累积统计和输出

```
    cumulative_total_distance += current_total_distance
    print(f"Current total distance at time {current_time}: {current_total_distance:.2f} km")
    print(f"Cumulative total distance at time {current_time}: {cumulative_total_distance:.2f} km")
    plot_routes_all_centers(center_routes_dict, current_time)
```

更新累积总配送距离 cumulative_total_distance。打印当前时间段内的总配送距离和累积总配送距离。调用 plot_routes_all_centers 函数绘制无人机路径图并保存。

（7）未分配的订单重新入堆

```
for order in pending_orders:
    priority=0 if order['priority'] == 'urgent' else 1 if order['priority'] == 'semi-urgent' else 2
    if order in pending_orders:
        heapq.heappush(orders_heap, (priority, order['time'], order['id'], order))
    else:
        print(f"Order {order['id']} not found in all_orders. Skipping heappush.")
```

将未处理的订单重新添加到 orders_heap 中，以便下一时间段的处理。

第 5 章　实验结果

5.1　地图随机生成结果

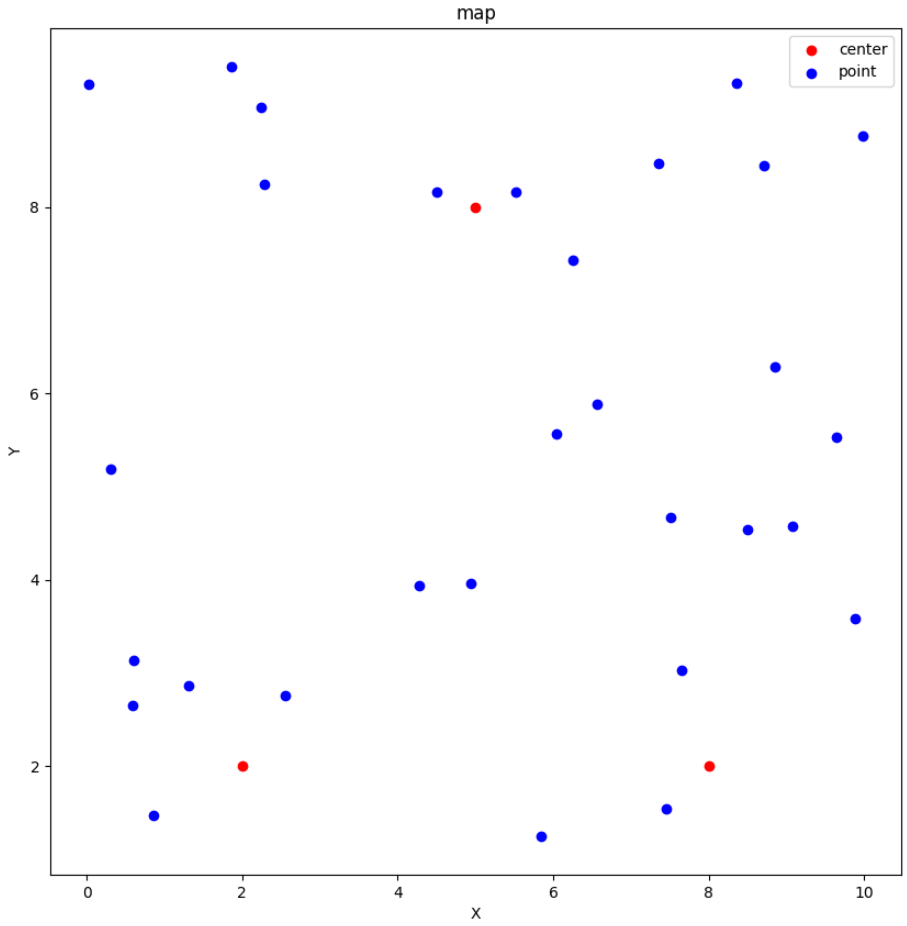

图 1　地图随机生成结果

如图 1，在本次实验中，将地图初始化为 3 个配送中心，30 个卸货点，地图大小为 10×10（公里）

5.2 订单生成示例

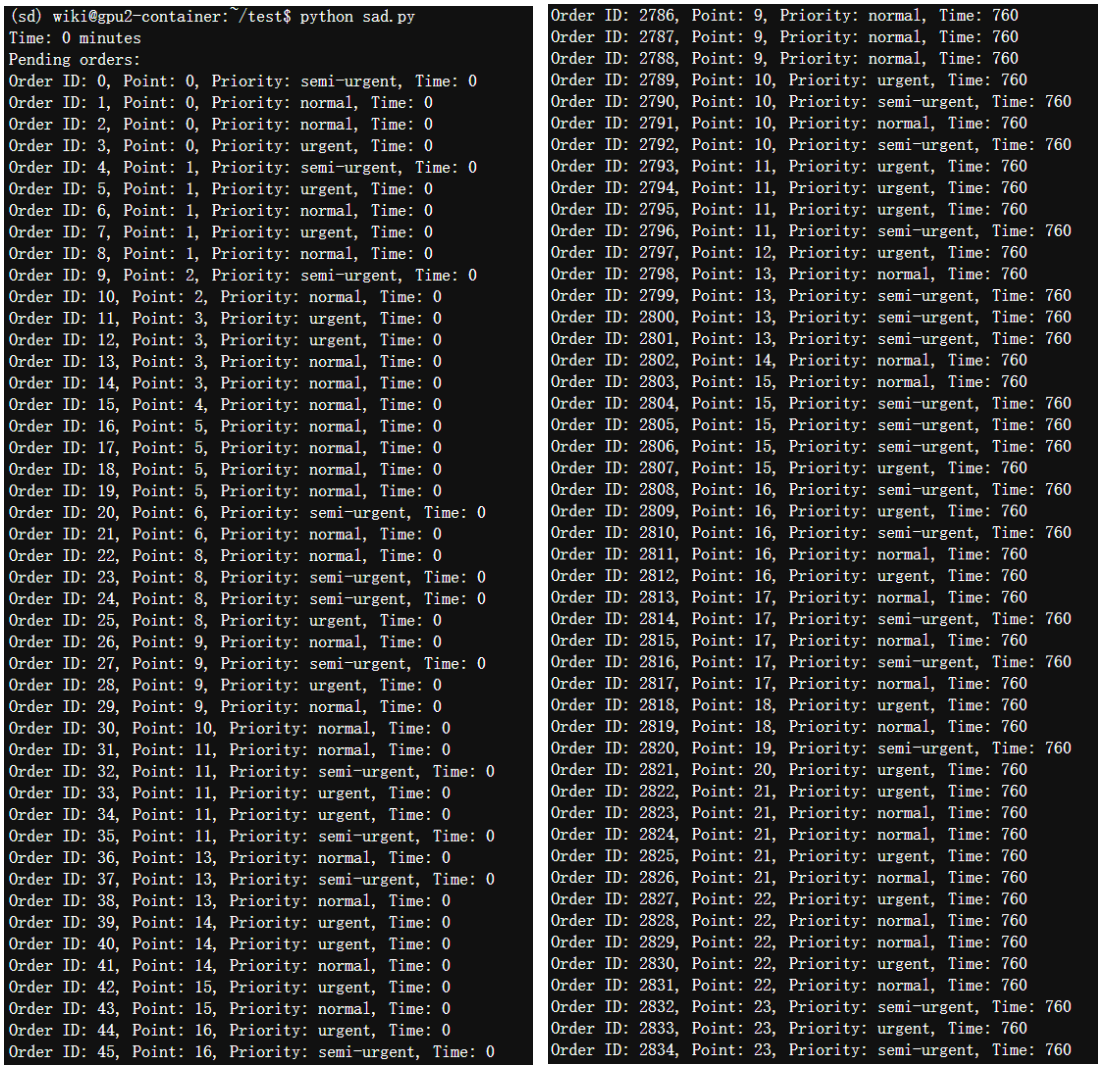

图 2　订单生成结果

如图 2，每隔 t 时间（本实验中设为 20min），每个卸货点随机生成订单，每个订单包含全局订单 ID，卸货点 ID，优先级，时间戳等属性。

5.3　无人机路径可视化

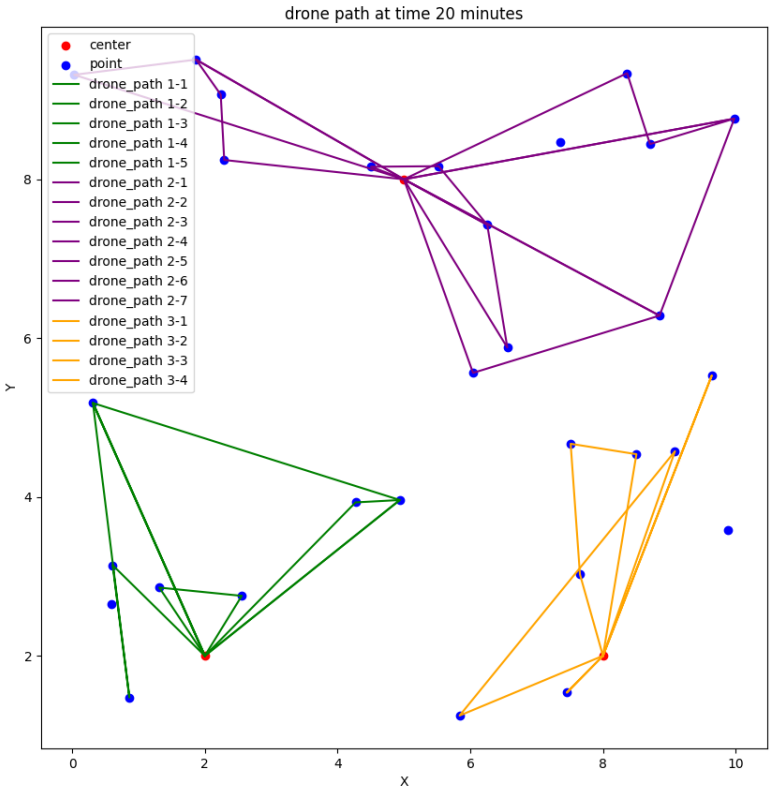

图 3　20 分钟时刻无人机调度路线

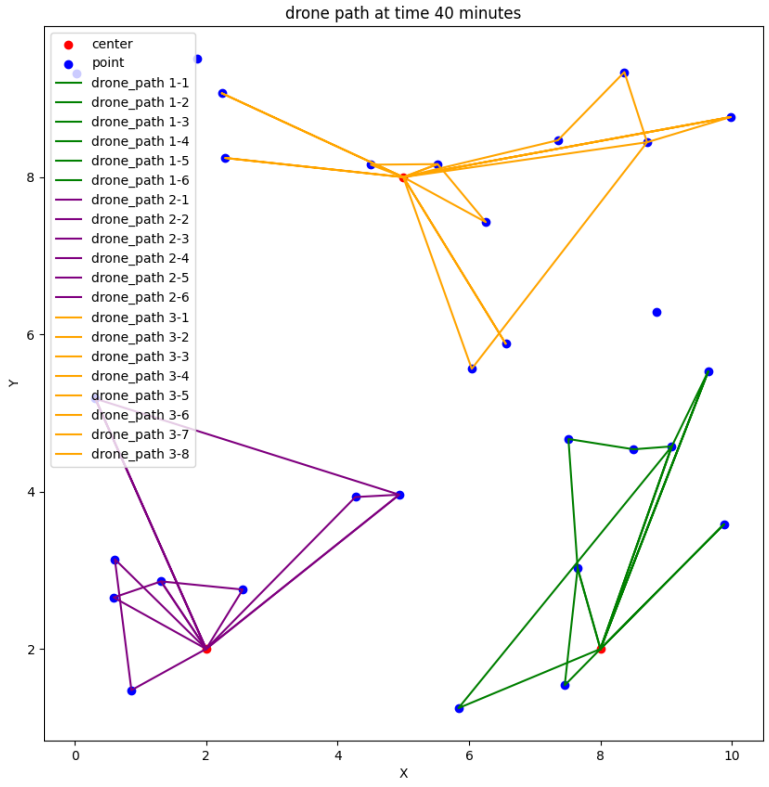

图 4　40 分钟时刻无人机调度路线

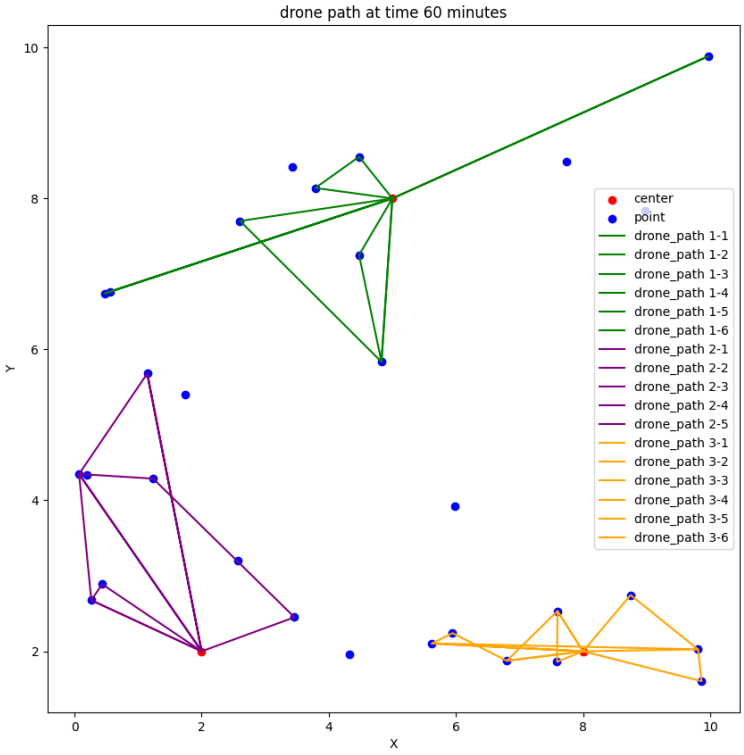

图 5　60 分钟时刻无人机调度路线

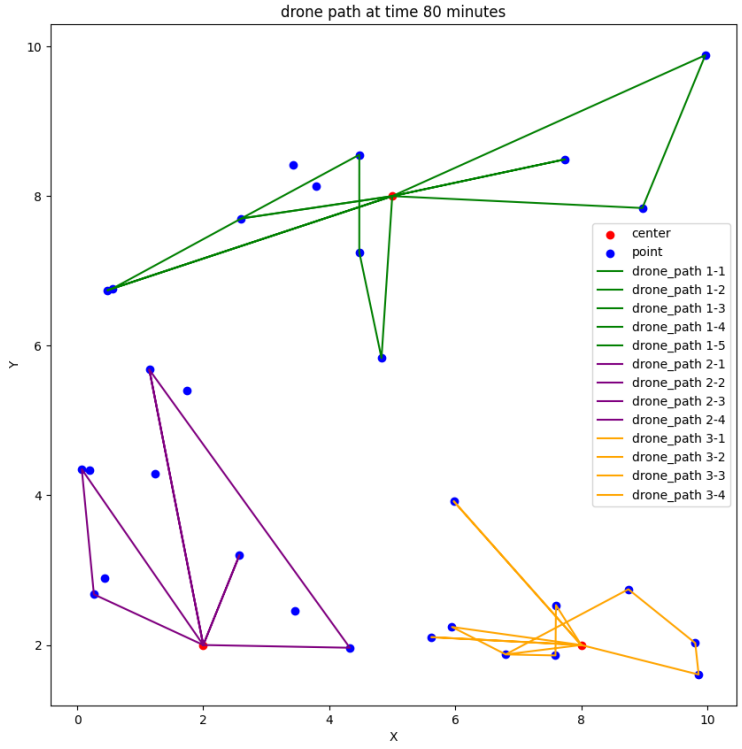

图 6　80 分钟时刻无人机调度路线

如图 3、4、5、6，可以看到算法可以找到当前时刻满足约束条件的最优路线，并在之后的时刻处理累积的不那么紧急的订单。

5.4 总路径计算

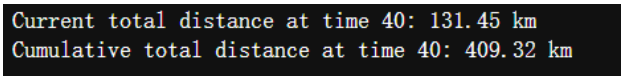

图 8　40 分钟时刻无人机路径总长度

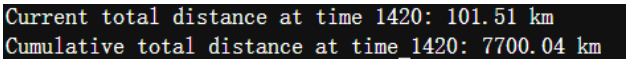

图 9　一天下来无人机路径总长度

附录：源代码

```
import matplotlib.pyplot as plt
import random
import math
import heapq
from collections import defaultdict

# plt.rc("font", family='Microsoft YaHei')

# 参数设置
j = 3  # 配送中心数量
k = 30  # 卸货点数量
t = 20  # 决策时间段（分钟）
max_capacity = 5  # 无人机最大容量
m = 5  # 每次生成订单的最大数量
max_distance = 20  # 公里
drone_speed = 60 / 60  # 公里/分钟
priority_limits = {'urgent': 0.5 * 60, 'semi-urgent': 1.5 * 60, 'normal': 3 * 60}  # 分钟
max_coord = 10  # 地图最大坐标

order_id = 0  # 全局订单 ID

# 手动设置配送中心和卸货点坐标
distribution_centers = [(max_coord / 2, max_coord * 0.8), (max_coord * 0.2, max_coord * 0.2),
(max_coord * 0.8, max_coord * 0.2)]
delivery_points = [(random.uniform(0, max_coord), random.uniform(0, max_coord)) for _ in range(k)]

# 绘制配送区域图
plt.figure(figsize=(10, 10))
plt.scatter([p[0] for p in distribution_centers], [p[1] for p in distribution_centers], color='red',
label='center')
plt.scatter([p[0] for p in delivery_points], [p[1] for p in delivery_points], color='blue',
label='point')
plt.title('map')
plt.xlabel('X')
plt.ylabel('Y')
plt.legend()
plt.savefig("地图")

# 生成订单
def generate_orders(current_time):
```

```
    global order_id
    orders = []
    for point in range(k):
        for _ in range(random.randint(0, m)):
            order = {
                'id': order_id,
                'time': current_time,
                'point': point,
                'priority': random.choice(['urgent', 'semi-urgent', 'normal']),
            }
            orders.append(order)
            order_id += 1
    return orders

# 距离计算
def calculate_distance(p1, p2):
    return math.sqrt((p1[0] - p2[0]) ** 2 + (p1[1] - p2[1]) ** 2)

# 路径规划（改进的最近邻算法）
def nearest_neighbor_algorithm(center, orders):
    routes = []
    while orders:
        route = []
        current_point = center
        current_distance = 0
        current_time = 0
        current_capacity = 0

        while orders and current_capacity < max_capacity and current_distance < max_distance / 2:
            next_order = None
            min_distance = float('inf')

            for order in orders:
                distance = calculate_distance(current_point, delivery_points[order['point']])
                delivery_time = current_time + distance / drone_speed

                if (current_capacity + 1) <= max_capacity and (current_distance + distance) <=
max_distance / 2:
                    if distance < min_distance and delivery_time <= priority_limits[order['priority']]:
                        next_order = order
                        min_distance = distance

            if next_order is None:
                break
```

```
            route.append(next_order)
            orders.remove(next_order)
            current_distance += min_distance
            current_time += min_distance / drone_speed
            current_capacity += 1
            current_point = delivery_points[next_order['point']]

        routes.append(route)

    return routes

def assign_drone(center, route):
    current_point = center
    current_time = 0
    current_capacity = 0

    for order in route:
        distance_to_order = calculate_distance(current_point, delivery_points[order['point']])
        travel_time = distance_to_order / drone_speed
        current_time += travel_time

        if (current_capacity + 1) > max_capacity:
            print(f"Drone capacity exceeded. Unable to deliver order at point {order['point']}")
            break

        if current_time - order['time'] > priority_limits[order['priority']]:
            print(f"Failed to deliver order at point {order['point']} within the required time")
        else:
            print(f"Order at point {order['point']} delivered at time {current_time:.2f} minutes")

        current_point = delivery_points[order['point']]
        current_capacity += 1


    distance_to_center = calculate_distance(current_point, center)
    return_time = distance_to_center / drone_speed
    current_time += return_time

    print(f"Drone returned to center at time {current_time:.2f} minutes")

def genetic_algorithm(initial_routes):
    population_size = 50
    generations = 100
    mutation_rate = 0.1
```

```
    def calculate_delivery_time(current_position, order):
        distance = calculate_distance(current_position, delivery_points[order['point']])
        travel_time = distance / drone_speed
        delivery_time = current_time + travel_time

        if order['priority'] == 'normal':
            time_limit = 180
        elif order['priority'] == 'semi-urgent':
            time_limit = 90
        elif order['priority'] == 'urgent':
            time_limit = 30

        return delivery_time

    def initial_population():
        population = []
        for _ in range(population_size):
            route = initial_routes[:]
            random.shuffle(route)
            population.append(route)
        return population

    def fitness(route):
        total_distance = 0
        penalty = 0

        for i in range(len(route) - 1):
            total_distance += calculate_distance(delivery_points[route[i]['point']],
delivery_points[route[i + 1]['point']])

        for order in route:
            delivery_time = calculate_delivery_time(delivery_points[0], order)
            if delivery_time > priority_limits[order['priority']]:
                penalty += 1000

        fitness_value = total_distance - penalty

        return fitness_value

    def crossover(parent1, parent2):
        if len(parent1) < 3 or len(parent2) < 3:
            # 如果任一路线长度小于 3，不进行交叉或以特殊方式处理
            return parent1[:], parent2[:]
```

```
        crossover_point1 = random.randint(1, len(parent1) - 2)
        crossover_point2 = random.randint(1, len(parent2) - 2)

        # 确保交叉点不同
        if crossover_point1 == crossover_point2:
            crossover_point2 = (crossover_point2 + 1) % len(parent2)

        child1 = parent1[:crossover_point1] + [gene for gene in parent2 if  gene not in
parent1[:crossover_point1]]
        child2 = parent2[:crossover_point2] + [gene for gene in parent1 if  gene not in
parent2[:crossover_point2]]

        return child1, child2

    def mutate(route):
        for i in range(len(route)):
            if random.random() < mutation_rate:
                j = random.randint(0, len(route) - 1)
                route[i], route[j] = route[j], route[i]
        return route

    def select_parents(population):
        fitnesses = [fitness(route) for route in population]
        total_fitness = sum(fitnesses)
        if total_fitness == 0:  # 添加检查以避免除以零
            return population[0], population[1]  # 或者选择其他适当的处理方式
        probabilities = [fit / total_fitness for fit in fitnesses]
        parents = []
        for _ in range(2):
            r = random.random()
            cumulative_probability = 0
            for i, probability in enumerate(probabilities):
                cumulative_probability += probability
                if r < cumulative_probability:
                    parents.append(population[i])
                    break
        return parents

    population = initial_population()

    for _ in range(generations):
        new_population = []
```

```
        while len(new_population) < population_size:
            parent1, parent2 = select_parents(population)
            child1, child2 = crossover(parent1, parent2)
            child1 = mutate(child1)
            child2 = mutate(child2)
            new_population.extend([child1, child2])

        population = sorted(new_population, key=lambda route: fitness(route))[:population_size]

    best_route = min(population, key=lambda route: fitness(route))
    return best_route

def plot_routes_all_centers(center_routes_dict, current_time):
    plt.figure(figsize=(10, 10))
    plt.scatter([p[0] for p in distribution_centers], [p[1] for p in distribution_centers],
color='red', label='center')
    plt.scatter([p[0] for p in delivery_points], [p[1] for p in delivery_points], color='blue',
label='point')

    colors = ['green', 'purple', 'orange']

    for center_idx, (center, routes) in enumerate(center_routes_dict.items()):
        for idx, route in enumerate(routes):
            route_points = [center] + [delivery_points[order['point']] for order in route] + [center]
            plt.plot([p[0] for p in route_points], [p[1] for p in route_points],
color=colors[center_idx % len(colors)], label=f'drone_path {center_idx + 1}-{idx + 1}')

    plt.title(f'drone path at time {current_time} minutes')
    plt.xlabel('X')
    plt.ylabel('Y')
    plt.legend()
    plt.savefig(f"无人机路径_{current_time}_minutes")
    plt.close()

def calculate_total_distance(center, route):
    total_distance = 0
    current_point = center
    for order in route:
        total_distance += calculate_distance(current_point, delivery_points[order['point']])
        current_point = delivery_points[order['point']]
    total_distance += calculate_distance(current_point, center)  # Return to center
    return total_distance

# 主决策循环
```

```
orders_dict = {}
orders_heap = []
cumulative_total_distance = 0

for current_time in range(0, 24 * 60, t):
    new_orders = generate_orders(current_time)
    for order in new_orders:
        orders_dict[order['id']] = order
        priority = 0 if order['priority'] == 'urgent' else 1 if order['priority'] == 'semi-urgent' else
2
        priority = 3 - priority
        heapq.heappush(orders_heap, (priority, order['time'], order['id']))
    # 打印当前时间和所有待处理订单
    print(f"Time: {current_time} minutes")
    print("Pending orders:")
    for order_id, order in orders_dict.items():
        print(f"Order ID: {order_id}, Point: {order['point']}, Priority: {order['priority']}, Time:
{order['time']}")

    pending_orders = []
    while orders_heap:
        priority, order_time, order_id = heapq.heappop(orders_heap)
        if current_time - order_time <= priority_limits[orders_dict[order_id]['priority']]:
            order = orders_dict[order_id]
            pending_orders.append(order)
        else:
            print(f"Order at point {order['point']} expired and was removed from the queue")

    center_orders = defaultdict(list)
    for order in pending_orders:
        closest_center = min(distribution_centers, key=lambda center: calculate_distance(center,
delivery_points[order['point']]))
        center_orders[closest_center].append(order)

    center_routes_dict = defaultdict(list)
    current_total_distance = 0

    for center_idx, (center, orders) in enumerate(center_orders.items()):
        initial_routes = nearest_neighbor_algorithm(center, orders)
        optimized_routes = [genetic_algorithm(route) for route in initial_routes]
        center_routes_dict[center] = optimized_routes

        for route in optimized_routes:
            assign_drone(center, route)
```

```
            for order in route:
                pending_orders.remove(order)

            current_total_distance += calculate_total_distance(center, route)

    cumulative_total_distance += current_total_distance

    print(f"Current total distance at time {current_time}: {current_total_distance:.2f} km")
    print(f"Cumulative total distance at time {current_time}: {cumulative_total_distance:.2f} km")

    plot_routes_all_centers(center_routes_dict, current_time)

    for order in pending_orders:
        priority = 0 if order['priority'] == 'urgent' else 1 if order['priority'] == 'semi-urgent' else
2
        if order in pending_orders:
            heapq.heappush(orders_heap, (priority, order['time'], order['id'], order))
        else:
            print(f"Order {order['id']} not found in all_orders. Skipping heappush.")
```