《高级算法设计与分析》课程作业

无人机配送路径规划问题

院（系）名 称：国家网络安全学院

专 业 名 称 ： 网络空间安全

学 生 姓 名 ： 曹若辰

学 生 学 号 ： 2023202210026

指 导 教 师 ： 林 海

二〇二四年六月

目录

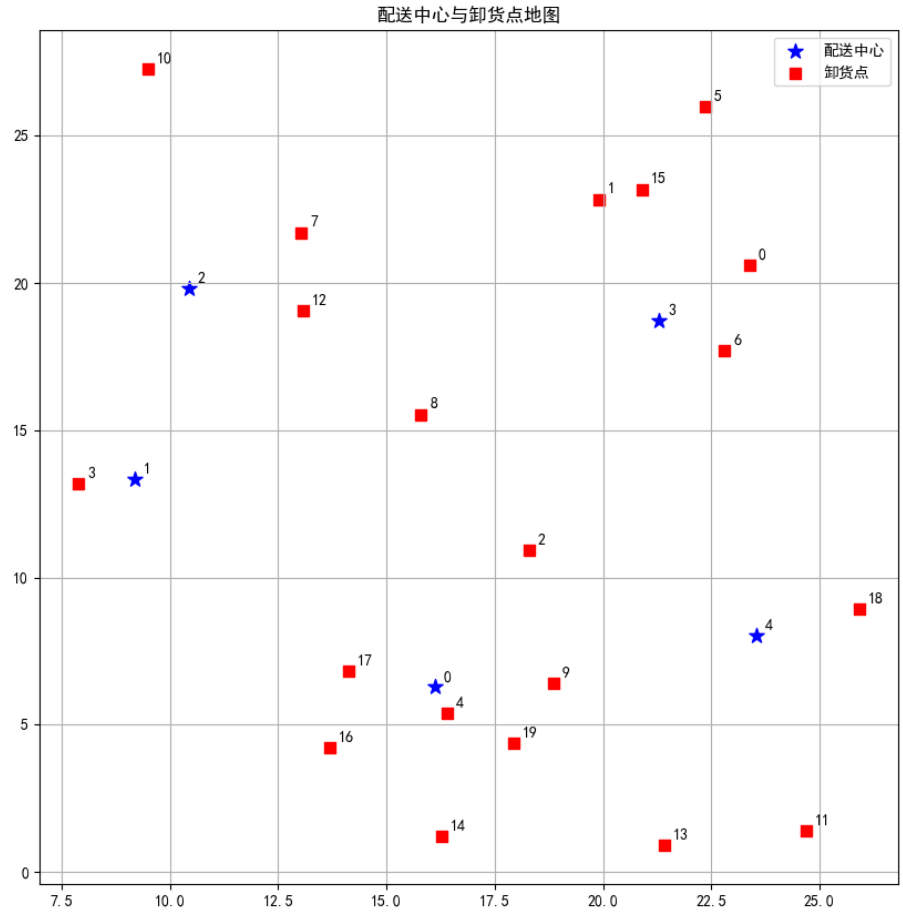
本文涉及的符号一览表

t	最小时间间隔（分钟）
T	一天的总时间（分钟）
j	单个配送中心进行的配送规划
J	配送中心集合
d	单个无人机的配送规划
D	单个配送中心的无人机集合
p	无人机飞行路径中的边
p_d	无人机 d 的飞行路径
o	单个订单
p_o	无人机飞行路径中飞向卸货点的边
t_o	订单 o 的配送时限
v	无人机飞行速度（公里/分钟）
k	单个卸货点
K	卸货点集合
O_k	卸货点 k 的订单
s	无人机最大飞行路程
n_d	无人机 d 搭载的货物数量
n	无人机最多能承载的货物数量
r	一个可行解所包含的一条子路径
R	一个可行解对应的所有子路径
n_r	无人机对应的子路径 r 所搭载的货物量
S_{ij}	一对卸货点 k_i 与 k_j 的节约值
$l_{a,b}$	地图上 a，b 两点间的直线距离

1 问题建模

1.1 问题实例化

假设本问题配送点数量 j 为 5，卸货中心 k 数量为 20，且配送点与卸货中心分布如下：



其中，每个卸货点 10 公里内必定存在至少一个配送中心，配送中心之间的最短间隔为 5 公里，地图长宽均为 30 公里，配送中心距离地图边界的最短距离为 5 公里。

假设每隔 t=30 分钟时，每个卸货点生成 0 至 m=5 个订单，每个订单有 50%的概率是一般优先级（180 分钟内送达），33%概率是较紧急优先级（90 分钟内

送达），17%概率是紧急优先级（30 分钟内送达），同时每隔 t=30 分钟，任务指派中心进行一次无人机调度，共计执行一天的时间，即 1440 分钟。

假设每架无人机每次最多携带 n=10 个物品，最大飞行距离为 20 公里，飞行速度为 1 公里/分钟（即 60 公里/小时），而每个配送中心的无人机数量无限。

1.2 问题形式化描述

问题建模为一个最优化问题，具体描述如下：

目标函数：

$$\min \sum_{t=0}^{T} \sum_{j \in J} \sum_{d \in D} \sum_{p \in p_d} p$$

其中 t 为单次时间间隔进行的所有路径规划，T 为一天的分钟数 1440；j 为单个配送中心进行的配送规划，J 为配送中心集合；d 为单个无人机的单次路径规划，D 为单次路径规划所派出的所有无人机；p_d 为无人机 d 被指派的飞行路径所遍历的边，p 为全连接图边的长度。目标函数是 min 优化，即希望一天内所有指派的所有无人机的全部飞行路径累计之和最小。

约束条件：

$$s.t. \quad \frac{\sum_{p_o \in p_d} p_o}{v} \leq t_o, \forall o \in O_k, d \in D, k \in K$$

$$\sum_{p \in p_d} p \leq s, \forall d \in D$$

$$n_d \leq n, \forall d \in D$$

$$p, n_d \geq 0, \forall p \in p_d, d \in D$$

约束 1 中 p_o 为 p_d 中无人机飞向卸货点的边（即除返回配送中心以外的边），v 为无人机飞行速度，t_o 为订单 o 的配送时限，O_k 为卸货点 k 的订单集合，K 为卸货点集合；约束 2 中 s 为无人机最大飞行距离；约束 3 中 n_d 为无人机 d 承载的货物数量，n 为无人机最大载货数量。约束中的变量 p 与 n_d 都非负，且 n_d 是整数。

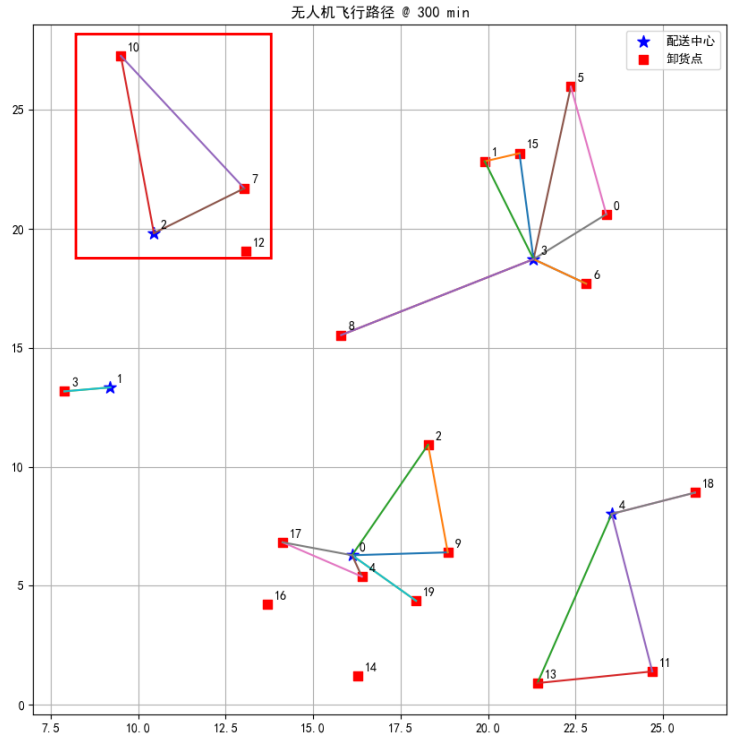
2 算法设计

本方案将遗传算法与节约算法相结合，解决无人机数量不确定的路径规划问题。

2.1 遗传算法设计

2.1.1 个体染色体设计

每个个体都对应无人机飞行路径的一个可行解，而个体的染色体则编码了数个无人机飞行路径的节点序列，每个节点序列被称为子路径，每个基因为一个配送中心节点或卸货点节点。以 DC（Distribution Center）代表配送中心，UP（Unloading Point）卸货点，如下图的全部子路径之集合代表了一个染色体，该染色体中的子序列[DC2, UP7, UP10, DC2]所代表的路径如下图红框所示：



2.1.2 种群初始化与精英解设计

遗传算法的种群包含了所有符合要求的个体，本算法中的种群即所有满足当前要求的无人机路径集合。

然而需要注意的是，由于在每个时间间隔分配任务时都要确保所有卸货点订单的需求被满足，**本算法需要重点关注那些在本时间间隔内不得不处理的订单**（即无法推迟到下个时间间隔运送的订单），并确保种群中一定存在向这些卸货点运送货物的解。

为此，在种群初始化时，本算法根据当前时间间隔内的优先订单（必须处理的订单），首先生成经过这些卸货点的精英解，**精英解可以确保最紧急的订单一定会被满足**。精英解的初始化方式采用的是节约算法，具体将在 2.2 章节中展开介绍。

种群中的精英解生成完毕后，在精英解的基础上进一步生成其他个体，具体方法为将精英解的基因随机打乱，即对应新的无人机飞行路径。

此外，为了确保遗传算法的种群多样性，本算法还设置了种群适应度多样性阈值，若新初始化得到的新个体的适应度较种群的整体水平稍差，但如果二者差值小于阈值，则允许这个较差的新个体进入种群，以防止种群适应度过早收敛；而随着种群规模扩大，阈值也逐渐降低，逐渐只允许优秀的初始化个体进入种群。

2.1.3 适应度函数

本算法定义个体的适应度函数如下：

$$fitness(R) = \sum_{r \in R} \frac{\sum_{p \in r} p}{n_r + \varepsilon}$$

其中 R 单个个体，r 为个体染色体中的子路径（即一架无人机的单次飞行路径），p 为 r 中所包含的边，n_r 为该子路径无人机所搭载的货物数量，ε 为一个小量，使得分母不为 0。适应度函数的意义是对一个个体所包含的全部子路径，求和无人机一次飞行的总路程平均到每一个货物的代价。

需要注意的是，本算法中，**个体的适应度越小，对应其越优秀**。

2.1.4 选择算子

本算法使用二元锦标赛选择方法，随机选择种群中一半的个体，取其中最优秀的两个个体（适应度函数最小）进行下一步的交叉。由于本问题存在多个局部最优解，二元锦标赛选择能够有效维持种群多样性，使算法选择不同个体。

2.1.5 交叉算子

本算法的交叉算子采用的是两点匹配交叉与部分匹配交叉，分别选择两个亲本的两个基因片段进行交换，若出现重复的基因片段，则将未交换部分出现的重复值替换为交换片段重复值所映射的基因，从而替换重复的基因。这样一来，一对亲本将产生两个子代，本算法随机选择一个子代加入种群。

2.1.6 变异算子

本算法的变异方式尝试在个体的染色体中间插入随机卸货点，前提是约束条件依然满足，形式化表达如下：

$$r = insert(k', r), k' \in K$$

$$s.t. \quad \frac{\sum_{p_o \in r} p_o}{v} \leq t_o, \forall o \in O_k, r \in R, k \in K$$

$$\sum_{p \in r} p \leq s, \forall r \in R$$

$$n_r \leq n, \forall r \in R$$

$$p, n_r \geq 0, \forall p \in r, r \in R$$

随机选择卸货点 k’在染色体子路径 r 中间插入，前提是优先订单时限要求、无人机负载上限以及无人机飞行路程上限 3 个约束依然满足。

此外，由于初始化与交叉可能导致子路径的起终点不是同一个配送中心（甚至不是配送中心），突变算子还负责将子路径中段的配送中心节点删除，并保证子路径两端是同一个配送中心。

需要注意的是，在交叉和变异步骤执行过后，种群中可能存在非法解，即使个体对应的无人机飞行路径是从配送中心开始到配送中心结束的完整回路，但该

路程可能超过无人机的最大飞行路程，同时可能导致原本精英解所顾及的优先卸货点订单不再被满足。

因此，在变异步骤结束后，本算法将检查种群中的解，并删除其中的非法解。由于初始化产生的精英解始终被放回式抽样，没有被破坏，**因此精英解一定还存在于种群中，优先订单依然可以被满足**。

2.2 节约算法设计

节约算法又称 C-W 算法，是物流管理中重要的启发式算法，最初用来解决物流问题中运输车辆数量不确定的相关问题。节约算法在本问题中用于生成遗传算法中种群初始化时的精英解，以确保精英解能够满足优先订单的配送，同时使得精英解的路程总代价最小。

2.2.1 初始化路径

首先进行路径初始化。对于每个卸货点 k，若该卸货点 k 有当前时间间隔内必须满足的订单（即下个时间间隔完成配送的时间已大于该订单的时限），将 k 与最近的配送中心 j 直接往返相连，代表一条直接往返于配送中心 j 与该卸货点 k 的无人机回路。

2.2.2 计算节约值

对于同一配送中心 j 的每对卸货点 k_a、k_b，定义其节约值 S_{ab} 如下：

$$S_{ab} = l_{j,a} + l_{j,b} - l_{a,b}$$

其中 $l_{x,y}$ 代表 x、y 两点在地图上的直线距离，由于本问题的运载工具是无人机，因此任意两点间都存在边，地图为全连接图。$l_{j,a}$ 与 $l_{j,b}$ 分别代表配送中心 j 到 k_a、k_b 的最短距离，$l_{a,b}$ 为两个卸货点的直线距离。节约值 S_{ab} 代表了将 k_a、k_b 的初始运输路径合并所能“节约”的路径代价，节约值越大说明这两条路径越值得合并，所能节省的路径越多。

2.2.3 合并路径

计算节约值后，对节约值从大到小进行排序。依次尝试合并节约值所对应的路径，合并的前提是无人机的单次飞行路程、无人机货物容量以及优先订单的时限依然满足约束要求。形式化描述如下：

$$r = merge(r_1, r_2)$$

$$s.t. \quad \frac{\sum_{p_o \in r} p_o}{v} \leq t_o, \forall o \in O_k, r \in R, k \in K$$

$$\sum_{p \in r} p \leq s, \forall r \in R$$

$$n_r \leq n, \forall r \in R$$

$$p, n_r \geq 0, \forall p \in r, r \in R$$

其中 merge 函数将两条子路径进行合并，即将从同一配送中心出发分别访问两个卸货点的路径合并为依次访问两个卸货点。如 r_1=[DC0, UP1, DC0]，r_2=[DC0, UP2, DC0]，合并后的 r=[DC0, UP1, UP2, DC0]。约束条件如 1.2 中所描述。

2.2.4 更新路径

重复 2.2.3 的合并路径过程，在满足约束条件的前提下，尽可能多的合并路径，直到所有合并都不满足要求。举例而言，2.2.3 中得到 r 后，令 r_1=r_2=r，若存在可合并的 r_3=[DC0, UP3, DC0]，合并得到 r'=[DC0, UP1, UP2, UP3, DC0]。若此时以 DC0 为起点的路径都已不满足合并约束，则合并停止，精英解生成完毕。

需要注意的是，**本算法的节约算法不合并以不同配送中心为起点的路径**，只以精英解满足优先订单需求为标准，尽可能减少无人机总路程代价，将精英解作为满足优先订单的基本标准。

2.3 算法可行性分析

本方案将遗传算法与节约算法相结合，在节约算法求解的精英解基础上，通过遗传算法进一步逼近最优解，从而解决全体路径最小化问题。下面进行论证。

对于每个配送中心 j 而言，通过节约算法得到的精英解一定能够满足所有附

近 10 公里内卸货点的优先订单需求，**最差的情况为一个卸货点 k 距离 j 恰好 10 公里，需要一个无人机直接往返 j、k 进行送货，而无人机的最大飞行里程为 20 公里，恰好能够满足需求，而该路径显然无法进一步合并（除非存在共线的卸货点）**。而由于 1.1 中的定义，所有卸货点 k 的 10 公里内必定至少存在 1 个配送中心 j，因此可以得到推论，**不存在卸货点 k'，其优先订单无法被节约算法生成的精英解满足**，因为一定存在一个 j，使得 j 的 10 公里内包含 k'，通过精英解满足其订单需求。

得到结论 1：**节约算法生成的精英解是任意 t 时刻对应的全部订单集合 D 的无人机路径规划问题的一个可行解。**

在精英解的基础上，本方案通过遗传算法进一步优化种群质量，以寻找代价更小的解法。在节约算法中，不难看出有一种最优化方向没有纳入考虑范畴，即**没有尝试合并以不同配送中心为起点的路径**，例如，有 r_1=[DC0, UP1, DC0]，r_2=[DC1, UP2, DC1]，当前的节约算法无法合并得到[DC0, UP1, UP2, DC0]或[DC1, UP1, UP2, DC1]。因此，需要引入遗传算法，对精英解路径进行更大幅度的改变。通过随机初始化、交叉、变异操作，本方案可以产生以精英解为基础的、更有创造性的个体。如随机初始化 r_2（随机打乱顺序），得到 r_2'=[DC1, DC1, UP2]，将 r_1 与 r_2'通过交叉得到 r_3=[DC0, UP1, UP2]以及 r_4=[DC1, DC0, DC1]，变异算子规范化 r_3 即可得到[DC0, UP1, UP2, DC0]（使染色体起终点为同一个配送中心）；或者直接变异个体 r_1，有可能可以得到[DC0, UP1, UP2, DC0]（插入 UP2）。本质上来讲，随机初始化、交叉以及变异操作可以以精英解为基底，通过线性变换得到任意解序列，因此可以得到推论，**以精英解为初始种群，遗传算法的随机初始化、交叉与变异方式可以（在删除非法解后）得到任意满足约束的可行解**。而遗传算法所设计的适应度函数及选择算子可以视为挑选可行解的一种贪心算法，而算法所包含的适应度多样性阈值也能够包容适应度稍差的解，增加种群多样性，避免陷入局部最优解。

得到结论 2：**本方案结合遗传算法和节约算法的算法是无人机配送路径规划问题最优解的一个近似解。**

总体算法如下：

结合遗传算法与节约算法的无人机配送路径规划问题算法

1: **Input**: 配送中心与卸货点地图 map，最小时间间隔 t，总时长 T

```
2:  Output: t'时刻规划的全部路径 map_t，总路程 L
3:  while t' ≤ T do
4:      种群 ← []
5:      节约值列表 ← []
6:      for 卸货点 i, 卸货点 j in map do
7:          节约值列表 ← 节约值列表 + (dist(卸货点 i, 配送中心) + dist(卸货点 j, 配送中心) – dist(卸货点 i, 卸货点 j))
8:      for 节约值 in 节约值列表 do
9:          if can_merge(节约值) then
10:             merge(节约值)
11:     精英解 ← 节约值列表
12:     随机解 ← shuffle(精英解)
13:     种群 ← 精英解 + 随机解
14:     while 迭代次数 and 适应度多样性阈值 do
15:         亲代 ← 锦标赛选择(种群)
16:         子代 ← 两点部分匹配交叉(亲代)
17:         if 适应度函数(子代) – avg(适应度函数(种群)) < 适应度多样性阈值 then
18:             种群 ← 种群 + 子代
19:             适应度多样性阈值 --
20:         迭代次数 --
21:     for 个体 in 种群 do
22:         if random.random > 变异阈值 then
23:             mutate(个体)
24:         规范化(个体)
25:     最佳路线 ← max(适应度函数(种群))
26:     map_t ← map_t + 最佳路线
27:     L ← L + sum(最佳路线)
28:     t' ← t' + t
29: return map_t, L
```

3 算法实现

3.1 确定常量与超参数

题目已经给定的常量：

```python
# 确定的常量
MAX_DISTANCE = 10  # 卸货节点距配送中心的最大距离
DAY_TIME = 1440  # 一天的总分钟数
DRONE_MAX_FLIGHT = 20  # 无人机一次飞行最远路程
DRONE_SPEED = 1  # 无人机速度
```

自行规定的常量，其数值在章节 1.1 中也有体现：

```python
# 自定义变量
J = 5  # 配送中心数
K = 20  # 卸货点数
T = 30  # 任务与下达命令的时间间隔
M = 5  # 卸货点每次生成的最大订单数
N = 10  # 每个无人机最多携带的物品数
MAP_LENGTH = 30  # 地图最大长度
MARGIN = 5  # 配送中心距离地图的边界
MIN_INTERVAL = 5  # 配送中心之间的最短间隔
```

遗传算法涉及的超参数：

```python
# 遗传算法超参数
POPULATION = 50  # 种群大小
GENERATION = 1000  # 迭代次数
MUTATION_RATE = 0.1  # 变异概率
DELTA = 0.5  # 适应度多样性阈值
```

3.2 功能性类与函数

3.2.1 配送中心类

DistributionCenter 类，包含 id 与坐标：

```python
# 配送中心
class DistributionCenter:
    def __init__(self, id, coords=(0, 0)):
        self.id = id
        self.coords = coords
```

3.2.2 随机生成订单函数

random_order 函数，分别以 50%、33%、17%的概率返回一般优先级、较紧急优先级和紧急优先级的订单：

```python
# 随机生成订单
def random_order():
    priority = random.randint(1, 6)
    if priority in (1, 2, 3):
        return 180
    elif priority in (4, 5):
        return 90
    elif priority == 6:
        return 30
```

3.2.3 卸货点类

UnloadingPoint 类，包括 id、坐标、订单列表、最近的配送中心及距离：

```python
# 卸货点
class UnloadingPoint:
    def __init__(self, id, coords=(0, 0)):
        self.id = id
        self.coords = coords
        self.orders = []
        self.nearest_distribution_center_id = None
        self.nearest_center_distance = None
```

update_point 方法，每隔时间间隔 t，更新订单的时限，调用 random_order，生成 0-m 个新订单并根据时限排序：

```python
    # 更新节点
    def update_point(self, m, t):
        for index, order in enumerate(self.orders):
            self.orders[index] -= t

        n = random.randint(0, m)
        for i in range(n):
            self.orders.append(random_order())
        self.orders.sort()
```

get_priority_orders 方法，根据订单时限，返回在本时间间隔内必须完成的订单：

```python
    # 获取优先订单
    def get_priority_orders(self):
        return [order for order in self.orders if order <= T]
```

finish_order 方法，移除已完成的订单：

```python
    # 已完成的订单
    def finish_order(self, finished_orders):
        for finished_order in finished_orders:
            self.orders.remove(finished_order)
```

3.2.4 地图类

Map 类，包括配送中心和卸货点的坐标以及实例，还有配送中心到卸货点以及卸货点之间的距离矩阵：

```python
# 地图
class Map:
    def __init__(self):
        self.distribution_center_coords = []  # 配送中心坐标
        self.unloading_point_coords = []  # 卸货点坐标
        self.distribution_centers = []  # 配送中心实例
        self.unloading_points = []  # 卸货点实例
        self.distance_matrix = {"DC2UP": [], "UP2UP": []}  # 距离矩阵
```

load_map 方法，加载之前生成的地图，如果没有则新生成地图，初始化配送中心与卸货点实例，计算距离矩阵：

```python
    # 加载地图
    def load_map(self, map_path, j_num, k_num):
        # 若存在已生成的地图，读取之
        if os.path.exists(map_path):
            with open(map_path, "rb") as file:
                map = pickle.load(file)

            self.distribution_center_coords = map["distribution_center_coords"]
            self.unloading_point_coords = map["unloading_point_coords"]

        # 否则生成新地图
        else:
            self.generate_map(j_num, k_num, map_path)

        # 初始化配送中心和卸货点
        self.init_distribution_center()
        self.init_unloading_point()

        # 计算配送中心和卸货点之间的距离，初始化距离矩阵
        self.calculate_distance_matrix()
```

generate_map 方法，生成新地图，先随机生成 j 个配送中心，确保它们之间的间距以及与地图边界的距离；再生成 k 个卸货点，确保每个卸货点附近 10 公里至少有一个配送中心：

```python
    # 生成地图
    def generate_map(self, j_num, k_num, map_path):
        dc_coords = []
        up_coords = []
        while len(dc_coords) < j_num:
            x, y = random.uniform(MARGIN, MAP_LENGTH - MARGIN), random.uniform(MARGIN, MAP_LENGTH - MARGIN)
            acceptable = True
            for center in dc_coords:
                # 保证配送中心之间的间隔
                if math.sqrt((x - center[0]) ** 2 + (y - center[1]) ** 2) < MIN_INTERVAL:
                    acceptable = False
                    break
            if acceptable:
                dc_coords.append((x, y))

        while len(up_coords) < k_num:
            x, y = random.uniform(0, MAP_LENGTH), random.uniform(0, MAP_LENGTH)
            for center in dc_coords:
                # 保证卸货点至少在一个装货点附近
                if math.sqrt(((x - center[0]) ** 2 + (y - center[1])) ** 2) < MAX_DISTANCE:
                    up_coords.append((x, y))
                    break

        self.distribution_center_coords = dc_coords
        self.unloading_point_coords = up_coords
```

保存并展示地图：

```
        # 保存地图
        map = {"distribution_center_coords": self.distribution_center_coords,
               "unloading_point_coords": self.unloading_point_coords}

        with open(map_path, "wb") as file:
            pickle.dump(map, file)

        # 展示地图
        plt.figure(figsize=(10, 10))
        plt.rcParams["font.sans-serif"] = ["SimHei"]
        plt.scatter(*zip(*self.distribution_center_coords), s=100, color="blue", marker="*", label="配送中心")
        plt.scatter(*zip(*self.unloading_point_coords), s=50, color="red", marker="s", label="卸货点")
        for index, dc_coords in enumerate(self.distribution_center_coords):
            plt.annotate(str(index), [coord + 0.2 for coord in dc_coords])
        for index, up_coords in enumerate(self.unloading_point_coords):
            plt.annotate(str(index), [coord + 0.2 for coord in up_coords])
        plt.title("配送中心与卸货点地图")
        plt.legend()
        plt.grid(True)
        plt.savefig("./map.png")
        plt.show()
```

init_distribution_center 方法，创建配送中心实例：

```
    # 初始化配送中心
    def init_distribution_center(self):
        for index, dc_coords in enumerate(self.distribution_center_coords):
            self.distribution_centers.append(DistributionCenter(index, dc_coords))
```

init_unloading_point 方法，创建卸货点实例：

```
    # 初始化卸货点
    def init_unloading_point(self):
        for index, up_coords in enumerate(self.unloading_point_coords):
            self.unloading_points.append(UnloadingPoint(index, up_coords))
```

calculate_distance_matrix 方法，计算距离矩阵，要计算的内容包括配送中心到各个卸货点的距离、各个卸货点之间的距离、距离每个卸货点最近的配送中心及其距离：

```
# 计算距离矩阵
def calculate_distance_matrix(self):
    # 计算配送中心到各个卸货点的距离
    for index, dc_coord in enumerate(self.distribution_center_coords):
        self.distance_matrix["DC2UP"].append([])
        for up_coord in self.unloading_point_coords:
            self.distance_matrix["DC2UP"][index].append(
                math.sqrt((dc_coord[0] - up_coord[0]) ** 2 + (dc_coord[1] - up_coord[1]) ** 2))

    # 计算各个卸货点之间的距离
    for index, up_coord_1 in enumerate(self.unloading_point_coords):
        self.distance_matrix["UP2UP"].append([])
        for up_coord_2 in self.unloading_point_coords:
            self.distance_matrix["UP2UP"][index].append(
                math.sqrt((up_coord_1[0] - up_coord_2[0]) ** 2 + (up_coord_1[1] - up_coord_2[1]) ** 2))

    # 计算距离每个卸货点最近的配送中心及其距离
    for up_index, up in enumerate(self.unloading_points):
        shortest_distance = math.inf
        nearest_id = 0
        for dc_index in range(J):
            if self.distance_matrix["DC2UP"][dc_index][up_index] < shortest_distance:
                nearest_id = dc_index
                shortest_distance = self.distance_matrix["DC2UP"][dc_index][up_index]

        up.nearest_distribution_center_id = nearest_id
        up.nearest_center_distance = shortest_distance
```

3.2.5 路线类

Routes 类，包含总距离及子路线：

```python
# 路径
class Routes:
    def __init__(self, routes):
        if routes is None:
            routes = []
        self.total_distance = 0
        self.sub_routes = routes
```

print_routes 方法，输出求解的路线：

```python
    # 打印路径
    def print_routes(self, unloading_destinations):
        for sub_route in self.sub_routes:
            center_id = unloading_destinations[sub_route[0]].nearest_distribution_center_id
            print("无人机飞行路径: " + "配送中心" + str(distribution_starts[center_id].id) + "->" + "".join(
                ["卸货点" + str(unloading_destinations[index].id) + "->" for index in sub_route]) + "配送中心" + str(
                distribution_starts[center_id].id))
```

save_routes 方法，保存求解的路线图：

```python
    # 保存路径
    def save_routes(self, map, current_time):
        plt.figure(figsize=(10, 10))
        plt.rcParams["font.sans-serif"] = ["SimHei"]
        plt.scatter(*zip(*map.distribution_center_coords), s=100, color="blue", marker="*", label="配送中心")
        plt.scatter(*zip(*map.unloading_point_coords), s=50, color="red", marker="s", label="卸货点")
        for index, dc_coords in enumerate(map.distribution_center_coords):
            plt.annotate(str(index), [coord + 0.2 for coord in dc_coords])
        for index, up_coords in enumerate(map.unloading_point_coords):
            plt.annotate(str(index), [coord + 0.2 for coord in up_coords])
        plt.title(f"无人机飞行路径 @ {current_time} min")
        plt.legend()
        plt.grid(True)

        for sub_route in self.sub_routes:
            dc = map.distribution_centers[map.unloading_points[sub_route[0]].nearest_distribution_center_id]
            ups = [map.unloading_points[index] for index in sub_route]

            nodes = [dc]
            nodes.extend(ups)

            for index, node in enumerate(nodes):
                plt.plot([node.coords[0], nodes[(index + 1) % (len(nodes))].coords[0]],
                         [node.coords[1], nodes[(index + 1) % (len(nodes))].coords[1]])
        plt.savefig(f"./output/Drone Flight Route @ {current_time} min")
```

3.3 遗传算法实现

3.3.1 查找最短路径类

Search 类，只需要初始化地图实例：

```python
# 查找最短路径
class Search:
    def __init__(self, map):
        self.map = map
```

get_priority_order_list 方法，返回当前时间间隔内优先级最高的订单列表，调用了卸货点类的 get_priority_orders 方法：

```
485     # 计算优先处理的订单
486     def get_priority_order_list(self):
487         priority_points = []
488         priority_orders = [0] * K
489         priority_orders_detail = [[]] * K
490 
491         for unloading_destination in self.map.unloading_points:
492             priority_order = unloading_destination.get_priority_orders()
493             if priority_order:
494                 priority_points.append(unloading_destination.id)
495                 priority_orders_detail[unloading_destination.id] = priority_order
496                 priority_orders[unloading_destination.id] = len(priority_order)
497         if priority_points:
498             return priority_points, priority_orders, priority_orders_detail
499         else:
500             return None
```

3.3.2 染色体解码函数

permutation_to_routes 函数，负责将染色体转换成对应的无人机飞行路径。循环检查卸货点，若当前子路径为空，则将新建子路径，添加卸货点，更新负载货物数、飞行距离：

```
356 # 路径解码
357 def permutation_to_routes(permutation):
358     global total_items, distance_list, flight_time
359     routes = []
360     current_route = None
361     for up in permutation:
362         unloading_destination = unloading_destinations[up]
363         if not current_route:
364             current_route = [up]
365             center_id = unloading_destination.nearest_distribution_center_id
366             total_items = len(unloading_destination.orders)
367             distance_list = [map.distance_matrix["DC2UP"][center_id][up]]
368             flight_time = current_time + (distance_list[0] / DRONE_SPEED)
```

若子路径不为空，则检查当前卸货点能否加入该子路径，根据是 1.2 中的 3 个约束条件，如果可以则将其加入，更新相关信息，否则需要开启新的子路径；遍历完所有卸货点后，将所有子路径加入 Routes 对象，并将其返回，作为该染色体所编码的无人机路径规划路线：

```
            else:
                total_items += len(unloading_destination.orders)
                if total_items > N:
                    routes.append(current_route)
                    current_route = [up]
                    total_items = flight_time = 0
                    distance_list = []
                    continue

                distance_list.append(map.distance_matrix["UP2UP"][current_route[-1]][up])
                total_distance = sum(distance_list)

                if total_distance > DRONE_MAX_FLIGHT:
                    routes.append(current_route)
                    current_route = [up]
                    continue

                flight_time += (map.distance_matrix["UP2UP"][current_route[-1]][up] / DRONE_SPEED)
                if flight_time > unloading_destination.orders[0]:
                    routes.append(current_route)
                    current_route = [up]
                    continue

                current_route.append(up)
        routes.append(current_route)
```

3.3.3 初始化种群

init_population 为 Search 类中的方法，负责种群初始化。首先根据节约算法，生成精英解，并根据适应度函数计算精英解的适应度，将其全部加入种群；接下来进行随机解初始化，从种群中随机选择一个个体，将其基因打乱（对应不同的无人机飞行路径），计算新个体的适应度，若新个体适应度较种群中每个个体的适应度差值均不大于适应度多样性阈值，则将新个体加入种群，多样性阈值随着新添加的个体数量递减，循环直到迭代次数达到设定的值后结束，就此获得了由精英解和随机解组成的种群：

```
    # 初始化种群
    def init_population(self):
        population = []
        priority_list = self.get_priority_order_list()

        # 生成精英解
        elite_solution = savings_algorithm(unloading_destinations, priority_list)
        elite_permutation = []
        for sub_route in elite_solution.sub_routes:
            elite_permutation += sub_route
        fitness = self.calculate_fitness(elite_solution)
        population.append((elite_permutation, fitness))

        # 生成剩余解
        delta = DELTA
        count = 0
        while len(population) < POPULATION:
            individual = elite_permutation[:]
            random.shuffle(individual)
            routes = permutation_to_routes(individual)
            fitness = self.calculate_fitness(routes)

            # 检查多样性
            if all(abs(fitness - individual[1]) >= delta for individual in population):
                population.append((individual, fitness))
            if count >= GENERATION:
                count = 0
                delta -= 0.1
            count += 1
        return population
```

3.3.4 适应度函数

Search 类中 calculate_fitness 方法计算个体的适应度函数。对于每个子路径，统计路径的距离之和，将其与无人机所搭载的货物数量作比，求和作为个体的适应度值（适应度越小说明个体对于每个货物所平均的路径代价越小，即个体越优秀）：

```
502     # 计算路线适应度
503     def calculate_fitness(self, routes, payload=0):
504         total_distance = 0
505         total_items = 0
506         for sub_route in routes.sub_routes:
507             center_id = unloading_destinations[sub_route[0]].nearest_distribution_center_id
508             total_distance += map.distance_matrix["DC2UP"][center_id][sub_route[0]]
509             for i in range(len(sub_route) - 1):
510                 total_items += len(unloading_destinations[i].orders)
511                 total_distance += map.distance_matrix["UP2UP"][sub_route[i]][sub_route[i + 1]]
512             total_items += len(unloading_destinations[sub_route[-1]].orders)
513             total_distance += map.distance_matrix["DC2UP"][center_id][sub_route[-1]]
514 
515         routes.total_distance = total_distance
516         # 适应度设置为运送每一个货物的平均距离
517         return total_distance / (payload + 0.01)
```

3.3.5 选择算子

在主循环的过程中，随机选择种群中的一半数量个体（与 2 取 max 值，避免种群中个体数量过少），选择其中适应度最优秀的两个个体作为亲本，进入下面的交叉环节：

```
594         # 主循环
595         if len(best_solution) > 1:
596             for generation in range(GENERATION):
597                 count = 0
598                 while True:
599                     tournament = random.sample(population, max((len(population) // 2), 2))
600                     tournament.sort(key=lambda x: x[1])
601                     parent1, parent2 = tournament[:2]
602                     parent1 = parent1[0]
603                     parent2 = parent2[0]
604                     child = [0] * len(parent1)
605                     generate_child(child, parent1, parent2)
606                     child_routes = permutation_to_routes(child)
607                     fitness = search.calculate_fitness(child_routes)
```

3.3.6 交叉算子

generate_child 函数根据两个亲本，采用部分匹配交叉方法，生成子代。首先随机选择要交叉的染色体索引 i 和 j，并使用 flag1 和 flag2 分别标记基因是否在子代中出现：

```
# 生成子代
def generate_child(child, parent1, parent2):
    N = len(parent1)
    child1, child2 = [0] * N, [0] * N
    flag1, flag2 = [0] * K, [0] * K

    i = random.randint(0, N - 1)
    j = i
    while j == i:
        j = random.randint(0, N - 1)
    if j < i:
        i, j = j, i

    for k in range(i, j + 1):
        child1[k] = parent1[k]
        flag1[child1[k]] = True
        child2[k] = parent2[k]
        flag2[child2[k]] = True
```

从交叉区间结束的位置 j 开始，处理剩余基因，将父代中未在子代出现过的基因依次填入子代，最后得到两个子代，随机返回其中一个：

```
    index = (j + 1) % N
    p1, p2 = index, index
    while index != i:
        while flag1[parent2[p2]]:
            p2 = (p2 + 1) % N
        while flag2[parent1[p1]]:
            p1 = (p1 + 1) % N
        child1[index] = parent2[p2]
        flag1[child1[index]] = True
        child2[index] = parent1[p1]
        flag2[child2[index]] = True
        index = (index + 1) % N

    if random.random() < 0.5:
        child[:] = child1
    else:
        child[:] = child2
```

3.3.7 变异算子

个体变异操作由 mutate_individual 函数实现。首先初始化 best_cost 为子路径总距离，记录最佳代价，random_up 为随机卸货点，并找到包含它的子路径：

```
# 变异个体
def mutate_individual(child, child_routes):
    global next_up_index
    best_cost = child_routes.total_distance

    random_up_index = random.randint(0, len(child) - 1)
    random_up = child[random_up_index]

    random_up_route = []
    for sub_route in child_routes.sub_routes:
        if random_up in sub_route:
            random_up_route = sub_route
            break
```

计算当前路径中的代价 s1，v1 和 v2 分别是 random_up 前后的卸货点，s1 为 random_up 与其前后卸货点的距离变化：

```
    v1 = v2 = 0
    if random_up_index > 0:
        v1 = child[random_up_index - 1]
    if random_up_index < len(child) - 1:
        v2 = child[random_up_index + 1]

    if v1 and v2 and random_up != random_up_route[0]:
        s1 = map.distance_matrix["UP2UP"][random_up][v1] + map.distance_matrix["UP2UP"][random_up][v2] - \
             map.distance_matrix["UP2UP"][v1][v2]
    else:
        s1 = map.distance_matrix["DC2UP"][unloading_destinations[random_up].nearest_distribution_center_id][
                 random_up] + map.distance_matrix["UP2UP"][random_up][v2] - map.distance_matrix["DC2UP"][
                 unloading_destinations[v2].nearest_distribution_center_id][v2]
```

遍历子路径，尝试插入 random_up：

```
    route_index = better_route_index = 0
    for sub_route in child_routes.sub_routes:
        if sub_route != random_up_route and sum([len(unloading_destinations[up].orders) for up in sub_route]) + \
                len(unloading_destinations[random_up].orders) <= N:
            for i in range(len(sub_route)):
                v1 = sub_route[0] if i == 0 else sub_route[i - 1]
                v2 = sub_route[i]
                new_route = sub_route[:i] + [random_up] + sub_route[i:]
```

检查插入 random_up 后的子路径是否满足无人机容量、无人机飞行距离以及订单时限约束，若满足则进行插入：

```
                if i == 0:
                    center_id = unloading_destinations[random_up].nearest_distribution_center_id
                    distance_list = [map.distance_matrix["DC2UP"][center_id][random_up]]
                else:
                    center_id = unloading_destinations[sub_route[0]].nearest_distribution_center_id
                    distance_list = [map.distance_matrix["DC2UP"][center_id][sub_route[0]]]
                for k in range(len(new_route) - 1):
                    distance_list.append(
                        map.distance_matrix["UP2UP"][new_route[k]][new_route[k + 1]])
                distance_list.append(map.distance_matrix["DC2UP"][center_id][new_route[-1]])

                total_distance_tmp = sum(distance_list)
                if total_distance_tmp > DRONE_MAX_FLIGHT:
                    continue

                flight_time = [current_time + (sum(distance_list[:k + 1]) / DRONE_SPEED) for k in
                               range(len(new_route) - 1)]
                constraint_list = [unloading_destinations[k].orders[0] for k in new_route]
                if flight_time > constraint_list:
                    continue
```

计算新路径的总距离和代价 s2，并更新最佳代价 best_cost：

```
                if i == 0:
                    s2 = unloading_destinations[random_up].nearest_center_distance + \
                         map.distance_matrix["UP2UP"][random_up][v2] - unloading_destinations[
                             sub_route[0]].nearest_center_distance
                else:
                    s2 = map.distance_matrix["UP2UP"][v1][random_up] + \
                         map.distance_matrix["UP2UP"][v2][random_up] - \
                         map.distance_matrix["UP2UP"][v1][v2]
                temp_cost = (best_cost - s1 - s2)

                if temp_cost < best_cost:
                    best_cost = temp_cost
                    next_up_index = i
                    better_route_index = route_index
        route_index += 1
```

如果找到更好的路径，则更新子路径，将 random_up 插入到新的子路径中，并从原子路径中移除：

```
473     if best_cost < child_routes.total_distance:
474         random_up_route.remove(random_up)
475         child_routes.sub_routes[better_route_index].insert(next_up_index, random_up)
476         if [] in child_routes.sub_routes:
477             child_routes.sub_routes.remove([])
```

3.4 节约算法实现

节约算法由 savings_algorithm 函数实现，输入为优先级最高的卸货点以及订单，输出为满足优先订单的精英解。

3.4.1 节约值计算

初始化路径的工作已经在 Map 类中完成，记录在了距离矩阵中，于是首先根据 2.2.2 中公式计算节约值，从大到小进行排序：

```
239 # 节约算法
240 def savings_algorithm(unloading_destinations, priority_list):
241     priority_points = priority_list[0]
242     print("优先级最高的卸货点：", priority_points)
243     routes = [[up] for up in priority_points]
244     savings = []
245 
246     for index, ud1 in enumerate(unloading_destinations):
247         if index < len(unloading_destinations) - 1:
248             for ud2 in unloading_destinations[index + 1:]:
249                 nearest_center_id = ud1.nearest_distribution_center_id
250                 saving = (map.distance_matrix["DC2UP"][nearest_center_id][ud1.id] +
251                           map.distance_matrix["DC2UP"][nearest_center_id][ud2.id] -
252                           map.distance_matrix["UP2UP"][ud1.id][ud2.id])
253                 savings.append((saving, ud1.id, ud2.id, nearest_center_id))
254 
255     savings.sort(reverse=True)
```

3.4.2 节约值合并

依次尝试合并节约值：

```
    # 约束1：无人机一次最多只能携带n个物品
    # 约束2：无人机一次飞行最远路程为20公里
    # 约束3：所有卸货点的需求都必须满足
    while len(savings) > 0:
        saving, ud1, ud2, nearest_center_id = savings[0]
        savings = savings[1:]
        route1 = None
        route2 = None

        for route in routes:
            if ud1 == route[-1]:
                route1 = route
            if ud2 == route[0]:
                route2 = route

        if route1 is None or route2 is None:
            continue
```

根据 2.2.3 的 3 个约束条件，判断合并的节约值是否依然满足约束，即优先订单是否能够在时限内送达、无人机负载是否超过限重以及无人机飞行路程是否超过最大里程：

```
    if route1 is not None and route2 is not None and route1 != route2:
        new_route = route1 + route2
        total_items = sum(priority_list[1][point_id] for point_id in new_route)
        if total_items > N:
            continue

        center_id = unloading_destinations[new_route[0]].nearest_distribution_center_id
        distance_list = [map.distance_matrix["DC2UP"][center_id][new_route[0]]]
        for k in range(len(new_route) - 1):
            distance_list.append(map.distance_matrix["UP2UP"][new_route[k]][new_route[k + 1]])
        distance_list.append(map.distance_matrix["DC2UP"][center_id][new_route[-1]])
        total_distance = sum(distance_list)
        if total_distance > DRONE_MAX_FLIGHT:
            continue

        flight_time = [(sum(distance_list[:k + 1]) / DRONE_SPEED) for k in range(len(new_route))]
        constraint_list = [priority_list[2][node][0] for node in new_route]
        if sum([flight_time[i] > constraint_list[i] for i in range(len(constraint_list))]):
            continue

        routes.remove(route1)
        routes.remove(route2)
        routes.append(new_route)
```

3.4.3 节约值更新

如果满足约束，则更新节约值，重复以上过程，直到没有节约值可更新，返回节约算法求解出的精英解：

```
            # 更新节约值
            for saving, ud1, ud2, center in savings:
                if ud1 in new_route and ud2 in new_route:
                    savings.remove((saving, ud1, ud2, center))
                elif ud2 in new_route[1:]:
                    savings.remove((saving, ud1, ud2, center))
                elif ud1 in new_route[:-1]:
                    savings.remove((saving, ud1, ud2, center))
                elif ud1 == new_route[-1]:
                    savings.remove((saving, ud1, ud2, center))
                    saving = (map.distance_matrix["DC2UP"][nearest_center_id][ud1] +
                              map.distance_matrix["DC2UP"][nearest_center_id][ud2] -
                              map.distance_matrix["UP2UP"][ud1][ud2])
                    savings.append((saving, ud1, ud2, nearest_center_id))
            savings.sort(reverse=True)

    return Routes(routes)
```

3.5 主函数

主函数首先确定输出无人机飞行路径的目录，读取地图文件并初始化时间、总距离、优先订单：

```
if __name__ == "__main__":
    # 确保输出目录存在
    output_dir = "output"
    if os.path.isdir(output_dir):
        shutil.rmtree(output_dir)
    os.makedirs(output_dir, exist_ok=True)

    map = Map()
    map.load_map("map.pkl", J, K)
    search = Search(map)

    distribution_starts = map.distribution_centers
    unloading_destinations = map.unloading_points

    current_orders = deque()
    current_time = 0
    total_distance = 0
    time_list = []
    total_distance_list = []
```

每隔 t 时间间隔，更新每个卸货点的订单需求，统计优先级最高的订单（本时间间隔内必须处理的订单），初始化种群：

```
570     while current_time <= DAY_TIME:
571         time_list.append(current_time)
572         print("\n" * 3)
573         print("-" * 50)
574         print(f"当前时间：{current_time}/{DAY_TIME}min")
575         for point in unloading_destinations:
576             # 更新订单情况
577             point.update_point(M, T)
578             # 打印订单情况
579             print(f"卸货点{point.id}现有订单：{point.orders}")
580 
581         priority_points = search.get_priority_order_list()
582         if priority_points is None:
583             current_time += T
584             total_distance_list.append(total_distance)
585             continue
586 
587         # 初始化种群
588         population = search.init_population()
589         best_solution = population[0][0]
590         best_fitness = float('inf')
591 
592         delta = DELTA
```

遗传算法的主循环中，在迭代次数达到预期数值前，循环选择亲本进行交叉，产生新子代：

```
594     # 主循环
595     if len(best_solution) > 1:
596         for generation in range(GENERATION):
597             count = 0
598             while True:
599                 tournament = random.sample(population, max((len(population) // 2), 2))
600                 tournament.sort(key=lambda x: x[1])
601                 parent1, parent2 = tournament[:2]
602                 parent1 = parent1[0]
603                 parent2 = parent2[0]
604                 child = [0] * len(parent1)
605                 generate_child(child, parent1, parent2)
606                 child_routes = permutation_to_routes(child)
607                 fitness = search.calculate_fitness(child_routes)
```

对个体进行变异，若新个体适应度较种群中每个个体的适应度差值均不大于适应度多样性阈值，则将新个体加入种群，多样性阈值随着新添加的个体数量递减，种群中适应度最靠后的个体将被移除，维持种群个体数量一定，最优解被设为适应度最高的个体。循环完成后，通过 permutation_to_routes 将最佳个体的染色体转换为最佳路径：

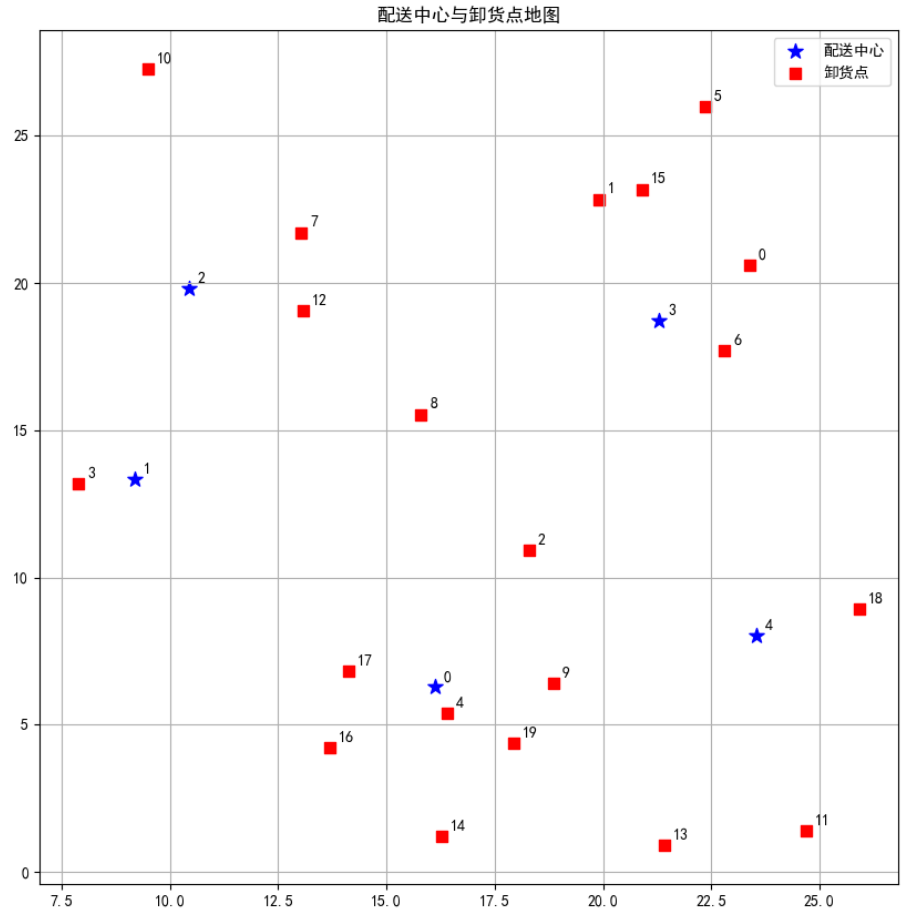
```
                # 局部搜索改进
                if random.random() < MUTATION_RATE:
                    mutate_individual(child, child_routes)
                # 检查多样性
                if all(abs(fitness - ind[1]) >= delta for ind in population):
                    population.append((child, fitness))
                    break
                else:
                    count += 1
                    if count >= GENERATION:
                        count = 0
                        delta -= 0.1

            # 选出被淘汰的个体
            population.sort(key=lambda ind: ind[1])
            population = population[:POPULATION]

        best_solution = population[0][0]
    best_routes = permutation_to_routes(best_solution)
```

删去最佳路径中不必要的路径，进行规范化，将染色体中间的配送中心节点删除，并使得染色体前后的配送中心一致，记录最佳路径的 Routes 实例，增加 total_distance，打印相关信息，保存最佳路径图，完成相关订单，current_time 增加一个时间间隔，代表这个时间间隔的事务已经全部处理完毕，继续循环下一个时间间隔。算法执行完毕后，打印总路径代价：

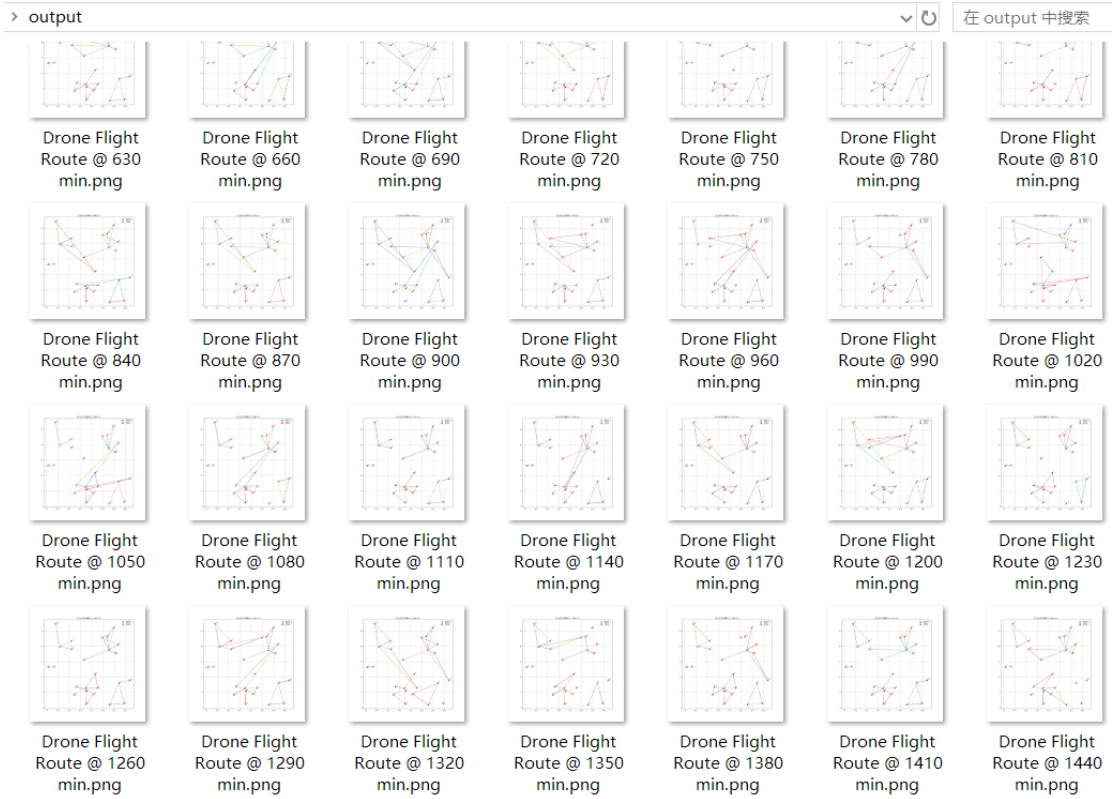
```
    # 删去不必要的路径
    final_routes = []
    for route in best_routes.sub_routes:
        if route and any(point in priority_points[0] for point in route):
            nearest_center_id = unloading_destinations[route[0]].nearest_distribution_center_id
            final_routes.append(route)

    best_routes = Routes(final_routes)
    best_routes.save_routes(map, current_time)
    search.calculate_fitness(best_routes)
    best_routes.print_routes(unloading_destinations)
    total_distance += best_routes.total_distance
    total_distance_list.append(total_distance)

    priority_order_list = search.get_priority_order_list()
    for up in best_solution:
        unloading_destinations[up].finish_order(priority_order_list[2][up])

    current_time += T

print("所有无人机总飞行路程：", total_distance)
```

4 实验效果

随机生成的配送中心与卸货点地图如下：



对于每个时间间隔 t=30，算法均产生了一个最优路径图：



分析其中 3 次生成结果：

300 分钟时，各卸货点现有订单如下：

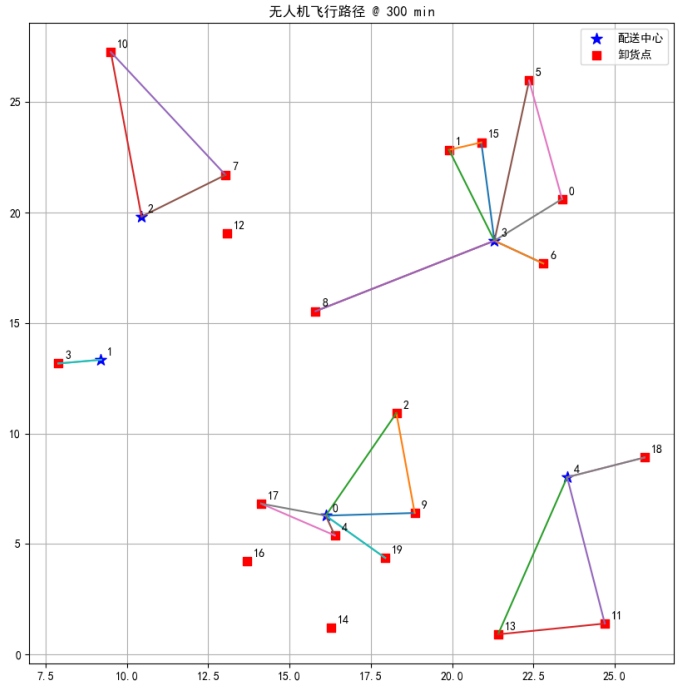
```
--------------------------------------------------
当前时间：300/1440min
卸货点0现有订单：[30, 30, 30, 60, 90, 120, 180]
卸货点1现有订单：[30, 30, 30, 90, 90, 90, 90, 180, 180]
卸货点2现有订单：[30, 60, 60, 60, 60, 60, 60, 60, 90, 90]
卸货点3现有订单：[30, 30, 30, 60, 60, 60, 60, 90, 120, 120, 120, 120, 120, 150, 180, 180]
卸货点4现有订单：[30, 30, 60, 60, 60, 90, 90, 90, 120, 120, 150]
卸货点5现有订单：[30, 60, 90, 150, 150, 180]
卸货点6现有订单：[30, 30, 30, 60, 60, 90, 90, 90, 150]
卸货点7现有订单：[30, 30, 30, 90, 120, 120, 150]
卸货点8现有订单：[30, 30, 30, 30, 30, 60, 60, 90, 90, 120, 120, 120, 150]
卸货点9现有订单：[30, 30, 60, 60, 60, 60, 60, 60, 90, 120, 150, 180, 180]
卸货点10现有订单：[30, 30, 60, 60, 120, 150, 150, 150, 150, 180, 180]
卸货点11现有订单：[30, 30, 60, 90, 90, 120, 150, 150]
卸货点12现有订单：[90, 120, 120, 120, 150]
卸货点13现有订单：[30, 30, 30, 30, 30, 60, 60, 90, 90, 120, 120, 150]
卸货点14现有订单：[60, 60, 60, 90, 150, 150, 180, 180, 180]
卸货点15现有订单：[30, 60, 60, 90, 90, 120, 120, 180, 180, 180, 180]
卸货点16现有订单：[60, 90, 90, 150]
卸货点17现有订单：[30, 30, 30, 30, 120, 120, 120, 150, 180]
卸货点18现有订单：[30, 30, 60, 60, 60, 60, 90, 90, 90, 120, 120, 120, 150, 150, 180]
卸货点19现有订单：[30, 60, 60, 60, 60, 90, 120, 120, 120, 150, 180, 180]
优先级最高的卸货点： [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 13, 15, 17, 18, 19]
```

优先级最高的卸货点包括 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 13, 15, 17, 18, 19。

算法给出的最优路径为：

```
无人机飞行路径：配送中心3->卸货点6->配送中心3
无人机飞行路径：配送中心4->卸货点13->卸货点11->配送中心4
无人机飞行路径：配送中心3->卸货点5->卸货点0->配送中心3
无人机飞行路径：配送中心1->卸货点3->配送中心1
无人机飞行路径：配送中心0->卸货点9->卸货点2->配送中心0
无人机飞行路径：配送中心2->卸货点10->卸货点7->配送中心2
无人机飞行路径：配送中心4->卸货点18->配送中心4
无人机飞行路径：配送中心0->卸货点19->配送中心0
无人机飞行路径：配送中心3->卸货点15->卸货点1->配送中心3
无人机飞行路径：配送中心3->卸货点8->配送中心3
无人机飞行路径：配送中心0->卸货点4->卸货点17->配送中心0
```

即下方路径图：



750 分钟时，各卸货点订单如下：

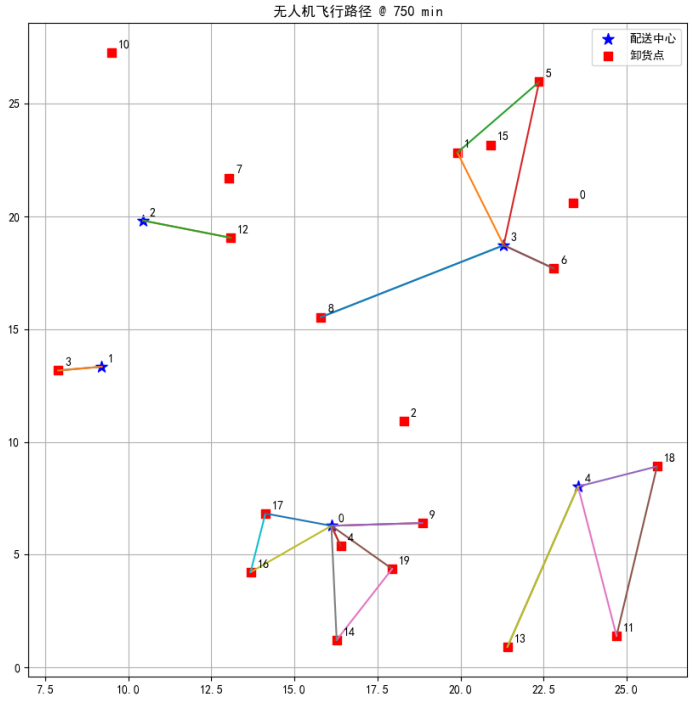
```
--------------------------------------------------
当前时间：750/1440min
卸货点0现有订单：[60, 60, 120, 120, 150]
卸货点1现有订单：[30, 30, 30, 30, 30, 60, 60, 90, 90, 90, 90, 120, 120, 120, 150, 180]
卸货点2现有订单：[90, 120, 180]
卸货点3现有订单：[30, 30, 30, 60, 60, 60, 90, 90, 120, 150, 180]
卸货点4现有订单：[30, 30, 30, 30, 30, 30, 30, 60, 90, 90, 90, 90, 120, 180, 180, 180]
卸货点5现有订单：[30, 30, 90, 90, 90, 150, 150]
卸货点6现有订单：[30, 30, 60, 60, 60, 90, 90, 150]
卸货点7现有订单：[60, 90, 90, 90, 90, 150, 150, 150, 180]
卸货点8现有订单：[30, 30, 30, 30, 60, 60, 60, 60, 90, 90, 90, 120, 180, 180, 180]
卸货点9现有订单：[30, 30, 30, 30, 60, 120, 120, 120, 150, 150, 180, 180, 180]
卸货点10现有订单：[60, 60, 60, 60, 90, 90, 90, 150, 150, 150, 150, 180]
卸货点11现有订单：[30, 60, 60, 90, 120, 150, 180, 180]
卸货点12现有订单：[30, 30, 30, 30, 60, 60, 90, 90, 120, 120, 150, 180, 180]
卸货点13现有订单：[30, 30, 30, 30, 90, 150, 180]
卸货点14现有订单：[30, 30, 30, 60, 90, 90, 120, 150, 150]
卸货点15现有订单：[60, 90, 90, 120, 120, 150, 150, 180]
卸货点16现有订单：[30, 30, 60, 60, 60, 60, 90, 90, 120, 120, 180]
卸货点17现有订单：[30, 30, 60, 90, 90, 90, 150, 180, 180]
卸货点18现有订单：[30, 30, 30, 30, 60, 60, 60, 90, 90, 90, 120, 120, 150, 150, 180]
卸货点19现有订单：[30, 30, 30, 30, 60, 60, 60, 90, 90, 120, 120, 180, 180, 180]
优先级最高的卸货点： [1, 3, 4, 5, 6, 8, 9, 11, 12, 13, 14, 16, 17, 18, 19]
```

优先级最高的卸货点包括 1, 3, 4, 5, 6, 8, 9, 11, 12, 13, 14, 16, 17, 18, 19。

算法给出的最优路径为：

无人机飞行路径：配送中心1->卸货点3->配送中心1
无人机飞行路径：配送中心0->卸货点4->配送中心0
无人机飞行路径：配送中心4->卸货点18->卸货点11->配送中心4
无人机飞行路径：配送中心4->卸货点13->配送中心4
无人机飞行路径：配送中心3->卸货点8->配送中心3
无人机飞行路径：配送中心2->卸货点12->配送中心2
无人机飞行路径：配送中心0->卸货点9->配送中心0
无人机飞行路径：配送中心0->卸货点19->卸货点14->配送中心0
无人机飞行路径：配送中心0->卸货点16->卸货点17->配送中心0
无人机飞行路径：配送中心3->卸货点1->卸货点5->配送中心3
无人机飞行路径：配送中心3->卸货点6->配送中心3

即下方路径图：



1230 分钟时，各卸货点现有订单如下：

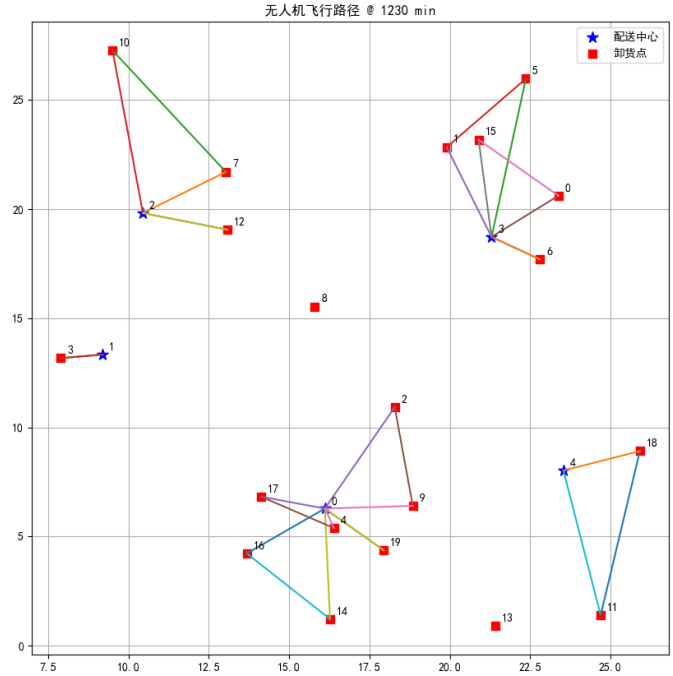
```
--------------------------------------------------
当前时间：1230/1440min
卸货点0现有订单：[30, 60, 60, 60, 60, 60, 90, 90, 90, 120, 180, 180, 180]
卸货点1现有订单：[30, 30, 30, 30, 60, 60, 90, 120, 150]
卸货点2现有订单：[30, 30, 60, 60, 60, 150, 180]
卸货点3现有订单：[30, 30, 30, 60, 60, 90, 90]
卸货点4现有订单：[30, 30, 30, 60, 60, 90, 90, 90, 150]
卸货点5现有订单：[30, 30, 30, 30, 60, 90, 90, 120, 120, 180, 180, 180]
卸货点6现有订单：[30, 30, 60, 90, 90, 90, 90, 120, 120, 120, 150]
卸货点7现有订单：[30, 30, 30, 30, 30, 60, 90, 120]
卸货点8现有订单：[60, 60, 90, 90, 120, 120, 150, 150]
卸货点9现有订单：[30, 30, 60, 60, 60, 60, 90, 120]
卸货点10现有订单：[30, 30, 30, 30, 30, 60, 60, 90, 120, 150]
卸货点11现有订单：[30, 60, 60, 90, 90, 90, 90, 90, 150, 150, 180, 180, 180]
卸货点12现有订单：[30, 30, 30, 30, 60, 60, 60, 60, 90, 120, 120, 150, 150, 150, 150, 180, 180]
卸货点13现有订单：[60, 60, 90, 120, 120, 120, 120, 150, 150, 180]
卸货点14现有订单：[30, 30, 30, 30, 60, 90, 90, 90, 120, 150]
卸货点15现有订单：[30, 30, 30, 30, 30, 60, 90, 120, 150]
卸货点16现有订单：[30, 30, 30, 30, 30, 120, 180, 180, 180]
卸货点17现有订单：[30, 30, 30, 30, 60, 60, 60, 60, 60, 90, 90, 120, 120, 150, 150, 180, 180]
卸货点18现有订单：[30, 30, 30, 60, 60, 60, 90, 90, 120]
卸货点19现有订单：[30, 30, 60, 60, 60, 90, 180]
优先级最高的卸货点： [0, 1, 2, 3, 4, 5, 6, 7, 9, 10, 11, 12, 14, 15, 16, 17, 18, 19]
```

优先级最高的卸货点包括 0, 1, 2, 3, 4, 5, 6, 7, 9, 10, 11, 12, 14, 15, 16, 17, 18, 19。

算法给出的最优路径为：

```
无人机飞行路径：配送中心3->卸货点6->配送中心3
无人机飞行路径：配送中心3->卸货点5->卸货点1->配送中心3
无人机飞行路径：配送中心3->卸货点0->卸货点15->配送中心3
无人机飞行路径：配送中心0->卸货点14->卸货点16->配送中心0
无人机飞行路径：配送中心2->卸货点7->卸货点10->配送中心2
无人机飞行路径：配送中心0->卸货点2->卸货点9->配送中心0
无人机飞行路径：配送中心0->卸货点19->配送中心0
无人机飞行路径：配送中心4->卸货点11->卸货点18->配送中心4
无人机飞行路径：配送中心1->卸货点3->配送中心1
无人机飞行路径：配送中心0->卸货点17->卸货点4->配送中心0
无人机飞行路径：配送中心2->卸货点12->配送中心2
```

即下方路径图：



最终，算法对该地图的总路程代价为 6979：

所有无人机总飞行路程： 6979.069182262332