高级算法大作业课程报告

基于遗传算法的无人机配送路径规划问题求解

课程： 高级算法

学号： 2023202210108

姓名： 赵家璇

专业： 网络空间安全

班级： 3 班

目录

一、 引言

1.1 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，实现图1所示的无人机配送的路径规划。在 $40km \times 40km$ 大小的区域中，共有 5 个配送中心，任意一个配送中心有用户所需要的商品，其数量无限，同时任一配送中心的无人机数量无限。该区域同时有 60 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 $1-3$ 个订单，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：

1. **一般**：3 小时内配送到即可
2. **较紧急**：1.5 小时内配送到
3. **紧急**：0.5 小时内配送到

将时间离散化，每隔 30 分钟，所有的卸货点会生成订单（0-3 个订单），同时每隔 30 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

需要注意的是，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：一天时间内 $(24hours)$，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需求。

假设条件：

1. 无人机一次最多只能携带 10 个物品
2. 无人机一次飞行最远路程为 $20km$（无人机送完货后需要返回配送点）
3. 无人机的速度为 $60km/h$
4. 配送中心的无人机数量无限
5. 无人机一次最多只能携带 10 个物品

1.2 数据生成

1.2.1 地图生成

首先，考虑到每个无人机的最大飞行距离为 $20km$。所以每个配送中心的实际有效服务区域是一个以其坐标为圆心，半径为 $10km$ 的圆。在建立配送中心时需要每一个配送中心尽可能涵盖更多的服务范围，多个配送中心可以尽可能涵盖整个地图，同时各个配送中心之间服务区域的交集尽可能小以减少配送资源的浪费。综上所述，我们定义地图的尺寸为 $40km \times 40km$，并且定义配送中心数量为 5，分别在地图四个角与四角相切和中心确定性生成，确保尽可能覆盖整个配送区域。

其次，为了确保地图一定有解，需要保证对于任意卸货点，至少存在一个配送中心与该卸货点之间的距离小于无人机单次飞行最大距离的一半（需要往返）$10km$。所以考虑对每个配送中心的有效服务区域的圆中随机生成卸货点。共 60 个卸货点，平均每个配送中心可负责 12 个卸货点的订单配送。

生成的地图如图1。

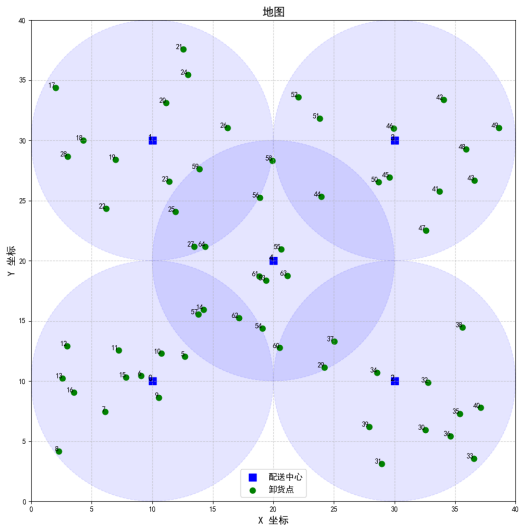

图 1: 地图生成

在生成地图时，还需要预先计算所有点对（配送中心--卸货点、卸货点--卸货点）之间的距离，存储在一个二维列表 $distance\_matrix$ 中，这是方便后续计算路径长度时不再需要重新计算两点之间的距离，而是可以直接查表获得距离，可以避免多次重复计算相同点对之间的距离，大大降低后续的计算复杂度。地图生成部分代码如下：

```
# 随机生成地图，预计算所有点对的距离矩阵，空间换时间，降低时间复杂度

```

```python
def generate_map(num_centers, num_points, map_size, max_distance):
    # 在地图的四个角和中心生成配送中心
    centers = [
        (0, (max_distance/2, max_distance/2)),
        (1, (max_distance/2, map_size-max_distance/2)),
        (2, (map_size-max_distance/2, max_distance/2)),
        (3, (map_size-max_distance/2, map_size-max_distance/2)),
        (4, (map_size / 2, map_size / 2))
    ]

    points = []
    point_id = num_centers # 编号从配送中心之后开始

    for center in centers:
        center_id, center_coord = center
        num_points_per_center = num_points // num_centers #
            平均分配给每个配送中心的卸货点数量
        for _ in range(num_points_per_center):
            while True:
                # 在配送中心为圆心，最大飞行距离的一半为半径的范围内生成卸货点
                angle = random.uniform(0, 2* np.pi)
                radius = random.uniform(1, max_distance / 2)
                point_x = center_coord[0] + radius * np.cos(angle)
                point_y = center_coord[1] + radius * np.sin(angle)

                # 确保生成的点在地图范围内
                if 0<= point_x <= map_size and 0<= point_y <= map_size:
                    point = (point_id, (point_x, point_y))
                    points.append(point)
                    point_id += 1
                    break

    # 预计算所有点对之间的距离
    all_points = centers + points
```

```
    num_all_points = len(all_points)
    distance_matrix = np.zeros((num_all_points, num_all_points))

    for i in range(num_all_points):
        for j in range(i + 1, num_all_points):
            point1 = all_points[i][1]
            point2 = all_points[j][1]
            distance = math.sqrt((point1[0] - point2[0]) ** 2+ (point1[1] -
                point2[1]) ** 2)
            distance_matrix[i][j] = distance
            distance_matrix[j][i] = distance

    return centers, points, distance_matrix
```

1.2.2 订单生成

根据题目要求，每隔 30 分钟生成一次订单，每个卸货点随机生成 0-3 个订单，并且每个订单随机分配“一般”、“较紧急”、“紧急”的优先级，并根据订单的优先级属性和订单生成时间，计算其配送截止时间。具体代码实现如下：

```
# 生成订单函数，使用模拟时间
def generate_orders(current_time, points):
    orders = []
    for point in points:
        num_orders = random.randint(0, 3)
        for _ in range(num_orders):
            priority = random.choice(['一般', '较紧急', '紧急'])
            order_time = current_time
            if priority == '一般':
                deadline = current_time + 180# 3小时
            elif priority == '较紧急':
                deadline = current_time + 90# 1.5小时
            else: # '紧急'
                deadline = current_time + 30# 30分钟
```

```
        orders.append((point, priority, order_time, deadline)) # 添加订单信息
    return orders
```

1.3 环境介绍

参数	值
处理器 (CPU)	13th Gen Intel(R) Core(TM) i9-13980HX 2.20 GHz
内存 (RAM)	32 GB DDR5
操作系统	Windows 11 专业版 64-bit
编程语言	Python 3.11.9
开发环境	Anaconda
IDE	PyCharm

表 1: 实验环境详细参数

二、 问题建模

该算法题目可以抽象为一个带时间窗和容量约束的无人机路径优化问题，需要在一定时间范围内(订单生成时间---订单截止时间的时间窗)，在满足约束条件（订单数量需求、无人机载重、无人机单次飞行最大距离）和目标最优化（无人机飞行总路程最短）的前提下，将客户的订单需求从配送中心送达到卸货点，并返回到配送中心。

2.1 参数定义

- n： 配送中心的数量。
- m： 卸货点的数量。
- k： 无人机的数量。
- D： 最大飞行距离（20 公里）。

- v：无人机飞行速度（60 公里/小时）。
- C：无人机的最大载重量（10 个物品）。
- O：订单集，每个订单包括位置、优先级、订单生成时间和截止时间。
- T：时间间隔（30 分钟）。
- P：优先级集，包括一般、较紧急、紧急。

2.2 变量定义

- x_{ij}：变量，表示是否从点 i 飞行到点 j （0 或 1）。
- y_{ij}：变量，表示是否将订单从配送中心 i 送到卸货点 j （0 或 1）。
- d_{ij}：点 i 和点 j 之间的距离。
- t_{ij}：从点 i 飞行到点 j 所需的时间。

2.3 优化模型

目标函数：最小化所有无人机的总飞行距离：

$$\min \sum_{i=1}^{n} \sum_{j=1}^{m} d_{ij} x_{ij}$$

约束条件：

- 无人机只能从配送中心出发，且数量不超过总数：

$$\sum_{j=1}^{m} y_{ij} \leq k \quad \forall i \in \{1, \ldots, n\}$$

- 每个订单需要在规定的时间内送达，满足优先级要求：

$$t_{ij} \leq \text{Deadline}_p \quad \forall p \in P, \forall (i,j) \in O$$

- 无人机的载重量不能超过最大载重量：

$$\sum_{j=1}^{m} \text{weight}(O_j) y_{ij} \leq C \quad \forall i \in \{1, \ldots, n\}$$

- 无人机单次路径长度不能超过最大飞行距离:

$$\sum_{j=1}^{m} d_{ij} y_{ij} \leq D \quad \forall i \in \{1, \ldots, n\}$$

- 每个订单只能被一个无人机配送:

$$\sum_{i=1}^{n} y_{ij} \leq 1 \quad \forall j \in \{1, \ldots, m\}$$

- 卸货点流量平衡，进出无人机数量相等:

$$\sum_{j=1}^{m} x_{ij} = \sum_{j=1}^{m} x_{ji} \quad \forall i \in \{1, \ldots, n\}$$

- 禁止自循环，无人机不会再同一个卸货点循环:

$$x_{ii} = 0 \quad \forall i \in \{1, \ldots, n\}$$

- 所有无人机都必须从配送中心出发，并最终返回配送中心:

$$x_{ij} = 1, \quad x_{ji} = 1 \quad \forall i \in \{1, \ldots, n\} \quad j \in \{1, \ldots, m\}$$

三、 算法设计

根据建模得到的单目标优化模型，可以考虑使用 0-1 整数规划模型进行求解，但是整数规划算法求解过难，复杂度过高，所以考虑使用遗传算法这一启发式算法来进行求解。

启发式算法是一类通过利用经验或者问题的特定结构来设计近似解决方案的方法。它们通常用于解决复杂的优化问题，其中传统的精确穷举方法难以应对大规模数据或者高计算复杂度。启发式算法不同于穷举法的搜索算法，不会枚举所有的方案，而是仅仅考察少量的方案就得出结果。启发式算法不保证找到全局最优解，但能在合理的时间内找到较优解。

遗传算法是一种受自然选择和遗传机制启发而来的优化算法。它模拟了生物进化过程中的基因遗传、突变和适应度选择的过程。通过维护一个种群，遗传算法通过基因重组（交叉）和突变操作来生成新的解，并通过适应度评估选择出最优解。这种算法适用于复杂的搜索空间和多模态优化问题。

综合上述原因，选择使用遗传算法对该无人机路径优化问题的近似最优解进行求解。

3.1 种群初始化

种群初始化是遗传算法中的关键步骤，它决定了算法初始解的质量和后续优化的方向。一个种群中包含多个个体，每一个个体都代表规划问题的一个可行解，因此我们需要首先确定个体（染色体）的编码方式，这代表了解的表示形式。

3.1.1 个体染色体编码

在一次订单生成后的系统决策中，笔者考虑**将所有无人机的路径集合编码为染色体**，即由一个二维列表组成，该列表中的每个元素都是一个列表，表示一个无人机的配送路径，该路径由一系列节点的编号组成，首尾是该无人机所属的配送中心的编号，路径中间依次经过不同的卸货点，将所有节点的节点编号顺序视为染色体的基因序列。

每一个节点（配送中心和卸货点）除了其坐标信息外，还应该含有其在本次路径决策中需要配送的订单数量（会消耗等额的无人机载重）信息。该信息由 $order\_counts$ 列表记录，具体代码实现如下：

```python
def maintain_order_counts(centers, points, orders):
    # 初始化订单数量数组，配送中心的订单数量为0
    num_all_points = len(centers) + len(points)
    order_counts = [0] * num_all_points

    # 记录每个点的订单数量
    for order in orders:
        point_id = order[0][0]
        order_counts[point_id] += 1

    return order_counts
```

本文没有考虑前文优化模型章节中设计的遍历方式的染色体编码方案，因为这种形式的编码方式数据结构复杂、且含有大量的冗余信息，不方便遗传算法的后续求解。

3.1.2 个体生成

个体生成过程在遗传算法中负责初始化种群，并确保种群中每个个体都是一个有效的解。笔者考虑生成一个为空的个体（一定不是可行解），并对该个体进行修复和优化从而初始化得到一个优质的可行解。因此该步骤的重点在于对个体的修复。

个体修复函数 ($individual\_fix$) 需要确保该个体所代表的解符合下面三个约束条件：

- 路径长度小于无人机最大飞行距离、且各个卸货点的订单数量和小于无人机的最大载货量，若不符合该约束，需要对路径进行裁剪。
- 所有在本次决策中有订单需求的卸货点都需要被访问一次，若存在订单数量不为 0 的卸货点没有被访问，则需要安排无人机进行配送。
- 确保个体中的所有路径都是合法的完整路径且不重复。

下面我将就这三个约束条件分别进行介绍

路径裁剪 首先，需要检查个体中的每条路径是否超出了无人机的最大飞行距离或最大载货量。如果超出限制，则通过移除路径中最靠后的卸货点，使得路径长度和订单量都符合约束条件，这一部分功能的代码实现如下：

```
def individual_fix(individual, order_counts):
    # 移除超出最大飞行距离或最大载货量的部分
    for path in individual:
        if len(path) > 2:
            path_distance = sum(distance_matrix[path[i][0]][path[i+1][0]] for i
                in range(len(path) - 1))
            path_orders = sum(order_counts[point[0]] for point in path)
            if path_distance > max_distance or path_orders > n: # 超出限制
                excess_distance = path_distance - max_distance
                excess_orders = path_orders - n
                while (excess_distance > 0or excess_orders > 0) and len(path) > 2:
                    # 循环删除节点直至路径合法
                    second_last_point = path[-2]
                    path.remove(second_last_point)
                    excess_distance -=
                        distance_matrix[second_last_point[0]][path[-1][0]]
                    excess_orders -= order_counts[second_last_point[0]]
```

路径规划 接下来需要对所有存在货物配送需求（节点订单数量大于 0）但不存在于个体编码的所有路径所包含的节点集合中的节点（没有被编入到个体路径中被无人机配送）进行路径规划，对于一个卸货点，本文决策由距离该卸货点最近的配送中心进行配送，这是因为如果选择相对较远距离的配送中心进行配送，那么最后这条路径的长度显然大于选择距离最近的配送中心所形成的路径长度（前提是

题目要求一架无人机最终一定会返回其所属的配送中心，不可以从一个配送中心出发，返回到另一个配送中心），一定不是最优解。考虑如下近似算法进行高效的路径规划求解：

1. 统计所有没有添加到路径中的卸货点集合 S
2. 从 S 中随机取出一个卸货点，搜索所有距离其最近配送中心开始的路径，如果将节点添加到路径中依旧满足无人机最大飞行距离和最大载货量的约束条件，则将其添加到该路径中，随机选择可以保证个体的多样性，以便得到最优解。
3. 否则，生成一条由最近配送中心开始的新的路径，将该卸货点添加至新路径中。
4. 循环处理节点集合 S 中的所有卸货点，直到所有卸货点都有无人机配送，循环终止。

显然，该路径规划的近似算法是 **2-近似算法**，可以用较低的复杂度求解得到质量相对较高的可行解。提高后续遗传算法的求解效率。这一部分的完整代码实现如下：

```
# 检查是否有卸货点未配送
unvisited_points = [point for point in points if point not in
    sum(individual, [])]
random.shuffle(unvisited_points) #打乱所有未配送节点
new_path = [[center] for center in centers] # 为每个配送中心初始化一条路径
while unvisited_points:
   point = unvisited_points.pop(0)
   if order_counts[point[0]] == 0:
      continue
   # 找到最近的配送中心
   nearest_center = min(centers, key=lambda center:
       distance_matrix[center[0]][point[0]])
   for path in new_path:
      if path[0][0] != nearest_center[0]:
         continue
      current_distance = sum(distance_matrix[path[i][0]][path[i+1][0]] for
          i in range(len(path) - 1))
      distance_to_point = distance_matrix[path[-1][0]][point[0]]
      current_orders = sum(order_counts[point[0]] for point in path)
   # 判断加入当前点后是否超过最大距离或最大载货量
      if current_distance + distance_to_point +
          distance_matrix[point[0]][nearest_center[0]] <= max_distance and
```

```
            current_orders + order_counts[point[0]] <= n:
                path.append(point)
                break
            else:
                new_path.append([nearest_center, point]) # 开始新路径
                break
```

路径检查 最后个体修复函数需要实现对路径的合法性进行检查，并最后将合法的路径添加到个体中，主要包括以下几个方面：

1. 所有路径的首尾必须是相同的配送中心
2. 一个个体中不存在相同的两条路径（非法解可由染色体交叉产生）
3. 路径中至少包含一个卸货点（非法解可由染色体交叉或变异产生）

在确保所有路径都合法后，将新生成的路径添加到个体中。至此完成了个体的修复工作。这一部分代码实现如下：

```
    for path in new_path:
        if len(path) > 1:
            individual.append(path)
    # 确保所有路径以配送中心结束
    for path in individual:
        if path and path[-1][0] != path[0][0]:
            path.append(path[0])
    # 使用集合去重
    individual_set = {tuple(path) for path in individual}
    individual = [list(path) for path in individual_set]
    individual = [path for path in individual if len(path) > 2]
    return individual
```

综上，利用上述算法随机生成 500 个个体作为初始种群。

3.2 适应度计算

适应度函数是遗传算法中的关键部分，用于评价每个个体（即每个可能的解决方案）的质量。在本次无人机配送路径规划问题中，适应度函数的目标是尽量缩短总配送路径长度，同时避免路径长度超

出无人机的最大飞行距离和订单数量的限制。个体中所有的路径长度总和通过计算可以得到，而无人机最大飞行距离和最大载货量的限制可以通过两个惩罚项来实现，若该个体中存在超出限制的非法路径，则在分母添加一个非常大的常数作为惩罚项，可以大幅降低该个体的适应度，代表这不是一个优秀的可行解。具体来说，设 P 是个体路径集合，p_i 是个体中的第 i 条路径，$d(p_i)$ 是第 i 条路径的距离，$o(p_i)$ 是第 i 条路径的订单数量。

适应度函数 f 定义如下：

$$f(P) = \frac{1}{\sum_{i=1}^{n} d(p_i) + \text{penalty} + \epsilon}$$

其中，

$$\text{penalty} = \sum_{i=1}^{n} (\mathcal{I}[d(p_i) > D] \cdot \lambda + \mathcal{I}[o(p_i) > O] \cdot \lambda)$$

$\mathcal{I}[\cdot]$ 是指示函数，当条件为真时取值为 1，否则为 0；D 是最大飞行距离；O 是最大订单数量；λ 是惩罚值（这里设为 1000000）；ϵ 是一个很小的常数，用于避免除零错误。

适应度函数具体代码实现如下：

```
def fitness(individual):
    total_distance = 0
    penalty = 0# 添加 penalty 变量
    for path in individual:
        path_distance = 0
        path_orders = 0
        for i in range(len(path) - 1):
            path_distance += distance_matrix[path[i][0]][path[i + 1][0]]
            path_orders += order_counts[path[i][0]]
        if path_distance > max_distance:
            penalty += 1000000# 如果路径超过最大飞行距离，给予较大的惩罚
        if path_distance > n:
            penalty += 1000000# 如果订单数量超过最大载货量，给予较大的惩罚
        total_distance += path_distance # 计算路径总长度

    return 1/ (total_distance + penalty + 0.0000000000001) #
        加入惩罚项的适应度计算
```

3.3 个体选择

在遗传算法中，选择操作用于从当前种群中选择适应度较高的个体，以生成下一代个体。选择操作的目标是保留优良的基因并引入适当的多样性，从而提升种群整体的适应度。本文采用轮盘赌选择（Roulette Wheel Selection）方法实现选择操作。

轮盘赌选择方法基于个体适应度的概率分配，适应度越高的个体被选择的概率越大。具体步骤如下：

1. **计算适应度总和:** 对当前种群中的每个个体计算其适应度，并将所有个体的适应度累加得到适应度总和。
2. **计算选择概率:** 对每个个体，其选择概率等于其适应度除以适应度总和。
3. **生成随机数:** 在 0 到适应度总和之间生成一个随机数。
4. **选择个体:** 从当前种群中按照计算的选择概率依次累加个体的适应度，当累加的适应度值首次超过随机数时，选择该个体进入下一代。

下面是轮盘赌选择操作的具体代码实现：

```
def selection(population, fitnesses):
    selected_indices = np.random.choice(range(len(population)), size=2,
        p=fitnesses / fitnesses.sum(), replace=False)
    return population[selected_indices[0]], population[selected_indices[1]]
```

在这段代码中，*population* 是当前种群，*fitnesses* 是当前种群中每个个体的适应度。函数返回被选择的两个个体。选择操作的核心是保证适应度高的个体更有可能被选择，但同时保留了一定的随机性，以维持种群的多样性。轮盘赌选择方法通过适应度比例分配选择概率，使得优秀个体有更高的概率进入下一代，从而推动种群适应度的逐步提升。

3.4 部分匹配交叉

在遗传算法中，通过交叉操作将两个父代个体的基因组合生成新的子代个体，从而引入新的基因组合，增加种群的多样性。在本文中，采用部分匹配交叉（PMX, Partially Matched Crossover）方法进行染色体交叉操作。

改进的部分匹配交叉（PMX）方法 部分匹配交叉（PMX）是一种常用于解决排序问题的交叉方法，适合路径规划等问题。但是应用部分匹配交叉的前提是两个个体所表示的解经过的节点集合相同，而在本文的编码方式中，一个个体表示所有无人机的路径集合，而对于其中任意一条路径而言，其所经

过的卸货点集合大概率不相同，所以即使应用部分匹配交叉也无法保证交叉后得到的解在路径上一定合法，所以本文考虑对部分匹配交叉方法进行一点改进，具体来说，将部分匹配交叉操作的应用对象由两个个体本身改为其中经过卸货点完全相同的路径，对于两个节点集合相同的路径使用交叉匹配可以解决经过卸货点不同所导致的冲突，从而保证每个子代个体基因的合法性和多样性。算法流程如下：

1. **寻找经过卸货点相同的两条路径：**遍历两个父代个体的路径，寻找两个路径节点集合相同的路径进行部分匹配交叉
2. **选择交叉点：**在两个路径中随机选择两个交叉点，将路径分为三部分：左侧、中间和右侧。
3. **交换中间部分：**将两个路径在交叉点之间的基因互换，形成初始子代。
4. **修正冲突：**修正剩余部分中由于交换引起的重复基因和缺失基因，确保每个基因在路径中唯一。

部分匹配交叉操作具体代码实现如下：

```
def pmx_crossover(parent1, parent2):
    def pmx_single(parent_a, parent_b):
        size = min(len(parent_a), len(parent_b))
        child = [None] * size
        # 选择两个交叉点
        point1, point2 = sorted(random.sample(range(1, size - 1), 2))

        # 将交叉点之间的部分复制到子代
        child[point1:point2] = parent_a[point1:point2]

        # 处理交叉点之外的部分
        for i in range(point1, point2):
            if parent_b[i] not in child:
                j = i
                while point1 <= j < point2:
                    j = parent_b.index(parent_a[j])
                child[j] = parent_b[i]

        # 处理剩余位置
        for i in range(size):
            if child[i] is None:
                child[i] = parent_b[i]
```

```
23
24          return child if len(child) != 0else parent_a
25
26      if random.random() < crossover_rate: # 一定概率进行交叉
27          children1 = []
28          children2 = []
29
30          for i in range(min(len(parent1), len(parent2))):
31              parent1_nodes = set([node for node in parent1[i] if node[0] !=
                    parent1[i][0][0]])
32              parent2_nodes = set([node for node in parent2[i] if node[0] !=
                    parent2[i][0][0]])
33
34              if parent1_nodes == parent2_nodes: # 只有节点集相同时才执行交叉
35                  if len(parent1[i]) > 3and len(parent2[i]) > 3:
36                      child1 = pmx_single(parent1[i], parent2[i])
37                      child2 = pmx_single(parent2[i], parent1[i])
38                  else:
39                      child1 = parent1[i]
40                      child2 = parent2[i]
41                  if child1:
42                      children1.append(child1)
43                  if child2:
44                      children2.append(child2)
45              else:
46                  children1.append(parent1[i])
47                  children2.append(parent2[i])
48          children1 = [child for child in children1 if len(child) > 2]
49          children2 = [child for child in children2 if len(child) > 2]
50          # 确保所有路径以配送中心结束
51          for child in children1:
52              if child[-1][0] != child[0][0]:
53                  child.append(child[0])
54          for child in children2:
```

```
        if child[-1][0] != child[0][0]:
            child.append(child[0])

    return individual_fix(children1, order_counts),
        individual_fix(children2, order_counts) # 修复个体
else:
    return individual_fix(parent1, order_counts), individual_fix(parent2,
        order_counts)
```

在部分匹配交叉操作完成后，通过调用个体修复函数 $individual\_fix$ 确保孩子个体是一个可行解。

通过部分匹配交叉操作，本文实现了高效的路径基因交换和冲突修正，从而确保每个子代个体的基因合法性和多样性。在路径规划问题中，这种交叉操作能够有效探索更优路径组合，提升种群整体适应度。

3.5 变异操作

变异操作是遗传算法中的重要步骤，通过随机改变个体的基因结构来引入新的基因组合，增加种群的多样性，避免陷入局部最优解。在本文中，变异操作将按照概率对一个个体中的所有路径随机进行多种类型的变异操作，主要包括以下四种类型：路径交换、路径倒置、路径删除、以及随机选择一个点插入到另一条路径。接下来，我将分别就这四种类型进行介绍

路径节点交换 在路径节点交换变异中，从路径中随机选择两个卸货点，交换它们的位置。这一操作通过改变点的顺序来引入新的基因组合，从而找到一条更优长度的路径。具体代码实现如下：

```
def mutate(individual):
    for i, path in enumerate(individual): # 对所有路径都进行变异
        if random.random() < mutation_rate and len(path) > 2: #
            根据变异概率判断是否进行变异
            mutation_type = random.randint(1, 4) # 随机选择变异类型 (1-4)
            unloading_points = [idx for idx, point in enumerate(path) if idx != 0
                and idx != len(path) - 1]
            if mutation_type == 1and len(unloading_points) > 1: #
                路径交换，确保至少有两个卸货点
                swap_points = random.sample(unloading_points, 2)
                path[swap_points[0]], path[swap_points[1]] = path[swap_points[1]],
```

```
                path[swap_points[0]]
```

路径倒置 路径倒置变异选择一个路径的子段，并将其倒置。这一操作可以显著改变路径的结构，帮助跳出局部最优解，这一操作代码实现如下：

```
        elif mutation_type == 2and len(unloading_points) > 1: #
            路径倒置，确保至少有两个卸货点
            reverse_start = random.choice(unloading_points)
            valid_end_points = [idx for idx in unloading_points if idx >
                reverse_start]
            if valid_end_points: # 确保有合法的结束点
                reverse_end = random.choice(valid_end_points)
                path[reverse_start:reverse_end + 1] =
                    reversed(path[reverse_start:reverse_end + 1])
```

路径删除 在某些情况下，当前无人机配送路径所包含的卸货点组合可能不是最优的选择，所以考虑将整个路径删除，并在变异操作结束后调用个体修复函数 $individual\_fix$ 对该个体进行修复，来确保本次路径删除导致没有被配送的卸货点被安排至新的路径。以此来引入更加彻底的基因变化，具体代码实现如下：

```
        elif mutation_type == 3:
            individual[i] = []
```

随机选点插入到另一条路径 该变异操作在路径中随机选择一个卸货点，并将其插入到由相同配送中心负责的另一条路径中。这一操作增加了不同路径之间的基因流动，可以帮助跳出局部最优解，具体代码实现如下：

```
        elif mutation_type == 4and len(unloading_points) > 0: #
            随机选择一个点插入到另一条路径
            selected_point_index = random.choice(unloading_points)
            selected_point = path.pop(selected_point_index)

            # 找到该点所属的配送中心
            center_id = path[0][0]
```

```
            # 找到所有其他由该配送中心负责的路径
            other_paths = [p for p in individual if len(p) > 2and p[0][0] ==
                center_id and p != path]

            if other_paths:
                # 随机选择一个目标路径
                target_path = random.choice(other_paths)
                # 随机选择一个插入位置
                insert_position = random.randint(1, len(target_path) - 1)
                target_path.insert(insert_position, selected_point)
            else:
                # 如果没有其他路径，则将点插回原路径的随机位置
                insert_position = random.randint(1, len(path) - 1)
                path.insert(insert_position, selected_point)
    # 使用 individual_fix 函数修复路径
    individual = individual_fix(individual, order_counts)
    return individual
```

通过多种变异操作的组合，本算法能够有效地增加种群的多样性，避免陷入局部最优解。路径交换和路径倒置操作主要针对路径内部结构进行优化，而点插入操作则引入不同路径之间的基因流动，进一步增加了种群的多样性。

在具体实现中，变异操作通过一定的概率进行，以平衡种群的稳定性和多样性。此外，使用 $individual\_fix$ 函数对变异后的路径进行修复，确保路径的有效性和合理性，从而提高变异操作的效果和算法的整体性能。

3.6 遗传算法决策流程

前文对遗传算法求解当前所需配送订单的最优路径所需的功能函数进行了详细的描述并给出了代码实现，接下来我将介绍 $genetic\_algorithm$ 函数的流程，该函数通过调用前文所提到的组件，实现了遗传算法对一次生成订单的路径决策，首先给出详细代码实现：

```
def genetic_algorithm(centers, points, population_size, generations,
    max_distance, order_counts, n):
    population = initialize_population(centers, points, population_size,
        max_distance, order_counts, n) # 初始化种群
```

```
    best_individual = None # 初始化最佳个体为空
    best_fitness = float('-inf') # 初始化最佳适应度为负值

    for generation in range(generations):
        fitnesses = np.array([fitness(ind) for ind in population]) #
            计算适应度函数

        new_population = []
        for _ in range(population_size // 2):
            parent1, parent2 = selection(population, fitnesses) #
                轮盘赌选择两个父本个体
            child1, child2 = pmx_crossover(parent1, parent2) #
                对父本个体应用部分匹配交叉得到两个孩子个体
            mutate(child1) # 对孩子个体进行变异
            mutate(child2)
            new_population.extend([child1, child2]) # 繁育出新种群

        population = new_population

        current_best = population[np.argmax(fitnesses)]
        current_fitness = max(fitnesses)

        if current_fitness > best_fitness: # 更新最佳个体
            best_fitness = current_fitness
            best_individual = current_best
        print(f"Generation {generation + 1}: Best Fitness = {best_fitness}")

    return best_individual # 返回最优个体
```

遗传算法完整流程如下：

1. **初始种群生成：**使用 *initialize_population* 函数生成初始种群。初始种群的规模由参数 *population_size* 决定，该参数值通过定步长搜索的方案寻早最佳的迭代轮数，经过多次测试选择迭代轮次为 30 时可以兼顾决策所需的迭代时间和得到的近似解的质量，取得不错的效果。

2. **初始化最佳个体和适应度值：**初始化变量 *best_individual* 和 *best_fitness*，用于记录当前最优

的个体及其适应度值。

3. **主循环（迭代）：**主循环迭代次数由参数 $generations$ 决定。在每一代中，进行以下步骤：
   - **适应度计算:** 计算种群中每个个体的适应度值，适应度函数 $fitness$ 用来衡量个体的优劣。
   - **新种群生成:** 通过选择、交叉和变异操作生成新的种群。
   - **更新最佳个体:** 计算当前种群中最优个体的适应度值，并更新 $best\_individual$ 和 $best\_fitness$，如果当前最优个体优于之前的最优个体。
4. **返回结果:** 循环结束后，返回最优个体及其适应度值。

上述代码实现的遗传算法通过迭代选择、交叉和变异等操作，不断优化种群中的个体，寻找单次生成订单下的无人机路径优化的最优解。在整个算法过程中，通过多样化的变异操作、部分匹配交叉以及轮盘赌选择操作，避免了种群陷入局部最优解，提高了全局搜索能力。

3.7 订单决策

前文介绍的遗传算法可以求解一次订单生成时无人机的最佳路径规划问题，题目要求需要对一天时间内每个时间段生成的订单都进行决策，那么订单的分配决策就十分重要了，我们需要决策每次生成的订单中哪些订单需要在当次配送，哪些订单可以暂时不配送，等后续订单生成后一起配送。显而易见的，每一次生成的订单都在本次决策中配送出去的方案是不能得到很好的近似解的，这是因为对一天的订单配送进行路径规划是一个典型的在线问题，如果能够将优先级不是很高的订单拖延一会儿，随后续的订单一起配送，可以使后续的算法求解过程所能决策的可行解空间增大，也就更有可能获得更好的近似解。极端的说，如果可以将所有订单都推迟到最后一次决策进行处理，那么这个算法就变成了离线算法，理论上可以求出最优解。综上所述，本文考虑根据生成订单的优先级来判断该订单是否需要在本次决策配送。

基于优先级的订单决策 具体来说，本文考虑在全局维护一个待配送订单列表，每隔 30 分钟生成一次订单后，根据订单的截止配送时间和当前时间来判断订单的紧急程度，若订单处于“紧急”或“较紧急”状态，则在本次进行配送，否则将订单添加到待配送订单列表中，以供后续轮次的决策进行配送。同时，每一次生成新的订单后，也需要对待配送订单列表中的订单进行紧急程度的更新，若列表中的订单紧急程度更新为“紧急”或“较紧急”状态，也需要在这一次决策中进行配送。此外，需要注意的是，若当前时间是一天中最后一次进行系统决策，则不管订单处于什么状态，都将通过遗传算法进行路径规划，从而将这一整天的所有订单都配送完成。详细的代码实现如下：

```
def process_orders(new_orders, remaining_orders, current_time):
    current_to_deliver = []
```

```
    updated_remaining_orders = [] # 复制剩余订单列表，防止直接修改原始列表

    # 将剩余订单中的紧急和较紧急订单加入到当前配送列表,其余的加入待配送列表
    for order in remaining_orders:
        point, priority, order_time, deadline = order
        time_left = deadline - current_time
        urgency_level = calculate_urgency(time_left)
        if urgency_level in ['紧急', '较紧急']:
            current_to_deliver.append(order)
        else:
            updated_remaining_orders.append(order)

    # 根据紧急程度决定本次配送和更新剩余订单列表
    for order in new_orders:
        point, priority, order_time, deadline = order
        time_left = deadline - current_time
        urgency_level = calculate_urgency(time_left)
        if urgency_level in ['紧急', '较紧急']:
            current_to_deliver.append(order)
        else:
            updated_remaining_orders.append(order)
    if current_time == time_limit - t:
        current_to_deliver.extend(updated_remaining_orders)
    return current_to_deliver, updated_remaining_orders


def calculate_urgency(time_left):
    # 根据截止时间剩余的时间长度计算紧急程度
    if time_left <= 30:
        return '紧急'
    elif time_left <= 90:
        return '较紧急'
    else:
        return '一般'
```

上述订单决策方案可以优先处理紧急订单，将非紧急订单留到后续进行更精确的最优解决策，从而提高无人机配送效率，更容易找到更好的解。

3.8 算法主循环

主循环用于模拟无人机在一天中的多次配送决策和路径规划。通过结合订单生成、处理、遗传算法路径规划，确保无人机能够高效地完成配送任务。算法完整流程如下：

1. **加载地图数据：** 加载之前生成并保存的地图数据，包括配送中心、卸货点和距离矩阵。
2. **初始化变量：** 初始化配送中心数量、卸货点数量、无人机最大飞行距离、遗传算法种群大小等参数的值，以及订单列表、总配送路径长度和待处理订单列表等变量的值。
3. **主循环：** 主循环遍历一天中的多个时间段，间隔为 $t = 30$ 分钟。在每一个时间间隔中，进行以下步骤：
   - **生成当前时间段的订单：** 根据当前时间生成新的订单。
   - **订单决策：** 将新生成的订单和上次未配送的订单进行处理，得到当前需要配送的订单和更新后的待处理订单列表。
   - **计算每个卸货点订单数量统计：** 统计每个卸货点的订单数量，无人机经过该卸货点时会消耗等量大小的载重。
   - **运行遗传算法优化路径规划：** 使用遗传算法进行路径规划，得到当前时间段的无人机最优配送路径。
   - **计算当前最佳路径长度并累加至总路径长度：** 输出当前时间段的最优路径，并计算路径长度和决策时间等信息，绘制最优路径图。
4. **返回结果：** 主循环结束后，打印总路径长度。

主循环部分的代码实现如下：

```
# 生成地图
# centers, points, distance_matrix = generate_map(num_centers, num_points,
    map_size, max_distance) # 随机生成地图
# save_map(centers, points, distance_matrix, 'map_data.pkl') # 保存地图文件
centers, points, distance_matrix = load_map('map_data.pkl')
orders = []
total_daily_distance = 0
todo_list = []
```

```
for current_time in range(0, time_limit, t):
    # 生成当前时间段的订单
    new_orders = generate_orders(current_time, points)
    start_time = time.time() # 获取开始时间
    current_orders, todo_list = process_orders(new_orders, todo_list,
        current_time)
    # 计算订单数量统计
    order_counts = maintain_order_counts(centers, points, current_orders)
    print(f"当前时间:  {current_time} order_counts: {order_counts}")
    # 运行遗传算法优化路径规划
    best_result = genetic_algorithm(centers, points, population_size,
        generations, max_distance, order_counts, n)
    # 获取结束时间
    end_time = time.time()
    use_time = end_time - start_time
    print(f"决策用时: {use_time:.2f}s")
    current_best_distance = sum(distance_matrix[path[i][0]][path[i + 1][0]] for
        path in best_result for i in range(len(path) - 1))
    total_daily_distance += current_best_distance
    # 输出最优路径
    hours = current_time // 60
    minutes = current_time % 60
    print(f"当前时间: {hours} 小时 {minutes} 分钟")
    print_best_individual(best_result, current_orders)
    # 绘制最佳路径
    plot_map(centers, points, best_result, max_distance)

print("所有时间段订单处理完毕。")
print(f"总路径长度: {total_daily_distance:.2f}")
```

通过上述流程，模拟了无人机在一天中多个时间段的配送决策和路径规划过程。主循环在每个时间段生成新的订单，并结合上次未配送的订单进行处理，使用遗传算法优化路径规划。整个过程确保了无人机能够高效地完成配送任务，最大限度地减少配送路径长度，提高配送效率，最终完整求解无人机配送路径规划问题，得到近似最优解。

四、 问题求解

4.1 参数初始化

```
# 参数
num_centers = 5    # 配送中心数量
num_points = 60    # 卸货点数量
map_size = 40      # 地图尺寸 单位为公里
t = 30             # 时间段间隔，单位为分钟
n = 10             # 无人机最大可承载订单数量
max_distance = 20  # 无人机最大飞行距离，单位为公里
speed = 60         # 无人机飞行速度，单位为公里/小时
time_limit = 24* 60# 一天的时间限制，单位为分钟（24小时）
population_size = 500# 遗传算法种群规模
generations = 30   # 遗传算法运行的迭代次数
mutation_rate = 0.3 # 变异率
crossover_rate = 0.9 # 交叉率
```

4.2 地图初始化

生成的地图如图2。

4.3 部分决策方案和对应的路径规划图

在一天的时间内，每隔半小时生成一次订单并进行决策，共计需要 48 次决策，将所有的决策方案和对应的路径图展示出来过于冗余，于是只截取个别的决策进行展示，详细的结果可以直接运行本文附录中的代码即可得到。

0：00 决策 可以看到由于是第一次生成订单，所有所有被安排配送的订单都是优先级为紧急或较紧急的订单，优先级一般的订单留到后续决策轮次进行配送。本轮决策用时 11.02s

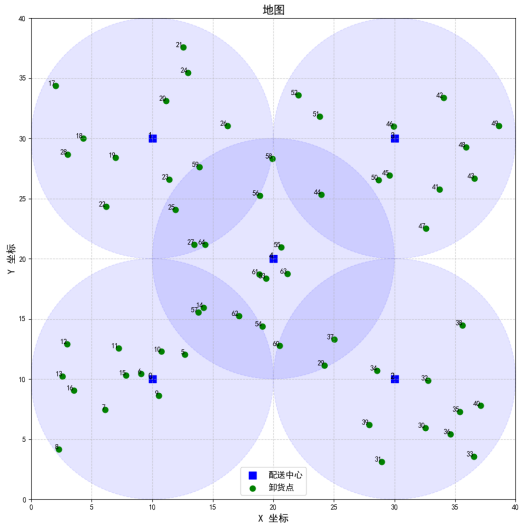

图 2: 地图初始化

```
决策用时: 11.02s
当前时间: 0 小时 0 分钟
最优路径和路径长度:
路径: 3 -> 41 -> 3, 长度: 11.31
  点 41 的订单 - 优先级: 较紧急, 下单时间: 0 小时 0 分钟, 截止时间: 1 小时 30
  点 41 的订单 - 优先级: 紧急, 下单时间: 0 小时 0 分钟, 截止时间: 0 小时 30
路径: 3 -> 50 -> 42 -> 3, 长度: 17.69
  点 50 的订单 - 优先级: 较紧急, 下单时间: 0 小时 0 分钟, 截止时间: 1 小时 30
  点 42 的订单 - 优先级: 紧急, 下单时间: 0 小时 0 分钟, 截止时间: 0 小时 30
  点 42 的订单 - 优先级: 紧急, 下单时间: 0 小时 0 分钟, 截止时间: 0 小时 30
  点 42 的订单 - 优先级: 较紧急, 下单时间: 0 小时 0 分钟, 截止时间: 1 小时 30
路径: 3 -> 49 -> 3, 长度: 17.39
  点 49 的订单 - 优先级: 紧急, 下单时间: 0 小时 0 分钟, 截止时间: 0 小时 30
  点 49 的订单 - 优先级: 紧急, 下单时间: 0 小时 0 分钟, 截止时间: 0 小时 30
  点 49 的订单 - 优先级: 较紧急, 下单时间: 0 小时 0 分钟, 截止时间: 1 小时 30
路径: 4 -> 14 -> 4, 长度: 14.16
  点 14 的订单 - 优先级: 紧急, 下单时间: 0 小时 0 分钟, 截止时间: 0 小时 30
  点 14 的订单 - 优先级: 紧急, 下单时间: 0 小时 0 分钟, 截止时间: 0 小时 30
路径: 1 -> 18 -> 17 -> 1, 长度: 19.67
  点 18 的订单 - 优先级: 较紧急, 下单时间: 0 小时 0 分钟, 截止时间: 1 小时 30
  点 18 的订单 - 优先级: 较紧急, 下单时间: 0 小时 0 分钟, 截止时间: 1 小时 30
  点 18 的订单 - 优先级: 较紧急, 下单时间: 0 小时 0 分钟, 截止时间: 1 小时 30
  点 17 的订单 - 优先级: 紧急, 下单时间: 0 小时 0 分钟, 截止时间: 0 小时 30
  点 17 的订单 - 优先级: 较紧急, 下单时间: 0 小时 0 分钟, 截止时间: 1 小时 30
路径: 0 -> 13 -> 10 -> 0, 长度: 18.19
  点 13 的订单 - 优先级: 紧急, 下单时间: 0 小时 0 分钟, 截止时间: 0 小时 30
  点 13 的订单 - 优先级: 紧急, 下单时间: 0 小时 0 分钟, 截止时间: 0 小时 30
  点 10 的订单 - 优先级: 紧急, 下单时间: 0 小时 0 分钟, 截止时间: 0 小时 30
```

图 3: 配送方案

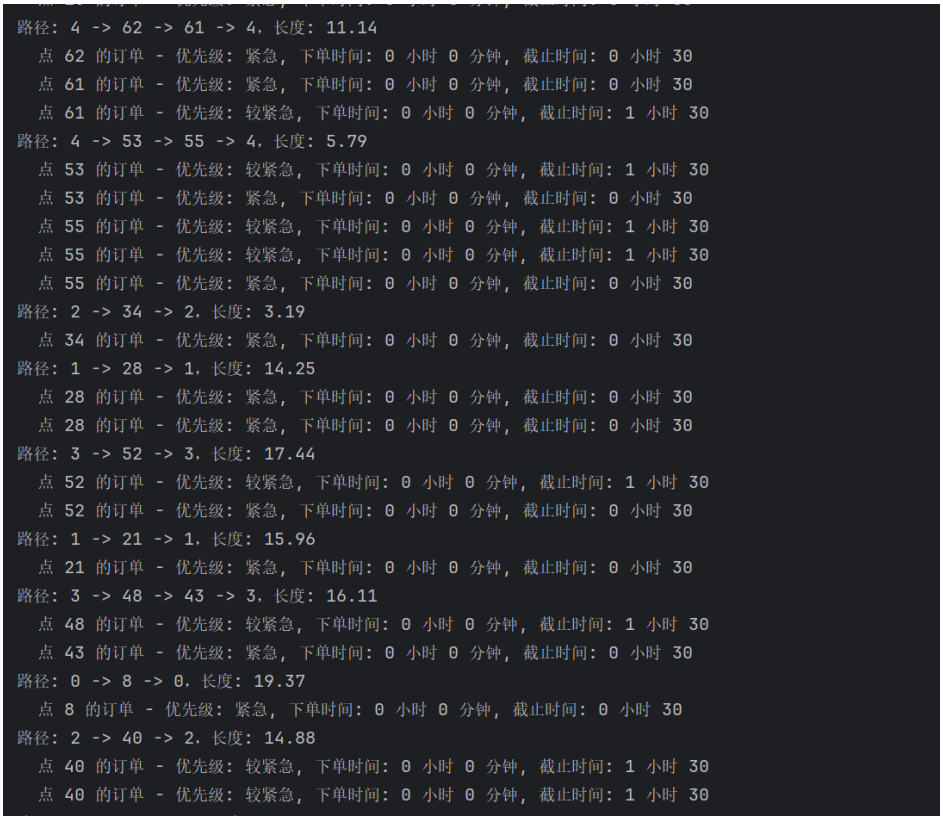
路径: 4 -> 62 -> 61 -> 4, 长度: 11.14
点 62 的订单 - 优先级: 紧急, 下单时间: 0 小时 0 分钟, 截止时间: 0 小时 30
点 61 的订单 - 优先级: 紧急, 下单时间: 0 小时 0 分钟, 截止时间: 0 小时 30
点 61 的订单 - 优先级: 较紧急, 下单时间: 0 小时 0 分钟, 截止时间: 1 小时 30
路径: 4 -> 53 -> 55 -> 4, 长度: 5.79
点 53 的订单 - 优先级: 较紧急, 下单时间: 0 小时 0 分钟, 截止时间: 1 小时 30
点 53 的订单 - 优先级: 紧急, 下单时间: 0 小时 0 分钟, 截止时间: 0 小时 30
点 55 的订单 - 优先级: 较紧急, 下单时间: 0 小时 0 分钟, 截止时间: 1 小时 30
点 55 的订单 - 优先级: 较紧急, 下单时间: 0 小时 0 分钟, 截止时间: 1 小时 30
点 55 的订单 - 优先级: 紧急, 下单时间: 0 小时 0 分钟, 截止时间: 0 小时 30
路径: 2 -> 34 -> 2, 长度: 3.19
点 34 的订单 - 优先级: 紧急, 下单时间: 0 小时 0 分钟, 截止时间: 0 小时 30
路径: 1 -> 28 -> 1, 长度: 14.25
点 28 的订单 - 优先级: 紧急, 下单时间: 0 小时 0 分钟, 截止时间: 0 小时 30
点 28 的订单 - 优先级: 紧急, 下单时间: 0 小时 0 分钟, 截止时间: 0 小时 30
路径: 3 -> 52 -> 3, 长度: 17.44
点 52 的订单 - 优先级: 较紧急, 下单时间: 0 小时 0 分钟, 截止时间: 1 小时 30
点 52 的订单 - 优先级: 紧急, 下单时间: 0 小时 0 分钟, 截止时间: 0 小时 30
路径: 1 -> 21 -> 1, 长度: 15.96
点 21 的订单 - 优先级: 紧急, 下单时间: 0 小时 0 分钟, 截止时间: 0 小时 30
路径: 3 -> 48 -> 43 -> 3, 长度: 16.11
点 48 的订单 - 优先级: 较紧急, 下单时间: 0 小时 0 分钟, 截止时间: 1 小时 30
点 43 的订单 - 优先级: 紧急, 下单时间: 0 小时 0 分钟, 截止时间: 0 小时 30
路径: 0 -> 8 -> 0, 长度: 19.37
点 8 的订单 - 优先级: 紧急, 下单时间: 0 小时 0 分钟, 截止时间: 0 小时 30
路径: 2 -> 40 -> 2, 长度: 14.88
点 40 的订单 - 优先级: 较紧急, 下单时间: 0 小时 0 分钟, 截止时间: 1 小时 30
点 40 的订单 - 优先级: 较紧急, 下单时间: 0 小时 0 分钟, 截止时间: 1 小时 30

图 4: 配送方案

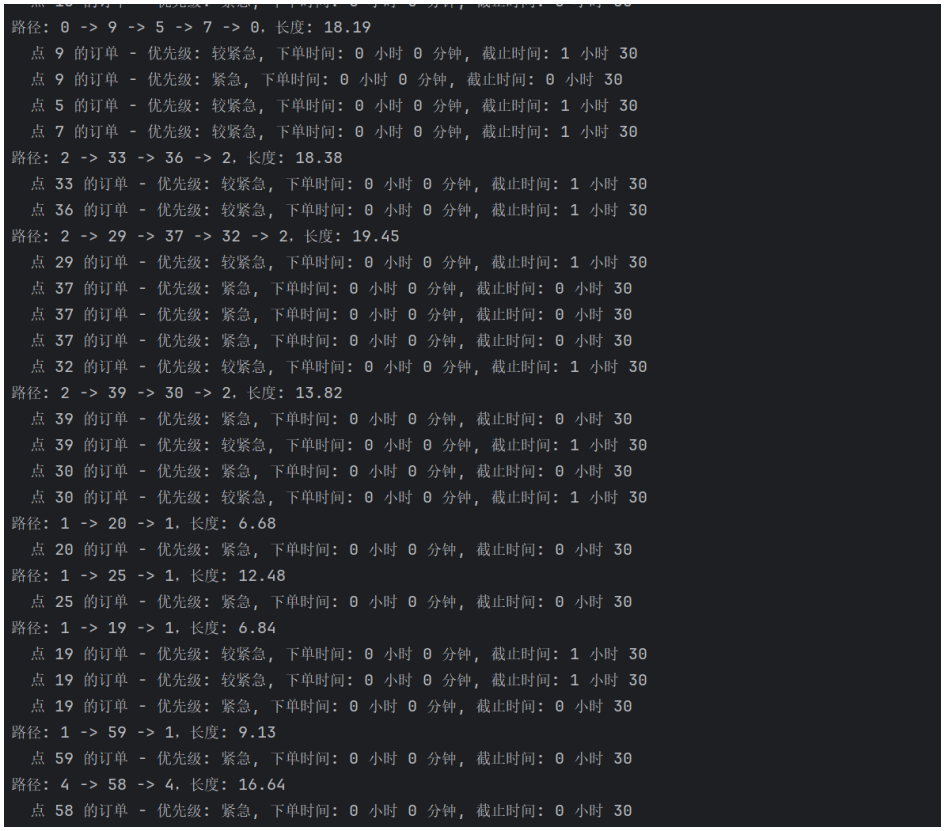
路径: 0 -> 9 -> 5 -> 7 -> 0, 长度: 18.19
点 9 的订单 - 优先级: 较紧急, 下单时间: 0 小时 0 分钟, 截止时间: 1 小时 30
点 9 的订单 - 优先级: 紧急, 下单时间: 0 小时 0 分钟, 截止时间: 0 小时 30
点 5 的订单 - 优先级: 较紧急, 下单时间: 0 小时 0 分钟, 截止时间: 1 小时 30
点 7 的订单 - 优先级: 较紧急, 下单时间: 0 小时 0 分钟, 截止时间: 1 小时 30
路径: 2 -> 33 -> 36 -> 2, 长度: 18.38
点 33 的订单 - 优先级: 较紧急, 下单时间: 0 小时 0 分钟, 截止时间: 1 小时 30
点 36 的订单 - 优先级: 较紧急, 下单时间: 0 小时 0 分钟, 截止时间: 1 小时 30
路径: 2 -> 29 -> 37 -> 32 -> 2, 长度: 19.45
点 29 的订单 - 优先级: 较紧急, 下单时间: 0 小时 0 分钟, 截止时间: 1 小时 30
点 37 的订单 - 优先级: 紧急, 下单时间: 0 小时 0 分钟, 截止时间: 0 小时 30
点 37 的订单 - 优先级: 紧急, 下单时间: 0 小时 0 分钟, 截止时间: 0 小时 30
点 37 的订单 - 优先级: 紧急, 下单时间: 0 小时 0 分钟, 截止时间: 0 小时 30
点 32 的订单 - 优先级: 较紧急, 下单时间: 0 小时 0 分钟, 截止时间: 1 小时 30
路径: 2 -> 39 -> 30 -> 2, 长度: 13.82
点 39 的订单 - 优先级: 紧急, 下单时间: 0 小时 0 分钟, 截止时间: 0 小时 30
点 39 的订单 - 优先级: 较紧急, 下单时间: 0 小时 0 分钟, 截止时间: 1 小时 30
点 30 的订单 - 优先级: 紧急, 下单时间: 0 小时 0 分钟, 截止时间: 0 小时 30
点 30 的订单 - 优先级: 较紧急, 下单时间: 0 小时 0 分钟, 截止时间: 1 小时 30
路径: 1 -> 20 -> 1, 长度: 6.68
点 20 的订单 - 优先级: 紧急, 下单时间: 0 小时 0 分钟, 截止时间: 0 小时 30
路径: 1 -> 25 -> 1, 长度: 12.48
点 25 的订单 - 优先级: 紧急, 下单时间: 0 小时 0 分钟, 截止时间: 0 小时 30
路径: 1 -> 19 -> 1, 长度: 6.84
点 19 的订单 - 优先级: 较紧急, 下单时间: 0 小时 0 分钟, 截止时间: 1 小时 30
点 19 的订单 - 优先级: 较紧急, 下单时间: 0 小时 0 分钟, 截止时间: 1 小时 30
点 19 的订单 - 优先级: 紧急, 下单时间: 0 小时 0 分钟, 截止时间: 0 小时 30
路径: 1 -> 59 -> 1, 长度: 9.13
点 59 的订单 - 优先级: 紧急, 下单时间: 0 小时 0 分钟, 截止时间: 0 小时 30
路径: 4 -> 58 -> 4, 长度: 16.64
点 58 的订单 - 优先级: 紧急, 下单时间: 0 小时 0 分钟, 截止时间: 0 小时 30

图 5: 配送方案

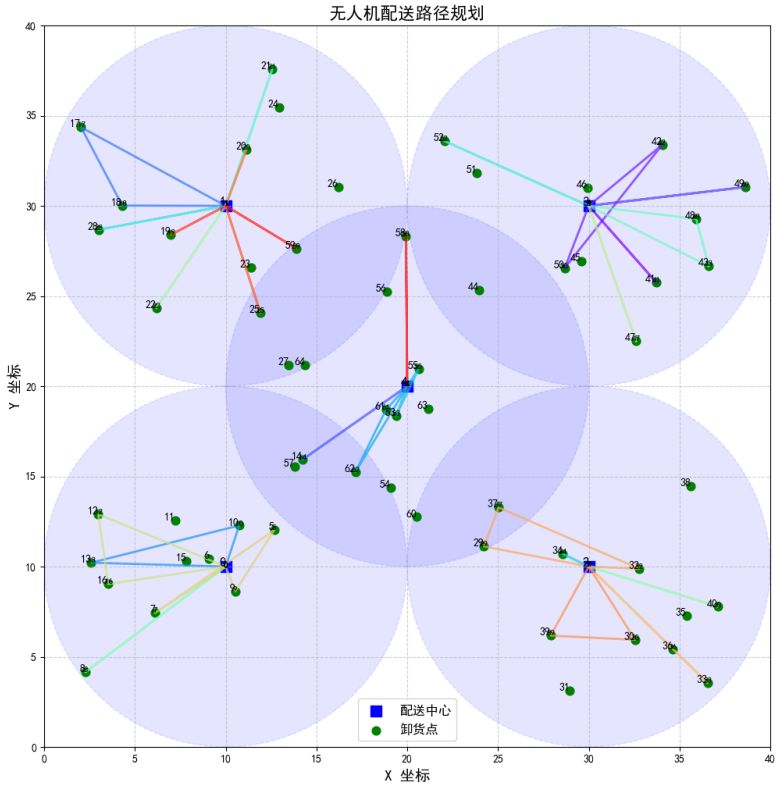

图 6: 0：00 路径规划图

23：30 决策 最后一次决策，需要配送所有还没有配送的订单，下图是部分配送方案以及总的路径规划图，本次决策用时 18.66s

```
决策用时: 18.66s
当前时间: 23 小时 30 分钟
最优路径和路径长度:
路径: 2 -> 34 -> 2, 长度: 3.19
  点 34 的订单 - 优先级: 紧急, 下单时间: 23 小时 30 分钟, 截止时间: 24 小时 0
  点 34 的订单 - 优先级: 一般, 下单时间: 22 小时 30 分钟, 截止时间: 25 小时 30
  点 34 的订单 - 优先级: 一般, 下单时间: 23 小时 0 分钟, 截止时间: 26 小时 0
  点 34 的订单 - 优先级: 一般, 下单时间: 23 小时 30 分钟, 截止时间: 26 小时 30
  点 34 的订单 - 优先级: 一般, 下单时间: 23 小时 30 分钟, 截止时间: 26 小时 30
路径: 0 -> 7 -> 0, 长度: 9.34
  点 7 的订单 - 优先级: 一般, 下单时间: 22 小时 0 分钟, 截止时间: 25 小时 0
  点 7 的订单 - 优先级: 较紧急, 下单时间: 23 小时 30 分钟, 截止时间: 25 小时 0
  点 7 的订单 - 优先级: 一般, 下单时间: 23 小时 30 分钟, 截止时间: 26 小时 30
  点 7 的订单 - 优先级: 一般, 下单时间: 23 小时 30 分钟, 截止时间: 26 小时 30
路径: 3 -> 41 -> 3, 长度: 11.31
  点 41 的订单 - 优先级: 一般, 下单时间: 22 小时 0 分钟, 截止时间: 25 小时 0
  点 41 的订单 - 优先级: 一般, 下单时间: 22 小时 30 分钟, 截止时间: 25 小时 30
  点 41 的订单 - 优先级: 一般, 下单时间: 23 小时 30 分钟, 截止时间: 26 小时 30
路径: 1 -> 18 -> 1, 长度: 11.38
  点 18 的订单 - 优先级: 一般, 下单时间: 22 小时 0 分钟, 截止时间: 25 小时 0
  点 18 的订单 - 优先级: 较紧急, 下单时间: 23 小时 30 分钟, 截止时间: 25 小时 0
  点 18 的订单 - 优先级: 一般, 下单时间: 22 小时 30 分钟, 截止时间: 25 小时 30
  点 18 的订单 - 优先级: 一般, 下单时间: 22 小时 30 分钟, 截止时间: 25 小时 30
  点 18 的订单 - 优先级: 一般, 下单时间: 23 小时 30 分钟, 截止时间: 26 小时 30
路径: 0 -> 15 -> 5 -> 0, 长度: 10.68
```

图 7: 配送方案

```
路径: 4 -> 62 -> 4, 长度: 11.09
  点 62 的订单 - 优先级: 较紧急, 下单时间: 23 小时 30 分钟, 截止时间: 25 小时 0
  点 62 的订单 - 优先级: 较紧急, 下单时间: 23 小时 30 分钟, 截止时间: 25 小时 0
  点 62 的订单 - 优先级: 较紧急, 下单时间: 23 小时 30 分钟, 截止时间: 25 小时 0
  点 62 的订单 - 优先级: 一般, 下单时间: 22 小时 30 分钟, 截止时间: 25 小时 30
路径: 2 -> 40 -> 2, 长度: 14.88
  点 40 的订单 - 优先级: 一般, 下单时间: 22 小时 0 分钟, 截止时间: 25 小时 0
  点 40 的订单 - 优先级: 较紧急, 下单时间: 23 小时 30 分钟, 截止时间: 25 小时 0
  点 40 的订单 - 优先级: 一般, 下单时间: 22 小时 30 分钟, 截止时间: 25 小时 30
路径: 4 -> 54 -> 64 -> 4, 长度: 19.69
  点 54 的订单 - 优先级: 紧急, 下单时间: 23 小时 30 分钟, 截止时间: 24 小时 0
  点 54 的订单 - 优先级: 较紧急, 下单时间: 23 小时 30 分钟, 截止时间: 25 小时 0
  点 54 的订单 - 优先级: 一般, 下单时间: 22 小时 30 分钟, 截止时间: 25 小时 30
  点 54 的订单 - 优先级: 一般, 下单时间: 23 小时 30 分钟, 截止时间: 26 小时 30
  点 64 的订单 - 优先级: 一般, 下单时间: 22 小时 30 分钟, 截止时间: 25 小时 30
  点 64 的订单 - 优先级: 一般, 下单时间: 23 小时 30 分钟, 截止时间: 26 小时 30
  点 64 的订单 - 优先级: 一般, 下单时间: 23 小时 30 分钟, 截止时间: 26 小时 30
路径: 1 -> 22 -> 28 -> 1, 长度: 19.34
  点 22 的订单 - 优先级: 一般, 下单时间: 23 小时 30 分钟, 截止时间: 26 小时 30
  点 22 的订单 - 优先级: 一般, 下单时间: 23 小时 30 分钟, 截止时间: 26 小时 30
  点 22 的订单 - 优先级: 一般, 下单时间: 23 小时 30 分钟, 截止时间: 26 小时 30
  点 28 的订单 - 优先级: 较紧急, 下单时间: 23 小时 30 分钟, 截止时间: 25 小时 0
  点 28 的订单 - 优先级: 一般, 下单时间: 23 小时 0 分钟, 截止时间: 26 小时 0
  点 28 的订单 - 优先级: 一般, 下单时间: 23 小时 0 分钟, 截止时间: 26 小时 0
路径: 2 -> 29 -> 2, 长度: 11.76
```

图 8: 配送方案

4.4 总路径长度

得到最终一天无人机配送路径长度总和为 21108.44km

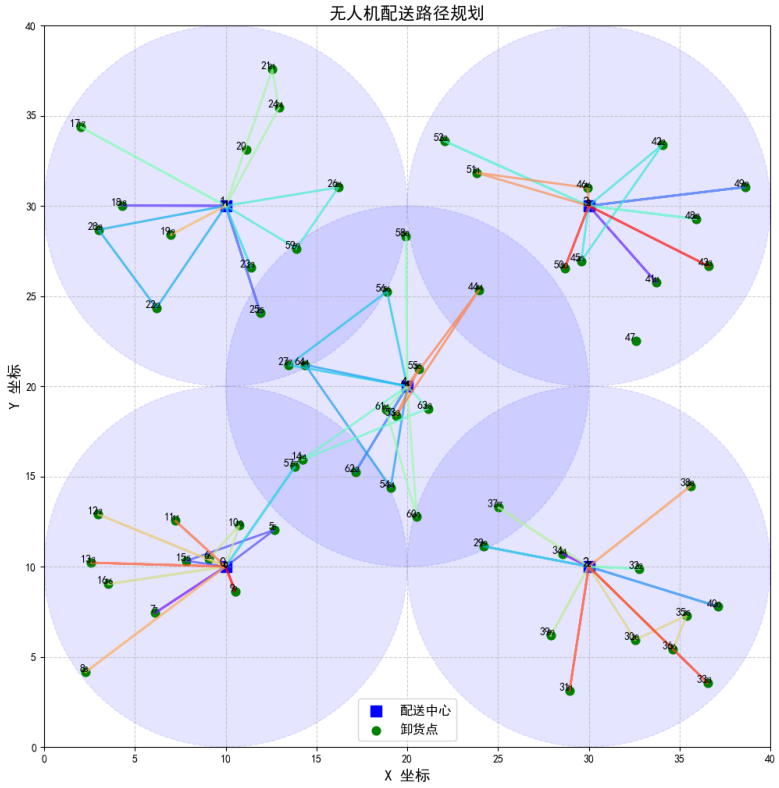

图 9: 23：30 路径规划图

图 10: 计算得到总路径长度

4.5 性能分析

在 5 个配送中心、60 个卸货点、地图大小为 $40km \times 40km$、每个卸货点在每半小时内生成 1-3 个订单的数据规模下，单轮决策用时平均为 11.32 秒，最终总路径长度为 $21108.44km$。综合考量决策用时和得到最优解的质量来看，基于遗传算法的无人机路径规划求解方案可以取得相对不错的效果。

五、 总结

本文提出了一个基于遗传算法优化无人机配送路径的方案，在算法的设计过程中，实现了下面几个关键步骤：

1. **订单生成与决策：**根据时间段生成订单，并根据紧急程度分配给当前或未来的配送任务。
2. **遗传算法优化：**利用遗传算法优化路径规划。
3. **适应度计算：**通过添加惩罚项实现对符合无人机距离和载重限制的可行解的有效选择。
4. **路径规划：**利用 2-近似度的近似算法规划单次配送的最优路径。
5. **改良部分匹配交叉：**针对该问题个体编码的特殊性，对常见的部分匹配交叉进行改进，以适用于本方案，改良后的 PMX 操作可以有效探索更优路径组合，提升种群整体适应度。
6. **多种变异操作：**通过多种变异操作的组合，本算法能够有效地增加种群的多样性，避免陷入局部最优解。

通过这些步骤和技术，我们建立了一个完整的算法框架，能够处理实时生成的订单，并通过遗传算法优化无人机的配送路径，以提升配送效率和服务质量。在 5 个配送中心、60 个卸货点、地图大小为 $40km \times 40km$、每个卸货点在每半小时内生成 1-3 个订单的数据规模下，单轮决策用时平均为 11.32 秒，最终一天总配送路径长度为 $21108.44km$。综合考量决策用时和得到最优解的质量来看，基于遗传算法的无人机路径规划求解方案可以取得相对不错的效果。

六、 完整代码

```
import numpy as np
import random
import math
import matplotlib.pyplot as plt
from pylab import mpl
from matplotlib.patches import Circle
import time
import pickle

# 设置显示中文字体
mpl.rcParams["font.sans-serif"] = ["SimHei"]


# 参数
num_centers = 5    # 配送中心数量
num_points = 60    # 卸货点数量
map_size = 40      # 地图尺寸 单位为公里
t = 30             # 时间段间隔，单位为分钟
n = 10             # 无人机最大可承载订单数量
max_distance = 20  # 无人机最大飞行距离，单位为公里
speed = 60         # 无人机飞行速度，单位为公里/小时
time_limit = 24* 60# 一天的时间限制，单位为分钟（24小时）
population_size = 500# 遗传算法种群规模
generations = 30   # 遗传算法运行的迭代次数
mutation_rate = 0.3 # 变异率
crossover_rate = 0.9 # 交叉率

def save_map(centers, points, distance_matrix, filename):
   with open(filename, 'wb') as f:
      pickle.dump((centers, points, distance_matrix), f)


```

```
33 def load_map(filename):
34     with open(filename, 'rb') as f:
35         centers, points, distance_matrix = pickle.load(f)
36     return centers, points, distance_matrix
37 
38 
39 def generate_map(num_centers, num_points, map_size, max_distance):
40     # 在地图的四个角和中心生成配送中心
41     centers = [
42         (0, (max_distance/2, max_distance/2)),
43         (1, (max_distance/2, map_size-max_distance/2)),
44         (2, (map_size-max_distance/2, max_distance/2)),
45         (3, (map_size-max_distance/2, map_size-max_distance/2)),
46         (4, (map_size / 2, map_size / 2))
47     ]
48 
49     points = []
50     point_id = num_centers # 编号从配送中心之后开始
51 
52     for center in centers:
53         center_id, center_coord = center
54         num_points_per_center = num_points // num_centers #
               平均分配给每个配送中心的卸货点数量
55         for _ in range(num_points_per_center):
56             while True:
57                 # 在配送中心为圆心，最大飞行距离的一半为半径的范围内生成卸货点
58                 angle = random.uniform(0, 2* np.pi)
59                 radius = random.uniform(1, max_distance / 2)
60                 point_x = center_coord[0] + radius * np.cos(angle)
61                 point_y = center_coord[1] + radius * np.sin(angle)
62 
63                 # 确保生成的点在地图范围内
64                 if 0<= point_x <= map_size and 0<= point_y <= map_size:
65                     point = (point_id, (point_x, point_y))
```

```
                points.append(point)
                point_id += 1
                break

    # 计算所有点对之间的距离
    all_points = centers + points
    num_all_points = len(all_points)
    distance_matrix = np.zeros((num_all_points, num_all_points))

    for i in range(num_all_points):
        for j in range(i + 1, num_all_points):
            point1 = all_points[i][1]
            point2 = all_points[j][1]
            distance = math.sqrt((point1[0] - point2[0]) ** 2+ (point1[1] -
                point2[1]) ** 2)
            distance_matrix[i][j] = distance
            distance_matrix[j][i] = distance

    return centers, points, distance_matrix


# 生成订单函数，使用模拟时间
def generate_orders(current_time, points):
    orders = []
    for point in points:
        num_orders = random.randint(0, 3) # 每个卸货点都至少有一个订单
        for _ in range(num_orders):
            priority = random.choice(['一般', '较紧急', '紧急'])
            order_time = current_time
            if priority == '一般':
                deadline = current_time + 180# 3小时
            elif priority == '较紧急':
                deadline = current_time + 90# 1.5小时
            else: # '紧急'
```

```
            deadline = current_time + 30# 30分钟
        orders.append((point, priority, order_time, deadline)) # 添加订单信息
    return orders


def process_orders(new_orders, remaining_orders, current_time):
    current_to_deliver = []
    updated_remaining_orders = [] # 复制剩余订单列表，防止直接修改原始列表

    # 将剩余订单中的紧急和较紧急订单加入到当前配送列表,其余的加入待配送列表
    for order in remaining_orders:
        point, priority, order_time, deadline = order
        time_left = deadline - current_time
        urgency_level = calculate_urgency(time_left)
        if urgency_level in ['紧急', '较紧急']:
            current_to_deliver.append(order)
        else:
            updated_remaining_orders.append(order)

    # 根据紧急程度决定本次配送和更新剩余订单列表
    for order in new_orders:
        point, priority, order_time, deadline = order
        time_left = deadline - current_time
        urgency_level = calculate_urgency(time_left)
        if urgency_level in ['紧急', '较紧急']:
            current_to_deliver.append(order)
        else:
            updated_remaining_orders.append(order)
    if current_time == time_limit - t:
        current_to_deliver.extend(updated_remaining_orders)
    return current_to_deliver, updated_remaining_orders


def calculate_urgency(time_left):
```

```
133     # 根据截止时间剩余的时间长度计算紧急程度
134     if time_left <= 30:
135         return '紧急'
136     elif time_left <= 90:
137         return '较紧急'
138     else:
139         return '一般'
140 
141 
142 def maintain_order_counts(centers, points, orders):
143     # 初始化订单数量数组，配送中心的订单数量为0
144     num_all_points = len(centers) + len(points)
145     order_counts = [0] * num_all_points
146 
147     # 记录每个点的订单数量
148     for order in orders:
149         point_id = order[0][0]
150         order_counts[point_id] += 1
151 
152     return order_counts
153 
154 
155 def print_individual(individual):
156     for path in individual:
157         path_str = " -> ".join(str(point[0]) for point in path)
158         path_distance = sum(distance_matrix[path[i][0]][path[i+1][0]] for i in
              range(len(path) - 1))
159         print(f"路径: {path_str}, 长度: {path_distance:.2f}")
160 
161 
162 def initialize_population(centers, points, population_size, max_distance,
      order_counts, n):
163     population = []
164     for i in range(population_size):
```

```
165         individual = [[] for _ in centers] # 为每个配送中心初始化一条路径
166         population.append([path for path in individual_fix(individual,
                order_counts) if path])
167         # print(population[i])
168     return population
169
170
171 def fitness(individual):
172     total_distance = 0
173     penalty = 0# 添加 penalty 变量
174     for path in individual:
175         path_distance = 0
176         path_orders = 0
177         for i in range(len(path) - 1):
178             path_distance += distance_matrix[path[i][0]][path[i + 1][0]]
179             path_orders += order_counts[path[i][0]]
180         if path_distance > max_distance:
181             penalty += 1000000# 如果路径超过最大飞行距离，给予较大的惩罚
182         if path_distance > n:
183             penalty += 1000000
184         total_distance += path_distance
185
186     return 1/ (total_distance + penalty + 0.0000000000001) #
            加入惩罚项的适应度计算
187
188
189 # 选择操作
190 def selection(population, fitnesses):
191     selected_indices = np.random.choice(range(len(population)), size=2,
            p=fitnesses / fitnesses.sum(), replace=False)
192     return population[selected_indices[0]], population[selected_indices[1]]
193
194
195 # 交叉操作
```

```
196 def pmx_crossover(parent1, parent2):
197     def pmx_single(parent_a, parent_b):
198         size = min(len(parent_a), len(parent_b))
199         child = [None] * size
200         # 选择两个交叉点
201         point1, point2 = sorted(random.sample(range(1, size - 1), 2))
202 
203         # 将交叉点之间的部分复制到子代
204         child[point1:point2] = parent_a[point1:point2]
205 
206         # 处理交叉点之外的部分
207         for i in range(point1, point2):
208             if parent_b[i] not in child:
209                 j = i
210                 while point1 <= j < point2:
211                     j = parent_b.index(parent_a[j])
212                 child[j] = parent_b[i]
213 
214         # 处理剩余位置
215         for i in range(size):
216             if child[i] is None:
217                 child[i] = parent_b[i]
218 
219         return child if len(child) != 0else parent_a
220 
221     if random.random() < crossover_rate:
222         children1 = []
223         children2 = []
224 
225         for i in range(min(len(parent1), len(parent2))):
226             parent1_nodes = set([node for node in parent1[i] if node[0] !=
                    parent1[i][0][0]])
227             parent2_nodes = set([node for node in parent2[i] if node[0] !=
                    parent2[i][0][0]])
```

```
228
229         if parent1_nodes == parent2_nodes: # 只有节点集相同时才执行交叉
230             if len(parent1[i]) > 3and len(parent2[i]) > 3:
231                 child1 = pmx_single(parent1[i], parent2[i])
232                 child2 = pmx_single(parent2[i], parent1[i])
233             else:
234                 child1 = parent1[i]
235                 child2 = parent2[i]
236             if child1:
237                 children1.append(child1)
238             if child2:
239                 children2.append(child2)
240         else:
241             children1.append(parent1[i])
242             children2.append(parent2[i])
243       children1 = [child for child in children1 if len(child) > 2]
244       children2 = [child for child in children2 if len(child) > 2]
245       # 确保所有路径以配送中心结束
246       for child in children1:
247          if child[-1][0] != child[0][0]:
248             child.append(child[0])
249       for child in children2:
250          if child[-1][0] != child[0][0]:
251             child.append(child[0])
252
253       return individual_fix(children1, order_counts),
              individual_fix(children2, order_counts)
254    else:
255       return individual_fix(parent1, order_counts), individual_fix(parent2,
              order_counts)
256
257
258 def individual_fix(individual, order_counts):
259    # 移除超出最大飞行距离或最大载货量的部分
```

```
260     for path in individual:
261         if len(path) > 2:
262             path_distance = sum(distance_matrix[path[i][0]][path[i+1][0]] for i
                    in range(len(path) - 1))
263             path_orders = sum(order_counts[point[0]] for point in path)
264             if path_distance > max_distance or path_orders > n: # 超出限制
265                 excess_distance = path_distance - max_distance
266                 excess_orders = path_orders - n
267                 while (excess_distance > 0or excess_orders > 0) and len(path) > 2:
                        # 循环删除节点直至路径合法
268                     second_last_point = path[-2]
269                     path.remove(second_last_point)
270                     excess_distance -=
                            distance_matrix[second_last_point[0]][path[-1][0]]
271                     excess_orders -= order_counts[second_last_point[0]]
272
273     # 检查是否有卸货点未配送
274     unvisited_points = [point for point in points if point not in
            sum(individual, [])]
275     random.shuffle(unvisited_points)
276     new_path = [[center] for center in centers] # 为每个配送中心初始化一条路径
277     while unvisited_points:
278         point = unvisited_points.pop(0)
279         if order_counts[point[0]] == 0:
280             continue
281         # 找到最近的配送中心
282         nearest_center = min(centers, key=lambda center:
                distance_matrix[center[0]][point[0]])
283         for path in new_path:
284             if path[0][0] != nearest_center[0]:
285                 continue
286             current_distance = sum(distance_matrix[path[i][0]][path[i+1][0]] for
                    i in range(len(path) - 1))
287             distance_to_point = distance_matrix[path[-1][0]][point[0]]
```

```
            current_orders = sum(order_counts[point[0]] for point in path)
        # 判断加入当前点后是否超过最大距离
            if current_distance + distance_to_point +
                distance_matrix[point[0]][nearest_center[0]] <= max_distance and
                current_orders + order_counts[point[0]] <= n:
                path.append(point)
                break
            else:
                new_path.append([nearest_center, point]) # 开始新路径
                break
        # new_path[path_index] = path

    for path in new_path:
        if len(path) > 1:
            individual.append(path)
    # 确保所有路径以配送中心结束
    for path in individual:
        if path and path[-1][0] != path[0][0]:
            path.append(path[0])
    # 使用集合去重
    individual_set = {tuple(path) for path in individual}
    individual = [list(path) for path in individual_set]
    individual = [path for path in individual if len(path) > 2]
    return individual


def mutate(individual):
    for i, path in enumerate(individual):
        if random.random() < mutation_rate and len(path) > 2:
            mutation_type = random.randint(1, 4) # 随机选择变异类型
            unloading_points = [idx for idx, point in enumerate(path) if idx != 0
                and idx != len(path) - 1]
            if mutation_type == 1and len(unloading_points) > 1: #
                路径交换，确保至少有两个卸货点
```

```
            swap_points = random.sample(unloading_points, 2)
            path[swap_points[0]], path[swap_points[1]] = path[swap_points[1]],
                path[swap_points[0]]
        elif mutation_type == 2and len(unloading_points) > 1: #
            路径倒置，确保至少有两个卸货点
            reverse_start = random.choice(unloading_points)
            valid_end_points = [idx for idx in unloading_points if idx >
                reverse_start]
            if valid_end_points: # 确保有合法的结束点
                reverse_end = random.choice(valid_end_points)
                path[reverse_start:reverse_end + 1] =
                    reversed(path[reverse_start:reverse_end + 1])
        elif mutation_type == 3:
            individual[i] = []
        elif mutation_type == 4and len(unloading_points) > 0: #
            随机选择一个点插入到另一条路径
            selected_point_index = random.choice(unloading_points)
            selected_point = path.pop(selected_point_index)

            # 找到该点所属的配送中心
            center_id = path[0][0]
            # 找到所有其他由该配送中心负责的路径
            other_paths = [p for p in individual if len(p) > 2and p[0][0] ==
                center_id and p != path]

            if other_paths:
                # 随机选择一个目标路径
                target_path = random.choice(other_paths)
                # 随机选择一个插入位置
                insert_position = random.randint(1, len(target_path) - 1)
                target_path.insert(insert_position, selected_point)
            else:
                # 如果没有其他路径，则将点插回原路径的随机位置
                insert_position = random.randint(1, len(path) - 1)
```

```
346                 path.insert(insert_position, selected_point)
347     # 使用 individual_fix 函数修复路径
348     individual = individual_fix(individual, order_counts)
349     return individual
350 
351 
352 def genetic_algorithm(centers, points, population_size, generations,
        max_distance, order_counts, n):
353     population = initialize_population(centers, points, population_size,
            max_distance, order_counts, n) # 初始化种群
354     best_individual = None # 初始化最佳个体为空
355     best_fitness = float('-inf') # 初始化最佳适应度为负值
356 
357     for generation in range(generations):
358         fitnesses = np.array([fitness(ind) for ind in population]) #
                计算适应度函数
359 
360         new_population = []
361         for _ in range(population_size // 2):
362             parent1, parent2 = selection(population, fitnesses) #
                    轮盘赌选择两个父本个体
363             child1, child2 = pmx_crossover(parent1, parent2) #
                    对父本个体应用部分匹配交叉得到两个孩子个体
364             mutate(child1) # 对孩子个体进行变异
365             mutate(child2)
366             new_population.extend([child1, child2]) # 繁育出新种群
367 
368         population = new_population
369 
370         current_best = population[np.argmax(fitnesses)]
371         current_fitness = max(fitnesses)
372 
373         if current_fitness > best_fitness: # 更新最佳个体
374             best_fitness = current_fitness
```

```
            best_individual = current_best
        print(f"Generation {generation + 1}: Best Fitness = {best_fitness}")

    return [path for path in best_individual if len(path) > 2] # 返回最优个体


# 绘制地图
def plot_map(centers, points, paths, max_distance):
    plt.figure(figsize=(12, 12))

    # 绘制配送中心
    centers_x, centers_y = zip(*[center[1] for center in centers])
    centers_labels = [center[0] for center in centers]
    plt.scatter(centers_x, centers_y, c='blue', marker='s', s=100,
        label='配送中心')
    for i, txt in enumerate(centers_labels):
        plt.annotate(txt, (centers_x[i], centers_y[i]), fontsize=12,
            fontweight='bold', ha='right')

    # 绘制配送中心的服务范围圆
    for center in centers:
        circle = Circle(center[1], max_distance / 2, color='blue', alpha=0.1,
            linestyle='--')
        plt.gca().add_patch(circle)

    # 绘制卸货点
    points_x, points_y = zip(*[point[1] for point in points])
    points_labels = [point[0] for point in points]
    plt.scatter(points_x, points_y, c='green', marker='o', s=60, label='卸货点')
    for i, txt in enumerate(points_labels):
        plt.annotate(txt, (points_x[i], points_y[i]), fontsize=10, ha='right')

    # 绘制路径
    colors = plt.cm.rainbow(np.linspace(0, 1, len(paths))) #
```

```
        使用不同颜色绘制每条路径
    for path, color in zip(paths, colors):
        path_x, path_y = zip(*[point[1] for point in path])
        plt.plot(path_x, path_y, '-', color=color, linewidth=2, alpha=0.7)
        for point in path:
            plt.annotate(point[0], point[1], fontsize=8, ha='center')

    plt.title('无人机配送路径规划', fontsize=16, fontweight='bold')
    plt.xlabel('X 坐标', fontsize=14)
    plt.ylabel('Y 坐标', fontsize=14)
    plt.legend(loc='best', fontsize=12)
    plt.grid(True, linestyle='--', alpha=0.6)
    plt.axis((0, map_size, 0, map_size))
    plt.show()


def print_best_individual(best_result, orders):
    print("最优路径和路径长度: ")
    for path in best_result:
        path_str = " -> ".join(str(point[0]) for point in path)
        path_distance = sum(distance_matrix[path[i][0]][path[i + 1][0]] for i in
            range(len(path) - 1))
        print(f"路径: {path_str}, 长度: {path_distance:.2f}")

        # 输出订单信息
        for point in path:
            point_id = point[0]
            if point_id < len(centers):
                continue # 跳过配送中心

            point_orders = [order for order in orders if order[0][0] == point_id]
            for order in point_orders:
                order_priority = order[1]
                order_time = order[2]
```

```
438             order_deadline = order[3]
439             print(f" 点 {point_id} 的订单 - 优先级: {order_priority}, 下单时间:
                    {order_time // 60} 小时 {order_time % 60} 分钟, 截止时间:
                    {order_deadline // 60} 小时 {order_deadline % 60}")
440
441
442 # 生成地图
443 # centers, points, distance_matrix = generate_map(num_centers, num_points,
        map_size, max_distance) # 随机生成地图
444 # save_map(centers, points, distance_matrix, 'map_data.pkl') # 保存地图文件
445 centers, points, distance_matrix = load_map('map_data.pkl')
446 orders = []
447 total_daily_distance = 0
448 todo_list = []
449 for current_time in range(0, time_limit, t):
450     # 生成当前时间段的订单
451     new_orders = generate_orders(current_time, points)
452     start_time = time.time() # 获取开始时间
453     current_orders, todo_list = process_orders(new_orders, todo_list,
            current_time)
454     # 计算订单数量统计
455     order_counts = maintain_order_counts(centers, points, current_orders)
456     print(f"当前时间:  {current_time} order_counts: {order_counts}")
457     # 运行遗传算法优化路径规划
458     best_result = genetic_algorithm(centers, points, population_size,
            generations, max_distance, order_counts, n)
459     # 获取结束时间
460     end_time = time.time()
461     use_time = end_time - start_time
462     print(f"决策用时: {use_time:.2f}s")
463     current_best_distance = sum(distance_matrix[path[i][0]][path[i + 1][0]] for
            path in best_result for i in range(len(path) - 1))
464     total_daily_distance += current_best_distance
465     # 输出最优路径
```

```
    hours = current_time // 60
    minutes = current_time % 60
    print(f"当前时间: {hours} 小时 {minutes} 分钟")
    print_best_individual(best_result, current_orders)
    # 绘制最佳路径
    plot_map(centers, points, best_result, max_distance)

print("所有时间段订单处理完毕。")
print(f"总路径长度: {total_daily_distance:.2f}")
```