

武汉大学国家网络安全学院

《高级算法设计与分析》课程报告

无人机配送路径规划实验报告

院（系）名 称：网络安全学院

专 业 名 称：网络空间安全

任 课 教 师：林 海

学 生 姓 名：谢彩云

学 生 学 号：2023202210163

二〇二四年六月

目录

1 实验背景	3
2 前置知识	4
2.1 多配送中心的 VRP 问题	4
2.2 基于图的路径规划问题	4
3 算法设计	5
4 代码实现	6
4.1 相关参数设置	6
4.2 图的初始化与数据预处理	6
4.3 卸货点与对应订单类对象设计	10
4.4 路径择优算法实现	11
5 实验结果	13
5.1 初始化数据	13
5.2 求解 24 小时内的最短路径	17
6 总结	18

1 实验背景

本实验基于无人机配送优化问题展开，其中需要设计一个算法来实现在特定区域内的无人机路径规划，以最小化所有无人机总配送路径的目标。在本实验中提取的重要信息梳理如下：

1、问题描述：

在一个特定区域内，有多个配送中心和卸货点。

配送中心有无限数量的商品和无人机。

卸货点随机生成订单，订单有三个不同的优先级别：一般、较紧急和紧急，对应不同的配送时间要求。

2、实验目标：

设计一个算法，使得一段时间内所有无人机的总配送路径最短。

必须满足订单的优先级别要求，即保证所有订单按时送达。

3、约束条件：

每架无人机一次最多能携带 N 个物品。

无人机一次飞行最远路程为 20 公里，包括送货和返回配送中心。

无人机的速度为 60 公里/小时。（隐含条件：最远卸货点不超过 10 公里）

4、假设条件：

配送中心的无人机数量无限。

任何配送中心都能满足用户的订货需求。

5、关键考虑因素：

订单优先级管理：需要实时决策哪些订单需要立即处理以满足紧急配送要求。

路径规划和配送决策：每隔一段时间，系统需决定每个配送中心派出多少无人机，以及每架无人机的具体配送路径，包括顺序和返回路径。

优化目标：最小化所有无人机总配送路径，这包括考虑无人机的载货量、最大飞行距离限制、速度等因素，同时满足订单优先级和时间要求。

6、算法设计挑战：

动态决策问题：需设计有效的算法来动态调整无人机的配送路径和数量，以应对随时间变化的订单量和优先级要求。

路径优化算法：需要考虑典型的路径优化问题，如 TSP（旅行商问题），以确保无人机的回程路径也被最小化。

实时性和效率：算法需要在每个时间间隔内快速做出决策，保证系统的实时性和效率。

7、参考资料：

本实验在实现时参考的部分在线电子资料如下。

[1] https://blog.csdn.net/weixin_51545953/article/details/128937834

[2] <https://blog.csdn.net/tangshishe/article/details/116382590>

2 前置知识

2.1 多配送中心的 VRP 问题

多配送中心的 VRP (Vehicle Routing Problem) 涉及到在多个配送中心之间有效地规划车辆路线, 以满足一定数量的配送需求。在这类问题中的客户通常是需要服务的点, 而配送中心是从中派发车辆进行配送的点。车辆用于从配送中心出发, 访问客户并返回的运输工具, 具有一定的容量限制和行驶距离限制。

在本实验中, 多中心 VRP 指的是多个配送中心服务多个卸货点, 无人机可以受到配货中心的限制, 即必须返回配货中心。同时, 由于订单优先级的存在, 该实验中还存在时间窗口: 即卸货点有指定的服务时间窗口, 在这段时间内必须到达。

这类问题通常有两种解法: 精确方法和启发式方法。精确方法比如分枝定界法, 将问题分解为子问题, 通过限界和分支来找到最优解; 动态规划, 通过状态空间和状态转移方程来逐步求解; 整数规划, 将 VRP 建模为整数线性规划问题, 利用线性规划的求解算法求解。而启发式和元启发式方法包括邻域搜索, 通过邻域操作 (如交换、插入、反转) 来改进当前解; 遗传算法使用进化过程中的选择、交叉和变异操作进行优化; 模拟退火利用随机化搜索和概率接受策略来探索解空间; 禁忌搜索*避免短期优化, 并通过禁忌表来管理已访问解的历史记录。

然而, 在实际应用中应该考虑许多因素。比如实时性, 即求解算法的速度对实时路线规划至关重要。算法复杂度也尤为重要, 问题规模的增长可能导致求解难度急剧增加, 需要适应大规模数据的处理能力。满足约束条件限制, 考虑实际约束如时间窗口、容量限制、交通状况等因素。

2.2 基于图的路径规划问题

在本实验中适用的路径规划算法应需要解决从起点到终点或多个点之间最优路径的问题。比如 Dijkstra 算法, 用于解决单源最短路径问题的经典算法, 适用于所有边权重非负的有向图或无向图。它通过贪心策略逐步扩展最短路径集合, 直到找到起点到目标节点的最短路径。从起点开始, 每次选择当前最短路径的节点进行扩展, 更新与该节点直接相连的节点的最短路径。但是仅用于无负权重环路的最短路径问题, 例如道路网络中的最短驾驶路径规划。再就是 A*算法结合了 Dijkstra 算法的最短路径查找和启发式搜索的思想, 适用于有向图或无向图中的路径规划问题, 特别是在有额外信息 (如启发式函数) 可用时, 可以更快地找到最优路径。通过维护一个开放列表, 根据启发式函数估计的最短路径代价来扩展节点, 直到找到终点。适用于需要快速找到起点到目标节点的最优路径, 例如在游戏地图中的路径规划或机器人的运动规划中。还有比如最小生成树算法 (如 Prim 和 Kruskal 算法), 虽然通常用于生成树问题, 但最小生成树算法也可以在

某些路径规划问题中使用，例如需要通过最小成本连接所有节点的问题。**Prim 算法**：基于节点集合的贪心算法，逐步扩展生成树，直到覆盖所有节点。**Kruskal 算法**：基于边集合的贪心算法，通过增加边来构建生成树，直到满足所有节点的连接。这些算法为解决不同类型的路径规划问题提供了基础和工具，根据具体问题的特点选择合适的算法可以高效地求解路径规划问题。

3 算法设计

在本实验中，为了尽量减少所有无人机的总路径长度，我们需要尽可能使每架无人机的负载达到最大。由于每次无人机的决策仅基于当前可知信息，无法预知未来订单的生成情况。因此，我们的目标是在每次决策中尽量减少无人机的飞行距离，确保每架无人机在执行任务时达到最大负载，以减少无人机的数量和单次飞行的距离。

基于以上分析，我们认为应该在订单的最迟发货时间之前派遣无人机，并且努力使每架无人机在任务执行时负载达到最大。订单的最迟发货时间是指如果当前决策周期不发货，则订单将无法在规定时间内送达目的地。负载最大化策略要求，当某个地址的订单数量达到 K 个时，如果第 K 架无人机未满载，首先检查是否存在其他路径小于无人机的最大飞行距离，并且这些路径上存在紧急订单（即必须按时交付的订单）。如果存在这样的路径，就会安排无人机执行这些任务，以达到满载。如果该路径上的所有节点都无法满足满载条件，则考虑执行该卸货点的次优先级紧急订单，以确保无人机负载最大化。如果在以上情况下依然无法满足负载最大化条件，则无人机直接派往目的地，因为此时算法已经尽力最大化了单次飞行的订单量，无法进一步优化。

上述为本算法的路径规划策略。该算法采用整数规划的思想，旨在设计尽可能精确的最短总路径算法。比如在无人机未满载时，当选定非直达路径时，相当于将该路径所在的路径赋值为 1，此时其他路径不可选，这架无人机的路程已被固定，但是由于未来订单的生成未知性，我们尽可能的使无人机满载，可以减少未来决策轮次上的无人机的负担，达到每一轮次的总体最优，并通过该决策方式达到全局的极大最优。

为了减少算法复杂度。首先，算法需要建立一个在配送点和卸货点形成的图网络上的所有满足无人机最大里程数的路径的表，该表面向所有节点，包括卸货点和配货点，并按照路径距离从小到大排序，使找最优最短路径的过程尽可能快，类似于算法课中所讲述的隐枚举方法的排序的作用。

其次，建立一个面向卸货点的直飞路径表和一个面向卸货点的不包括直飞路径且满足条件的路径顺序表，这两个表分别对应于直飞方案和非直飞方案。当直飞方案不被采用时，算法仅需检索不包含直飞的路径表，无需再遍历图网络即可知道下一步的决策方向是选择新的路径还是选择该卸货点下一决策轮次的订单。

4 代码实现

4.1 相关参数设置

根据前述的实验条件分析，我初步设置了一些条件限制，具体设置如下图。

```
# === other config ===
Max_Route_Len = 20 # 最长路径长度
Max_Order_Num = 20 # 最大订单数目
Decision_Time_Interval = 10 # 决策时间间隔
Max_Drone_Carry = 10 # 无人机最多承载商品数目
Item_Priority = [30, 90, 180]
Drone_Speed = 1
Time_to_Work = 24
```

图 4.1 实验设置

上述的 `Max_Route_Len = 20` 表示最长路径长度为 20，即无人机的最大里程数；`Max_Order_Num = 20` 为最大订单数目，即每个卸货点每次随机生成的最大订单数为 0~20 个订单；`Decision_Time_Interval = 10` 为决策时间间隔，即在实验中，我们每隔 10 分钟做一次决策，即对无人机本轮次是否飞行和飞行路径做决策的决策时间间隔为 10 分钟；`Max_Drone_Carry = 10` 表示无人机最多承载商品数目为 10 个订单；`Item_Priority = [30, 90, 180]` 表示订单的紧急程度，分别为紧急-30 分钟内送达，较紧急-90 分钟内送达，一般-180 分钟内送达；`Drone_Speed = 1` 表示无人机的飞行速度为 1 公里一分钟，该数值通过 60 公里每小时计算得到；`Time_to_Work = 24` 表示本实验处理的订单时间段为 24 小时。

4.2 图的初始化与数据预处理

```
def generate_points(num_delivery, num_drop, max_distance):
    dy_points = []
    dp_points = []

    # Generate delivery points
    for _ in range(num_delivery):
        x = random.randint(10, 30)
        y = random.randint(10, 30)
        dy_points.append((x, y))

    # Generate drop points ensuring each is within max_dist from at least one delivery point
    for _ in range(num_drop):
        while True:
            # Randomly select a delivery point
            dy_point = random.choice(dy_points)
            dp_point = (random.randint(0, 40), random.randint(0, 40))
            # Check if distance constraint is satisfied
            if any(calculate_distance(dp_point, dy_point) <= max_distance for dp in dy_points):
                dp_points.append(dp_point)
                break

    return dy_points, dp_points
```

图 4.2 生成配送点和卸货点的坐标

为了计算无人机路径规划问题的最短总路径，首先需要初始化各个配货点和卸货点的坐标。随机化生成坐标的代码如图 4.2 所示。为了使卸货点符合条件，确保每个卸货点与至少一个配送点的距离不超过指定的最大距离，在生成卸货点时，使用了 `calculate_distance` 的函数来计算两点之间的距离，该函数用于计算两个二维点之间的欧氏距离，使用了平方根来计算两点之间的直线距离。在本实验中，限定的距离最大为 10。`calculate_distance` 函数的具体实现如图 4.3 所示。

```
def calculate_distance(point1, point2):
    return ((point1[0] - point2[0]) ** 2 + (point1[1] - point2[1]) ** 2) ** 0.5
```

图 4.3 `calculate_distance` 函数的具体实现

在计算得到配送点和卸货点后，我们通过 `draw` 函数来可视化各个配货点和卸货点之间的位置关系。`draw` 函数使用 `plt.scatter` 函数绘制散点图。`draw` 函数的具体实现如图 4.4 所示。

```
def draw(points1, points2):
    x, y = [], []
    for xy in points1:
        x.append(xy[0])
        y.append(xy[1])
    plt.scatter(x, y, color='red', marker='*', s=100) # 使用scatter函数绘制散点图

    x, y = [], []
    for xy in points2:
        x.append(xy[0])
        y.append(xy[1])
    plt.scatter(x, y, color='blue', s=50) # 使用scatter函数绘制散点图

    # 设置图形属性
    plt.title('Address Scatter Plot with Coordinates')
    plt.xlabel('X-axis')
    plt.ylabel('Y-axis')
    # 显示图形
    plt.grid(True)
    plt.savefig("AddressScatterPlot.png")
    plt.show()
```

图 4.4 `draw` 函数的具体实现

至此，选址已经确定下来，下一步使用 `dis_matrix` 函数来构建邻接矩阵。

```
def dis_matrix(position1, position2):
    distance_matrix = []
    for i_xy in position1:
        dis_ls = []
        for j_xy in position2:
            dis_tmp = calculate_distance(i_xy, j_xy)
            # 1、距离配货点超过10不可达，设置为100
            # 2、卸货点间超过10没必要派无人机遍历、双无人机的路径最差结果比单无人机好，且运送订单是单无人机的2倍
            dis_tmp = round(dis_tmp, 2) if dis_tmp <= 10 else 100
            dis_ls.append(dis_tmp)
        distance_matrix.append(dis_ls)
    return distance_matrix
```

图 4.5 `dis_matrix` 函数的具体实现

`dis_matrix` 函数的具体实现如图 4.5 所示。在函数中，我们设置距离配货点超过 10 为不可达，设置为 100；而卸货点间超过 10 没必要派无人机遍历，以为此时双无人机的路径最差结果比单无人机好，且运送订单是单无人机的 2 倍。

上述的距离矩阵用于后续遍历图使用，为了更好地体现对应关系，我们在可视化的基础上，加上了可达点之间的边的构建，并使用 `draw_lines` 函数进行绘制。`draw_lines` 函数的具体实现如图 4.6 所示。

```
def draw_lines(points1, points2, distance_dp, distance_dy):
    x, y = [], []
    for xy in points1:
        x.append(xy[0])
        y.append(xy[1])
    plt.scatter(x, y, color='red', marker='*', s=100) # 使用scatter函数绘制散点图
    x, y = [], []
    for xy in points2:
        x.append(xy[0])
        y.append(xy[1])
    plt.scatter(x, y, color='blue', s=50) # 使用scatter函数绘制散点图

    for i in range(len(distance_dp)):
        for j in range(len(distance_dp) - i - 1):
            if distance_dp[i][j + i + 1] <= 10:
                plt.plot([points2[i][0], points2[j + i + 1][0]],
                        [points2[i][1], points2[j + i + 1][1]], color='green')
    for i in range(len(distance_dy)):
        for j in range(len(distance_dy[i])):
            if distance_dy[i][j] <= 10:
                plt.plot([points1[i][0], points2[j][0]],
                        [points1[i][1], points2[j][1]], color='green')
```

图 4.6 `draw_lines` 函数的具体实现

同时，在算法设计中提到，我们需要构建一个在配送点和卸货点形成的图网络上的所有满足无人机最大里程数的路径的表，该表面向所有节点，包括卸货点和配货点，并按照路径距离从小到大排序，使找最优最短路径的过程尽可能快。为了构建该表，我们使用 `find_cycles_sat` 函数来寻找符合条件的回路并返回。`find_cycles_sat` 函数的具体实现如图 4.7 所示。该函数通过遍历来寻找回路，并且寻找的图是有全图，边的权重就是节点间的距离。`visited` 是一个集合，用于跟踪在深度优先搜索（DFS）过程中已经访问过的节点。`cycles` 和 `cycles_len` 是列表，用于存储找到的回路及其对应的权重（回路的总权重）。嵌套的 DFS 函数 `dfs` 从 `node_now` 开始执行深度优先搜索，`start` 是当前正在探索的循环的起始节点，`path` 用于记录当前 DFS 路径中的节点，`weights` 用于跟踪当前路径中边的累积权重。当发现一个回路时，即 `neighbor = start` 并且路径长度大于 1 时，将 `path` 添加到 `cycles`，将 `weights` 添加到 `cycles_len`，用于记录节点和跳到该跳节点的代价，递归地探索未访问过的 `node_now` 的邻居节点。`find_cycles_sat` 的主循环是为了遍历所有配送点，在图中我们的配货点均以 ‘dy’ 作为前缀。在完成所有配送点的回路遍历后，过滤掉总权重超过 20 的回路，因为该回路不满足无人机最大里程数为 20 的条件。


```

def find_cycles_sat(graph_to_find):
    visited = set()
    cycles = []
    cycles_len = []

    def dfs(node_now, start, path, weights):
        visited.add(node_now)
        for neighbor, weight in graph_to_find.get(node_now, {}).items():
            if neighbor == start and len(path) > 1:
                cycles.append(path + [neighbor])
                cycles_len.append(weights + [round(weights[-1] + weight, 2)])
            elif neighbor not in visited:
                dfs(neighbor, start, path + [neighbor], weights + [round(weights[-1] + weight, 2)])
        visited.remove(node_now)

    for node in graph_to_find:
        if 'dy' in node:
            dfs(node, node, [node], [0.0])

    cycles_sat = []
    cycles_len_sat = []
    for routes, route_lens in zip(cycles, cycles_len):
        if route_lens[-1] <= 20:
            flag = True
            for cycle in cycles_sat:
                reversed_cycle = cycle[::-1]
                if reversed_cycle == routes: # 如果两个回路相等，则不加入路径list
                    flag = False
                    break
            if flag is True:
                cycles_sat.append(routes)
                cycles_len_sat.append(route_lens)
                print(routes, route_lens, route_lens[-1])
    print(cycles_sat, cycles_len_sat)
    return cycles_sat, cycles_len_sat

```

图 4.7 find_cycles_sat 函数的具体实现

其次，还需要建立一个面向卸货点的直飞路径表和一个面向卸货点的不包括直飞路径且满足条件的路径顺序表，建表的代码实现如图 4.8 所示。

```

dy2dp_direct = []
for i in range(len(routes_sat)):
    if len(routes_sat[i]) == 3:
        dy2dp_direct.append([routes_sat[i][1], routes_len[i][-1], routes_sat[i]])
print(dy2dp_direct)

dy2dp_direct = [...]

dps = []
for dp in dp_list:
    for rt_i in range(len(routes_sat)):
        if len(routes_sat[rt_i]) != 3:
            if dp in routes_sat[rt_i]:
                dps.append([dp, routes_sat[rt_i], routes_len[rt_i]])
print(sorted(dps))

dps = [...]

```

图 4.8 建表的代码实现

4.3 卸货点与对应订单类对象设计

该小节分别定义卸货点类 DP_Point 和卸货点集合类 DP_Points。

```
class DP_Point:
    def __init__(self, location):
        self.location = location
        self.current_orders = []

    def generate_orders(self, start_time):
        num_orders = random.randint(0, Max_Order_Num)
        for _ in range(num_orders):
            priority = random.choice(Item_Priority) # 随机生成订单优先级
            end_time_satisfy = start_time + priority # 最晚送达时间
            self.current_orders.append(end_time_satisfy)
        self.current_orders.sort(key=lambda x: x) # 按照优先级排序
        return

    def display_orders(self, start_time):
        print(f"在第{start_time}分钟时, 卸货点{self.location}"
              f"当前所有订单最迟送达时间: {self.current_orders}")
```

图 4.9 DP_Point 类的具体实现

首先, DP_Point 类对象用于表示一个卸货点。self.location 存储卸货点的位置信息。self.current_orders 用于存储当前卸货点的订单信息。generate_orders(self, start_time)用于随机生成一定数量的订单, 并按照订单的优先级排序。start_time 定订单生成的起始时间, 并使用 random.randint(0, Max_Order_Num)随机生成订单数量用以控制订单数量的上限; random.choice(Item_Priority)随机选择订单的优先级。end_time_satisfy = start_time + priority 计算了该卸货点每个订单的最晚送达时间。在所有订单生成后, 使用 self.current_orders.append(end_time_satisfy)将每个订单的最晚送达时间添加到 self.current_orders 列表中, 并使用根据订单的最晚送达时间列表进行排序, 使后续遍历卸货点的订单时, 优先处理紧急订单。display_orders(self, start_time)用于打印卸货点当前的订单信息, 包括卸货点位置、当前所有订单的最迟送达时间。DP_Point 类的具体实现如图 4.9 所示。

```
class DP_Points:
    def __init__(self, locations):
        self.dp_point_list = []
        for xy in locations:
            self.dp_point_list.append(DP_Point(xy))

    def generate_new_orders(self, start_time):
        for dp in self.dp_point_list:
            dp.generate_orders(start_time)

    def display_all_orders(self, start_time):
        for dp in self.dp_point_list:
            dp.display_orders(start_time)
```

图 4.10 DP_Points 类的具体实现

再就是 DP_Points 类,该类方法用于管理多个卸货点,首先通过 locations 参数来初始化初始化多个 DP_Point 实例并存储在 dp_point_list 中。然后,在后续的 generate_new_orders 方法中会调用每个卸货点实例的 generate_orders 方法,以生成新的订单。对应的 display_all_orders 方法会调用每个卸货点实例的 display_orders 方法,以显示所有卸货点当前的订单信息。DP_Points 类的具体实现如图 4.10 所示。

4.4 路径择优算法实现

路径择优算法实现如图 4.11 所示。该部分主要实现了在各类分支情况下,无人机路径应该如何规划的问题,即将第三章的内容复现成代码形式并执行。

```
dp_points = DP_Points(DP_position)
now_time = 0
time_scale = 60 * Time_to_Work # 规划12小时内的决策 并让订单在第9小时停止完成 让12小时内可以完成所有订单
route_in_12 = []
while now_time <= time_scale:
    if now_time <= time_scale - 60 * 3: # 小于截止时间前3小时均可以生成订单
        dp_points.generate_new_orders(now_time)
    dp_points.display_all_orders(now_time)
    order_urgent_list = [] # 存储当前决策轮次必须处理的订单数目
    dp_ls = dp_points.dp_point_list
    for dp_i in range(len(dp_ls)):
        dp_order = dp_ls[dp_i].current_orders
        dp_new_order = [x for x in dp_order if x - now_time > 10]
        num = len(dp_order) - len(dp_new_order)
        if num == 0:
            continue
        order_urgent_list.append([dp_i, num]) # 当前轮次最紧急的订单先处理
        dp_ls[dp_i].current_orders = dp_new_order
    order_urgent_list = sorted(order_urgent_list, key=lambda x: x[1]) # 先处理需求量大的,让无人机尽可能满
    print(order_urgent_list)
    route_now_time = []
    while len(order_urgent_list) != 0:
        # 对于订单量划分两部分
        # 第一部分为专用航线仅飞往目的地的
        order_urgent_max = order_urgent_list.pop() # 接收的是索引,无参数返回最后一个元素,即最多订单的卸货点
        dp, order_num = order_urgent_max[0], order_urgent_max[1]
        drone_num = order_num // Max_Drone_Carry # 专航两点直飞需要的无人机数量
        dy2dp_direct_now = []
        for tmp in dy2dp_direct:
            if dp == tmp[0]:
                dy2dp_direct_now = [tmp[1], tmp[2]]
        if drone_num != 0:
            route_now_time.append([drone_num, dy2dp_direct_now[0], dy2dp_direct_now[1]]) # 无人机数目
        # 第二部分为无人机未实现满载时
        drone_remainder_num = order_num % Max_Drone_Carry # 剩余的order数目
        if drone_remainder_num == 0:
            continue
        if len(order_urgent_list) != 0:
            dps_dp = list(filter(lambda x: dp == x[0], dps))
            if len(dps_dp) != 0: # 存在其他路径
                for dps_dp_info in dps_dp: # 按照路径长短遍历路径寻找
                    # 找出该条路径经过的节点
                    dps_dp_route = list(filter(lambda x: 'dp' in x, dps_dp_info[1])).remove(dp)
                    order_urgent_dp = [x[0] for x in order_urgent_list]
                    dp_to_expand = duplicate_removal(dps_dp_route, order_urgent_dp)
                    if len(dp_to_expand) == 0: # 若该路径与urgent相交为空 则换下一条路径
                        continue
```

图 4.11(a) 路径择优算法实现部分代码

```

else:
    while len(dp_to_expand) != 0:
        dp_exp = dp_to_expand.pop() # 'dpX'
        ind = order_urgent_dp.index(dp_exp)
        order_info = order_urgent_list[ind]
        if order_info[1] + drone_remainder_num > Max_Drone_Carry:
            # 扣除需要的订单并更新urgent
            order_urgent_list[ind][1] = order_info[1] - (Max_Drone_Carry - drone_remainder_num)
            drone_remainder_num = Max_Drone_Carry
            break
        elif order_info[1] + drone_remainder_num == Max_Drone_Carry:
            # 删除相应的订单并更新urgent
            drone_remainder_num = Max_Drone_Carry
            order_urgent_list.pop(ind)
            break
        elif order_info[1] + drone_remainder_num < Max_Drone_Carry:
            # 更新飞机上的订单数量drone_remainder_num 并进入下一轮循环
            drone_remainder_num = order_info[1] + drone_remainder_num
            order_urgent_list.pop(ind)
    if drone_remainder_num <= Max_Drone_Carry: # 飞机已经满载 找到路径添加到route记录即可
        need_to_pop = Max_Drone_Carry - drone_remainder_num
        if need_to_pop != 0:
            dp_orders = dp_points.dp_point_list[dp_list.index(dp)].current_orders
            if len(dp_orders) <= need_to_pop:
                dp_points.dp_point_list[dp_list.index(dp)].current_orders = []
            else:
                dp_points.dp_point_list[dp_list.index(dp)].current_orders = dp_orders[need_to_pop:]
            route_now_time.append([1, dps_dp_info[2][-1], dps_dp_info[1]]) # 无人机数目 路径
            break # 执行至此 路径已经锁死 没必要继续遍历路径
    else: # == 不存在其他路径则调用下一轮次的同一地点的货物使其飞机满载
        need_to_pop = Max_Drone_Carry - drone_remainder_num
        if need_to_pop != 0:
            dp_orders = dp_points.dp_point_list[dp_list.index(dp)].current_orders
            if len(dp_orders) <= need_to_pop:
                dp_points.dp_point_list[dp_list.index(dp)].current_orders = []
            else:
                dp_points.dp_point_list[dp_list.index(dp)].current_orders = dp_orders[need_to_pop:]
            route_now_time.append([1, dy2dp_direct_now[0], dy2dp_direct_now[1]]) # 无人机数目 路径 路径
    route_in_12.append(route_now_time)
    now_time = now_time + Decision_Time_Interval

```

图 4.11(b) 路径择优算法实现部分代码

上述代码实现了在一段时间内的多次决策的路径规划问题。在截止时间的前 3 小时以内，调用 `dp_points.generate_new_orders(now_time)` 生成订单。每次循环都调用 `dp_points.display_all_orders(now_time)` 显示当前时间下所有卸货点的订单信息。然后，在每轮决策中，都优先进行紧急订单处理：初始化一个空列表 `order_urgent_list` 用来存储当前决策轮次必须处理的订单数量。获取所有卸货点的订单列表 `dp_points.dp_point_list`，遍历每个卸货点的订单列表过滤器筛选出时

间戳大于当前时间加上 10 的订单，即不紧急的订单。计算必须在本决策轮次处理的订单数量并将卸货点信息及其紧急订单数量添加到 `order_urgent_list` 中等待处理。为了优先处理订单数目较多的卸货点，算法根据订单数量进行排序，使需求量大的订单优先处理，以便使无人机尽可能满载。

对于每个卸货点的紧急订单，我们计算需要的无人机数量，并将已经满载的无人机通过直飞路径表检索直接发出；而未满载的无人机通过遍历非直飞路径表来采取其他卸货点的紧急订单遍历的方式使无人机满载。

若上述操作均无法使无人机满载，则取出将该卸货点的下一轮次紧急订单使无人机尽可能满载启航。

```
def duplicate_removal(A, B):  
    if A is None or B is None:  
        return []  
    return list(set(A).intersection(set(B)))
```

图 4.12 duplicate_removal 函数

```
def is_urgent(x, now):  
    return x - now >= 10
```

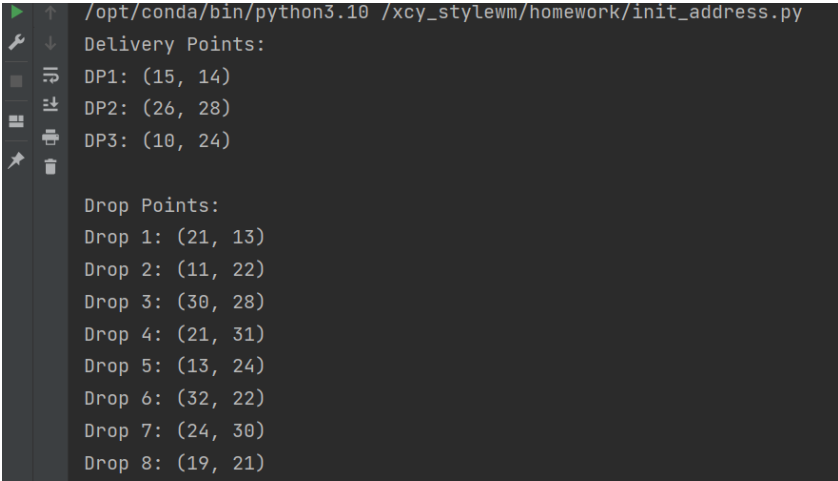
图 4.13 is_urgent 函数

在上述代码中，使用到的 `duplicate_removal` 函数和 `is_urgent` 函数如图 4.12 和图 4.13 所示。其中，`duplicate_removal` 函数有效地从两个列表中移除重复元素，并返回唯一的共同元素列表，使用集合操作来进行高效的重复元素移除，并且能够优雅地处理 `None` 输入，返回一个空列表。

5 实验结果

5.1 初始化数据

执行 `draw` 函数生成的随机配货点、卸货点和对应位置关系可视化如图 5.1 和图 5.2 所示。其中，在图 5.2 中，红色星星表示配货点，蓝色圆点表示卸货点。



```
/opt/conda/bin/python3.10 /xyc_stylewm/homework/init_address.py  
Delivery Points:  
DP1: (15, 14)  
DP2: (26, 28)  
DP3: (10, 24)  
  
Drop Points:  
Drop 1: (21, 13)  
Drop 2: (11, 22)  
Drop 3: (30, 28)  
Drop 4: (21, 31)  
Drop 5: (13, 24)  
Drop 6: (32, 22)  
Drop 7: (24, 30)  
Drop 8: (19, 21)
```

图 5.1 draw 函数的生成 3 个配送点和 8 个卸货点的输出

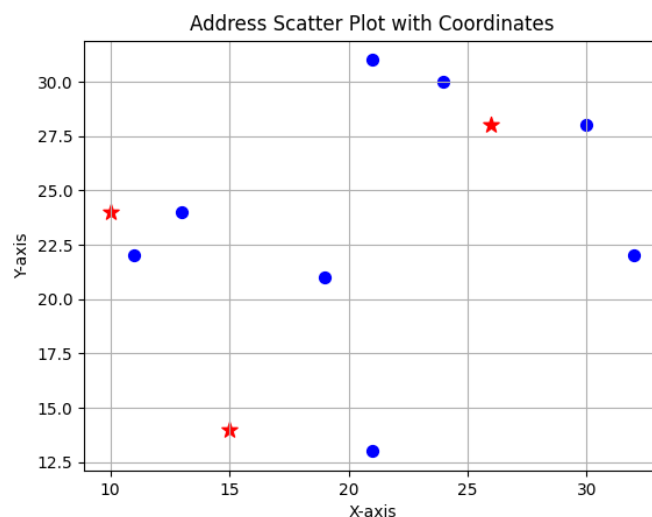


图 5.2 配货点和卸货点的可视化图

构建出邻接字典来存储该图。具体构建出的图的字典表示如图 5.3 所示。

```
graph = {
    'dy1': {'dp1': 6.08, 'dp2': 8.94, 'dp8': 8.06},
    'dy2': {'dp3': 4, 'dp4': 5.83, 'dp6': 8.49, 'dp7': 2.83, 'dp8': 9.9},
    'dy3': {'dp2': 2.24, 'dp5': 3, 'dp8': 9.49},
    'dp1': {'dy1': 6.08, 'dp8': 8.25},
    'dp2': {'dy1': 8.94, 'dy3': 2.24, 'dp5': 2.83, 'dp8': 8.06},
    'dp3': {'dy2': 4, 'dp4': 9.49, 'dp6': 6.32, 'dp7': 6.32},
    'dp4': {'dy2': 5.83, 'dp3': 9.49, 'dp7': 3.16},
    'dp5': {'dy3': 3, 'dp2': 2.83, 'dp8': 6.71},
    'dp6': {'dy2': 8.49, 'dp3': 6.32},
    'dp7': {'dy2': 2.83, 'dp3': 6.32, 'dp4': 3.16},
    'dp8': {'dy1': 8.06, 'dy2': 9.9, 'dy3': 9.49, 'dp1': 8.25, 'dp2': 8.06, 'dp5': 6.71}
}
```

图 5.3 图的字典表示

为了更好地体现对应关系，我们在可视化的基础上，加上了可达点之间的边的构建，可视化效果如图 5.4 所示。

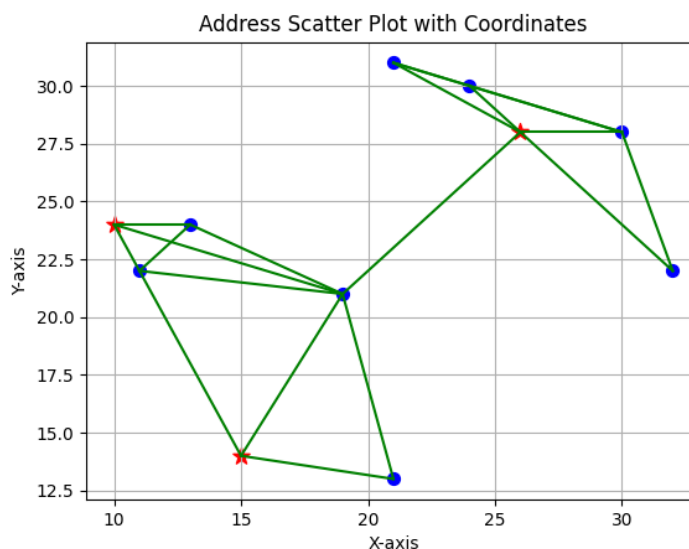


图 5.4 满足条件的回路的可视化效果

上述地址图对应的邻接矩阵如图 5.5 所示。

```
# === all data ===
distance_dp2dp = dis_matrix(DP_position, DP_position)
distance_dy2dp = dis_matrix(DY_position, DP_position)
distance_dp2dp = [[0.0, 100, 100, 100, 100, 100, 100, 8.25],
                  [100, 0.0, 100, 100, 2.83, 100, 100, 8.06],
                  [100, 100, 0.0, 9.49, 100, 6.32, 6.32, 100],
                  [100, 100, 9.49, 0.0, 100, 100, 3.16, 100],
                  [100, 2.83, 100, 100, 0.0, 100, 100, 6.71],
                  [100, 100, 6.32, 100, 100, 0.0, 100, 100],
                  [100, 100, 6.32, 3.16, 100, 100, 0.0, 100],
                  [8.25, 8.06, 100, 100, 6.71, 100, 100, 0.0]]
distance_dy2dp = [[6.08, 8.94, 100, 100, 100, 100, 100, 8.06],
                  [100, 100, 4.0, 5.83, 100, 8.49, 2.83, 9.9],
                  [100, 2.24, 100, 100, 3.0, 100, 100, 9.49]]
```

图 5.5 图对应的邻接矩阵和其计算方式

在本实验中，为了实现高效的回路检索，在初始化阶段便存储可满足的所有路径信息。所有用到的路径信息表构建结果如下图所示。所有变量命名均对应于算法中所使用到的变量。

```
dy_list = ['dy1', 'dy2', 'dy3']
dp_list = ['dp1', 'dp2', 'dp3', 'dp4', 'dp5', 'dp6', 'dp7', 'dp8']
# routes_sat, routes_len = find_cycles_sat(graph)
routes_sat = [['dy1', 'dp1', 'dy1'],
              ['dy1', 'dp2', 'dy1'],
              ['dy1', 'dp8', 'dy1'],
              ['dy2', 'dp3', 'dy2'],
              ['dy2', 'dp3', 'dp4', 'dy2'],
              ['dy2', 'dp3', 'dp4', 'dp7', 'dy2'],
              ['dy2', 'dp3', 'dp6', 'dy2'],
              ['dy2', 'dp3', 'dp7', 'dy2'],
              ['dy2', 'dp3', 'dp7', 'dp4', 'dy2'],
              ['dy2', 'dp4', 'dy2'],
              ['dy2', 'dp4', 'dp7', 'dy2'],
              ['dy2', 'dp6', 'dy2'],
              ['dy2', 'dp7', 'dy2'],
              ['dy2', 'dp8', 'dy2'],
              ['dy3', 'dp2', 'dy3'],
              ['dy3', 'dp2', 'dp5', 'dy3'],
              ['dy3', 'dp2', 'dp8', 'dy3'],
              ['dy3', 'dp5', 'dy3'],
              ['dy3', 'dp5', 'dp8', 'dy3'],
              ['dy3', 'dp8', 'dy3']]
```

图 5.6 所有满足里程数小于 20 的回路存储矩阵

上图为所有回路信息，并且仅从配货点出发，遍历所有可能的回路，最终回到配货点。下图为图 5.6 中对应的所有回路的路径累计数值，列表最后的值即为该路径的总路径代价。

```

routes_len = [[0.0, 6.08, 12.16],
               [0.0, 8.94, 17.88],
               [0.0, 8.06, 16.12],
               [0.0, 4.0, 8.0],
               [0.0, 4.0, 13.49, 19.32],
               [0.0, 4.0, 13.49, 16.65, 19.48],
               [0.0, 4.0, 10.32, 18.81],
               [0.0, 4.0, 10.32, 13.15],
               [0.0, 4.0, 10.32, 13.48, 19.31],
               [0.0, 5.83, 11.66],
               [0.0, 5.83, 8.99, 11.82],
               [0.0, 8.49, 16.98],
               [0.0, 2.83, 5.66],
               [0.0, 9.9, 19.8],
               [0.0, 2.24, 4.48],
               [0.0, 2.24, 5.07, 8.07],
               [0.0, 2.24, 10.3, 19.79],
               [0.0, 3.0, 6.0],
               [0.0, 3.0, 9.71, 19.2],
               [0.0, 9.49, 18.98]]

```

图 5.7 对应的所有回路的路径累计数值

为了检索效率，本算法初始化的面向卸货点的路径直飞矩阵和非直飞矩阵如下图所示。具体来说，对于直飞表，仅采用最短的直飞，因为此时可能多个配货点都可以直飞该卸货点，此时仅采用最短距离的配送方式；而非直飞表则包含了其他节点，为了算法在无人机未满载时提升检索其他回路的效率，便于合并多个卸货点的订单。

```

dps = [['dp2', ['dy3', 'dp2', 'dp5', 'dy3'], [0.0, 2.24, 5.07, 8.07]],
        ['dp2', ['dy3', 'dp2', 'dp8', 'dy3'], [0.0, 2.24, 10.3, 19.79]],
        ['dp3', ['dy2', 'dp3', 'dp7', 'dy2'], [0.0, 4.0, 10.32, 13.15]],
        ['dp3', ['dy2', 'dp3', 'dp6', 'dy2'], [0.0, 4.0, 10.32, 18.81]],
        ['dp3', ['dy2', 'dp3', 'dp7', 'dp4', 'dy2'], [0.0, 4.0, 10.32, 13.48, 19.31]],
        ['dp3', ['dy2', 'dp3', 'dp4', 'dy2'], [0.0, 4.0, 13.49, 19.32]],
        ['dp3', ['dy2', 'dp3', 'dp4', 'dp7', 'dy2'], [0.0, 4.0, 13.49, 16.65, 19.48]],
        ['dp4', ['dy2', 'dp4', 'dp7', 'dy2'], [0.0, 5.83, 8.99, 11.82]],
        ['dp4', ['dy2', 'dp3', 'dp7', 'dp4', 'dy2'], [0.0, 4.0, 10.32, 13.48, 19.31]],
        ['dp4', ['dy2', 'dp3', 'dp4', 'dy2'], [0.0, 4.0, 13.49, 19.32]],
        ['dp4', ['dy2', 'dp3', 'dp4', 'dp7', 'dy2'], [0.0, 4.0, 13.49, 16.65, 19.48]],
        ['dp5', ['dy3', 'dp2', 'dp5', 'dy3'], [0.0, 2.24, 5.07, 8.07]],
        ['dp5', ['dy3', 'dp5', 'dp8', 'dy3'], [0.0, 3.0, 9.71, 19.2]],
        ['dp6', ['dy2', 'dp3', 'dp6', 'dy2'], [0.0, 4.0, 10.32, 18.81]],
        ['dp7', ['dy2', 'dp4', 'dp7', 'dy2'], [0.0, 5.83, 8.99, 11.82]],
        ['dp7', ['dy2', 'dp3', 'dp7', 'dy2'], [0.0, 4.0, 10.32, 13.15]],
        ['dp7', ['dy2', 'dp3', 'dp7', 'dp4', 'dy2'], [0.0, 4.0, 10.32, 13.48, 19.31]],
        ['dp7', ['dy2', 'dp3', 'dp4', 'dp7', 'dy2'], [0.0, 4.0, 13.49, 16.65, 19.48]],
        ['dp8', ['dy3', 'dp5', 'dp8', 'dy3'], [0.0, 3.0, 9.71, 19.2]],
        ['dp8', ['dy3', 'dp2', 'dp8', 'dy3'], [0.0, 2.24, 10.3, 19.79]]

```

图 5.8 面向卸货点的非直飞路径信息矩阵


```
dy2dp_direct = [['dp1', 12.16, ['dy1', 'dp1', 'dy1']],
                 ['dp2', 4.48, ['dy3', 'dp2', 'dy3']],
                 ['dp3', 8.0, ['dy2', 'dp3', 'dy2']],
                 ['dp4', 11.66, ['dy2', 'dp4', 'dy2']],
                 ['dp5', 6.0, ['dy3', 'dp5', 'dy3']],
                 ['dp6', 16.98, ['dy2', 'dp6', 'dy2']],
                 ['dp7', 5.66, ['dy2', 'dp7', 'dy2']],
                 ['dp8', 16.12, ['dy1', 'dp8', 'dy1']]]
```

图 5.9 面向卸货点的直飞路径信息矩阵

5.2 求解 24 小时内的最短路径

本次实验将时间设置为 24 小时，即一天内的无人机路径规划。具体输出的结果如下图所示。

```
assign_order
/opt/conda/bin/python3.10 /xcy_stylewm/homework/assign_order.py
[['dp1', 12.16, ['dy1', 'dp1', 'dy1']], ['dp2', 17.88, ['dy1', 'dp2', 'dy1']], ['dp8', 16.12, ['dy1', 'dp8', 'dy1']], ['dp3', 8.0, ['dy2', 'dp3', 'dy2']],
[['dp2', ['dy3', 'dp2', 'dp5', 'dy3']], [0.0, 2.24, 5.07, 8.07]], ['dp2', ['dy3', 'dp2', 'dp8', 'dy3']], [0.0, 2.24, 10.3, 19.79]], ['dp3', ['dy2', 'dp3', 'dy2']],
在第0分钟时，卸货点[21, 13]当前所有订单最迟送达时间：[30, 30, 30, 90, 90, 90, 90, 180, 180, 180]
在第0分钟时，卸货点[11, 22]当前所有订单最迟送达时间：[30, 30, 30, 30, 30, 30, 30, 90, 90, 90, 180]
在第0分钟时，卸货点[30, 28]当前所有订单最迟送达时间：[30, 30, 30, 90, 90, 90, 90, 90, 180, 180, 180]
在第0分钟时，卸货点[21, 31]当前所有订单最迟送达时间：[30, 30, 30, 30, 90, 90, 90, 90, 90, 180, 180]
在第0分钟时，卸货点[13, 24]当前所有订单最迟送达时间：[30, 30, 30, 30, 30, 90, 90, 90, 90, 180, 180, 180]
在第0分钟时，卸货点[32, 22]当前所有订单最迟送达时间：[30, 30, 30, 30, 30, 90, 90, 90, 90, 90, 180, 180, 180]
在第0分钟时，卸货点[24, 30]当前所有订单最迟送达时间：[30, 30, 30, 30, 30, 30, 30, 90, 90, 90, 90, 90, 90, 90, 90, 180, 180, 180, 180]
在第0分钟时，卸货点[19, 21]当前所有订单最迟送达时间：[90, 90]
order_urgent_list: []
在第10分钟时，卸货点[21, 13]当前所有订单最迟送达时间：[30, 30, 30, 40, 40, 40, 40, 40, 40, 40, 40, 40, 40, 90, 90, 90, 90, 100, 100, 100, 100, 180, 180, 180, 180, 180, 180]
在第10分钟时，卸货点[11, 22]当前所有订单最迟送达时间：[30, 30, 30, 30, 30, 30, 30, 30, 40, 40, 40, 40, 40, 40, 90, 90, 90, 90, 90, 100, 100, 100, 100, 180, 180, 180]
在第10分钟时，卸货点[30, 28]当前所有订单最迟送达时间：[30, 30, 30, 40, 40, 40, 40, 40, 90, 90, 90, 90, 90, 90, 90, 100, 100, 100, 180, 180, 180, 180, 190, 190]
在第10分钟时，卸货点[21, 31]当前所有订单最迟送达时间：[30, 30, 30, 30, 90, 90, 90, 90, 90, 90, 90, 180, 180]
在第10分钟时，卸货点[13, 24]当前所有订单最迟送达时间：[30, 30, 30, 30, 30, 40, 40, 40, 40, 40, 40, 40, 40, 90, 90, 90, 90, 100, 100, 100, 180, 180, 180, 180, 180, 180, 180]
在第10分钟时，卸货点[32, 22]当前所有订单最迟送达时间：[30, 30, 30, 30, 30, 40, 40, 40, 40, 40, 40, 40, 40, 40, 90, 90, 90, 90, 90, 100, 100, 100, 180, 180, 180, 180]
在第10分钟时，卸货点[24, 30]当前所有订单最迟送达时间：[30, 30, 30, 30, 30, 30, 30, 40, 90, 90, 90, 90, 90, 90, 90, 90, 90, 90, 100, 100, 180, 180, 180, 180, 180, 190, 190]
在第10分钟时，卸货点[19, 21]当前所有订单最迟送达时间：[40, 40, 40, 90, 90, 100, 100, 100, 100, 180, 180, 190, 190]
order_urgent_list: []
在第20分钟时，卸货点[21, 13]当前所有订单最迟送达时间：[30, 30, 30, 40, 40, 40, 40, 40, 40, 40, 40, 40, 40, 50, 50, 50, 50, 90, 90, 90, 90, 100, 100, 100, 100, 180, 180]
在第20分钟时，卸货点[11, 22]当前所有订单最迟送达时间：[30, 30, 30, 30, 30, 30, 30, 30, 40, 40, 40, 40, 40, 40, 40, 50, 90, 90, 90, 90, 90, 100, 100, 100, 100, 180, 180]
在第20分钟时，卸货点[30, 28]当前所有订单最迟送达时间：[30, 30, 30, 40, 40, 40, 40, 40, 50, 90, 90, 90, 90, 90, 90, 90, 90, 100, 100, 100, 180, 180, 180, 180, 190, 190]
在第20分钟时，卸货点[21, 31]当前所有订单最迟送达时间：[30, 30, 30, 30, 50, 50, 50, 50, 90, 90, 90, 90, 90, 90, 90, 90, 110, 110, 110, 110, 110, 110, 110, 110, 180, 180]
在第20分钟时，卸货点[13, 24]当前所有订单最迟送达时间：[30, 30, 30, 30, 40, 40, 40, 40, 40, 40, 40, 40, 40, 40, 50, 90, 90, 90, 90, 90, 100, 100, 100, 110, 110, 110, 180]
在第20分钟时，卸货点[32, 22]当前所有订单最迟送达时间：[30, 30, 30, 30, 30, 40, 40, 40, 40, 40, 40, 40, 40, 40, 40, 40, 40, 90, 90, 90, 90, 90, 90, 90, 90, 100, 100, 100]
在第20分钟时，卸货点[24, 30]当前所有订单最迟送达时间：[30, 30, 30, 30, 30, 30, 30, 40, 50, 50, 50, 50, 50, 50, 50, 90, 90, 90, 90, 90, 90, 90, 90, 90, 90, 90, 90, 100, 110]
在第20分钟时，卸货点[19, 21]当前所有订单最迟送达时间：[40, 40, 40, 50, 50, 50, 50, 50, 50, 90, 90, 100, 100, 100, 100, 110, 110, 110, 110, 110, 190, 190, 190]
order_urgent_list: [['dp1', 3], ['dp3', 3], ['dp4', 4], ['dp5', 5], ['dp6', 5], ['dp7', 7], ['dp2', 8]]
在第30分钟时，卸货点[21, 13]当前所有订单最迟送达时间：[40, 40, 50, 50, 50, 50, 60, 60, 90, 90, 90, 90, 100, 100, 100, 100, 110, 110, 110, 120, 120, 120, 120, 180, 180]
在第30分钟时，卸货点[11, 22]当前所有订单最迟送达时间：[40, 40, 40, 40, 50, 50, 90, 90, 90, 90, 90, 90, 100, 100, 100, 100, 110, 110, 180, 190, 190, 200, 200]
在第30分钟时，卸货点[30, 28]当前所有订单最迟送达时间：[40, 40, 40, 40, 50, 90, 90, 90, 90, 90, 90, 90, 90, 90, 90, 90, 90, 100, 100, 100, 180, 180, 180, 190, 190, 200]
在第30分钟时，卸货点[21, 31]当前所有订单最迟送达时间：[50, 50, 50, 50, 60, 60, 60, 60, 60, 90, 90, 90, 90, 90, 90, 90, 90, 110, 110, 110, 110, 110, 110, 110, 110, 110, 110]
在第30分钟时，卸货点[13, 24]当前所有订单最迟送达时间：[40, 40, 40, 40, 40, 40, 50, 60, 60, 60, 60, 60, 60, 60, 90, 90, 90, 90, 90, 90, 90, 90, 90, 90, 90, 90, 90, 90]
在第30分钟时，卸货点[32, 22]当前所有订单最迟送达时间：[40, 40, 40, 40, 40, 40, 40, 40, 40, 40, 40, 40, 40, 40, 40, 90, 90, 90, 90, 90, 90, 90, 90, 90, 90, 90, 90, 90]
在第30分钟时，卸货点[24, 30]当前所有订单最迟送达时间：[40, 50, 50, 50, 50, 50, 50, 60, 60, 60, 60, 60, 60, 60, 90, 90, 90, 90, 90, 90, 90, 90, 90, 90, 90, 90, 90, 90]
在第30分钟时，卸货点[19, 21]当前所有订单最迟送达时间：[40, 40, 40, 50, 50, 50, 50, 50, 50, 60, 60, 60, 60, 60, 90, 90, 90, 90, 90, 90, 90, 90, 90, 90, 90, 90, 90, 90]
order_urgent_list: [['dp7', 1], ['dp1', 2], ['dp8', 3], ['dp2', 5], ['dp3', 5], ['dp5', 7], ['dp6', 10]]
```

图 5.10 前 40 分钟的路径决策

如图 5.10 所示，第 0 分钟到第 10 分钟的决策的紧急订单列表为空，因为前 20 分钟内，没有要求在 10 分钟内送达的订单（本实验最紧急为 30 分钟）。

从第 20 分钟开始，紧急订单列表开始运行，即算法开始为该列表规划最短路径。可以看到，在第 30 分钟开始决策时，第 20 分钟决策时的订单已经被送出，列表中不存在存在最迟送达时间为第 30 分钟的订单。这也表明了算法成功将订单准时准点送达卸货点。

如图 5.11 所示，图中展示了最后 20 分钟的路径规划决策，可以看到在 1430 分钟时，紧急订单列表将所有订单都添加到列表中，并且在最后一轮，也就是第

1440 分钟时可以看到，所有订单已被处理完毕，列表均为空。同时在最后我们还打印了所有轮次遍历的路径数据，包括使用的无人机个数、单次航行路线距离、航行的回路经过的节点。同时最后还输出了总路径信息。

```
assign_order x
↑ order_urgent_list: [['dp2', 3], ['dp5', 3], ['dp8', 8]]
↓ 在第1430分钟时，卸货点[21, 13]当前所有订单最迟送达时间: []
⇨ 在第1430分钟时，卸货点[11, 22]当前所有订单最迟送达时间: []
⇩ 在第1430分钟时，卸货点[30, 28]当前所有订单最迟送达时间: []
📄 在第1430分钟时，卸货点[21, 31]当前所有订单最迟送达时间: [1440, 1440]
🗑 在第1430分钟时，卸货点[13, 24]当前所有订单最迟送达时间: []
  在第1430分钟时，卸货点[32, 22]当前所有订单最迟送达时间: [1440]
  在第1430分钟时，卸货点[24, 30]当前所有订单最迟送达时间: [1440, 1440, 1440, 1440, 1440]
  在第1430分钟时，卸货点[19, 21]当前所有订单最迟送达时间: []
  order_urgent_list: [['dp6', 1], ['dp4', 2], ['dp7', 5]]
  在第1440分钟时，卸货点[21, 13]当前所有订单最迟送达时间: []
  在第1440分钟时，卸货点[11, 22]当前所有订单最迟送达时间: []
  在第1440分钟时，卸货点[30, 28]当前所有订单最迟送达时间: []
  在第1440分钟时，卸货点[21, 31]当前所有订单最迟送达时间: []
  在第1440分钟时，卸货点[13, 24]当前所有订单最迟送达时间: []
  在第1440分钟时，卸货点[32, 22]当前所有订单最迟送达时间: []
  在第1440分钟时，卸货点[24, 30]当前所有订单最迟送达时间: []
  在第1440分钟时，卸货点[19, 21]当前所有订单最迟送达时间: []
  order_urgent_list: []
  [[], [], [[1, 12.16, ['dy1', 'dp1', 'dy1']]], [[1, 16.98, ['dy2', 'dp6', 'dy2']]],
  6623.679999999947
  Process finished with exit code 0
```

图 5.11 最后 20 分钟的路径决策

从算法逻辑和输出结果可以得出，本实验设计的算法从最大程度上最小化了无人机的航行距离，并且依据检索表最大化了该最优化算法的效率，使得算法能够在短时间内取得尽可能好的结果。

6 总结

在本次算法设计的实现过程中，我对图的搜索、最短路径搜索、多配送中心的 VRP 问题等有了更深的理解，深刻认识到了设计一个好的算法存在的挑战。在实验的准备中，我发现选择合适的数据结构有利于提升算法的性能，适当的预处理，例如检索表的构建能够极大地提升算法效率，这对于实际的算法应用场景有极大的帮助。

总的来说，本次的无人机配送的总路径最小化实验让我从知识层面和实践层面对多配送中心的 VRP 问题有了更深入的理解，并意识到了在实际项目中需要解决的一些关键问题。通过不断地学习和实践，我相信我能够不断提升学习和实际应用的能力，为未来的实际场景项目打下坚实基础。