《高级算法设计与分析》大作业报告

——2023202210020 郭晟男

1 问题描述与要求

无人机配送路径规划问题：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，实现如下图所示区域的无人机配送的路径规划。在此区域中，共有j个配送中心，任意一个配送中心有用户所需要的商品，其数量无限，同时任一配送中心的无人机数量无限。该区域同时有k个卸货点（无人机只需要将货物放到相应的卸货点即可），假设每个卸货点会随机生成订单，一个订单只有一个商品，但这些订单有优先级别，分为三个优先级别（用户下订单时，会选择优先级别，优先级别高的付费高）：

- 一般：3小时内配送到即可；
- 较紧急：1.5小时内配送到；
- 紧急：0.5小时内配送到。

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

1. 哪些配送中心出动多少无人机完成哪些订单；
2. 每个无人机的路径规划，即先完成那个订单，再完成哪个订单，...，最后返回原来的配送中心；

注意：系统做决策时，可以不对当前的某些订单进行配送，因为当前某些订单可能紧急程度不高，可以累积后和后面的订单一起配送。

目标：一段时间内（如一天），所有无人机的总配送路径最短

约束条件：满足订单的优先级别要求

假设条件：

1. 无人机一次最多只能携带n个物品；
2. 无人机一次飞行最远路程为20公里（无人机送完货后需要返回配送点）；
3. 无人机的速度为60公里/小时；
4. 配送中心的无人机数量无限；
5. 任意一个配送中心都能满足用户的订货需求；
6. 图和订单数据自行设计。

2 实现思路

本人在着手对本问题进行实现时，在完成数据初始化包括地图生成、订单生成等操作之后，最先想到的思路就是使用**贪心算法**来求解，核心想法就是**依次遍历每个订单，将订单分配给距离该订单生成的卸货点最近的配送中心，之后每个配送中心优先配送优先级最高的订单，在优先级相同的情况下，优先配送距离最近的卸货点，直至达到无人机续航路程限制的最远距离或者达到无人机的最大载重量时，派出新的一架无人机进行配送**，之后重复上述的步骤，完成所有订单的配送。

之后为了与贪心算法进行对比，又想了一种其他的实现方法，考虑到本问题的背景与旅行商问题有些相似，故第二种实现方法选择的是相对比较熟悉的**遗传算法**，在完成与贪心算法一致的数据初始化，以及订单生成与处理等操作后，路径规划的核心思路是依次遍历每个订单，将订单分配给距离该订单生成的卸货点最近的配送中心，**之后把每个生成订单的卸货点当做旅行商问题中的城市，在设定评估适应度（代码中实现的适应度函数不仅考虑路径规划总路程最短，还一并考虑了优先级的因素），以及完成种群初始化之后，依次进行选择、交叉和变异操作，通过遗传算法生成最优路径，分配给无人机执行，如果达到无人机续航路程限制的最远距离或者最大载重量的约束条件时时，派出新的一架无人机进行配送，该新无人机的路径继续由遗传算法得到**，具体的遗传算法设计会在第4节中详细介绍。

贪心算法和遗传算法都是按照规划的路径执行配送任务，更新订单状态，之后记录无人机的总飞行距离和出动次数。最后，汇总两种算法在不同时间点的实验结果，绘制图表，分析遗传算法和贪心算法的性能，得出结论。另外，还进行了遗传算法中迭代次数对路径规划结果影响的实验，并进行了实验分析与结论总结。

两种算法的实验公共部分设置（如地图生成，订单生成，无人机参数设置等）保持一致，将在第3节中详细介绍，并详细介绍公共部分的代码实现。

3 实验公共部分设置

3.1 数据初始化

- **地图生成:**

本代码使用`pygame`库实现可视化操作以及地图生成，本文生成一幅500px*500px的地图，代表实际边长为20km的正方形地图。设定配送中心数量为5，然后再生成的地图中随机生成5个配送中心的坐标（为了便于实验结果的复现，根据论文`《Torch.manual_seed(3407) is all you need: On the influence of random seeds in deep learning architectures for computer vision》`将随机种子设置为3047）。

之后根据题意——无人机可以解决最后10公里的物流配送问题，所以在每个配送中心为圆心，半径为10km（在图中为250px）的范围内设置卸货点，设置的个数为每个配送中心周围设置随机8到10个卸货点，完成配送中心和卸货点的位置的初始化。下面即为卸货点生成函数：

```python
# 生成卸货点函数
def generate_delivery_points(center, num_points_range, distance_px):
    points = []
    for _ in range(random.randint(*num_points_range)):
        while True:
            angle = random.uniform(0, 2 * np.pi)
            r = random.uniform(0, distance_px)
            x = center[0] + r * np.cos(angle)
            y = center[1] + r * np.sin(angle)
            if 0 <= x <= 500 and 0 <= y <= 500:
                points.append((int(x), int(y)))
                break  # 满足条件后退出循环
    return points
```

之后根据像素到米的转换系数，本代码中将像素和km都转换为了以十米为单位，生成配送中心以及各配送点之间的距离矩阵，便于后续调用，防止由于重复计算导致的效率过于低下。各节点的坐标和距离矩阵通过计算欧几里得距离获得，生成距离矩阵的函数如下：

```python
# 计算距离矩阵函数
def distance_matrix(locations):
    # 假设的像素到米的转换系数
    pixels_to_meters = 2000/500 # 500px:20km
    size = len(locations)
    dist_matrix = np.zeros((size, size))
    for i in range(size):
        for j in range(size):
            if i != j:
                dist_matrix[i][j] = np.linalg.norm(np.array(locations[i]) -
np.array(locations[j])) * pixels_to_meters
    return dist_matrix
```

- **其他参数设置：**

参数（变量名）	数值
无人机速度（drone_speed）	60 km/h
无人机最大载重量（max_drone_capacity）	5个
无人机最大续航距离（max_drone_range）	20 km
模拟时间段（time_period）	60 min
订单生成间隔（t）	10 min

3.2 订单生成与处理

`generate_orders` 函数用于在指定的配送点生成随机订单。每个订单包含一个唯一标识符、配送点位置及其索引、优先级、生成时间和截止时间。优先级分为三类（0：紧急（30分钟），1：较紧急（90分钟），2：一般（180分钟）），每隔t分钟生成订单，并记录订单的下单时间（由 `generate_time` 表示），系统在每个配送点生成最多 `max_orders` 个订单，本代码的实现中设定为3，即每个卸货点将在每个时间段内随机生成0到3个订单。订单的生成基于随机种子 `seed`，确保在实验中可以复现相同的订单生成结果。该函数返回所有生成的订单列表，算法会根据订单的优先级和时间约束，计算紧急度，决定处理顺序，用于后续的路径规划和调度决策。

```
# 生成订单函数
def generate_orders(delivery_points, seed, max_orders=3, current_time=0):
    orders = []
    random.seed(seed)
    order_id = 0  # 添加订单唯一标识符
    for i, point in enumerate(delivery_points):
        num_orders = random.randint(0, max_orders)
        for _ in range(num_orders):
            priority = random.choice([0,1,2])# 优先级列表，越小优先级越高
            order = {
                'id': order_id,
                'point': point,
                'point_index':all_locations.index(point),
                'priority': priority,
                'generate_time': current_time,
                'deadline': current_time + priority_time_limits[priority] * 60  #
转换为秒
            }
            orders.append(order)
            order_id += 1  # 增加订单唯一标识符
    return orders
```

3.3 图形可视化

本代码的可视化部分使用 `pygame` 库实现，`move_drones` 函数用于在 `pygame` 窗口中动态模拟和可视化无人机的配送路径。函数接受一组路径作为输入，并初始化无人机的位置和目标点。每一帧中，函数根据无人机的速度和经过的时间计算新的位置，并更新无人机的当前位置。当无人机到达路径中的下一个目标点时，更新目标点并继续移动。函数在 `pygame` 窗口中绘制配送中心、卸货点、路径和无人机当前位置，通过循环刷新窗口实现无人机的动态可视化移动效果。

```
def move_drones(routes):
    positions = [[all_locations[route[0]]] for route in routes]
    targets = [1] * len(routes)
    # 定义初始时间和速度
    start_time = time.time()
    running = True
    while running:
        for event in pygame.event.get():
            if event.type == pygame.QUIT:
                running = False
        screen.fill((255, 255, 255))
        # 绘制配送中心和卸货点
        for i, center in enumerate(delivery_centers):
            pygame.draw.circle(screen, (0, 0, 255), center, 5)
            label = pygame.font.Font(None, 24).render(f'C{i}', True, (0, 0, 255))
            screen.blit(label, (center[0]+10, center[1]))
        for i, point in enumerate(delivery_points):
            pygame.draw.circle(screen, (255, 0, 0), point, 5)
            label = pygame.font.Font(None, 24).render(f'P{i}', True, (255, 0, 0))
            screen.blit(label, (point[0]+10, point[1]))
        # 绘制路径
        for route in routes:
            for i in range(len(route) - 1):
                start_pos = all_locations[route[i]]
```

```
                end_pos = all_locations[route[i + 1]]
                pygame.draw.line(screen, (74, 74, 74), start_pos, end_pos, 1)
        # 绘制无人机位置
        drone_speed = 60  # 假设无人机速度是 60 单位/秒
        # 更新无人机位置并绘制
        elapsed_time = time.time() - start_time
        for i, route in enumerate(routes):
            if targets[i] < len(route):
                start_pos = all_locations[route[targets[i] - 1]]
                end_pos = all_locations[route[targets[i]]]
                cur_pos = positions[i][-1]
                vector = np.array(end_pos) - np.array(cur_pos)
                distance = np.linalg.norm(vector)

                if distance < drone_speed * elapsed_time:
                    positions[i].append(end_pos)
                    targets[i] += 1
                    start_time = time.time()  # 重置时间
                else:
                    direction = vector / distance
                    new_pos = np.array(cur_pos) + direction * drone_speed *
elapsed_time
                    positions[i].append(tuple(new_pos.astype(int)))
                pygame.draw.circle(screen, (103,255,2), positions[i][-1], 5)
        pygame.display.flip()
        clock.tick(30)  # 降低帧率以减慢动画速度
        # 添加图例
        font = pygame.font.Font(None, 30)
        legend1 = font.render("delivery center", True, (0, 0, 255))
        legend2 = font.render("unloading point", True, (255, 0, 0))
        legend3 = font.render("UAV path", True, (0, 255, 0))
        screen.blit(legend1, (10, 10))
        screen.blit(legend2, (10, 40))
        screen.blit(legend3, (10, 70))
        pygame.display.flip()
        clock.tick(60)
```

3.4 其它函数

```
def calculate_route_distance(route):
    """计算路径的总距离"""
    total_distance = 0
    for i in range(len(route) - 1):
        total_distance += dist_matrix[route[i]][route[i + 1]]
    return total_distance

def validate_route(route):
    """验证路径是否满足无人机的最大飞行距离限制"""
    route_distance = calculate_route_distance(route)
    return route_distance <= max_drone_range / 2  # 单程最大飞行距离为20公里，返回后为
10公里

def split_orders(orders):
    """将订单按最大载重进行拆分"""
```

```
    chunks = [orders[i:i + max_drone_capacity] for i in range(0, len(orders),
max_drone_capacity)]
    return chunks
```

4 遗传算法

4.1 问题建模

遗传算法（Genetic Algorithm, GA）是一种启发式搜索算法，模仿自然进化过程，通过选择、交叉、变异等操作，从种群中不断进化出适应度更高的个体。

使用遗传算法解决无人机配送路径规划问题时，可以将该问题建模为一个变形的旅行商问题（TSP），然后考虑优先级、无人机最大飞行距离以及无人机最大载重量等多重约束。在完成配送中心和卸货点位置的确定之后，各卸货点每隔t分钟生成订单，每个订单有一个卸货点和优先级。每t分钟系统使用遗传算法，依次进行种群初始化、选择、交叉、变异、迭代等操作进行一次决策，根据优先级和时间约束，计算每个订单的紧急度，决定哪些订单需要立即处理，哪些订单可以延迟，然后确定无人机的最优规划路径，派出无人机完成配送任务。

4.2 算法设计

- **染色体编码**

在该问题中，每个染色体表示一个完整的配送路径包括起点、若干个卸货点和返回起点的路径。染色体由一系列基因（节点）组成，每个基因表示一个配送点的索引。染色体的结构如下：

```
[start_point, delivery_point_1, delivery_point_2, ..., delivery_point_n,
start_point]
```

- **种群初始化**

种群初始化通过`create_individual`函数生成多个初始染色体。函数以生成订单的卸货点的索引列表为初始化，保证起点为配送中心不变，然后对其余订单点进行随机洗牌，以增加初始种群的多样性，每个个体表示一条可能的配送路径。此过程为遗传算法提供了多样化的初始解，有助于在进化过程中找到全局最优解。

```
def create_individual(order_points_indices_with_center):
    """ 创建基于订单点索引的个体，长度等于订单点数量 """
    start_point = order_points_indices_with_center[0]  # 假设起点是第一个位置
    remaining_points = order_points_indices_with_center[1:]  # 除去起点的其他位置
    random.shuffle(remaining_points)  # 洗牌操作
    return creator.Individual([start_point] + remaining_points)
```

- **适应度函数**

`evalTSP_with_priority`函数用于评估遗传算法中每个个体（配送路径）的适应度。函数计算路径中所有连续节点之间的总距离，并累加路径中订单的优先级权重。首先，函数遍历路径上的每一段，如果发现任何一段路径的距离为零，则标记路径为无效。然后，将路径的总距离和优先级权重累加起来，最后计算加权适应度值。适应度值的计算公式如下，总飞行距离和优先级加权和，本次代码中设置$Weight_{distance}$值为1，$Weight_{priority}$值为100：

$$Fitness = Weight_{distance} * Distance_{total} + Weight_{priority} * Priority_{total}$$

若路径无效或总距离为零，函数返回无穷大的惩罚值，以确保该路径不会被选择。此适应度函数结合了路径的总飞行距离和订单的优先级，帮助遗传算法在多目标优化问题中找到最优解。

```
def evalTSP_with_priority(individual, priorities, distance_weight=1.0,
priority_weight=100):
    distance = 0
    total_priority = 0
    valid_path = True  # 假设路径初始有效
    # 计算路径上所有连续城市之间的距离
    for i in range(1, len(individual)):
        if dist_matrix[individual[i-1]][individual[i]] == 0:
            valid_path = False  # 如果发现两个连续城市之间的距离为0，标记路径为无效
        distance += dist_matrix[individual[i-1]][individual[i]]
        total_priority += priorities[i]
    # 添加从最后一个城市回到第一个城市的距离
    if dist_matrix[individual[-1]][individual[0]] == 0:
        valid_path = False
    distance += dist_matrix[individual[-1]][individual[0]]
    total_priority += priorities[0]
    # 如果路径无效，返回一个非常大的数作为惩罚
    if not valid_path or distance == 0:
        return (float('inf'),)  # 使用无穷大作为惩罚值，确保这种路径不会被选择
    # 计算加权适应度值，优先级越高，适应度值应越低
    fitness_value = distance_weight * distance + priority_weight * total_priority
    return (fitness_value,)  # 返回元组形式的适应度
```

- **选择**

使用锦标赛选择，选择适应度高的个体进入下一代。本代码中锦标赛规模设置为3。

```
toolbox.register("select", tools.selTournament, tournsize=3)
```

- **交叉**

本代码实现的是单点交叉，`custom_crossover` 函数用于遗传算法中的交叉操作，旨在生成新的个体（子代）。该函数接收两个父代个体，选择一个有效的交叉点 `cxpoint`（确保起点保留，仅交叉路径的其他部分）。然后，交换两个父代个体从交叉点开始的部分基因序列，将父代个体的部分基因组合成新的个体，提高了种群的多样性，进而提升遗传算法的搜索能力和解的质量。

```
def custom_crossover(ind1, ind2):
    # 确保两个个体至少有两个元素，才能进行交叉和随机互换操作
    if len(ind1) > 1 and len(ind2) > 1:
        size = min(len(ind1), len(ind2))
        if size > 1:# 保留起点，仅交叉其他部分
            cxpoint = random.randint(1, size - 1) # 选择有效的交叉点
            ind1[cxpoint:], ind2[cxpoint:] = ind2[cxpoint:], ind1[cxpoint:]
    return ind1, ind2
```

- **变异**

本代码实现 `custom_mutation` 函数用于变异操作，随机改变个体的一部分基因，提高种群的多样性，防止算法陷入局部最优。该函数接收一个个体 `individual` 和变异概率 `indpb` 作为参数。首先，函数生成一个随机数，如果该随机数小于变异概率 `indpb`，则执行变异操作。在保证起点不变的前提下，变异操作有两种方式：如果个体长度大于1，随机选择个体中的一个位置并随机改变该位置的城市索引；或者随机交换个体中两个城市的位置。

```
def custom_mutation(individual, indpb):
    if random.random() < indpb:
        if len(individual) > 1 :
            index = random.randint(1, len(individual) - 1)  # 避免选择起点
            if random.random() > 0.5:  # 变异：随机改变一个城市
                individual[index] = random.choice(range(len(individual)))
            else:  # 变异：随机交换两个城市的位置，除了起点
                index2 = random.randint(1, len(individual) - 1)
                individual[index], individual[index2] = individual[index2],
individual[index]
    return (individual,)
```

- **迭代**

算法通过不断迭代，重复选择、交叉、变异等操作，使得种群中的个体逐渐进化。经过多代进化后，直至达到最大迭代次数或找到满意的解，即最优或近似最优的配送路径。根据最优染色体，分配无人机执行配送任务。

- **遗传算法参数设置**

参数（变量名）	值
种群规模（*pop_size*）	100
交叉概率（*cxpb*）	0.7
变异概率（*mutpb*）	0.2
最大迭代次数（*ngen*）	200
精英个体数量（*elite_size*）	5

4.3 遗传算法流程函数

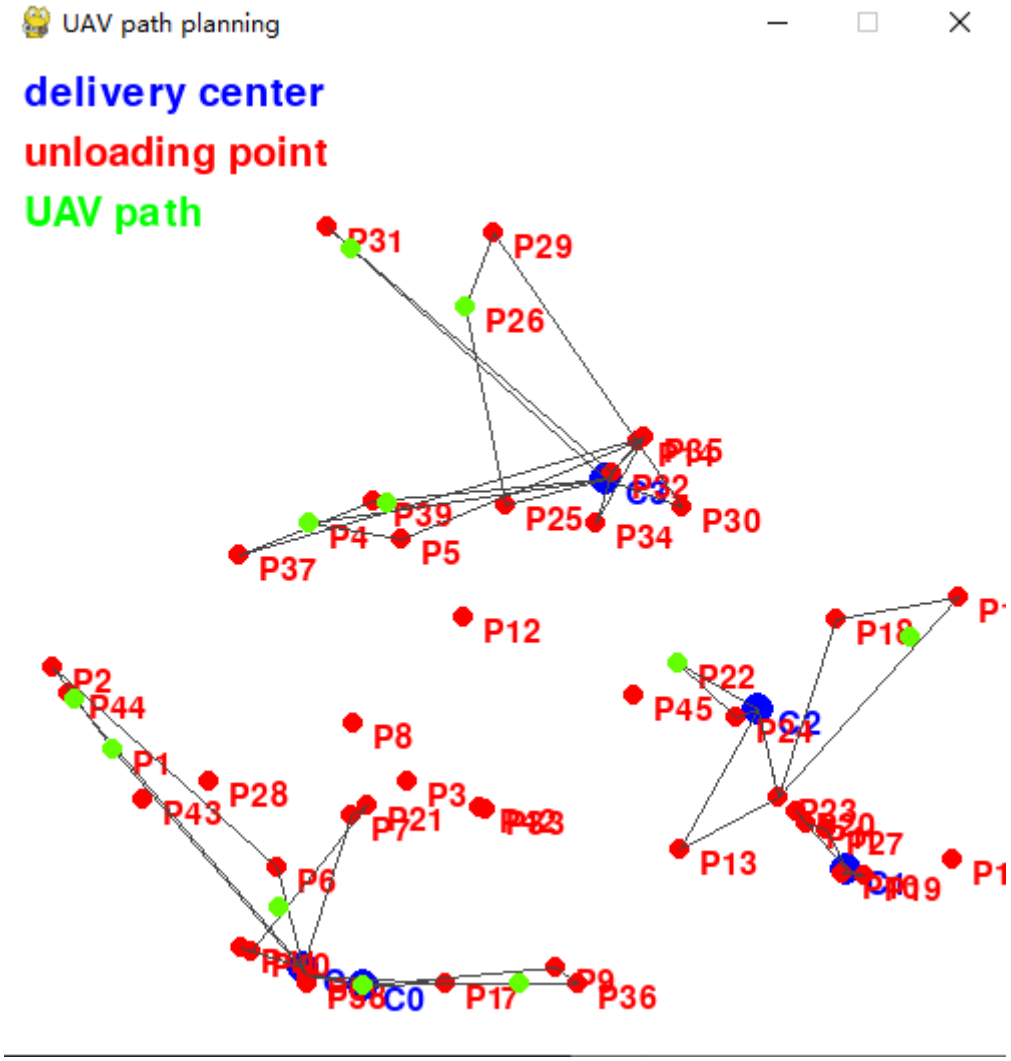
该函数实现了基于遗传算法的无人机配送路径规划。调用DEAP 库中提供的一个标准的遗传算法框架，接受初始化的种群、工具箱，通过初始化种群、交叉、变异和选择操作，迭代优化个体的适应度值，从而找到最优或近似最优的配送路径。最后根据无人机的最大飞行距离和载重量限制进行路径分隔，将最优路径分割为若干子路径。

```
def genetic_algorithm_tsp(order_points, start_index, max_distance, pop_size=100,
cxpb=0.7, mutpb=0.2, ngen=200, elite_size=5):
    pop = toolbox.population_custom(n=pop_size) # 创建种群
    hof = tools.HallOfFame(elite_size)
    stats = tools.Statistics(lambda ind: ind.fitness.values)
    stats.register("avg", np.mean)
    stats.register("min", np.min)
    drones_used = 0
    for gen in range(ngen):
        offspring = tools.selBest(pop, elite_size)
        offspring.extend(toolbox.select(pop, len(pop) - elite_size))
        offspring = algorithms.varAnd(offspring, toolbox, cxpb, mutpb)
        fits = map(toolbox.evaluate, offspring)
        for fit, ind in zip(fits, offspring):
            ind.fitness.values = fit
        pop[:] = offspring
```

```
        hof.update(pop)
        record = stats.compile(pop)
        print(f"Generation {gen}: {record}")
    best_route = hof[0]

    all_routes = [] # 分割路径以满足最大距离限制
    current_route = [start_index]  # 当前子路径，初始包含起点
    current_distance = 0  # 当前子路径的总距离
    for i in range(1, len(best_route)):
        next_point = best_route[i]  # 下一个点
        distance_to_next = dist_matrix[current_route[-1]][next_point]
        distance_back_to_start = dist_matrix[next_point][start_index]
        if current_distance + distance_to_next + distance_back_to_start <=
max_distance:
            current_route.append(next_point)  # 添加下一个点到当前子路径
            current_distance += distance_to_next  # 更新当前子路径的总距离
            if(len(current_route) - 1 > max_drone_capacity):
                current_route.pop() # 去掉新点
                current_distance -= distance_to_next
                current_route.append(start_index)
                all_routes.append(current_route)
                current_route = [start_index, next_point]
                current_distance = dist_matrix[start_index][next_point]
        else:
            current_route.append(start_index)  # 当前子路径回到起点
            all_routes.append(current_route)  # 将当前子路径添加到所有子路径中
            current_route = [start_index, next_point]  # 开始新的子路径
            current_distance = dist_matrix[start_index][next_point]  # 重置当前子路
径的总距离
    current_route.append(start_index)
    all_routes.append(current_route)
    drones_used = len(all_routes)
    return all_routes, drones_used
```

4.4 实验结果截图

本小节主要展示实验结果截图，第6节将进行详细的结果分析与对比。下图为遗传算法的可视化界面：



下图为遗传算法的输出截图：

```
Generation 185: {'avg': np.float64(3309.8602163767646), 'min': np.float64(3309.8602163767637)}
Generation 186: {'avg': np.float64(3316.4697465853533), 'min': np.float64(3309.8602163767637)}
Generation 187: {'avg': np.float64(inf), 'min': np.float64(3309.8602163767637)}
Generation 188: {'avg': np.float64(3310.152232750815), 'min': np.float64(3309.8602163767637)}
Generation 189: {'avg': np.float64(inf), 'min': np.float64(3309.8602163767637)}
Generation 190: {'avg': np.float64(3309.8602163767646), 'min': np.float64(3309.8602163767637)}
Generation 191: {'avg': np.float64(3324.616620707952), 'min': np.float64(3309.8602163767637)}
Generation 192: {'avg': np.float64(3313.9146671568506), 'min': np.float64(3287.7919327895997)}
Generation 193: {'avg': np.float64(3340.6925415557216), 'min': np.float64(3287.7919327895997)}
Generation 194: {'avg': np.float64(inf), 'min': np.float64(3287.7919327895997)}
Generation 195: {'avg': np.float64(3293.3090036863928), 'min': np.float64(3287.7919327895997)}
Generation 196: {'avg': np.float64(3290.0079514812746), 'min': np.float64(3287.7919327895997)}
Generation 197: {'avg': np.float64(3303.6911744352174), 'min': np.float64(3287.7919327895997)}
Generation 198: {'avg': np.float64(3293.1964017503615), 'min': np.float64(3287.7919327895997)}
Generation 199: {'avg': np.float64(3287.791932789601), 'min': np.float64(3287.7919327895997)}
Best route indices: [[4, 43, 5, 12, 8, 4, 4], [4, 12, 26, 4, 43, 45, 4], [4, 46, 5, 4]]
Drones used for center 4: 3
Best route indices: [4, 43, 5, 12, 8, 4, 4]
Best route actual locations: [(149, 456), (151, 464), (151, 460), (173, 380), (201, 363), (149, 456), (149, 456)]
配送中心(149, 456)的无人机路径规划: [4, 43, 5, 12, 8, 4, 4]
[np.float64(32.984845004941285), np.float64(16.0), np.float64(331.87949620306466), np.float64(131.0267148332736), np.float64(426.20183012277175), np.float64(0.0)]
Best route indices: [4, 12, 26, 4, 43, 45, 4]
Best route actual locations: [(149, 456), (173, 380), (181, 375), (149, 456), (151, 464), (123, 448), (149, 456)]
配送中心(149, 456)的无人机路径规划: [4, 12, 26, 4, 43, 45, 4]
[np.float64(318.7977415227404), np.float64(37.73592452822641), np.float64(348.36762191684807), np.float64(32.984845004941285), np.float64(128.9961239727768), np.float64(108.81176406988355)]
Best route indices: [4, 46, 5, 4]
Best route actual locations: [(149, 456), (118, 446), (151, 460), (149, 456)]
配送中心(149, 456)的无人机路径规划: [4, 46, 5, 4]
[np.float64(130.29197979921864), np.float64(143.38758663147937), np.float64(17.88854381999832)]
第1次总飞行距离:112.95244560472612 km
第1次无人机出动总数量:11 架次
```

5 贪心算法

5.1 问题建模

贪心算法（Greedy Agorithm）将无人机配送路径规划问题简化为逐步选择最优路径点的问题，贪心算法每一步都选择当前最优解。

5.2 算法设计

在完成配送中心和卸货点位置的生成之后，各卸货点每隔t分钟生成订单，每个订单有一个卸货点和优先级。首先，依次遍历每个订单，将订单分配给距离该订单生成的卸货点最近的配送中心，对于每个配送中心，初始化无人机路径。之后每个配送中心优先配送优先级最高的订单，在优先级相同的情况下，从当前所在位置选择距离最近的、在最大飞行距离内的卸货点作为下一个目标，计算每个路径的总距离和载重量，确保不超过无人机的限制。直至达到限制时，派出新的一架无人机进行配送。贪心算法在每个决策点选择局部最优解，逐步构建完整路径。根据贪心算法生成的路径，分配无人机执行配送任务。

5.3 贪心算法流程函数

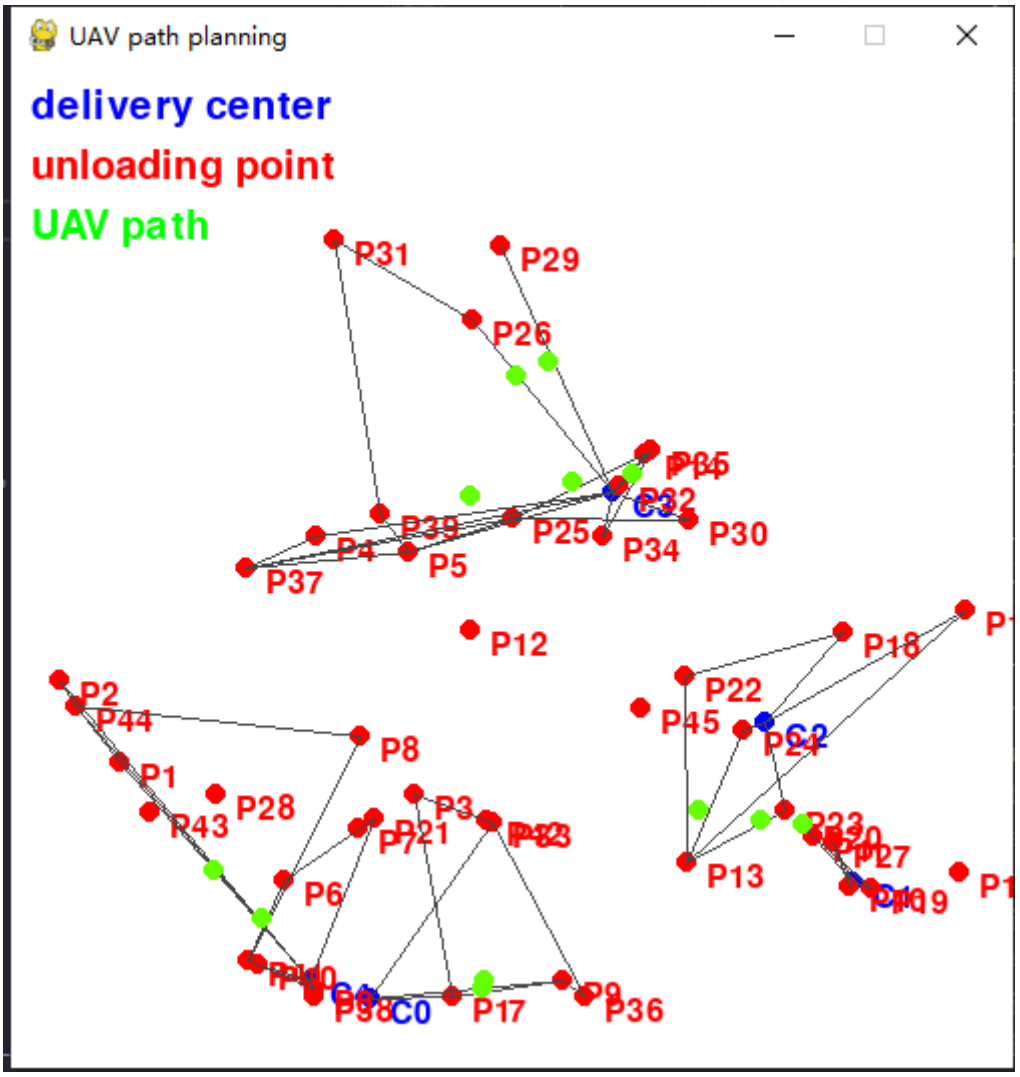
本文在`modified_greedy_algorithm`函数中实现了一种改进的贪心算法，用于无人机配送路径规划。该函数从配送中心出发，优先选择优先级最高且距离最近的订单点，逐步构建配送路径。在构建路径时，函数考虑无人机的最大飞行距离和载重量限制，确保每条路径的总距离和载重量在可行范围内。函数通过循环迭代，生成满足所有约束条件的配送路径，直至所有订单点都被访问。最终返回所有生成的路径和使用的无人机数量。此算法能够快速生成高效的配送路径。

```
def modified_greedy_algorithm(start_index, order_points, priorities,
max_distance):
    unvisited = set(range(len(order_points)))
    drones_used = 0
    all_paths = []

    def can_return_to_start(next_index):
        return (dist_matrix[current_index][next_index] + dist_matrix[next_index]
[start_index]) <= max_distance

    while unvisited:
        path = [start_index]
        current_index = start_index
        total_distance = 0
        distance_next_to_start = 0
        while unvisited:
            feasible_points = [idx for idx in unvisited if
can_return_to_start(order_points[idx])]
            if not feasible_points: break
            next_idx = min(feasible_points, key=lambda idx: (priorities[idx],
dist_matrix[current_index][order_points[idx]])) # 选择优先级最高且最近的点
            next_index = order_points[next_idx]
            distance_to_next = dist_matrix[current_index][next_index]
            distance_next_to_start = dist_matrix[next_index][start_index]
            if total_distance + distance_to_next + distance_next_to_start >
max_distance:   break
            if(len(path)>max_drone_capacity):
                break
            total_distance += distance_to_next
            path.append(next_index)
            unvisited.remove(next_idx)
            current_index = next_index

        path.append(start_index)
        total_distance += distance_next_to_start
        print(unvisited)
        all_paths.append(path)
        drones_used += 1
```

```
    return all_paths, drones_used
```

5.4 实验结果截图

本小节主要展示实验结果截图，第6节将进行详细的结果分析与对比。下图为贪心算法的可视化界面：



下图所示为使用贪心算法进行规划的输出截图：

```
Center: 4
{1, 3, 4, 5, 9}
{9}
set()
Best route indices: [[4, 45, 11, 33, 45, 5, 4], [4, 11, 33, 13, 13, 4], [4, 49, 4]]
Drones used for center 4: 3
配送中心(149, 456)的无人机路径规划: [4, 45, 11, 33, 45, 5, 4]
[np.float64(108.81176406988355), np.float64(175.86358349584486), np.float64(219.27152117865194), np.float64(350.222786237560
5), np.float64(121.85236969382254), np.float64(17.88854381999832)]
配送中心(149, 456)的无人机路径规划: [4, 11, 33, 13, 13, 4]
[np.float64(206.64946164943183), np.float64(219.27152117865194), np.float64(310.4834939252005), np.float64(0.0), np.float64(
498.14054241749886)]
配送中心(149, 456)的无人机路径规划: [4, 49, 4]
[np.float64(720.6441562935205), np.float64(720.6441562935205)]
第6次总飞行距离:122.58265417899665 km
第6次无人机出动总数量:11 架次
第6次生成的订单长度:39 单
总飞行距离:891.7600511173791 km
无人机出动总数量:74 架次
订单总数量:293 单
```

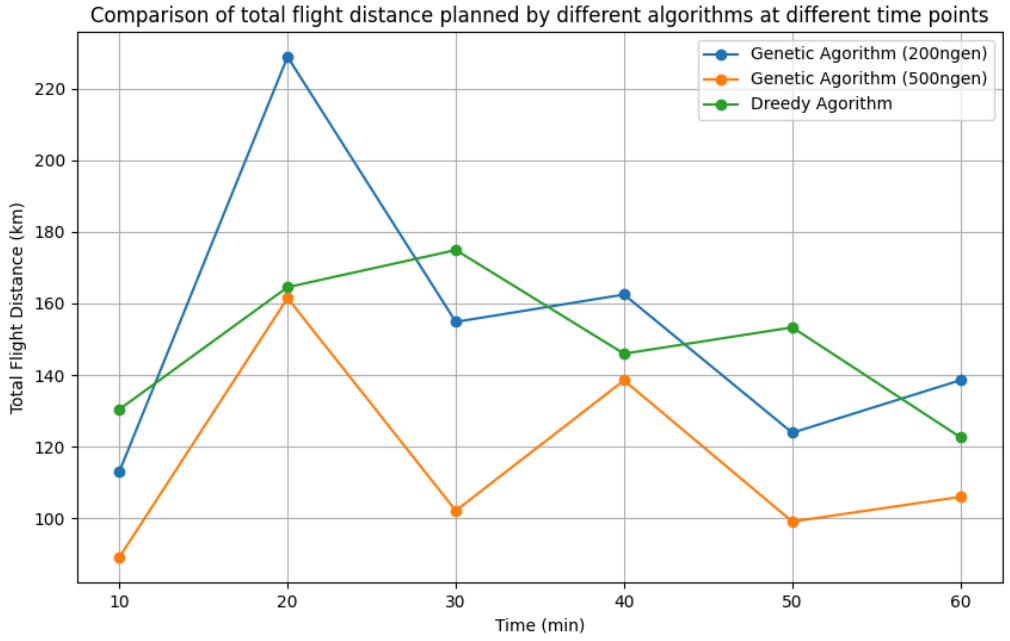
6 实验对比

下表是关于每个时间段的订单生成量统计，作为后面关于不同时间点总飞行距离以及无人机出动总价次的参考。

时间点	10	20	30	40	50	60	总计
订单数量	44	54	56	50	50	39	293

6.1 总飞行距离对比

为了进一步探究迭代次数对于算法性能的影响，故补充了迭代次数为500的遗传算法实验，保证其他参数保持不变。下图为三种算法在不同时间点无人机规划的总飞行距离（km）对比折线图，展示在不同时间点，遗传算法和贪心算法的总飞行距离。：



下表为三种算法在不同时间点无人机规划的总飞行距离的详细数据。可以发现，迭代次数为200的遗传算法在总飞行距离方面不如贪心算法，有一定差距，可能是由于迭代次数不够，远远没收敛到最优解。而增加迭代次数后，迭代次数为500的遗传算法表现优异，特别是在长时间的配送任务中，总距离明显小于贪心算法。

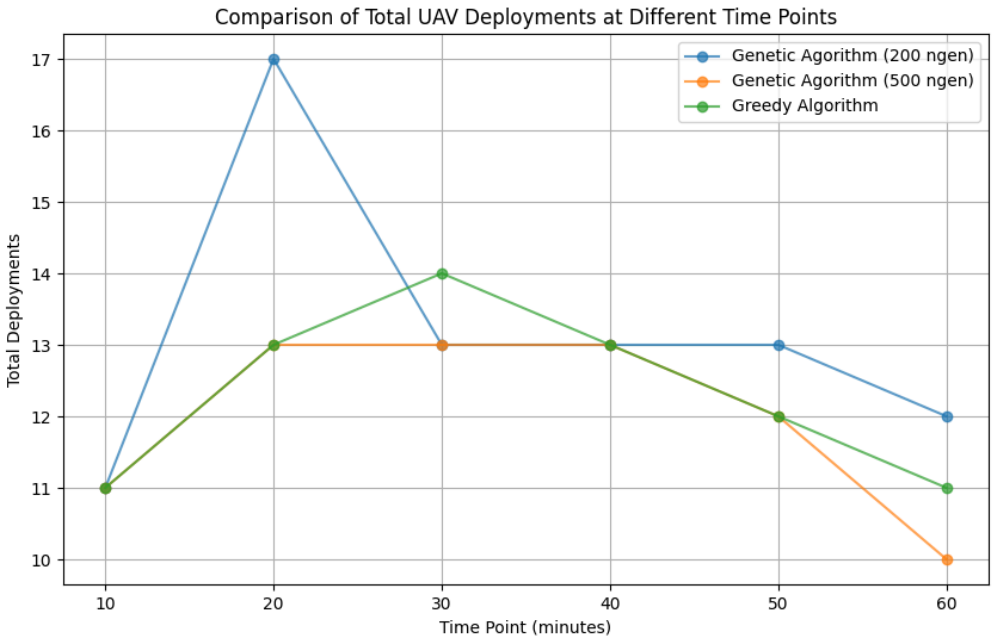
算法\时间点	10	20	30	40	50	60	总计
遗传算法（200ngen）	112.95	228.98	154.86	162.51	123.85	138.56	921.68
遗传算法（500ngen）	89.02	161.53	102.06	138.51	99.02	105.93	696.06
贪心算法	130.39	164.56	174.95	145.96	153.32	122.58	891.76

总体而言，迭代次数为500的遗传算法在无人机路径规划问题中表现最佳，能够显著减少总飞行距离。增加迭代次数（从200到500）显著减少了总飞行距离。这表明更多的迭代次数可以帮助遗传算法找到更优的解。虽然贪心算法简单易实现、计算速度快，但容易陷入局部最优解，可能无法找到全局最优解，在复杂的路径优化问题中，其效果不如迭代次数多的遗传算法理想。但在订单数量较少、实时性要求高、

复杂度低时，由于遗传算法计算量大、收敛速度慢、参数设置复杂，使用贪心算法要优于遗传算法。

6.2 无人机出动总架次对比

下图为三种算法在不同时间点无人机出动总架次的对比，展示在不同时间点，各算法的无人机出动次数：



下表展示的是三种算法在不同时间点无人机出动总架次的详细数据。可知，迭代次数为200的遗传算法d的无人机出动总架次总体上要高于贪心算法，而增加迭代次数（从200到500）明显减少了总出动架次，优于了贪心算法。这表明更多的迭代次数有助于遗传算法找到更优的解决方案，减少无人机的出动次数。特别是在20分钟和60分钟时，500ngen的出动架次显著低于200ngen，说明迭代次数的增加在这些时间点上效果尤为明显。

算法\时间点	10	20	30	40	50	60	总计
遗传算法（200ngen）	11	17	13	13	13	12	79
遗传算法（500ngen）	11	13	13	13	12	10	72
贪心算法	11	13	14	13	12	11	74

总体而言，迭代次数为500的遗传算法在优化无人机出动架次方面表现最佳，能够显著减少总出动架次，表明在高优先级订单处理方面具有优势。在时间成本等条件允许的情况下，增加遗传算法的迭代次数能够有效提升算法的性能，减少无人机的出动次数，提升配送效率。

6.3 实验结论

遗传算法：

- 优点：能找到全局最优解或近似最优解，适用于复杂问题。
- 缺点：计算量大，收敛速度慢，参数设置复杂。
- 适用场景：订单数量大、优先级要求高、需要全局优化时。

贪心算法：

- 优点：简单易实现，计算速度快。

- 缺点：容易陷入局部最优解，可能无法找到全局最优解。
- 适用场景：订单数量较少、实时性要求高。

7 个人感想

不得不说林老师布置的大作业还是非常有水平的，开始刚拿到题目的时候还觉得很容易实现，当真正开始着手实现代码的时候，才发现这次大作业题目的复杂之处，需要考虑的部分还是挺多的。我觉得通过完成这次大作业，进一步提高了我全面思考问题的能力，也使得我对于上课时林老师所讲的算法有了更深的认识与理解。课上、课后收获都很大，很庆幸研究生阶段能够上了这么一门有收获的课程。