Report on Assignment 1

Su Kexin, 24102174G

I. INTRODUCTION

The Traveling Salesman Problem is a classic optimization problem in combinatorial optimization. The objective is to find the shortest possible route that allows a salesman to visit each city exactly once and return to the original city. TSP is known for its complexity, as the number of possible routes increases factorially with the number of cities. It is NP-hard, meaning that no efficient solution exists for larger instances. TSP has practical applications in logistics, planning, and manufacturing, among others.

Genetic Algorithms are a class of optimization algorithms inspired by the principles of natural selection and genetics. They are used to solve complex problems by evolving solutions over generations. The process typically involves the following steps: initialization, selection, crossover, mutation, iteration.

II. METHODOLOGY

A. Task 1

Overall framework: Task 1 uses several steps of genetic algorithm to find the shortest path, mainly through functions to initialize the population, select parents, generate offspring, mutate, update the population, and track the best solution. Through multiple generations of evolution, the algorithm gradually approaches the optimal solution to solve the traveling salesman problem.

The calculate-distance and count_distance functions: Calculate the Euclidean distance between two points and calculate the total distance of a route. Traverse each pair of adjacent cities along the route, accumulate their distances, and finally return the total.

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

```
def total_distance(route):
    distance = 0
    for i in range(len(route)):
        distance += calculate_distance(customers[route[i]], customers[route[(i + 1) % len(route)]])
    return distance
```

Select function: In this function, five individuals are randomly selected, the total distance is calculated for these five individuals, and they are added to the ascending sorting to return the minimum value of tourism [0]. This individual will be selected as the parent to complete the generation of offspring.

```
def select(population):
    tournament = random.sample(population, 5)
    tournament.sort(key=lambda route: total_distance(route))
    return tournament[0]
```

Crossover function: In this function, set the crossover probability to 0.8, randomly select a segment from parent1 as a part of the offspring, and use parent2 to complete the remaining part. During the completion process, cities that have already appeared need to be skipped. When not overcrossing, return parent1 directly as the offspring.

```
def crossover(parent1, parent2):
    if random.random() < 0.8:
        size = len(parent1)
        start, end = sorted(random.sample(range(size), 2))
        child = [None] * size

        #parent1
        child[start:end] = parent1[start:end]

        #parent2
        p2_index = 0
        for i in range(size):
            if child[i] is None:
                while parent2[p2_index] in child:
                    p2_index += 1
                child[i] = parent2[p2_index]
                p2_index += 1

        return child
    else:
        return parent1[:]
```

Mutation function: In this function, the cities at two random points along the route are exchanged with a probability of 0.1, thereby increasing the diversity of the population.

```
def mutate(route):
    if random.random() < 0.1:
        idx1, idx2 = random.sample(range(len(route)), 2)
        route[idx1], route[idx2] = route[idx2], route[idx1]  # swap
```

Genetic_algorithm function: This function is the process of genetic algorithm, including creating random routes to initialize the population, calculating the total distance, setting the number of iterations to determine the total generations, completing the selection of parents, crossing to generate new offspring, completing mutation based on probability, storing new individuals and updating the population, and comparing to generate the shortest distance.

```
def genetic_algorithm(pop_size, num_customers, generations):
    population = create_initial_population(pop_size, num_customers)
    best_route = min(population, key=lambda route: total_distance(route))

    for _ in range(generations):
        new_population = []
        for _ in range(pop_size):
            parent1 = select(population)
            parent2 = select(population)
            child = crossover(parent1, parent2)
            mutate(child)
            new_population.append(child)
        population = new_population

        # Update best route
        current_best = min(population, key=lambda route: total_distance(route))
        if total_distance(current_best) < total_distance(best_route):
            best_route = current_best

    return best_route
```

Parameter settings and chart drawing: visualize the round-trip route and provide the total distance.

```
pop_size = 100
num_customers = len(customers)
generations = 1000
```

```
# Run the genetic algorithm
best_route = genetic_algorithm(pop_size, num_customers, generations)
```

```
# Calculate total distance of the best route
best_distance = total_distance(best_route)
```

```
best_route_coordinates = customers[best_route]
plt.figure(figsize=(10, 6))
plt.plot(best_route_coordinates[:, 0], best_route_coordinates[:, 1], marker='o')
plt.plot([best_route_coordinates[0, 0], best_route_coordinates[-1, 0]],
         [best_route_coordinates[0, 1], best_route_coordinates[-1, 1]], 'ro')  # Return to start
plt.title('Best Route Found')
plt.xlabel('X Coordinate')
plt.ylabel('Y Coordinate')
plt.grid()
plt.show()

print(f'Total distance of the best route: {best_distance}')
```

B. Task 2

Overall framework: Task 2 is to solve the dynamic TSP problem, which lies in the fact that the number and location of customers will change with the environment e. Here, e=0, 1... 5. The environment variable e changes every 100 generations, which means that the location and quantity of customers will change every 100 generations. The changes include: when environment e=0, there are 50 customers, when e=1, there are 60 customers and so on, until e=5, there are 100 customers; And the customer's location will be adjusted according to the environment e:

$$x_{new} = x + 2e \cdot \cos\left(\frac{\pi}{2}e\right),$$
$$y_{new} = y + 2e \cdot \sin\left(\frac{\pi}{2}e\right),$$

Add some functions on the basis of Task 1 to solve the dynamic TSP problem.

get_customers_for_env function: In this function, when e>0, new coordinates are calculated based on the given formula, and an empty list is created to store the updated coordinates. When e=0, the coordinates remain unchanged. The final result returned is a numpy array containing updated coordinates.For each environment e, the number of cities to be considered in the current environment is calculated using the formula num_customs=50+10 * e.

```
def get_customers_for_env(e):
    num_customers = 50 + 10 * e
    adjusted_data = data[:num_customers, :].copy()
    if e != 0:
        factor = 2 * e
        adjusted_data[:, 1] += factor * np.cos(pi / (2 * e))
        adjusted_data[:, 2] += factor * np.sin(pi / (2 * e))
    return adjusted_data
```

main function: Two scenarios for whether to reuse have been added to the main function: not using the genetic algorithm of the previous population:Call genetic_algorithm (customers, reuse_previous=False) to execute the genetic algorithm without using the population from the previous environment. Print the optimal distance without using the previous population in the current environment. Use the genetic algorithm of the previous population: If the preview_population is not None (i.e. at least one environment has been processed), call the genetic_algorithm (customers, reuse_previous=True, preview_population=preview_population) to execute the genetic algorithm, this time using the population from the previous environment. Print the optimal distance for using the previous population in the current environment.

```
def main():
    previous_population = None
    for e in range(6):
        customers = get_customers_for_env(e)
        best_route, best_distance = genetic_algorithm(customers, reuse_previous=False)
        print(f"Environment {e} without reuse: Best Distance = {best_distance}")

        if previous_population is not None:
            best_route_reuse, best_distance_reuse = genetic_algorithm(customers, reuse_previous=True, previous_population=previous_population)
            print(f"Environment {e} with reuse: Best Distance = {best_distance_reuse}")

        previous_population = [best_route[i:i+50+10*e] for i in range(0, len(best_route), 50+10*e)]
```

genetic_algorithm function: Compared to the GA algorithm in Task 1, in this task, the current population is initialized based on whether to reuse the previous population. If reuse_previous is True and preview_population is provided, the previous population will be expanded (by adding new cities to each individual) and merged with the newly created population until the population size POP_SIZE is reached. Otherwise, create a completely new population.

```
def genetic_algorithm(customers, reuse_previous=False, previous_population=None):
    num_customers = len(customers)
    distance_matrix = calculate_distance_matrix(customers)

    if reuse_previous and previous_population:
        population = [ind + list(range(len(ind), num_customers)) for ind in previous_population]
        population += create_initial_population(POP_SIZE - len(population), num_customers)
    else:
        population = create_initial_population(POP_SIZE, num_customers)

    for generation in range(GENS):
        fitnesses = [total_distance(ind, distance_matrix) for ind in population]
        new_population = []
        for _ in range(POP_SIZE):
            parent1 = select(population, distance_matrix)
            parent2 = select(population, distance_matrix)
            child = crossover(parent1, parent2)
            mutate(child)
            new_population.append(child)
        population = new_population

    best_index = np.argmin([total_distance(ind, distance_matrix) for ind in population])
    return population[best_index], total_distance(population[best_index], distance_matrix)
```

C. Task 3

Overall framework: Task 3 is the application of k-means problem. The main logic of this problem is to randomly select K data points as initial cluster centers. For each data point, calculate its distance to each cluster center and assign it to the cluster represented by the nearest cluster center. For each cluster, calculate the mean of all data points in the cluster as the new cluster center. Repeat steps 2 and 3 until a certain stopping condition is met, such as the cluster center no longer changing or reaching the maximum number of iterations. And when selecting routes within the region, continue to use TSP to solve the problem.

Add k-means algorithm on the basis of Task 1.

Data preparation: A copy of the data was created using pandas' copy method and assigned to new_customs. Increase each value by 100. This is equivalent to creating a new set of customer data, whose X coordinate is different from the original data. Use pandas' concat function to merge the original data with the newly created data new_customs.

```
data = pd.read_csv('TSP.csv')

# 给每个顾客的X坐标增加100，并添加新顾客数据
new_customers = data.copy()
new_customers['  XCOORD.   '] += 100
data = pd.concat([data, new_customers], ignore_index=True)
```

K-means clustering: Create an instance of KMeans clustering algorithm, set the number of clusters to 5 (divide customer data into 5 regions), and set random_state=0 to ensure the repeatability of clustering results every time the code is run. Then use the fit method to fit the coordinate data of customers, that is, perform the clustering algorithm calculation process. After clustering is completed, use labels to identify the region to which each customer belongs. Then perform data grouping, clustered_data contains 5 elements, each element is a DataFrame, corresponding to customer data in a clustering area.

```
# K-means进行聚类，假设分为5个区域
kmeans = KMeans(n_clusters=5, random_state=0).fit(customers)
labels = kmeans.labels_

# 将顾客数据按照聚类结果分组
clustered_data = [data[labels == i] for i in range(5)]
```

Calculate the optimal path for each cluster region: First, create an empty list optimal_maths to store the optimal path for each cluster region. Start traversing each cluster region in the clustered_data list. For each clustering region, calculate the number of customers within that region. Call the genetic_algorithm function to calculate the optimal path for the current clustering region. Add the optimal path results for each clustering region to the optimal_caths list.

```
optimal_paths = []
for cluster in clustered_data:
    num_customers_in_cluster = len(cluster)
    ga_result = genetic_algorithm(pop_size=50, num_customers=num_customers_in_cluster, generations=100)
    optimal_paths.append(ga_result)
```

Calculate total path length: Initialize the total path length to 0. Use the zip function to traverse both the optimalbaths list and the clustered_data list simultaneously. For each cluster region, path is the optimal path for that region (a list of customer access orders), and cluster is the corresponding customer data DataFrame for that region. Call the count_destance function to calculate the length of the optimal path in the current clustering area, and add it to the total path length total. By traversing all clustering regions, the total path length of the entire problem is obtained, which is the total distance traveled by the travel agent to visit all customers.

```
total_path_length = 0
for path, cluster in zip(optimal_paths, clustered_data):
    total_path_length += total_distance(path)
```

D. Task 4

Overall framework: In this task, a multi-objective optimization task regarding the traveling salesman problem needs to be solved. Travel agents need to plan a route to visit a range of customer locations. This problem requires meeting two objectives simultaneously: minimizing the travel distance of round-trip routes while also maximizing sales profits.

Method 1 is based on a weighted objective function to solve the multi-objective optimization problem of the traveling salesman problem. By weighting and combining two objective functions (total travel distance and total sales profit), it is transformed into a single objective optimization problem.

Method 2 uses an evolutionary algorithm based on Pareto dominance selection to handle this multi-objective optimization problem. Pareto dominance refers to the situation in multi-objective optimization problems where a solution is not inferior to another solution in all objectives and is strictly superior to another solution in at least one objective.

Below are the details of each section:

1. Method 1

Create a distance matrix: Create a 100 * 100 zero matrix, extract customer locations from the data file, calculate the distance between customers, and fill in the matrix.

```
# 客户之间的欧几里得距离矩阵
def calculate_distance_matrix(data):
    num_customers = len(data)
    distance_matrix = np.zeros((num_customers, num_customers))
    for i in range(num_customers):
        for j in range(num_customers):
            if i!= j:
                x1, y1 = data.loc[i, '  XCOORD.   '], data.loc[i, 'YCOORD.']
                x2, y2 = data.loc[j, '  XCOORD.   '], data.loc[j, 'YCOORD.']
                distance_matrix[i][j] = math.sqrt((x2 - x1)**2 + (y2 - y1)**2)
    return distance_matrix

distance_matrix = calculate_distance_matrix(data)
```

Calculate f1 and f2: Calculation of total distance: Accumulate the distances between adjacent customers in the input route to obtain the total travel distance for that route. Calculation of total sales profit: If the current customer is not the first customer, subtract the travel time (the distance between the previous customer (route [i-1]) and the current customer (route [i])) from the current customer's profit, and add up to obtain the final total profit.

```
# 总距离
def total_travel_distance(route, distance_matrix):
    total_distance = 0
    for i in range(len(route) - 1):
        total_distance += distance_matrix[route[i]][route[i + 1]]
    total_distance += distance_matrix[route[-1]][route[0]]
    return total_distance

# 总利润
def total_sales_profit(route, data, distance_matrix):
    total_profit = 0
    for i in range(len(route)):
        customer_profit = data.loc[route[i], 'PROFIT']
        if i > 0:
            travel_time = distance_matrix[route[i - 1]][route[i]]
            customer_profit -= travel_time
        total_profit += customer_profit
    return total_profit
```

Weighted Objective Function: This function implements optimization based on weighted objective functions. According to the formula, return the value of f1- lambda value * f2, which is the total travel distance minus the weight multiplied by the total sales profit.

```
# 加权
def weighted_objective_function(route, data, distance_matrix, lambda_value):
    f1 = total_travel_distance(route, distance_matrix)
    f2 = total_sales_profit(route, data, distance_matrix)
    return f1 - lambda_value * f2
```

Find_optimal_solution function: Initialize the optimal route best_route to None and the optimal value best_ralue to infinity to ensure that any found solution can update the optimal solution in the subsequent search process. Because I am tired of using the GA algorithm again, multiple random searches are used here to find a better solution. Set the iteration count to 1000 and find the optimal solution in 1000 searches.

```
# 寻找最优解
def find_optimal_solution(data, distance_matrix, lambda_value):
    best_route = None
    best_value = np.inf
    for _ in range(1000):
        route = np.random.permutation(len(data))
        value = weighted_objective_function(route, data, distance_matrix, lambda_value)
        if value < best_value:
            best_value = value
            best_route = route
    return best_route
```

2. Method 2

Keep the basic data import and distance matrix consistent with the previous method, and choose a method based on Pareto dominance selection to complete the task.

pareto_dominance_method function: This code implements a multi-objective optimization method based on Pareto advantage selection to solve the traveling salesman problem. By continuously randomly generating routes and comparing Pareto advantages, a set of non dominated solutions is found. Route=np. random. permutation (len (data)): Randomly generate a solution, which is a route for a traveling salesman. Calculate the total travel distance and total sales profit of the randomly generated route separately. Initialize a flag variable is_rominated to indicate whether the current solution is dominated by other solutions. Traverse each solution in the existing solution set in the following loop. Calculate the total travel distance and total sales profit of the existing solution for comparison with the current solution. Determine whether the current solution is dominated by an existing solution based on the Pareto dominance criterion of if (f1_other<=f1 and f2_other>f2) or (f1_other<f1 and f2_other>=f2). The final return contains a list of all non dominated solutions.

```
def pareto_dominance_method(data, distance_matrix, num_iterations):
    solutions = []
    for _ in range(num_iterations):
        # 随机生成一个解
        route = np.random.permutation(len(data))
        f1 = total_travel_distance(route, distance_matrix)
        f2 = total_sales_profit(route, data, distance_matrix)
        is_dominated = False
        for other_solution in solutions:
            f1_other = total_travel_distance(other_solution, distance_matrix)
            f2_other = total_sales_profit(other_solution, data, distance_matrix)
            if (f1_other <= f1 and f2_other > f2) or (f1_other < f1 and f2_other >= f2):
                is_dominated = True
                break
        if not is_dominated:
            solutions.append(route)
    return solutions
```

E. Task 5

Overall framework: In this task, in addition to the requirements of minimizing the total travel distance and maximizing the total sales profit in task 4, the concept of time window has been added. If a travel agent visits customers outside the time window, the violation value is the absolute difference between the actual visit time and the time window boundary. Find a suitable non dominated solution under the above three constraints.

Introduced a **Deap package** to assist in completing the code. The Deap package provides a complete set of tools and frameworks that enable more efficient implementation of evolutionary algorithms in solving this multi-objective traveling salesman problem, and can effectively handle multi-objective optimization and various complex operational logics.

The creator module: This module allows for the definition of fitness types and individual types. FitnessMulti is a multi-objective fitness type with weights set to (-1.0, 1.0, -1.0). Negative weights indicate to minimize, while positive weights indicate to maximize. The order of these weights is the return value of evaluate (individual), which is totaled, total_profit, total_violation.

```
creator.create("FitnessMulti", base.Fitness, weights=(-1.0, 1.0, -1.0))
creator.create("Individual", list, fitness=creator.FitnessMulti)
```

The toolbox object: The toolbox object of deal conveniently initializes the population by registering indices, individuals, and population operations.

The indices operation randomly samples a set of indices that represent the order in which a customer accesses.

Individual operation, which uses initIteration to create an individual of type Individual based on the index generated by individuals, and this individual has the fitness of the FitnessMulti type defined earlier.

The population operation uses initRepeat to repeatedly execute the toolbox.individual operation to create 100 individuals, forming a population.

```
toolbox = base.Toolbox()
toolbox.register("indices", random.sample, range(len(customers)), len(customers))
toolbox.register("individual", tools.initIterate, creator.Individual, toolbox.indices)
toolbox.register("population", tools.initRepeat, list, toolbox.individual, 100)
```

Evaluate function: This function is used to evaluate the fitness of an individual, calculate the fitness of the route according to three evaluation criteria, and finally return totals, total_profit, total_violation.

```
def evaluate(individual):
    total_distance = 0
    total_profit = 0
    total_violation = 0
    for i in range(len(individual)):
        if i == 0:
            total_distance += distance_matrix[0][individual[i]]
        else:
            total_distance += distance_matrix[individual[i - 1]][individual[i]]
        arrival_time = total_distance
        if arrival_time < ready_times[individual[i]]:
            total_violation += ready_times[individual[i]] - arrival_time
        elif arrival_time > due_times[individual[i]]:
            total_violation += arrival_time - due_times[individual[i]]
        if i == 0:
            total_profit += profits[individual[i]] - distance_matrix[0][individual[i]]
        else:
            total_profit += profits[individual[i]] - distance_matrix[individual[i - 1]][individual[i]]

    return total_distance, total_profit, total_violation
```

Toolbox.register method: Crossover, mutation, and selection operations were registered using the toolbox.register method. The selection operation selNSGA2 here is based on the principle of non dominated sorting genetic algorithm II (NSGA-II), which selects excellent individuals from the population to enter the next generation. It comprehensively considers three optimization objectives to select individuals, and selects which individuals can enter the next generation population based on their multi-objective fitness values.

```
toolbox.register("evaluate", evaluate)
toolbox.register("mate", tools.cxPartialyMatched)
toolbox.register("mutate", tools.mutShuffleIndexes, indpb=0.05)
toolbox.register("select", tools.selNSGA2)
```

III. Experimental results

A. Task 1

Overall framework:

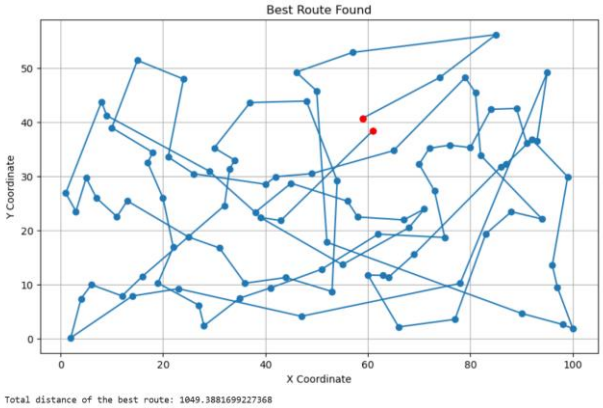
By conducting sensitive studies on different parameters such as population size, crossover probability, and mutation rate, we can explore their impact on the performance of genetic algorithms.

Setting comparison parameters includes:

Pop_Size	Crossover Rate	Mutation Rate
Small (50)	Low (0.6)	Low (0.01)
Medium (100)	Medium (0.8)	Medium (0.1)
Large (200)	High (1.0)	High (0.2)

Using Best Route Distance as an evaluation metric.

When the parameters are pop_Size=100, Mutate=0.1, Crossover=0.8, the final distance can be obtained as 1049.4, and the resulting route is shown in the figure:

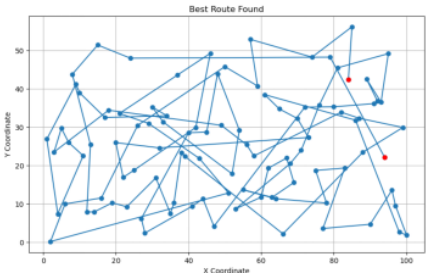


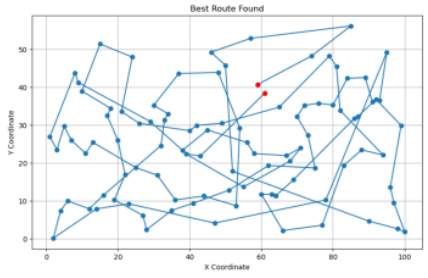
On this basis, change the parameter size to understand the impact of parameters on experimental results.

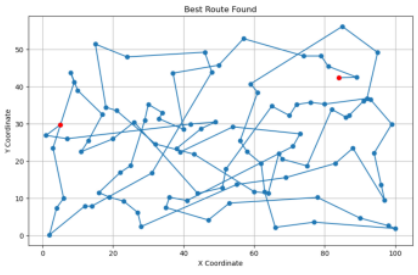
Experimental Results:

1. Change Pop Size

Pop Size	50	100	200
best_distance	1317.5	1049.4	894.4

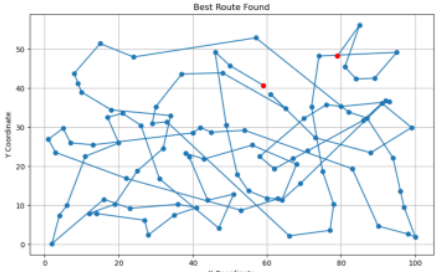


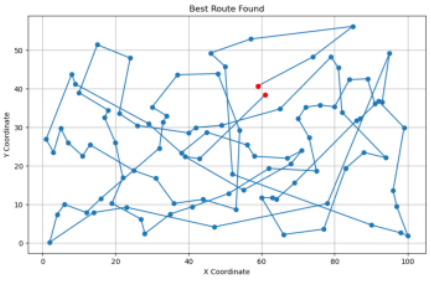


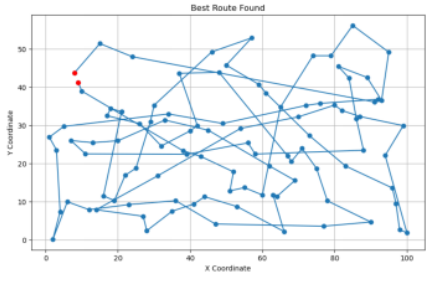


2. Change Crossover rate

Crossover rate	0.6	0.8	1.0
best_distance	1076.5	1049.4	1132.9

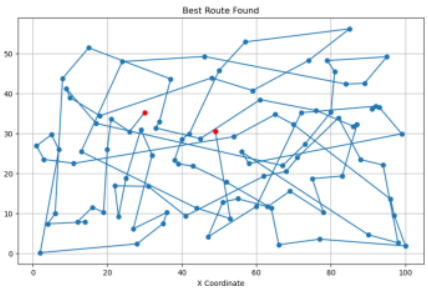


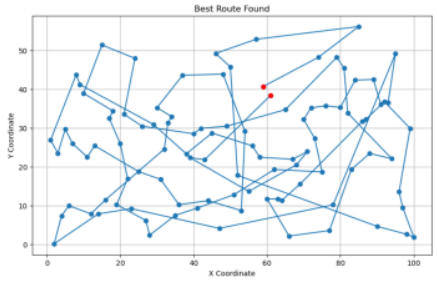


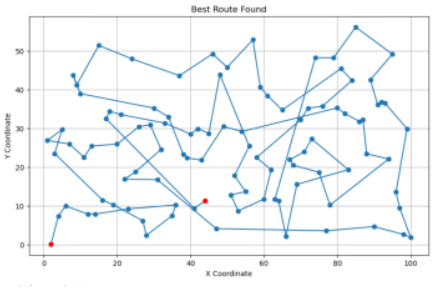


3. Change Mutate rate

Mutate rate	0.01	0.1	0.2
best_distance	1354.7	1049.4	1080.7







Experimental conclusion:

1. Increasing population size often leads to more optimized experimental results, and in this experiment, the final total distance will increase with the increase of population size.
2. A suitable crossover probability will lead to better experimental results. In this task, 0.8 is the most suitable probability. A lower probability may make it difficult to effectively utilize genetic diversity, while a higher probability may disrupt good individual structures.
3. A suitable mutation rate will produce good results. In this task, 0.1 is a relatively appropriate probability, and a lower probability will not increase diversity. A higher probability will result in the loss of good individual structures.

B. Task 2

Experimental result:

When selecting parameters with generation set to 200, mutation probability set to 0.1, crossover probability set to 0.8, and pop_size set to 100, the following results can be obtained:

```
Environment 0 without reuse: Best Distance = 443.7731492254424
Environment 1 without reuse: Best Distance = 538.2915180684603
Environment 1 with reuse: Best Distance = 511.6464596414719
Environment 2 without reuse: Best Distance = 652.126889722547
Environment 2 with reuse: Best Distance = 535.1184918030843
Environment 3 without reuse: Best Distance = 776.2699298651197
Environment 3 with reuse: Best Distance = 647.4267137921229
Environment 4 without reuse: Best Distance = 836.5015073575114
Environment 4 with reuse: Best Distance = 786.7242839622049
Environment 5 without reuse: Best Distance = 974.2675578142722
Environment 5 with reuse: Best Distance = 869.8142216960377
```

When selecting parameters with generation set to 600

```
Environment 0 without reuse: Best Distance = 366.1845160561044
Environment 1 without reuse: Best Distance = 467.8156503923555
Environment 1 with reuse: Best Distance = 426.96100141892214
Environment 2 without reuse: Best Distance = 515.6658429668007
Environment 2 with reuse: Best Distance = 480.06368556292244
Environment 3 without reuse: Best Distance = 587.1247457078897
Environment 3 with reuse: Best Distance = 535.317438644063
Environment 4 without reuse: Best Distance = 720.938567370804
Environment 4 with reuse: Best Distance = 617.4214728871272
Environment 5 without reuse: Best Distance = 858.1163983413866
Environment 5 with reuse: Best Distance = 704.620893809896
```

Experimental conclusion:

We can see some advantages of dynamic TSP: when dealing with dynamically changing environments, it can converge to a better solution faster by utilizing the optimal solution of the previous environment as the initial population or reference for the next environment. Especially when the environmental changes are not particularly drastic, this method can avoid genetic algorithms from completely searching from scratch in new environments, saving a lot of computing resources and time.

C. Task 3

Overall framework:

Conduct sensitivity analysis on population size, mutation probability, and crossover probability. The initial parameter is set to pop_size=50, generations=100，Mutate rate=0.1, Crossover rate=0.8, number of clusters=5. The following experimental results can be obtained for ballistic testing:

```
最终路程: 2229.279768968941
```

As the impact of pop_Size, mutation rate, and crossover rate on the final results has already been discussed in Task 1, the sensitivity testing of the K-Means algorithm will focus on the influence of generations and number of clusters on the results.

Setting comparison parameters includes:

Pop_Size	Generations	Number of clusters
Small (50)	Small (100)	Low (2)
Medium (100)	Medium (400)	Medium (5)
Large (200)	Large (700)	High (10)

Experimental Results:

1. Change Pop Size

Pop Size	50	100	150
best_distance	2229.3	1938.9	1910.2

2. Change generations

Generations	100	250	400
best_distance	2229.3	1971.4	1927.7

3. Change number of clusters

Number of clusters	2	5	10
best_distance	2525.3	2229.3	2399.5

Experimental conclusion:

1. As the population size increases, genetic algorithms are usually able to find better solutions when solving the traveling salesman problem, as they have a larger search space and higher genetic diversity. However, an excessively large population size can also increase computational costs, so a balance needs to be struck between search capability and computational efficiency.

2. As the number of iterations increases from 100 to 400, the algorithm has more opportunities for selection, crossover, and mutation operations, allowing it to search the solution space more thoroughly. In more iterations, the population has more time to

3. evolve and may find a better path solution.Analysis of optimal distance results generated by different numbers of clusters:

1)Less number of clusters (2 clusters)

Larger clustering scale:When the number of clusters is 2, there are relatively more data points in each cluster. This may result in a large search space for path planning within each cluster, making it difficult to find the optimal path.

For example, due to the concentration of data points, there may be situations where paths intersect and repeat, thereby increasing the total path length.

Possible unbalanced distribution:If the distribution of data is uneven between two clusters, it may make path planning more difficult in some clusters.

For example, if the data points in one cluster are very dense while the data points in another cluster are relatively sparse, the algorithm may have difficulty finding a balanced path between the two clusters.

2) Medium number of clusters (5 clusters)

When the number of clusters is 5, it may be more reasonable to divide the data into multiple smaller regions. There are relatively few data points in each region, making path planning easier.

This can reduce the possibility of path intersection and repetition, thereby reducing the total path length.

Balanced search space:Five clusters can achieve a good balance between the size and diversity of the search space. It will neither have a search space that is too large like 2 clusters, nor a search space that is too subdivided like 10 clusters.

3) More clusters (10 clusters)

Excessive segmentation:When the number of clusters is 10, the data is excessively subdivided. This may result in very few data points in each cluster, making path planning too localized.

D. Task 4

Due to the random selection of the initial population without optimization algorithm, the overall distance is relatively large.

Experimental Results:

1.Method based on weighted objective function

Set the initial iteration count to 1000 and λ to 0.5:

```
基于加权目标函数的优化结果:
最佳路线:  [45 38 71 64 90 40  3 25 59 28 22 66  0 60 95 15 21 61 68 81 23 16 80 77
 67 29 57 39 26 79 84 34 96 49 78  4  5 55 50  6 31 53 24 42 41 70 56 73
 87 43 54 99 18 17  7 33 48 74 36 52 47 58  9 88  8 94 83 30 46 35 97 37
 75 11 72 92  1  2 19 14 76 85 63 91 98 32 12 13 10 44 69 65 62 89 86 82
 51 27 93 20]
总旅行距离:  2190.6997464338483
总销售利润:  26967.781641942132
```

Change λ（lambda_value）:

lambda_value	0	0.25	0.5	0.75	1.0	1.25
total distance	2117.0	2141.7	2190.7	2199.6	2206.3	2209.5
total sales profit	27026.7	27005.5	26967.8	26941.4	26950.8	26943.3

Method based on weighted objective function:It can be seen that as λ increases, the total travel distance gradually increases, while the total sales profit increases. It can be concluded that if there is a greater preference for total sales profit, a larger value of λ can be chosen; If more emphasis is placed on the total travel distance, a smaller value of λ can be chosen. If λ=1, the total travel distance and total sales profit are roughly equally important.

2.Method based on Pareto advantage

When the iteration number is set to 1000, three non dominated solutions were obtained based on the Pareto dominance selection method. They balance the two goals of total travel distance and total sales profit to varying degrees.

```
非支配解路线:  [28 27 69 76 80 77 32 64 13 91 17 84 60 68 22 67 51 54 96 90 31 72 71 50
 38 52 35 88 23  7 44 19 73 62 45 10 21 20 87 94 24 42 79 53 74  4 99 93
  9 29 82 81 25 26  1 30 98 36 92  2 86 37 41 58 49 59 55 46 70 95 33  6
 14 78 11 89  0 39 18 12 65 61 16 40 43 83 97  5 34 15  8  3 47 56 63 85
 57 66 75 48]
总旅行距离:  2435.6475473734013
总销售利润:  26713.86503690327
非支配解路线:  [54  2 18 70 48 35 27 53 59 51 16 84 92 72 31 33 98 99 58 44 36  7 66 74
  6  3  8 25 71 28 86 68 77 63 46  9 94 17 49 21 80  0 34  1 24  5 39 65
 91 83 97 45 10 41 95 79 69 19 57 93 89 82 38 55 73 87 30  4 37 90 15 50
 62 40 12 60 42 29 61 47 67 20 32 22 85 14 96 76 88 23 75 26 43 13 56 78
 64 81 11 52]
总旅行距离:  2206.9716018720396
总销售利润:  26927.264642253074
非支配解路线:  [26 52 42  6 62 35 30 20 47 11 85 29 45 72 79  0 57 89 38 39 64 87 16 83
 19 76  2 32  9 61 12 10 31 75  5 73 25 55  8 56 99 46 84 60  7 93 92 33
 41 23 91 88 21 77 43 40 15 51 66 96 74 17 86 24 94 67 14 78 81  1 22 18
 53 63 71 90 49 48 34 28  3 70 95 54 13 68 37  4 82 50 27 65 58 80 36 44
 69 98 59 97]
总旅行距离:  2140.4092028257583
总销售利润:  27005.36877820814
```

Comparison of two methods:

1.Method based on weighted objective function

Advantage:

Simple and intuitive: By assigning weights to the objective function, the multi-objective problem can be transformed into a single objective problem, which is easy to understand and implement.

Efficient computation: No need for complex non dominated solution comparison and screening, conventional algorithms can be used to find the optimal solution, and the computational cost is low.

Adjustable weight: The weight can be flexibly adjusted according to actual needs and the preferences of decision-makers, such as increasing the weight for heavier profits.

Disadvantages:

Difficulty in determining weights: Determining appropriate weights is a challenge, as different weights can lead to different optimal solutions. It is difficult to determine which weight combination meets the requirements first.

Information loss: Only obtaining one optimal solution cannot provide all non dominated solutions, which may limit the decision-maker's understanding of the problem and ignore other valuable solutions.

2.Method based on Pareto advantage

Advantage:

Comprehensive information: able to find all non dominated solutions, allowing decision-makers to see the trade-off relationship between goals, understand various optimal solution combinations, and make rational decisions.

No need for weighting: There is no need to determine the target weight in advance like the weighting method, to avoid the problem of inappropriate weights.

Disadvantages:

Computational complexity: To determine non dominated solutions, it is necessary to compare each solution in the population pairwise. As the problem or population size increases, the computational complexity increases dramatically, resulting in high time and space complexity.

E. Task 5

Overall framework:

Due to the use of the Deap package, it is easy to use the GA algorithm to gradually make individuals in the population tend towards better solutions.

When the population size is set to 100, the crossover probability is 0.7, the mutation probability is 0.3, and the generation is 100, a set of non dominated solutions can be obtained under the three objectives of total travel distance, total sales profit, and total violation value in the time window:

```
----------
总距离:  1341.079361198338
总利润:  27787.12063880166
总违反值:  34785.59006810444
----------
总距离:  1341.605855655164
总利润:  27786.594144344836
总违反值:  34781.00725346775
----------
总距离:  1342.8056893470275
总利润:  27785.394310652973
总违反值:  34756.124857619034
----------
总距离:  1343.3321838038535
总利润:  27784.867816196147
总违反值:  34751.45952273382
```

After the previous four tasks, it can be seen that as the population size increases, the algorithm can explore a wider solution space, with a decreasing trend in total travel distance and total violation value, and an increase in total sales profit. With the increase of evolutionary generations, algorithms have more time to search, and the results gradually move towards better directions.

IV. Conclusion

In Task 1: Traveling Salesman Problem code task and its sensitive research, the following important knowledge points and skills can be learned:

1. Understand the basic principles of genetic algorithms

Selection, crossover, and mutation: Learn how to simulate the process of natural selection by selecting the optimal individual, crossover to generate new individuals, and random mutation, in order to optimize solutions.

Fitness assessment: Understand how to use fitness functions (such as total distance) to evaluate individual strengths and weaknesses, and guide algorithms towards better solutions.

2. Parameter sensitivity analysis

Parameter impact: By changing parameters such as population size, crossover probability, and mutation rate, observe how these changes affect the performance of the algorithm

Optimization strategy: Learn how to determine the optimal parameter configuration through experiments to improve the efficiency and effectiveness of the algorithm.

In Task 2: Learned how to design algorithms to adapt to dynamically changing environments. In this task, by adjusting the number and coordinates of cities based on the environmental variable e, and using the optimal solution of the previous environment as the initial population for the next environment, a way of dealing with dynamic optimization problems was mastered.:

In Task 3: In this task, data preprocessing was performed, such as adding 100 to the customer's X coordinate and adding new customer data, as well as using K-Means for clustering. I have gained a certain understanding of large-scale optimization problems and am familiar with the application of k-means clustering technology.

In addition, it can be understood that in daily life, for practical problems such as logistics and distribution, this experiment provides a method to solve path planning problems. The combination of genetic algorithm and K-Means clustering can effectively optimize delivery routes, reduce costs, and improve efficiency.

In Task 4: In this task, understand the actual situation where the goals of total travel distance and total sales profit may conflict with each other in the traveling salesman problem. I have learned two approaches to solving multi-objective optimization problems, including transforming multi-objective optimization problems into single objective optimization problems (weighted objective function method) and generating non dominated solution sets (Pareto dominance selection method).

In Task 5: In this task, I mainly came into contact with and became familiar with the usage of DEAP library, including creating fitness types and individual types, registering genetic operations and selection strategies, executing evolutionary algorithms, etc. Understanding the selection strategy of using Non Dominated Sorting Genetic Algorithm II (NSGA-II) to find the Pareto optimal solution set provides a more convenient solution for practical problems.