In [45]:
```
from sko.GA import GA
import numpy as np
import random
import math
```

T1 背包问题

In [46]:
```
weights=[920,1021,1065,1038,1041,1089,1016,1081,920,1035,977,1039,976,979,926,1085,9
values = [189,149,177,158,140,192,155,165,160,102,134,100,174,188,102,166,135,101]
max_weight = 10000  # 背包的容量
```

算子定义

In [47]:
```
population_size = 50
generations = 1000
mutation_rate = 0.1

def generate_individual():
    return [random.randint(0, 1) for _ in range(len(weights))]

def fitness(individual):
    total_weight = sum(w * i for w, i in zip(weights, individual))
    total_value = sum(v * i for v, i in zip(values, individual))

    # 惩罚超过背包容量的个体
    if total_weight > max_weight:
        return 0
    else:
        return total_value

def crossover(parent1, parent2):
    # 一点交叉
    crossover_point = random.randint(1, len(parent1) - 1)
    child1 = parent1[:crossover_point] + parent2[crossover_point:]
    child2 = parent2[:crossover_point] + parent1[crossover_point:]
    return child1, child2

def mutate(individual):
    # 随机变异一个基因
    mutation_point = random.randint(0, len(individual) - 1)
    individual[mutation_point] = 1 - individual[mutation_point]

def genetic_algorithm():
    # 初始化种群
    population = [generate_individual() for _ in range(population_size)]

    for generation in range(generations):
        # 计算适应度
        fitness_scores = [fitness(ind) for ind in population]

        # 选择父母
        parents = random.choices(population, weights=fitness_scores, k=2)

        # 交叉生成子代
        offspring1, offspring2 = crossover(parents[0], parents[1])

        # 变异子代
        if random.random() < mutation_rate:
            mutate(offspring1)
        if random.random() < mutation_rate:
```

```
            mutate(offspring2)

        # 替换最差的两个个体
        min_fitness_index = fitness_scores.index(min(fitness_scores))
        population[min_fitness_index] = offspring1

        second_min_fitness_index = fitness_scores.index(sorted(fitness_scores)[1])
        population[second_min_fitness_index] = offspring2

        # 输出每代的最佳适应度
        best_fitness = max(fitness_scores)
        if generation>989:
            print(f"Generation {generation + 1}, Best Fitness: {best_fitness}")

    # 找到最优解
    best_individual = population[fitness_scores.index(best_fitness)]
    print("Best Solution:", best_individual)
```

In [48]:
```
genetic_algorithm()
```

```
Generation 991, Best Fitness: 1618
Generation 992, Best Fitness: 1618
Generation 993, Best Fitness: 1618
Generation 994, Best Fitness: 1618
Generation 995, Best Fitness: 1618
Generation 996, Best Fitness: 1618
Generation 997, Best Fitness: 1618
Generation 998, Best Fitness: 1618
Generation 999, Best Fitness: 1618
Generation 1000, Best Fitness: 1618
Best Solution: [1, 0, 0, 1, 0, 1, 1, 1, 1, 0, 0, 0, 1, 1, 1, 0, 1, 0]
```

In [49]:
```
# 模拟退火算法参数
initial_temperature = 1000
cooling_rate = 0.95
iterations_per_temperature = 50

def generate_solution():
    return [random.randint(0, 1) for _ in range(len(weights))]

def fitness(solution):
    total_weight = sum(w * i for w, i in zip(weights, solution))
    total_value = sum(v * i for v, i in zip(values, solution))

    # 惩罚超过背包容量的解
    if total_weight > max_weight:
        return 0
    else:
        return total_value

def neighbor(solution):
    # 随机翻转一个基因
    neighbor_solution = solution.copy()
    flip_index = random.randint(0, len(neighbor_solution) - 1)
    neighbor_solution[flip_index] = 1 - neighbor_solution[flip_index]
    return neighbor_solution

def acceptance_probability(current_fitness, new_fitness, temperature):
    if new_fitness > current_fitness:
        return 1.0
    else:
        return math.exp((new_fitness - current_fitness) / temperature)

def simulated_annealing():
```

```
    current_solution = generate_solution()
    current_fitness = fitness(current_solution)
    best_solution = current_solution
    best_fitness = current_fitness

    temperature = initial_temperature

    while temperature > 0.1:
        for _ in range(iterations_per_temperature):
            new_solution = neighbor(current_solution)
            new_fitness = fitness(new_solution)

            if acceptance_probability(current_fitness, new_fitness, temperature) > r
                current_solution = new_solution
                current_fitness = new_fitness

                if current_fitness > best_fitness:
                    best_solution = current_solution
                    best_fitness = current_fitness

        temperature *= cooling_rate

    return best_solution, best_fitness


best_solution, best_fitness = simulated_annealing()
print("Best Solution:", best_solution)
print("Best Fitness:", best_fitness)
```

```
Best Solution: [1, 1, 1, 0, 0, 1, 1, 1, 1, 0, 0, 0, 1, 1, 0, 0, 0, 1]
Best Fitness: 1650
```

可能存在局部最优解，未完全冷却；同时种群质量较差，随机性较强，存在不能收敛到全局最优的情况。

T5

模拟退火算法

In [50]:

```
import random
import math
import numpy as np

def generate_matrix():
    matrix = np.zeros((20, 20), dtype=int)
    ones_indices = random.sample(range(20), 8)
    matrix[ones_indices, :] = 1
    return matrix

def evaluate_solution(matrix):
    return np.linalg.det(matrix)

def neighbor(matrix):
    new_matrix = matrix.copy()
    row_to_change = random.randint(0, 19)
    new_matrix[row_to_change, :] = 1 - new_matrix[row_to_change, :]
    return new_matrix

def acceptance_probability(current_value, new_value, temperature):
    if new_value > current_value:
        return 1.0
    else:
```

```
        return math.exp((new_value - current_value) / temperature)

def simulated_annealing():
    current_matrix = generate_matrix()
    current_value = evaluate_solution(current_matrix)
    best_matrix = current_matrix
    best_value = current_value

    temperature = 2.0

    while temperature > 0.1:
        new_matrix = neighbor(current_matrix)
        new_value = evaluate_solution(new_matrix)

        if acceptance_probability(current_value, new_value, temperature) > random.ra
            current_matrix = new_matrix
            current_value = new_value

            if current_value > best_value:
                best_matrix = current_matrix
                best_value = current_value

        temperature *= 0.95

    return best_matrix, best_value

# 模拟退火求解
best_matrix_sa, best_value_sa = simulated_annealing()
print("Simulated Annealing Best Matrix:")
print(best_matrix_sa)
print("Simulated Annealing Best Determinant:", best_value_sa)
```

```
Simulated Annealing Best Matrix:
[[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
 [1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1]
 [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
 [1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1]
 [1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1]
 [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
 [1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1]
 [1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1]
 [1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1]
 [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
 [1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1]
 [1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1]
 [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]]
Simulated Annealing Best Determinant: 0.0
```

In [51]:

```
import random
import numpy as np

def generate_individual():
    individual = np.zeros((20, 20), dtype=int)
    ones_indices = random.sample(range(20), 8)
    individual[ones_indices, :] = 1
    return individual
```

```python
def fitness(individual):
    det_value = abs(np.linalg.det(individual))
    return det_value + 1e-6  # 加上一个小常数以避免权重总和为零

def crossover(parent1, parent2):
    crossover_point = random.randint(1, 19)
    child1 = np.vstack((parent1[:crossover_point, :], parent2[crossover_point:, :]))
    child2 = np.vstack((parent2[:crossover_point, :], parent1[crossover_point:, :]))
    return child1, child2

def mutate(individual):
    mutation_row = random.randint(0, 19)
    individual[mutation_row, :] = 1 - individual[mutation_row, :]
    return individual

def genetic_algorithm():
    population_size = 50
    generations = 100
    mutation_rate = 0.1

    population = [generate_individual() for _ in range(population_size)]

    for generation in range(generations):
        fitness_scores = [fitness(ind) for ind in population]

        parents = random.choices(population, weights=fitness_scores, k=2)

        offspring1, offspring2 = crossover(parents[0], parents[1])

        if random.random() < mutation_rate:
            offspring1 = mutate(offspring1)
        if random.random() < mutation_rate:
            offspring2 = mutate(offspring2)

        min_fitness_index = fitness_scores.index(min(fitness_scores))
        population[min_fitness_index] = offspring1

        second_min_fitness_index = fitness_scores.index(sorted(fitness_scores)[1])
        population[second_min_fitness_index] = offspring2

    best_individual = max(population, key=lambda x: fitness(x))
    best_value_ga = fitness(best_individual)

    return best_individual, best_value_ga

# 遗传算法求解
best_matrix_ga, best_value_ga = genetic_algorithm()
print("Genetic Algorithm Best Matrix:")
print(best_matrix_ga)
print("Genetic Algorithm Best Determinant:", best_value_ga)
```

```
Genetic Algorithm Best Matrix:
[[1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1]
 [1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1]
 [1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1]
 [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
 [1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1]
 [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
 [1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1]
 [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
 [1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1]
 [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
 [1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1]
 [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
 [1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1]
 [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
 [1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1]
 [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]]
Genetic Algorithm Best Determinant: 1e-06
```

当涉及到解决旅行商问题（Traveling Salesman Problem，TSP）时，遗传算法（Genetic Algorithm，GA）和蚁群算法（Ant Colony Optimization，ACO）是两种常用的优化算法。以下是对这两种算法在解决TSP中的应用的简要整理：

遗传算法（Genetic Algorithm，GA）

1. **初始化种群：** 随机生成一组初始解作为种群。
2. **适应度函数：** 定义适应度函数，衡量每个个体的质量。在TSP中，适应度函数可以是旅行路径的总长度，需要最小化这个长度。
3. **选择：** 根据适应度函数选择一些个体作为父代，通常采用轮盘赌选择方式。
4. **交叉（Crossover）：** 将父代个体的信息组合，生成新的个体。在TSP中，可以通过交叉两个父代路径来生成新的路径。
5. **变异（Mutation）：** 随机改变个体的一些信息。在TSP中，可以随机交换路径上的两个城市。
6. **生成下一代：** 通过选择、交叉和变异生成下一代个体。
7. **重复迭代：** 重复上述步骤，直到达到设定的迭代次数或满足停止条件。
8. **输出结果：** 输出最优或近似最优的旅行路径。

蚁群算法（Ant Colony Optimization，ACO）

1. **初始化信息素：** 在每条路径上放置初始的信息素。
2. **蚁群移动：** 蚂蚁根据信息素浓度和启发式信息（如距离的倒数）选择路径。蚂蚁按照一定的概率选择路径，携带物质，并更新路径上的信息素。
3. **信息素更新：** 信息素挥发，路径上的信息素逐渐减少，并根据蚂蚁的移动更新。

4. **迭代：** 重复蚂蚁的移动和信息素更新，直到达到设定的迭代次数或满足停止条件。
5. **输出结果：** 输出最优或近似最优的旅行路径。

在实际应用中，这两种算法都具有一些参数需要调整，如种群大小、交叉率、变异率等（对于GA），以及信息素挥发率、启发因子等（对于ACO）。选择合适的参数对算法的性能影响显著，通常需要通过实验和调整来获得最佳效果。

总体而言，GA和ACO都是有效的解决TSP问题的算法，具有一些优点和局限性，具体的选择取决于问题的特性和实际需求。