南開大學
Nankai University

人 工 智 能 技 术 实 验
实 验 报 告

实验名称：基于遗传算法的旅行商问题

姓　　名：刘崇轩

学　　号：2312801

专　　业：智能科学与技术

人工智能学院

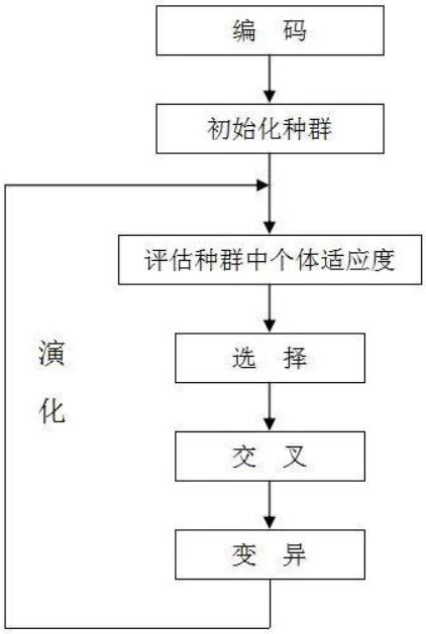
2025 年 12 月

一、 问题简述

旅行商问题：设有 N 个互相可直达的城市，给定每对城市之间的距离，某推销商准备从其中的 A 城出发，周游各城市一遍，最后又回到 A 城，要求访问每一座城市并回到起始城市的最短回路。这是非常经典的组合优化问题中的 NP 难问题（多项式复杂程度的非确定性问题），通过常规的暴力枚举，遍历等方法难以计算出最优解，因此，本实验使用遗传算法进行最优解的计算。

遗传算法是根据大自然中生物体进化规律而设计提出的，是模拟达尔文生物进化论的自然选择和遗传学机理的生物进化过程的计算模型，是一种通过模拟自然进化过程搜索最优解的方法。该算法通过数学的方式，利用计算机仿真运算，将问题的求解过程转换成类似生物进化中的染色体基因的交叉、变异等过程。在求解较为复杂的组合优化问题时，相对一些常规的优化算法，通常能够较快地获得较好的优化结果。

对于本实验，求解我国 34 个城市的旅行商问题，给定 34 个城市的名称和坐标数据，求遍历一次所有城市回到起点期间，所经过路程的较小值。



算法实现流程：

1. 初始化阶段：

   初始化城市数据：城市数目、城市位置、城市距离矩阵；

   初始化基本参数：种群数目、交叉概率、变异概率、交叉方式、变异方式；

   初始化种群：随机生成 n 个城市序列，n 为种群数目。
2. 计算种群适应度：根据城市序列和城市距离矩阵计算种群适应度。
3. 生成下一代种群（选择、交叉、变异）

选择：轮盘赌选择、锦标赛选择等方式；

交叉：单点交叉、两点交叉等方式；

变异：均匀变异、基本位变异等方式。

4. 迭代操作：重复步骤 2 和 3，并记录每一次迭代的最优解
5. 结果输出：输出迭代过程中产生的最短路径长度、最短路径；

二、 实验目的

1. 了解旅行商问题的基本概念，理解旅行商问题的求解难度；
2. 了解遗传算法的基本原理及实现流程；
3. 能够利用遗传算法解决旅行商问题；
4. 进一步理解可视化设计。（选做）

三、 实验内容

1. 实现基于穷举法的 8 个城市（No.1-No.8）的旅行商问题，求解访问每一座城市一次并最终回到起始城市的最短回路，理解穷举法的局限性；
2. 实现基于遗传算法的 34 个城市的旅行商问题，求解访问每一座城市一次并最终回到起始城市的最短回路；
3. 对比遗传算法下不同种群规模、交叉概率、变异概率等参数下，遗传算法对问题求解的效果影响并分析原因。
4. 进行遗传算法可视化，展示搜索过程中每代种群最优路径长度的变化和迭代最优结果。（选做）

四、 编译环境

1. 操作系统：Windows 11
2. Python：3.12
3. 主要第三方库：matplotlib、math、random、time
4. 在 PyCharm Community Edition 下编译

五、 实验步骤（包括重要程序代码解释）

（1）穷举法：

```
import random
from math import sqrt,factorial
from time import time
from matplotlib import pyplot as plt
plt.rcParams['font.sans-serif'] = ['SimHei']
plt.rcParams['axes.unicode_minus'] = False
cities = [
    ("北京", 116.46, 39.92),("天津", 117.2, 39.13),("上海", 121.48, 31.22),
    ("重庆", 106.54, 29.59),("拉萨", 91.11, 29.97),("乌鲁木齐", 87.68, 43.77),
```

```
    ("银川", 106.27, 38.47),("呼和浩特", 111.65, 40.82),("南宁", 108.33, 22.84),
    ("哈尔滨", 126.63, 45.75),("长春", 125.35, 43.88),("沈阳", 123.38, 41.8),
    ("石家庄", 114.48, 38.03),("太原", 112.53, 37.87),("西宁", 101.74, 36.56),
    ("济南", 117, 36.65),("郑州", 113.65, 34.76),("南京", 118.78, 32.04),
    ("合肥", 117.27, 31.86),("杭州", 120.19, 30.26),("福州", 119.3, 26.08),
    ("南昌", 115.89, 28.68),("长沙", 113, 28.21),("武汉", 114.31, 30.52),
    ("广州", 113.23, 23.16),("台北", 121.5, 25.05),("海口", 110.35, 20.02),
    ("兰州", 103.73, 36.03),("西安", 108.95, 34.27),("成都", 104.06, 30.67),
    ("贵阳", 106.71, 26.57),("昆明", 102.73, 25.04),("香港", 114.1, 22.2),
    ("澳门", 113.33, 22.13),
]
start = cities[0]
n = 8

### 路径长度（适应度）计算：路径越短，适应度越高
def path_cost(path):
    cost = 0
    for i in range(len(path)):
        n1, x1, y1 = cities[path[i]]
        if i<len(path)-1:
            n2, x2, y2 = cities[path[i+1]]
            dis = sqrt((x1 - x2) ** 2 + (y1 - y2) ** 2)
            cost += dis
        else:
            n2, x2, y2 = cities[path[0]]
            dis = sqrt((x1 - x2) ** 2 + (y1 - y2) ** 2)
            cost += dis
    return cost

### 最佳路径可视化
def best_path_visible(path, best_cost):
    x, y, names = [], [], []
    # 提取坐标
    for i in path:
        names.append(cities[i][0])
        x.append(cities[i][1])
        y.append(cities[i][2])
    # 闭环：添加起点到终点
    x.append(x[0])
    y.append(y[0])
    plt.figure(figsize=(12, 8))
    # 1. 画路径线
    plt.plot(x, y, 'o-', color='blue', alpha=0.6, linewidth=1, markersize=4)
    # 2. 画起点（标红）
```

```
    plt.plot(x[0], y[0], 'o', color='red', markersize=8, label='起点/终点')
    # 3. 标注城市名
    for i in range(len(names)):
        # 稍微偏移一点文字位置，避免挡住点
        plt.text(x[i] + 0.3, y[i] + 0.3, names[i], fontsize=9)
    plt.title(f"穷举算法最终优化路径 (总距离: {best_cost:.2f})")
    plt.xlabel("经度")
    plt.ylabel("纬度")
    plt.grid(True, linestyle='--', alpha=0.5)
    plt.legend()
    plt.show()

if __name__ == "__main__":
    best_cost, best_path = float("inf"), [None]
    start_time = time()
    for curr_path in [random.sample(range(1,n),n-1)
                      for _ in range(factorial(n))]:
        curr_path.insert(0,0)
        curr_cost = path_cost(curr_path)
        if curr_cost<best_cost:
            best_cost = curr_cost
            best_path = curr_path
    end_time = time()
    print(f"最短路径总长为：{best_cost:.2f}")
    print("最短路径为：", best_path)
    print(f"运行时间为：{end_time-start_time:.4f}秒")
    best_path_visible(best_path, best_cost)
```

（2）遗传算法：（与穷举法相同部分省略）

```
...
length = len(cities)
### 路径长度（适应度）计算：路径越短，适应度越高
def path_cost(path):
    ...

### 锦标赛选择函数
def select(sample_size):
    new_pops, new_costs = [], []
    while len(new_pops) < popsize:
        samples = random.sample(range(popsize),sample_size)
        sample_costs = [costs[i] for i in samples]
        new_costs.append(min(sample_costs))
        best_sample_idx = samples[sample_costs.index(min(sample_costs))]
        new_pops.append(pops[best_sample_idx])
    return new_costs, new_pops
```

```
### 交叉函数
def cross(parent_pops1, parent_pops2):
    child_pops = []
    for i in range(popsize):
        child = [None]*length
        parent1 = parent_pops1[i]
        parent2 = parent_pops2[i]
        if random.random() > pcross: # 交叉概率应用
            child = parent1.copy()
        else:
            # 随机生成交叉的起点和终点
            start, end = sorted(random.sample(range(length),2))
            # 复制 parent1 的中间段到 child
            child[start:end+1] = parent1[start:end+1].copy()
            # 从 parent2 中按顺序填满 child 剩余位置
            parent2_remain = [i for i in parent2 if i not in child[start:end+1]]
            count = 0
            for j in range(length):
                if child[j] is None:
                    child[j] = parent2_remain[count]
                    count += 1
        child_pops.append(child)
    return child_pops

### 变异函数
def mutate(pops):
    mutate_pops = []
    for i in range(popsize):
        if random.random() > pmutate:
            mutate_pop = pops[i].copy()
        else:
            point1, point2 = random.sample(range(length),2)
            mutate_pop = pops[i].copy()
            mutate_pop[point1], mutate_pop[point2] = mutate_pop[point2],
mutate_pop[point1]
        mutate_pops.append(mutate_pop)
    return mutate_pops
### 最佳路径可视化
def best_path_visible(path, best_cost):
    ...
### 迭代过程可视化
def plot_cost_history(cost_history):
    plt.figure(figsize=(10, 6))
    plt.plot(cost_history, color='green', linewidth=1.5)
```

```
    plt.title("遗传算法收敛过程 (最优路径长度变化)")
    plt.xlabel("迭代次数 (Generation)")
    plt.ylabel("路径长度 (Best Cost)")
    plt.grid(True, linestyle='--', alpha=0.5)
    # 在最后一点标记数值
    final_cost = cost_history[-1]
    plt.text(len(cost_history) - 1, final_cost, f"{final_cost:.2f}",
             color='red', ha='right', va='bottom', fontsize=10, fontweight='bold')
    plt.show()

generation = 1000 # 迭代次数
popsize = 100 # 种群大小
pcross = 0.9 # 交叉概率
pmutate = 0.1 # 变异概率

if __name__ == '__main__':
    start_time = time()
    pops = [random.sample(range(len(cities)),len(cities))
            for _ in range(popsize)] # 初始化种群
    ### 计算初代种群最优值
    costs = [None]*popsize
    for i in range(popsize):
        costs[i] = path_cost(pops[i])
    best_cost = min(costs)
    best_pop = pops[costs.index(best_cost)]
    print(f"初代最优路径长度: {best_cost:.2f}")
    best_cost_list = []
    best_cost_list.append(best_cost)
    ### 主循环
    for gen in range(generation):
        # 1、选择
        _, selected_pops = select(sample_size=5)
        # 2、交叉
        pops1 = selected_pops.copy()
        pops2 = selected_pops.copy()
        random.shuffle(pops2)
        child_pops = cross(pops1, pops2)
        # 3、变异
        child_pops = mutate(child_pops)
        # 4、计算子代路径距离
        child_costs = [None]*popsize
        for i in range(popsize):
            child_costs[i] = path_cost(child_pops[i])
        best_child_cost = min(child_costs)
```

```
        # 5、与父种群一一对比筛选出最好的
        for i in range(popsize):
            if child_costs[i] < costs[i]:
                costs[i] = child_costs[i]
                pops[i] = child_pops[i]
        if best_cost > best_child_cost:
            best_cost = best_child_cost
            best_pop = pops[costs.index(best_cost)]

        best_cost_list.append(best_cost)
        if gen % 50 == 0:
            print(f"第{gen}代最优路径长度: {best_cost:.2f}")

    end_time = time()
    print(f"最终最优路径长度: {best_cost:.2f}")
    print(f"最终最优路径: {best_pop}")
    print(f"运行时间为: {end_time-start_time:.4f}秒")
    plot_cost_history(best_cost_list)
    best_path_visible(best_pop, best_cost)
```

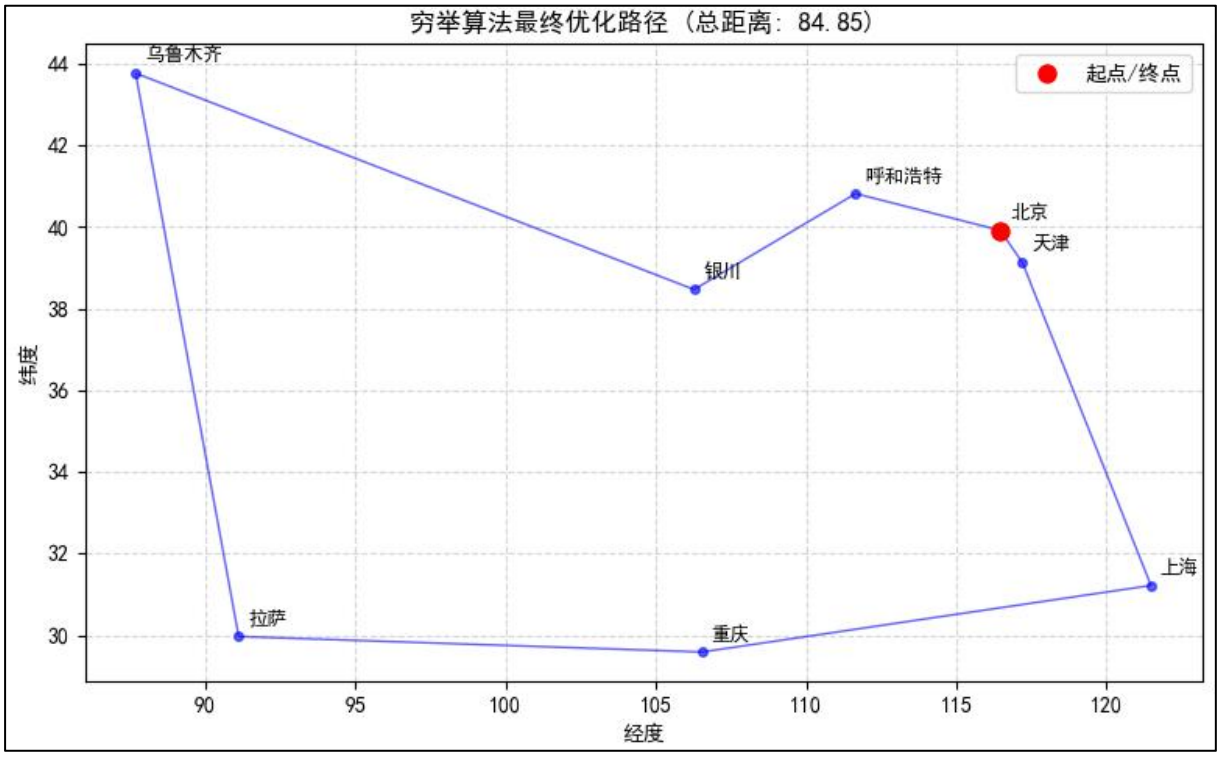
六、 实验结果

（1）穷举法：（固定起点为“北京”）

最短路径总长为：84.85

最短路径为： [0, 7, 6, 5, 4, 3, 2, 1]

运行时间为：0.2257 秒



局限性：

当 n = 9 时，运行时间增加至 2.2393 秒，约为 n = 8 时的 9 倍。

当 n = 10 时，运行时间增加至 29.8149 秒，约为 n = 9 时的 10 倍。

可知：当 n 越来越大，即问题规模越来越大的时候，穷举法的时间成阶乘级增长，十分缓慢，不适应于大规模问题求解。

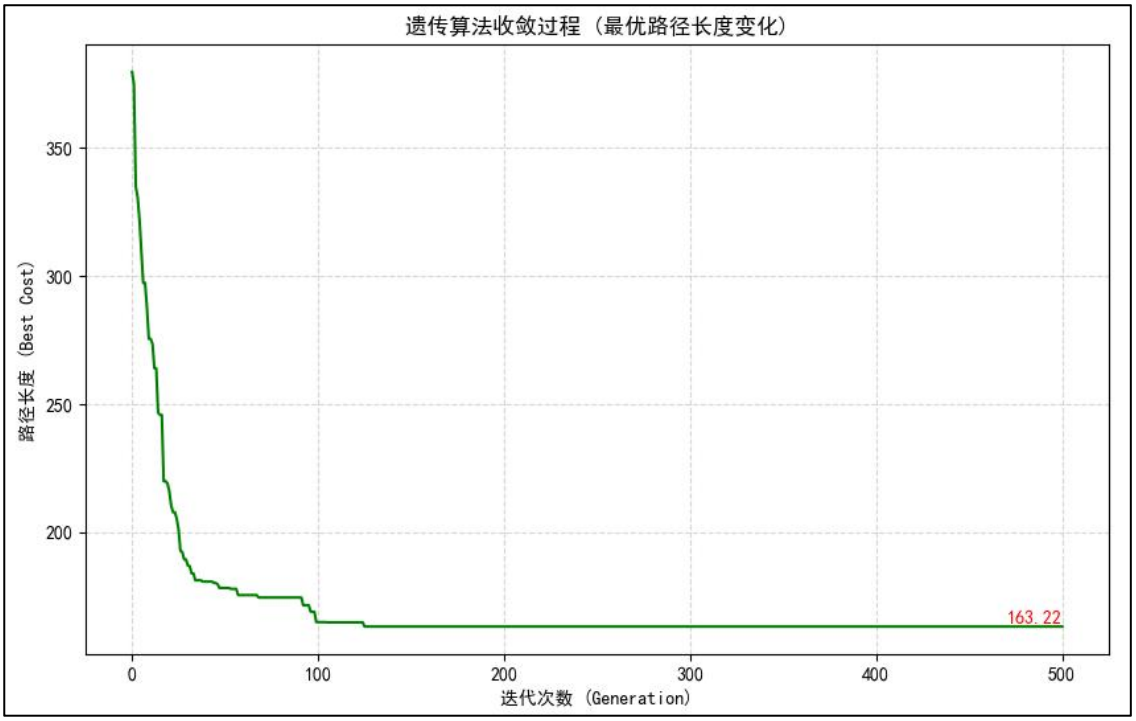
（2）遗传算法：

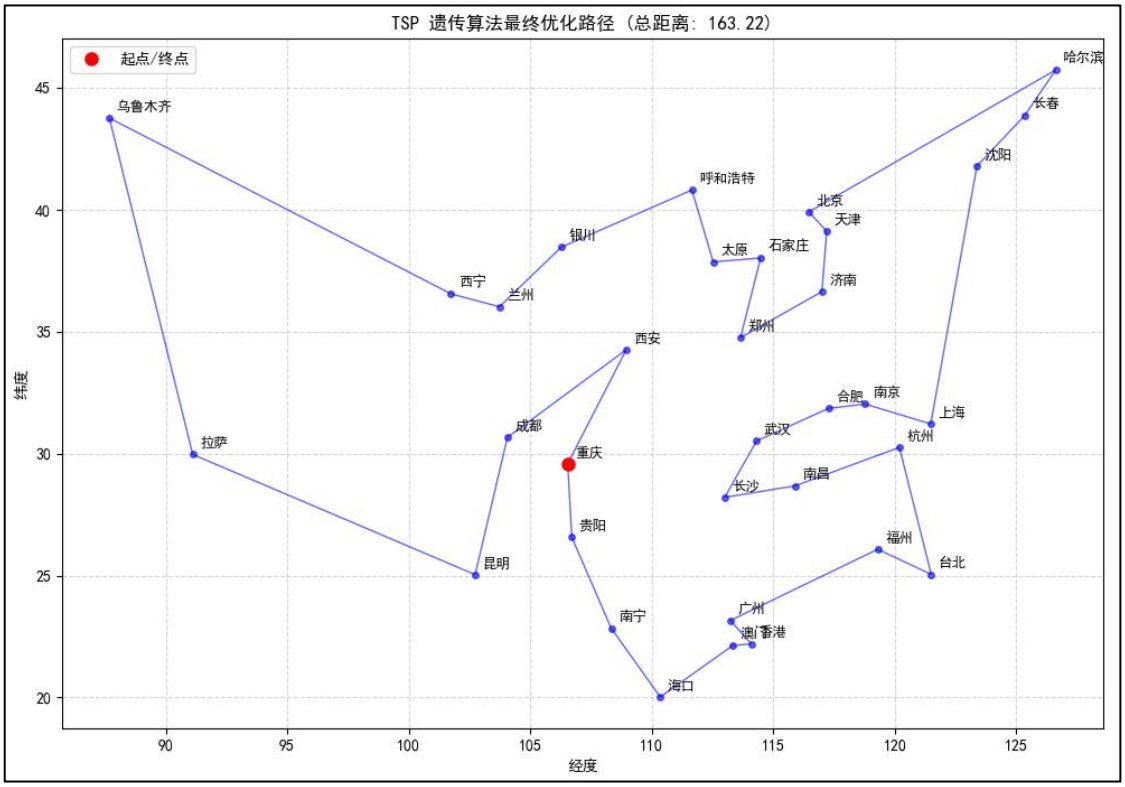
种群规模 300；迭代次数 500；交叉概率 0.9；变异概率 0.1

最终最优路径长度: 163.22

最终最优路径：[3, 28, 29, 31, 4, 5, 14, 27, 6, 7, 13, 12, 16, 15, 1, 0, 9, 10, 11, 2, 17, 18, 23, 22, 21, 19, 25, 20, 24, 32, 33, 26, 8, 30]

运行时间为：3.4475 秒





对比参数：

	第 1 组	第 2 组	第 3 组	第 4 组	第 5 组	第 6 组	第 7 组	第 8 组
种群规模	300	500	50	300	300	300	300	300
迭代次数	500	500	500	500	500	500	100	1000
交叉概率	0.9	0.9	0.9	0.1	0.9	0.9	0.9	0.9
变异概率	0.1	0.1	0.1	0.1	0.001	0.5	0.1	0.1
最小开销	163.22	174.47	180.82	205.22	207.95	170.30	176.45	171.76
运行时间/s	3.4605	5.7927	0.6313	2.3949	3.4377	3.6527	0.7337	7.0405

结果分析：（同时观察了迭代过程）

(1) 种群规模 (对比：第 3 组 vs 第 1 组 vs 第 2 组)

结论：种群规模不是越大越好。

规模太小（50），样本不够，找不到好解。

规模太大（500），计算时间翻倍，但解的质量反而不如 300（受随机性影响，且有边际递减效应）。

300 是一个性价比最高的平衡点。

(2) 交叉概率 (对比：第 4 组 vs 第 1 组)

结论：交叉概率必须高。

低交叉（0.1）导致算法几乎失效（开销>200），说明遗传算法的核心动力来自于基因重组。如果交叉太少，优秀基因无法传递。

(3) 变异概率 (对比：第 5 组 vs 第 1 组 vs 第 6 组)

结论：变异概率极其敏感，不能太高也不能太低。

极低 (0.001)：导致早熟。结果是所有组中最差的，说明算法一开始就陷入局部最优跳不出来了。

极高 (0.5)：导致乱跑。破坏了优秀的路径结构，解的质量下降。

(4) 迭代次数 (对比：第 7 组 vs 第 1 组 vs 第 8 组)

结论：单纯增加迭代次数收益有限。

500 代时已经基本收敛。

增加到 1000 代，时间翻倍（7 秒），但结果并没有比 500 代更好，说明后期算法已经停滞。

七、 分析总结

通过本次实验，得出以下核心结论：

（1）遗传算法效率远大于穷举法

对于 34 个城市的问题，穷举法因计算量呈阶乘级增长；而遗传算法仅需约 3.5 秒就能规划出一条总长约 163 的优质路径，非常适合解决此类复杂问题。

（2）参数设置的关键规律

1、交叉概率较高（0.9）

这是算法进化的核心动力，交叉太低会导致优良基因无法传递，算法直接失效。

2、变异概率适中（0.1）

太低（0.001）会导致算法“钻牛角尖”（早熟收敛）；太高（0.5）会导致算法“瞎蒙”（随机乱跑）。0.1 是维持种群活力的最佳平衡点。

3、种群和迭代适中

实验证明，种群 300、迭代 500 是性价比最高的配置。继续增加数量和代数，虽然能微调结果，但时间成本会成倍增加，收益呈现边际递减。

（3）算法的随机性特征

实验数据中出现的部分反直觉现象（如种群 500 的结果略逊于种群 300）揭示了遗传算法的本质特征：它是一种基于概率的随机启发式搜索算法。

内在随机性：初始种群生成的随机性、选择操作的概率性以及变异发生的偶然性，共同决定了单次运行结果的波动性。

结论：即使参数设置合理，算法仍有一定概率陷入深层局部最优；反之，参数稍差时也可能因“运气好”而快速收敛。因此，在实际应用中，通常建议对同一组参数进行多次独立运行并取平均值，以获得更客观的性能评估。