一、实验内容

1. **问题描述：**旅行商问题，即TSP问题（Traveling Salesman Problem）又译为旅行推销员问题、货郎担问题，是数学领域中著名问题之一。假设有一个旅行商人要拜访n个城市，他必须选择所要走的路径，路径的限制是每个城市只能拜访一次，而且最后要回到原来出发的城市。路径的选择目标是要求得的路径路程为所有路径之中的最小值。

2.**内容提要：**以N个节点的TSP（旅行商问题）问题为例，应用遗传算法并用选定的编程语言，设计简单的遗传优化系统对问题进行求解，求出问题的最优解，通过实验培养学生利用遗传算法进行问题求解的基本技能。

下面给出30个城市的位置信息:

表1　Oliver TSP问题的30个城市位置坐标

城市编号	坐标	城市编号	坐标	城市编号	坐标
1	(87，7)	11	(58，69)	21	(4，50)
2	(91，38)	12	(54，62)	22	(13，40)
3	(83，46)	13	(51，67)	23	(18，40)
4	(71，44)	14	(37，84)	24	(24，42)
5	(64，60)	15	(41，94)	25	(25，38)
6	(68，58)	16	(2，99)	26	(41，26)
7	(83，69)	17	(7，64)	27	(45，21)
8	(87，76)	18	(22，60)	28	(44，35)
9	(74，78)	19	(25，62)	29	(58，35)

10	(71，71)	20	(18，54)	30	(62，32)

也可取前10个城市的坐标进行测试:

表2　Oliver TSP问题的10个城市位置坐标

城市编号	坐标
1	(87，7)
2	(91，38)
3	(83，46)
4	(71，44)
5	(64，60)
6	(68，58)
7	(83，69)
8	(87，76)
9	(74，78)
10	(71，71)

上述30/10个城市的求解中编号从0开始，把所有路径搜索完又返回到出发节点。

二、实验设备

1. 实验设备：台式机/笔记本等不限
2. 平台：Visual C++ / Python等不限

三、实验步骤

- 1. 生成N个二维坐标节点。
- 2. 应用遗传算法并用选定的编程语言，设计简单的遗传优化系统对问题进行求解，求出问题的最优解。
- 3. 选择适当可视化方法显示结果。
- 4.分析适应度函数对启发式搜索算法的影响。
- 5.*扩展选做题：考虑不同数值N对最终结果和求解性能的影响

```
def mutation(routes,n_cities):
    #变异操作
    prob=0.3#设置变异概率
    p_rand=np.random.rand(len(routes))#生成变异序列，0到1的随机数
    for i in range(len(routes)):
        if p_rand[i]<prob:#  判断当前路线是否满足变异条件（随机数小于变异概率）
            mut_position = np.random.choice(range(n_cities), size=2, replace=False)
            l, r = mut_position[0], mut_position[1]
            routes[i,l],routes[i,r]=routes[i,r],routes[i,l]
    return routes
```

无设置交叉概率，导致每次都交叉

四、分析说明（包括结果图表分析说明，主要核心代码及解释）

实验思路：

旅行商问题是一个经典的组合优化问题，问题的描述是：给定一组城市和城市之间的距离，要求找到一条最短的路径，使得旅行商从某一个城市出发，经过每一个城市且仅经过一次，最后回到出发城市。

1. 初始化

```
def init_route(n_route, n_cities):#初始化路线

    routes = np.zeros((n_route, n_cities)).astype(int)#初始化

    for i in range(n_route):#随机生成不重复的n_route条在限定城市的数量

        routes[i] = np.random.choice(range(n_cities), size=n_cities, replace=False)

     return routes
```

2. 生成N个二维坐标节点

```
def random_cities(cities_n): #  假设城市的坐标范围在0到100之间

    range_limit = 100

    coordinates=[]



     while len(coordinates) < cities_n:

          x = random.randint(0, range_limit)

          y = random.randint(0, range_limit)

          coordinates.append((x,y))#加入坐标

    return coordinates
```

3. 计算路线适应度

```
def get_two_cities_dist(city1,city2):#返回两地欧氏距离

    x_1,y_1=city1

     x_2,y_2=city2

     return math.sqrt(math.pow(x_1-x_2,2)+math.pow(y_1-y_2),2)
```

```
def get_cities_distance(cities):#获取城市间的距离矩阵
    dist_matrix=np.zeros((len(cities),len(cities)))#初始化
    n_cities=len(cities)
     for i in range(n_cities-1):
          for j in range(i+1,n_cities):
               dist=get_two_cities_dist(cities[i],cities[j])
               dist_matrix[i,j]=dist
               dist_matrix[j,i]=dist
          return dist_matrix
def get_all_routes_fitness_value(routes,dist_matrix):#计算所有路线的适应度
    fitness_values=np.zeros(len(routes))#一维数组
    for i in range(len(routes)):
          f_value=get_route_fitness_value(routes[i],dist_matrix)
          fitness_values[i]=f_value
     return fitness_values
```

4. 生成随机路径

```
for i in range(n_route):#随机生成不重复的n_route条在限定城市的数量
    routes[i] = np.random.choice(range(n_cities), size=n_cities, replace=False)
```

5. 选择个体

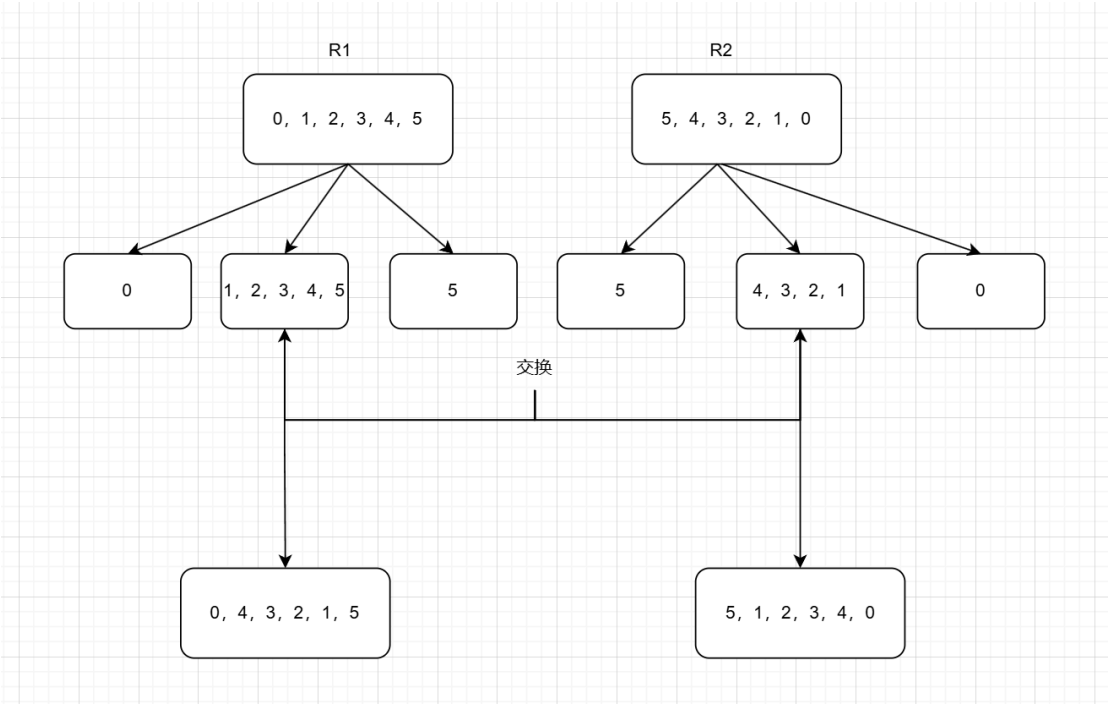
```
def selection(routes,fitness_values):#选择操作
    selected_routes=np.zeros(routes.shape).astype(int)#初始化选择路线
    probability=fitness_values/np.sum(fitness_values)#可能性，轮盘赌，按比例
    n_routes=routes.shape[0]#获取路线数量
    for i in range(n_routes):
        choice=np.random.choice(range(n_routes),p=probability)#按概率随机选取一条
路线
        selected_routes[i]=routes[choice]
     return selected_routes
```

6. 交叉操作

```
def crossover(routes,n_cities):#交叉操作
    for i in range(0,len(routes),2):#每次处理两条路线
        # 新建一维数组，用于存储交叉后的新路线
        r1_new, r2_new = np.zeros(n_cities), np.zeros(n_cities)
          # 随机选择交叉点
        seg_point = np.random.randint(0, n_cities)
          # 计算交叉段的长度
        cross_len = n_cities - seg_point
          #获取当前处理的两条路线r1，r2
          r1, r2 = routes[i], routes[i + 1]
          #提取交叉段:从交叉点开始提取r2的后半段给r1_cross，提取r1的后半段给
r2_cross。
```

```
        r1_cross, r2_cross = r2[seg_point:], r1[seg_point:]
            #提取非交叉段:使用np.in1d函数找到r1中不在r1_cross中的部分（即非交叉段），同
理找到r2中不在r2_cross中的部分。
        r1_non_cross = r1[np.in1d(r1, r1_cross, invert=True)]
            r2_non_cross = r2[np.in1d(r2, r2_cross, invert=True)]
            #组合成新路线：将交叉段放入新路线的前半段，将非交叉段放入新路线的后半段
        r1_new[:cross_len], r2_new[:cross_len] = r1_cross, r2_cross
            r1_new[cross_len:], r2_new[cross_len:] = r1_non_cross, r2_non_cross
            #更新原路线
        routes[i], routes[i + 1] = r1_new, r2_new
    return routes
```

交叉操作示意图:



交叉操作示意图

7. 变异操作

```
def mutation(routes,n_cities):
    #变异操作
    prob=0.01#设置变异概率
    p_rand=np.random.rand(len(routes))#生成变异序列，0到1的随机数
    for i in range(len(routes)):
        if p_rand[i]<prob:#  判断当前路线是否满足变异条件（随机数小于变异概率）
            mut_position = np.random.choice(range(n_cities), size=2, replace=False)
            l, r = mut_position[0], mut_position[1]
            routes[i,l],routes[i,r]=routes[i,r],routes[i,l]
    return routes
```

8. 遗传算法

```
# 8. 遗传算法
def genetic_algorithm(file_path, n_routes, max_epoch):
    cities = read_cities_from_file(file_path)
    dist_matrix = get_cities_distance(cities)
    routes = init_route(n_routes, len(cities))
    fitness_values = get_all_routes_fitness_value(routes, dist_matrix)
    best_route_index = np.argmax(fitness_values)
    best_route = routes[best_route_index]
    best_fitness = fitness_values[best_route_index]
```

```
    for epoch in range(max_epoch):
        routes = selection(routes, fitness_values)
        routes = crossover(routes, len(cities))
        routes = mutation(routes, len(cities))
        fitness_values = get_all_routes_fitness_value(routes, dist_matrix)
        current_best_index = np.argmax(fitness_values)

        if fitness_values[current_best_index] > best_fitness:
            best_route = routes[current_best_index]
            best_fitness = fitness_values[current_best_index]

        if (epoch + 1) % 10 == 0:
            print(f"迭代次数：{epoch + 1}，当前最优距离：{1 / best_fitness:.2f}")

    print(f"最优路线：{best_route}，最优距离：{1 / best_fitness:.2f}")
    plot_best_route(cities, best_route)
```

9. 选择适当可视化方法显示结果

```
def plot_best_route(cities, best_route):

    # 提取城市坐标
    x = [cities[i][0] for i in best_route]
```

```
    y = [cities[i][1] for i in best_route]


    # 输出正确的路径顺序
    print("最佳路径为:")
    for i in best_route:
        print(i, end=" ")
    print("\n")


    # 绘制城市坐标点
    plt.figure(figsize=(10, 10))
    plt.scatter(x, y, s=100, c='r', marker='o')


    # 在每个坐标点上添加编号标注
    for i, (x_, y_) in enumerate(zip(x, y)):
        plt.text(x_, y_, str(best_route[i]), ha='center', va='bottom', fontsize=10)


        # 绘制最佳路径
    plt.plot(x + [x[0]], y + [y[0]], '-b', linewidth=2)


    # 设置标题和坐标轴标签
    plt.title("最佳路径")
    plt.xlabel("X")
```

```
plt.ylabel("Y")

plt.grid(True)

plt.show()
```

10. 分析适应度函数对启发式搜索算法的影响

原始适应度函数:

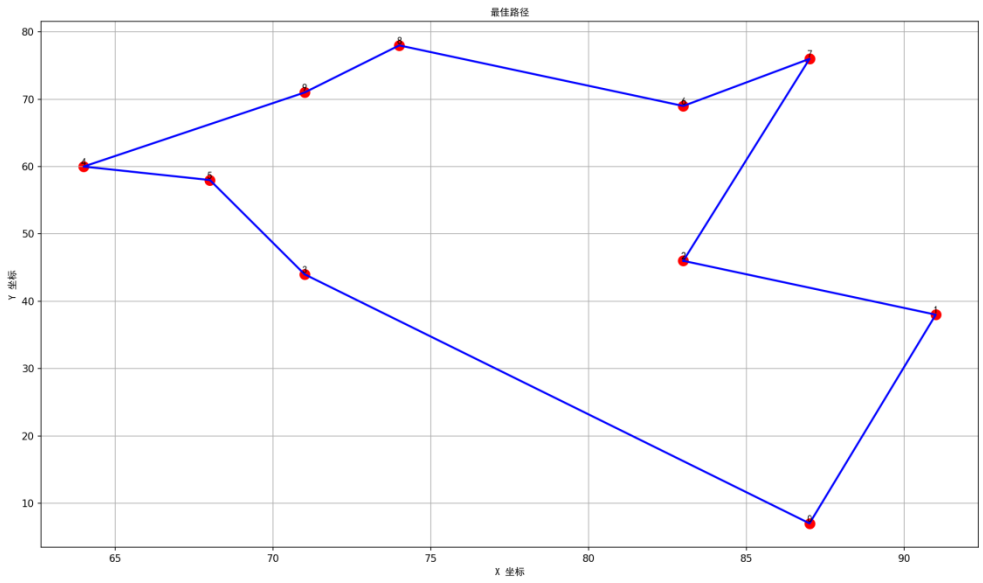
$$Fitness=1/distance$$

优点：简单高效，能引导算法优化路径。
缺点：适应度差异可能过小，影响选择压力。
改进的适应度函数:

$$Fitness=1/(distance*distance)$$

优点：放大优解和劣解的差异，增加选择压力。
缺点：可能导致过早收敛。

实验结果:




```
最优路线：[2 1 0 3 5 4 9 8 6 7]，最优距离：173.38
最佳路径为：
2 1 0 3 5 4 9 8 6 7
```

10个城市的TSP结果

五、总结心得

1.交叉操作时没有考虑到交叉之后的城市会出现重复的现象

修改前：

```
#提取交叉段:从交叉点开始提取r2的后半段给r1_cross，提取r1的后半段给r2_cross。
r1_cross, r2_cross = r2[seg_point:], r1[seg_point:]
#提取非交叉段:使用np.in1d函数找到r1中不在r1_cross中的部分（即非交叉段），同理找到r2中
不在r2_cross中的部分。
r1_non_cross = r1[np.in1d(r1, r1_cross)]
r2_non_cross = r2[np.in1d(r2, r2_cross)]
#组合成新路线：将交叉段放入新路线的前半段，将非交叉段放入新路线的后半段
r1_new[:cross_len], r2_new[:cross_len] = r1_cross, r2_cross
r1_new[cross_len:], r2_new[cross_len:] = r1_non_cross, r2_non_cross
```

修改后：

```
#提取交叉段:从交叉点开始提取r2的后半段给r1_cross，提取r1的后半段给r2_cross。
r1_cross, r2_cross = r2[seg_point:], r1[seg_point:]
#提取非交叉段:使用np.in1d函数找到r1中不在r1_cross中的部分（即非交叉段），同理找到r2中
不在r2_cross中的部分。
r1_non_cross = r1[np.in1d(r1, r1_cross, invert=True)]
r2_non_cross = r2[np.in1d(r2, r2_cross, invert=True)]
#组合成新路线：将交叉段放入新路线的前半段，将非交叉段放入新路线的后半段
r1_new[:cross_len], r2_new[:cross_len] = r1_cross, r2_cross
r1_new[cross_len:], r2_new[cross_len:] = r1_non_cross, r2_non_cross
```

2.将系统设定的路线数量和迭代次数修改成手动输出，可以更好地调试参数

修改前：

```
n_routes_ = 100  # 路线

epoch = 100000  # 迭代次数
```

修改后（手动输入）：

```
randomcities=int(input("请输入随机生成的城市数："))#生成城市

# 手动输入路线数量

n_routes_ = int(input("请输入路线数量："))

# 手动输入迭代次数

epoch = int(input("请输入迭代次数："))
```

3.计算适应度函数时没有算上最后一段返回出发点的距离，可能会导致计算的距离不准确。

修改前：

```
def get_route_fitness_value(route,dist_matrix):#计算路线适应度

    dist_sum=0#适应度设为0

     for i in range(len(route)-1):#route是一个一维的数组，包括一条路线

         dist_sum+=dist_matrix[route[i],route[i+1]]

     return 1/dist_sum
```

修改后：

```
def get_route_fitness_value(route,dist_matrix):#计算路线适应度

    dist_sum=0#适应度设为0
```

```
    for i in range(len(route)-1):#route是一个一维的数组，包括一条路线
        dist_sum+=dist_matrix[route[i],route[i+1]]
    dist_sum+=dist_matrix[route[len(route)-1],route[0]]#加入最后返回原点的距离
    return 1/dist_sum
```

4. 将特定输入改为随机输入

修改前:

```
cities_ = load_data('./cities.txt')    #  导入数据
cities_=random_cities(randomcities)    #随机生成
```

修改后:

```
randomcities=int(input("请输入随机生成的城市数："))#生成城市
cities_=random_cities(randomcities)    #随机生成
```

5. 停止迭代条件：连续2000次没有改变最优路线

修改后:

```
if not_improve_time >= 2000:#当迭代到2000次无变化结束
    print('连续2000次迭代都没有改变最优路线，结束迭代')
     break
```

6. 每隔200次迭代输出最优路线距离

修改后:

```
if (i_ + 1) % 200 == 0:#每迭代两百次就显示一次
    print('epoch: {}，当前最优路线距离：  {}'.format(i_ + 1, 1 /
get_route_fitness_value(best_route, dist_matrix_)))
```

7. 路线随机生成得到的结果不理想，结果发现是调的变异概率太低，多次迭代变异能力不强：

```
prob=0.01#设置变异概率
```

8. 总结

通过本实验，我深入了解了遗传算法解决TSP问题的基本思路和实现过程。同时也认识到，遗传算法的性能很大程度上依赖于适应度函数设计、参数设置以及操作方式的改进。未来的改进方向是引入动态调整机制、多种交叉变异方式以及高效的收敛判定条件，从而进一步提升算法性能。

不足：

1.缺乏动态调整机制（如动态变异概率）。
2.对较大规模的城市（如 50 或 100 个城市）求解性能下降明显。

附录（所有代码）

```
import numpy as np
import math
import matplotlib.pyplot as plt
from matplotlib.font_manager import FontProperties


# 设置中文字体
def get_chinese_font():
    return FontProperties(fname="C:/Windows/Fonts/simhei.ttf")  # 确保路径正确


# 1. 从文件读取城市坐标
```

```python
def read_cities_from_file(file_path):
    coordinates = []
    with open(file_path, 'r') as file:
        for line in file:
            parts = line.strip().split()
            if len(parts) == 3:  # 确保格式为 "索引 X Y"
                _, x, y = map(int, parts)
                coordinates.append((x, y))
    return coordinates



# 2. 初始化种群
def init_route(n_route, n_cities):
    routes = np.zeros((n_route, n_cities)).astype(int)  # 初始化
    for i in range(n_route):  # 随机生成不重复的n_route条在限定城市的数量
        routes[i] = np.random.choice(range(n_cities), size=n_cities, replace=False)
    return routes



# 3. 计算路线适应度相关函数
def get_two_cities_dist(city1, city2):
    x_1, y_1 = city1
```

```
    x_2, y_2 = city2
    return math.sqrt((x_1 - x_2) ** 2 + (y_1 - y_2) ** 2)



def get_cities_distance(cities):
    dist_matrix = np.zeros((len(cities), len(cities)))  # 初始化
    n_cities = len(cities)
    for i in range(n_cities - 1):
        for j in range(i + 1, n_cities):
            dist = get_two_cities_dist(cities[i], cities[j])
            dist_matrix[i, j] = dist
            dist_matrix[j, i] = dist
    return dist_matrix



def get_route_fitness_value(route, dist_matrix):
    total_distance = sum(dist_matrix[route[i], route[i + 1]] for i in range(len(route)
- 1))
    total_distance += dist_matrix[route[-1], route[0]]  # 回到起点
    return 1 / (total_distance + 1e-6)  # 避免除零


def get_all_routes_fitness_value(routes, dist_matrix):
```

```
    fitness_values = np.zeros(len(routes))  # 一维数组
    for i in range(len(routes)):
        f_value = get_route_fitness_value(routes[i], dist_matrix)
        fitness_values[i] = f_value
    return fitness_values



# 4. 选择操作
def selection(routes, fitness_values):
    selected_routes = np.zeros(routes.shape).astype(int)  # 初始化选择路线
    probability = fitness_values / np.sum(fitness_values)  # 轮盘赌按比例
    n_routes = routes.shape[0]  # 获取路线数量
    for i in range(n_routes):
        choice = np.random.choice(range(n_routes), p=probability)  # 按概率随机选取一条路线
        selected_routes[i] = routes[choice]
    return selected_routes



# 5. 交叉操作
def crossover(routes, n_cities):

    for i in range(0, len(routes), 2):  # 每次处理两条路线
```

```
        if i + 1 >= len(routes):
            break  # 如果是奇数条路线，最后一条保持不变
        # 获取当前处理的两条父路径
        r1, r2 = routes[i], routes[i + 1]
        # 随机选择两个交叉点，确保它们的顺序正确
        p1, p2 = sorted(np.random.choice(range(n_cities), size=2, replace=False))
        # 初始化子路径，先用 -1 填充，表示未赋值
        child1, child2 = np.full(n_cities, -1), np.full(n_cities, -1)
        # 交换两个父路径的交叉段
        child1[p1:p2] = r2[p1:p2]
        child2[p1:p2] = r1[p1:p2]
        # 填充子路径的非交叉段
        # 处理 child1
        for city in r1:
            if city not in child1:
                # 找到第一个空位置（-1）并填入
                idx = np.where(child1 == -1)[0][0]
                child1[idx] = city
        # 处理 child2
        for city in r2:
            if city not in child2:
                # 找到第一个空位置（-1）并填入
```

```
                idx = np.where(child2 == -1)[0][0]
                    child2[idx] = city
            # 更新种群中的这两条路线
        routes[i], routes[i + 1] = child1, child2

    return routes


# 6. 变异操作
def mutation(routes, n_cities):
    prob = 0.01  # 设置变异概率
    p_rand = np.random.rand(len(routes))  # 生成变异序列，0到1的随机数
    for i in range(len(routes)):
        if p_rand[i] < prob:  # 判断当前路线是否满足变异条件
            mut_position = np.random.choice(range(n_cities), size=2, replace=False)
            l, r = mut_position[0], mut_position[1]
            routes[i, l], routes[i, r] = routes[i, r], routes[i, l]
    return routes


# 7. 可视化最佳路径
def plot_best_route(cities, best_route):
```

```
    x = [cities[i][0] for i in best_route]
    y = [cities[i][1] for i in best_route]
    print("最佳路径为:")
    for i in best_route:
        print(i, end=" ")
    print("\n")

    plt.figure(figsize=(10, 10))
    plt.scatter(x, y, s=100, c='r', marker='o')

    # 显示城市编号
    for i, (x_, y_) in enumerate(zip(x, y)):
        plt.text(x_, y_, str(best_route[i]), ha='center', va='bottom', fontsize=10,
fontproperties=get_chinese_font())

    # 绘制路径
    plt.plot(x + [x[0]], y + [y[0]], '-b', linewidth=2)
    plt.title("最佳路径", fontproperties=get_chinese_font())
    plt.xlabel("X 坐标", fontproperties=get_chinese_font())
    plt.ylabel("Y 坐标", fontproperties=get_chinese_font())
    plt.grid(True)
    plt.show()
```

```python
# 8. 遗传算法
def genetic_algorithm(file_path, n_routes, max_epoch):
    cities = read_cities_from_file(file_path)
    dist_matrix = get_cities_distance(cities)
    routes = init_route(n_routes, len(cities))
    fitness_values = get_all_routes_fitness_value(routes, dist_matrix)
    best_route_index = np.argmax(fitness_values)
    best_route = routes[best_route_index]
    best_fitness = fitness_values[best_route_index]

    for epoch in range(max_epoch):
        routes = selection(routes, fitness_values)
        routes = crossover(routes, len(cities))
        routes = mutation(routes, len(cities))
        fitness_values = get_all_routes_fitness_value(routes, dist_matrix)
        current_best_index = np.argmax(fitness_values)

        if fitness_values[current_best_index] > best_fitness:
            best_route = routes[current_best_index]
            best_fitness = fitness_values[current_best_index]
```

```
            if (epoch + 1) % 10 == 0:
                print(f"迭代次数：{epoch + 1}，当前最优距离：{1 / best_fitness:.2f}")

        print(f"最优路线：{best_route}，最优距离：{1 / best_fitness:.2f}")
        plot_best_route(cities, best_route)



# 主程序
if __name__ == "__main__":
    file_path = "cities.txt"  # 替换为用户上传的文件路径
    n_routes = int(input("请输入种群数量："))
    max_epoch = int(input("请输入最大迭代次数："))
    genetic_algorithm(file_path, n_routes, max_epoch)
```