遗传算法求解 TSP 问题

一、实验目的：

熟悉和掌握遗传算法的原理、流程和编码策略，并利用遗传求解函数优化问题，理解求解 TSP 问题的流程并测试主要参数对结果的影响。

二、实验原理：

旅行商问题，即 TSP 问题（Traveling Salesman Problem）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，路经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。TSP 问题是一个组合优化问题。该问题可以被证明具有 NPC 计算复杂性。因此，任何能使该问题的求解得以简化的方法，都将受到高度的评价和关注。

遗传算法的基本思想正是基于模仿生物界遗传学的遗传过程。它把问题的参数用基因代表，把问题的解用染色体代表（在计算机里用二进制码表示），从而得到一个由具有不同染色体的个体组成的群体。这个群体在问题特定的环境里生存竞争，适者有最好的机会生存和产生后代。后代随机化地继承了父代的最好特征，并也在生存环境的控制支配下继续这一过程。群体的染色体都将逐渐适应环境，不断进化，最后收敛到一族最适应环境的类似个体，即得到问题最优的解。要求利用遗传算法求解 TSP 问题的最短路径。

三、实验内容：

1、参考互联网上的遗传算法核心代码，用遗传算法求解 TSP 的优化问题，分析遗传

算法求解不同规模 TSP 问题的算法性能。（尽量使用 python 语言）

2、对于同一个 TSP 问题，分析种群规模、交叉概率和变异概率对算法结果的影响。

3、增加 1 种变异策略和 1 种个体选择概率分配策略，比较求解同一 TSP 问题时不同变异策略及不同个体选择分配策略对算法结果的影响。

4、上交源代码。

四、实验报告要求：

1、画出遗传算法求解 TSP 问题的流程图。

遗传算法流程可以大致分为以下几步：

1. 初始化：生成一个初始种群。这个种群由多个个体组成，每个个体代表一种可能的解。

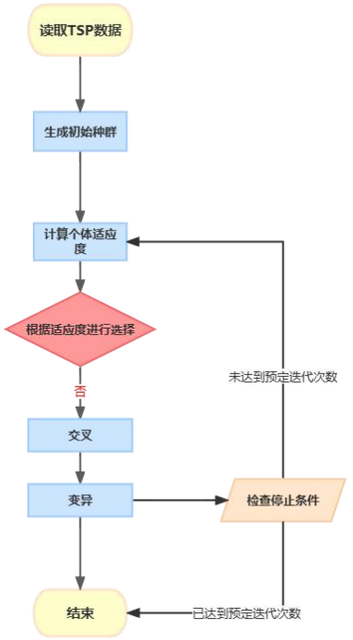
2. 评估：计算种群中每个个体的适应度。

3. 选择：根据个体的适应度，从种群中选择出用于繁殖的个体。适应度高的个体有更高的机会被选中。

4. 交叉：通过某种方式将两个被选中的个体（称为"亲代"）组合，生成新的个体（称为"子代"）。

5. 变异：以一定的概率改变新生成个体的某些部分

6. 新一代：将生成的子代形成新的种群，然后重复上述过程（评估-选择-交叉-变异），直到达到预定的停止条件（如达到最大的迭代次数，或找到了满足要求的解）。



读取文件核心代码：

```
def parse_tsp_file(filename):
    with open(filename, "r") as file:
        lines = file.readlines()[:-1]


    dimension = None
    for line in lines:
        if line.startswith('DIMENSION'):
            dimension = int(line.split(':')[1])

    coords = []
    for line in lines[-dimension:]:
        parts = line.split(' ')
        coords.append((float(parts[1]), float(parts[2])))

    return coords
```

单个路径类核心代码：

```
class Individual:
    def __init__(self, cities):
        self.cities = cities
        self.distance = self.calculate_distance()

    @classmethod
    def create_random(cls, coords):
        cities = list(range(len(coords)))
        random.shuffle(cities)
        return cls(cities)

    def calculate_distance(self):
        total = 0
        for i in range(len(self.cities)):
            total += self.distance_between(self.cities[i-1], self.cities[i])
        return total

    def distance_between(self, city1, city2):
        x_diff = coords[city1][0] - coords[city2][0]
        y_diff = coords[city1][1] - coords[city2][1]
        return math.sqrt(x_diff**2 + y_diff**2)
```

GA 遗传算法核心代码：

```
def run_genetic_algorithm(coords, population_size=200, generations=500, crossover_prob=0.1,
mutation_prob=0.1):
    def selection(population):
        fitnesses = [1 / individual.distance for individual in population]
        total_fitness = sum(fitnesses)
        probs = [fitness / total_fitness for fitness in fitnesses]
        return random.choices(population, weights=probs, k=1)[0]

    def crossover(parent1, parent2):
        if random.random() < crossover_prob:
            size = min(len(parent1.cities), len(parent2.cities))
            p1, p2 = [0] * size, [0] * size
            for i in range(size):
                p1[parent1.cities[i]] = i
```

```
            p2[parent2.cities[i]] = i
        cxpoint1 = random.randint(0, size)
        cxpoint2 = random.randint(0, size - 1)
        if cxpoint2 >= cxpoint1:
            cxpoint2 += 1
        else:
            cxpoint1, cxpoint2 = cxpoint2, cxpoint1
        for i in range(cxpoint1, cxpoint2):
            temp1 = parent1.cities[i]
            temp2 = parent2.cities[i]
            parent1.cities[i], parent1.cities[p1[temp2]] = temp2, temp1
            parent2.cities[i], parent2.cities[p2[temp1]] = temp1, temp2
            p1[temp1], p1[temp2] = p1[temp2], p1[temp1]
            p2[temp1], p2[temp2] = p2[temp2], p2[temp1]

        return Individual(parent1.cities)
    else:
        return Individual(parent1.cities[:])

def mutate(individual):
    if random.random() < mutation_prob:
        i = random.randint(0, len(individual.cities) - 1)
        j = random.randint(0, len(individual.cities) - 1)

        individual.cities[i], individual.cities[j] = individual.cities[j], individual.cities[i]

population = [Individual.create_random(coords) for _ in range(population_size)]

for _ in range(generations):
    population.sort(key=lambda individual: individual.distance)
    new_population = population[:population_size // 10]
    for _ in range(population_size - len(new_population)):
        parent1 = selection(new_population)
        parent2 = selection(new_population)
        child = crossover(parent1, parent2)
        mutate(child)
        new_population.append(child)
    population = new_population
return population[0]
```

2、 分析遗传算法求解不同规模的 TSP 问题的算法性能。

核心代码：

```
import time
for file in ['berlin52.tsp','pma343.tsp','djb2036.tsp']:
    print(f'执行{file}文件结果：')
    start_time = time.time()
    coords = parse_tsp_file(file)
    best_individual = run_genetic_algorithm(coords)
    end_time = time.time()

    print(f"最短距离: {best_individual.distance}")
    print(f"最优路径: {best_individual.cities}")
    print(f"运行时间: {end_time - start_time}秒")
```

选取了三个规模大小差异较大的数据集分别为 berlin52,pma343 以及 djb2036，分别代表各自有 52，343，2036 个城市节点。

```
执行berlin52.tsp文件结果：
最短距离: 13357.27459907984
最优路径: [43, 20, 23, 31, 4, 18, 48, 17, 51, 21, 35, 30, 19, 34, 15, 13, 49, 27, 50, 39, 5, 45, 25, 36, 14
运行时间: 4.423575401306152秒
执行pma343.tsp文件结果：
最短距离: 13699.53103955927
最优路径: [210, 230, 231, 215, 221, 185, 161, 137, 116, 103, 113, 99, 11, 6, 10, 0, 134, 112, 132, 177, 204
运行时间: 16.256539344787598秒
执行djb2036.tsp文件结果：
最短距离: 186468.0328294216
最优路径: [706, 212, 79, 1195, 162, 307, 1791, 273, 843, 646, 562, 825, 1094, 1493, 910, 1064, 1583, 1729,
运行时间: 89.86314988136292秒
```

可以看出在规模较小时算法运行效率较高，但当城市数量增大时，运行时间也会以成倍的速度增加。此外，运行时间也与种群大小和迭代次数相关。

3、对于同一个 TSP 问题，分析种群规模、交叉概率和变异概率对算法结果的影响。

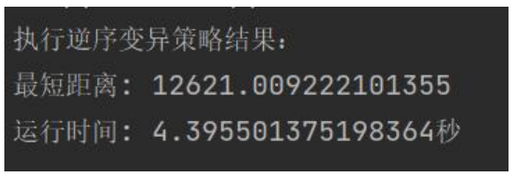
核心代码：

```
import time
file = 'berlin52.tsp'
for population_size in range(50,501,50):
    print(f'执行种群大小为{population_size}结果：')
    start_time = time.time()
    coords = parse_tsp_file(file)
    best_individual = run_genetic_algorithm(coords, population_size=population_size)
    end_time = time.time()

    print(f"最短距离: {best_individual.distance}")
    print(f"运行时间: {end_time - start_time}秒")


for crossover_prob in [0.1,0.2,0.3,0.4,0.5,0.6,0.7,0.8]:
    print(f'执行交叉概率为{crossover_prob}结果：')
    start_time = time.time()
    coords = parse_tsp_file(file)
    best_individual = run_genetic_algorithm(coords, crossover_prob=crossover_prob)
    end_time = time.time()

    print(f"最短距离: {best_individual.distance}")
    print(f"运行时间: {end_time - start_time}秒")

for mutation_prob in [0.01,0.05,0.1,0.3,0.5,0.7]:
    print(f'执行变异概率为{mutation_prob}结果：')
    start_time = time.time()
    coords = parse_tsp_file(file)
    best_individual = run_genetic_algorithm(coords, mutation_prob=mutation_prob)
    end_time = time.time()

    print(f"最短距离: {best_individual.distance}")
    print(f"运行时间: {end_time - start_time}秒")
```

上面所有实验结果的数据集均为 berlin52。首先从种群大小可以看出，越大的种群大小往往可以得到更好的优化结果，在大概 300-350 种群大小时收敛。而交叉概率则相反，越大的交叉概率会导致优秀的个体被快速拆解，导致效果不佳。变异概率与交叉概率道理相同。

4、增加 1 种变异策略和 1 种个体选择概率分配策略，比较求解同一 TSP 问题时不同变异策略及不同个体选择分配策略对算法结果的影响。

变异策略方面，除了单点变异外，我们可以改为使用更复杂一些的逆序变异策略，即随机选取某个子序列，然后将其逆序。在这个变异策略中，我们首先随机选择两个位置，然后将这两个位置之间（包括这两个位置）的城市序列逆序。这样的变异操作会保持路径的有效性（即每个城市都恰好出现一次，不存在重复城市），同时可以引入新的路径选择。



结果为种群大小为 200，交叉变异概率为 0.1 的优化结果，在 TSP 问题中，逆序变异策略几乎不会对优化结果产生影响，

逆序变异核心代码：

```
def mutate(individual):
    if random.random() < mutation_prob:
        i = random.randint(0, len(individual.cities) - 1)
        j = random.randint(0, len(individual.cities) - 1)

        individual.cities[i], individual.cities[j] = individual.cities[j], individual.cities[i]
```

而个体选择策略方面，可以从基本的轮盘赌更改为锦标赛选择策略。在锦标赛选择中，我们从种群中随机选择一定数量的个体，然后从这些个体中选取最优的一个。这样的策略同时考虑了随机性和个体的适应度。

```
执行锦标赛策略结果：
最短距离: 11711.470893016833
运行时间: 2.73315167427063秒
```

同样为种群大小为 200，交叉变异概率为 0.1 的优化结果，可以看出相较于轮盘赌，锦标赛策略在 TSP 优化问题中优势更大，可以取得更好的最短距离。

锦标赛选择核心代码：

```
def selection(population):
    tournament = random.sample(population, tournament_size)
    return min(tournament, key=lambda individual: individual.distance)
```