中山大学计算机学院

人工智能

本科生实验报告

（2022 学年春季学期）

课程名称：Artificial Intelligence

教学班级	**计科 2 班**	专业（方向）	**计算机科学与技术**
学号	**22336126**	姓名	**李漾**

一、 实验题目

遗传算法解决 TSP 问题

二、 实验内容

1. 算法原理

1. 初始化种群

- 从问题的解空间中随机生成一组个体，构成初始种群。
- 每个个体表示为染色体，染色体上的基因代表问题的一个解。

2. 评估适应度

- 对种群中的每个个体计算适应度（fitness），即问题的目标函数值。
- 适应度函数评估个体在问题空间中的性能。

3. 选择操作

- 通过选择操作，从当前种群中选择适应度较高的个体作为父代。
- 常用的选择方法包括轮盘赌选择、竞标赛选择等。

4. 交叉操作

- 选取父代个体，并进行交叉操作，产生子代。
- 交叉操作模拟了生物学中的基因重组，将父代的染色体片段进行组合。

5. 变异操作

- 对子代进行一定概率的变异操作，引入新的遗传信息。
- 变异操作增加了种群的多样性，有助于避免陷入局部最优解。

6. 替换操作

- 将子代替换掉部分父代，形成新的种群。
- 保留适应度高的个体，淘汰适应度低的个体。

7. 迭代优化

- 重复执行选择、交叉、变异和替换等操作，直到满足终止条件。
- 终止条件可以是达到最大迭代次数、找到满意的解或达到预定的收敛条件等。

8. 输出结果

- 输出最终优化得到的个体，即问题的最优解或近似解。
- 输出结果可能是最优个体的染色体表示，或者是个体对应的目标函数值。

2. 伪代码

```
class GeneticAlgorithm:
    function __init__(file_path, population_size=20, mutation_rate=0.6,
max_iterations=100):
        // 初始化遗传算法对象
        map_ = load_map(file_path)  // 加载城市坐标
        self.population_size = population_size  // 种群大小
        self.mutation_rate = mutation_rate  // 变异概率
        self.max_iterations = max_iterations  // 最大迭代次数
        self.population = initialize_population()  // 初始化种群

    function load_map(file_path):
        // 加载城市坐标文件，返回城市坐标列表
        city_coords = []
        for each line in file_path:
            fields = split line by whitespace
            city_coords.append((fields[0], fields[1], fields[2]))
        return city_coords

    function initialize_population():
        // 初始化种群，随机生成路径
        population = []
        for i in range(population_size):
            random_path = random.sample(range(1, len(map_)+1), len(map_))
            population.append(random_path)
        return population

    function distance(a, b):
```

```
    // 计算城市 a 和城市 b 之间的距离
    return sqrt((a.x - b.x) ^ 2 + (a.y - b.y) ^ 2)

function evaluate(path):
    // 计算路径的总距离，即适应度
    total_distance = 0
    for i in range(len(path) - 1):
        total_distance += distance(path[i], path[i+1])
    total_distance += distance(path[-1], path[0])  // 回到起点
    return total_distance

function reproduce_ox(p1, p2):
    // 使用 OX 交叉法进行交叉
    sta, end = sorted(random.sample(range(1, length), 2))
    child = p1[sta:end]
    c1, c2 = [], []
    for i in range(0, length):
        if p2[i] not in child:
            if i < sta:
                c1.append(p2[i])
            else:
                c2.append(p2[i])
    return c1 + child + c2

function neighbour3(path):
    // 对路径进行三种变异操作之一
    charge = random.random()
    if charge < 0.4:
        a1, a2, a3 = sorted(random.sample(range(1, length-1), 3))
        tmp = path[0:a1] + path[a2:a3] + path[a1:a2] + path[a3:length]
    elif charge < 0.6:
        i = random.randint(1, length-2)
        j = random.randint(2, length-1)
        if i != j:
            path[i], path[j] = path[j], path[i]
            tmp = path[:]
            path[i], path[j] = path[j], path[i]
        else:
            tmp = path[:]
    else:
        k1, k2 = sorted(random.sample(range(1, length-1), 2))
        tmp = path[0:k1] + path[k1:k2][::-1] + path[k2:length]
    return tmp
```

```
    function select_parent(population):
        // 从种群中选择父代
        the_weight = [1 / evaluate(individual) for individual in population]
        ch1 = random.choices(population, the_weight, k=1)
        ch2 = random.choices(population, the_weight, k=1)
        return ch1[0], ch2[0]

    function iterate():
        // 迭代过程
        best_distance = infinity
        best_individual = None
        for iteration in range(max_iterations):
            new_population = []
            for count in range(population_size // 2):
                ch1, ch2 = select_parent(population)
                child1 = reproduce_ox(ch1, ch2)
                child2 = reproduce_ox(ch2, ch1)
                if random.random() < mutation_rate:
                    child1 = neighbour3(child1)
                    child2 = neighbour3(child2)
                if child1 == child2:
                    child1 = neighbour3(child1)
                    child2 = neighbour3(child2)
                new_population.append(child1)
                new_population.append(child2)
            population = new_population[:]
            best_individual = min(population, key=evaluate)
            best_distance = min(best_distance, evaluate(best_individual))
            print(iteration, best_distance)

        return best_individual, best_distance

def main():
    tsp = GeneticAlgorithm("temp2.txt")
    best_tour, best_distance = tsp.iterate()
    print("Best tour found:", best_tour)
    print("Best distance:", best_distance)

if __name__ == "__main__":
    main()
```

3. 关键代码展示（带注释）

load_map 方法:

功能: 加载城市坐标文件。

参数:

- file_path (str): 城市坐标文件路径。

描述:

- 该方法从文件中读取城市坐标数据。文件中每一行包含一个城市的编号、横坐标和纵坐标。
- 返回的城市坐标列表中，每个元素是一个包含城市编号、横坐标和纵坐标的元组。

```
def load_map(self, file_path):
        """
        加载城市坐标文件
        城市数据我选取了放在 temp 里
        参数:
            file_path (str): 城市坐标文件路径

        返回:
            list: 城市坐标列表，每个元素为(city_id, x, y)
        """
        city_coords = []
        with open(file_path, "r") as file:
            for line in file:
                fields = line.strip().split()
                city_coords.append((int(fields[0]), float(fields[1]),
float(fields[2])))
        return city_coords
```

initialize_population 方法:

功能: 初始化种群。

描述:

- 该方法创建一个种群，种群中的每个个体都是一个城市编号列表，代表了一条可能的旅行路径。
- 每个个体都由不同的城市编号组成，且不重复。

```
def initialize_population(self):
        """
        初始化种群

        返回:
            list: 种群列表，每个个体为城市编号列表
        """
```

```
        population = []
        for _ in range(self.population_size):
            random_path = random.sample(range(1, len(self.map_)+1),
len(self.map_))
            population.append(random_path)
        return population
```

distance 方法：

功能：计算两个城市之间的距离。

参数：

- a (int)：城市 A 的编号。
- b (int)：城市 B 的编号。

描述：

- 根据给定的城市编号，从城市坐标列表中获取对应的横纵坐标，并使用欧式距离公式计算两个城市之间的距离。

```
def distance(self, a, b):
        """
        计算两个城市之间的距离

        参数:
            a (int): 城市 A 的编号
            b (int): 城市 B 的编号

        返回:
            float: 城市 A 到城市 B 的距离
        """
        ax, ay = self.map_[a-1][1], self.map_[a-1][2]
        bx, by = self.map_[b-1][1], self.map_[b-1][2]
        return math.sqrt((ax - bx) ** 2 + (ay - by) ** 2)
```

evaluate 方法：

功能：计算路径的总距离。

参数：

- path (list)：城市编号列表，表示一条路径。

描述：

- 该方法计算给定路径的总距离，包括访问所有城市后回到起点的距离。
- 通过调用 distance 方法计算相邻城市之间的距离，并将其累加得到总距离。

```
def evaluate(self, path):
```

```
        """
        计算路径的总距离

        参数:
            path (list): 城市编号列表，表示一条路径

        返回:
            float: 路径的总距离
        """
        total_distance = 0
        for i in range(len(path) - 1):
            total_distance += self.distance(path[i], path[i+1])
        total_distance += self.distance(path[-1], path[0])  # 回到起点
        return total_distance
```

reproduce_ox 方法:

功能: 使用 OX 交叉法进行交叉。

参数:

- p1 (list): 父代 1 的城市编号列表。
- p2 (list): 父代 2 的城市编号列表。

描述:

- 该方法实现了 OX（Ordered Crossover）交叉法，用于生成子代个体。
- 首先从父代个体中随机选择一个子序列，然后按照另一个父代的顺序填充子代，保留另一个父代中未在子序列中出现的城市。

```
def reproduce_ox(self, p1, p2):
        """
        使用 OX 交叉法进行交叉

        参数:
            p1 (list): 父代 1 的城市编号列表
            p2 (list): 父代 2 的城市编号列表

        返回:
            list: 子代的城市编号列表
        """
        length = len(p1)
        sta, end = sorted(random.sample(range(1, length), 2))
        child = p1[sta:end]
        c1, c2 = [], []
        for i in range(0, length):
            if p2[i] not in child:
```

```
            if i < sta:
                c1.append(p2[i])
            else:
                c2.append(p2[i])
        return c1 + child + c2
```

neighbour3 方法:

功能: 对路径进行三种变异操作之一。

参数:

- path (list): 城市编号列表，表示一条路径。

描述:

- 该方法实现了三种不同的变异操作：部分随机交换、部分逆转和部分插入。
- 根据随机概率选择其中一种变异操作，并在给定路径上执行该操作。

```
def neighbour3(self, path):
        """
        对路径进行三种变异操作之一
        1.部分随机交换：随机选择三个不同位置的城市，然后将这三个位置上的城市进行交换。
        2.部分逆转：随机选择两个不同位置的城市，然后将这两个位置之间的城市顺序逆转。
        3.部分插入：随机选择两个不同位置的城市，然后将这两个位置之间的城市插入到另一个位置之后。
        参数:
            path (list): 城市编号列表，表示一条路径

        返回:
            list: 变异后的路径
        """
        length = len(path)
        charge = random.random()
        if charge < 0.4:
            a1, a2, a3 = sorted(random.sample(range(1, length-1), 3))
            tmp = path[0:a1] + path[a2:a3] + path[a1:a2] + path[a3:length]
        elif charge < 0.6:
            i = random.randint(1, length-2)
            j = random.randint(2, length-1)
            if i != j:
                path[i], path[j] = path[j], path[i]
                tmp = path[:]
                path[i], path[j] = path[j], path[i]
            else:
                tmp = path[:]
```

```
        else:
            k1, k2 = sorted(random.sample(range(1, length-1), 2))
            tmp = path[0:k1] + path[k1:k2][::-1] + path[k2:length]
        return tmp
```

select_parent 方法:

功能: 从种群中选择父代。

参数:

- population (list): 种群列表，每个个体为城市编号列表。

描述:

- 该方法根据个体的适应度，也就是路径长度的倒数，以一定概率从种群中选择两个父代个体。

```
def select_parent(self, population):
        """
        从种群中选择父代

        参数:
            population (list): 种群列表，每个个体为城市编号列表

        返回:
            tuple: 选中的两个父代
        """
        the_weight = [1000000 / self.evaluate(individual) for individual in
population]
        ch1 = random.choices(population, the_weight, k=1)
        ch2 = random.choices(population, the_weight, k=1)
        return ch1[0], ch2[0]
```

iterate 方法:

功能: 执行遗传算法的迭代过程。

描述:

- 该方法是遗传算法的主要执行过程，其中包含生成新种群、交叉、变异、选择父代、更新种群、计算最优个体和最短距离等步骤。
- 在每一代迭代中，更新种群并计算最优个体和最短距离。同时记录每一代的最优路径，用于后续动画和最短距离的变化过程的绘制。

```
def iterate(self):
        """
        迭代函数，执行遗传算法

        返回:
```

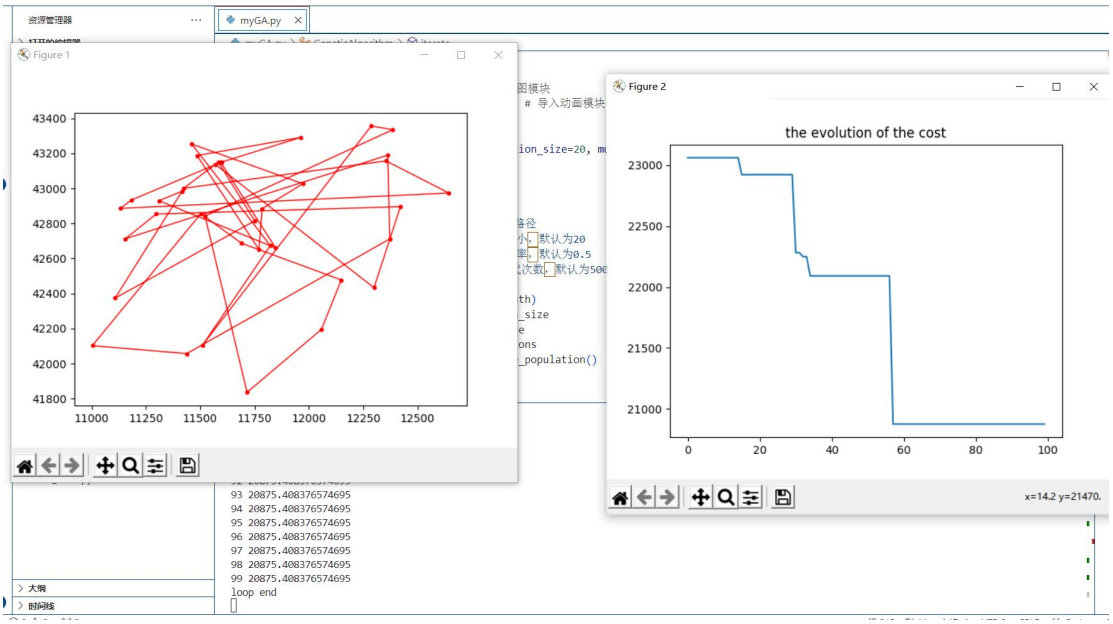
```
            tuple: 最优路径和对应的最短距离
        """
        ims = []  # 用于存储动画帧
        dis_change = []  # 存储迭代过程中的最短距离
        iteration = 0  # 迭代次数计数器
        best_distance = float('inf')  # 最短距离初始值
        best_individual = None  # 最优路径初始值

        # 开始迭代
        while iteration < self.max_iterations:
            new_population = []  # 新种群

            # 生成新种群
            for count in range(self.population_size // 2):
                ch1, ch2 = self.select_parent(self.population)
                child1 = self.reproduce_ox(ch1, ch2)
                child2 = self.reproduce_ox(ch2, ch1)

                # 变异
                if random.random() < self.mutation_rate:
                    child1 = self.neighbour3(child1)
                    child2 = self.neighbour3(child2)

                # 防止两个子代相同
                if child1 == child2:
                    child1 = self.neighbour3(child1)
                    child2 = self.neighbour3(child2)

                new_population.append(child1)
                new_population.append(child2)

            # 更新种群
            self.population = new_population[:]
            # 计算最优个体和最短距离
            best_individual = min(self.population, key=self.evaluate)
            best_distance = min(best_distance,
self.evaluate(best_individual))
            print(iteration, best_distance)
            dis_change.append(best_distance)

            # 每隔一定次数记录一次动画帧
            if iteration % 10 == 0:
                x1, y1 = zip(*[self.map_[idx-1][1:] for idx in best_individual])
                x1 += (self.map_[best_individual[0]-1][1],)
```

```
                y1 += (self.map_[best_individual[0]-1][2],)
                im = plt.plot(x1, y1, marker='.', color='red', linewidth=1)
                ims.append(im)

            iteration += 1

        print("loop end")
        # 绘制最优路径
        x1, y1 = zip(*[self.map_[idx-1][1:] for idx in best_individual])
        x1 += (self.map_[best_individual[0]-1][1],)
        y1 += (self.map_[best_individual[0]-1][2],)
        plt.plot(x1, y1, marker='.', color='red', linewidth=1)
        # 保存动画
        ani = animation.ArtistAnimation(plt.figure(1), ims, interval=200,
repeat_delay=1000)
        ani.save("GA.gif", writer='pillow')
        # 绘制最短距离的变化过程
        plt.figure(2)
        plt.title('the evolution of the cost')
        x_ = [i for i in range(len(dis_change))]
        plt.plot(x_, dis_change)
        plt.show()

        return best_individual, best_distance
```

三、 实验结果及分析

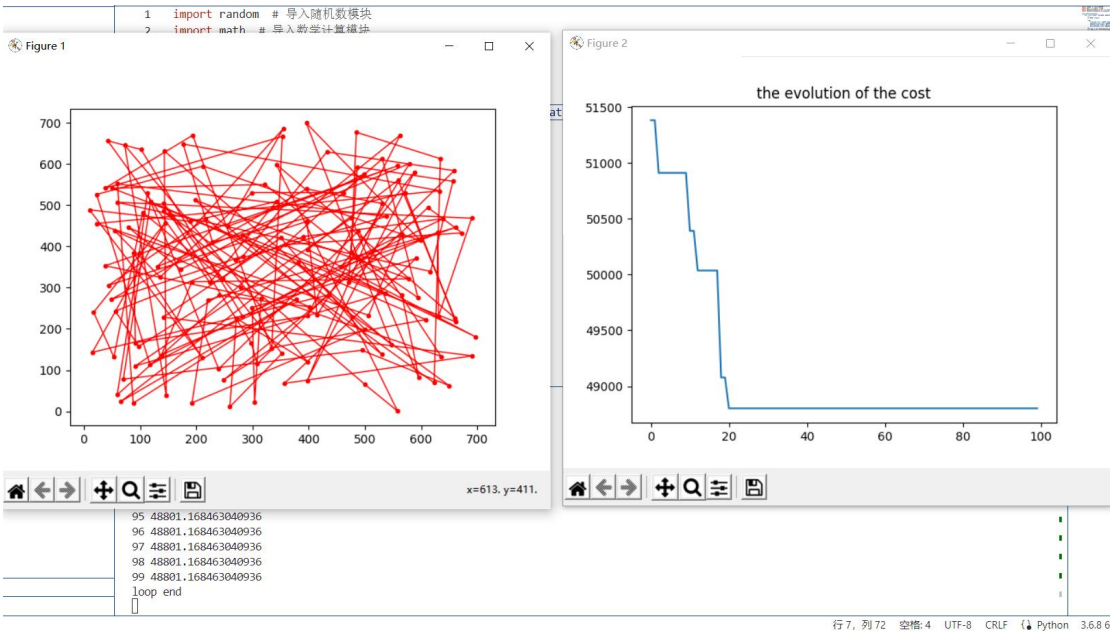
1. 实验结果展示示例（可图可表可文字，尽量可视化）

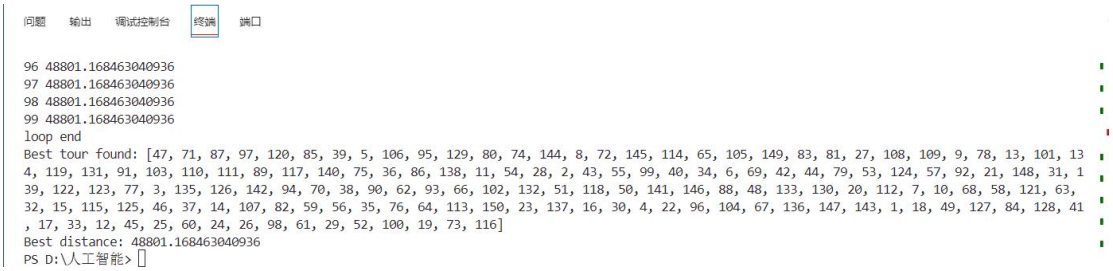
1.数据集为 dj38 采取了 100 次迭代




```
问题  输出  调试控制台  终端  端口

93 20875.408376574695
94 20875.408376574695
95 20875.408376574695
96 20875.408376574695
97 20875.408376574695
98 20875.408376574695
99 20875.408376574695
loop end
Best tour found: [19, 17, 22, 2, 8, 7, 26, 18, 5, 3, 38, 33, 9, 16, 32, 37, 6, 4, 34, 35, 10, 1, 13, 20, 30, 29, 21, 15, 36, 31, 14, 25, 11, 28, 24,
 23, 12, 27]
Best distance: 20875.408376574695
PS D:\人工智能> 
```

2. 数据集为 ch150，采取 100 次迭代





2. 评测指标展示及分析（机器学习实验必须有此项，其它可分析运行时间等）

最短路径长度：

代码通过调用 evaluate()方法计算路径的总距离，然后找到最短路径。

迭代过程中通过 best_distance 记录了最短距离，并在每次迭代中更新它。

通过打印输出和绘制收敛曲线，观察到算法是否收敛到最优解，并且得到最短路径长度。

收敛速度：

通过观察收敛曲线，我发现我的算法在处理较大数据集时收敛速度较快，而数据集较少时出现的平峰较多。且进行测试后发现迭代数若为 10000 则会出现很多次无用的迭代，迭代为 500-1000 时耗时较少且不会产生太多次无用迭代

稳定性：

代码使用了随机性，包括初始化种群、选择父代和变异操作等。

通过多次运行代码，每次运行得到的最优解虽然有差别，但是基本接近。变异概率在 0.5-0.6 时得到的较好