《人工智能导论》

实验二：遗传算法

学　号 22920212204396

姓　名 黄子安

2024 年 3 月 18 日

实验二：遗传算法

229202212204396 黄子安

一、实验目的

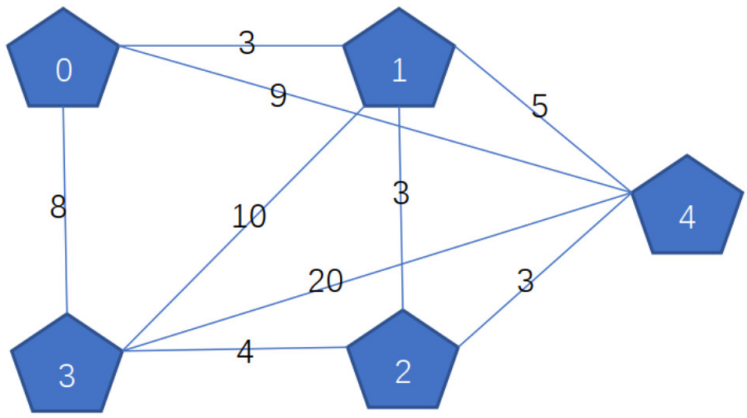
遗传算法(GeneticAlgorthm,GA）起源于对生物系统所进行的计算机模拟研究。其本质是一种高效、并行、全局搜索的方法，能在搜索过程中自动获取和积累有关搜索空间的知识，并自适应地控制搜索过程以求得最佳解。本实验通过解决旅行商问题，实现更好的熟悉和掌握遗传算法。

二、实验内容

利用遗传算法解决旅行商问题

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，每两座城市之间的距离是不同的，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**。

例如对于下图所示的无向图，对应的答案回路为[4, 1, 0, 3, 2, 4]，路径之和为23



三、实验过程

采用遗传算法解决 TSP 问题，首先定义对应的目标函数，该问题的**目标函数**即为遍历完图上所有点之后再回到原点的距离之和，当给定一条路径后直接进行累加求和即可

```
# 旅行商问题的目标函数：总路径长度
def total_distance(path, distances):
    total = 0
    for i in range(len(path) - 1):
        total += distances[path[i]][path[i + 1]]
    total += distances[path[-1]][path[0]]  # 回到起点
    return total
```

本题中个体为一条路径，选择多个个体作为**初始种群**，在该函数中使用随机打乱函数生成对应的个体，直到生成指定数量的个体数后完成种群的初始化

```
# 初始化种群
def initialize_population(num_cities, population_size):
    population = []
    for _ in range(population_size):
        individual = list(range(num_cities))
        random.shuffle(individual)
        population.append(individual)
    return population
```

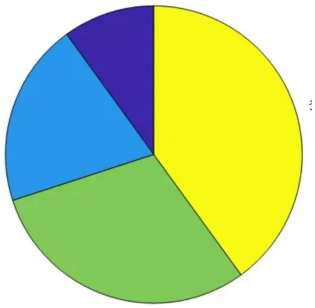
对于**个体适应度**，使用个体对应路径的长度倒数，因为本题希望使得目标函数最小，因此取一个倒数，使得每次选出的都是适应度更高的个体，更加符合对应的模型意义

```
# 适应度函数
def fitness(individual, distances):
    return 1 / total_distance(individual, distances)
```

选择函数使用轮盘赌法进行操作，先计算当前种群中所有个体对应的适应度，之后进行求和得到总的适应度，并得到每一个个体的权重比，最后根据轮盘算法选出 k 个个体

```
# 选择函数：轮盘赌法
def selection(population, distances):
    fitness_values = [fitness(individual, distances) for individual in population]
    total_fitness = sum(fitness_values)
    probabilities = [f / total_fitness for f in fitness_values]
    selected = random.choices(population, probabilities, k=len(population))
    return selected
```

轮盘赌算法根据对应的个体概率进行选择，每个个体在选择时都会被分配一个“扇区”，该扇区的大小与个体的适应度成比例，经过选择后适应度较高的个体会有更高概率被保留下来，也就是符合**自然选择**



交叉函数先随机获得生成一个位置[start,end]，对于 parent1 和 parent2 的该部分路径进行交叉，之后重新扫描两个 child，如果 child 中缺失了对应 parent 某些路径上的点，就将其加入到 child 中，最终这样生成的两个 child 都将符合路径的要求（即所有点出现且出现一次）

```python
# 交叉函数：部分映射交叉
def crossover(parent1, parent2):
    size = min(len(parent1), len(parent2))
    start = random.randint(0, size - 1)
    end = random.randint(start + 1, size)
    child1 = [None] * size
    child2 = [None] * size
    for i in range(start, end):
        child1[i] = parent2[i]
        child2[i] = parent1[i]
    for i in range(size):
        if parent1[i] not in child1:
            for j in range(size):
                if child1[j] is None:
                    child1[j] = parent1[i]
                    break
        if parent2[i] not in child2:
            for j in range(size):
                if child2[j] is None:
                    child2[j] = parent2[i]
                    break
    return child1, child2
```

变异函数只需要选出个体上的两个点，之后进行交换即可

```python
# 变异函数：随机交换
def mutate(individual):
    index1, index2 = random.sample(range(len(individual)), 2)
    individual[index1], individual[index2] = individual[index2], individual[index1]
    return individual
```

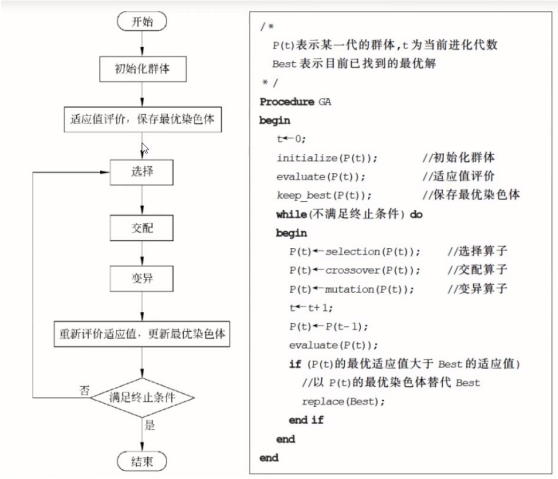
在**主函数**中先进行种群的初始化，之后进行不断迭代，这里设置迭代的默认次数为 1000 次；在每一轮迭代中先进行自然选择，按照一定概率选出适应度高的个体，淘汰掉适应度较低的个体，再依次选择两个个体进行交配生成两个后代，并对后代进行变异；迭代结束后选出具有最高适应度的个体作为最终的答案

```python
# 主函数
def genetic_algorithm(num_cities, distances, population_size=100, generations=1000):
    population = initialize_population(num_cities, population_size)
    for _ in range(generations):
        population = selection(population, distances)
        next_population = []
        for i in range(0, len(population), 2):
            parent1, parent2 = population[i], population[i + 1]
            child1, child2 = crossover(parent1, parent2)
            child1 = mutate(child1)
            child2 = mutate(child2)
            next_population.extend([child1, child2])
        population = next_population
    best_individual = max(population, key=lambda x: fitness(x, distances))
    return best_individual, total_distance(best_individual, distances)
```

最后编写对应的调用函数，给定对应图的邻接矩阵信息，从而进行算法的运行，并将最后的结果进行输出

```python
if __name__ == "__main__":
    INF = 1e7
    num_cities = 5
    distances = [[0, 3, INF, 8, 9],
                 [3, 0, 3, 10, 5],
                 [INF, 3, 0, 4, 3],
                 [8, 10, 4, 0, 20],
                 [9, 5, 3, 20, 0]]
    best_path, shortest_distance = genetic_algorithm(num_cities, distances)
    print("最佳路径:", best_path)
    print("最短距离:", shortest_distance)
```

总体流程如下：



四、实验结果

程序运行后在较短的时间内给出了题目中测试数据的结果，并且经过手工验证发现是最短的路径

```
C:\Users\26401\AppData\Local\Microsoft\WindowsApps\python3.10.exe D:\Desktop\learning\3.2\人工智能导论\lab\lab2\code.py
最佳路径: [2, 4, 1, 0, 3]
最短距离: 23

Process finished with exit code 0
```

最后选择 a280.tsp 数据集进行测试，对应的文件读取如下所示

```python
# 读取数据集
def read_tsp(file_path):
    with open(file_path, 'r') as file:
        lines = file.readlines()
        node_coord_start = None
        dimension = None
        node_coords = []

        for i, line in enumerate(lines):
            if line.startswith("DIMENSION"):
                dimension = int(line.split(":")[1])
            elif line.startswith("NODE_COORD_SECTION"):
                node_coord_start = i + 1
            elif node_coord_start and i >= node_coord_start:
                if line.strip() == "EOF":
                    break
                parts = line.strip().split()
                node_coords.append((float(parts[1]), float(parts[2])))

        return dimension, node_coords


def calculate_distance(coord1, coord2):
    x1, y1 = coord1
    x2, y2 = coord2
    return ((x1 - x2) ** 2 + (y1 - y2) ** 2) ** 0.5


def create_adjacency_matrix(node_coords):
    dimension = len(node_coords)
    adjacency_matrix = [[0] * dimension for _ in range(dimension)]

    for i in range(dimension):
        for j in range(i + 1, dimension):
            distance = calculate_distance(node_coords[i], node_coords[j])
            adjacency_matrix[i][j] = distance
            adjacency_matrix[j][i] = distance

    return adjacency_matrix


if __name__ == "__main__":

    tsp_file = "st70.tsp"
    # 读取数据集
    dimension, node_coords = read_tsp(tsp_file)
    distances = create_adjacency_matrix(node_coords)
    print(dimension)
    best_path, shortest_distance = genetic_algorithm(dimension, distances)
    print("最佳路径:", best_path)
    print("最短距离:", shortest_distance)
```

数据集内部如下所示，给出了每一个城市的 x，y 坐标，因此要进行计算欧式距离

Code Blame Executable File · 77 lines (77 loc) · 734 Bytes Code 55% faster with GitHub Copilot

```
NAME: st70
TYPE: TSP
COMMENT: 70-city problem (Smith/Thompson)
DIMENSION: 70
EDGE_WEIGHT_TYPE : EUC_2D
NODE_COORD_SECTION
1 64 96
2 80 39
3 69 23
4 72 42
5 48 67
6 58 43
7 81 34
8 79 17
9 30 23
10 42 67
11 7 76
12 29 51
13 78 92
14 64 8
15 95 57
16 57 91
17 40 35
18 68 40
19 92 34
20 62 1
21 28 43
```

运行之后得到的结果如下图所示，已知的该数据集最优解为 675，参数 population_size=1000, generations=10000，花费较长时间后得到的解不是最优，说明算法还可以有提升空间，当然这个算法只是遗传算法最基础的形式，各方面性能还都可以优化，例如改进遗传因子、使用并行计算等等

```
D:\anaconda3\envs\GUI\python.exe D:\Desktop\learning\3.2\人工智能导论\lab\lab2\code.py
70
最佳路径: [43, 51, 29, 41, 31, 17, 62, 52, 9, 15, 46, 59, 33, 2, 53, 65, 61, 27, 67, 48, 5, 14, 13, 12, 0, 34, 35, 63, 50, 42, 64, 68, 38, 16, 60, 66, 28, 39, 11, 58,
 21, 30, 69, 23, 6, 20, 26, 4, 1, 19, 47, 10, 36, 24, 25, 57, 18, 44, 55, 37, 56, 22, 7, 45, 8, 32, 49, 40, 3, 54]
最短距离: 3162.0124457230913

Process finished with exit code 0
```