《人工智能导论》

实验三：粒子群算法

学　号 22920212204396

姓　名 黄子安

2024 年 4 月 18 日

实验三：粒子群算法

229202212204396 黄子安

一、实验目的

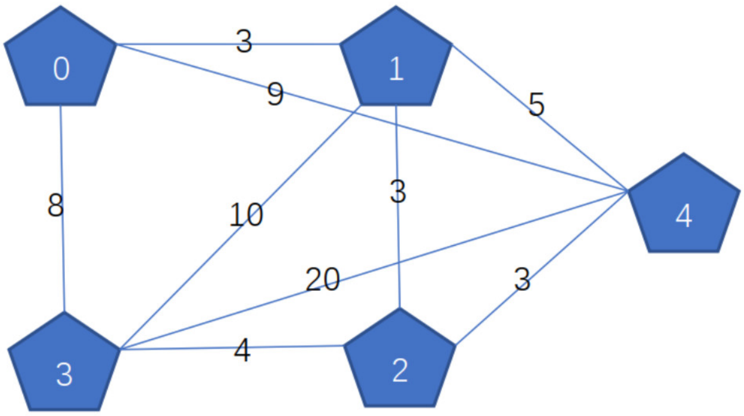
粒子群优化（PSO）算法是一种群体智能算法，由 Kennedy 和 Eberhart 于 1995 年用计算机模仿鸟群寻食这一简略的社会行动时，遭到启示，简化之后而提出的。本实验通过解决旅行商问题，更好地熟悉和掌握粒子群优化算法。

二、实验内容

利用粒子群优化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。



例如对于上图所示的无向图，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

先定义粒子类，其中含有若干个参数，num_cities 表示城市的总数，粒子的当前位置代表了一种可行的路径，因此通过一个随机排列数作为粒子的初始位置，此时粒子历史中最优的位置就是该位置，因为粒子总共也只有走过一个位置，同理该粒子的历史最优适应度先初始化为无穷大，因为算法中适应度就是对应路径序列的路径之和，所以适应度越小结果越好

粒子的速度描述了路径序列中城市的交换方式，具体在后边进行解释

```
class Particle:
    def __init__(self, num_cities):
        self.num_cities = num_cities
        self.position = np.random.permutation(num_cities)
        self.velocity = np.random.permutation(num_cities)
        self.best_position = self.position.copy()
        self.best_fitness = float('inf')
```

之后定义一个类用于封装解决 TSP 问题的 PSO 算法，其中有 c1、c2、粒子数、最大迭代次数等参数，外界需要提供对应城市的邻接矩阵，在该类中含有一个变量 global_best_position 表示所有粒子综合后得到的最优解，global_best_fitness 表示当前的全局最优适应度，同理先初始化为无穷

```
class PSO_TSP:
    def __init__(self, num_cities,  distance_matrix, num_particles = 50, max_iter = 40,
                 c1 = 2, c2 = 2, vmax = 5):
        self.num_cities = num_cities
        self.num_particles = num_particles
        self.max_iter = max_iter
        self.c1 = c1
        self.c2 = c2
        self.vmax = vmax
        self.particles = [Particle(num_cities) for _ in range(num_particles)]
        self.global_best_position = np.zeros(num_cities)
        self.global_best_fitness = float('inf')
        self.distance_matrix = distance_matrix
```

计算粒子当前适应度的函数如下所示，即计算对应粒子位置所代表的路径序列的总路径总和

```
    def evaluate_fitness(self, particle):
        fitness = 0
        for i in range(self.num_cities - 1):
            fitness += distance_matrix[particle.position[i]][particle.position[i + 1]]
        fitness += distance_matrix[particle.position[-1]][particle.position[0]]  # 回到起点
        return fitness
```

粒子的位置更新函数如下所示，首先粒子会获得新的速度，对应的公式如下所示：

$$v_i = v_i + c_1 * rand() * (pbest_i - x_i) + c_2 * rand() * (gbest_i - x_i)$$

公式由三个部分组成：

- 第一部分为**记忆项**，表示上次速度大小和方向的影响；
- 第二部分为**自身认知项**，是从当前点指向粒子历史记录中最好的点的矢量，表示粒子的动作来源于自己经验的部分；
- 第三部分为**群体认知项**，是从当前点指向种群记录中最好的点的矢量，反映了粒子间的协同合作和知识共享。

从而粒子通过自己的经验和同伴中最好的经验来决定下一步的运动

但是该公式适用于连续值，对于 TSP 问题离散值并不适用，需要进行一定的修改，这里采用《Solving City Routing Issue with Particle Swarm Optimization》论文中的方法，将速度定义为交换序列

$position1 + velocity = position2$意思是$position1$通过$velocity$这个交换序列变成$position2$，速度中每一个索引对应的数字表示为$position1$中索引为该数字的城市与$position1$中索引为速度索引的城市对调，例如：

$$position1 = [5,4,3,2,1]，position2 = [1,2,3,4,5]$$

由于$position2(1) = position1(5)$，因此 $position1(1)$ 与 $position1(5)$ 互换位置 $velocity(1) = 5$，交换位置之后：$position1 = [1,4,3,2,5], position2 = [1,2,3,4,5]$

又因为$position2(2) = position1(4)$，因此$position1(2)$与$position1(4)$互换位置 $velocity(2) = 4$，交换位置之后：$position1 = [1,2,3,4,5]$，$position2 = [1,2,3,4,5]$

……

由此可以推导 $velocity = [5,4,3,4,5]$，定义$position2 - position1$即为速度

对于**速度乘以常数**解释为速度中的每一个值以该常数的概率保留从而不参与交互，避免过早陷入局部最优解。

所以得到对应的粒子更新部分代码为：

```python
# 定义速度更新函数
def update_velocity(self, x_best, position, c):
    velocity = []
    for i in range(len(position)):
        if position[i] != x_best[i]:
            j = np.where(position == x_best[i])[0][0]
            so = (i, j, c)  # 得到交换子
            velocity.append(so)
            position[i], position[j] = position[j], position[i]  # 执行交换操作
    return velocity

# 定义位置更新函数
def update_position(self, position, ss):
    for i, j, r in ss:
        rand = np.random.random()
        if rand <= r:
            position[i], position[j] = position[j], position[i]
    return position

def update_particle(self, particle):

    # 计算交换序列，即 v = r1(pbest-xi) + r2(gbest-xi)
    ss1 = self.update_velocity(particle.best_position, particle.position, self.c1)
    ss2 = self.update_velocity(self.global_best_position, particle.position, self.c2)
    new_velocity = ss1 + ss2
    new_position = self.update_position(particle.position, new_velocity)

    particle.position = new_position
    particle.velocity = new_velocity
```

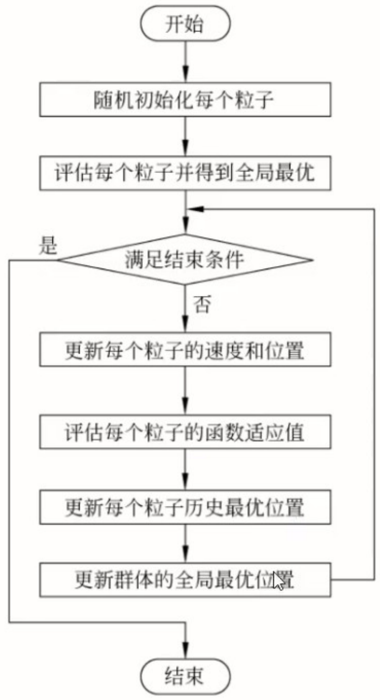
python

之后是 PSO 算法类的主函数部分，会循环迭代 max_iter 次，在每一次循环中先更新粒子的位置，之后计算粒子的适应度，如果当前新的适应度比对应粒子历史记录中适应度更小，则更新对应的适应度记录；同样对于全局的适应度也是同理 进行更新

```python
def optimize(self):
    for iter in range(self.max_iter):
        for particle in self.particles:
            self.update_particle(particle)

        for particle in self.particles:
            fitness = self.evaluate_fitness(particle)
            if fitness < particle.best_fitness:
                particle.best_fitness = fitness
                particle.best_position = particle.position.copy()
            if fitness < self.global_best_fitness:
                self.global_best_fitness = fitness
                self.global_best_position = particle.position.copy()
```

对应的流程图和伪代码与实验文档所提供的相同




```
//功能：粒子群优化算法伪代码
//说明：本例以求问题最小值为目标
//参数：N为群体规模
procedure PSO
  for each particle i
    Initialize velocity Vi and position Xi for
    particle i
    Evaluate particle i and set pBesti=Xi
  end for
  gBest=min{pBesti}
  while not stop
    for i=1 to N
       Update the velocity and position of
       particle i
      Evaluate particle i
      if fit (Xi)<fit(pBesti)
        pBesti=Xi;
      if fit(pBesti)<fit (gBest)
        gBest= pBesti;
      end for
  end while
  print gBest
end procedure
```

这样经过多次迭代之后，便完成了 PSO 算法，寻找到一个相对较优的解，编写一个主函数用于测试

```python
if __name__ == '__main__':
    num_cities = 5
    INF = 1000
    distance_matrix = [[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]]

    pso_tsp = PSO_TSP(num_cities, distance_matrix)
    pso_tsp.optimize()

    print("最优路径: ", pso_tsp.global_best_position)
    print("最短路径长度: ", pso_tsp.global_best_fitness)
```

四、实验结果

运行代码之后输出题目的解，可以发现该解确实是最优解

```
D:\anaconda3\python.exe D:\Desktop\learning\3.2\人工智能导论\lab\lab3\code.py
最优路径： [3 2 4 1 0]
最短路径长度： 23

Process finished with exit code 0
```

和实验二一样，对较大的数据集 st70.tsp 进行操作，该部分具体的读取文件代码与实验二完全相同，可以参照之前的实验报告

设置的超参数为粒子数为 100，迭代次数为 1000 次，其余的参数使用 SPO 类的默认值，最后跑出的结果如下图所示：

```
D:\anaconda3\python.exe D:\Desktop\learning\3.2\人工智能导论\lab\lab3\code.py
最优路径： [26 56 48 54 24 42  7 67 33 46 52 57 69 40 41 43 16 17 63 27 25 31  9 37
 51 14 38 39 59 21  2  8 11 62 10 44 55 45 29 20  5 15 28 30  3 66 47 64
 61 53 60 23 18  4 12  1 49 58 36 50 68 22 13 19 35  6 32 34  0 65]
最短路径长度： 3113.5998283793456

Process finished with exit code 0
```

五、思考题

与实验二遗传算法对比，总结粒子群算法和遗传算法的优劣：

算法	描述
粒子群算法优点	1.算法简单易于理解，PSO 算法的概念相对简单，易于理解和实现，因此适用于初学者和快速原型开发。 2.收敛速度较快，PSO 算法在搜索空间中的“粒子”之间共享信息，这有助于加速收敛速度，尤其是在高维空间中。 3.参数调整少，相比于遗传算法，PSO 算法的参数较少，且不太敏感，通常不需要大量的参数调整。
粒子群算法缺点	1.可能陷入局部最优，由于粒子之间的信息共享，PSO 算法容易受到局部最优解的影响，难以跳出局部最优。 2.对高维空间的适应性有限，在高维空间中，粒子之间的搜索可能变得低效，导致算法性能下降。 3.对于离散型的问题实际速度和位置设计可能比较困难
遗传算法优点	1.全局搜索能力强，遗传算法通过种群的进化过程进行全局搜索，有较强的寻优能力，能够跳出局部最优解。 2.适应性广泛，遗传算法适用于不同类型的问题，包括连续型、离散型和组合型等多种优化问题。 3.可并行化，由于遗传算法的种群性质，可以很容易地实现并行化，加速搜索过程。
遗传算法缺点	1.参数设置较多，遗传算法需要调整较多的参数，如种群大小、交叉概率、变异概率等，且对参数敏感，需要较多参数调优工作。 2.计算成本较高，由于遗传算法涉及到种群的进化过程，每一代都需要对种群进行选择、交叉和变异等操作，计算成本较高。 3.编码方式影响效率，遗传算法的性能可能受到编码方式的影响，不同的问题可能需要不同的编码方式和操作符。

综合来看，如果问题空间较简单，且要快速解决方案，则可以选择粒子群算法；如果问题空间较复杂，需要更全面的搜索能力，则遗传算法可能更适合。