### 11. 人工蜂群算法（ABC）+ 差分进化（DE）组合模型案例题目

**题目：工业机器人焊接路径优化问题**

* **问题背景**：某汽车零部件工厂的焊接机器人需对变速箱壳体的 20 个焊点进行焊接，当前路径存在重复移动（空行程占比 30%），导致单件焊接时间达 8 分钟，制约产能。机器人运动时需避障（如夹具、凸起结构），最小转弯半径 0.5m。
* **问题描述**：需优化焊接点的访问顺序和运动路径，目标包括：① 最小化总运动时间（≤5 分钟）；② 最小化路径长度；③ 确保相邻焊点的运动轨迹平滑（转弯角度≤90°）。需处理机器人运动学的非线性约束。
* **数据情况**：提供各焊点的三维坐标（x,y,z）、机器人当前运动路径记录、各轴运动速度参数（m/s）、障碍物的位置和尺寸、不同路径下的实际焊接时间。

### 11. 人工蜂群算法（ABC）+ 差分进化（DE）求解工业机器人焊接路径优化代码

|  |
| --- |
| import numpy as np  import matplotlib.pyplot as plt  import random  import copy  from mpl\_toolkits.mplot3d import Axes3D  from matplotlib.patches import Circle, PathPatch  from matplotlib.path import Path  import time  # 设置随机种子，保证结果可复现  np.random.seed(42)  random.seed(42)  # 问题数据初始化  def initialize\_welding\_data(n\_points=25, workspace\_size=(1000, 800, 500)):  """  初始化工业机器人焊接路径优化问题数据  n\_points: 焊接点数量  workspace\_size: 工作空间尺寸 (x, y, z)，单位：毫米  """  # 生成焊接点坐标（3D空间）  x\_range, y\_range, z\_range = workspace\_size  points = np.zeros((n\_points, 3))    # 模拟工件表面的焊接点分布（假设为不规则曲面）  # 基础曲面：轻微弯曲的平面  base\_curve = np.linspace(0, x\_range, n\_points)  z\_base = 100 + 50 \* np.sin(base\_curve / x\_range \* 2 \* np.pi) # 基础高度    for i in range(n\_points):  points[i, 0] = random.uniform(0, x\_range) # x坐标  points[i, 1] = random.uniform(0, y\_range) # y坐标  points[i, 2] = z\_base[i] + random.uniform(-30, 30) # z坐标（在曲面上波动）    # 焊接点属性  # 1. 焊接难度系数（1-5，影响焊接时间）  difficulty = np.random.randint(1, 6, size=n\_points)    # 2. 焊接精度要求（1-5，影响路径平滑度需求）  precision = np.random.randint(1, 6, size=n\_points)    # 3. 热变形系数（相邻点焊接顺序影响变形量）  heat\_deformation = np.zeros((n\_points, n\_points))  for i in range(n\_points):  for j in range(n\_points):  if i != j:  # 距离越近，热变形影响越大  dist = np.sqrt(np.sum((points[i] - points[j])\*\*2))  heat\_deformation[i, j] = max(0, 1 - dist / 500) # 500mm为影响阈值    # 4. 机器人移动参数  robot\_params = {  'max\_speed': 500, # 最大移动速度（mm/s）  'acceleration': 200, # 加速度（mm/s²）  'tool\_radius': 15, # 焊枪半径（mm）  'min\_distance': 10 # 最小安全距离（mm）  }    # 5. 焊接工艺参数  welding\_params = {  'base\_time': 2.0, # 基础焊接时间（秒）  'speed\_factor': 0.005, # 移动时间系数  'precision\_factor': 0.3 # 精度对时间的影响系数  }    return {  'points': points,  'difficulty': difficulty,  'precision': precision,  'heat\_deformation': heat\_deformation,  'robot\_params': robot\_params,  'welding\_params': welding\_params,  'n\_points': n\_points,  'workspace\_size': workspace\_size  }  # 路径成本计算  def calculate\_path\_cost(path, data):  """  计算焊接路径的总成本  path: 焊接点索引的有序列表  成本包括：移动时间 + 焊接时间 + 热变形惩罚  """  if len(path) <= 1:  return 0.0    points = data['points']  n\_points = data['n\_points']  robot = data['robot\_params']  welding = data['welding\_params']    total\_cost = 0.0  total\_move\_time = 0.0  total\_weld\_time = 0.0  deformation\_penalty = 0.0    # 计算移动时间和焊接时间  for i in range(len(path)):  # 焊接时间（与难度和精度相关）  point\_idx = path[i]  weld\_time = welding['base\_time'] \* data['difficulty'][point\_idx] \* (1 + 0.1 \* data['precision'][point\_idx])  total\_weld\_time += weld\_time    # 移动时间（到下一个点）  if i < len(path) - 1:  next\_idx = path[i+1]  dist = np.sqrt(np.sum((points[point\_idx] - points[next\_idx])\*\*2))    # 计算移动时间（考虑加速到最大速度）  # 加速阶段距离：v²/(2a)  max\_speed = robot['max\_speed']  acceleration = robot['acceleration']  加速距离 = (max\_speed\*\*2) / (2 \* acceleration)    if dist <= 2 \* 加速距离:  # 未达到最大速度就需要减速  time = 2 \* np.sqrt(dist / acceleration)  else:  # 达到最大速度后匀速运动  time = (2 \* 加速距离) / max\_speed + (dist - 2 \* 加速距离) / max\_speed    total\_move\_time += time    # 热变形惩罚（基于焊接顺序）  deformation\_penalty += data['heat\_deformation'][point\_idx][next\_idx] \* 2.0 # 惩罚系数    # 总路径成本：时间（秒）+ 惩罚  total\_cost = total\_move\_time + total\_weld\_time + deformation\_penalty    return total\_cost, {  'move\_time': total\_move\_time,  'weld\_time': total\_weld\_time,  'deformation': deformation\_penalty  }  # 差分进化（DE）算法组件  class DifferentialEvolution:  def \_\_init\_\_(self, data, pop\_size=30, f=0.5, cr=0.7, max\_iter=50):  """  差分进化算法初始化  pop\_size: 种群大小  f: 缩放因子  cr: 交叉概率  max\_iter: 最大迭代次数  """  self.data = data  self.pop\_size = pop\_size  self.f = f # 缩放因子  self.cr = cr # 交叉概率  self.max\_iter = max\_iter  self.n\_points = data['n\_points']    # 初始化种群  self.population = []  self.fitness = []    # 生成初始路径（随机排列）  for \_ in range(pop\_size):  path = list(np.random.permutation(self.n\_points))  self.population.append(path)  cost, \_ = calculate\_path\_cost(path, data)  self.fitness.append(cost)    # 记录最优解  self.best\_idx = np.argmin(self.fitness)  self.best\_path = self.population[self.best\_idx]  self.best\_cost = self.fitness[self.best\_idx]  self.history = [self.best\_cost]    def mutate(self, idx):  """变异操作：生成变异向量"""  # 随机选择3个不同的个体  idxs = list(range(self.pop\_size))  idxs.remove(idx)  a, b, c = random.sample(idxs, 3)    # 生成变异向量  mutant = []  for i in range(self.n\_points):  # 基于位置的差分变异  val = self.population[c][i] + self.f \* (self.population[a][i] - self.population[b][i])  # 确保在有效范围内  val = int(round(np.clip(val, 0, self.n\_points - 1)))  mutant.append(val)    # 修复重复元素（保持排列特性）  return self.\_repair\_duplicates(mutant)    def crossover(self, target, mutant):  """交叉操作：生成试验向量"""  trial = []  for i in range(self.n\_points):  if random.random() < self.cr or i == random.randint(0, self.n\_points - 1):  trial.append(mutant[i])  else:  trial.append(target[i])    # 修复重复元素  return self.\_repair\_duplicates(trial)    def \_repair\_duplicates(self, vector):  """修复向量中的重复元素，确保是有效排列"""  seen = set()  duplicates = []  new\_vector = []    for val in vector:  if val not in seen:  seen.add(val)  new\_vector.append(val)  else:  duplicates.append(val)    # 补充缺失的元素  missing = [x for x in range(self.n\_points) if x not in seen]  for i in range(len(duplicates)):  new\_vector.append(missing[i])    return new\_vector    def select(self, trial, idx):  """选择操作：保留较优个体"""  trial\_cost, \_ = calculate\_path\_cost(trial, self.data)  if trial\_cost < self.fitness[idx]:  self.population[idx] = trial  self.fitness[idx] = trial\_cost    # 更新全局最优  if trial\_cost < self.best\_cost:  self.best\_path = trial  self.best\_cost = trial\_cost    def optimize(self):  """执行差分进化优化"""  print(f"DE初始最优成本: {self.best\_cost:.2f}秒")    for iter in range(self.max\_iter):  for i in range(self.pop\_size):  # 变异  mutant = self.mutate(i)  # 交叉  trial = self.crossover(self.population[i], mutant)  # 选择  self.select(trial, i)    # 记录历史  self.history.append(self.best\_cost)    # 定期输出  if (iter + 1) % 10 == 0:  print(f"DE迭代{iter+1}/{self.max\_iter}, 最优成本: {self.best\_cost:.2f}秒")    return self.best\_path, self.best\_cost  # 人工蜂群算法（ABC）组件  class ArtificialBeeColony:  def \_\_init\_\_(self, data, de\_solution=None, colony\_size=30, limit=20, max\_iter=50):  """  人工蜂群算法初始化  colony\_size: 蜂群大小  limit: 放弃食物源的限制次数  max\_iter: 最大迭代次数  de\_solution: 差分进化得到的初始解（用于初始化）  """  self.data = data  self.colony\_size = colony\_size  self.limit = limit # 侦查蜂阈值  self.max\_iter = max\_iter  self.n\_points = data['n\_points']    # 雇佣蜂和观察蜂各占一半  self.employed\_bees = colony\_size // 2  self.onlooker\_bees = colony\_size - self.employed\_bees    # 初始化食物源（焊接路径）  self.food\_sources = []  self.fitness = []  self.trials = [] # 记录各食物源未改进的次数    # 如果有DE的初始解，优先使用  if de\_solution is not None:  self.food\_sources.append(de\_solution)  cost, \_ = calculate\_path\_cost(de\_solution, data)  self.fitness.append(cost)  self.trials.append(0)  # 补充其他随机解  for \_ in range(colony\_size - 1):  path = list(np.random.permutation(self.n\_points))  self.food\_sources.append(path)  cost, \_ = calculate\_path\_cost(path, data)  self.fitness.append(cost)  self.trials.append(0)  else:  # 全部随机初始化  for \_ in range(colony\_size):  path = list(np.random.permutation(self.n\_points))  self.food\_sources.append(path)  cost, \_ = calculate\_path\_cost(path, data)  self.fitness.append(cost)  self.trials.append(0)    # 记录最优解  self.best\_idx = np.argmin(self.fitness)  self.best\_path = self.food\_sources[self.best\_idx]  self.best\_cost = self.fitness[self.best\_idx]  self.history = [self.best\_cost]    def employed\_bee\_phase(self):  """雇佣蜂阶段：搜索邻域解"""  for i in range(self.employed\_bees):  current\_path = self.food\_sources[i]  current\_cost = self.fitness[i]    # 生成邻域解（随机交换两个点）  neighbor\_path = self.\_generate\_neighbor(current\_path)    # 评估新解  neighbor\_cost, \_ = calculate\_path\_cost(neighbor\_path, self.data)    # 贪婪选择  if neighbor\_cost < current\_cost:  self.food\_sources[i] = neighbor\_path  self.fitness[i] = neighbor\_cost  self.trials[i] = 0 # 重置计数    # 更新全局最优  if neighbor\_cost < self.best\_cost:  self.best\_path = neighbor\_path  self.best\_cost = neighbor\_cost  else:  self.trials[i] += 1 # 增加计数    def onlooker\_bee\_phase(self):  """观察蜂阶段：根据适应度选择食物源"""  # 计算选择概率（基于成本的倒数）  total\_fitness = sum(1.0 / cost for cost in self.fitness[:self.employed\_bees])  probabilities = [(1.0 / cost) / total\_fitness for cost in self.fitness[:self.employed\_bees]]    for \_ in range(self.onlooker\_bees):  # 轮盘赌选择食物源  idx = np.random.choice(self.employed\_bees, p=probabilities)  current\_path = self.food\_sources[idx]  current\_cost = self.fitness[idx]    # 生成邻域解（随机反转一段路径）  neighbor\_path = self.\_generate\_neighbor(current\_path, method='reverse')    # 评估新解  neighbor\_cost, \_ = calculate\_path\_cost(neighbor\_path, self.data)    # 贪婪选择  if neighbor\_cost < current\_cost:  self.food\_sources[idx] = neighbor\_path  self.fitness[idx] = neighbor\_cost  self.trials[idx] = 0 # 重置计数    # 更新全局最优  if neighbor\_cost < self.best\_cost:  self.best\_path = neighbor\_path  self.best\_cost = neighbor\_cost  else:  self.trials[idx] += 1 # 增加计数    def scout\_bee\_phase(self):  """侦查蜂阶段：替换长时间未改进的食物源"""  for i in range(self.employed\_bees):  if self.trials[i] > self.limit:  # 随机生成新解  new\_path = list(np.random.permutation(self.n\_points))  new\_cost, \_ = calculate\_path\_cost(new\_path, self.data)    # 替换旧解  self.food\_sources[i] = new\_path  self.fitness[i] = new\_cost  self.trials[i] = 0    # 可能的局部优化：应用2-opt操作  improved = True  while improved:  improved = False  for a in range(len(new\_path) - 1):  for b in range(a + 1, len(new\_path)):  if b - a == 1:  continue # 相邻点不交换  # 反转a到b之间的路径  reversed\_path = new\_path[:a] + new\_path[a:b+1][::-1] + new\_path[b+1:]  reversed\_cost, \_ = calculate\_path\_cost(reversed\_path, self.data)  if reversed\_cost < new\_cost:  new\_path = reversed\_path  new\_cost = reversed\_cost  improved = True  break  if improved:  break    # 更新解  self.food\_sources[i] = new\_path  self.fitness[i] = new\_cost    # 更新全局最优  if new\_cost < self.best\_cost:  self.best\_path = new\_path  self.best\_cost = new\_cost    def \_generate\_neighbor(self, path, method='swap'):  """生成邻域解"""  neighbor = copy.copy(path)    if method == 'swap':  # 随机交换两个点  i, j = random.sample(range(len(path)), 2)  neighbor[i], neighbor[j] = neighbor[j], neighbor[i]  elif method == 'reverse':  # 随机反转一段路径  i, j = sorted(random.sample(range(len(path)), 2))  neighbor[i:j+1] = neighbor[i:j+1][::-1]  elif method == 'insert':  # 随机插入一个点  i, j = random.sample(range(len(path</doubaocanvas> |