

## 多种群进化优化

Geatpy 支持多种群进化优化，多种群的数据结构是一个列表，列表的每个元素是每种种群对象（Population 类对象），可以对已有的算法模板略加修改，实现多种群进化协同与竞争进化。Geatpy 内置的进化算法模板中带“multi”字样的为多种群的进化算法模板，下面以 soea\_multi\_SEGA\_templet 算法模板为例，剖析如何进行基本的多种群协同进化。

查看“soea\_multi\_SEGA\_templet.py”，代码如下：

```
# -*- coding: utf-8 -*-
import geatpy as ea # 导入geatpy库
import numpy as np
from sys import path as paths
from os import path
paths.append(path.split(path.split(path.realpath(__file__))[0])[0])

class soea_multi_SEGA_templet(ea.SoeaAlgorithm):

    """
    soea_multi_SEGA_templet : class - Multi-population Strengthen Elitist
        GA templet(增强精英保留的多种群协同遗传算法模板)

    模板说明：
    该模板是内置算法模板soea_SEGA_templet的多种群协同版本，为不同的种群设置不同的重组和变异概率。
    注意：本算法模板中的population为一个存储种群类对象的列表，
        而不是单个种群类对象。

    算法描述：
    本模板实现的是增强精英保留的多种群协同遗传算法。算法流程如下：
    1) 循环population列表，初始化列表中的各个种群的染色体，并将所有种群所有个体的数目记录为NindAll。
    2) 若满足进化算法停止条件则停止，否则继续执行。
    3)
        循环对各个种群独立进行选择、重组、变异，得到各个种群的子代，并将父代和子代个体合并。
    4) 对所有种群的所有个体进行统一的适应度评价。
    5)
        根据适应度调用选择算子进行环境选择，选择出NindAll个个体形成新一代种群。
    6) 根据概率进行种群间个体迁移。
    7) 回到第2步。
    """

    def __init__(self, problem, population):
        ea.SoeaAlgorithm.__init__(self, problem, population) # 先调用父类构造方法
        if type(population) != list:
            raise RuntimeError('传入的种群对象列表必须为list类型')
        self.name = 'multi-SEGA'
        self.PopNum = len(population) # 种群数目
        self.selFunc = 'tour' # 锦标赛选择算子
        self.migFr = 5 # 发生种群迁移的间隔代数
        self.migOpsers = ea.Migrate(MIGR = 0.2, Structure = 2, Select = 1, Replacement = 2) # 生成种群迁移算子对象
        # 为不同的种群设置不同的重组、变异算子
        self.recOpsers = []
        self.mutOpsers = []
        Pms = np.linspace(1/self.problem.Dim, 1, self.PopNum) # 生成变异概率列表，为不同的种群分配不同的变异概率
        Pcs = np.linspace(0.2, 1, self.PopNum) # 生成重组概率列表，为不同的种群分配不同的重组概率
        for i in range(self.PopNum): # 遍历种群列表
            pop = population[i] # 得到当前种群对象
            if pop.Encoding == 'P':
                recOper = ea.Xovpmx(XOVR = Pcs[i]) # 生成部分匹配交叉算子对象
                mutOper = ea.Mutinv(Pm = float(Pms[i])) # 生成逆转变异算子对象
            else:
                recOper = ea.Xovdp(XOVR = Pcs[i]) # 生成两点交叉算子对象
            if pop.Encoding == 'BG':
                mutOper = ea.Mutbin(Pm = float(Pms[i])) # 生成二进制变异算子对象
            elif pop.Encoding == 'RI':
                mutOper = ea.Mutbga(Pm = float(Pms[i]), MutShrink = 0.5, Gradient = 20) # 生成breeder GA变异算子对象
            else:
                raise RuntimeError('编码方式必须为'BG'、'RI'或'P'.')
            self.recOpsers.append(recOper)
            self.mutOpsers.append(mutOper)

        def unite(self, population):
            """
            合并种群，生成联合种群。
            注：返回的unitePop不携带Field和Chrom的信息，因为其Encoding=None。
            """
            # 遍历种群列表，构造联合种群
            unitePop = ea.Population(None, None, population[0].sizes, None,
                                     ObjV = population[0].ObjV,
                                     FitnV = population[0].FitnV,
                                     CV = population[0].CV,
                                     Phen = population[0].Phen)
            for i in range(1, self.PopNum):
                unitePop += population[i]
            return unitePop

        def calFitness(self, population):
            """
            计算种群个体适应度，population为种群列表
            该函数直接对输入参数population中的适应度信息进行修改，
            因此函数不用返回任何参数。
            """
            ObjV = np.vstack(list(pop.ObjV for pop in population))
            CV = np.vstack(list(pop.CV for pop in population))
            FitnV = ea.scaling(ObjV, CV, self.problem.maxormins) # 统一计算适应度
            # 为各个种群分配适应度
            idx = 0
            for i in range(self.PopNum):
                population[i].FitnV = FitnV[idx: idx + population[i].sizes]
                idx += population[i].sizes

        def EnvSelection(self, population, NUM): # 环境选择，选择个体保留到下一代
            FitnVs = list(pop.FitnV for pop in population)
            NewChrxs = ea.mselecting('dup', FitnVs, NUM) # 采用基于适应度排序的直接复制选择
            for i in range(self.PopNum):
                population[i] = (population[i])[NewChrxs[i]]
            return population

        def run(self, prophetPops = None): # prophetPops为先知种群列表（即包含先验知识的种群列表）
            #=====初始化配置=====
            self.initialization() # 初始化算法模板的一些动态参数
            population = self.population # 密切注意本模板的population是一个存储种群类对象的列表
            NindAll = 0 # 记录所有种群个体总数
            #=====准备进化=====
            for i in range(self.PopNum): # 遍历每个种群，初始化每个种群的染色体矩阵
                NindAll += population[i].sizes
                population[i].initChrom(population[i].sizes) # 初始化种群染色体矩阵
                self.call_aimFunc(population[i]) # 计算种群的目标函数值
                # 插入先验知识
                #（注意：这里不会对先知种群列表prophetPops的合法性进行检查）
                if prophetPops is not None:
                    population[i] = (prophetPops[i] + population[i][:population[i].sizes]) # 插入先知种群
                self.calFitness(population) # 统一计算适应度
                unitePop = self.unite(population) # 得到联合种群unitePop
            #=====开始进化=====
            while self.terminated(unitePop) == False:
                for i in range(self.PopNum): # 遍历种群列表，分别对各个种群进行重组和变异
                    pop = population[i] # 得到当前种群
                    # 选择
                    offspring = pop[ea.selecting(self.selFunc, pop.FitnV, pop.sizes)]
                    # 进行进化操作
                    offspring.Chrom = self.recOpsers[i].do(offspring.Chrom) # 重组
                    offspring.Chrom = self.mutOpsers[i].do(offspring.Encoding, offspring.Chrom, offspring.Field) # 变异
                    # 求进化后个体的目标函数值
                    offspring.Phen = offspring.decoding() # 染色体解码
                    self.call_aimFunc(offspring) # 计算目标函数值
                    population[i] = population[i] + offspring # 父子合并
                    self.calFitness(population) # 统一计算适应度
                    population = self.EnvSelection(population, NUM = NindAll) # 选择个体得到新一代种群
                if self.currentGen % self.migFr == 0:
                    population = self.migOpsers.do(population) # 进行种群迁移
                    unitePop = self.unite(population) # 更新联合种群
            return self.finishing(unitePop) # 调用finishing完成后续工作并返回结果
```

代码分析：

上述代码与 soea\_SEGA\_templet 的最大区别是里面的 population 是一个列表，而不是单一的种群对象。在初始化算法模板对象时，为不同的种群分配不同概率的重组和变异算子。由于该算法模板继承的是 Geatpy 框架中的 SoeaAlgorithm 类，意味着在每一代对个体进行进化记录的时候面对的是所有个体（详见 Algorithm.py），即算法模板所继承的 stat() 函数传入的是一个种群类，因此，上述代码新增了一个 unite() 函数，用于将种群列表中的所有种群合并成一个种群。

关于合并种群这一块需要注意的是：多种群允许各个种群的编码方式、译码矩阵、染色体结构都相互不一样，因此不能直接将种群列表中的所有种群对象通过“+”运算符进行合并。此时需要用到 Population 类的一种特殊用法：设置 Encoding 为 None。详见 Population.py 源码。一旦设置了 Encoding 为 None，种群类将清空译码矩阵 Field 和染色体矩阵 Chrom 的信息，使得种群在不携带与染色体直接相关的信息之下能够合并。另外，正因为在这里我们只需要利用合并的联合种群来记录一些与染色体无直接相关的信息，如最优个体的目标函数值、种群平均目标函数值、最优个体的决策变量值等（详见 Algorithm.py 中的 SoeaAlgorithm 类的 stat() 函数代码），因此可以用这种设置 Encoding 为 None 的方法来方便合并种群。

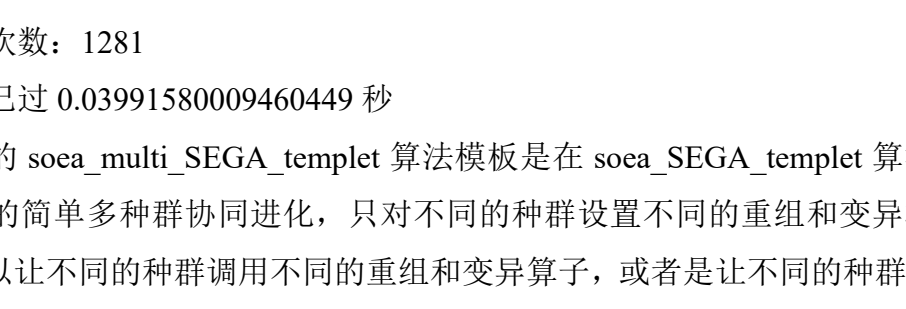
在进化过程中，需要遍历种群列表中的每一个种群进行：选择个体、对所选的个体进行重组和变异生成子代、将父代和子代合并。这个过程与 soea\_SEGA\_templet 是完全一样的。合并种群后，同样是从父子合并的种群中择优保留个体形成新一代种群。不同之处在于，多种群的版本在生成新一代种群时，调用的是内核函数 mselecting() 来选择。该函数是通过传入各个种群的适应度列向量，在所有种群的所有个体中筛选个体，同时保证每个种群起码有一定的个体被选择。

此外，上述代码中还设置了每连续进化一定的代数时进行种群间的个体迁移。要注意种群间个体迁移仅适用于每个种群的编码方式、译码矩阵、染色体结构相同的情况。当每个种群的编码方式、译码矩阵、染色体结构有不相同的时候，无法进行种群间个体迁移。

另外，由于多种群的进化算法在初始化时需要传入的是种群列表而不是单一种群对象，因此在执行脚本里面实例化算法模板对象时需要传入种群列表。以 soea\_demo9 为例，其执行脚本 main.py 的代码如下：

```
# -*- coding: utf-8 -*-
import numpy as np
import geatpy as ea # import geatpy
from MyProblem import MyProblem # 导入自定义问题接口
if __name__ == '__main__':
    """=====实例化问题对象====="""
    problem = MyProblem() # 生成问题对象
    """=====种群设置====="""
    Encoding = 'RI' # 编码方式
    NINDs = [5, 10, 15, 20] # 种群规模
    population = [] # 创建种群列表
    for i in range(len(NINDs)):
        Field = ea.crtfld(Encoding, problem.varTypes, problem.ranges, problem.borders) # 创建区域描述器
        population.append(ea.Population(Encoding, Field, NINDs[i])) # 实例化种群对象（此时种群还没被初始化，仅仅是完成种群对象的实例化）
    """=====算法参数设置====="""
    myAlgorithm = ea.soea_multi_SEGA_templet(problem, population) # 实例化一个算法模板对象
    myAlgorithm.MAXGEN = 50 # 最大进化代数
    myAlgorithm.trappedValue = 1e-6 # “进化停滞”判断阈值
    # 进化停滞计数器最大上限值，
    # 如果连续maxTrappedCount代被判定进化陷入停滞，则终止进化
    myAlgorithm.maxTrappedCount = 5
    """=====调用算法模板进行种群进化====="""
    [pop, obj_trace, var_trace] = myAlgorithm.run() # 执行算法模板
    pop.save() # 把最后一代种群的信息保存到文件中
    # 输出结果
    best_gen = np.argmin(problem.maxormins * obj_trace[:, 1]) # 记录最优种群个体是在哪一代
    best_ObjV = obj_trace[best_gen, 1]
    print('最优的目标函数值为：'%s%(best_ObjV))
    print('最优的控制变量值为：')
    for i in range(var_trace.shape[1]):
        print(var_trace[best_gen, i])
    print('有效进化代数：%s'%(obj_trace.shape[0]))
    print('最优的一代是第 %s 代'%(best_gen + 1))
    print('评价次数：%s'%(myAlgorithm.evalsNum))
    print('时间已过 %s 秒'%(myAlgorithm.passTime))
```

从中可以看出在实例化种群 soea\_multi\_SEGA\_templet 算法模板对象时，需要传入一个种群列表。执行 main.py 得到的结果如下：



种群信息导出完毕。

最优的目标函数值为：3.8502737666865334

最优的控制变量值为：

1.85054716726561

有效进化代数：25

最优的一代是第 22 代

评价次数：1281

时间已过 0.03991580009460449 秒

上面的 soea\_multi\_SEGA\_templet 算法模板是在 soea\_SEGA\_templet 算法模板的基础上实现的简单多种群协同进化，只对不同的种群设置不同的重组和变异概率。事实上，还可以让不同的种群调用不同的重组和变异算子，或者是让不同的种群在不同的变量搜索区间上进行进化优化（通过设置不同的译码矩阵 Field），另外甚至可以让不同的种群使用不同的进化算法来进行进化优化。例如：让每个种群用独特的进化算法跑若干代，在各个种群得出较优结果后，进行一次竞争选择（调用 mselecting），然后再继续让每个种群用独特的进化算法跑若干代。这些都可以通过自定义算法模板来实现。