进化算法模板

1. 进化算法模板类一览

进化算法模板是 Geatpy 进化算法框架的核心。进化算法的核心流程是部体现在进化算法模板类中。

Geatpy 目前内置以下进化算法模板，详细的算法流程可查看对应的源码。

- soea_DE_best_1_bin_templet (差分进化 DE/best/1/bin 算法模板)
- soea_DE_best_1_L_templet (差分进化 DE/best/1/L 算法模板)
- soea_DE_rand_1_bin_templet (差分进化 DE/rand/1/bin 算法模板)
- soea_DE_rand_1_L_templet (差分进化 DE/rand/1/L 算法模板)
- soea_DE_targetToBest_1_bin_templet (差分进化 DE/targetToBest/1/bin 算法模板)
- soea_DE_targetToBest_1_L_templet (差分进化 DE/targetToBest/1/L 算法模板)
- soea_ES_1_plus_1_templet ((1+1) 进化策略模板)
- soea_EGA_templet (精英保留的遗传算法模板)
- soea_SEGA_templet (增强精英保留的遗传算法模板)
- soea_SGA_templet (最简单、最经典的遗传算法模板)
- soea_GGAP_SGA_templet (带代沟的简单遗传算法模板)
- soea_studGA_templet (种马遗传算法模板)
- soea_steady_GA_templet (稳态遗传算法模板)
- soea_psy_EGA_templet (精英保留的多染色体遗传算法模板)
- soea_psy_SEGA_templet (增强精英保留的多染色体遗传算法模板)
- soea_psy_SGA_templet (最简单、最经典的多染色体遗传算法模板)
- soea_psy_GGAP_SGA_templet (带代沟的多染色体简单遗传算法模板)
- soea_psy_studGA_templet (多染色体种马遗传算法模板)
- soea_psy_steady_GA_templet (多染色体稳态遗传算法模板)
- soea_multi_SEGA_templet (增强精英保留的多种群协同遗传算法模板)
- moea_awGA_templet (基于 awGA 算法的多目标进化算法模板)
- moea_NSGA2_DE_templet (基于 NSGA-II-DE 算法的多目标进化算法模板)
- moea_NSGA2_archive_templet (带全局存档的多目标进化 NSGA-II 算法模板)
- moea_NSGA2_templet (基于 NSGA-II 算法的多目标进化算法模板)
- moea_NSGA3_DE_templet (基于 NSGA-III-DE 算法的多目标进化算法模板)
- moea_NSGA3_templet (基于 NSGA-III 算法的多目标进化算法模板)
- moea_RVEA_templet (基于 RVEA 算法的多目标进化算法模板)
- moea_RVEA_RES_templet (基于带参考点再生策略的 RVEA 算法的多目标进化算法模板)
- moea_psy_awGA_templet (基于 awGA 算法的多染色体多目标进化算法模板)
- moea_psy_NSGA2_archive_templet (带全局存档的多染色体多目标进化 NSGA-II 算法模板)
- moea_psy_NSGA2_templet (基于 NSGA-II 算法的多染色体多目标进化算法模板)
- moea_psy_NSGA3_templet (基于 NSGA-III 算法的多染色体多目标进化算法模板)
- moea_psy_RVEA_templet (基于 RVEA 算法的多染色体多目标进化算法模板)
- moea_psy_RVEA_RES_templet (基于带参考点再生策略的多染色体 RVEA 算法的多目标进化算法模板)

其中，以“soea”开头的是单目标进化算法模板，以“moea”开头的是多目标进化算法模板。带“psy”字样的是多染色体进化算法模板，带“multi”字样的是多种群进化算法模板。前面的“Geatpy 总览”章节中提到“Algorithm”是进化算法模板的顶级父类。但上述这些进化算法模板并不是直接继承该顶级父类，而是继承顶级父类之下的“中间”算法模板类。

“中间”算法模板类有“单目标进化优化算法模板类”(SoeaAlgorithm) 和“多目标进化优化算法模板类”(MoeaAlgorithm)。这两个算法模板类定义了一些在具体的进化算法中公用的一些属性以及实现了一些公用的函数。比如用于记录进化过程动态图的“ax”变量、进化记录器等属性，以及判断是否终止进化的“terminated()”函数、进化完成后调用的用于一些后续处理的函数“finishing()”、在进化过程中用于分析记录的“stat()”函数等等。这些属性和函数都是算法模板顶级父类没有定义的，且具体算法模板类公用的一些属性和函数。

值得注意的是：在这些“中间”算法模板类中定义的属性和函数都是与具体进化算法核心流程无关的。并且建议在自定义进化算法模板时，不要把一些与具体进化算法核心流程有关的代码放在这些“中间类”中，如果有连续地自定义了若干个非常相近的算法模板，那么可以另外创建一个位于“中间类”和具体算法模板类之间的一个类。比如要自定义很多改进的 NSGA-II 算法，那么可以创建一个名为“BasicNSGA2”的类，来实现那些在所有改进 NSGA2 中公用的代码。

2. 多目标优化 NSGA-II 算法模板详解

下面以 NSGA-II 的算法模板为例，深入分析该进化算法的整个执行过程。

```python
# -*- coding: utf-8 -*-
import numpy as np
import geatpy as ea # 导入geatpy库
from sys import path as paths
from os import path
paths.append(path.split(path.split(path.realpath(__file__))[0])[0])
class moea_NSGA2_templet(ea.MoeaAlgorithm):
    """
    moea_NSGA2_templet : class - 多目标进化NSGA-II算法模板

    算法描述:
    采用NSGA-II进行多目标优化，算法详见参考文献[1]。

    模板使用注意:
    本模板调用的目标函数形如：aimFunc(pop),
    其中pop为Population类的对象，代表一个种群，
    pop对象的Phen属性（即种群染色体的表现型）等价于种群所有个体的决策变量组成的矩阵，
    该函数根据该Phen计算得到种群所有个体的目标函数值组成的矩阵，并将其赋值给pop对象的ObjV属性。
    若有约束条件，则在计算违反约束程度矩阵CV后赋值给pop对象的CV属性（详见Geatpy数据结构）。
    该函数不返回任何的返回值，求得的目标函数值保存在种群对象的ObjV属性中，
    违反约束程度矩阵保存在种群对象的CV属性中。
    例如：population为一个种群对象，则调用aimFunc(population)即可完成目标函数值的计算，
    此时可通过population.ObjV得到求得的目标函数值，population.CV得到违反约束程度矩阵。
    若不符合上述规范，则请修改算法模板或自定义新算法模板。

    参考文献:
    [1] Deb K , Pratap A , Agarwal S , et al. A fast and elitist
        multiobjective
    genetic algorithm: NSGA-II[J]. IEEE Transactions on Evolutionary
    Computation, 2002, 6(2):0-197.
    """
    def __init__(self, problem, population):
        ea.MoeaAlgorithm.__init__(self, problem, population) #
            先调用父类构造方法
        if str(type(population)) != "<class 'Population.Population'>":
            raise RuntimeError('传入的种群对象必须为Population类型')
        self.name = 'NSGA2'
        if self.problem.M < 10:
            self.ndSort = ea.ndsortESS # 采用ENS_SS进行非支配排序
        else:
            self.ndSort = ea.ndsortTNS #
                高维目标采用T_ENS进行非支配排序，速度一般会比ENS_SS要快
        self.selFunc = 'tour' # 选择方式，采用锦标赛选择
        if population.Encoding == 'P':
            self.recOper = ea.Xovpmx(XOVR = 1) # 生成部分匹配交叉算子对象
            self.mutOper = ea.Mutinv(Pm = 1) # 生成逆转变异算子对象
        elif population.Encoding == 'BG':
            self.recOper = ea.Xovud(XOVR = 1) # 生成均匀交叉算子对象
            self.mutOper = ea.Mutbin(Pm=None) #
                生成二进制变异算子对象,Pm设置为None时,具体取变异算子中的默认值
        elif population.Encoding == 'RI':
            self.recOper = ea.Recsbx(XOVR = 1, n = 20) #
                生成模拟二进制交叉算子对象
            self.mutOper = ea.Mutpolyn(Pm = 1/self.problem.Dim, DisI =
                20) # 生成多项式变异算子对象
        else:
            raise RuntimeError('编码方式必须为''BG''、''RI''或''P''.')

    def reinsertion(self, population, offspring, NUM):
        """
        描述:
        重插入个体产生新一代种群（采用父子合并选择的策略）。
        NUM为所需要保留到下一代的个体数目。
        注：这里对原版NSGA-II进行等价的修改：先按帕累托分级和拥挤距离来计算出种群个体的适应度，
        然后调用dup选择算子(详见help(ea.dup))来根据适应度从大到小的顺序选择出个体保留到下一代。
        这跟原版NSGA-II的选择方法所得的结果是完全一样的。
        """

        # 父子两代合并
        population = population + offspring
        # 选择个体保留到下一代
        [levels, criLevel] = self.ndSort(self.problem.maxormins *
            population.ObjV, NUM, None, population.CV) #
            对NUM个个体进行非支配分层
        dis = ea.crowdis(population.ObjV, levels) # 计算拥挤距离
        population.FitnV[:, 0] = np.argsort(np.lexsort(np.array([dis,
            -levels])), kind = 'mergesort') # 计算适应度
        chooseFlag = ea.selecting('dup', population.FitnV, NUM) #
            调用低级选择算子dup进行基于适应度排序的选择，保留NUM个个体
        return population[chooseFlag]

    def run(self):
        #==========================初始化配置===========================
        population = self.population
        NIND = population.sizes
        self.initialization() # 初始化算法模板的一些动态参数
        #===========================准备进化============================
        if population.Chrom is None:
            population.initChrom() #
                初始化种群染色体矩阵（内含解码，详见Population类的源码）
        else:
            population.Phen = population.decoding() # 染色体解码
        self.problem.aimFunc(population) # 计算种群的目标函数值
        self.evalsNum = population.sizes # 记录评价次数
        [levels, criLevel] = self.ndSort(self.problem.maxormins *
            population.ObjV, NIND, None, population.CV) #
            对NIND个个体进行非支配分层
        population.FitnV[:, 0] = 1 / levels #
            直接根据levels来计算初代个体的适应度
        #===========================开始进化============================
        while self.terminated(population) == False:
            # 选择个体参与进化
            offspring = population[ea.selecting(self.selFunc,
                population.FitnV, NIND)]
            # 对选出的个体进行进化操作
            offspring.Chrom = self.recOper.do(offspring.Chrom) # 重组
            offspring.Chrom = self.mutOper.do(offspring.Encoding,
                offspring.Chrom, offspring.Field) # 变异
            offspring.Phen = offspring.decoding() # 解码
            self.problem.aimFunc(offspring) # 求进化后个体的目标函数值
            self.evalsNum += offspring.sizes # 更新评价次数
            # 重插入生成新一代种群
            population = self.reinsertion(population, offspring, NIND)
        return self.finishing(population) #
            调用finishing完成后续工作并返回结果
```

该算法模板实现的是经典的 NSGA2 算法。但细心的读者会发现，在保留个体到下一代的操作中，上述进化算法模板会跟原版 NSGA2 算法在代码逻辑上有所不同。原版 NSGA2 算法的个体保留方法如下：

step1: 设父代种群和子代种群的个体数目都为 $Nind$，则在父代和子代个体合并后，对合并的种群进行非支配分层，同时找出位于临界层的个体，处于临界层及后面的个体则不需要继续进行非支配分层。

step2: 处于临界层之前的个体（总是不大于 $Nind$）将会被直接保留到下一代，而处于临界层的个体则计算其拥挤距离，根据拥挤距离从大到小的顺序选择若干个个体保留到下一代，直至新一代种群的个体数目为 $Nind$。

如果用图来表示则为：

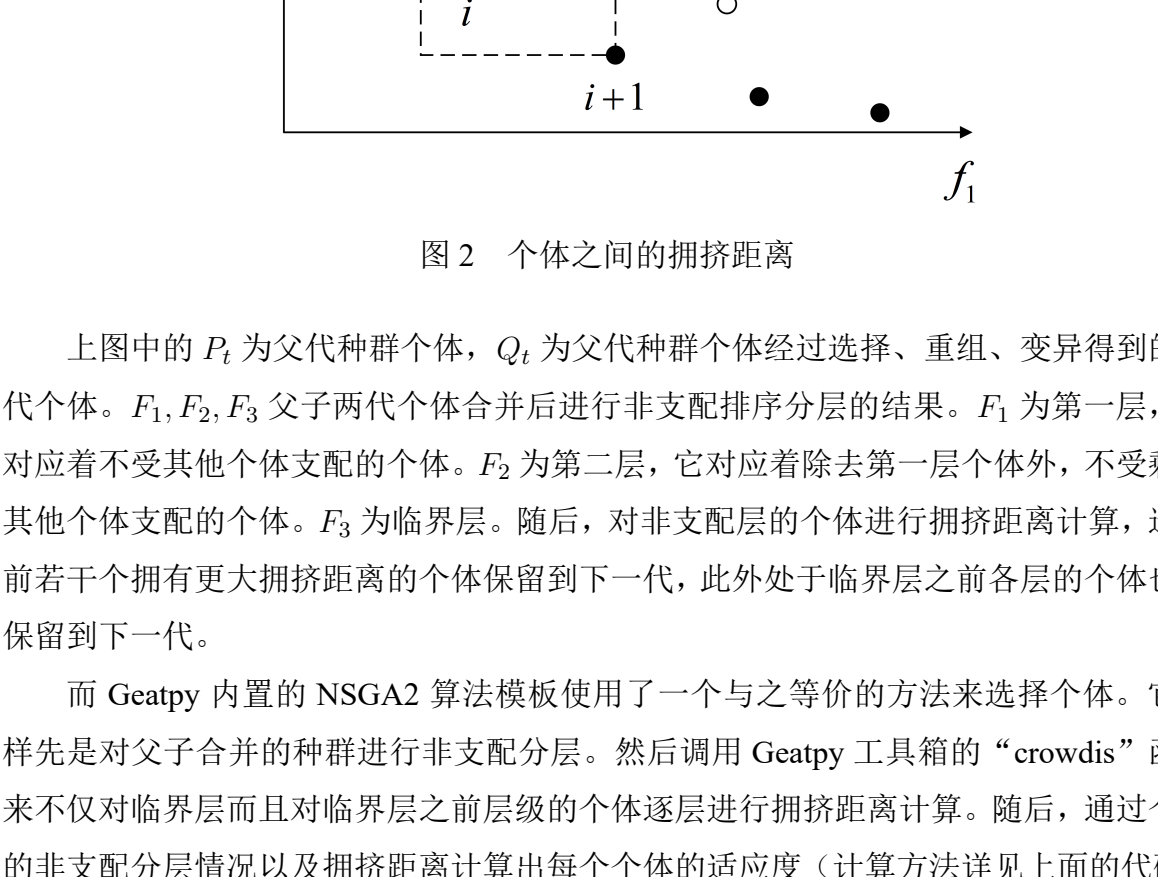


图 1 NSGA-II 新一代种群生成过程示意图

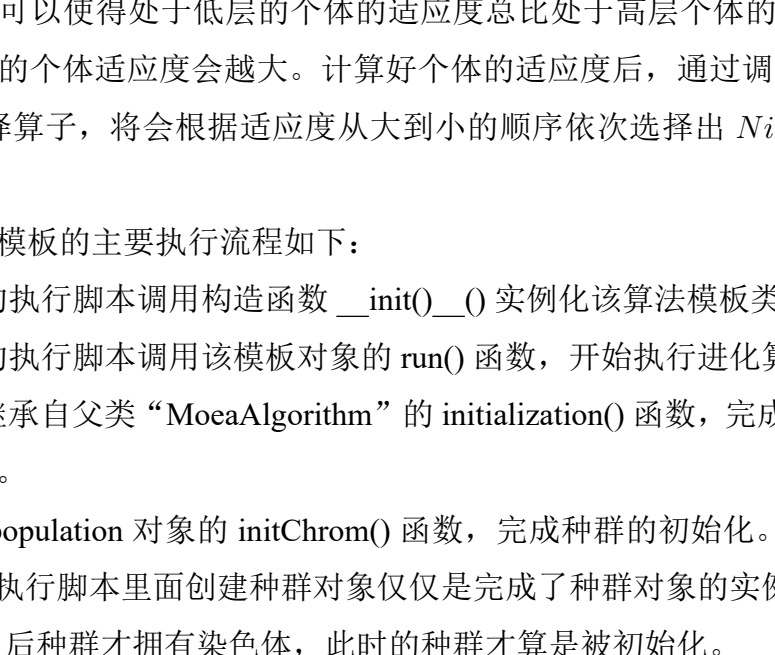


图 2 个体之间的拥挤距离

上图中的 P_t 为父代种群个体，Q_t 为父代种群个体经过选择、重组、变异得到的子代个体。F_1, F_2, F_3 父子两代个体合并后进行非支配排序分层的结果。F_1 为第一层，它对应着不受其他个体支配的个体。F_2 为第二层，它对应着除去第一层个体外，不受剩余其他个体支配的个体。F_3 为临界层。随后，对非支配层的个体进行拥挤距离计算，选出前若干个拥有更大拥挤距离的个体保留到下一代，此外处于临界层之前各层的个体也被保留到下一代。

而 Geatpy 内置的 NSGA2 算法模板使用了一个与之等价的方法来选择个体。它同样先是对父子合并的种群进行非支配分层。然后调用 Geatpy 工具箱的“crowdis”函数来不仅对临界层而且对临界层之前层级的个体逐层进行拥挤距离计算。随后，通过个体的非支配分层情况以及拥挤距离计算出每个个体的适应度（计算方法详见上面的代码）。此时该适应度可以使得处于低层的个体的适应度总比处于高层个体的大，且每一层中，拥挤距离越大的个体适应度会越大。计算好个体的适应度后，通过调用 Geatpy 工具箱的“dup”选择算子，将会根据适应度从大到小的顺序依次选择出 $Nind$ 个个体保留到下一代。

上述算法模板的主要执行流程如下：

(1) 外部的执行脚本调用构造函数 __init()__() 实例化该算法模板类的一个对象。

(2) 外部的执行脚本调用该模板对象的 run() 函数，开始执行进化算法。

(3) 调用继承自父类“MoeaAlgorithm”的 initialization() 函数，完成所需的一些动态参数的初始化。

(4) 调用 population 对象的 initChrom() 函数，完成种群的初始化。在上一章“快速入门”中提到执行脚本里面创建种群对象仅仅是完成了种群对象的实例化，在这里调用了 initChrom() 后种群才拥有染色体，此时的种群才算是被初始化。

(5) 把 population 对象传入 problem 问题对象的目标函数 aimFunc() 中，计算种群个体对应的目标函数值。执行完这一步之后，population.ObjV 便存储好了所有个体的所有目标函数值（详见“Geatpy 数据结构”章节）。如果在目标函数 aimFunc() 中生成了 CV 矩阵 (种群个体违反约束程度矩阵)，则此时 population.CV 便存储好了所有个体的违反各个约束条件的程度。在计算完目标函数值后，要随之更新记录评价次数，即上面代码中的 self.evalsNum = population.sizes。这是因为此时种群的所有个体进行了一次目标函数值的计算，因此评价次数等于种群规模，即 population.sizes。

(6) 至此完成初代种群的相关工作，此时将进入 while() 循环。while 循环将在每次循环之初调用继承自父类“MoeaAlgorithm”的“terminated()”函数，用于判断是否满足进化终止条件，同时对这一代的种群进行统计分析（调用 stat() 函数）。具体代码详见“Algorithm.py”源码，因为“MoeaAlgorithm”类编写在“Algorithm.py”中。如果需要其他自定义的终止条件，可以在完成算法模板的实例化后，对其“terminated()”函数进行重写。

(7) 在此之后便是选择交配个体、重组、变异、重插入生成新一代种群的相关操作，具体可详见上面的代码。

(8) 完成进化后，调用继承自“MoeaAlgorithm”父类的“finishing()”函数完成一些后续工作。这些后续工作往往是，也建议是跟算法模板所实现的 NSGA2 算法核心流程是无关的。这些后续工作包括调用非支配排序筛选出种群中的非支配个体、根据种群的 CV 矩阵彻底排除结果里面的非可行解、更新用时记录、绘图等等。最后得到的种群命名为 NDSet，意味着这是算法得到的非支配集，它跟 population 一样也是 Population 类型。此时外部的执行脚本将获得这一返回值，并进行后续的一些工作，如结果的指标分析、进化过程的指标追踪分析等等（详见上一章的执行脚本代码部分）。

3. 其他算法模板

上面详细讲解了 Geatpy 内置的多目标优化 NSGA-II 进化算法模板，实际上，Geatpy 内置的其他算法模板都与之类似，不同的是算法的核心流程，其他的辅助性工作 (如绘图、分析记录等) 都是基本一样的。可分别打开算法模板的源码文件进行分析研究，每个算法模板都有详尽的注释。

4. 约束条件的处理

在上一章“Geatpy 快速入门”中讲述了 Geatpy 处理约束条件的两种方法：罚函数法和可行性法则。其中罚函数法是跟进化算法模板没有任何关系的，它在目标函数中就已经对非可行解个体的目标函数值进行了惩罚。这里重点讲解可行性法则下如何生成合适的 CV 矩阵（即种群个体违反约束程度矩阵）。

在 CV 矩阵中，每一行对应种群的一个个体，每一列对应一个约束条件，元素大于 0 表示违反约束条件，反之表示满足约束条件。当没有设置约束条件时，种群类对象在实例化时会自动生成一个具有一列而且元素全为 0 的列向量，此时表示种群所有个体都是可行解。

对于不同表现形式的约束条件，需要进行适当的转换才能生成正确的 CV 矩阵。主要包含 4 种类型：

(1) $\geq$ 型：例如 $x_1 + x_2 \geq 3$。此时可将不等式左边移到右边即可得到 CV 矩阵：

$$CV_1 = 3 - x_1 - x_2$$

这样就能够确保 CV_1 中大于 0 的对应的是非可行解。

需要注意的是：这里的 CV_1 需要确保是只有一列的列向量。

此外，$\leq$ 型可以转化为 $\geq$ 型，用同样的方法得到 CV_1。

(2) $>$ 型：例如 $x_1 - 2x_2 > 1$。此时单是把不等式左边移到右边，并不能得到满足条件的 CV 矩阵。如：

$$CV_2 = 1 - x_1 + 2x_2$$

这是因为它将会误把 $1 - x_1 + 2x_2 = 0$ 的误认为是可行解。此时需要额外添加代码来处理：

```python
import numpy as np
CV2[np.where(1-x1+2*x2==0)[0]] = 1
```

这里把算式等于 0 对应位置的 CV_2 的值设为 1，实际上可以设任意大于 0 的值。

这样就能够确保 CV_2 中大于 0 的对应的是非可行解。

此外，$<$ 型可以转化为 $>$ 型，用同样的方法得到 CV_2。

(3) $\neq$ 型：例如 $x_3 \neq 2$。此时可以先生成一个全为 0 的列向量，然后用类似 (2) 中的方法对等式的情况进行标记即可，代码如下：

```python
import numpy as np
CV3 = np.zeros((N, 1)) # 设N为种群个体数目
CV3[np.where(x3 == 2)[0]] = 1 # 标记为任意大于0的值均可
```

(4) $=$ 型：例如 $x_1 + x_2 + x_3 = 1$。这种类型便是常见的等式约束。可把等式右边移到左边，然后取绝对值来生成对应的 CV：

$$CV_4 = |x_1 + x_2 + x_3 - 1|$$

特别注意：上面均是对整个种群而言的，意味着 x_1, x_2, x_3 都是整个种群所有个体的 x_1, x_2, x_3，而不是单个个体的 x_1, x_2, x_3。如果要分开对每个个体生成 CV_i(i 为个体序号)，则此时必须所有个体都生成对应的 CV_i，且此时的 CV_i 是一个一行多列的向量，每个元素对应一个约束条件。最后需要通过拼接的方式生成符合 Geatpy 数据结构的 CV 矩阵。

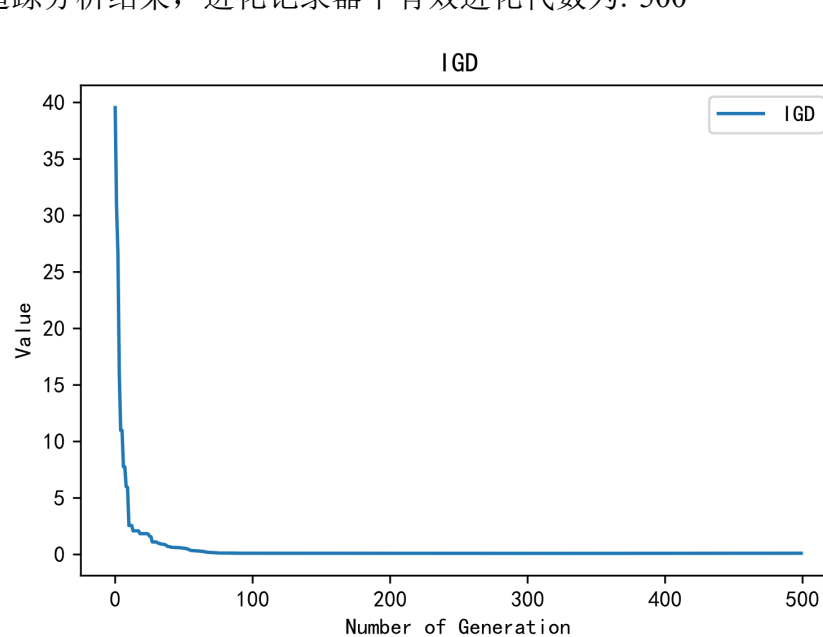
5. “遗忘策略”

Geatpy 的单目标优化和多目标优化都额外引入了“遗忘策略”(详见“Algorithm.py”的源码)，作用是通过种群的 CV 矩阵判断当代种群是否拥有满足约束条件的个体，如果一个都没有，那么进化记录器将不对这一代种群进行记录，同时进化代数-1，这意味着如果以“最大进化代数”作为进化停止的判断条件，那么此时由于当代种群没有一个个体是满足约束条件的，因此后面需要多进化一代来作为弥补。如果一直进化都没有出现满足约束条件的个体，达到所设定的上限时，进化同样会停止。

6. 使用算法模板求解问题

在上一章“快速入门”里讲述了如何使用内置的算法模板求解一个带约束的单目标优化问题和一个带约束的双目标优化问题。在 Geatpy 目录下的“demo”和“testbed”文件夹下均有大量案例，这些案例都是调用 Geatpy 内置的进化算法模板来求解问题的，涵盖了单目标优化、多目标优化、组合优化、约束优化等。代码均有详尽注释，可边运行边结合代码进行深入研究。

其中“testbed”是 Geatpy 的进化优化实验平台，里面实现了很多单目标和多目标的测试函数。以多目标优化“DTLZ-1”测试函数为例，“testbed/moea_test/moea_test_DTLZ/”文件夹中的“moea_test_DTLZ.py”是“DTLZ”系列优化测试的执行脚本。它提供目标维数“M”的设置，这里设置 M=10，种群规模 NIND=275，设置调用“moea_RVEA_templet”算法模板，最大进化代数为 500，执行“moea_test_DTLZ.py”脚本，结果如下：



种群信息导出完毕。

用时：3.130703 秒

评价次数：137500 次

非支配个体数：275 个

单位时间找到帕累托前沿点个数：87 个

GD 0.0012735503409334154

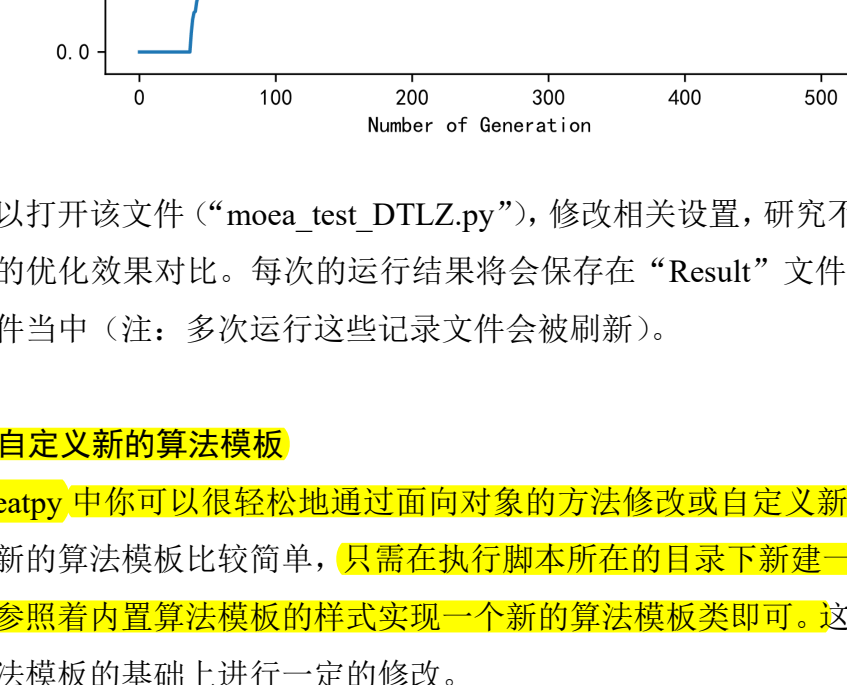
IGD 0.10778142268201366

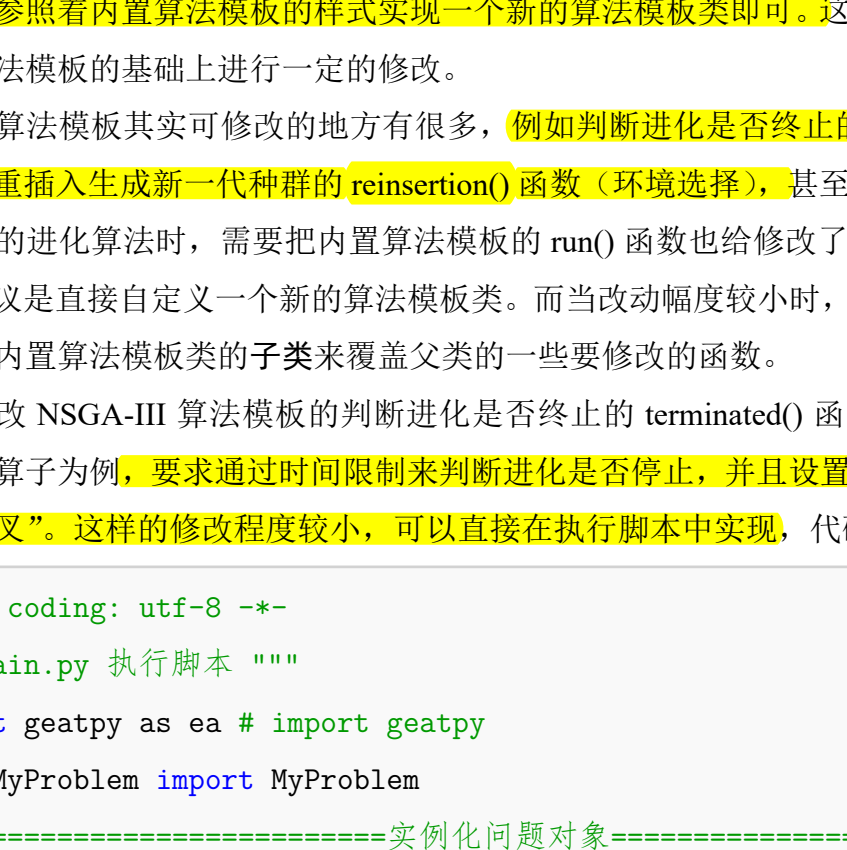
HV 1.0

Spacing 0.02819853408975837

正在进行进化追踪指标分析，请稍后......

指标追踪分析结束，进化记录器中有效进化代数为: 500





你可以打开该文件（“moea_test_DTLZ.py”），修改相关设置，研究不同算法、不同参数设置下的优化效果对比。每次的运行结果将会保存在“Result”文件夹下面的“.csv”后缀的文件当中（注：多次运行这些记录文件会被刷新）。

7. 修改或自定义新的算法模板

在 Geatpy 中你可以很轻松地通过面向对象的方法修改或自定义新的算法模板。由于自定义新的算法模板比较简单，只需在执行脚本所在的目录下新建一个文件，然后在该文件中参照着内置算法模板的样式实现一个新的算法模板类即可。这里重点讲解如何在内置算法模板的基础上进行一定的修改。

内置算法模板其实可修改的地方有很多，例如判断进化是否终止的 terminated() 函数、进行重插入生成新一代种群的 reinsertion() 函数（环境选择），甚至说当如果需要实现自适应的进化算法时，需要把内置算法模板的 run() 函数也给修改了。如果改动幅度较大，建议是直接自定义一个新的算法模板类。而当改动幅度较小时，可以通过定义一个继承该内置算法模板类的**子类**来覆盖父类的一些要修改的函数。

以修改 NSGA-III 算法模板的判断进化是否终止的 terminated() 函数以及修改其调用的重组算子为例，要求通过时间限制来判断进化是否停止，并且设置重组算子为“正态分布交叉”。这样的修改程度较小，可以直接在执行脚本中实现，代码如下：

```python
# -*- coding: utf-8 -*-
""" main.py 执行脚本 """
import geatpy as ea # import geatpy
from MyProblem import MyProblem
"""=========================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
NIND = 100             # 种群规模
Field = ea.crtfld(Encoding, problem.varTypes, problem.ranges,
    problem.borders) # 创建区域描述器
population = ea.Population(Encoding, Field, NIND) #
    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========================"""
class My_moea_NSGA3_templet(ea.moea_NSGA3_templet):
    def terminated(self, pop): # 判断是终止进化，pop为当代种群对象
        self.stat(pop) # 进行统计分析，更新进化记录器
        if self.passTime >= self.MAXTIME: # 增加时间限制
            return True
        if self.currentGen + 1 >= self.MAXGEN or self.forgetCount >=
            self.maxForgetCount:
            return True
        else:
            self.currentGen += 1 # 进化代数+1
            return False
myAlgorithm = My_moea_NSGA3_templet(problem, population) #
    实例化一个算法模板对象
myAlgorithm.MAXTIME = 0.2 # 限时0.2秒
myAlgorithm.MAXGEN = 500 # 最大进化代数
if population.Encoding == 'RI':
    myAlgorithm.recOper = ea.Recndx(XOVR = 1) # 生成正态分布交叉算子对象
myAlgorithm.drawing = 1 #
    设置绘图方式（0：不绘图；1：绘制结果图；2：绘制过程动画）
"""===================调用算法模板进行种群进化===================
调用run执行算法模板，得到帕累托最优解集NDSet。
NDSet是一个种群类Population的对象。
NDSet.ObjV为最优解个体的目标函数值；NDSet.Phen为对应的决策变量值。
详见Population.py中关于种群类的定义。
"""
NDSet = myAlgorithm.run() # 执行算法模板，得到非支配种群
NDSet.save()          # 把结果保存到文件中
```

在上面的代码里，通过实现一个继承“moea_NSGA3_templet”类来实现一个新的算法模板类“My_moea_NSGA3_templet”，来修改 terminated() 函数，在当中加入了通过判断 passTime 是否达到 MAXTIME 来限制进化的耗时。除此之外进化算法的其他代码不用再进行编写，调用的时候调用的是其父类的代码。对于重组算子，上面的代码中在实例化了算法模板对象后通过设置“recOper”为“ea.Recndx”来完成重组算子的修改（NSGA-III 算法模板在种群染色体编码为’RI’ 时默认的重组算子是模拟二进制交叉“Recsbx”，详见“moea_NSGA3_templet.py”）。

利用这种定义子类的方式来修改算法模板，可以在不改动原有的算法模板代码的基础上很方便地实现新增的功能，从而避免不慎把内置的代码给改坏了，而且代码量也比较小。当然地，如果改动幅度较大，实际上建议新建一个文件来自定义一个改进的算法模板类。