# 进化计算与交互式进化计算设计书

记录人：郝国生

代码开始时间：2005年8月

设计书开始时间：2020年6月14日

目录

[进化计算与交互式进化计算设计书 1](#_Toc94694430)

[1 引言 2](#_Toc94694431)

[2 结构 11](#_Toc94694432)

[遍历进化计算实现 11](#_Toc94694433)

[环境营养能力的定义 12](#_Toc94694434)

[可视化及其实现 13](#_Toc94694435)

[关于Knowledge 13](#_Toc94694436)

[增加knowledge原因—早熟收敛 13](#_Toc94694437)

[Knowledge情况 14](#_Toc94694438)

[最优个体保留 14](#_Toc94694439)

[最优个体池 15](#_Toc94694440)

[最优个体保留的方法 15](#_Toc94694441)

[最差个体集合 15](#_Toc94694442)

[最优个体与最差个体求取时间 15](#_Toc94694443)

[想实现 16](#_Toc94694444)

[总找不到最优解，想要找到它。 16](#_Toc94694445)

[想看看适应值的变化情况 16](#_Toc94694446)

[想增加一个新的测试函数 16](#_Toc94694447)

[关于Java的克隆 16](#_Toc94694448)

[运行时配置文件 27](#_Toc94694449)

[实验数据 28](#_Toc94694450)

[数据内容 28](#_Toc94694451)

[实验数据存储 28](#_Toc94694452)

[表结构 28](#_Toc94694453)

[实验数据处理 29](#_Toc94694454)

[处理的简捷性表达 29](#_Toc94694455)

[测试函数 32](#_Toc94694456)

[加入单目标测试函数3步骤 37](#_Toc94694457)

[搜索空间 38](#_Toc94694458)

[进行grid化处理 38](#_Toc94694459)

[2020-03-15 43](#_Toc94694460)

[排斥距离 43](#_Toc94694461)

[2020-8-15 44](#_Toc94694462)

[2020-8-16 44](#_Toc94694463)

[2020-08-18 47](#_Toc94694464)

[在一个类中加入一个属性 47](#_Toc94694465)

[在Individual中加入属性，实现BirdEey 47](#_Toc94694466)

[竟争交叉与变异 48](#_Toc94694467)

[全局唯一性对象 48](#_Toc94694468)

[Exploration和Exploitation的比率参数 49](#_Toc94694469)

[2020-08-19 49](#_Toc94694470)

[种群按适应值排序 49](#_Toc94694471)

[关于三角选择算子 50](#_Toc94694472)

[关于交叉点个数 50](#_Toc94694473)

[关于实数与二进制编码的变异 50](#_Toc94694474)

[决策变量与个体 51](#_Toc94694475)

[2020-08-20 51](#_Toc94694476)

[关于Exclusion operator的再认识 51](#_Toc94694477)

[Population中TreeSet 51](#_Toc94694478)

[202-08-21 52](#_Toc94694479)

[可以被调节的参数 52](#_Toc94694480)

[测试 52](#_Toc94694481)

[2020-08-22 53](#_Toc94694482)

[构造函数中完成初始化 53](#_Toc94694483)

[新个体的生成 53](#_Toc94694484)

[基于淘汰半径的方法 53](#_Toc94694485)

[基于BirdEye的方法 53](#_Toc94694486)

[2020-08-23 53](#_Toc94694487)

[二进制编码中天然含有BirdEye 53](#_Toc94694488)

[2020-08-24 54](#_Toc94694489)

[EA的核心问题 54](#_Toc94694490)

[Exclusion同时考虑收敛性 54](#_Toc94694491)

[实验出问题的特征 54](#_Toc94694492)

[2020-08-25 54](#_Toc94694493)

[一段对话Oracle和Algorithm 54](#_Toc94694494)

[进化状态 56](#_Toc94694495)

[现状不变指数1--适应值f(x)没有太大提高 56](#_Toc94694496)

[多样性与收敛性指数1 56](#_Toc94694497)

[2020-08-28 57](#_Toc94694498)

[重写遗传算法的部分代码 57](#_Toc94694499)

[2020-09-05 57](#_Toc94694500)

[在0到n内生成n个不重复的随机数 57](#_Toc94694501)

[Parent过滤 58](#_Toc94694502)

[2020-09-07 58](#_Toc94694503)

[HistoryIndividuals 58](#_Toc94694504)

[不能保存进化过程中的所有个体信息 58](#_Toc94694505)

[2020-09-08 58](#_Toc94694506)

[ParentFilter先get再put 58](#_Toc94694507)

[2020-09-09 59](#_Toc94694508)

[在offspringFilter之后，再求适应值 59](#_Toc94694509)

[替换排斥个体 59](#_Toc94694510)

[2020-09-11 59](#_Toc94694511)

[利用TreeSet的上界与下界功能 59](#_Toc94694512)

[2020-09-12 60](#_Toc94694513)

[聚类--利用TreeSet进行处理 60](#_Toc94694514)

[2020-09-14 60](#_Toc94694515)

[种群多样性的指标 60](#_Toc94694516)

[2020-09-15 61](#_Toc94694517)

[调节diversity使用父代个体间的排斥距离 61](#_Toc94694518)

[编码方法对变异功能的影响 61](#_Toc94694519)

[2020-09-16 62](#_Toc94694520)

[Exploration和Exploitation的切换 62](#_Toc94694521)

[2020-09-16 62](#_Toc94694522)

[续：Exploration和Exploitation的切换 62](#_Toc94694523)

[2020-09-17 63](#_Toc94694524)

[续：Exploration和Exploitation的切换 63](#_Toc94694525)

[编码 63](#_Toc94694526)

[2020-09-26 63](#_Toc94694527)

[进化计算参数 63](#_Toc94694528)

[202-11-19 63](#_Toc94694529)

[种群初始化{杨启文, 2020 #3567} 63](#_Toc94694530)

[2021-03-22 64](#_Toc94694531)

## 

## 1 引言

### 1.1 已有平台

已经有的进化计算平台，包括jMetal等，但发现改造jMetal比较困难，就自己动手搭一个。

### 1.2 系统结构

参考了jMetal，系统主要包括两个主体：算法Algorithm、问题Problems。

#### 关于通信

需要的参数或资料等一般可以在相应的功能包中找到，当多个包共同使用时，则可以从ControlParameters中获得，这里提供了一个类似于黑板的机制。

#### 类簇及关系

Individual类与问题挂钩，不同的问题具有不同的Individual。在实际运行时，给出Individual的类型。

### 1.3 本系统的优点

开源：https://gitee.com/guoshenghao/Interactive-and-Evolutionary-Algorithms/tree/master

Robustness：每k代就有一个备份，因为一些复杂的高维问题耗时。正因为有了备份功能，才在可能断电、死机、outof memory的时候，有保存的结果，可以直接在原来的基础上恢复现场继续运行。利用了数据库，数据库能高效保存现场数据，并方便恢复数据。

Easy use：易用性体现在导出数据直接在Excel中；体现在种群规模参数可以一次设置多个；而不用每次设置一个；实验完成后，再重新设置下一个；在配置较低的机器上也可以运行。

高效：不允许在同一种群中有重复的个体出现。不允许被访问过的个体再被访问机制也加入了进来。

可以指定算法运行次数，然后记录运行过程中最优个体变化，以及运行时间等。

实验结果可以选择保存到Excel文件中，也可以保存到数据库中。

### 1.4 缺点

速度慢，原因1，采用历史个体不重复出现的机制；原因2，数据库的使用，相当于从外存调用数据进入内存，会受数据库速度和外存速度影响；

**函数式编程的缺点**。虽然优点比较明显，即不对传入的对象做修改，但是缺点也明显，那就是，如果返回的结果是一个对象的话，那么每次调用函数都会新增对象，因此，当迭代次数很多时，就会增加内存，容易产生out of memory的错误。

正是考虑到上述缺点，本系统的选择、变异等操作不采用函数式编程，即直接对传入的对象做改变。而对于交叉操作，则由于parent是随机选择的，则采用函数式编程，而不对原parent进行改变。

### 1.5 一些约定

在构造函数中完成初始化工作。

* **求最大值方法。**本系统采用求取最大值的方法。因此，如果要利用本系统求取最小值时，需要转换换为求最大值的问题

## 2 算法

### EA的核心问题

核心问题是解的生成问题。

生成新的解的依据是{x,f(x)}这一带标注的数据集

### 流程

算法主要的流程包括：初始化、evaluate、排序、求最优解、选择、交叉、变异、用最优解替换一个随机解、再evaluate，接排序等直到达到算法结束条件。

### 交叉算子

### 选择算子

* 选择算子没有基于排序。

### 变异算子

#### 面向exploration和exploitation的变异

当采用二进制编码时，变异位置的不同，功能不同。在低位变异，侧重于exploitation，在高位变异，侧重于exploration。

当采用实数编码时，为了保持变异这种既能explore，也能exploit的功能，让实数编码的变异位也采用在对应维的区间内任意取值的功能，即用一个区间内的任意值替换它。当新值接近于原值时，相当于exploit；当远离原值，相当于explore。

鉴于上述原因，将类中的下述代码作了如下修改：

修改前：

|  |  |
| --- | --- |
| 1  2 | double tem = rand < FactoryAlgorithm.rationOfExploration ? 1 : rand;  temDoubleArray[muteLocation] = this.getGeneCodes()[muteLocation] +  tem \* FactoryProblems.currentProblem.getVariableProperties()[muteLocation][2]  \* mutateStep; |

修改后：

|  |  |
| --- | --- |
| 1  2  3 | double tem = rand < FactoryAlgorithm.rationOfExploration  ? rand\*(FactoryProblems.currentProblem.getVariableProperties()[0][0]-FactoryProblems.currentProblem.getVariableProperties()[0][1])  : rand;  temDoubleArray[muteLocation] = this.getGeneCodes()[muteLocation]  + tem\* mutateStep; |

* 自适应调整变异概率。当种群的适应值较上一代下降时，会自动增加变异概率和增加变异步长（对于float类型的变量来说）。
* 根据规定的范围，变异会自动判断解的合法性。交叉算子不用判断其合法性，因为交叉算子不会越界。

### 种群

* 种群内不允许出现重复个体，无论是否采用排除机制，这一条始终起作用。

#### 种群初始化

种群初始化方法一般有两种情形{杨启文}：a)随机初始化；b)均匀初始化。

随机初始化方法简单，但会导致候选解在解空间的有些区域分布“密集”，有些区域分布“稀疏”。一旦最优解区域无法得到有效采样，则会在一定程度上延长最优解的发现时间。

均匀初始化方法有三种：正交实验法{蔡自兴, 2010 #3564}、随机均匀设计法{陈明华, 2010 #3565}和佳点集法{龙文, 2012 #3566}。这三种方法都能对解空间进行均匀采样，但正交实验法在高维空间内需要较多的采样点{张铃, 2001 #3568}，因此，正交实验法更适合应用于低维问题的优化方案中。随机均匀设计法与佳点集法的效果相差不大。故本文采用佳点集法进行种群初始化。

下图是随机初始化方法和佳点集初始化方法分别在边长为1的二维正方形区域内100个采样点的分布情况。对比可以看出，佳点集产生的点在同样的正方形区域内分布更加均匀，这对于增加候选解的多样性、防止“早熟收敛”有着重要的意义。

（a）随机初始化 （b）佳点集初始化

### 一些约定

* 当一个GSU或Scheme或individual经过一定次数的迭代后，尤其是进行了较多的exploit后，适应值没有提高，exploitation或exploration应向对立面转换。
* 个体太聚集，则要散开些。聚久当散。个体太分散，则要聚集些。散久当聚。
* 大种群有利于多样性，小种群不利用多样性
* 变异算子：对二进制的高位变异有利于多样性，对低位变异有利于收敛性
* 交叉算子：距离远的个体交叉有利于多样性，距离近的个体交叉有利于收敛性
* 选择算子：有助于收敛
* 社会学习：向pbest学习有助于多样性，向gbest学习有助于收敛性；不同拓扑学习结构，多样性机理不一样
* 粒子相互作用：排斥有助于多样性，吸引有助于收敛性
* Bird-Eye view：飞得高有助于多样性，飞得低有助于收敛性
* 个体的适应值求取或评价，尤其当目标较多，或决策变量维数过高时，会耗费大量的时间，因此，避免重复或无效的适应值评价，而是在offspringFilter之后，再求适应值。
* 当个体落在排斥范围之内时，替换它。替换的方法是当随机数小于FactoryAlgorithm.rationOfExploration时，与最优个体交叉；反之，对当前个体进行变异。
* 任何静态的参数，即在EA运行过程中保持固定的值，都是不合适的[9]。那么一个功能强大的平台，应该能够自适应调整所有参数，包括种群规模、交叉概率、变异概率，交叉与变异竞争概率、选择算子的类型等。

### 全局唯一性对象

在本系统中，如果某对象是全局唯一性的，则不需要作为参数传递，可以直接作为静态对象获取。

满足这些条件的对象包括：

* 具体的算法。在实际运行过程中，不会让多个算法同时执行。如遗传算法与粒子群算法不会同时作为两个算法执行，即只会有一个算法在一个时刻执行，因此这是全局唯一性对象。
* 具体的问题。在实际运行过程中，不会让算法去同时优化多个问题，只会针对一个问题进行优化，如果涉及到多个问题，则分别在独立优化，而不会出现在同一个JVM中。因此，这也是全局唯一性对象。
* 与算法伴随的具体策略，如PopulationFilter。

因此，当其他对象或方法需要具体的算法或具体的问题时，则可以直接从静态方法中获取，而不必传递参数。

在使用这种对象时，需要注意的是：在每次重新开始实验后，要对其成员进行初始化。如果不初始化，则上一次实验的内容会影响本次实验。

### Exploration和Exploitation

#### 比率参数

在类FactoryAlgorithm中添加了一个属性rationOfExploration ，作为exploration的比率。

作用：在Exploration和Exploitation之间进行切换，如在进化初期用0.9或在进化后期用0.2，又如当算法在0.1下长期运行没有进展时，切换到较高值。

用处：主要用在变异中。如在DoubleIndividual类的方法mutate()中。

#### Exploration和Exploitation的切换

切换所在的主体或层次：算法主体？种群层次？个体层次？

无论放在算法主体或种群层次上，最后的实现都是通过个体来实现的。因此合理的做法是放在个体层次上。

而个体到底是采用Exploration或Exploitation，可以由其在种群中的地位来确定。其在种群中的地位则由适应值的对比来确定，即用个体之间的占优关系来确定。

采用Exploration或Exploitation的操作，由变异步长来操作。

GA最初采用二进制编码方法时，对于Exploration和Exploitation的切换，较为直观：在低位变异是exploitation，在高位变异，是exploration。当然，采用8或16进制编码时，也有类似的效果。比起采用实数编码直观。

因此，变异步长，可以结合个体编码来实现。例如，采用二进制编码。

那么，有一个问题是：二进制编码长度有限，对于连续的搜索空间如何优化？我们可以考虑一种多阶段缩小搜索空间优化的方法。在多个阶段中，每个阶段可以重复实验k次，取其中最好的结果，然后把搜索空间缩小到最优解附近，继续开始下一阶段优化；每进入下一阶段，搜索空间的精度就提高一次，这样，就可以一直逼近最优解。当然，在实际的工程优化中，这个精度要受限于技术水平。例如，当前半导体的最小精度是纳米级的。 而更高精度的技术还有待进一步发展。

变异步长的确定的另一个方法来自于[8]。该文变异步长是一个正态分布，均值为0，但方差却是各维上下界差值最大值(maxi(*xiU*-*xiL*))的1%~5%。也就是说，可以调整1%~5%的值，使得变异成为exploration或exploitation

<https://www.cnblogs.com/bianqi/p/12183952.html>

给出BSP树及KD树的Java代码实现

主讲老师网页:<https://github.com/shengxinjing>

### 种群按适应值排序

为种群内个体指定进行exploration或exploitation的依据之一是其适应值，如果适应值大，则进行exploitation，反之则进行exploration。或者说，如果这只鸟找到了好的食物，就舍不得离开，继续找；如果找到的食物不多，则要飞起来继续找。

为实现上述操作，就需要按适应值进行排序。如果自己使用排序方法进行处理，也可以；但如果语言本身提供了排序的数据结构，就可以直接拿来使用。Java提供了TreeSet，作为一种排序的数据结构，因此，Population类可以直接使用TreeSet来维护Individual。

TreeSet有两个重要的特点：一个是Tree，另一个是Set。Tree是一个有序树，如果是二叉树，则左子树上的所有结点的值都比右子树上所有结点的值小。Set是集合，其中的个体无重复！！这意味着，Population中自动成为无重复的个体。自动实现了NoReappearOntheSamePopulation的功能。当然这个set借助Tree的功能，成为有序的了。

借助TreeSet上面的特点，Population可以避免实现Sort方法，可以避免实现checkReappearance。而历史个体也可以借助其适应值的Compare方法，实现有序无重复存储。

但是需要注意的是：在实现compareTo时，如果只用适应值进行对比，那么对于非单调的优化问题，很多解会被认为相等；为了避免这种情况，在compareTo方法中加上决策变量的对比；即决策变量与目标变量都相等，才算相等。

当然，也想利用TreeSet实现另一个目标：存储搜索空间上的个体，观察其收敛性与多样性。为了实现这个目标，需要单独定义一个类，在遗传算法中是基因编码，在PSO中是多维实数变量。

但是，选择算子后得到的交叉与变异候选池则需要有重复的，因此，在选择算子中，还需要用List。

TreeSet默认是升序，有pollFirst方法可以被调用把最差个体移除；有poolLast方法把最优个体移除。

使用TreeSet保存Individual，带来的一个好处是，实现了同一种群内的Exclusion，即NoRepeatOnSmaePopulationKnowledge这种策略代替了SimpleGA。

### 可以被调节的参数

* GeneticAlgorithm类中的方法operatePopulation中利用了ControlParameters.birdLevelNumber，可调
* RegularMutation类中的方法getMutationResult中使用了FactoryMutation.mutationPointNum，可调
* RegularMutation类中的方法mutate中使用了类Individual中的Explo\_ra\_itationFact，可调

### 关于三角选择算子

删除了cos和ctan选择算子，因为sin和tan具备了相应功能了。

### 关于交叉点个数

一点交叉和多点交叉，都是生成新个体的方法。虽然多点交叉似乎由于交叉点个数增多，会增加解的多样性。但搜索空间中，多样性的度量中所依赖的解的不同性的概念，并不依赖于有多么不同，即只要两个解编码不同，就认为不同，因此在计算多样性时贡献相同。另外，在搜索空间中，解的分布性也并不会因为交叉点个数的增加而增加。所以结论是：多点交叉与一点交叉没有本质的区别，因此，删除了多点交叉。

在上述结论基础上，算法的exploitation和exploration不能作用于交叉算子上，只能依赖变异算子。

### 排除算子

#### 算子流程

Exclusion mechanism--排除机制,作为一个算子，可以加入到任何算法中，因此，在Algorithm接口中有对应的方法。对应的流程如图1 排除算子。

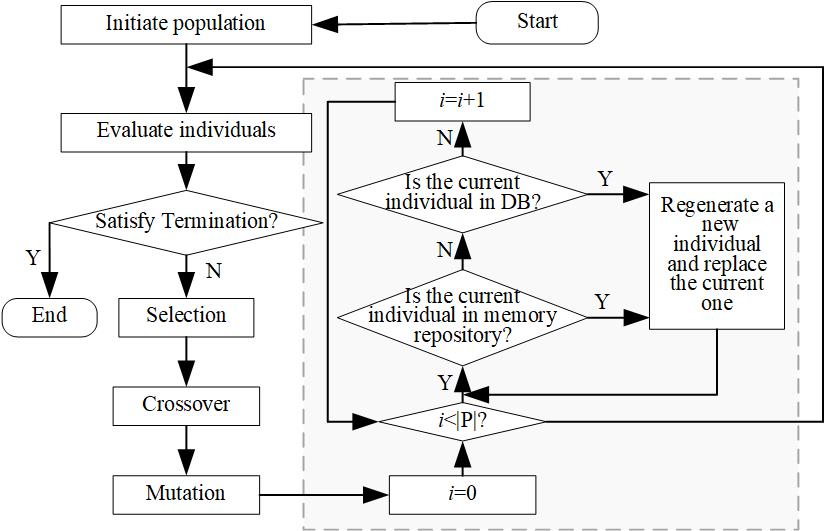


图1 排除算子

##### 流程图中的位置

排除机制检查只放在种群要进行evaluation之前，不参与选择、交叉和变异操作。

排除机制只检查其他个体，不检查最优个体；最优个体替换最后一个个体，因此，最后一个个体不被检查。

#### 补充被排除个体

补充被排除个性，主要是通过生成新个体进行补充。

* 第一种：从最优解生成；会快速收敛，但exploration不好,但exploitation比较好
* 第二种：从当前解生成；exploration仍不好，但exploitation比较好
* 第三种：随机生成；exploration比较好，但exploitation不太好

为此，选择第二和三种办法，当有种子个体时，则利用种子个体，但只给一次机会；当利用种子个体仍然得不到未被访问的解时；则使用第三种方法。

如果需要补充新个体，则把种群中未出现的个体做为种子进行补充。这一点以及很多点都要求population的改变只是其中的individual的改变，而population本身从来不改变。

对于搜索空间的格划分，每个格都有一个属性，标示其对应的适应值。如果未被访问，适应值未知，如果已被访问，则用一个最高或平均适应值作为格的适应值。该适应值可以被用来聚类或作为分配搜索资源的依据。

#### 利用数据库记录排除个体

保存内容包括：solution的编码和对应的特征。每当插入一条记录时，trigger计算其特征征。

该特征主要用于检索。首先利用特征进行检索，如果没有相同特征的记录，则表明记录不存在；反之，进一步检索满足特征的记录。

对于检索，可以采用基于分类的方法。首先找到满足该特征的类，然后再在类中查找。对于高维浮点类型，如10维以上，计算特征、检索特征等带来的成本小于直接检索带来的成本时，就可以利用特征检索。采用分类的方法，还能带来另一个好处是方便进行分布式处理，即不同的类别的数据可以放在不同的机器上。

特征计算举例。以两个特征为例，分别取前解各维取值之和x1+x2+x3+x4+...及差x1-x2+x3-x4+...。

**检索存储方法**。在内存中也保存一定数量的最新历史solution，例如保存5|P|个最近的解。查找时，首先从内存中查找，如果没找到，再到数据库中查找。这样做的理论依据是，生成的新解是最近解的邻居，与刚刚访问的解相近的概率大；这样做的好处是：提高检索的效率；如果能在内存中一次命中，就不必到数据库中查找。

进化计算开始时，读取现场保留表，如果发现有现场保护的内容，则继续旧的进化计算。

历史出现过的解不存入在内存中，因为当问题复杂，如高维问题优化时，容易产生OutOfMemoryError: GC overhead limit exceeded的问题。因而考虑存入数据库的方法，这种方法的流程是：进化开始-连接数据库-写入数据-查询数据-进化结束-关闭数据库连接。

#### 在Searchspace中记录需要排除的个体

每个problem有一个searchspace,每个searchspace有一个visitedNode，用来保存已经被访问过的个体。

如果采用类似于地图的办法的话，可以利用逻辑值标记是否访问过。个体对应的hash位置的值设置为true，表明已经出现过了。这样在生成新个体后，检索其对应位置，如果为false的话，则设置为true；反之，则丢弃新个体。

#### 基于排斥算子的遍历算法

分为两个层次的遍历：种群(Population)层次和搜索空间(SearchSpace)层次。

种群层次：每一代(generation)都是一个种群，因此，一个种群的寿命是一代。当新的一代开始时，开始了一个新的种群，遍历排斥所依赖的list（祖父代）应该clear掉，重新统计。

搜索空间层次：整个进化过程，都共享一个排斥所依赖的list。以前的实现是放在SearchSpace接口中，因此其他子类要实现对应的方法，后来发现，搜索空间是种群的一个扩展，其可以在新的种群开始时，不clear就可以保留下来使用。显然，这种办法比在两类class中（Population和SearchSpace）中都维护要简单得多。

种群层次的排斥与小生境的区别是什么？

P(t+1)个体的选择依赖于与P(t)中个体的距离；当P(t+1)经过P(t)距离筛选完成后，P(t)完成了历史使用，可以clear掉了，此时P(t+1)媳妇熬成婆，需要从historyIndividual列表中移除P(t)。如果再查找的话，显然比较耗时，为此，采用把P(t)放在hashset中的方法，移除时，直接在hashset中查找，找到则移除。这种方法比较快速。 此时需要极其小心地做一件事：确保hashset中的individual与historyIndividual中的个体是对象相等，也就是说，P(t)中的个体在进化操作中没有被改变。

🡪这个较难保证，这是其一，其二时从linkedlist中删除个体，效率较低；因为要从头指针查找。🡪采用List<List<Individual>>的方法，其长度决定了禁忌表的长度；当其中元素个数为2时，分别表示保存P(t)与P(t+1)；在奇数代，用第1个元素作为当前P(t)，在偶数代，用第0个元素作为当前P(t)；查找是否排斥时，只在当前P(t)中查。在检查是否排斥时，由于不断插入P(t+1)中的个体，这个链表已经被“污染”了，所以在后边可以直接清空，而用P(t+1)代替。当其中的元素个数为m时，每一代都把最老的一代替换掉，而这最老的一代，可以用一个整数index指向，其取值为index=index%m，每开始新的一个，都用index++。🡪这个方法也存在缺点，那就是多个list不能保证是排序的。

🡪还得返回到用hashSet的方法，即从hashSet中删除相同的个体，需要保证population中的个体未被修改。

Population类有3个桶，分别盛着上一代种群(lastPopulation，桶1)，当前种群(individuals，桶2)和历史种群(historyIndivdiuals，桶3，是有序的)。程序的流程就是围绕这3个桶展开的。步骤如下：

初始化完成后，桶2准备好了；

然后把桶2内容的引用给桶3一份；

下面开始进化操作：

把桶2内容给桶1一份；

执行选择、交叉和变异，利用桶2内容生成P‘(t+1)，并放入桶2

依据桶3内容对桶2进行修正，执行exclusion operator，此时，桶3和桶2内容发生改变；

桶2中得到了P(t+1)

桶3保存了P(t)和P(t+1)的individual，此时个数应该是2|P|-1，因为最优个体保留，替换了一个；

从桶3中移除P(t)

#### 排斥距离

当个体适应值小时，则其排斥半径较大，否则其排斥半径较小。个体适应值好，则该半径可以小到0，反之，该半径可以大到最大值。固定这个最大值可能会影响算法性能，所以采用了随机的办法，取值在[|P|/2,|P|]。这个值，可以在类algorithm.operator.generator.SolutionGenerator中找到。

排斥半径存放在variableProperties中每一维上的第3个标量（这3个标量分别表示最大、最小以及粒度中“粒度”）。粒度除了有这个用途外，在人脸优化中每一个点也有一个变异范围依赖于此值。

另外，排斥距离，主要在NoReappearOnSearchSpaceKnowledge策略中使用。该策略的优点是可以求取一个点的邻居，缺点是采用折半查找非常耗时。因为很少使用，所以排斥半径也就不再做为本系统的重点了。

当个体适应值较好时，对其多进行exploit；反之，当个体适应值较差时，对其多进行explore。为此设置个体排斥半径，凡是在该半径内的个体，都是被排斥的。

#### Exclusion operator作用于决策空间

Exclusion operator是作用于决策空间，而不是适应值空间。

正是因为这个原因，当使用HashSet数据结构时，其元素必然是Decisionvariable，一定不是Individual，因为Individual的hashCode方法的依据是既包括适应值又包括决策变量，而DecisionVariable的hashCode方法只包括决策变量，运算更准确。

发现即使利用数据库，也不能保存进化过程中的所有个体信息，尤其是面对高维问题时，决策变量采用double类型时，存储长度将变长，即便采用derby中的LONG VARCHAR类型，也容不下。

#### 基于淘汰半径的方法

在类PopulationFilter的方法generate2ndStpe中，但目前未被调用

#### Exclusion同时考虑收敛性

因为Exclusion的存在，x(t)的质量可能不比x(t-1)。

PSO在这方面做得好，因为在PSO中，无论好的解（pbest和gbest）出现时t多么老，只要是好的解，都要在生成新解时参考。

### 最优个体保留

关于保留最优个体的方法，如果采用替换第0个个体的办法，那么有如下一个缺点：由于第0个个体是进化操作后生成的，当采用历史个体不再重复出现的策略时，它已经进入了历史个体，因此它不会被评价，也不会再次出现。如果它是比当前最优解还好的解，那么算法的性能因此会有所损失。

针对这个缺点，我们采用的方法包括两步骤：

步骤1：在变异时少生成一个个体；

步骤2：最优保留时，把最优的个体插入到链表前面。

* 最优个体保留到新的一代的目的是在选择算子中使用，所以在选择前执行，至于从外观指标上来说，则是用于监控算法用的。

#### 最优个体池

采用list数据结构，是一个根据fitness排序的list。第0个是全局最好的。不限制容量。

当有新的最优个体时，采用折半查找的方法，如果没找到，则求距离；如果太近，则保留适应值大的；否则，就插入。

#### 最优个体保留的方法

在AbstractAlgorithm中，表面看起来，会使种群规模增1，从而增加了较差个体的生存压力。但是，由于population采用的数据结构是TreeSet<>，所以当加入的最优个体已经存在于population中的时候，种群规模并不会增1。因为是在Set中不允许出现重复个体。

|  |
| --- |
| public void reserveBestInd(Population population) {  population.getIndividuals().add(HistoryIndividuals.getInstance().getHistory4Parent().last().getIndividual().clone());  } |

最优个体保留有两个地方有：第一个地方是选择算子执行前，第二个地方是进化操作完成后。第二个地方的目的是为了输出，记录每代的最好结果。

#### 最差个体集合

采用hashSet数据结构。不限制容量。

#### 最优个体与最差个体求取时间

由于算法执行不需要对种群排序，所以，在每次求取适应值时，就更新最优个体与最差个体，而不是调用sort方法。

### 算法迭代

controllers.Connection4SetAndRunning类中的方法beginEvolving中while循环中的控制变量是由3个条件决定的：

1）population.getBestIndividual().compareTo(stopFitnessIndividual) >= 0

2）generationNum \* algorithm.getPopulationSize() >= ControlParameters.stopEvaluationNumber

3）genrationNum - fitnessSteps.get(fitnessSteps.size() - 1) > 500 \* population.getIndividuals().size()

### 环境营养能力--适应值的新定义

环境意指被优化的问题，即F(x)。

当x适应值低时，反映出x所在的这个地方领域贫瘠，所以不能营养太多的individual，而当x适应值高时，表示这个领域比较富裕，所以可以营养较多的individual.

采用exclusion operator时，当适应值大时，则可以很近，极端情况是只要两者不相等，就认为不等；而适应值小时，则需要距离很大，因为新的individual会抢夺生存资源，因为贫瘠的环境营养不了太多的individual。

从另一个角度看，一个不好的个体会占据较大的地盘，而一个较好的个体则占据较小的地盘。从而实现搜索空间的不等分分割。

从知识利用的角度看，这也是利用fitness knowledge的一种方法。

再加入邻居梯度信息，向适应值好的方向，能营养individual的数目增加，而向适应值差的方向，则数目减少。从知识利用的角度看，这也是每个individual向邻居学习知识的方法。

### Knowledge问题

尤其对于人脸来说，在PreFaceOptimization中还有一个deltaValue的设置及使用，用户赋予适应值时的正态分布的认知规律问题。

### 种群多样性的指标

一个种群多样性的指标是用种群的超体积除以搜索空间的超体积。

搜索空间的超体积在类Problem的方法getVolume中给出。

刘益平也给出了一个指标，是求理想的多样性和实际多样性的差值。理想的多样性，也在类Problem中，对应的方法是getStdEv。

超体积不能过大，如对于1000维，每维上下界为[-100,100]的问题来说，其体积为2001000，而这个数值在计算机中越界了，因为对于64位机器来说，目前机器能处理的最大双精度数值Double.MAX\_VALUE是1.7976931348623157E308。

因此，为防止超体积过大，求体积时采用的方法不是积的1/D次方，而是1/D次方的积。

这个值作为类Problem的一个属性。

另一个多样性指标是用殴氏距离，不同于超体积，欧氏距离用各维距离的平方和开方，因此出现数值越界的情况会减少。且求和的计算量应该比求积的计算量少些。在测度度量方面，这两种测度应该相等。

### 调节diversity使用父代个体间的排斥距离

当父代个体距离小于排斥距离时，则删除适应值差的个体。当排斥距离较大时，个体间距变大，因此，多样性会变好；反之，当排斥距离较小时，个体间距离会变小。通过调节排斥距离，实现调节父代个体多样性的目的，从而达到调节后代diversity的目的。

对排斥距离的调节是在类AdaptiveTricks中利用方法adapvtiveDistanceBetweenVarialbe来调节的。在这个方法中，指定了排斥距离的上限upthreshold与下限downthreshold。让排斥距离在这个区间来回调节。现在的问题

## 3 Problem

Problem簇类与Individual簇类的联系只有两个方面：Problem只能限定Individual的类型，包括决策变量数组的类型（浮点数或字符串）以及目标空间的个数（单目标优化或多目标优化）。

### 个体编码

#### 编码涉及的类层次

本项目中individual的编码有多种格式，如二进制，浮点型等。对应的population也有这些类型。再往上，对应的search space也有多种类型。

一致性问题：individual初始化时采用二进制，而population却使用浮点类型；同样，对于search space也会存在这种情况，怎么办？如何避免这种不一致性？

考虑到individual是population的元素，而后者又是search space的子类，参考社会工作中，不能越级办事的规律，如果想得到individual，则请求population帮忙，而population是归search space负责管理的，则需要请求search space帮忙；而search space是与problem挂钩或属于后者的，因此，应该请后者帮忙。这样一个problem面对algorithm时，是一个统一的整体。

或者说是一个唯一的”**前台**”，而不是一个”**入口**”。即可以在前台这儿等待后边给办理，而不是进入里边去办理（NTU的人事部门办理的方法）。这也是本项目只能看到两个主要的包的原因，这两上包是problem和algorithm。

二进制编码的另一个优点是不用考虑越界的问题，无论采用什么样的交叉或变异算子。

#### Problem类的编码

每个problem可以有多种编码格式，如二进制是一种万能的编码格式，即可以用来处理实数，也可以用来处理字符串；

但是为每个problem设置多种编码格式，会因此要设置多种种群初始化方法，多种搜索空间检索化方法，这些设置的复杂性和易错性，可能会抵消掉采用多种编码格式带来的炫技好处（除了妶技，好像没有太多的其他好处）

为此，为每个问题只设置一种编码格式，从而在界面上，用户不必选择编码格式，而是在定义问题时（在Controlparameter类中），直接指定编码格式。

#### 编码精度与Bird View

##### 关于Bird View Strategy

其核心思想是：当Bird在高处飞，可以执行exploration，在低处飞可以执行exploitation。

优化既需要exploration，也需要exploitation。为此，我们采用至少两个种群的办法，一个执行exploration，一个执行exploitation。当然也可能采用多种群多层次的情况，不同种群保持的exploration/exploitation的层次不同。好处是能保证种群不会出现完全执行exploration或exploitation的不平衡情况。

不同层次的种群之间的信息传递采用高层向低层传递的模式。每一次迭代飞得高的bird种群就将explore到的前n（n>=1）个好位置发送到下一层次，目的是让下一层次的种群在这个好位置进行exploitation。依此类推，直到最底一层的种群收到上一层的好位置。

采用多层次种群的好处是，不必考虑单种群时，种群的层次或位置，即如果是单种群需要考虑，什么时候往上飞，执行explore；什么时候往下降，执行exploit，即往上飞和往下降的触发条件是什么？往上飞多少或往下降多少，其决定因素是什么？

实现中需要定义飞行在各层次上的bird群的层数以及各层的bird群所在的高度，高度即对应于对搜索空间分割时采用的精度，层次越高，resolution越小，精度越粗，反之，层次越底，resolution越大。种群飞行所在的层次，不同的层次，对应的精度precision或分辨率resolution不同，所以explore和exploit的程度不一样。利用层次数实现了定量地控制exploration和exploitation.

每个bird群（种群）有一个属性flyLevel对应于自己所在的层次。具体的resolution精度值，从BirdEyeStrategy中取，而BirdEyeStrategy则由算法来调节或控制。

##### 基于exclusion机制与Bird-Eye strategy结合的进化计算

Exclusion有一个不足是exploit不足，而explore则发挥作用较多，因此出现当种群规模趋大时，所求得的最优解甚至不如canonical GA的现象。

采用exclusion和Bird-Eye strategy结合，能克服exclusion的上述不足。

对Bird-Eye Strategy编程采用从上到下的编程过程，即从主函数入手，当发现需要解决底层问题，再解决的办法，即用需求驱动的办法编程。编程实现中的一个重要方面就是精搜索空间精度划分的选择与确定。因此，在生成个体时，需要传递搜索精度，即代替VariableProperties()[i][2]的取值。

最底层的种群可以不限制其精度，即可以认为是连续的搜索空间。而顶层的种群，则需要限制其搜索精度，因此需要回答如下问题：搜索精度如何确定？顶层种群规模为|P0|，搜索迭代次数为T，则意味着顶层种群在exclusion机制下要搜索|P0|\*T个解。假如规定其能力为m的话，即

|S|/m=|P0|\*T

即访问一个解，能够代表m个解。因此，m是搜索精度确定的一个主要依据。

|S|=|P0|\*T\*m

易知

∏*i*=[1,*D*](*xiu*-*xid*)/ε*i*=|S|

规定:

(*xiu*-*xid*)/ε*i*=(*xju*-*xjd*)/ε*j*，*i*,*j*∈[1,D]

即在各维上分配的搜索资源相同，或者说在各维上分割的份数相同，则有

((*xiu*-*xid*)/ε*i*)*D*=|S|

从而可以得到

ε*i* =(*xiu*-*xid*)/|S|1/*D*=(*xiu*-*xid*)/(|P0|\*T\*m)1/*D*

其中，这个参数说明如下：

*xiu*, *xid*, *D*是关于优化问题的参数，是确定的；

|P0|是关于算法设置的；

m相当于efficiency的值，是与所使用的算法，算法的参数策略等相关的，是一个几乎不可知的和不确定的数，只能估计设定，为提高算法搜索的解的质量，可以设置为较小值，即对算法的性能不抱太大期望，从而可以为其分配较多的资源，期望得到不因为算法较差，而搜索解的质量也差。

T是算法迭代次数，取决于计算资源或容忍时长，如果计算资源充足，且容忍时间足够长，则T可以取较大的值，反之，T取较小的值。在算法性能对比中，通常设置算法与优化问题交互次数相同，即算法传递入具有black box本质的优化问题的*x*的个体是相同的，或者说，从优化问题本身得到的关于问题的响应次数是一样的。结合前面的讨论，即TΣ *i*=[1,*D*]|P|相同。从而不同算法在拥有关于问题的相同响应次数情况下，充分利用这些信息，给出自己的最优解。另一方面，

T=Ne/(Σ *i*=[1,*lp*]|P*i*|)

其中，*lp*是种群层数，而P*i*是第*i*层的种群。

在不同层次，通过m值的变化，调节resolution的值或精度。上面已经给出了飞在最高层的鸟的精度计算方法，而在最底层，即设置其精度为0，即为连续空间。而不*k*∈(1,*lp*)层，则m取值为：

*mk*=*mk*

*m*为设置的最高层的精度

在不同层次飞行的鸟之间信息沟通，采用信息从飞得高的鸟群向飞得低的鸟群传递的方式进行，即依据精度从粗向细的方向，进行最优解传递。传递的信息，主要传递最优解。在是否要传递最差解的问题上，由于结合了Exclusion机制，各层种群都会排除已经访问过的个体，因此，不再传递最差解信息。

在不同层次的鸟群，其exploration和exploitation的侧重点不同。顶层的更侧重于exploration，而底层的更侧重于exploitation。因此，对应的变异点数目也有所不同，顶层的变异在各个GSU都变异，而底层的变异，则只在一个GSU上变异。假如个体的GSU个数为gsu，则层次与变异点个数m的关系为：

m=gsu, flylevel=顶层

m=1，flylevel=底层

m=(gsu-1)/层数

由于不同层次在搜索空间的resolution不同，所以其种群初始化和变异的粒度也不同。

在变异方面，其不同主要体现在变异步长方面，即不同层次的个体变异，控制在一个步长范围。用RandomGenerator.getRandom(0, step)进行控制的，其中step即为精度。

在种群初始化方面，则个体之间在各个维上的距离是步长的整数倍数。

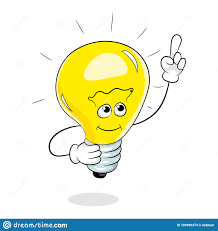
##### 在Individual中加入属性，实现BirdEey

例如在Individual中加入Index属性，可以实现PSO的particle标识，为实现前面的BirdEye操作，在Individual中加入两个属性：

* 一个属性explo\_ra\_itation，该值取0，则进行exploration操作，取1，则进行exploitation操作，取-1，则进行自由随机操作。
* 另一个属性是ε，控制精度或resolution。从而可以实现与前面相同的BirdEye的操作，即在不同层次上飞行时，执行exploration或exploitation。这个值采用大于1的方法，即如果取1，则是飞行最低的鸟。

在算法运行过程中，根据运行状态自如调整这两个属性，从而达到灵活控制算法的目的。可以实现的功能有：

* 种群大小的调整。原来使用多个种群，实现鸟群在多层次飞行；如果想增加或减少种群的个数是一件极其麻烦的事情，因为涉及到数组第一维的调整，而Population这个类在多个地方用到，所以极其麻烦。而当采用在Individual中加入属性后，这种调整要灵活得多。鸟群的飞行层次，甚至可以实现在只有一层，或可以有|P|层。
* 随时控制exploitation和exploration的trade off，而且这种调节是无级调节。因为只要修改属性ε的值就可以了。

进一步，不需要explo\_ra\_itation属性，因为在属性ε已经可以实现对搜索空间的各种resolution上的分割了，而不需要明确地指明是exploration或exploitation!当属性ε值较大时，是在执行exploration，而属性ε的值较小时，则是在执行exploitation。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | 在GeneticAlgorithm类中的方法public void operatePopulation(Population pop)中实现BirdEyeView  if(ControlParameters.birdEye){//依据个体适应值设置个体的exploreration和exploitation  int i=pop.getIndividuals().size();  for (Object ind: pop.getIndividuals()) {  i--;  Individual temind=(Individual)ind;  temind.setExplo\_ra\_itationFact(i/ControlParameters.birdLevelNumber);  }  //在这里可以设置动态自适应的ControlParameters.birdLevelNumber，从而实现exploration和Exploitation的自适应调节  } |

##### 竟争交叉与变异

为了实现灵活的exploration与exploitation的动态调整，采用crossover与mutation竞争执行的方法。这样，mutation的概率与crossover的概率不必设置。

要求种群规模大于2；否则竞争时，交叉个体数目会太少。

##### 二进制编码中天然含有BirdEye

在遗传算法中，当采用二进制编码时，变异位置不同，带来的Exploitation和Exploration的效果不同，当变异位置在高位时，相当于Exploration；当变异位置在低位时，相当于Exploitation。因此，在传统的二进制的遗传算法中自然就包含了BirdEye了。

这也就是为什么二进制具有较好的优势，因为既具有Exploitation，也具有Exploration的功能，因此，它会在很多问题上有较好的性能。

但是当采用实数编码时，如果变异算法处理不当，会削弱变异的Exploration的功能。

##### 基于BirdEye的方法

在类GeneticAlgorithm的方法operatePopulation中，设置每个个体的explo\_ra\_itationFact值，而在类RegularMutation的方法mutate中调用，即通过变异，实现鸟的explore和exploit调节。

另一种调节办法是修改变异点数目，通常这个值等于决策变量维数。可以让其小于这个维数，实现exploit。

因此，将变异位数作为Individual的一个属性，从而可以实现灵活控制。 需要设置其初值，以及设置其调整的值。

#### 关于实数编码与二进制编码的变异

变异操作在这两种编码上带来的效果差异很大。

在实数编码上，一般可以在各维上增加或减少一个随机数rand.nextDouble()，实现变异。

在二进制编码上，一般通过0和1的反转，实现变异。

表面上看起来，都实现了变异，但是对于二进制来说，这种变异中既包括exploitation,也包括exploration。因为在高位变异，对应的实数变异前后差距会很大，所以是exploration；相反，在低位变异，对应的实数变异前后差距不会太大，所以是exploitation。然而，对于实数编码来说，前述变异中，则只包括了exploitation，而没有包括exploration。因此，编码的选择，要根据具体情况进行选择，另外，也要在算法中做相应的处理，例如对exploration和exploitation的调节与控制等。

### 测试函数

#### 来源

在下面的链接中提供了许多benchmark，可以借鉴。

<http://benchmarkfcns.xyz/fcns>

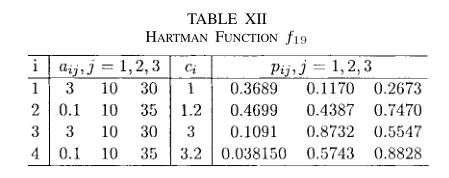
<https://www.cs.unm.edu/~neal.holts/dga/benchmarkFunction/>

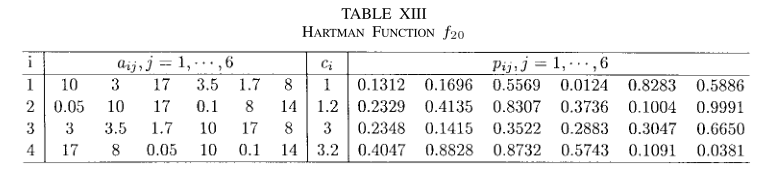
#### 系统已经包含的函数

**Table SI. Benchmark functions**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Name | Function | | Search range | Maximum value | Optima | *d* | Modal |
| *f*0: Triangle | *f*0(*x*)=*x*sin(10π*x*)+2 | | [-2, 2] | 3.8502736 | *x*=1.8505579 | 1 | *m* |
| *f*1: De Jong 1 | *f*1(*x*)=∑*i*=1,3*xi*2 | | [-5.12,5.12] | 78.643196484375 | *xi* =±5.12 | 2 | *m* |
| *f*2: De Jong 2 | *f*2(*x*)=100(*x*12- *x*2)2+(1-*x*1)2 | | [-2.048,2.048] | 3905.926 | *xi* =-2.048 | 2 | 1 |
| *f*3: De Jong 3 | *f*3(*x*)=∑*i*=1,5⌊*xi*⌋ | | [-5.12,5.12] | 25 | *xi* =5.12 | 5 | 1 |
| *f*4: Kowalik | *f*4(*x*)=∑*i*=1,11[*ai-x*1*(b2+bix*2*)/(bi2+bix*3*+x*4*)*]2  *ai* = {4.0, 2.0, 1, 1/2, 1/4, 1/8, 1/10, 1/12, 1/14, 1/16}  *bi =* {0.1957, 0.1947, 0.1735, 0.1600, 0.0844, 0.0627, 0.0456, 0.0342, 0.0323, 0.0235, 0.0246} | | [-5,5] | 0.0003074861 | *xi* =(0.192833, 0.190836, 0.123117, 0.135766) | 4 | 1 |
| *f*5: Shekel’s Foxholes | where [*aij*]= | | [-65.5,65.5] | 500 | *xi=*0 | 2 | 1 |
| *f*6: Schaffer 1 |  | | [-100,100] | 0 | *xi=*0 | 2 | 1 |
| *f*7: Schaffer 2 |  | | [-100,100] | 0 | *xi=*0 | 2 | 1 |
| *f*8: Goldstein -Price |  | | [-2,2] | -3 | *x=*(0,-1) | 2 | 1 |
| *f*9: Shubert | *f*9(*x*)=-∏*i*=1,5∑*j*=1,5(cos((*j*+1)*xi*+*j*)) | | [-10,10] | 186.7309 |  | 5 | *m* |
| *f*10: Six-hump Camel Function | *f*10(*x*)=-((4-2.1*x*12+*x*14/3) *x*12+*x*1*x*2 +(4*x*22-4)*x*22) | | [-2,2] | 1.0316 | (0.0898,-0.7126),  (-0.0898,0.7126) | 2 | 1 |
| *f*11: Sphere | *f*11=-∑*i*=1,*Dxi*2 | | [-100,100] | 0 | *xi=*0 | *D* | 1 |
| *f*12:SumSquares | *f*12=-∑*i* =1,*Dixi*2 | | [-10,10] | 0 | *xi=*0 | *D* | 1 |
| *F*13:Schwefel 2.22 | *f*13=-∑*i*=1,*D|xi*|+∏*i*=1,*D*|*xi*| | | [-10,10] | 0 | *xi=*0 | *D* | 1 |
| *f*14:Schwefel 1.2 | *f*14=-∑*i*=1,*D*(∑*j*=1,*i xj*)2 | | [-100,100] | 0 | *xi=*0 | *D* | 1 |
| *f*15:Schwefel 2.21 | *f*15=-max*i*{|*xi*|},1≤*i*≤*D* | | [-100,100] | 0 | *xi=*0 | *D* | 1 |
| *f*16:Exponential | *f*16=exp(-0.5∑*i*=1,*D xi*2) | | [-1,1] | -1 | *xi=*0 | *D* | 1 |
| *f*17:Tablet | *f*17=-106*x*12-∑*i*=2,*D xi*2 | | [-100,100] | 0 | *xi=*0 | *D* | 1 |
| *f*18:Zakharow | *f*18=-∑*i*=1,*D xi*2+(∑*i*=1,*D*0.5*ixi*)2+(∑*i*=1,*D*0.5*ixi*)4 | | [-5,10] | 0 | *xi=*0 | *D* | 1 |
| *f*19:Step | *f*19=-∑*i*=1,*D* (0.5+*xi*)2 | | [-100,100] | 0 | *xi=-*0.5 | *D* | 1 |
| *f*20:Rosenbrock | *f*20=-∑*i*=1,*D*-1(100(*xi*+1-*xi2*)2+(*xi*-1)2) | | [-30,30] | 0 | *xi=*1 | *D* | 1 |
| *f*21:Griewank | *f*21=-1-∑*i*=1,*D xi*2+∏*i*=1,*D*cos() | | [-600,600] | 0 | *xi=*0 | *D* | 1 |
| *f*22:Schaffer 2 | *f*22=-∑*i*=1,*D*-1(*xi*2+ *xi*+12)0.25(sin2(50(*xi*2+ *xi*+12)0.1)+1) | | [-100,100] | 0 | *xi=-*0 | *D* | 1 |
| *f*23:Schwefel 2.26 | *f*23=∑*i*=1,*D* (*xi* sin() | | [-500,500] | 418.983*D* |  | *D* | 1 |
| 本函数与下面的这个函数类似，但有所不同。  参见：<https://www.sfu.ca/~ssurjano/schwef.html> | | | | | | | |
| *f*24:Himmelblau | *f*24=-1/*D*∑*i*=1,*D*(*xi*4-16*xi*2+5*xi*) | | [-100,100] | 78.3323 |  | *D* | 1 |
| *f*25:Levy and Montalvo 1 | *f*25=-*π/D*(10sin2(*πy*1)+∑*i*=1,*D*-1(*yi*-1)2(1+10sin2(*πyi*+1))+(*yD*-1)2),*yi*=1+0.25(*xi*+1) | | [-10,10] | 0 |  | *D* | 1 |
| *f*26: Levy and Montalvo 2 | *f*26=-0.1(sin2(3*πx*1)+∑*i*=1,*D*-1(*xi*-1)2(1+sin2(3*πxi*+1))+(*xD*-1)2(1+sin2(2*πxD*)))) | | [-5,5] | 0 |  | *D* | 1 |
| *f*27:Ackley | *f*27=20exp(-0.02)+exp(*D*-1∑*i*=1,*Dcos*(2*πxi*))-20-e | | [-30,30] | 0 |  | *D* | 1 |
| *f*28:Rastrigin | *f*28=10*D*+∑*i*=1,*D*(*xi*2-10cos(2*πxi*)) | |  |  |  |  | 1 |
| *f*29:Penalized 1 | *f*29==-*π/D*(10sin2(*πy*1)+∑*i*=1,*D*-1(*yi*-1)2(1+sin(*πyi*+1)))- ∑*i*=1,*D u*(*xi*,10,100,4),  *yi*=1+0.25(*xi*+1),  *u*(*xi*,*a*,*k*,*m*)= | | [-50,50] | 0 |  | *D* | 1 |
| *f*30: Penalized 2 | *f*30==-0.1{sin2(3*πx*1)+∑*i*=1,*D*-1(*xi*-1)2[1+sin2(3*πxi*+1)]+(*xD*-1)2[1+sin2(2*πxD*)]}- ∑*i*=1,*D u*(*xi*,5,100,4), | | [-50,50] | 0 |  | *D* | 1 |
| *f*31:Neumarier 3 | *f*31=-∑*i*=1,*D* (*xi*-1)2-∑*i*=1,*D xi xi*-1-*D*(*D*+4)(*D*-1)/6 | | [-*D*2, *D*2] | 0 |  | *D* | 1 |
| *f*32:Salomon | *f*32=-1+cos(2*π*||*x*||)-0.1||*x*||,||*x*||=∑*i*=1,*D xi* | | [-100,100] | 0 |  | *D* |  |
| 原来的最小值问题在这里变为了最大值问题，下面这个链接虽然不同于本函数（本函数是求绝对值，而下面的是平方和的开方），  参见<http://benchmarkfcns.xyz/benchmarkfcns/salomonfcn.html> | | | | | | | |
| *f*33:Alpine | *f*33=-∑*i*=1,*D* |*xi*sin*xi*+0.1*xi*| | | [-10,10] | 0 |  | *D* | 1 |
| 原来的最小值问题，这里变为了最大值问题  参见<http://benchmarkfcns.xyz/benchmarkfcns/alpinen1fcn.html> | | | | | | | |
| *f*34:Noisy Quaric | *f*34=∑*i*=1,*D ixi*4+random(0,1) | | [-1.28,1.28]*D* | 0 |  | *D* | *m* |
| *f*35:Branin | *f*35=-(*x*2-5.1/4*π2*∙ *x*12+5/*π*∙ *x*1-6)2-10(1-0.125*π*)cos*x*1-10 | | [-5,10]x[0,15] | -0.397887 |  | 2 | 1 |
| *f*36: Hartman's family 4 | ∑*i*=1,4*ci*exp(-∑*i*=1,*D aij*(*xj*-*pij*)2) | | [0,1]*D* | 3.86278 |  | 4 | 1 |
| *f*37: Hartman's family 5 | [0,1]*D* | 3.32237 |  | 6 | 1 |
| *f*38: Shelel's family 5 | ∑*i*=1,*m*[(*x*-*ai*) (*x*-*ai*)*T*+*ci*]-1 | *m=*5 | [0,10] | 1/*c*5 | *a*5 | 4 | 1 |
| *f*39: Shelel's family 7 | *m=*7 | [0,10] | 1/*c*7 | *a*7 | 4 | 1 |
| *F*40: Shelel's family 10 | *m=*10 | [0,10] | 1/*c*10 | *a*10 | 4 | 1 |
| *f*41: Storn’s Chebyshev Polynomial Fitting | *f*41=-*p*1-*p*2 –*p*3  *p*1= *u*=∑*i*=1,*D xi*1.2*D*-*i*  *p*2= *v*=∑*i*=1,*D xi*(-1.2)*D*-*i*  *p*3= *wk*=∑*i*=1,*D xi*(2*k*/*m*-1)*D*-*i*  *k*=0,1,…,*m*, m=32*D*  *d*=72.661, for *D*=9 | | [-8192,8192] | 0 |  | 9 | 1 |
| *f*42: Inverse Hilbert Matrix | *f*42=∑*i*=1,*D*∑*k*=1,*D*|*wi,k*|  (*wi,k*)=W=HZ-I, I=  H=(*hi,k*), *hi,k* =1/(*i*+*k*-1),*i*,*k*=1,2,…,*n*,*n*=*D*0.5  *Z=*(*zi,k*), *zi,k=xi+n*(*k*-1) | | [-16384,16384] | 0 |  | 16 | 1 |
| *f*43: Lennard-Jones Minimum Energy | *f*43=12.7120622568+∑*i*=1,*n*-1∑*j*=*i*+1,*n*(1/*di*,*j*2-2/*di,j*),  *di,j*=(∑*k*=0,2(*x*3*i*+*k*-2-*x*3*j*+*k*-2)2)3,*n*=*D*/3 | | [-4,4] | 0 |  | 18 | 1 |
| *f*44: Weierstrass | *f*44= (∑*k*=1,*D*(∑*k*=0,*kmax*[*ak*cos(2π*bk*(*xi*+0.5))])-*D*(∑*k*=0,*kmaxakcos*(π*bk*)) | | [-100,100] | 0 |  | 10 | 1 |
| *f*45: Modified Schwefel | *f*45= 418.9829*D* -∑*i*=1,*Dg*(*zi*)  *zi*= *xi* + 420.9687462275036  *g*(*zi*)= | | [-100,100] | 0 |  | 10 | 1 |
| *f*46: Expanded Schaffer’s F6 | *f*46=*g*(*x*1,*x*2)+*g*(*x*2,*x*3)…+*g*(*xD*-1,*xD*)+*g*(*xD*,*x*1),  *g*(*x*,*y*)=0.5+(sin2[(*x*2+y2)0.5]-0.5)/(1+0.001[(*x*2+y2)0.5])2 | | [-100,100] | 0 |  | 10 | 1 |
| *f*47: Happy Cat | *f*46=*|∑i=*1,*D xi*2*-D|*0.25*+*(0.5*∑ i=*1,*D xi*2*+∑ i=*1,*D xi*)*/D+*0.5 | | [-100,100] | 0 |  | 10 | 1 |
| *f*48:系数估计 | y=x2+2x+3 | |  | （1,2,3） |  |  |  |

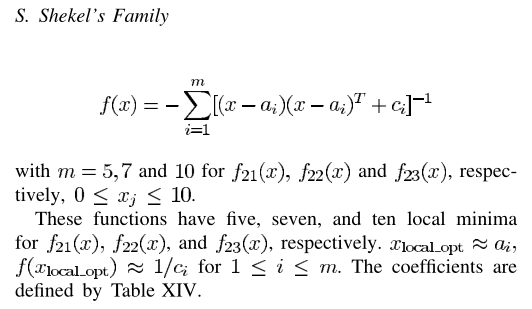
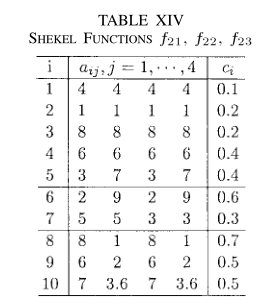
其中编号为11-33的benchmark来自于[1]的附录文件（该文献则参考了[2]中的函数），是求最小化问题。在本系统中这些问题被转换为最大化问题，且其编号分从11开始，一直到33，即从图中原来的fi变成了f(i+10)。

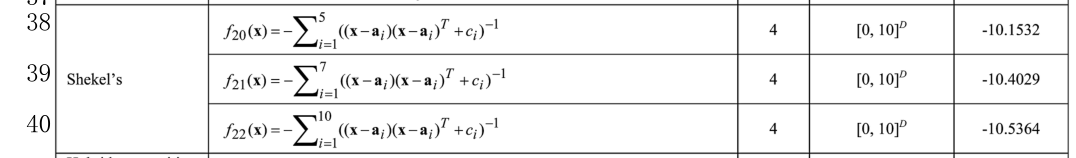
 上面表达式中应该有问题，即j=1,4，表中只组出了3列，j不可能达到4。









### 搜索空间

### 进行grid化处理

把搜索空间进行grid化处理，得到其对应的索引下标，完成。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44 | public class JavaApplication1 {  public static void main(String[] args) {  // TODO code application logic here  JavaApplication1 ja = new JavaApplication1();  int dimension=4;  int[][] begin={{0},{1}}; //第1维上的分割  int[] second={0,1};  int[][] a =new int[begin.length][begin[0].length];  //从begin拷贝到a  for (int i = 0; i < begin.length; i++) {  System.arraycopy(begin[i], 0, a[i], 0, begin[i].length);  }  for (int i = 1; i < dimension; i++) {  a=ja.composition(a, second);  }  for (int[] a1 : a) {  for (int j = 0; j < a1.length; j++) {  System.out.print(a1[j] + ",");  }  System.out.println("");  }  }  public int[][] composition(int[][] last, int [] split){  int[][] result=new int[last.length\*split.length][last[0].length+2];  //行变为原来的k倍,列新增2列，用于存储下限与上限    //首先复制原数组  for (int i = 0; i < last.length; i++) {  for (int j = 0; j < split.length; j++) {  System.arraycopy(last[i], 0, result[i\*split.length+j], 0, last[i].length);  }  }  //然后在新数组后追加新内容  for (int i = 0; i < last.length; i++) {  for (int j = 0; j < split.length; j++) {  result[i\*split.length+j][last[0].length]=j;  result[i\*split.length+j][last[0].length+1]=j+1;  }  }  return result;  }  } |

输出为：

|  |
| --- |
| 0,0,1,0,1,0,1,  0,0,1,0,1,1,2,  0,0,1,1,2,0,1,  0,0,1,1,2,1,2,  0,1,2,0,1,0,1,  0,1,2,0,1,1,2,  0,1,2,1,2,0,1,  0,1,2,1,2,1,2,  1,0,1,0,1,0,1,  1,0,1,0,1,1,2,  1,0,1,1,2,0,1,  1,0,1,1,2,1,2,  1,1,2,0,1,0,1,  1,1,2,0,1,1,2,  1,1,2,1,2,0,1,  1,1,2,1,2,1,2, |

接下来把下标变成实际的子空间上下区间值。

分割搜索空间，并且可以得到各子空间间的坐标。

写了搜索空间及其分割的相关类，动态进一步分割的类还没写。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38 | package searchSpace;  /\*\*  \*  \* @author HAO Guo-Sheng  \* 包含维数与各维区间，以及作为子结点的搜索空间所需要的索引index与层次level  \*/  public class SearchSpace {    private final int dimension;  private final double[][] space;//行数等于维数，列数为3列，第1列存储下限，第2列存储上限，第3列存储下限送去上限的值，第3列的作用是用空间换计算的时间，因为这个差值会被多次用到  private final int index;  private final int level;    public SearchSpace(int dimension, double[][] space,int index,int level) {  this.dimension=dimension;  this.space=space;  this.index=index;  this.level=level;  }  public int getDimension() {  return dimension;  }  public double[][] getSpace() {  return space;  }  public int getIndex() {  return index;  }  public int getLevel() {  return level;  }  } |

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89  90  91  92  93  94  95  96 | package searchSpace;  /\*\*  \*  \* @author HAO Guo-Sheng  \*/  public class InitialGridedSpace extends SearchSpace {  private int[] gridNumber;  int splitNumber = 0;  private double[][] upLowInterval;  public InitialGridedSpace(int dimension, double[][] space,int index,int level, int splitNumber) {  super(dimension, space,index,level);  this.splitNumber = splitNumber;  }  int[][] splitSpace;  public int[][] getSplitSpace() {  if (null == splitSpace) {  splitSearchSpace();  }  return splitSpace;  }  private void splitSearchSpace() {  int[][] begin = new int[splitNumber][1]; //第1维上的分割,这是一个列向量，与其他分割要进行组合  int[] second = new int[splitNumber];//  //组合的第2个元素  //标明要分割成几份，如果分割成2分用{0，1}表示，如果分割成3份用{0，1，2}表示  for (int i = 0; i < splitNumber; i++) {  begin[i][0] = i;  second[i] = i;  }  splitSpace = new int[begin.length][begin[0].length];  //a用来保存结果  //从begin拷贝到a，初始化a，然后再进化组合  for (int i = 0; i < begin.length; i++) {  System.arraycopy(begin[i], 0, splitSpace[i], 0, begin[i].length);  }  //进行组合  for (int i = 1; i < getDimension(); i++) {  splitSpace = composition(splitSpace, second);  }  }  /\*  功能：组合，对于给定的数组last与split数组中的每个元素进行组合  \*/  private int[][] composition(int[][] last, int[] split) {  int[][] result = new int[last.length \* split.length][last[0].length + 1];  //行变为原来的k倍,列新增1列，用于存储索引标识  //首先复制原数组，这种办法比较费时间，因为对于每一维都要被多次重复复制，但这种方法简单  for (int i = 0; i < last.length; i++) {  for (int j = 0; j < split.length; j++) {  System.arraycopy(last[i], 0, result[i \* split.length + j], 0, last[i].length);  }  }  //然后在新数组后追加新内容  for (int j = 0; j < split.length; j++) {  for (int i = 0; i < last.length; i++) {  result[i \* split.length + j][last[0].length] = j;  //last[0].length对应于维数  }  }  return result;  }  public int[] getGridNumber() {  return gridNumber;  }  public void setGridNumber(int[] gridNumber) {  this.gridNumber = gridNumber;  }  public double[][] getUpLowInterval() {  //各维度上分割得到子空间的上下标  if (null == upLowInterval) {  upLowInterval = new double[getDimension()][splitNumber + 1];  for (int i = 0; i < getDimension(); i++) {  for (int j = 0; j < splitNumber ; j++) {  upLowInterval[i][j]=getSpace()[i][0]+(getSpace()[i][2]/splitNumber)\*j;  }  upLowInterval[i][splitNumber] = getSpace()[i][1];  }  }  return upLowInterval;  }  } |

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51 | package searchSpace;  import java.util.ArrayList;  /\*\*  \*  \* @author HAO Guo-Sheng  \*/  public class NodeOfSearchSpace extends InitialGridedSpace {  private SearchSpace root;  private ArrayList<SearchSpace> childrenNodes;  public NodeOfSearchSpace(int dimension, double[][] space, int index, int level, int splitNumber) {  super(dimension, space, index, level, splitNumber);  }  public SearchSpace getRoot() {  if (null == root) {  root = new SearchSpace(getDimension(), getSpace(), getIndex(), getLevel());  }  return root;  }  public ArrayList<SearchSpace> getChildrenNodes() {  if (null == childrenNodes) {  int levelIndex = getLevel() + 1;  childrenNodes = new ArrayList<>();  int[][] splitSpaceIndex = this.getSplitSpace();  //关于各维数的下标索引，第i列对应i维，取值对应于分割的编号索引，取值范围为从0到splitNumber-1  double[][] upAndLowInterval = this.getUpLowInterval();  //关于各维的分割区间的上下范围  for (int i = 0; i < splitSpaceIndex.length; i++) {  //childrenNodes中子搜索空间个数与splitSpaceIndex数组的行数相等，都是幂指数splitNumber^dimension  double[][] spaceInterval = new double[getDimension()][3];  //搜索空间的维数及各维的区间上下值  for (int j = 0; j < getDimension(); j++) {//对每一维进行处理  spaceInterval[j][0] = upAndLowInterval[j][splitSpaceIndex[i][j]];  spaceInterval[j][1] = upAndLowInterval[j][splitSpaceIndex[i][j ]+ 1];  //SplitSpaceIndex的[i][j]+1对应的索引是其后一个索引，即根据下限找到上限  spaceInterval[j][2] = spaceInterval[j][1] - spaceInterval[j][0];  }  childrenNodes.add(new SearchSpace(getDimension(), spaceInterval, i, levelIndex));  //在第levelIndex层上的第i个结点  }  }  return childrenNodes;  }  } |

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26 | package javaapplication1;  import java.util.ArrayList;  import searchSpace.NodeOfSearchSpace;  import searchSpace.SearchSpace;  public class Main {  //Members  public static void main(String[] args) {  NodeOfSearchSpace nodeOfSearchSpace = new NodeOfSearchSpace(2, new double[][]{{0, 1, 1}, {0, 1, 1}}, 0, 0, 2);  ArrayList<SearchSpace> leaf = nodeOfSearchSpace.getChildrenNodes();  System.out.println("level:" + leaf.get(0).getLevel());  for (SearchSpace searchSpace : leaf) {  System.out.print("index:" + searchSpace.getIndex() + "---");  for (int i = 0; i < searchSpace.getDimension(); i++) {  for (int j = 0; j < searchSpace.getSpace()[i].length; j++) {  System.out.print(searchSpace.getSpace()[i][j] + ",");  }  }  System.out.println("");  }  }  } |

### 类Searchspace的作用

在实现Bird-Eye方法时，发现Searchspace几乎不起什么作用，完全可以用Problem类来承担其角色，因此将其移除了。

### 需要解决的问题

音乐问题中关于多旋律的问题没解决

### 决策变量与个体

以前只有一个类Individual，这个类中包括的属性有决策变量和适应值。决策变量的类型有Double，String，BinaryCode等。因此，对应的Problems系列类也有对应的这些类型，另外，对应的Population也有对应的这些类型。

由于TreeSet的引入，其Set特质之一是元素不重复，而这种不重复实现所依赖的方法是equal方法。Indivdual的equal需要判断两个对象的适应值是否相同，但是适应值相同，不一定是同一个个体，因为他们的决策变量可能不同。而如果在equal中加入决策变量因素，则又无法直接利用适应值，因此，决定把决策变量独立出来，新建一个包decisonVariable，在其中加入了接口，抽象类及相关类型，这些类型包括：Double，String，BinaryCode。这样使得Individual类不必再判断决策变量的类型。

在此基础上，进一步处理Population，原来该包中也包括Double，String，BinaryCode等多种类型，现在发现他们不同之处仅在于对Individual的初始化方法，而初始化方法可以委托给决策变量，因此，Population也不必包含这些类型了。

实现上述目标的过程中，使用了泛型方法，这其中有一个问题是如何根据模型类型实例化一个对象。

### Population中TreeSet

基于Population要执行选择操作，而基于Population中Individual也要执行exclusion operator，因此Population中有两套TreeSet，一套是关于决策变量的，一套是关于目标变量的。

经过了前述一系列调整，需要重新执行下面的检查步骤：

* 检查CGA是否可以运行；
* 检查4种Exclusion operator是否可以运行；
* 检查IGA是否可以运行
* 检查BirdEye策略是否可以运行
* 减少Clone()，减少内存消耗
* histroyIndividuals.add在生成个体后，就直接调用了，应该在真正选择算子执行前再执行这个操作，即确定已经在population呆过了，即和其他个体参与的竞争了。

完成检查后，需要重新做实验

### 新个体的生成

在类GeneticAlgorithm的方法operatePopulation中，当种群被filter而不足种群规模时，生成新的个体，采用的方法是由最优解生成(exploitation)和随机初始化(exploration)各占50%的概率。

## 

## 4 实验与运行

### 实验运行配置文件

运行时配置文件位于resources目录下，以Controlparameters开头，后接数字，该数字即是对应的测试函数编号。虽然该文件以conf结尾，但其实是一个文本文件，打开该文件后，可以看到相关配置说明。

运行时的参数配置依赖于该文件。如遗传算法中的交叉点个数与变异点个数，依赖于该文件中的dimension，分别等于该值减1和0。

参数配置中需要注意的是：种群规模要大于2，因为在选择算子的类SelectionCommonFunction中有一句话：

|  |
| --- |
| proUpLowNo[proUpLowNo.length - 1] = proUpLowNo[proUpLowNo.length - 2] + (proUpLowNo[proUpLowNo.length - 2] - proUpLowNo[proUpLowNo.length - 3]) / 10; |

其中数组proUpLowNo用于保存轮盘赌概率区间，种群规模为2，则下标最大值为3。上面这句话中用到了下标3，如果种群规模小于2，那么会出现求下标为负的数组索引。

#### 关于Knowledge

##### 增加knowledge原因—早熟收敛

当采用#NoRepeatOnSmaePopulationKnowledge,策略时，出现局部收敛的情况，原因是当检测出重复个体后，再生成新个体时，采用的是对最优解变异生成新个体的方法，因此，新个体的生成是一个重要的方法，为此采用下面几种方法：

1. 随机选择一个个体，在其基础上变异，生成新个体；
2. 随机初始化新个体；
3. 远离种群聚类中心生成新个体;

##### Knowledge情况

#SimpleGA, //非最优个体可以再现，同一种群内个体可以重复

#SimpleNoRepeatOnSmaePopulation,//同一种群内个体不重复，新个体从最优解变异得来---Exploitation-----------------这个方法极其容易出现局部收敛，跳不出来的，所以取消

#SimpleNoReappearOnThisAndLastPopulation,//在种群层次上非最优个体不再出现，新个体从最优解变异得来-----------------这个方法极其容易出现局部收敛，跳不出来，所以取消

#SimpleNoReappearOnSearchSpace,//在搜索空间层次上历史个体不再出现，新个体从最优解变异得来-----------------这个方法极其容易出现局部收敛，跳不出来，所以取消

#//下面是要在上面的这几个简单操作上加入不同的新个体生成方式，主要包括以下3种：

#//利用梯度信息--Knolwedge guess--try and error

#//随机初始化--Exploration

#//远离种群聚类中心--Exploration

#//再同基于最优解的变异结合起来，进行随机选择

#NoRepeatOnSmaePopulationKnowledge,//在NoRepeatOnSmaePopulation加上梯度信息

#NoReappearOnThisAndLastPopulationKnowledge,//在SimpleNoReappearOnThisAndLastPopulation加上梯度

#NoReappearOnSearchSpaceKnowledge,//NoReappearOnSearchSpace加上

knowledge=NoRepeatOnSmaePopulationKnowledgeknowledge

### 实验数据

#### 借助Excel

##### 多次写入

在写入实验数据到Excel文件中时，为什么反复调用写入文件，而不是一次性地写入？这是因为实验过程中不断产生数据，而这些数据需要及时地写到文件中，如果一次性写入会在内存中保存大量的数据，从而容易引起heap space out of memory的问题。

##### 数据内容

|  |
| --- |
| 实验数据中第1行至第6行的含义在D2AverageFormular2Cell中有一些描述，可以查看 第1行是运行时间，单位ms 第2行是阶跃情况，目前还没使用这个指标 第3行禁忌域的大小，我们现在没有使用这个指标 第4行是最优解所对应的X值 第5行是最优适应值maxf(X) 第6行是否hit最优解 第7行是找到最优解时或进化结束时的进化代数 |

##### 一些约定

不同的种群规模|P|用不同的sheet，每个sheet的名称即为种群规模的大小的数字。

每个sheet中的前n-1列代表实验运行过程中最优解的值，最后一列，即第n列，是平均值。

如果算法在达到最大评价次数前就找到了最优解，那么每一列的最后会用空缺，但是Excel会自动跳过他们，不影响求平均值。

每个sheet中的记录最优适应值变化的行数的最多行数等于指定的评价次数E除以种群规模|P|。

第列的最后两行分别记录了这次实验运行的时间以及f(x)阶跃频率。

#### 借助数据库

用POI把相关实验数据写到Excel文件中，经常出现java.lang.OutOfMemoryError: Java heap space的问题。采用数据库应该会避免出现这个错误。

数据库：Derby，版本10.13.1.1以上

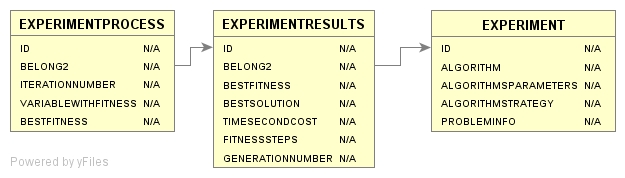
运行模式：采用Client模式

通过ij的连接方法：connect 'jdbc:derby://localhost:1527/experimentalData;user=myeas;password=myeas';

##### 表结构

|  |  |  |
| --- | --- | --- |
| experiment  实验 | create table experiment(id int not null GENERATED ALWAYS AS IDENTITY (START WITH 1, INCREMENT BY 1) primary key, algorithm varchar(20), algorithmsParameters varchar(1000), algorithmStrategy varchar(1000), problemInfo varchar(200)); | 实验配置及环境变量等 |
| experimentResults  实验结果 | create table experimentResults(id int not null GENERATED ALWAYS AS IDENTITY (START WITH 1, INCREMENT BY 1) primary key, belong2 int references experiment(id), bestFitness varchar(1000), bestSolution varchar(2000), timeSecondCost int, evaluationnumber int); | 记录最优适应值，最优解，实验花费时间，实验结束时的评价次数 |
| experimentProcess  实验过程 | create table experimentProcess(id int not null GENERATED ALWAYS AS IDENTITY (START WITH 1, INCREMENT BY 1) primary key, belong2 int references experimentResults(id), iterationNumber int, variablewithFitness LONG VARCHAR, bestfitness varchar(1000));  variablewithFitness字段中varaible与对应的fitness在一起，两者之间的分割符为”#”，种群内个体之间的分割符为“$”，个体的变量各维内部分割以“,”，适应值内部分割也是以“,” | 记录实验过程，主要包括这一种群的适应值与变量 |

表关系如下图：



### 实验数据处理

#### 处理的简捷性表达

下面的这些种群规模是在函数f23上的GAE2占优CGA的统计：2,3,4,6,7,9,10,20,30,40,50,60,70,80,90,100,110,120,130,140,150,160,170,190,200,210,230,250,260,270,280,290,300,320,330,340,350。

没人愿意看到这么密集的数据，如果能换成下面的情况，是不是就好多了。如果手工处理，那么会有32\*3=96个数据集需要处理，每个3分钟，再加上可能出错，纠错2分种，那么，就需要96\*5=480分钟，即8个小时。能不能编个小程序实现一下。

[2,4], [6,7], {9}, [10,170], [190,210], {230}, [250,300]

#### 代码实现

注意到结果中有两种情况，一种是整数区间，即中括号形式，一种是集合，即大括号形式。也就是说，如果连续的话，那么就是中括号，如果不连续，就用集合。 因此，先定义一个类，包容这两种形式：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | class IntegerIntervalOrSet {  private boolean interval = false;  //如果是区间，则为true；否则为false  private List<Integer> list = new LinkedList<>();  public IntegerIntervalOrSet() {  interval = false;  list = new LinkedList<>();  }  //Getor略  //Setor 略  } |

针对区间或集合，定义它们的字符串格式

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | private String outputList(IntegerIntervalOrSet list) {  StringBuilder sb = new StringBuilder();  if (list.isInterval()) {//是区间，只输出第1个和最后一个元素  sb.append("[");  sb.append(list.getList().get(0));  sb.append(",").append(list.getList().get(list.getList().size() - 1));  sb.append("], ");  } else {//是集合，输出所有元素  sb.append("{");  for (Integer number : list.getList()) {  sb.append(number).append(", ");  }  sb.append("}, ");  }  return sb.toString();  } |

如何判断是采用区间表示，还是集合表示？主要看数据元素是否相邻，如果是，则用区间表示，如果不是，则用集合表示。因此写一个函数，返回是否相邻。

|  |  |
| --- | --- |
| 1  2  3 | private boolean iscos(int a, int b, int intervla) {//想邻的间隔为interval  return (b - a == intervla);  } |

当遍历数组时，一个区间或集合结束，则要重新开始一个新的IntegerIntervalOrSet，因此写一个函数，完成重置。

|  |  |
| --- | --- |
| 1  2  3  4  5 | private IntegerIntervalOrSet endList(IntegerIntervalOrSet mylistSet, StringBuilder sb) {  sb.append(outputList(mylistSet));  mylistSet = new IntegerIntervalOrSet();//重新初始化  return mylistSet;  } |

该任务中有两种类型的整数区间，一种是间隔为1，即2~9之间间隔为1，而10~360间隔为10。除此以外，两种类型的处理方法相同。处理思路是，如果后边的数据与当前的数据连续，则归为区间，如果不连续，则归到集合中。从原来的区间到现在的集合，或者反之，需要给出区间或集合的结束符，并重新开始一个新的IntegerIntervalOrSet对象。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22 | private IntegerIntervalOrSet dealSet(IntegerIntervalOrSet mylistSet, int[] a, int i, StringBuilder sb, int upbound, int step) {  mylistSet.getList().add(a[i]);  if (i == a.length - 1) {//数组遍历结束  //sb.append(" and ");//在文章中，最后一种列举情况，加and  mylistSet = endList(mylistSet, sb);  return mylistSet;  }  if (a[i + 1] >= upbound) {  mylistSet = endList(mylistSet, sb);  } if (iscos(a[i], a[i + 1], step)) {//连续，继续  mylistSet.setInterval(true);  } else {//不连续了  if (mylistSet.isInterval()) {//说明有数据，所以要处理  mylistSet = endList(mylistSet, sb);  } else if (i + 2 < a.length) {  if (iscos(a[i + 1], a[i + 2], step)) {//后边的连续了，所以要结束了  mylistSet = endList(mylistSet, sb);  } else {//后边的也不连续，所以可以继续  }  }  }  return mylistSet;  } |

对于原数据，要分开两种情况处理，一种是大于9的，一种是小于10的。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | private StringBuilder dealArray(int[] a) {  StringBuilder sb = new StringBuilder();  IntegerIntervalOrSet mylistSet = new IntegerIntervalOrSet();  int i = 0;  for (; i < a.length - 1; i++) {  if (a[i] < 10) {  mylistSet = dealSet(mylistSet, a, i, sb, 10, 1);  } else {  break;  }  }  //下面处理大于9的情况  for (; i < a.length - 1; i++) {  mylistSet = dealSet(mylistSet, a, i, sb, 370, 10);  }  return sb;  } |

写一个主类Test，测试一下吧

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | public class Test {  public static void main(String[] args) {  int[] a = new int[]{2, 3, 4, 6, 7, 9, 10, 20, 30, 40, 50, 60, 70, 80, 90, 100, 110, 120, 130, 140, 150, 160, 170, 190, 200, 210, 230, 250, 260, 270, 280, 290, 300, 320, 330, 340, 350};  Test test = new Test();  test.dealArray(a);  System.out.println(test.dealArray(a).toString());  }  …  } |

用程序处理96组数据，每组不运行不超过1秒种，拷贝与粘贴假设需要29秒，那么整个数据处理需要时间只需要48分钟，不到1小时时间，最重要的是，程序不会像人一样出错。

### 加入单目标测试函数3步骤

步骤1：写一个类在包problem.TGA.singleObjective中；

步骤2：在包problem的类FactoryProblems的方法registerProblems中注册对应的类，给出编号与名字；

步骤3：在项目的resources目录中写出对应的Controlparameters.conf配置文件。

### 可视化及其实现

工具：借助xChart

使用方法：MVC和生产者消费者设计模式。

被观察对象：Population

视图对象：tools.xChart.MyRealtimeChart01

设计方法：在MyRealtimeChart01中注册被观察对象population（间接注册）；在population中注册视图对象(间接注册)。当数据取出后，还没有新数据之前，设置newdata=false，反之为true。MyRealtimeChart01把setData暴露给population，由population去更新数据。

分工：进化过程动态显示之绘制背景，属于MVC中的M的一个组成部分，应由problem负责，所以在每个具体的problem类中，会有返回一系列均匀分布的X和Y的功能。

调用：执行tools.xChart. RealtimeChartSerials01，即可查看适应值的变化情况。

### 实验出问题的特征

* 求出的最大值大于规定的最大值太多，可能是测试函数的代码写错了
* 实验数据显示只有从第0到第1代的适应值跃迁，测试函数中的stopFitness或函数写错了

## 5 使用的设计模式

### 5.1工厂模式

利用了工厂模式，处理Indivdiual、进化算子，包括选择、交叉与变异、适应值赋值方式、Population、处理编码与解码、Problem、Phenotype等。

PSO和GA两者都属于启发式算法，有许多共同的部分，就像宝马和奥迪一样，都属于车，有许多共同的部分，只不过组装时采用不同的品牌罢了。

系统通过接口定义选择、交叉和变异这3个算子作为遗传算法的成员，具体对象从他们对应的Factory中获取。而这3个算子的输出都是调用接口中的方法，具体实现则由具体的对象完成。

## 6 面向对象编程

### 6.1 参数粒度大小

定义方法时，对象参数既不要太大，也不要太小。太小好理解，太大则是常常容易被忽视的问题。

**参数对象太小的例子**。你开了一个修理铺，专营负责修改座椅。有一天有个人拿了一个木棍来，请你修理，说这是座椅的腿，你可能告诉他，如果腿掉下来了，可以粘上去，但不负责木棍的磨削刨油漆等。

**参数太大的例子**。又有一天，有个人开轿车来了，请你修理车上的座椅，你说好的，放这儿吧，明天或后来你来拿。注意，是让他来拿他的座椅，而不是轿车。轿车这个参数就太大了，因为你在修改座椅的过程中，会占有车，很多其他的业务会因此受影响，甚至不小心，把车的轮胎扎破了，把引擎碰坏了。又有一天，有个人开着飞机来了，让你修理座椅，这个对象参数就更大了。

对于**进化算子**，有人说其作用的对象是种群。虽然这种说法没错，但是还是太大，可以再小些，例如个体集合。因为种群中除了个体集合外，还包括最大适应值，最小适应值等多个属性。在进化操作处理过程中，一不小心，可能就把他们破坏了。

### 6.2 父类或抽象避免越俎代庖

定义一个接口的抽像类，把公共的方法都在抽象类中实现。但是需要注意的是，不要定义太多的成员属性，尤其不要把子类的所有成员属性都定义在抽象类中。因为如果是这要，无关的子类也会变得庞大。如果这些属性是被引用的，则可能会被子类修改；如果这些属性是被重新实例化的，则可能实例化后并不使用，带来无谓的空间浪费。

举例：在本项目中对于Individual的抽象类AbstractIndividual的定义。又如individual是否要定义mutationScope的讨论。每个individual都有一个变异范围，但大多数情况下，这个变异范围对同一问题是固定不变的（但也有例外，如人脸）。如果放进individual类中，这个变异范围的数组就会重复出现多次，尤其当种群规模很大时，将大大无谓增加内存负担。

解决办法之一：上例中，要使用变异范围，怎么办？传递一个problem引用过来，注意，这里传递的是引用，而不是对象值，引用意味着只是指向一个内存空间。

### 6.3抽象类不能通过this来调用自己

抽象类不能通过this来调用自己，因为运行时其具体子类未知。在本项目中，在接口Problem中有一个方法getIndividual(int individualNumber,Problem problem);注意到这里的第2个参数是problem，如果在抽象类AbstractProblem中实现该方法，并传递参数this作为problem的具体值，那么运行时将产生错误，而且对于这种错误，系统目前也很难定位；也就是说，可能产生一个语法上和逻辑上完全正确，但运行时却出错，且位置未知的错误。这种错误可能派1000人也不一定能检查出来。 但是采用这种方法会产生一个奇怪的调用 problem.getIndividual(1, problem)，即problem自己调用自己时，需要把自己作为参数传进来。原因是调用的是父类的方法，是告诉父类：“我来了”

### 6.4 对象的引用与赋值

#### 如果需要赋值，则要避免引用

例一：为什么在许多set方法中用了System.arraycopy方法？这是因为对数组直接赋值，是地址引用赋值，而非值赋值。当其他的对象对该数组做了修改后，也会影响到引用它的对象，这与程序的初衷是背离的。因为我们当初赋值后，是不希望被其他对象影响的，即保持独立性，而不是值的高耦合性。

### 6.5 关于Java的克隆

当所写类是面向类似于泛型所写，即不知道当前类型时什么时，如面向Individual所写，不知具体是FloatIndivdiual或StringIndividual或BianrycodeIndividual等时，利用当前population.get(0).getClone()，然后再setGenecodes或setGenecodes是一个好方法。这样编程，也同时可以避免判断问题是哪一种类型了

当然，对于某些情况下，个体的初始化，可以直接用Clone方法。

这个例子较好地解释了克隆和hashSet

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89  90  91  92  93  94  95  96  97  98  99  100  101  102  103  104  105  106  107  108  109  110  111  112  113  114  115  116  117  118  119  120  121  122  123  124  125  126  127  128  129  130  131  132  133  134  135  136  137  138  139  140  141  142  143  144  145  146  147  148  149  150  151  152  153  154  155  156  157  158  159  160  161  162  163  164  165  166  167  168  169  170  171  172  173  174  175  176  177  178  179  180  181  182  183  184  185  186  187  188  189  190  191  192  193  194  195  196  197  198  199  200  201  202  203  204  205  206  207  208  209  210  211  212  213  214  215  216  217  218  219  220  221  222  223  224  225  226  227  228  229  230  231  232  233  234  235  236  237  238  239  240  241  242  243  244  245  246  247  248  249  250  251  252  253  254  255  256  257  258  259  260  261  262  263  264  265  266  267  268  269  270  271  272  273  274  275  276  277  278  279  280  281  282  283  284  285  286  287  288  289  290  291  292  293  294  295  296  297  298  299  300  301  302  303  304  305  306  307  308  309  310  311  312  313  314  315  316  317  318  319  320  321  322  323  324  325  326  327  328  329  330  331  332  333  334  335  336 | package problem.Individuals;  import controllers.ControlParameters;  import java.util.HashSet;  import java.util.LinkedList;  import java.util.List;  import java.util.logging.Level;  import java.util.logging.Logger;  import org.apache.commons.lang3.builder.HashCodeBuilder;  import problem.population.DoublePopulation;  import tools.MyMath.RandomGenerator;  /\*\*  \*  \* @author 郝国生 HAO Guo-Sheng  \*/  public class DoubleIndividual extends AbstractIndividual {  //  // @Override  // public Individual copyXTo() {//不拷贝适应值，节约内存空间  // DoubleIndividual individual = null;  // try {  // individual = (DoubleIndividual) this.clone();  // individual.setFitness(null);  // } catch (CloneNotSupportedException ex) {  // Logger.getLogger(DoubleIndividual.class.getName()).log(Level.SEVERE, null, ex);  // }  // return individual;  // }  @Override  public boolean equals(Object obj) {  if (null == obj || !(obj instanceof DoubleIndividual)) {  return false;  }  if (obj == this) {  return true;  }  //每个决策变量都相等，才相等;而没考虑适应值  int i = 0;  DoubleIndividual tem = (DoubleIndividual) obj;  for (int j = 0; j < getGeneCodes().length; j++) {  if (tem.getGeneCodes()[j] == this.getGeneCodes()[j]) {  i++;  } else {  break;  }  }  return i >= getGeneCodes().length;  }  @Override  public int hashCode() {  return new HashCodeBuilder(17, 61).  // if deriving:appendSuper(super.hashCode()).  append(this.getGeneCodes()).  //append(this.getFitness()).  // append(Math.random()).//这所以加这个，就是想要克隆变为深克隆，而不是浅克隆  toHashCode();  }  @Override  public GenecodeType getGenecodeType() {  return GenecodeType.DOUBLECODE;//表示实数编码  }  public double getAdjusted(double value, int location, double[][] scope) {  double result = value;  if (value > scope[location][0]) {//超过了上限  result = scope[location][0] - RandomGenerator.nextDouble();  }  if (value < scope[location][1]) {//低于了下限  result = scope[location][1] + RandomGenerator.nextDouble();  }  return result;  }  @Override  public String getX() {  StringBuilder result = new StringBuilder();  for (int i = 0; i < getGeneCodes().length; i++) {  result.append(getGeneCodes()[i]);  }  return result.toString();  }  @Override  public String getGeneralGenecodeString() {  StringBuilder result = new StringBuilder();  for (int i = 0; i < geneCodes.length; i++) {  result.append(geneCodes[i]);  }  return result.toString();  }  @Override  public int getSolutionNumberDistance(Individual individual) {  int result = 1;  for (int i = 0; i < this.getGeneCodes().length; i++) {  result \*= Math.abs(this.getGeneCodes()[i] - ((DoubleIndividual) individual).getGeneCodes()[i]) / ControlParameters.currentProblem.getVariableProperties()[i][2];  }  //System.out.println("距离是:"+result);  return result;  }  @Override  public int compareToX(Individual o) {//在搜索空间中进行折半查找时用  for (int i = 0; i < this.getGeneCodes().length; i++) {  if (this.getGeneCodes()[i] > o.getGeneCodes()[i]) {  return 1;  } else if (this.getGeneCodes()[i] < o.getGeneCodes()[i]) {  return -1;  }  }  return 0;//两者相等，是同一个解  }  @Override  public List<Individual> crossover(Individual crossed) {  List<Individual> result = null;  try {  result = new LinkedList<>();  //定义两个结果的clone  Individual individual1 = (DoubleIndividual) this.clone();  Individual individual2 = (DoubleIndividual) crossed.clone();  //修改其基因编码  double[][] crossresult = swapDoubleArray(individual1.getGeneCodes(), individual2.getGeneCodes());  individual1.setGeneCodes(crossresult[0]);  individual2.setGeneCodes(crossresult[1]);  //把结果加入  result.add(individual1);  result.add(individual2);  } catch (CloneNotSupportedException ex) {  Logger.getLogger(DoubleIndividual.class.getName()).log(Level.SEVERE, null, ex);  }  return result;  }  public double[][] swapDoubleArray(double[] currentDoubleArray0, double[] currentDoubleArray1) {  int doubleLength = currentDoubleArray0.length;//基因变量个数  double[][] result = new double[2][doubleLength];//2表示交叉后得到两个个体  if (doubleLength > 1) {  for (int j = 0; j < ControlParameters.crossoverPointNum; j++) {//  int temInt = RandomGenerator.getRandom(0, doubleLength);  if (temInt == 0) {  temInt++;  }  //先拷贝前半部分  System.arraycopy(currentDoubleArray0, 0, result[0], 0, temInt);  System.arraycopy(currentDoubleArray1, 0, result[1], 0, temInt);  //再拷贝后半部分  System.arraycopy(currentDoubleArray0, temInt, result[1], temInt, currentDoubleArray0.length - temInt);  System.arraycopy(currentDoubleArray1, temInt, result[0], temInt, currentDoubleArray1.length - temInt);//  }  } else {//决策变量只有一个,//验证结果在这两个数值的中间  double max = Math.max(currentDoubleArray0[0], currentDoubleArray1[0]);  double min = Math.min(currentDoubleArray0[0], currentDoubleArray1[0]);  if (max == min) {//不进行任何操作，因为两者相等  result[0] = currentDoubleArray0;  result[1] = currentDoubleArray1;  } else {//这种方法只适合于凸问题  double error = max - min;  result[0][0] = min + RandomGenerator.nextDouble() \* error;  result[1][0] = max - RandomGenerator.nextDouble() \* error;  }  }  return result;  }  @Override  public Individual mutate() {  Individual result = this.cloneWithFitness();  if (this.getGeneCodes().length > 2) {  //首先确定变异的决策变量  double[] temDoubleArray = new double[this.getGeneCodes().length];//不能直接引用this.getGeneCodes，因为那样就变成引用了  System.arraycopy(this.getGeneCodes(), 0, temDoubleArray, 0, this.getGeneCodes().length);  int muteLocation = (int) RandomGenerator.getRandom(0, temDoubleArray.length - 1);  temDoubleArray[muteLocation] = this.getGeneCodes()[muteLocation] + RandomGenerator.nextDouble();  temDoubleArray[muteLocation] = this.getAdjusted(temDoubleArray[muteLocation], muteLocation, ControlParameters.currentProblem.getVariableProperties());  result.setGeneCodes(temDoubleArray);//不用设置，也会改变，因为数组是引用传递  } else {// 只有一个或两个决策变量，让第一个进行变异  double[] temDoubleArray = new double[this.getGeneCodes().length];//不能直接引用this.getGeneCodes，因为那样就变成引用了  System.arraycopy(this.getGeneCodes(), 0, temDoubleArray, 0, this.getGeneCodes().length);  temDoubleArray[0] = this.getGeneCodes()[0] + RandomGenerator.nextDouble();  temDoubleArray[0] = this.getAdjusted(temDoubleArray[0], 0, ControlParameters.currentProblem.getVariableProperties());  result.setGeneCodes(temDoubleArray);  }  return result;  }  //下面这个运行证明了equals中对比个体编码方法的有效性，即在HashSet中remove方法时利用了equal，对比编码，而不是在对象层次  public static void main(String[] args) {  HashSet<DoubleIndividual> setIndviduals = new HashSet<>();  DoubleIndividual tem = new DoubleIndividual();  tem.setGeneCodes(new double[]{0.5});  tem.setFitness(new double[]{1.0});  DoubleIndividual aClone = null, bClone = null;  try {  aClone = (DoubleIndividual) tem.clone();  bClone = (DoubleIndividual) tem.clone();  setIndviduals.add((DoubleIndividual) tem.clone());  setIndviduals.add(aClone);  setIndviduals.add(bClone);  System.out.println("setIndviduals的size:" + setIndviduals.size());//结果显示为1  bClone.setFitness(new double[]{0.8});  setIndviduals.add(bClone);  /\*  @Override  public boolean equals(Object obj) {  if (null == obj || !(obj instanceof DoubleIndividual)) {  return false;  }  if (obj == this) {  return true;  }  //每个决策变量都相等，才相等;而没考虑适应值  int i = 0;  DoubleIndividual tem = (DoubleIndividual) obj;  for (int j = 0; j < getGeneCodes().length; j++) {  if (tem.getGeneCodes()[j] == this.getGeneCodes()[j]) {  i++;  } else {  break;  }  }  return i >= getGeneCodes().length;  }  @Override  public int hashCode() {  return new HashCodeBuilder(17, 61).  // if deriving:appendSuper(super.hashCode()).  append(this.getGeneCodes()).  append(this.getFitness()).//  // append(Math.random()).//这所以加这个，就是想要克隆变为深克隆，而不是浅克隆  toHashCode();  }  \*/  System.out.println("setIndviduals的size:" + setIndviduals.size());//结果显示为2，表明在hashSet中未使用equal函数，而是使用了hashcode函数，因为按照equal函数的话，他们是相等的，但使用hashcode函数的话，他们是不相等的；正是因为不相等，所以结果为2  //由此可知，hashCode函数是为HashSet服务的，而HashSet与equals函数不同  //所以，如果想在hashSet只判断决策变量的话，hashCode应该写成只判断决策变量的方法，所以hashcode变成如下的样子。则结果变成了1，表明两个决策变量相同  /\*  @Override  public int hashCode() {  return new HashCodeBuilder(17, 61).  // if deriving:appendSuper(super.hashCode()).  append(this.getGeneCodes()).  //append(this.getFitness()).  // append(Math.random()).//这所以加这个，就是想要克隆变为深克隆，而不是浅克隆  toHashCode();  }  \*/  setIndviduals.add(bClone);  } catch (CloneNotSupportedException ex) {  Logger.getLogger(DoublePopulation.class.getName()).log(Level.SEVERE, null, ex);  }  System.out.println("aClone.equals(tem):" + aClone.equals(tem));//检查equals,结果显示结果为true,即两者内容一样  System.out.println("aClone==tem:" + (aClone == tem));//检查==，结果显示结果为false，即两者不是同一个对象  //修改了原来的对象  tem.setGeneCodes(new double[]{0.9});  tem.setFitness(new double[]{1.2});  //检查一下克隆是否被改变  setIndviduals.forEach((doubleIndividual) -> {  System.out.println(doubleIndividual.getGeneCodes()[0] + ":" + doubleIndividual.getFitness()[0]);  });  //结果显示，克隆没有改变，即对母体tem做改变，不影响clone，即clone不改变  //生成一个新的对象，但和原来的一模一样，但不是一个  tem = new DoubleIndividual();  tem.setGeneCodes(new double[]{0.5});  tem.setFitness(new double[]{1.0});  System.out.println("Before add remove:");  setIndviduals.remove(tem);  System.out.println("After add remove:");  System.out.println(setIndviduals.size());  setIndviduals.forEach((doubleIndividual) -> {  System.out.println(doubleIndividual.getGeneCodes()[0] + ":" + doubleIndividual.getFitness()[0]);  });  }  } |

### 6.6 Java的数据结构TreeSet

#### 利用TreeSet的上界与下界功能

使用TreeSet的celling和floor，从历史个体中获得对应的邻居，然后计算该个体是否落在这些邻居排斥半径内。

下面是一个简单的例子。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | public static void main(String[] args) {  // create tree set object  TreeSet<DecisionVariable> treeadd = new TreeSet<>();    // populate the TreeSet  treeadd.add(new DoubleDecisionVariable(new double[]{10.0}));  treeadd.add(new DoubleDecisionVariable(new double[]{20.0}));  treeadd.add(new DoubleDecisionVariable(new double[]{30.0}));  treeadd.add(new DoubleDecisionVariable(new double[]{40.0}));    // Print the TreeSet  System.out.println("TreeSet: " + treeadd);    // getting ceiling value for 25  // using ceiling() method  DecisionVariable tem=new DoubleDecisionVariable(new double[]{25.0});  DecisionVariable value = treeadd.ceiling(tem);  DecisionVariable value2=treeadd.floor(tem);    // printing the ceiling value  System.out.println("Ceiling value for 25: "  + value+"floor value for 25 is:"+value2);  } |

#### 聚类--利用TreeSet进行处理

在利用历史个体库的过程中，为了防止发生内存溢出的错误，要及时删除一些个体。那么删除哪些个体？一种方法是基于拥挤crowded原则的删除，这时就需要计算个体决策变量之间的距离，从这些距离中挑选最小距离对应的一个个体删除。而在新加入一个个体时，需要计算该个体与所有个体之间的距离，而这种计算其实会重复TreeSet的计算，因为TreeSet加入新的个体时会进行计算，能否利用TreeSet避免或减少距离计算？这就是利用TreeSet进行处理的来由。

现在采用的方法是：只求相邻个体的距离。这依赖于一个假设：对TreeSet进行遍历时，是依据compare()结果从小到大进行的，即是按序遍历。这样，就只需要计算遍历相邻的两个个体即可。当用于parentFilter时，保留适应值大的（当用于offspringFilter时，保留适应值小的，目前还没用到这种情况，因为offspring排斥时，用的是距离排斥）。

为此，需要一个包含属性（DecisionVariable,DecsionVariable, Double）的类，保存变量之间的距离。因为DecisionVariable采用引用，所以并不会太增加内存空间。再将这个结构的变量放在TreeHashSet中，每次删除则弹出first，从中删除适应值小的DecsionVariable即可。

## 一些问题

AbstractAlgorithm的方法onceGetFitness4TEC第43行添加个体到历史库中时，为什么不用克隆，而是直接add？

参考文献

[1] X. Zhou, C. Peng, J. Liu, Y. Zhang, and G. Zhang, "Underestimation-Assisted Global-Local Cooperative Differential Evolution and the Application to Protein Structure Prediction," *IEEE Transactions on Evolutionary Computation,* vol. 24, no. 3, pp. 536-550, 2020.

[2] W.-J. Yu *et al.*, "Differential Evolution With Two-Level Parameter Adaptation," *IEEE Transactions on Cybernetics,* vol. 44, no. 7, pp. 1080-1099, 2014.

[3] X. Yao, Y. Liu, and G. Lin, "Evolutionary programming made faster," *IEEE Transactions on Evolutionary computation,* vol. 3, no. 2, pp. 82-102, 1999.

[4] K. Price, N. Awad, M. Ali, and P. Suganthan, "The 100-digit challenge: Problem definitions and evaluation criteria for the 100-digit challenge special session and competition on single objective numerical optimization," *Nanyang Technological University,* 2018.

[5] B. R. Rajakumar and A. George, "APOGA: An Adaptive Population Pool Size based Genetic Algorithm," *AASRI Procedia,* vol. 4, pp. 288-296, 2013/01/01/ 2013.

[6] J. Arabas, Z. Michalewicz, and J. Mulawka, "GAVaPS-a genetic algorithm with varying population size," in *Proceedings of the First IEEE Conference on Evolutionary Computation*, 1994, pp. 73-78: IEEE.

[7] AI92, "深入浅出设计模式."

[8] S. Y. Yuen and C. K. Chow, "A Genetic Algorithm That Adaptively Mutates and Never Revisits," *IEEE Transactions on Evolutionary Computation,* vol. 13, no. 2, pp. 454-472, 2009.

[9] Á. E. Eiben, R. Hinterding, and Z. Michalewicz, "Parameter control in evolutionary algorithms," *IEEE Transactions on evolutionary computation,* vol. 3, no. 2, pp. 124-141, 1999.