遗传算法求解最优化问题的数值解

中国科学技术大学 计算机科学与技术学院
PB19030888 张舒恒

[摘要]遗传算法来源于生物学，本文基于此算法思想提出了一种求一般最优化问题（包括旅行商问题，函数局部最优值问题等等）数值解的通用方法，并给出了基于C++和python计算机语言的编程实现。

[关键词]遗传算法，最优化问题

一．优化问题

• 优化问题的背景

在日常生活中，任何对现有状况做出的改进都可以叫做优化，这是一个广义的概念。在数学中，狭义的优化特指一类问题，它有三个要素：优化变量、目标函数、约束。在满足约束的情况下，调整优化变量，使得目标函数的值最小或最大，这就是优化问题。

举个例子，我是一名本科生，怎样选课才能使期末的GPA最高？在这个问题中，优化变量是选择哪门课、不选择哪门课，目标函数是预期的GPA，约束是学分要求、课程安排以及你有限的精力。不懂优化的我们也可以解决这个问题，很显然，先选上你最擅长且学分最高的课程，然后是擅长但学分不高的课程，最后是不擅长且学分也不高的课程。每个徘徊在选课阶段的你可能会这样操作的，这和优化问题有没有什么关系？

优化问题的目标函数通常是一个多变量单输出函数，该例子也不例外。我们可以用x_i表示是否选择第i个，1表示选择，0表示不选择，用$f(x)$表示预期的GPA。为什么我们会先选择擅长且学分高的课程呢？因为将这些课程对应的x_i设为1会使得$f(x)$有较大的提高，说明这些变量对函数值的影响较大，用数学的语言来说，这些变量提供了函数的梯度方向。

沿着梯度方向前进，总能到达一个顶峰，这就是优化的目标。然而，现实很复杂，顶峰所处的位置不一定满足约束。也许优化后挑中了所有最擅长的课，结果发现这些课的上课时间是重叠的。或者发现只选一门最最擅长的课可以得到最高的GPA，但是另外一个问题是，学分修不够可毕不了业呀。这些约束都限制着我们的选课范围，从而限制了优化方向，使得优化问题变得棘手起来。

• 优化问题的种类

沿着上面的思路，优化问题可以进行分类。选课问题其实是一个离散、有约束、非凸、非确定性的优化问题，可以说是优化问题中的代表了。如果学校允许选小数个课程（比如选0.3个第i个课程，当然这是不可能的），这个问题就成了一个连续的优化问题。如果学校没有学分要求，不限制选择时间段重叠的课程，那它就变得无约束了。如果你事先非常确定某门课的期末考试你可以得多少分，一分不多一分不少，那这个问题就是确定性问题了。至于这个问题是不是凸的，很难说，它由目标函数的内在性质决定，目前无法得知。如果这一系列假设都满足，问题变成了连续、无约束、凸、确定性的问题，求解起来就会得心应手许多。

• 从定义到性质

对于连续无约束问题来说，其目标函数是$f(x)$，其中，优化变量为$x \in \mathbb{R}^n$，连续函数为$f: R^n \to R$。此外，我们进一步假设f是光滑函数，即处处可导，只有这样才能方便地运用后面的优化算法。这个定义看起来很空洞，十分抽象。但我们不妨考虑这样一个问题，假设优化变量的最优解是x^*，那么x^*需要满足什么要求？显然，对于任意的x，必须满足$f(x^*) \leq f(x)$，这样的解叫做全局极小值。

全局极小值往往是很难找到的，退而求其次，如果在x^*的邻域内，满足$f\left(x^*\right) \leq f\left(x\right)$，这样的解叫做局部极小值。在局部极小值的基础上稍微严格一点，把等号去掉，变成$f\left(x^*\right) < f\left(x\right)$，这就是严格局部极小值。这几个概念很好理解，但还有另外一个定义，叫做孤立的局部极小值。这种极小值可以保证，在任意小的邻域内，x^*都是唯一的局部极小值。按说严格局部极小值应该就可以满足这个要求了，但在我们的直观感受之外，的确存在一些函数，它在无穷小的邻域内存在着无穷多个严格局部极小值。想象在二次函数上叠加一种震荡，越接近原点，震荡的频率越高，但震荡的幅值越小，在原点处，震荡频率达到无穷大，但震荡幅值为0。在这个函数中，原点就是一个严格局部极小值，但在任意小的邻域内，都存在着无穷多个波峰和波谷，也就存在着无穷多个严格局部极小值，所以原点不是孤立的局部极小值。

现在定义了几种极小值，接下来的问题是，给定一个目标函数f，如何计算极小值x^*？我们可以考虑在极小值点x^*处，目标函数f会存在哪些特殊的性质。

首先考察目标函数的一阶导数，如果f连续可微，那么可以想象，在极小值点处的导数应该为0。

再来考察目标函数的二阶导数，在极小值点处，x^*附近的微小扰动都会导致目标函数不变或增大。如果把目标函数泰勒展开，如下所示：（最后一项为拉格朗日余项，$t \in (0,1)$）

$$f\left(x^*+\Delta x\right)=f\left(x^*\right)+\nabla f(x^*)^T \Delta x+\frac{1}{2} \Delta X^T \nabla^2 f\left(x^*+t \Delta x\right) \Delta x$$

由于一阶导为0，其一阶项也为0，只有二阶项会影响函数值。如果任意的扰动都导致函数值不变或增大，那么二阶导$\nabla^2 f(x^*)$， 一定是半正定矩阵。只有如此，上式的$\nabla^2 f\left(x^*+t \Delta x\right)$才可能也是半正定矩阵，从而保证目标函数值不下降。

以上两条，论述的都是极值点的必要条件，也就是当x^*是极值点时，目标函数的一阶导和二阶导具有的性质。接下来还有一个充分条件。

当一阶导为0，二阶导连续且正定时，x^*是严格局部极值点。这里是正定而非半正定，在该条件下，借助泰勒展开，我们就可以证明此结论。但该结论只是充分条件，而不是充要条件。有些函数，比如$f\left(x\right)=x^4$，在$x^*=0$处是严格极小值点，但其二阶导却不正定，其二阶导是0。

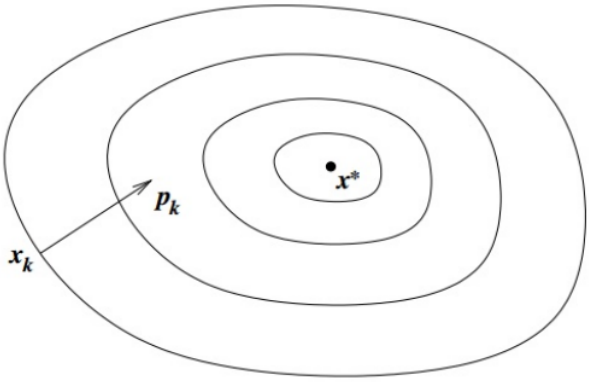
最后，还有一个比较直观的性质，当目标函数是个凸函数时，所有的局部极小值点都是全局极小值点。凸函数可以直观地理解为碗状的函数，它的任意边界都不存在凹陷的区域，因此绝无可能存在多个不相邻的局部极小值。

• 优化策略

（1）线搜索策略

线搜索方法的策略是，先确定优化变量的更新方向p_k，然后在该方向上确定一个最佳的步长α，使得目标函数沿着该方向前进α距离后下降得最多，相当于求解$\min_{\alpha>0} f\left(x_k+\alpha p_k\right)$。

那这个方向到底如何确定呢？看下面这张图：



这是一个二维变量上的优化问题，全局最优点为$x^* = 0$，当前位置为x_k，圆圈是目标函数的等高线。我们或许可以想到，垂直于等高线的方向应该是目前所处位置的最速下降方向，因为这是函数在该点的梯度或者叫导数方向（准确来说是梯度的反方向，但本文对此不做区分）。每次迭代都沿着梯度方向前进，这种线搜索策略就称作最速下降法。

最速下降法求解很简单，只需要求一阶导数即可。但它的缺点也很明显，梯度方向并不一定指向全局极小值，比如上图，p_k并没有直接指向$x^* = 0$，这是由目标函数的性质决定的。如果目标函数的等高线画出来恰好是一个个的同心圆，那显然梯度方向永远指向全局极小值。可惜实际中的目标函数各种各样，最速下降法很可能会走出一条极其曲折的路线，导致其收敛速度很慢，并不实用。

为什么最速下降法找出的方向不是我们想要的方向？归根结底，是因为它其实相当于对目标函数做了一阶近似，把一个非线性函数f线性化成了一次函数，这种近似自然是极其不精确的，只能反映x_k附近极小范围内的函数变化。既然如此，我们不如更进一步，对目标函数做二阶近似，用下面的二次多项式代替原函数：

$$f\left(x_k + p\right) \approx f_k + p^T \Delta f_k + \frac{1}{2} p^T \nabla^2 f_k p \overset{def}{=\!=} m_k\left(p\right)$$

求解使m_k最小的p。二次函数的极小值点比较好求，只需要令其一阶导等于0即可，解得

$$p_k^N = -\nabla^2 f_k^{-1} \nabla f_k$$

这一方法称为牛顿法（Newton method）。与最速下降法相比，牛顿法不需要额外确定步长α，因为对于一次函数来说，步长越大，函数值下降的越多，但过大的步长有可能偏离真实方向，所以需要限制步长，而对于牛顿法，求得的p_k^N已经包含了长度，它直接指向了二次函数的极小值，只要我们相信二次函数对原函数的近似是合理的，这一长度就可以采用。

牛顿法的求解过程依赖于目标函数的二阶导，在很多情况下这是不现实的。复杂多变量函数的二阶导有时很难求解，有时不是正定的，只有半正定性质，这都会影响牛顿法的求解。针对这些问题，研究者们提出了各种改进方案，我们可以看其中的一种，称为拟牛顿法。

拟牛顿法不需要计算目标函数二阶导的解析形式，而是迭代地计算一个近似值B_k，代替式中的$\nabla^2 f_k$。计算B_k的方式也有很多种，只要是对二阶导的近似，且计算简便即可。当然，怎样才算是对二阶导合理的近似，这里不详细展开。

（2）信赖域策略

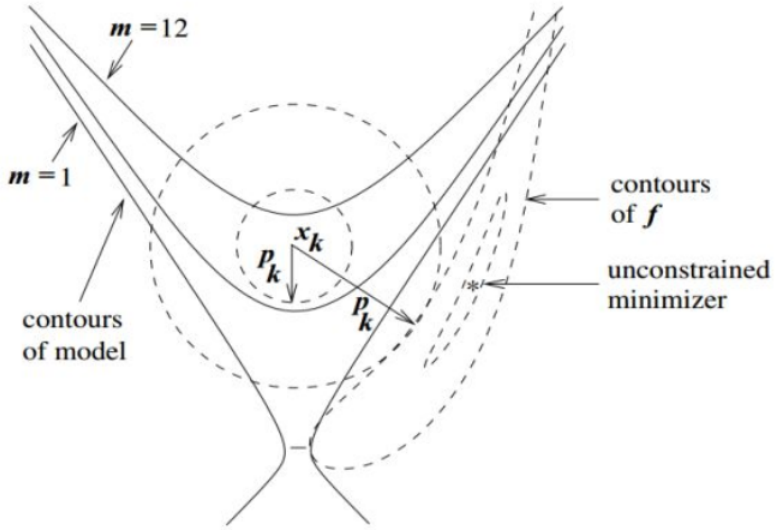
信赖域方法的策略则有所不同，先用另一个简单的模型m_k近似目标函数，然后确定一个信赖域半径Δ_k，在该半径限制的范围内寻找一个使得模型下降最多的更新量p，相当于求解下式：

$$\min_p m_k\left(x_k + p\right), where \|p\|_2 \leq \Delta_k$$

如何确定m_k？通常是用如下的二阶函数：

$$m\left(x_k + p\right) = f_k + p^T \Delta f_k + \frac{1}{2} p^T B_k p$$

当$B_k = \nabla^2 f_k$时，上式为目标函数在x_k处泰勒展开的前三项。确定近似的目标函数后，再选取一个信赖半径，然后在该半径上搜索一个最佳的方向，下图直观地描述了这一过程：



图中，右侧的虚线是实际目标函数的等高线，实线是近似的目标函数的等高线，且最上面一条代表$m=12$，最下面左右对称的两条代表$m=1$，中间的虚线同心圆表示信赖域半径。根据近似的目标函数，可以在外侧的圆上找到一个最优的方向，即图中较长的p_k，然后判断该p_k是否足以使原目标函数f产生足够的下降。如果不行，则在半径更小的圆上再次尝试，即图中内侧较短的p_k。可以发现，每次缩小信赖域半径，找到的优化方向都不一样，这是因为近似的目标函数在不同范围内具有不同的一阶和二阶性质，优化方向如何依赖于这些性质，有待后续进一步的研究。所以，线搜索和信赖域本质上是相通的，只是上层策略略有不同而已。

二．遗传算法

- **算法简介**

遗传算法是模拟达尔文生物进化论的自然选择和遗传学机理的生物进化过程的计算模型，是一种通过模拟自然进化过程搜索最优解的方法。遗传算法是在20世纪六七十年代由美国密歇根大学的 Holland教授创立。

遗传算法是一种仿生算法，最主要的思想是**物竞天择，适者生存**。这个算法很好的模拟了生物的进化过程，保留好的物种，同样一个物种中的佼佼者才会幸运的存活下来。转换到数学问题中，这个思想就可以很好的转化为优化问题，在求解方程组的时候，好的解视为好的物种被保留，坏的解视为坏的物种而淘汰，设置好进化次数以后开始迭代，记录下这些解里面最好的那个，就是要求解的方程组的解。

遗传算法是从代表问题可能潜在的解集的一个**种群**开始的，而一个种群则由经过**基因**编码的一定数目的**个体**组成。每个个体实际上是**染色体**带有特征的实体。染色体作为遗传物质的主要载体，即多个基因的集合，其内部表现（即基因型）是某种基因组合，它决定了个体的形状的外部表现，如黑头发的特征是由染色体中控制这一特征的某种基因组合决定的。

- **模型引入—求解函数极值**

例子：已知一元函数$2\sin x+\cos x$，目标求解该函数的最大值

- **形象理解—“袋鼠跳”问题**

遗传算法中每一条染色体/个体，对应着遗传算法的一个解决方案，一般我们用**适应性函数**来衡量这个解决方案的优劣。所以从一个基因组到其解的适应度形成一个映射。可以把遗传算法的过程看作是一个在多元函数里面求最优解的过程。可以这样想象，这个多维曲面里面有数不清的“山峰”，而这些山峰所对应的就是局部最优解。而其中也会有一个“山峰”的海拔最高的，那么这个就是全局最优解。而遗传算法的任务就是尽量爬到最高峰，而不是陷落在一些小山峰。（另外，值得注意的是遗传算法不一定要找“**最高的山峰**”，如果问题的适应度评价越小越好的话，那么全局最优解就是函数的最小值，对应的，遗传算法所要找的就是“**最深的谷底**”）

既然我们把函数曲线理解成一个一个山峰和山谷组成的山脉。那么我们可以设想所得到的每一个解就是一只袋鼠，我们希望它们不断的向着更高处跳去，直到跳到最高的山峰（尽管袋鼠本身不见得愿意那么做）。所以求最大值的过程就转化成一个“袋鼠跳”的过程。

模拟物竞天择的生物进化过程，通过维护一个潜在解的群体执行了多方向的搜索，并支持这些方向上的信息构成和交换。以面为单位的搜索，比以点为单位的搜索，更能发现全局最优解。

- **数学原理—生物学启发**

遗传算法的实现过程实际上就像自然界的进化过程那样。首先寻找一种对问题潜在解进行“数字化”编码的方案。（建立表现型和基因型的映射关系）然后用随机数初始化一个种群（那么第一批袋鼠就被随意地分散在山脉上），种群里面的个体就是这些数字化的编码。接下来，通过适当的解码过程之后（得到袋鼠的位置坐标），用适应性函数对每一个基因个体作一次适应度评估（袋鼠爬得越高，越是受我们的喜爱，所以适应度相应越高）。用选择函数按照某种规定择优选择（我们要每隔一段时间，在山上射杀一些所在海拔较低的袋鼠，以保证袋鼠总体数目持平。）。让个体基因变异（让袋鼠随机地跳一跳）。然后产生子代（希望存活下来的袋鼠是多产的，并在那里生儿育女）。

遗传算法并不保证能获得问题的最优解，但是使用遗传算法的最大优点在于你不必去了解如何去“找”最优解。（你不必去指导袋鼠向那边跳，跳多远。）而只要简单的“否定”一些表现不好的个体就行了。把那些总是爱走下坡路的袋鼠射杀，这就是遗传算法的精粹。

- **概念解释**

基因型(genotype)：性状染色体的内部表现；

表现型(phenotype)：染色体决定的性状的外部表现，或者说，根据基因型形成的个体的外部表现；

进化(evolution)：种群逐渐适应生存环境，品质不断得到改良。生物的进化是以种群的形式进行的。

适应度(fitness)：度量某个物种对于生存环境的适应程度。

选择(selection)：以一定的概率从种群中选择若干个个体。一般，选择过程是一种基于适应度的优胜劣汰的过程。

复制(reproduction)：细胞分裂时，遗传物质DNA通过复制而转移到新产生的细胞中，新细胞就继承了旧细胞的基因。

交叉(crossover)：两个染色体的某一相同位置处DNA被切断，前后两串分别交叉组合形成两个新的染色体。也称基因重组或杂交；

变异(mutation)：复制时可能（很小的概率）产生某些复制差错，变异产生新的染色体，表现出新的性状。

编码(coding)：DNA中遗传信息在一个长链上按一定的模式排列。遗传编码可看作从表现型到基因型的映射。

解码(decoding)：基因型到表现型的映射。

个体（individual）：指染色体带有特征的实体；

种群（population）：个体的集合，该集合内个体数称为种群的大小。

- **基因编码—编制染色体**

（1）二进制编码法

受到人类染色体结构的启发，我们可以设想一下，假设目前只有“0”，“1”两种碱基，我们也用一条链条把他们有序的串连在一起，因为每一个单位都能表现出 1 bit的信息量，所以一条足够长的染色体就能为我们勾勒出一个个体的所有特征。这就是二进制编码法，染色体大致如下：
010010011011011110111110

（2）浮点数编码

上面的编码方式虽然简单直观，但明显地，当个体特征比较复杂的时候，需要大量的编码才能精确地描述，相应的解码过程（类似于生物学中的DNA翻译过程，就是把基因型映射到表现型的过程。）将过分繁复，为改善遗传算法的计算复杂性、提高运算效率，提出了浮点数编码。染色体大致如下：1.2 –3.3 – 2.0 –5.4 – 2.7 – 4.3

● 物竞天择—适应性评分与及选择函数

（1）物竞——适应度函数

自然界生物竞争过程往往包含两个方面：生物相互间的搏斗与及生物与客观环境的搏斗过程。但在我们这个实例里面，你可以想象到，袋鼠相互之间是非常友好的，它们并不需要互相搏斗以争取生存的权利。它们的生死存亡更多是取决于你的判断。因为你要衡量哪只袋鼠该杀，哪只袋鼠不该杀，所以你必须制定一个衡量的标准。而对于这个问题，这个衡量的标准比较容易制定：袋鼠所在的海拔高度。（因为你单纯地希望袋鼠爬得越高越好。）所以我们直接用袋鼠的海拔高度作为它们的适应性评分。即适应度函数直接返回函数值就行了。

（2）天择——选择函数

自然界中，越适应的个体就越有可能繁殖后代。但是也不能说适应度越高的就肯定后代越多，只能是从概率上来说更多。（毕竟有些所处海拔高度较低的袋鼠很幸运，逃过了你的眼睛。）那么我们怎么来建立这种概率关系呢？我们可以利用一种常用的选择方法——轮盘赌选择法。

轮盘赌选择法是依据个体的适应度值计算每个个体在子代中出现的概率，并按照此概率随机选择个体构成子代种群。轮盘赌选择策略的出发点是适应度值越好的个体被选择的概率越大。因此，在求解最大化问题的时候，我们可以直接采用适应度值来进行选择。但是在求解最小化问题的时候，我们必须首先将问题的适应度函数进行转换，以将问题转化为最大化问题。下面给出最大化问题求解中遗传算法轮盘赌选择策略的一般步骤：

(1) 将种群中所有个体的适应度值叠加，得到总适应度值==1

(2) 每个个体的适应度值除以总适应度值得到个体被选择的概率

(3) 计算个体的累积概率以构造一个轮盘。

(4) 轮盘选择：产生一个[0,1]区间内的随机数，若该随机数小于或等于个体的累积概率且大于个体1的累积概率，选择个体进入子代种群。

重复步骤(4)次，得到的个体构成新一代种群。

● 遗传变异——基因重组（交叉）与基因突变

基因重组与基因突变是**使得子代不同于父代的根本原因**。对于这两种遗传操作，二进制编码和浮点型编码在处理上有很大的差异，其中二进制编码的遗传操作过程，比较类似于自然界里面的过程，下面将分开讲述。需要注意的是，子代不一定优于父代，只有经过自然的选择后，才会出现子代优于父代的倾向。

（1）基因重组/交叉

①二进制编码

二进制编码的基因交换过程非常类似高中生物中所讲的同源染色体的联会过程——随机把其中几个位于同一位置的编码进行交换，产生新的个体。

②浮点数编码

如果一条基因中含有多个浮点数编码，那么也可以用跟上面类似的方法进行基因交叉，不同的是进行交叉的基本单位不是二进制码，而是浮点数。而如果对于单个浮点数的基因交叉，就有其它不同的重组方式了，比如中间重组：随机产生就能得到介于父代基因编码值和母代基因编码值之间的值作为子代基因编码的值。比如5.5和6交叉，产生5.7，5.6。

考虑到“袋鼠跳”问题的具体情况——袋鼠的个体特征仅仅表现为它所处的位置。可以想象，同一个位置的袋鼠的基因是完全相同的，而两条相同的基因进行交叉后，相当于什么都没有做，所以我们不打算在这个例子里面使用交叉这一个遗传操作步骤。（当然硬要这个操作步骤也不是不行的，你可以把两只异地的袋鼠捉到一起，让它们交配，然后产生子代，再把它们送到它们应该到的地方。）

（2）基因突变

①二进制编码

基因突变过程：基因突变是染色体的某一个位点上基因的改变。基因突变使一个基因变成它的等位基因，并且通常会引起一定的表现型变化。正如上面所说，二进制编码的遗传操作过程和生物学中的过程非常相类似，基因串上的“ 0”或“ 1”有一定几率变成与之相反的“ 1”或“ 0”。例如下面这串二进制编码：101101001011001，经过基因突变后，可能变成以下这串新的编码：001101011011001

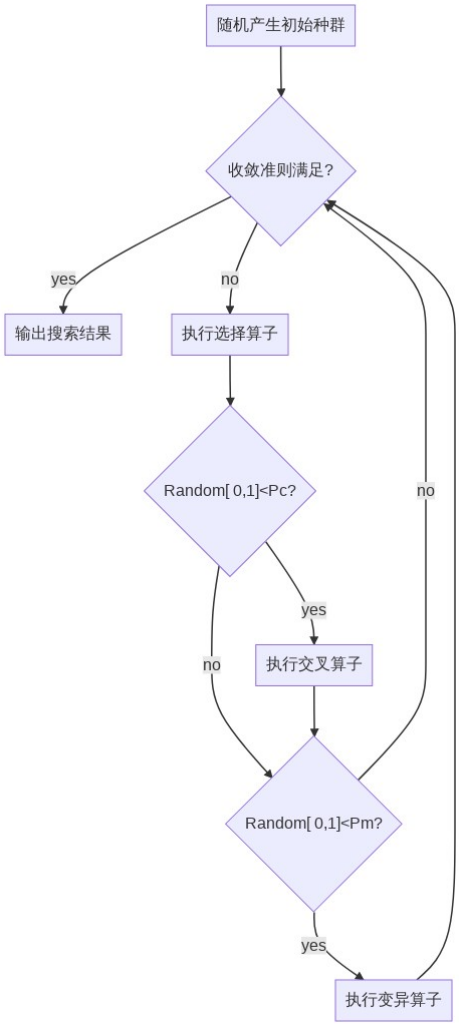
②浮点数编码

浮点型编码的基因突变过程一般是对原来的浮点数增加或者减少一个小随机数。比如原来的浮点数串如下：1.2,3.4,5.1, 6.0, 4.5变异后，可能得到如下的浮点数串：1.3,3.1,4.9, 6.3, 4.4

当然，这个小随机数也有大小之分，我们一般管它叫“步长”。（想想“袋鼠跳”问题，袋鼠跳的长短就是这个步长。）一般来说步长越大，开始时进化的速度会比较快，但是后来比较难收敛到精确的点上。而小步长却能较精确的收敛到一个点上。

很多时候为了加快遗传算法的进化速度，而又能保证后期能够比较精确地收敛到最优解，会采取动态改变步长的方法。

至此，我们阐述了遗传算法的各个步骤，流程图如下：



三．编程实现

- **实验环境**

C++源码采用标准库bits/stdc++.h，编译器 gcc 4.9.2，编译指令-std=c++11

python源码采用库numpy处理数组对象，使用库pandas进行数据分析，使用库random产生随机数序列，使用库mathmatplotlib进行绘图，使用库math生成数学表达式，编译器python 3.9.0，编译指令python city.py

- **旅行商问题**

（1）问题描述

给定n个城市和两两城市之间的距离，要求确定一条经过各城市当且仅当一次的最短路径。其图论描述为：给定图G=(V,A)，其中V为顶点集，A为各顶点相互连接组成的边集，一直各顶点间的连接距离，要求确定一条长度最短的Hamilton回路，即遍历所有顶点当且仅当一次的最短回路。

（2）初始化

初始化适应度数组，突变率，重组率，种群大小，迭代次数，加载城市数据，计算绘图的XY边界。创建种群：先生成等间距基因，再打乱加入种群数组中。计算初始的适应度数组：取其中一条基因，计算此基因优劣即距离长短，当前最优距离除以当前个体距离，越近适应度越高，最优适应度为1。

```
def __init__(self, c_rate, m_rate, pop_size, ga_num):
    self.fitness = np.zeros(self.pop_size)
    self.c_rate = c_rate
    self.m_rate = m_rate
    self.pop_size = pop_size
    self.ga_num = ga_num

def init(self):
    tsp = self
    tsp.load_Citys()  # 加载城市数据
    tsp.pop = tsp.creat_pop(tsp.pop_size)  # 创建种群
    tsp.fitness = tsp.get_fitness(tsp.pop)  # 计算初始种群适应度
    tsp.dw.bound_x = [np.min(tsp.citys[:, 0]), np.max(tsp.citys[:, 0])]  # 计算绘
图时的X界
    tsp.dw.bound_y = [np.min(tsp.citys[:, 1]), np.max(tsp.citys[:, 1])]  # 计算绘
图时的Y界
    tsp.dw.set_xybound(tsp.dw.bound_x, tsp.dw.bound_y)  # 设置边界

def creat_pop(self, size):
    pop = []
    for i in range(size):
        gene = np.arange(self.citys.shape[0])  # 问题的解，基因，种群中的个体：
[0, ..., city_size]
        np.random.shuffle(gene)  # 打乱数组[0, ..., city_size]
        pop.append(gene)  # 加入种群
    return np.array(pop)
```

（3）计算适应度

交换基因数组中任意两个值组成的解集是邻域，计算领域内所有可能的适应度。

```
def get_local_fitness(self, gen, i):
    """
    计算地i个城市的邻域
```

```
    交换基因数组中任意两个值组成的解集：称为邻域。计算领域内所有可能的适应度
    :param gen:城市路径
    :param i:第i城市
    :return:第i城市的局部适应度
    """
    di = 0
    fi = 0
    if i == 0:
        di = self.ct_distance(self.citys[gen[0]], self.citys[gen[-1]])
    else:
        di = self.ct_distance(self.citys[gen[i]], self.citys[gen[i - 1]])
    od = []
    for j in range(self.city_size):
        if i != j:
            od.append(self.ct_distance(self.citys[gen[i]], self.citys[gen[i -
1]]))
    mind = np.min(od)
    fi = di - mind
    return fi
```

(4) 基因突变

通过随机翻转i到j的基因片段实现基因突变

```
def mutate(self, gene):
    """突变"""
    if np.random.rand() > self.m_rate:
        return gene
    index1 = np.random.randint(0, self.city_size - 1)
    index2 = np.random.randint(index1, self.city_size - 1)
    newGene = self.reverse_gen(gene, index1, index2)
    if newGene.shape[0] != self.city_size:
        print('m error')
        return self.creat_pop(1)
    return newGene

def reverse_gen(self, gen, i, j):
    # 函数：翻转基因中i到j之间的基因片段
    if i >= j:
        return gen
    if j > self.city_size - 1:
        return gen
    parent1 = np.copy(gen)
    tempGene = parent1[i:j]
    newGene = []
    p1len = 0
    for g in parent1:
        if p1len == i:
            newGene.extend(tempGene[::-1])  # 插入基因片段
        if g not in tempGene:
            newGene.append(g)
        p1len += 1
    return np.array(newGene)
```

(5) 基因重组

随机交叉父代基因片段

```
def cross(self, parent1, parent2):
    """交叉p1,p2的部分基因片段"""
    if np.random.rand() > self.c_rate:
        return parent1
    index1 = np.random.randint(0, self.city_size - 1)
    index2 = np.random.randint(index1, self.city_size - 1)
    tempGene = parent2[index1:index2]  # 交叉的基因片段
    newGene = []
    p1len = 0
    for g in parent1:
        if p1len == index1:
            newGene.extend(tempGene)  # 插入基因片段
        if g not in tempGene:
            newGene.append(g)
        p1len += 1
    newGene = np.array(newGene)

    if newGene.shape[0] != self.city_size:
        print('c error')
        return self.creat_pop(1)
        # return parent1
    return newGene
```

(6) 极值优化

加入极值优化算法，而不使用传统遗传算法，这样有效避免收敛到局部最优而错过全局最优，可以在整个基因的领域内寻找一个最佳变换以更新基因。

```
def EO(self, gen):
    # 极值优化，传统遗传算法性能不好，这里混合EO
    # 其会在整个基因的领域内，寻找一个最佳变换以更新基因
    local_fitness = []
    for g in range(self.city_size):
        f = self.get_local_fitness(gen, g)
        local_fitness.append(f)
    max_city_i = np.argmax(local_fitness)
    maxgen = np.copy(gen)
    if 1 < max_city_i < self.city_size - 1:
        for j in range(max_city_i):
            maxgen = np.copy(gen)
            jj = max_city_i
            while jj < self.city_size:
                gen1 = self.exechange_gen(maxgen, j, jj)
                d = self.gen_distance(maxgen)
                d1 = self.gen_distance(gen1)
                if d > d1:
                    maxgen = gen1[:]
                jj += 1
    gen = maxgen
    return gen
```

(7) 选择

选择种群，优胜劣汰，可以有两中选择策略，一是替换所有低于平均适应度的，二是轮盘赌策略，使适应度低的替换的概率更大。

```
def select_pop(self, pop):
    # 选择种群，优胜劣汰，策略1：低于平均的要替换改变
    best_f_index = np.argmax(self.fitness)
    av = np.median(self.fitness, axis=0)
    for i in range(self.pop_size):
        if i != best_f_index and self.fitness[i] < av:
            pi = self.cross(pop[best_f_index], pop[i])
            pi = self.mutate(pi)
            pop[i, :] = pi[:]
    return pop

def select_pop2(self, pop):
    # 选择种群，优胜劣汰，策略2：轮盘赌，适应度低的替换的概率大
    probility = self.fitness / self.fitness.sum()
    idx = np.random.choice(np.arange(self.pop_size), size=self.pop_size,
replace=True, p=probility)
    n_pop = pop[idx, :]
    return n_pop
```

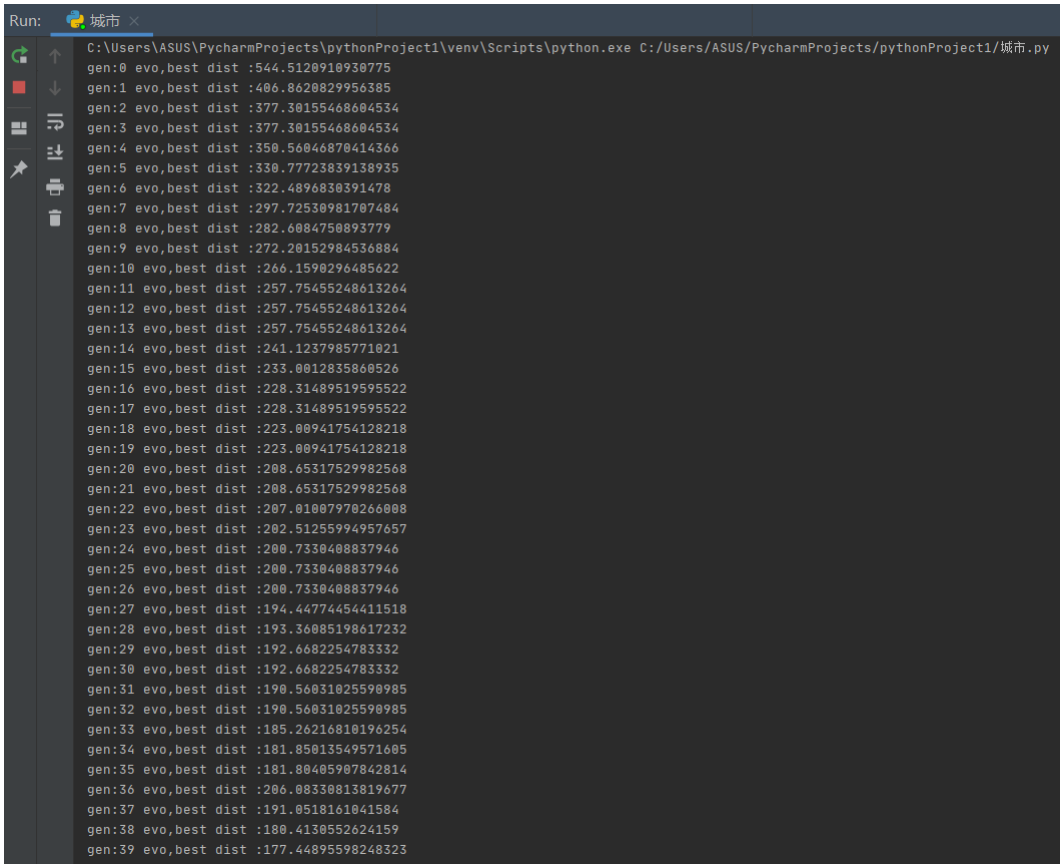
(8) 进化

反复迭代进化种群，先选择淘汰种群，计算种群适应度，再随机交叉种群中第j,r个体的基因，突变种群中第j个体的基因，进行极值优化，防止收敛局部最优，记录最优值和最优个体基因，进行绘图。

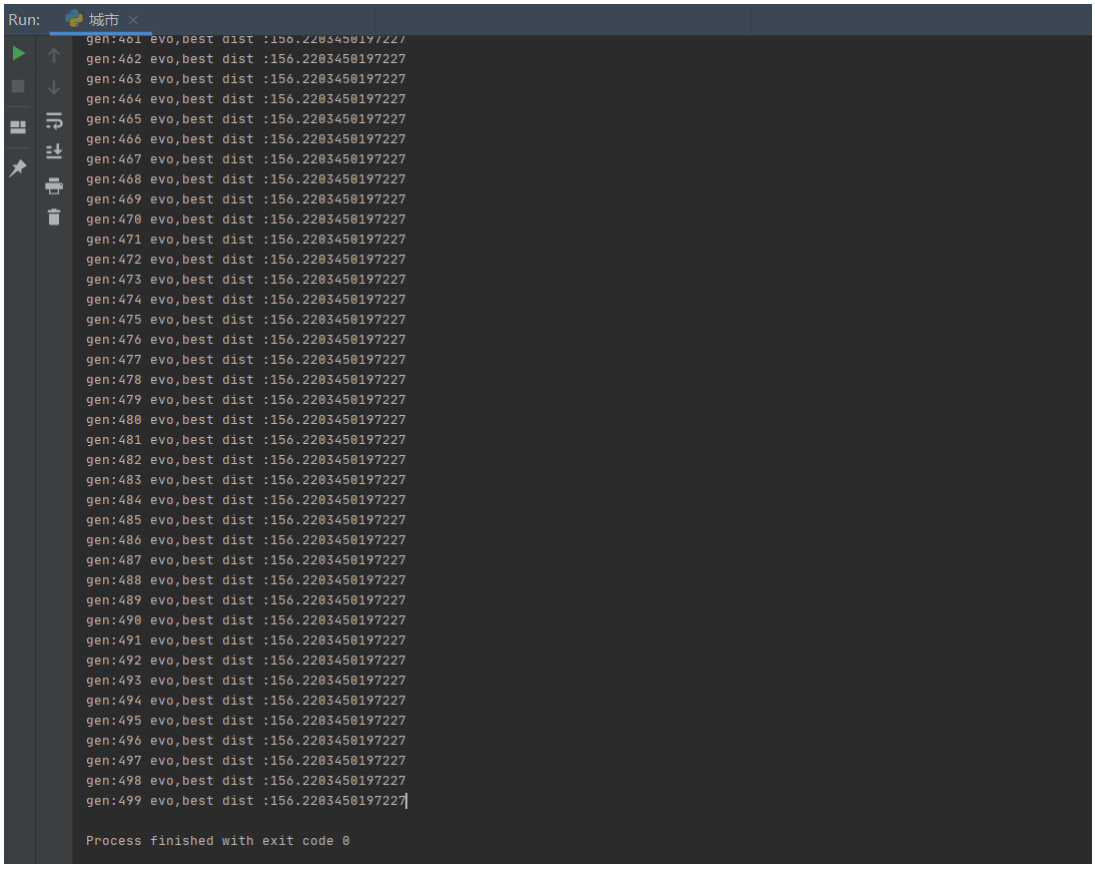
```
def evolution(self):
    # 主程序：迭代进化种群
    tsp = self
    for i in range(self.ga_num):
        best_f_index = np.argmax(tsp.fitness)
        worst_f_index = np.argmin(tsp.fitness)
        local_best_gen = tsp.pop[best_f_index]
        local_best_dist = tsp.gen_distance(local_best_gen)
        if i == 0:
            tsp.best_gen = local_best_gen
            tsp.best_dist = tsp.gen_distance(local_best_gen)

        if local_best_dist < tsp.best_dist:
            tsp.best_dist = local_best_dist  # 记录最优值
            tsp.best_gen = local_best_gen  # 记录最个体基因
            # 绘图
            tsp.dw.ax.cla()
            tsp.re_draw()
            tsp.dw.plt.pause(0.001)
        else:
            tsp.pop[worst_f_index] = self.best_gen
        print('gen:%d evo,best dist :%s' % (i, self.best_dist))

        tsp.pop = tsp.select_pop(tsp.pop)  # 选择淘汰种群
        tsp.fitness = tsp.get_fitness(tsp.pop)  # 计算种群适应度
        for j in range(self.pop_size):
            r = np.random.randint(0, self.pop_size - 1)
            if j != r:
                tsp.pop[j] = tsp.cross(tsp.pop[j], tsp.pop[r])  # 交叉种群中第j,r个
体的基因
                tsp.pop[j] = tsp.mutate(tsp.pop[j])  # 突变种群中第j个体的基因
        self.best_gen = self.EO(self.best_gen)  # 极值优化，防止收敛局部最优
        tsp.best_dist = tsp.gen_distance(self.best_gen)  # 记录最优值
```

(9) 运行结果

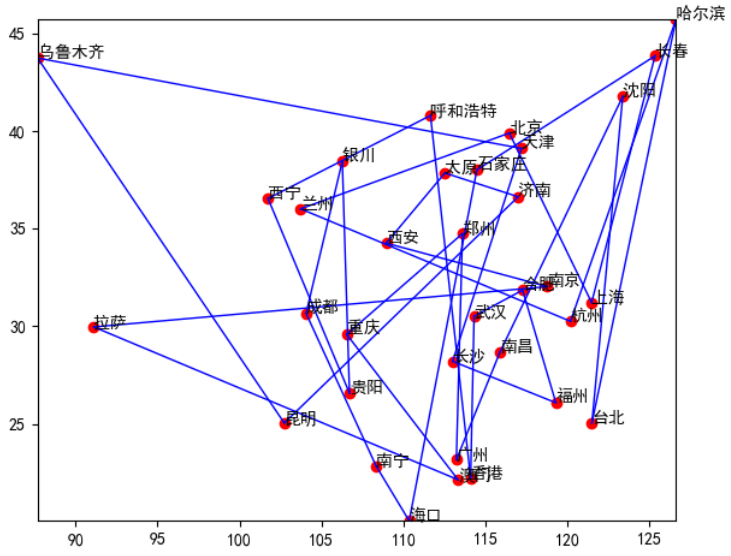
前面代数种群最优个体信息



后面代数种群最优个体信息



如下图所示，绘制出了34个省会城市最短的Hamilton回路，其最短路径长度为156.2203450197227（以球坐标下的经纬度为单位）

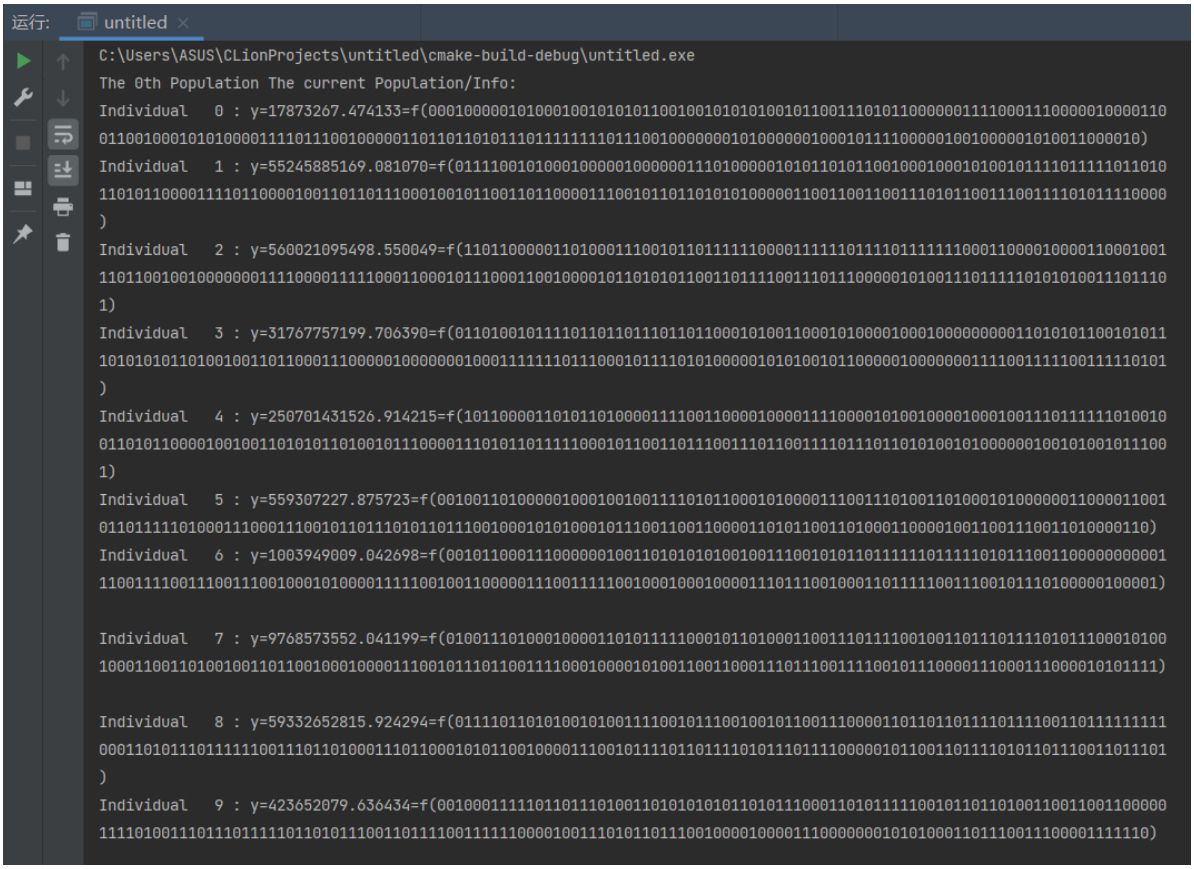


- **函数极值问题**

给定一个函数$f\left(x\right)$，求出其局部极值或全定义域上的最值，算法流程与上述TSP问题类似，详细过程可见源代码注释。

（1）C++实现$f\left(x\right)=x^4-x^3+x^2-x$最小值的求解

第0代种群中个体信息：



第100代种群个体信息：

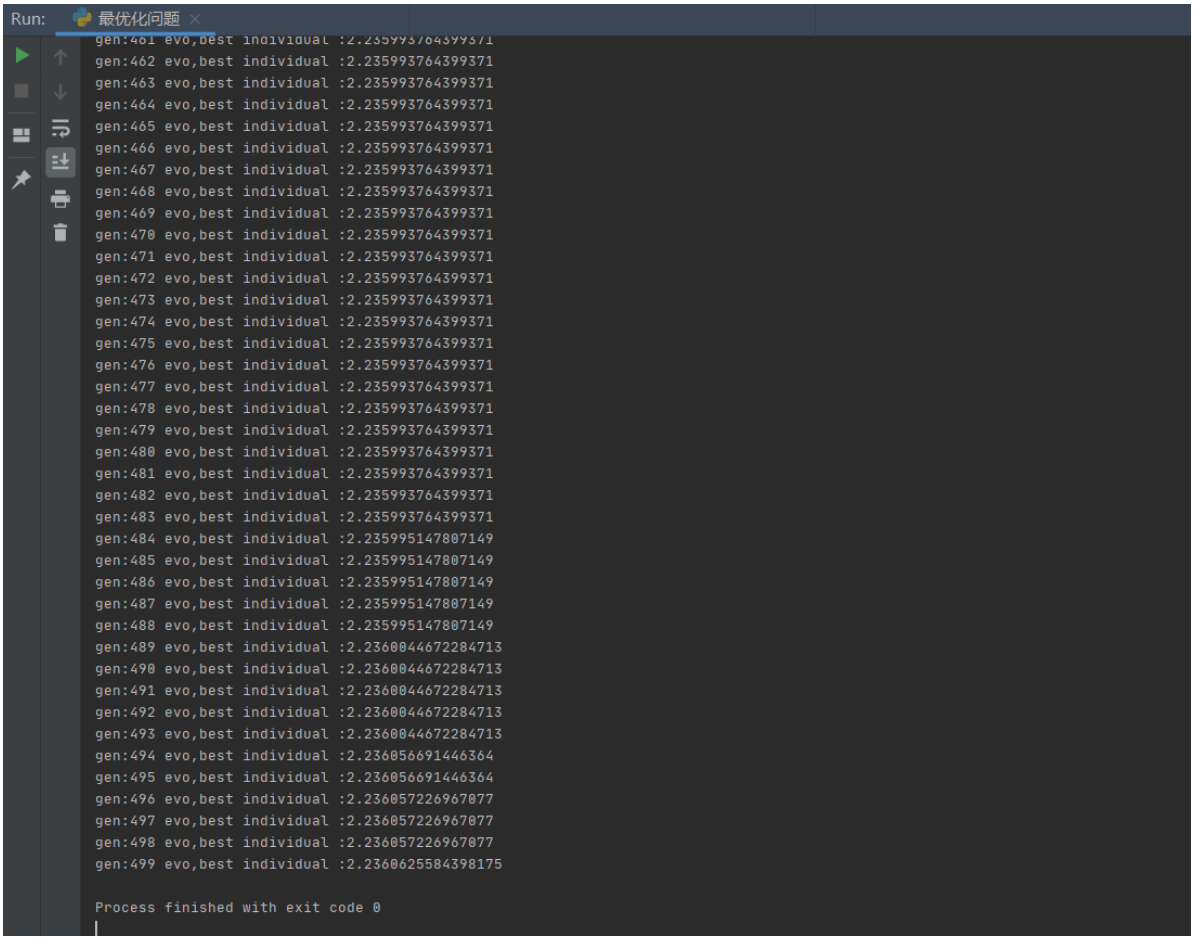
```
The 100th Population The Final Population/Info:
Individual   0 : y= -0.303702=f(00000000000111011111110111111100011001100000110011101001101010101001010010000001010110101110100111000111101011111110001001011000110011111100101011011000101111010000010001111110101001110111010011110)
Individual   1 : y= -0.303702=f(00000000000111011111110111111100011001100000110011101001101010101001010010000001101011010111010011100011110101111110001001011000110011111100101011011000101111010000010001111110101001110111010011110)
Individual   2 : y= -0.303702=f(00000000000111011111110111111100001001100000110001100001101010101001010010000001100011101101001110001111010001111000101111010000010001111110101001110111010011110)
Individual   3 : y= -0.303702=f(00000000000111011111110111111100001001100000110001100001101010101001010010000001100011101101010100101100110110101010)
Individual   4 : y= -0.303702=f(00000000000111011111110111111100001001100000110001100001101010101001010010000001100011101101010100101100110110101010)
Individual   5 : y= -0.303702=f(00000000000111011111110111111100001001100000110001100001101010101001010010000001100011101101010100101100110110101010)
Individual   6 : y= -0.303702=f(00000000000111011111110111111100001001100000110001100001101010101001010010000001100011101101010100101100110110101010)
Individual   7 : y= -0.303702=f(00000000000111011111110111111100001001100000110001100001101010101001011010000001100011101101010100101100110110101010)
Individual   8 : y= -0.303702=f(00000000000111011111110111111100001001100000110001100001101010101001010010000001100011101101010100101100110110101010)
Individual   9 : y= -0.303702=f(00000000000111011111110111111100001001100000110001100001101010101001010010000001100011101101010100101100110110101010)
```

（2）python实现$f(x)=2\sin x+\cos x$最大值的求解

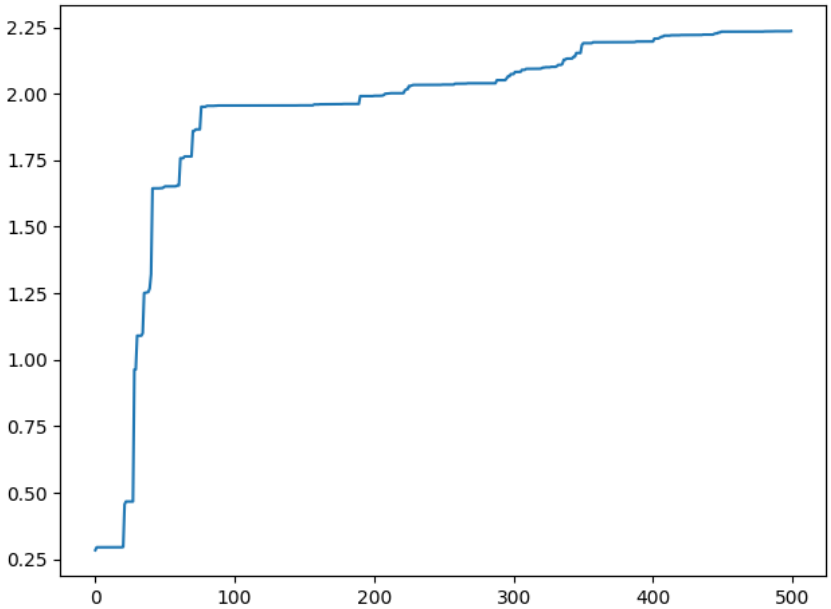
前面代数种群最优个体信息

```
Run:  最优化问题
C:\Users\ASUS\PycharmProjects\pythonProject1\venv\Scripts\python.exe C:/Users/ASUS/PycharmProjects/pythonProject1/最优化问题.py
gen:0 evo,best individual :0.9472974573053611
gen:1 evo,best individual :0.9472974573053611
gen:2 evo,best individual :0.9473167740662871
gen:3 evo,best individual :0.9520853707075426
gen:4 evo,best individual :0.9670323790357795
gen:5 evo,best individual :1.025676313500866
gen:6 evo,best individual :1.1024902809982895
gen:7 evo,best individual :1.1024902809982895
gen:8 evo,best individual :1.1024902809982895
gen:9 evo,best individual :1.2509573317007017
gen:10 evo,best individual :1.2509573317007017
gen:11 evo,best individual :1.2509573317007017
gen:12 evo,best individual :1.2509573317007017
gen:13 evo,best individual :1.5241377606304674
gen:14 evo,best individual :1.5241377606304674
gen:15 evo,best individual :1.58362538744979
gen:16 evo,best individual :1.58362538744979
gen:17 evo,best individual :1.58362538744979
gen:18 evo,best individual :1.58362538744979
gen:19 evo,best individual :1.58362538744979
gen:20 evo,best individual :1.58362538744979
gen:21 evo,best individual :1.58362538744979
gen:22 evo,best individual :1.58362538744979
gen:23 evo,best individual :1.58362538744979
gen:24 evo,best individual :1.58362538744979
gen:25 evo,best individual :1.58362538744979
gen:26 evo,best individual :1.5836404426852708
gen:27 evo,best individual :1.5837458253006063
gen:28 evo,best individual :1.5837458253006063
gen:29 evo,best individual :1.5837458253006063
gen:30 evo,best individual :1.5855212124069729
gen:31 evo,best individual :1.5855512864155998
gen:32 evo,best individual :1.5855512864155998
gen:33 evo,best individual :1.5855512864155998
gen:34 evo,best individual :1.5855512864155998
gen:35 evo,best individual :1.5855512864155998
gen:36 evo,best individual :1.5855512864155998
gen:37 evo,best individual :1.5855512864155998
gen:38 evo,best individual :1.5855512864155998
gen:39 evo,best individual :1.5855512864155998
```

后面代数种群最优个体信息



种群进化过程中最优个体变化动态图（横坐标为代数，纵坐标为最优个体表现型）



由此求出最大值为2.2360625584398175

四．总结与思考

在求解这些最优化问题的过程中我也尝试了其他算法如模拟退火算法，蚁群搜索算法等等，从实际建模角度，编程角度以及运行结果角度来看遗传算法和其他算法相比具有着以下的优点：

1. 适用性比较广，与问题的研究领域无关，都有较为广泛的应用。

2. 搜索从群体出发，具有潜在的并行性，可以进行多个个体的同时比较。

3. 搜索使用评价函数启发，优化过程简单。

4. 使用概率机制进行迭代，具有随机性。

5. 具有可扩展性，容易与其他算法结合，比如使用到的EO算法。

遗传算法在解决这些问题时也有一些显著的缺点：

1. 遗传算法的编程实现比较复杂,首先需要对问题进行编码,找到最优解之后还需要对问题进行解码。

2. 三个算子的实现也有许多参数,如交叉率和变异率,并且这些参数的选择严重影响解的品质,而目前这些参数的选择大部分是依靠经验。

3. 难以利用上一次迭代结果的反馈信息,故算法的搜索速度比较慢，要得要较精确的解需要较多的训练时间。

4. 算法对初始种群的选择有一定的依赖性，这需要结合一些启发算法进行改进。

5. 算法的并行机制的潜在能力没有得到充分的利用，在之前问题的求解过程中实际上只使用了一个种群进行反复迭代进化。

五．致谢

桃李不言，下自成蹊。首先，我想要感谢我的数学分析老师李娟，在两个学期的数学分析课程中，她不仅教会了我进入大学以来最先接触也是最重要的一门数学课程，更让我汲取到了分析的数学思想，以此发现最优化问题的数学求解的方法。

先生之风，山高水长。其次，我想要感谢“面向科学问题求解的编程实践”孙广中老师，老师儒雅谦和，课上阐述的一些算法原理和实例对我深有启发，一定程度上帮助了我完成编程实现。

最后我想要感谢我的同学还有课程的助教，在此之前我对如何查阅相关资料如何撰写一篇论文一无所知，在他们的帮助下才得以顺利得完成这篇论文的写作。

以梦为马，不负韶华。学习与成长的道路还很长，感谢一直不曾放弃的自己，故事不会停留在这一篇，我会保持清澈与若谷的心继续前行。

六．参考文献

[1]杨光慧,杨辉.有限理性下参数最优化问题解的稳定性.运筹学学报,2016年04期.

[2]魏旭媛.非方阵范数最优化问题的光滑化算法.复旦大学,硕士论文,2012年.

[3]杜杰.一类全局最优化问题的最优性条件及凸化方法研究.青岛科技大学,硕士论文,2016年.

[4]黄小津.线性约束的最优化问题和非线性方程组的无导数立方正则方法.清华大学,博士论文,2017年.

[5] HOLLAND J H． Adaptation in natural and artificial systems: an introductory analysis with applications to biology，control，and artificial intelligence［M］．2nd ed．Cambridge: MIT Press，1992．

[6]葛继科,邱玉辉,吴春明,蒲国林.遗传算法研究综述[J].计算机应用研究,2008(10):2911-2916.

[7]雷德明.多维实数编码遗传算法[J].控制与决策,2000(02):239-241.

[8]臧文科. DNA遗传算法的集成研究与应用[D].山东师范大学,2018.

[9]马永杰,云文霞.遗传算法研究进展[J].计算机应用研究,2012,29(04):1201-1206+1210.

[10]吉根林.遗传算法研究综述[J].计算机应用与软件,2004(02):69-73.

[11] Nix A E , Vose M D . Modeling genetic algorithms with Markov chains[J]. Annals of Mathematics & Artificial Intelligence, 1992, 5(1):79-88.

[12杨新武,杨丽军.基于交叉模型的改进遗传算法[J].控制与决策,2016,31(10):1837-1844.