现代优化计算方法实验一

姓名：卢韬

学号：20194127

班级：自动化1903

实验一

目录

[随机数生成 2](#_Toc101566100)

[问题1. 2](#_Toc101566101)

[方法一：线性同乘余法 2](#_Toc101566102)

[方法二：梅森素数法 2](#_Toc101566103)

[方法三：使用python的random库 3](#_Toc101566104)

[问题2 3](#_Toc101566105)

[法一：逆变法 3](#_Toc101566106)

[法二：numpy中的随机函数 4](#_Toc101566107)

[问题3. 4](#_Toc101566108)

[4.禁忌搜索算法求解TSP问题 5](#_Toc101566109)

[问题分析 5](#_Toc101566110)

[代码细节 6](#_Toc101566111)

[面向更广泛的测试样例：ATSP问题转化TSP 6](#_Toc101566112)

[加速算法运行效率 7](#_Toc101566113)

[超参数解耦合 8](#_Toc101566114)

[训练过程的可视化 9](#_Toc101566115)

[消融实验 9](#_Toc101566116)

[算法鲁棒性 9](#_Toc101566117)

[学习率衰减 10](#_Toc101566118)

[抖动 12](#_Toc101566119)

[接受非最优的第n个解 12](#_Toc101566120)

[其他实验 14](#_Toc101566121)

算法在jupyter notebook中完成，为了保证运行环境一致，这里给出anaconda的虚拟环境，使用pip install -r requirements.txt 即可。或者可以在colab的线上分享中快速运行。

线上环境：

https://drive.google.com/file/d/1WkCjeRRS7c865NCn8-WB9OvwsBAs\_f8J/view?usp=sharing

# 随机数生成

## 问题1.

想要生成在0，1区间得均匀分布，可以先使用随机整数的生成方式，然后将其归一化到0，1区间。

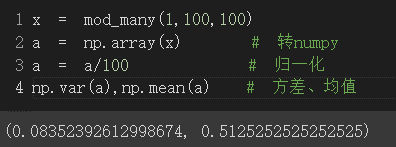
### 方法一：线性同乘余法

a和b同时被M除后取余数相等则为同余，有公式

(a +／- b) mod M = （(a mod M) +／- (b mod M)） mod m

a\*b mod M = ((a mod M) \* (b mod M) mod M

生成（0，100）得随机数后，归一化到（0，1）区间



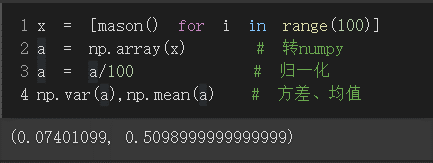
同乘余并归一化法

### 方法二：梅森素数法

梅森法是通过线性反馈移位寄存器来生成随机数的.线性反馈移位寄存器-LFSR，是指给定前一状态的输出，将该输出的线性函数再用作输入的移位寄存器也就是对寄存器的某些位进行异或操作后作为输入，再对寄存器中的各比特进行整体移位.

算法详细介绍：<https://www.cnblogs.com/liinux/p/9981436.html>

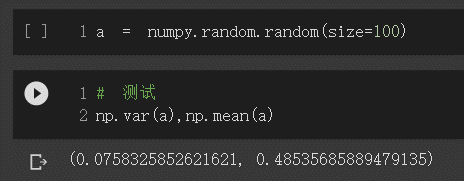
利用梅森法得到（0，100）得随机数后，归一化到（0，1）区间。



梅森素数法

### 方法三：使用python的random库

python 中的random库中，经常会用到如randint之类的方法来生成一定范围内的随机数.这之中主要用到的方法步骤为->指定一个特定的seed种子，根据seed再通过特定的随机数产生方法，就可以做到在[0，1]这个范围中取到随机分布的随机数，然后一定的变换，我们就可以得到一定范围内的随机数了.



**random库**

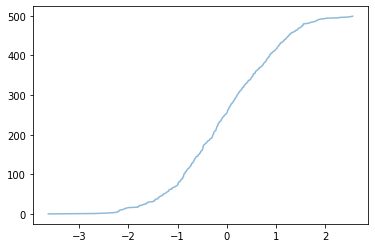
衡量随机数质量的测试网站如下，由于网络原因，加载太慢，未能实验。

<http://simul.iro.umontreal.ca/testu01/tu01.html>

## 问题2

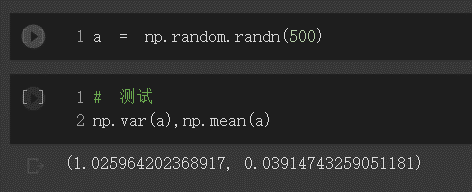
### 法一：逆变法

参考实验二，只需要将CDF换成正态分布即可。



CDF图

### 法二：numpy中的随机函数

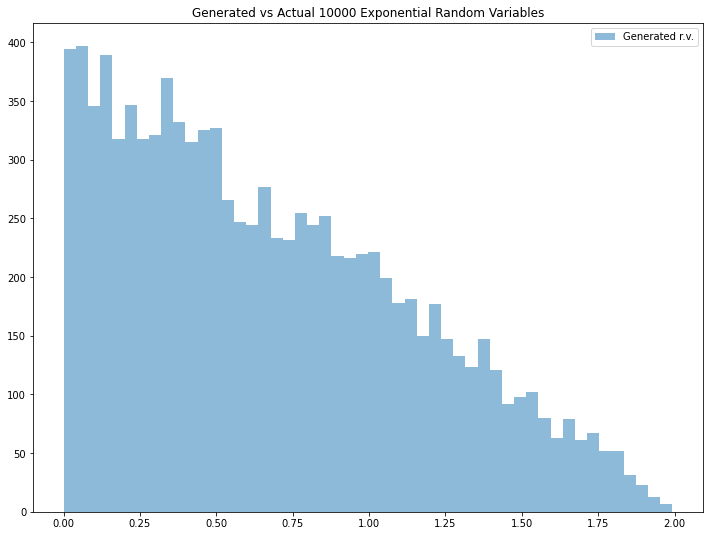


方差、期望

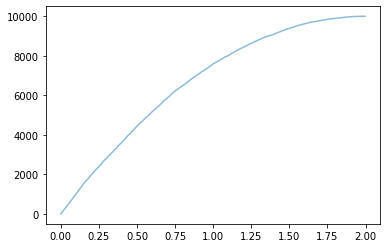
## 问题3.

根据逆变法的算法由概率密度函数（PDF）f（x）= 1-0.5\*x ，积分得概率分布函数（CDF）U=F（x）=x-0.25\*x^2。取逆变换，得x=2-2\*sqrt（1-\*U）。然后生成100个U（0，1）分布的数，即可模拟如图分布。

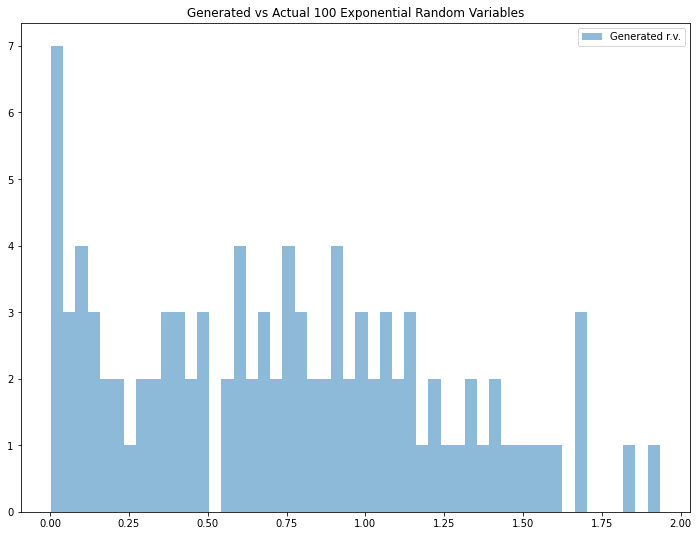
观察到随机次数越多，大数定律越明显。



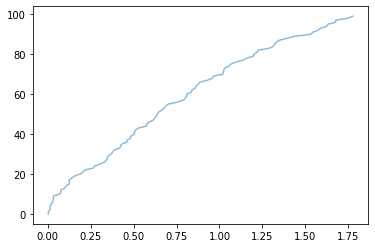
10000次实验的PDF



10000次实验的CDF



100次实验（由于统计步长问题显示不好）



100次实验的CDF

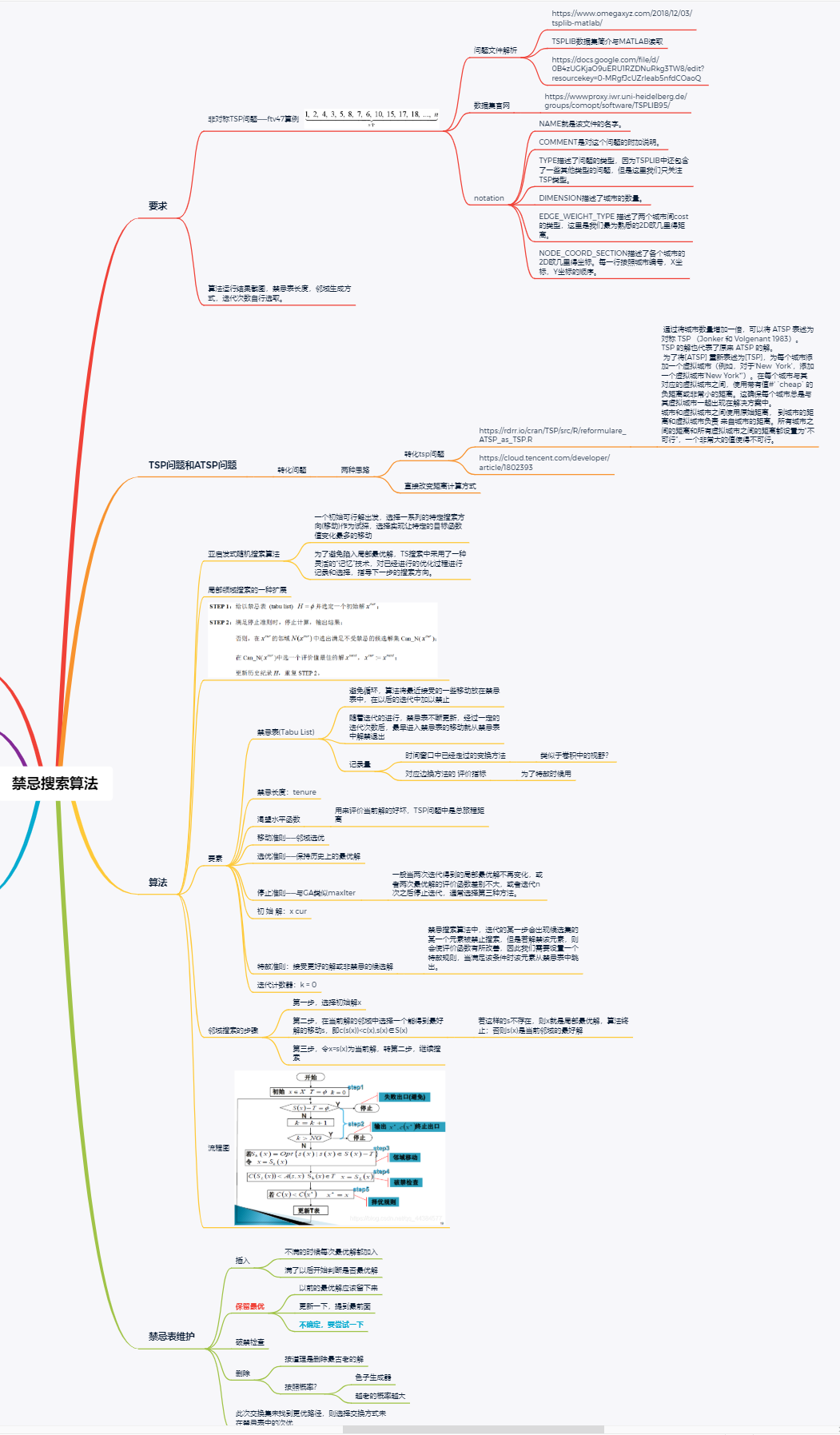
计算其方差得 0.19742496408307275，均值为 0.6403954076815396

# 4.禁忌搜索算法求解TSP问题

## 问题分析

数学问题转化：首先探究了ATSP问题，通过文献查找发现了ATSP转换TSP问题的方法（Jonker 和 Volgenant 1983）。使用numpy可以方便的进行矩阵转置来完成这一转化。此时问题有两个思路，转换为TSP或者按照ATSP的非对称路径来读取。我将两种思路都实现了，最终对比发现后者在储存、速度上都更好。

代码需求分析：根据所学内容，列出算法步骤，关键中间变量。



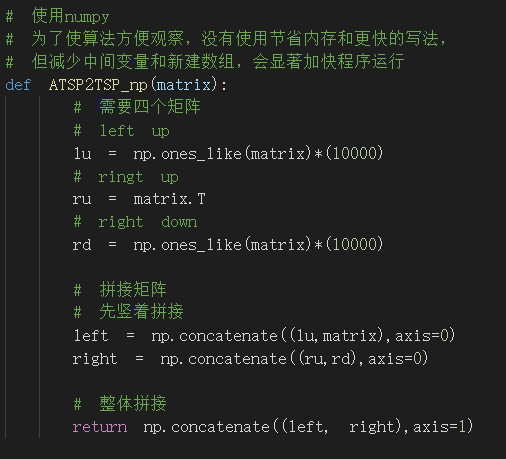
实验分析步骤（缩略图，原图在文件夹中）

## 代码细节

本实验首次完成时间较早，但是算法性能始终不好，因此反复尝试了一些技巧来提升性能。从最初的2500+（最好）到现在的1900+（最好）。本次实验的代码完全由本人完成，没有借鉴任何代码。

### 面向更广泛的测试样例：ATSP问题转化TSP

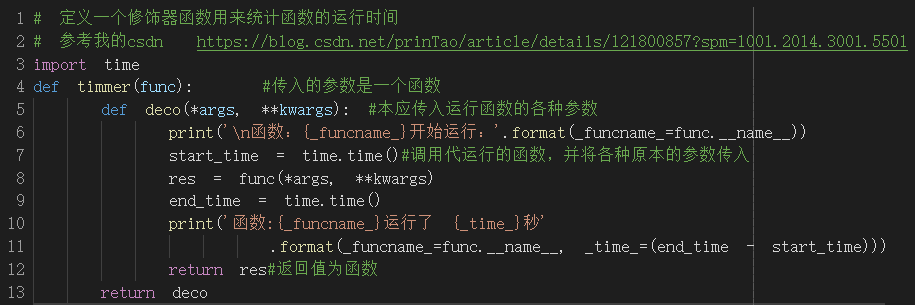
针对实验指定的ATSP问题，经过查阅文献，找到了将ATSP转换为TSP问题一般方法（但是由于矩阵翻倍，算法运行速度很慢）。因此算法得以在不同的测试样例之间快速切换。同时也使用warning、assert等错误处理机制，便于切换测试样例带来问题的排查。



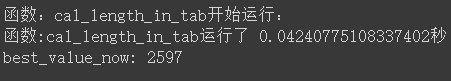
转化函数

### 加速算法运行效率

由于算法从零开始实现，因此数据结构的选取、迭代过程中中间数据的处理等细节有诸多实现方法。因此在完成实验内容后，我利用python中的修饰器技巧，通过函数包装来快速获取各个函数的运行时间，便于找到耗时的瓶颈位置。只需要在写好的函数上加入@timmer即可在调用过程中自动记录时间。此方法来源于我的个人博客，在初学python时候记录的笔记。

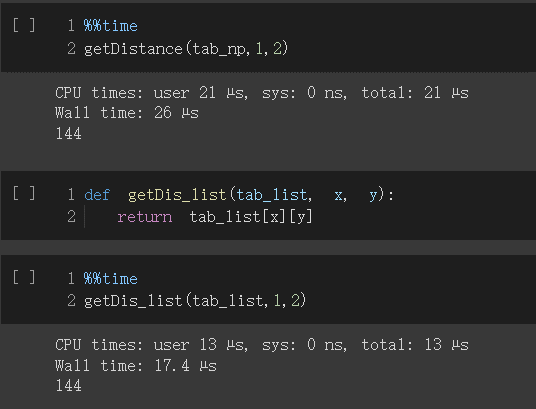


修饰器@timmer



修饰器效果图

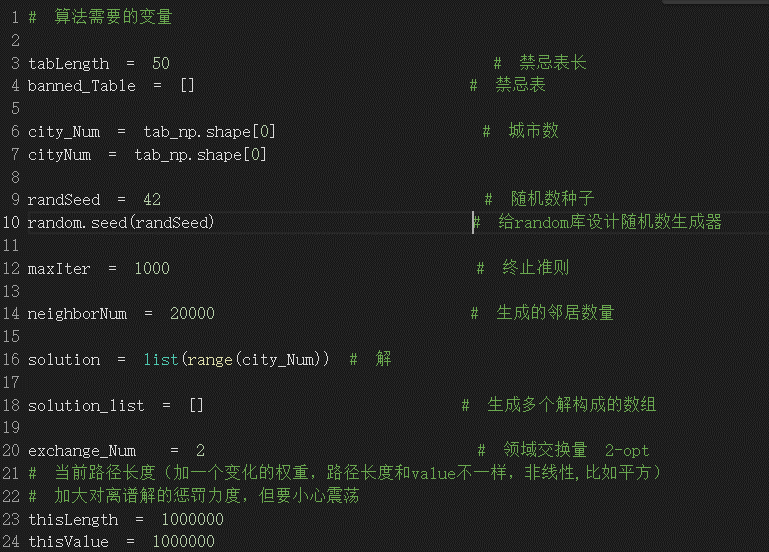
最终在算法运行速度优化上有总体85%时间优化。比如在读取距离方法、ATSP问题处理等多个算法耗时的瓶颈处做了消融实验，选取了运行速度更快的解决方案。同时在算法书写时也使用lamda表达式排序等方法尽量减少代码复杂度。在执行时间判断函数过程中，使用了jupyter notebook独有的魔术指令，在每个单元格中都运行多次，求取平均值作为最终结果。下图记录的是数据结构的决定。



不同数据结构的读取效率比较

### 超参数解耦合

为了保证算法的泛化性能，在代码实现过程中将所有可控变量解耦合。用较多的封装保证算法可以很容易的进行参数调节。在run（）函数中，可以通过改变超参数传入，容易的完成各项功能的消融实验。

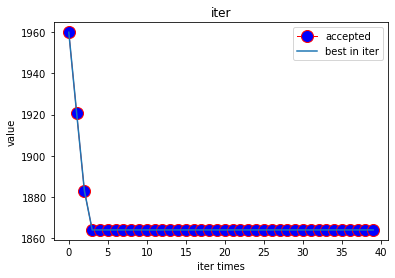


算法超参数单独单元定义

同时我也结合自己在深度学习中学习的技巧，将lerning rate（tabu长度）下降、抖动（长时间梯度不下降就跳开）、允许随机犯错等手段加入了算法实现过程中，尽可能优化算法。

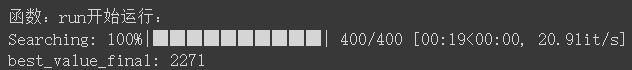
### 训练过程的可视化

使用了plt中的可视化函数，便于直观感受算法运行情况，给出调参数的方向。



如图为最好的一次解

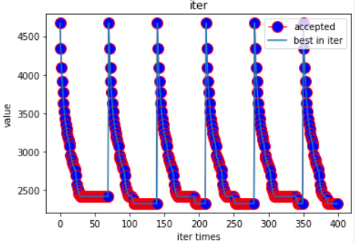
使用了tqdm库，来可视化算法的训练进程。在大规模测试情况下得以判断运行时间。



## 消融实验

### 算法鲁棒性

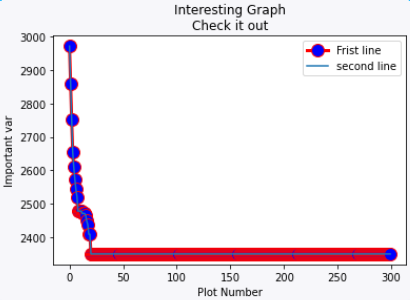
在某次试验中，发现尽管采用同一个初始解，算法收敛的最优值不同。因此固定随机数种子，多次运行来观察解的情况。在多次运行后，我发现最终充分运行（100iter无变化）的算法在最优值上误差不超过30，也就是说算法下降能力其实取决于初始值的选取，算法自身的下降能力是有限的。在改变种子运行多次后这样的结论仍然成立，说明对于此算法，初始解的给定要比算法设计更重要，算法的跨出局部能力有限。



同随机数种子、同初始解多次运行结果

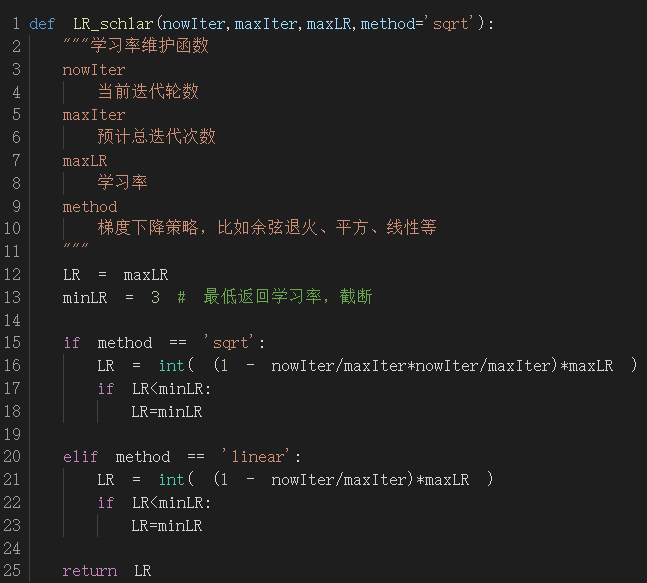
### 学习率衰减

在经过大规模的测试后，我发现算法在20次迭代后就很难找到更好的解。如下图所示。



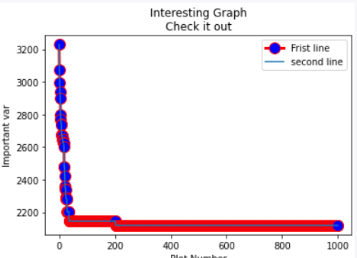
算法运行在20iter后难以下降

根据对禁忌表变量的debug逐步运行，我发现一个可能的远因是学习率（禁忌表长度）过大，导致算法不能在临近最优解时及时下降。因此我参考机器学习中梯度下降的策略，加入了学习率随迭代次数动态下降的策略。



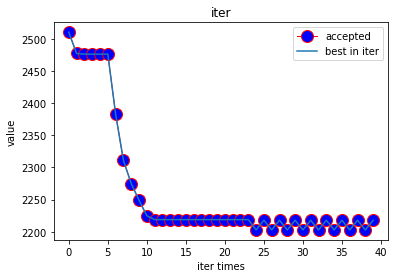
学习率维护函数（有平方和线性两种方法）

加入学习率衰减策略后，经消融实验判断后，尽管其有效（观察到在200iter仍有下降），但作用不大。



学习率衰减策略

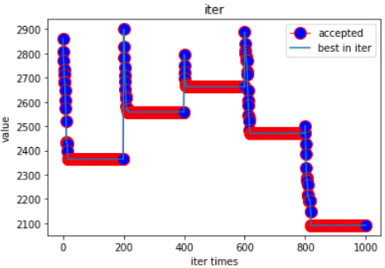
同时还发现了在学习率（禁忌表）过小的情况下，会出现抖动解



较小的表长使得算法结果发生抖动

### 抖动

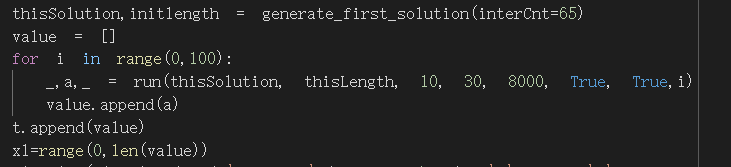
因此我又加入了随机接受不优解的策略，但提升仍较为有限。因此又一次参考深度学习中随机抖动的方法，在一定迭代次数都得不到更新的值后，加入抖动跳出次局部最优解。可以观察到算法性能来到了2100以下。



### 接受非最优的第n个解

某次试验采取了大规模的i，但是最终发现开始一段时间最优值的差距不大，也就是说这样接受次优解的行为反倒没有明显性能下降。这可能是由于问题的解空间随机性过强，不具有明显的邻域特性来执行类似于梯度下降的接近策略。

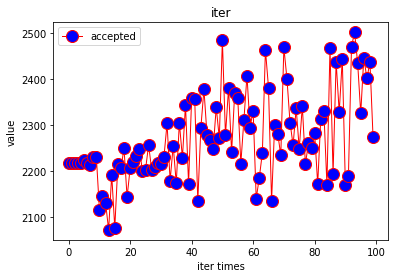
设计了如下消融实验，在使用同一个生成的初始值的情况下，运行100次完整的搜索算法，过程中每次的最优值选取位置向后移动一个。目的是为了探索贪婪程度对于算法性能的影响。



消融实验设置

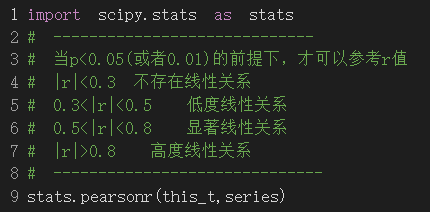
观察到在消融实验中，观察到随着接受位置从第i个值开始的i增加，抖动变大，但总体在最优值附近。最优的搜索能力出现在17，也就是说在更新禁忌表，更新最优值的时候，从第17个开始能有效防止过于贪婪导致陷在局部最优。

综上，开始位置k能防止算法过于贪婪，但过大的k将会降低算法性能。因此这也是又一个trade-off。（我观察到在深度学习和统计学习的方法中各种属性上都存在着诸多trade-off，这似乎是复杂系统的普遍现象）



最优解从第i个位置开始的100次算法运行结果

同时我也进行了线性相关性分析，相关系数为0.541，p值为 6.18e-09。根据统计学，参数k与算法性能存在着较强的线性相关性。



线性相关判据

### 其他实验

在迭代次数、随机接受接的随机程度、dropout（随机删掉禁忌表某些元素）、生成邻居算法等等策略上做过许多尝试。但是这些策略没有获得很好的解（有时是负面影响），没有写入报告。