遗传算法实验————解决TSP问题

> 本报告使用遗传算法解决TSP问题

1. 实验背景

旅行商问题，即TSP问题（Traveling Salesman Problem），也叫旅行推销员问题、货郎担问题，是数学领域中著名问题之一。假设有一个旅行商人要拜访n个城市，他必须选择所要走的路径，路径的限制是每个城市只能拜访一次，而且最后要回到原来出发的城市。路径的选择目标是要求得的路径路程为所有路径之中的最小值。

旅行推销员问题是图论中最著名的问题之一，即“已给一个n个点的完全图，每条边都有一个长度，求总长度最短的经过每个顶点正好一次的封闭回路”。Edmonds，Cook和Karp等人发现，这批难题有一个值得注意的性质，对其中一个问题存在有效算法时，每个问题都会有有效算法。

迄今为止，这类问题中没有一个找到有效算法。倾向于接受NP完全问题（NP-Complete或NPC）和NP难题（NP-Hard或NPH）不存在有效算法这一猜想，认为这类问题的大型实例不能用精确算法求解，必须寻求这类问题的有效的近似算法。

2. 遗传算法

2.1 算法简介

遗传算法（Genetic Algorithm）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。遗传算法是从代表问题可能潜在的解集的一个种群（population）开始的，而一个种群则由经过基因（gene）编码的一定数目的个体(individual)组成。每个个体实际上是染色体(chromosome)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。因此，在一开始需要实现从表现型到基因型的映射即编码工作。由于仿照基因编码的工作很复杂，我们往往进行简化，如二进制编码，初代种群产生之后，按照适者生存和优胜劣汰的原理，逐代（generation）演化产生出越来越好的近似解，在每一代，根据问题域中个体的适应度（fitness）大小选择（selection）个体，并借助于自然遗传学的遗传算子（genetic operators）进行组合交叉（crossover）和变异（mutation），产生出代表新的解集的种群。这个过程将导致种群像自然进化一样的后生代种群比前代更加适应于环境，末代种群中的最优个体经过解码（decoding），可以作为问题近似最优解。

模拟进化计算（Simulated Evolutionary Computation) 是近二十年来信息科学、人工智能与计算机科学的一大研究领域，由此所派生的求解优化问题的仿生类算法（遗传算法、演化策略、进化程序），由于其鲜明的生物背景、新颖的设计原理、独特的分析方法和成功的应用实践，正日益形成全局搜索与多目标优化理论的一个崭新分支。

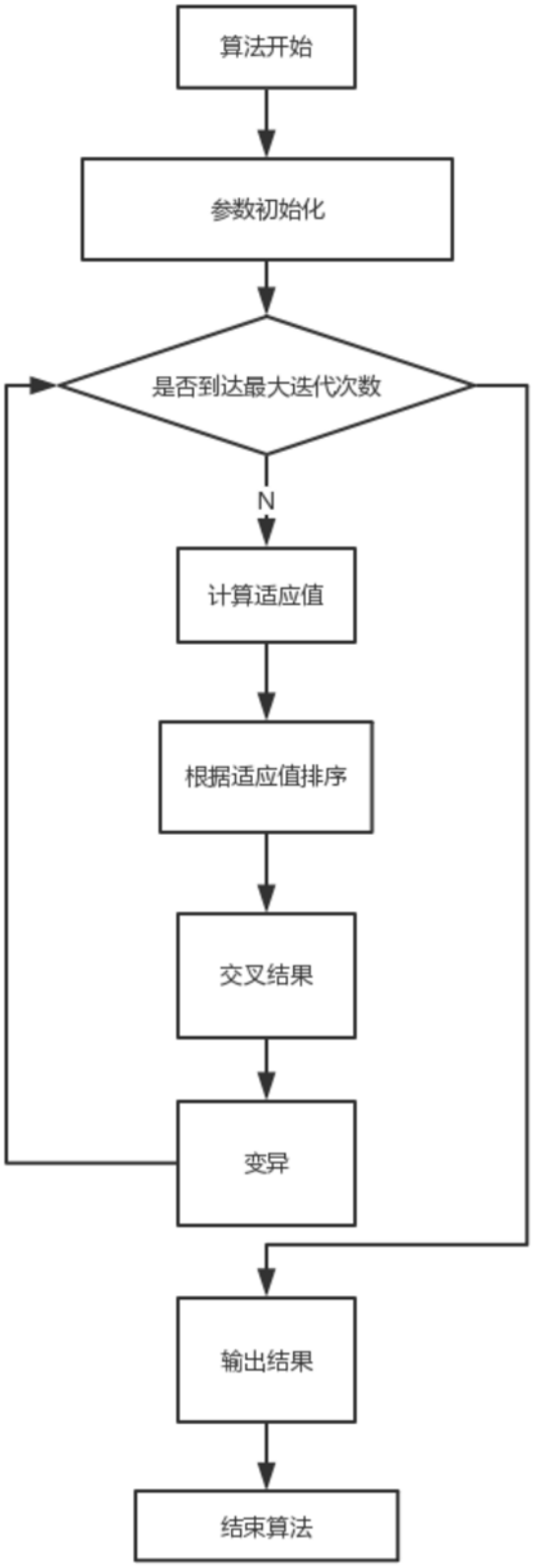
遗传算法(GeneticAlgorithm ， GA)是通过模拟生物进化过程来完成优化搜索的，由美国J. Holland 教授提出的一类借鉴生物界自然选择和自然遗传机制的随机化搜索算法。它起源于达尔文的进化论，是模拟达尔文的遗传选择和自然淘汰的生物进化过程的计算模型。其主要特点是群体搜索策略和群体中个体之间的信息交换，搜索不以梯度信息为基础。它尤其适用于处理传统搜索方法难于解决的复杂和非线性问题，可广泛应用于组合优化、机器学习、自适应控制、规划设计和人工生命等领域。作为一种全局优化搜索算法，遗传算法以其简单通用、鲁棒性强、适于并行处理以及应用范围广等特点，使其成为21 世纪智能计算核心技术之一。进入80 年代，遗传算法迎来了兴盛发展时期，无论是理论研究还是应用研究都成了十分热门的话题。

2.2 算法结合TSP

1. 种群初始化
   编码方式为顺序编码，一个种群中包括m个染色体（路线），一条染色体（个体）包含n个基因(城市),还需要的参数是交叉参数和变异参数。
2. 适应度函数
   计算每一个封闭路线（染色体）的长度作为适应度函数。此时，适应度越小越好。
3. 选择
   为了使编程简单，可以在迭代过程中保持种群数量m不发生变化，但是为了增加效率，在迭代过程中可以使种群数量发生变化。即每次产生m个新的个体后，将其与之前的个体混在一起，用轮盘赌法选出m个个体进行交叉产生下一代。因此，每一次的迭代都会使种群大小增加m个新的个体。
4. 交叉
   两个不同个体在对应位置交叉，注意保持交叉之后的个体仍然是1~n的排列。
5. 变异
   以概率选取个体进行两个基因的交换，这是最简单的办法，而且可以保证一个个体中的基因不会重复。

3. 算法设计流程

3.1 算法流程图



3.2 算法实现过程

使用python3作为设计语言，其中本次使用的版本是python3.6，其中使用了常规的的库有：

1. time
2. copy
3. random
4. math

使用的非内置库有：

1. numpy
2. matplotlib

3.2.1 算法设计核心

> 关键在于：
>
> 1. 变异方式
> 2. 交叉方式

1. 交叉方式
   使用经典的交叉方式，由于对于TSP问题，编码方式需要进行注意，也就是：取父代码的一段编码，随后按照母的顺序将未加入的数值加入，这样的话新的结果实际上对于顺序而言继承的很好。实际上TSP问题就是访问点的顺序问题，因此顺序非常重要。
2. 变异方式
   变异方式使用最基础的方式：
   随机两个数据然后交换位置，这样可以方便的保证不会有重复元素

3.2.2 算法设计细节

3.2.2.1 TSP问题框架

首先定义点：

```
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y
```

然后定义图的数据，注意到 `ask_distance_for_plan()` 方法

```
class Graph:
    def __init__(self, n):
        self.points = []
        self.point_n = n

    def add_point(self, p):
        self.points.append(p)
        self.point_n += 1

    def ask_distance(self, i, j):
        point_x = self.points[i]
        point_y = self.points[j]
        return math.sqrt(sqr(point_x.x - point_y.x) + sqr(point_x.y - point_y.y))

    def ask_distance_for_plan(self, plan):
        res = 0.0
        for i in range(1, self.point_n):
            res += self.ask_distance(plan[i], plan[i - 1])
        res += self.ask_distance(plan[0], plan[self.point_n - 1])
        # print(plan,' ',res)
        return res

    def show(self):
        for item in self.points:
            print(item.x, " ", item.y)
```

3.2.2.2 GA细节的设计

1. 设置初始值

```
# 初始化一个序列作为初始数据
def initPlan(self, n):
    random.seed(time.time())
    init_plan = []
    for i in range(0, n):
        init_plan.append(i)

    for i in range(n):
        j = random.randint(0, n - 1)
        while i == j:
            j = random.randint(0, n - 1)
        init_plan[i], init_plan[j] = init_plan[j], init_plan[i]

    return init_plan

def init_group(self, n):
    groups = []
    for i in range(self.MAX_GROUP):
        plan = self.initPlan(n)
        while plan in groups:
            plan = self.initPlan(n)
        groups.append(plan)
    return groups
```

2. 交叉设计

```
def exchange(self, parent1, parent2):
    index1 = random.randint(0, len(parent1) - 1)
    index2 = random.randint(index1, len(parent1) - 1)
    tempGene = parent2[index1:index2]  # 交叉的基因片段
    newGene = []
    p1len = 0
    for g in parent1:
        if p1len == index1:
            # 插入基因片段
            newGene.extend(tempGene)
            p1len += 1
        if g not in tempGene:
            newGene.append(g)
            p1len += 1
    return newGene
```

3. 变异设计

```
def mutate(self, list_x):
    random.seed(time.time())
    n = len(list_x)
    u = random.randint(0, n - 1)
    v = random.randint(0, n - 1)
    while v == u:
        v = random.randint(0, n - 1)
    list_x[u], list_x[v] = list_x[v], list_x[u]
    return list_xs
```

4. 关于排序
   在代码中保留一个排序的位置，但是排序对于当前的设计中效果不是很明显，基于对于速度的考虑，并没有调用这个函数
5. 整合
   最终整合出的代码，给出核心的 `run()` 方法

```
def run(self, n, graph):
    # n is the size of each individual
    init_group = self.init_group(n)
    # init_group = self.sort_group(init_group, graph)
    i_time = 0
    best_ans = {
        "fitness": graph.ask_distance_for_plan(init_group[0]),
        "plan": init_group[0]
    }
    print(init_group)
    ans_each = []
    while i_time < self.MAX_TIME:
        print(best_ans)
        for i in range(self.MAX_GROUP):
            if random.random() < self.EXCHANGE_RATE:
                j = random.randint(0, self.MAX_GROUP - 1)
                while i == j:
                    j = random.randint(0, self.MAX_GROUP - 1)
                new_plan = self.exchange(init_group[i], init_group[j])
                init_group[i] = copy.copy(new_plan)

            if random.random() < self.MUTE_RATE:
                init_group[i] = self.mutate(init_group[i])

            if graph.ask_distance_for_plan(init_group[i]) < best_ans["fitness"]:
                best_ans["fitness"] = graph.ask_distance_for_plan(init_group[i])
                best_ans["plan"] = copy.copy(init_group[i])

        ans_each.append(best_ans["fitness"])
        i_time += 1

    return best_ans,ans_each
```

4. 实验结果

4.1 实验数据

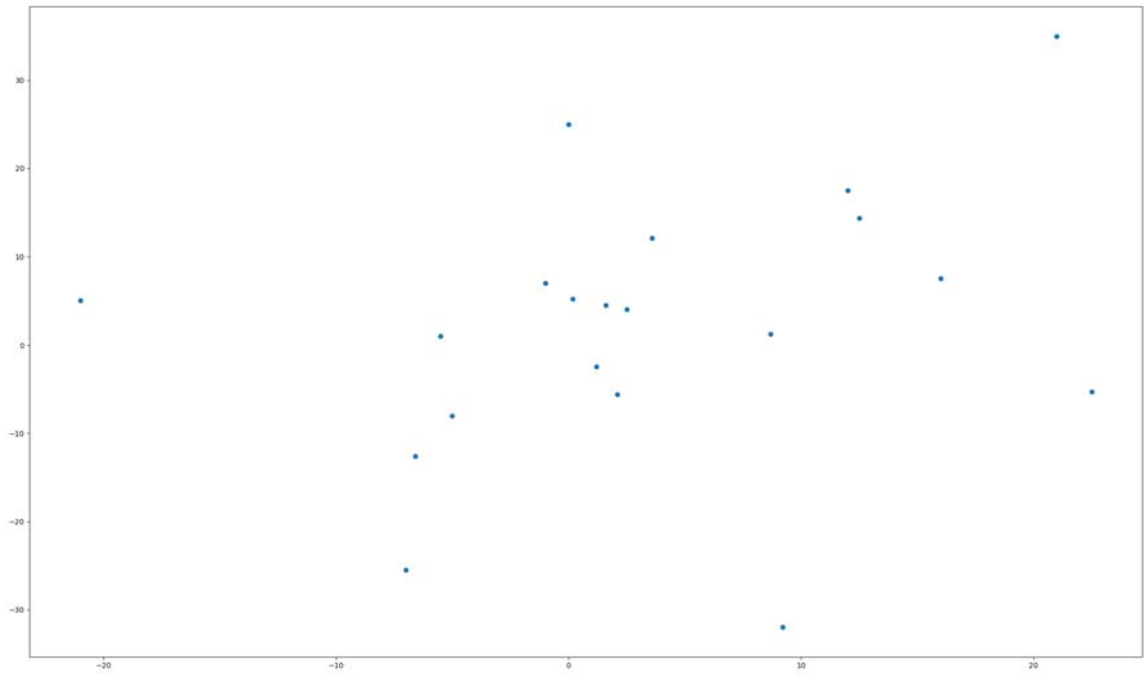
使用的实验数据如下：
给出所有点的坐标

```
A 2.5 4.0
B 1.2 -2.4
C 8.7 1.2
D 3.6 12.1
E -5.5 0.94
F -6.6 -12.6
G 0.18 5.219
H 12.5 14.3609
I 22.5 -5.26
J 1.61 4.5
K 2.1 -5.6
L 0 25
M 9.2 -32
N -1 7
O -5 -8
P 21 35
Q 16 7.5
R -21 5
S -7 -25.5
T 12 17.5
```

给出读入数据的python代码：

```python
graph = Graph(0)
with open("in.txt","r") as f:
    lines = f.readlines()
    for line in lines:
        line = str(line)
        # print(line)
        items = line.split(' ')
        x = float(items[1])
        y = float(items[2])
        # print('x =',x,' y = ',y)
        graph.add_point(Point(x,y))
```

给出一个简单的图：

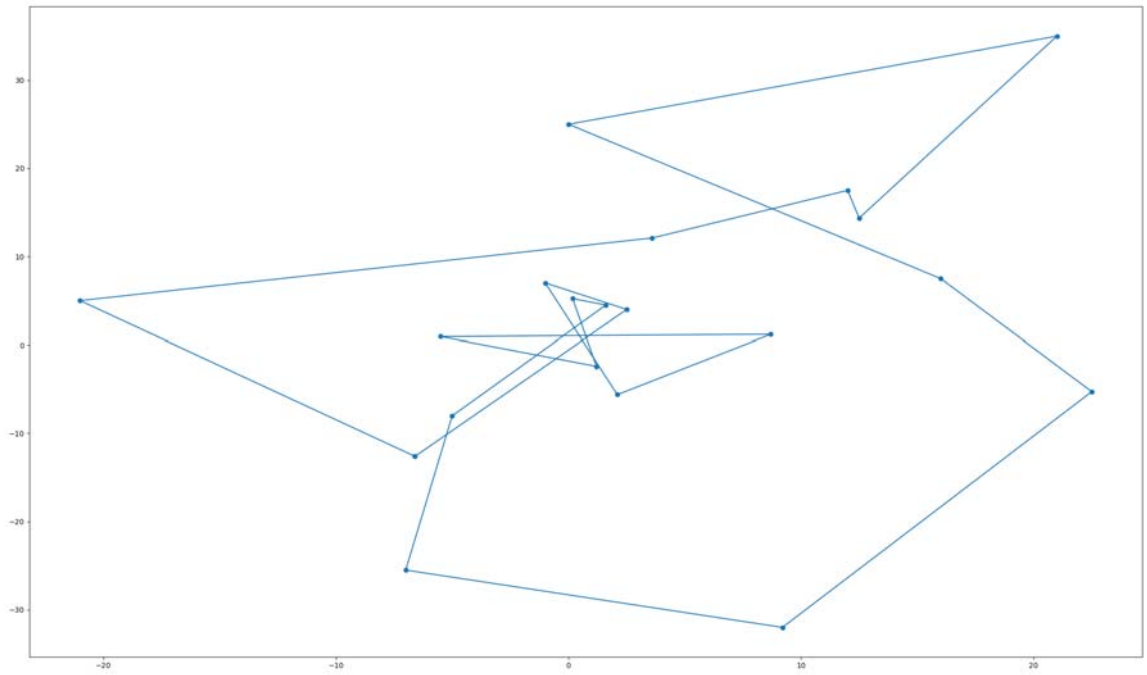


4.2 实验结果展示

最终运行的结果如下：

这是迭代1000，种群数为1000的结果：

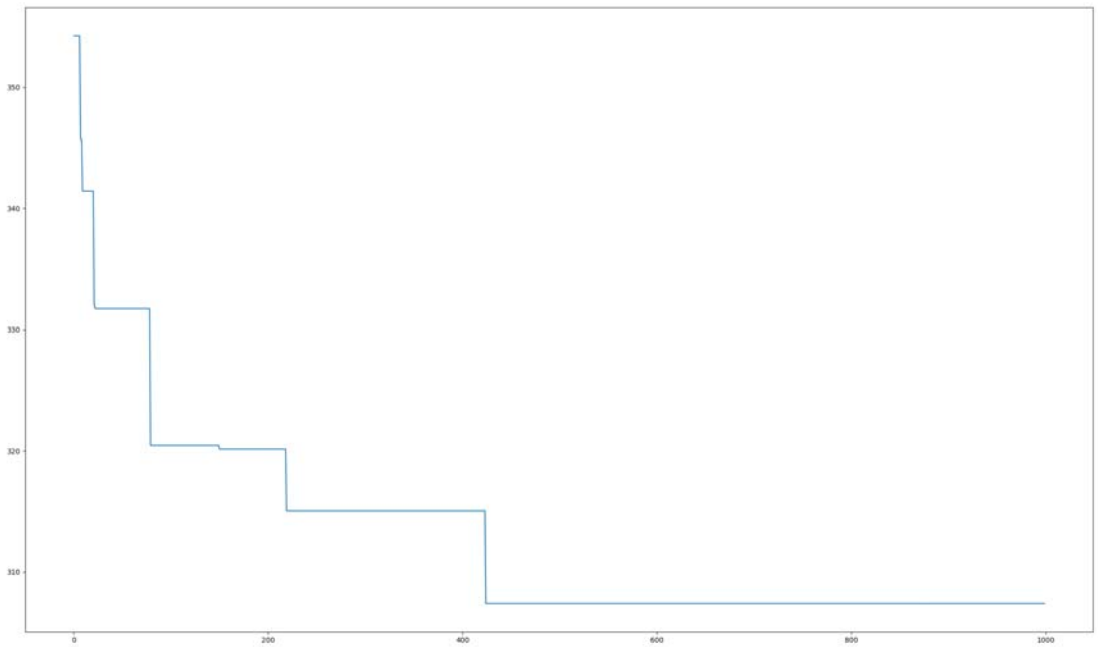
{'fitness': 301.1644602573964, 'plan': [5, 17, 3, 19, 7, 15, 11, 16, 8, 12, 18, 14, 9, 6, 1, 4, 2, 10, 13



GA算法在这种参数结构下，对于实际结果还是比较接近的。

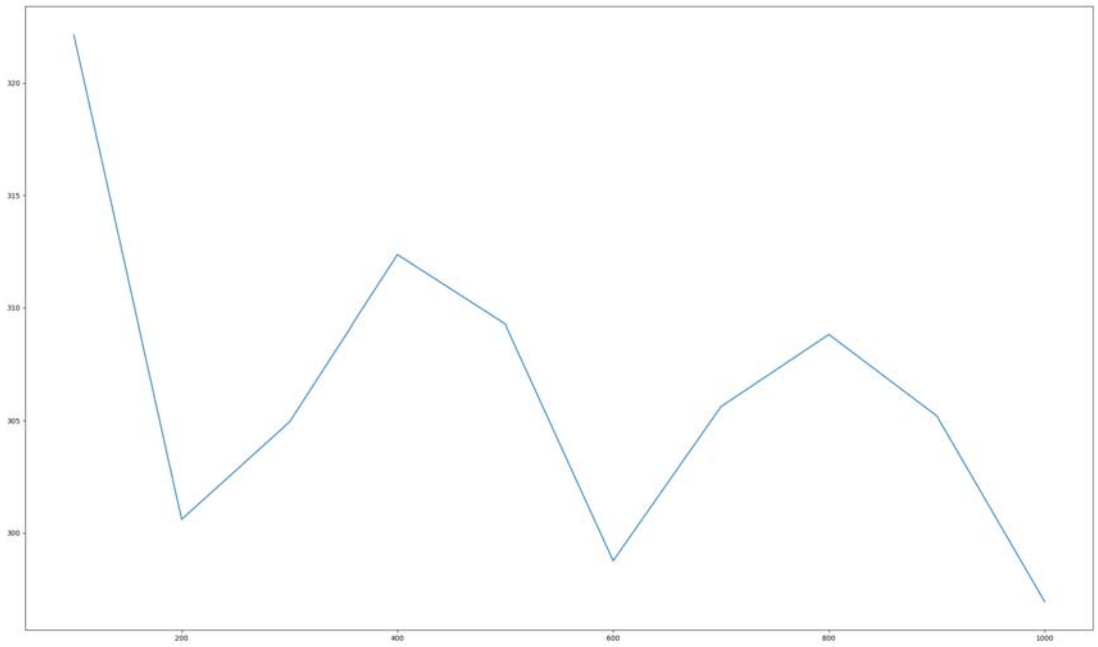
4.2 实验结果分析

1. 记录迭代中的结果
   我们记录迭代过程中的结果，这个结果同时可以表示在相同种群数下，迭代次数的增加对于最终结果的影响：

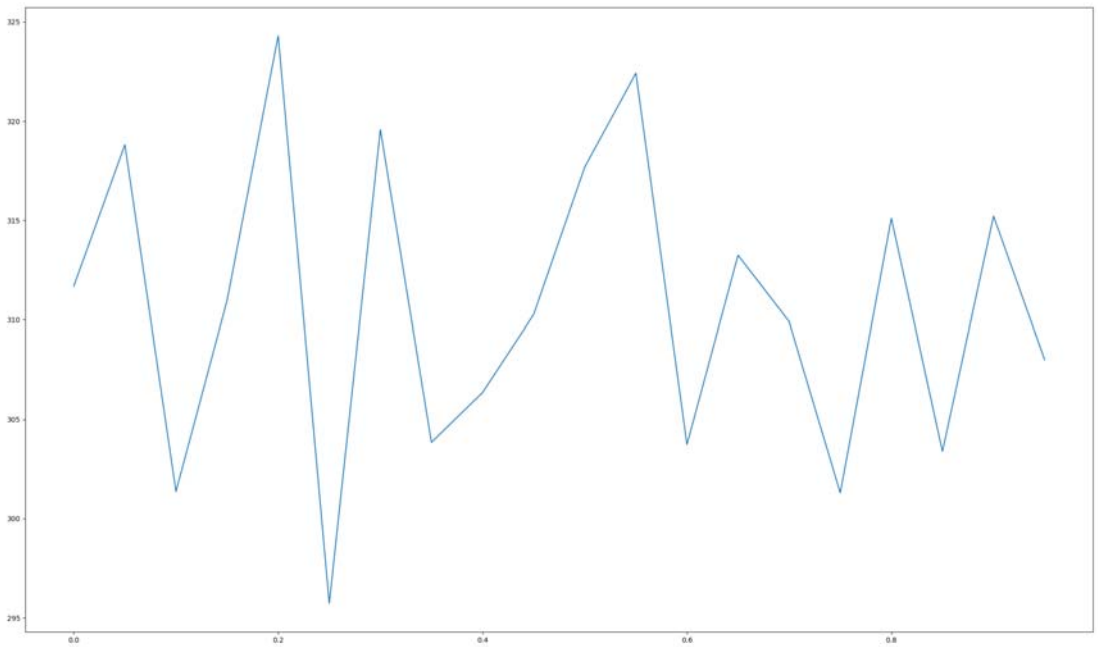


可以看到一开始的降落十分迅速，一直到最后趋于稳定

2. 种群数对于结果的影响
   我们修改种群数对于最终解的结果：



   可以看到基本上是一个下降的过程。中间差别不是很大的有所摇摆。
3. 变异率的影响
   如果我们改变变异率，最终结果的变化如下：



其中我们选择的数据：

```
[0.0, 0.05, 0.1, 0.15, 0.2, 0.25, 0.3, 0.35, 0.4, 0.45, 0.5, 0.55, 0.6, 0.65, 0.7, 0.75, 0.8, 0.85, 0.9,
```

可以看到结果还是比较不稳定的，这也可以看出这种mute方式，由于强行更换顺序，所以效果不尽如人意的结果