遗传算法实验报告

姓名：欧穗新
学号：16340173
日期：2019 年 1 月 12

摘要：

本次实验在 TSPLIB 中选了 ch130（130 个城市）的 TSP 问题，然后采用遗传算法进行求解（问题规模等和模拟退火求解 TSP 实验同）。在遗传算法中，使用 tsp 路径长度的倒数作为 tsp 解个体的适应度，使用了轮盘赌的选择策略选择父本解个体，然后使用二交换启发式交叉算子对父本进行交叉产生后代，再以一定的概率对后代进行变异，最后将后代与父本按照一定比例合起来构成新一代，反复如此迭代若干代得到最后种群，取其中最有个体作为遗传算法的解。然后本实验还将遗传算法与之前模拟退火算法进行对比，得出结论遗传算法具有更大的潜力，如果给与足够的时间加上足够好的变异交叉操作以保证种群多样性，遗传算法能够一直向全局最优解迈进，而不会陷入局部最优。而模拟退火则能够在较短时间内得出较好的局部最优解，如果参数设置的当，模拟退火算法也能达到很好的效果。最后本实验总结了设计高效遗传算法的一些经验，并将单点搜索和多点搜索进行了比较，得出多点搜索能够利用种群的力量求解问题从而得到稳定的、优秀的解，而单点搜索仅仅利用个体力量求解，具有一定的随机性，得到的解时好时坏。除此之外，代码能够提供可视化，观察路径的变化和交叉程度。

1．导言

旅行推销员问题（英语：Travelling salesman problem, TSP）是这样一个问题：给定一系列城市和每对城市之间的距离，求解访问每一座城市一次并回到起始城市的最短回路。它是组合优化中的一个 NP 困难问题，在运筹学和理论计算机科学中非常重要。
最早的旅行商问题的数学规划是由 Dantzig（1959）等人提出，并且是在最优化领域中进行了深入研究。许多优化方法都用它作为一个测试基准。尽管问题在计算上很困难，但已经有了大量的启发式算法和精确方法来求解数量上万的实例，并且能将误差控制在 1%内

使用的方法：

（1）模拟退火，模拟退火是对热力学退火过程的模拟,在某一给定初温下，通过缓慢下降温度参数，使算法能够在多项式时间内给出一个近似最优解。本质上也是蒙特卡洛算法。作为一种比较简单的智能算法，能以较高的效率解决优化问题，如求最值，tsp 问题等。

（2）遗传算法（英语：genetic algorithm (GA) ）是计算数学中用于解决最佳化的搜索算法，是进化算法的一种。 进化算法最初是借鉴了进化生物学中的一些现象而发展起来的，这些现象包括遗传、突变、自然选择以及杂交等。

2．实验过程

算法概述：

遗传算法：

（1） **设置初始种群**，我是随机产生的初始种群，个体 100 个。

尝试过设置一定量贪心解，效果并不好，虽然收敛很快，但是因为种群多样性受限，最终的结果并不好（哪怕仅仅设置一个贪心解，也会导致种群收敛很快，多样性不足，最终生成优解的潜力不足）

（2） **计算出当前种群的适应度**，计算 f（i）求和，f（i）是每个个体的路径总长度的倒数，即 1/path_length

（3） **选择**：使用轮盘赌从种群中随机选择两个到三个个体

（4） **交叉**：一开始的时候自己使用的交叉操作是随机的顺序交叉操作，没有启发式任何信息，后来发现效果极差（反正基本上达不到 10%以内），收敛的很慢甚至无法收敛，所以自己在网上查找比较好的交叉操作：两交换/三交换启发式交叉算子，使用两个/三个父代生成一个子代：

比如有如下三条路径作为父代：

A->B->C->D->E->F->A

A->D->C->F->E->B->A

A->C->F->E->B->D->A

（i） 这三条路径都是**从 A 点出发**（所以子代也从 A 出发）

（ii） 然后分别前往 B、D、C 三个城市，**我们从 A->B、A->D、A->C 三条路中选择最短的**（比如 A->B 是最短的，经过城市，可以**将剩余路径循环移位使得 B 出现在队首**：比如第二条路径剩余 D->C->F->E->B，循环移位得到 B->D->C->F->E，最后这个父代更新为 A->B->D->C->F->E->A

（iii） 不断执行（i）（ii），得到完整子代

（5） 操作（4）之后执行**变异**操作：

变异操作尝试了爬山法中（i）（ii）的操作，**最终选择了第二种，逆转一段路径，因为这种变异操作表现更好，但是其实逆转节点的操作也能够接受**

因为变异操作的作用就是增加种群的多样性

在遗传算法执行代数很多之后会收敛，这时候种群里面的个体基本上相同，交叉操作已经无法产生不同于父代的子代了。（所以没办法进一步优化种群）

这时候就通过变异操作获得不同于父代的个体，从而有小概率能够优化种群

（6） **反复执行（3）-（5）**生成 100*0.9 个后代，另外 100*（1-0.9）个后代从父代中选择（保留父代中优秀的个体），注意不能选太多的父代个体放到下一代，这会导致多样性不足

（7） **循环执行（2）到（6）**10000 代，将最后种群的最优后代输出作为结果

实现算法的程序主要流程，功能说明：

（1）generate_random_list 函数产生初始随机解，输入问题规模（有多少个城市），返回一种随机的走法：

```
def generate_random_list(point_num):
    point_list = [0]*(point_num - 1)
    for i in range(0, point_num - 1):
        point_list[i] = i+1
    random.shuffle(point_list)
    point_list.insert(0, 0)
    point_list.append(0)
    return point_list
```

（2）get_distance 获取两点间的距离，将两个城市的位置作为参数，返回距离，用于计算整个 tsp 路径长度

```
def get_distance(point_a, point_b):
    temp1 = math.pow(coordinate_x[point_a] - coordinate_x[point_b], 2)
    temp2 = math.pow(coordinate_y[point_a] - coordinate_y[point_b], 2)
    return math.pow(temp1 + temp2, 0.5)
```

（3）get_path_length 函数使用路径数组作为参数，用 get_distance 计算路径距离，得出整个 tsp 路径的长度

```
def get_path_length(point_list):
    path_length = 0
    for i in range(0, len(point_list)-1):
        path_length += get_distance(point_list[i], point_list[i+1])
    path_length += get_distance(point_list[len(point_list)-1], point_list[0])
    return path_length
```

GenericAlgorithm 类，其 solve 函数遗传算法求解 tsp 最短路

```
class GenericAlgorithm:
```

（4）GenericAlgorithm 类的 show_figure 函数用于将 tsp 问题可视化，显示出；路径求解的具体过程

GenericAlgorithm 类的 store_result 函数用于将求解的结果存入文件中

```
def show_figure(self):
def store_result(self):
```

（5）GenericAlgorithm 类的 solve 函数，用于求解 tsp 问题，在这个函数中，循环 1000 代，不断产生后代（使用稍后介绍到的 generate_next_generation 函数产生一代后代），然后将 100 代繁衍之后的种群最优解进行存储

```
def solve(self):
    while self.generation_count < MAX_GENERATION:
        self.generate_next_generation()
    self.store_result()
```

（6）GenericAlgorithm 类的 generate_next_generation 函数，该函数使用当前的种群产生下一代子代，具体过程是：（1）对当前种群估值（2）循环使用 generate_children 函数产生一个后代个体，直到达到一定数量（3）将生成的后代和一定比例的父代最有个体合并为新的种群（4）代数计数加一

```
def generate_next_generation(self):
    self.evaluate()

    new_generation = queue.PriorityQueue()

    # generate child
    while len(new_generation.queue) < self.population_size*(1 -
self.preserve_rate):
        temp = self.get_children()
        new_generation.put(Individual(get_path_length(temp), temp))

    # add children individual and parent individual:
    for i in range(0, int(self.population_size*self.preserve_rate)):
        new_generation.put(self.population.get())

    # change generation
    self.population = new_generation

    self.generation_count += 1
```

（7）GenericAlgorithm 类的 generate_children 函数，即产生一个后代的具体过程：（1）选择，根据轮盘赌策略获取两个父本，（2）交叉，使用之前提到过的二交叉启发式算子，利用两个父本交叉产生孩子（3）变异，生成的孩子以一定概率变异（这里的变异操作与之前模拟退火种局部搜索操作一样，有交换两个城市（交换基因）/逆转子路径（逆转基因片段）两种）

```
def get_children(self):
    new_children = []
    # selection
    parent1, parent2 = self.get_two_parent()
    # crossover
    new_children = self.crossover_1(parent1, parent2)
    # mutation
    if random.random() < self.mutation_rate:
        new_children = self.mutation(new_children)
    return new_children
```

（8）GenericAlgorithm 类的 evaluate 函数，即**估值函数**，使用 1/path_length 作为个体适应度，对所有的适应度求和，使用轮盘赌的策略选择父本（这里仅仅是计算适应度、求和，供在 get_two_parent 中轮盘赌策略使用）

```
def evaluate(self):
    self.population_roulette = 0
    for i in range(0, self.population_size):
        self.population_roulette += 1/self.population.queue[i].score
    # for i in range(0, self.population_size):
    #     print(i, 'th sol:', self.population.queue[i].score, 'in ',
self.generation_count, 'generation')
    if IF_SHOW_FIGURE and self.population.queue[0].score < self.min_cost:
        self.min_cost = self.population.queue[0].score
        self.min_cost_set.append(self.min_cost)
        self.best_solution = self.population.queue[0].point_list
        self.show_figure()
    elif self.population.queue[0].score < self.min_cost:
        self.min_cost = self.population.queue[0].score
        self.min_cost_set.append(self.min_cost)
        self.best_solution = self.population.queue[0].point_list
    print('best sol:', self.population.queue[0].score, 'in ',
self.generation_count, 'generation')
```

（9）GenericAlgorithm 类的 get_two_parent 函数，即**选择函数**，使用轮盘赌策略获取两个父本并返回

```
# selection
def get_two_parent(self):
    parent1 = -1
    parent2 = -1
    while parent1 == parent2:
        roulette_value1 = random.uniform(0, self.population_roulette)
        for i in range(0, self.population_size):
            roulette_value1 -= 1/self.population.queue[i].score
            if roulette_value1 < 0:
                parent1 = i
                break

        roulette_value2 = random.uniform(0, self.population_roulette)
        for i in range(0, self.population_size):
            roulette_value2 -= 1/self.population.queue[i].score
            if roulette_value2 < 0:
                parent2 = i
                break
    if parent1 == -1 or parent2 == -1:
        print('error')
        exit(0)
    return parent1, parent2
```

（10）GenericAlgorithm 类的 crossover 函数，即**交叉**操作，使用之前选择的父本，使用算法概述中提到的二交叉启发式算子得到一个子代 tsp 解

```
def crossover_1(self, parent1, parent2):
    temp_child1 = []
    temp_child2 = []
    for i in range(0, self.problem_size+1):
        temp_child1.append(self.population.queue[parent1].point_list[i])
        temp_child2.append(self.population.queue[parent2].point_list[i])

    for i in range(1, self.problem_size):
        temp_dist = min(get_distance(temp_child1[i - 1], temp_child1[i]),
                        get_distance(temp_child2[i - 1], temp_child2[i]))
        if temp_dist == get_distance(temp_child1[i-1], temp_child1[i]):
            temp_child2[i:self.problem_size] =
cycle_shift(temp_child2[i:self.problem_size], temp_child1[i])
        elif temp_dist == get_distance(temp_child2[i-1], temp_child2[i]):
            temp_child1[i:self.problem_size] =
cycle_shift(temp_child1[i:self.problem_size], temp_child2[i])

    return temp_child1
```

（11）GenericAlgorithm 类的 mutation 函数，即**变异**操作，对交叉产生的后代以一定概率进行变异操作，产生变异后代，这里自己实现了一下两种变异操作，分别对应于（1）交换两个城市（交换基因）（2）逆转 tsp 路径的子路径（逆转基因片段）两种局部搜索操作：

```
def mutation(self, new_child):
    index1 = random.randint(1, self.problem_size - 1)
    index2 = random.randint(1, self.problem_size - 1)
    for i in range(index1, math.floor((index1+index2)/2) + 1):
        temp = new_child[i]
        new_child[i] = new_child[index1+index2-i]
        new_child[index1+index2-i] = temp
    return new_child
```

```
# different mutation strategy by swap two point
def mutation_1(self, new_child):
    index1 = random.randint(1, self.problem_size - 1)
    index2 = random.randint(1, self.problem_size - 1)
    temp = new_child[index1]
    new_child[index1] = new_child[index2]
    new_child[index2] = temp
    return new_child
```

3．结果分析

实验环境：
该实验在 Windows 下使用 python3 编码实现

参数说明：

```
MAX_NUM = 10000000
MAX_GENERATION = 10000
```

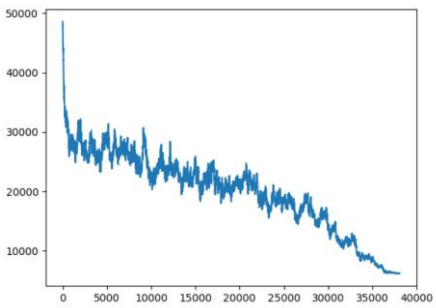
```
self.population_size = 50
self.generation_count = 0
self.crossover_rate = 1
self.mutation_rate = 0.9
self.preserve_rate = 0.2
```

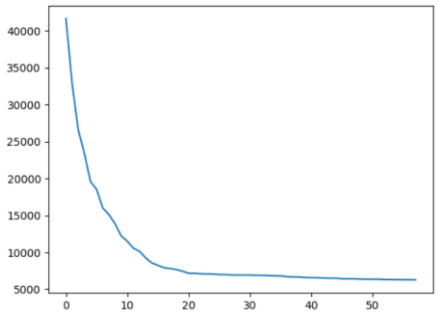
上面图片中显示，最大数字被设置为 10000000，种群繁衍代数为 10000 代，种群个体数为 50，交叉概率为 1，变异概率为 0.9（为了增加种群多样性，该参数应该设置的大一些），父代留存率 20%（新一代中有 20%是原本父代个体）

实验结果：（ch130 问题）
（1） 表格 1

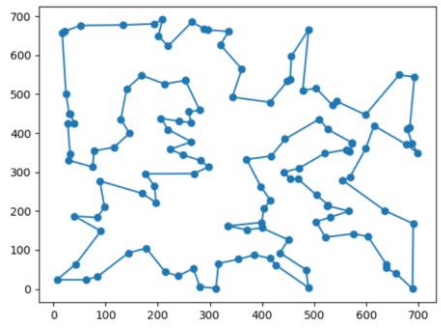
指标：（运行 10 次之后）	模拟退火	遗传算法
最好解	6203	6249
最差解	6382	6349
平均值	6305	6292.4
标准差	65.50159	36.42527
平均偏差值	3.19%	2.98%

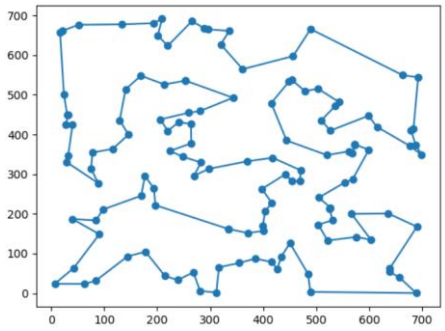
（2） 图像 1（取 10 次运行其中的一组）：
求解问题时随迭代次数增加，tsp 解对应路径长度变化曲线：（从左往右对应模拟退火算法、遗传算法求解时 tsp 对应路径长度变化曲线）





（3） 图像 2（取 10 次运行其中的一组）
最终 tsp 路径连线图：（从左往右依次对应模拟退火、遗传算法求解所得最终 tsp 路径连线图）





性能分析：

（1） 表格 1 对比分析

（i）对比模拟退火和遗传算法的**最好解**，模拟退火的最好解好于遗传算法，说明模拟退火如果参数设置得当，也能够得到相当好的解

（ii）对比它们的**最差解**，模拟退火差解更差，说明相比模拟退火，遗传算法有更好的稳定性

（iii）对比它们的解的**标准差**，模拟退火算法求得的解标准差大一倍以上，说明模拟退火算法得到的解不如遗传算法稳定，但是其实解之间的偏差都不算大

（iv）对比它们解的**平均值和平均偏差精度，都达到 3%左右，符合老师的要求，遗传算法微略优于模拟退火**，说明遗传算法具有更大的潜力，给足时间，它能够获取更优秀的解（不断向全局最优靠近），而模拟退火算法则是能够在较短时间得到相当优秀的解（前提是参数设置得当，我这里模拟退火的参数设置已经非常好了）

（2） 图像 1 对比分析

（i）使用遗传算法求解 tsp 问题用时很长，虽然看上去只有几百上千代，但是每一代中有 100 个个体，所以总的计算量非常大，**用时长，这是缺点，遗传算法用时甚至比模拟退火长的多，更不用说局部搜索了**

（ii）从迭代层数（代数）-tsp 路径长度变化曲线来看，在开始的时候收敛的非常快（这是因为自己使用的是启发式交叉算子，所以收敛很快），到代数很大时收敛很慢，但是仍在缓慢变化，即它不会陷入局部最优，**这是优点，因为交叉和变异总能够产生新的子代个体，种群总是在动态变化的，所以不会陷入局部最优，这一点不同于局部搜索和模拟退火，局部搜索和模拟退火会都陷入局部最优**（即使模拟退火有一定跳出局部最优的能力，但是仍然会无法跳出一些局部最优）；

（iii）**遗传算法的最优解个体不会波动（不会变差），而模拟退火的解有可能变差**，因此遗传算法得到的最终解是整个搜索过程中的最佳答案，而模拟退火算法得到的最终解有可能不是；

（3） 图像 2 对比分析

对比两个 tsp 解的路径连线图，都没有任何路径的交叉，只是在一些城市走法先后选择有所不同，这两个解都非常接近最优解了，从直观视觉效果看起来也都是不错的。

算法优缺点：

（1） 优点：本次遗传算法实验，自己使用了相当好的交叉操作----**二交换启发式交叉**，在交叉中加入启发式信息而不是随即交叉，从而达到加快种群收敛速度的效果，从而提高算法的效率，并且设计了多种变异操作（交换路径上两个城市/逆转路径的子路径），增加种群多样性，增强种群的潜力，从而有更多机会得到优秀的解

（2） 缺点：算法用时太长，这一点理应得到改进，自己使用的算法中，繁衍代数 10000 代，代数太多导致用时过长，但是减少代数又会导致解的精度不够，所以这一点还有待改进（算法用时）

4．结论。

关于设计高效的遗传算法的经验：

遗传算法的核心包括（1）适应度估值；（2）选择；（3）交叉；（4）变异；
以上四个核心步骤对于高效的遗传算法设计都十分重要

（1） 选择**合适的适应度估价算法**能够使得算法正确筛选出合适的父本，从而结合产生优秀的子代

（2） 选择算子，根据每个个体的适应度选择父本，**选择算子的改进空间不大**，基本上常用的选择算子就是轮盘赌和锦标赛

（3） 交叉算子，**交叉的设计非常重要，它决定了整个算法的收敛速度、算法潜力**，在完成本次实验的时候，一开始自己用的交叉算子是随即交叉（无启发式意义），结果整个种群收敛极慢，甚至无法收敛到 50%之内，最后的结果很差，后来自己在网上查阅资料，了解到三交换启发式算子对 tsp 解进行交叉，从而极大程度改进了算法，算法不仅能够收敛，并且能够稳定收敛到 5%内。但是这种启发式算子也带来了一些缺点----使得种群多样性不足，潜力下降，所以算法其实还能进一步改进，如果有很好的交叉操作，理论上能够兼顾收敛速度和算法潜力（种群多样性）

（4） **变异操作也很重要，它影响算法的潜力（和种群多样性有关）**，较好的变异操作能够增加种群多样性，使得在算法后期也能够不断产生不同的个体，增加种群多样性，帮助种群向更加优的方向前进，而如果变异操作不够好，那么生成的子代总是和父代相似，这样就陷入一个局部无法跳出了。所以理论上很好的变异操作能够避免增加种群多样性，陷入局部最优，从而提升算法潜力

单点搜索 vs 多点搜索之优缺点比较：

单点搜索（诸如模拟退火、局部搜索）都能够**比较快的得到比较优秀的解、可能具有一定跳过局部最优的能力但是仍然有可能陷入局部最优无法自拔**。至少模拟退火和局部搜索都是如此。

多点搜索则能够避免陷入局部最优（遗传算法），除非整个种群所有个体全部一样，那么就处在一个局部最优点，即使如此该种群也能够通过变异跳出局部最优，**总之只要给足够的计算时间，多点搜索具有很大的潜力**，能够一直向着最优解前进，**求解过程用时长是它的缺点**

单点搜索和多点搜索都需要设置好参数才能够有较好的表现，**多点搜索能够利用整体种群的力量求解问题，因此能够稳定得到好的解**，而**单点搜索使用个体求解问题，具有偶然性，因此单点搜索的最终解的好坏与初始解的选择有很大关系**（因此只有很小概率得到比较好的解）

主要参考文献(三五个即可)

[1] 遗传算法求解 TSP 问题[J]. 王辉. 计算机与现代化. 2009(07)
[2] 遗传模拟退火算法——黑龙江 TSP 问题[J]. 姚君. 价值工程. 2016(36)
[3] 改进遗传模拟退火算法求解 TSP[J]. 张雁翔,祁育仙. 智能计算机与应用. 2017(03)