

实验一实验报告

模拟退火算法与遗传算法

一. 实验要求

模拟退火算法：

在TSPLIB (<http://comopt.ifl.uni-heidelberg.de/software/TSPLIB95/>), 多个地址有备份; 其他网站还可以找到有趣的art TSP和national TSP) 中选一个大于100个城市数的TSP问题,

1. 采用多种邻域操作的局部搜索local search策略求解;
2. 在局部搜索策略的基础上, 加入模拟退火simulated annealing策略, 并比较两者的效果;
3. 要求求得的解不要超过最优值的10%, 并能够提供可视化, 观察路径的变化和交叉程度。

遗传算法：

用遗传算法求解TSP问题 (问题规模等和模拟退火求解TSP实验同), 要求:

- 1.设计较好的交叉操作, 并且引入多种局部搜索操作 (可替换通常遗传算法的变异操作)
- 2.和之前的模拟退火算法 (采用相同的局部搜索操作) 进行比较
- 3.得出设计高效遗传算法的一些经验, 并比较单点搜索和多点搜索的优缺点。

二、算法原理

1. TSP问题

TSP问题 (Travelling Salesman Problem) 即旅行商问题, 又译为旅行推销员问题、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。

TSP问题是一个组合优化问题。该问题可以被证明具有NPC计算复杂性。所有的TSP问题都可以用一个图（Graph）来描述：

- $V = \{c_1, c_2, \dots, c_i, \dots, c_n\}$, $i = 1, 2, \dots, n$, 是所有城市的集合, c_i 表示第 i 个城市, n 为城市的数目
- $E = \{(r, s) : r, s \in V\}$ 是所有城市之间连接的集合
- $C = \{c_{rs} : r, s \in V\}$ 是所有城市之间连接的成本度量（一般为城市之间的距离）

那么一个TSP问题可以表达为：

求解遍历图 $G = (V, E, C)$ 所有的节点一次并且回到起始节点，并使得连接这些节点的路径成本最低。

2. 局部搜索求解

局部搜索是一种局部择优的方法，采用启发式方法，每次从当前解的临近解空间中选择一个最优解作为当前解，直到达到一个局部最优解。

用局部搜索求解的步骤如下：

- 首先随机生成一段遍历所有城市的序列
- 采用某种邻域操作并比较操作前后的总距离，若距离变小则使这个邻域操作生效
- 重复以上过程，直到采用邻域操作后的总距离和之前的差小于某一阈值后停止
- 此时便得到了最优的序列

其中，邻域操作可以有很多种，我使用了如下三种：

- 随机找到一段序列后翻转
- 随机找到两段连续的序列后交换这两段序列的位置
- 随机找到两段连续的序列后将后一段序列翻转并交换这两段序列的位置

3. 模拟退火

- 初始化温度 T_0 , 令当前温度 $T = T_0$, 任取初始解
- 对当前解 S_1 进行邻域操作，然后产生一个新解 S_2
- 计算 S_2 的增量 $df = f(S_2) - f(S_1)$
- 若 $df < 0$, 则接受 S_2 作为新的当前解; 否则计算 S_2 的接受概率 $\exp\left(-\frac{df}{T}\right)$, 然后产生一个在 $(0, 1)$ 区间上均匀分布的随机数 $rand$, 若 $\exp\left(-\frac{df}{T}\right) > rand$ 则接受 S_2 作为新的当前

解 $S_1 = S_2$, 否则保留当前解 S_1

- 如果温度低于某一个值则输出当前解 S_1 为最优解, 程序结束; 否则将温度 T 乘上退火系数后返回第3步

4. 遗传算法

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, 通过模拟自然进化过程搜索最优解。通常的流程是先建立一个包含潜在的解的群体作为种群, 在环境作用下通过选择、交叉和变异一代代繁衍, 由于子代的环境适应力一般优于父代, 因此算法最终能够得到问题的较优解。

①选择

选择算法是指参照适应值函数, 按照预先选定的策略随机从父代中挑选一些个体生存下来, 剩下的个体则被淘汰。

这次实验我选择的是轮盘赌选择法。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, 步骤如下:

- 计算群体中每个个体的适应度
- 计算每个个体遗传到下一代的概率
- 计算每个个体的累计概率
- 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
- 重复上述步骤

②交叉

交叉是指仿照自然界基因传递的过程交配, 对存活下来的父代个体的某些基因进行优化组合, 办法是将两个父代个体某些对应位置的基因互换, 以产生新的个体。

这次实验我选择的是次序交叉法Order Crossover (OX):

- 随机选择一对染色体(父代)中几个基因的起止位置
- 生成一个子代, 并保证子代中被选中的基因的位置与父代相同
- 先找出第一步选中的基因在另一个父代中的位置, 再将其余基因按顺序放入上一步生成的子代中

③ 变异

变异操作我按照要求就是使用各种邻域操作。

三、算法实现

0. 初始化

- 首先定义一个城市类，用于保存编号及坐标

```
1 class city:
2     # 初始化城市编号及坐标
3     def __init__(self, index, x, y):
4         self.cityIndex = index
5         self.x = x
6         self.y = y
```

- 读取tsp文件获取城市信息

```
1 cities = []
2 # 读取tsp文件初始化城市
3 with open("../tc/rat195.tsp") as file:
4     cities = []
5     begin = False
6     for line in file.readlines()[0:-1]:
7         # 读取结束
8         if line.startswith('EOF'):
9             break
10        # 跳过开头开始读取
11        if line.startswith('NODE_COORD_SECTION'):
12            begin = True
13        elif begin == True:
14            info = re.split('[ ]+', line.strip())
15            cities.append(city(info[0], float(info[1]),
16                                float(info[2])))
16 numOfCity = len(cities)
```

- 由于城市间的距离需要反复访问，所以在一开始就计算每两个城市的距离存入表中

```
1 # 计算每两个城市的距离
2 for i in range(numOfCity):
3     dis = []
4     for j in range(len(cities)):
5         dis.append(int(math.sqrt(math.pow(cities[i].x -
6         cities[j].x, 2) + math.pow(cities[i].y - cities[j].y, 2))))
7     distance.append(dis)
```

1. 局部搜索

- 先随机一条路径并计算当前总距离

```
1 path = [i for i in range(numOfCity)]
2 random.shuffle(path)
3 path.append(path[0])
4 # 计算当前的总距离
5 dis = 0.0
6 for i in range(len(path) - 1):
7     dis += distance[path[i]][path[i+1]]
```

- 接下来进行邻域操作
 - 随机交换两段序列

```

1 first = random.randint(1, len(path)-3)
2 second = random.randint(first+1, len(path)-2)
3 # 计算总距离的改变
4 dE = distance[path[first-1]][path[second]] +
      distance[path[first]][path[second+1]]
5
6       ] - distance[path[first-1]][path[first]] -
      distance[path[second]][path[second+1]]
7 # 若距离更短则改变有效
8 if dE < 0:
9     path[first:second+1] = path[second:first-1:-1]
10    dis = dis + dE

```

- 随机找到两段连续的序列后交换这两段序列的位置

```

1 first = random.randint(1, len(path)-4)
2 second = random.randint(first+1, len(path)-3)
3 third = random.randint(second+1, len(path)-2)
4 dE = distance[path[first-1]][path[second+1]] +
      distance[path[second]][path[third+1]] +
      distance[path[third]][path[first]] - distance[path[first-1]][path[first]] - distance[path[second]][path[second+1]]
      - distance[path[third]][path[third+1]]
5 if dE < 0:
6     path[first:third+1] =
7     path[second+1:third+1]+path[first:second+1]
8     dis = dis + dE

```

- 随机找到两段连续的序列后将后一段序列翻转并交换这两段序列的位置

```

1 first = random.randint(1, len(path)-4)
2 second = random.randint(first+1, len(path)-3)
3 third = random.randint(second+1, len(path)-2)
4 dE = distance[path[first-1]][path[third]] +
      distance[path[second+1]][path[first]] +
      distance[path[second]][path[third+1]] -
      distance[path[first-1]][path[first]] -
      distance[path[second]][path[second+1]] -
      distance[path[third]][path[third+1]]
5 if dE < 0:
6     path[first:third+1] =
      path[third:second:-1]+path[first:second+1]
7     dis = dis + dE

```

2. 模拟退火

大框架与前面一致，只是当新路径距离变大时以接受概率 $\exp\left(-\frac{dE}{T}\right)$ 更新，核心代码如下：

```

1 while(T > 0.001):
2     for i in range(1000):
3         # 邻域操作
4         first = random.randint(1, len(path)-3)
5         second = random.randint(first+1, len(path)-2)
6         dE = distance[path[first-1]][path[second]] +
7             distance[path[first]][path[second+1]] - distance[path[first-1]]
8             [path[first]] - distance[path[second]][path[second+1]]
9         # 若距离变小直接更新若距离变大则以接受概率更新
10        if dE < 0 or random.random() < math.exp(-dE / T):
11            path[first:second+1] = path[second:first-1:-1]
12            dis = dis + dE
13    # 降温
14    T *= alpha

```

其中邻域操作与前面一致，便不再叙述

3. 遗传算法

- 初始化种群

我原本是想直接随机生成种群的，但是课上听了同学们的分享之后知道这样会使收敛效果较差，于是采用贪心从而加快成熟速度，每个结点都选择下一个未被加入路径的最近结点

```
1 population = []
2 individual = [i for i in range(numOfCity)]
3 for r in range(int(POP_NUM * 2 / 10)):
4     random.shuffle(individual)
5     population.append(individual[:])
6 for r in range(int(POP_NUM - len(population))):
7     start = random.randint(0, numOfCity-1)
8     t = []
9     t.append(start)
10    j = 1
11    while j < numOfCity:
12        m = float_info.max
13        i, best = 0, 0
14        while i < numOfCity:
15            if (i not in t) and i != t[-1] and distance[t[-1]]
16            [i] < m:
17                best = i
18                m = distance[t[-1]][i]
19                i += 1
20            j = j + 1
21            t.append(best)
22        population.append(t[:])
23    random.shuffle(population)
```

- 接下来的主函数核心部分如下：


```

1  curGen = 0
2  while curGen < MAX_GEN:
3      random.shuffle(population)
4      # 选择
5      population = select(population)
6      # 交叉
7      population = OX(population, numOfCity)
8      # 变异, 即局部搜索
9      population = localSearch(population, numOfCity)
10     population.sort(key=lambda x: calFitness(x))

```

- 计算个体适应度, 适应度定义为总距离的倒数

```

1  def calFitness(individual):
2      fitness = 0.0
3      for i in range(len(individual) - 1):
4          fitness += distance[individual[i]][individual[i+1]]
5      fitness += distance[individual[len(individual)-1]]
6      [individual[0]]
7      return fitness

```

- 轮盘赌选择, 流程与上面介绍的步骤是一致的

```

1  def select(population):
2      new = []
3      best = float_info.max
4      best = 0
5      fitness = []
6      fitnessTotal = 0.0
7      # 计算个体的适应度
8      for i in range(POP_NUM):
9          fit = calFitness(population[i])
10         fitness.append(1 / fit)
11         fitnessTotal += 1 / fit
12         if (best > fit):
13             best = fit
14             best = i
15     new.append(population[best])

```

```

16     # 计算累计概率
17     p = []
18     for i in range(POP_NUM):
19         if i == 0:
20             p.append(fitness[i] / fitnessTotal)
21         else:
22             p.append(fitness[i] / fitnessTotal + p[i-1])
23     # 轮盘赌选择
24     for i in range(POP_NUM-1):
25         pro = random.random()
26         for j in range(POP_NUM):
27             if p[j] >= pro:
28                 new.append(population[j])
29                 break
30     return new

```

- 次序交叉

```

1  def OX(population, numOfCity):
2      sub = []
3      for i in range(POP_NUM):
4          if random.random() <= p1:
5              first = random.randint(0, POP_NUM - 1)
6              second = random.randint(0, POP_NUM - 1)
7              while first == second:
8                  second = random.randint(0, POP_NUM - 1)
9              start = random.randint(0, numOfCity - 2)
10             end = random.randint(start + 1, numOfCity - 1)
11             new1 = []
12             new2 = []
13             k = 0
14             for j in range(numOfCity):
15                 if j >= start and j < end:
16                     new1.append(population[first][j])
17                     j = end
18                 else:
19                     while k < numOfCity:
20                         if population[second][k] not in
population[first][start:end]:

```

```

21         new1.append(
22             population[second][k])
23         k += 1
24         break
25     k += 1
26     k = 0
27     for j in range(numOfCity):
28         if population[second][j] in population[first]
[29 start:end]:
30             new2.append(population[second][j])
31         else:
32             if k == start:
33                 k = end
34             new2.append(population[first][k])
35             k += 1
36         sub.append(new1[:])
37         sub.append(new2[:])
38     sub.sort(key=lambda x: calFitness(x))
39     for i in range(len(sub)):
40         for j in range(POP_NUM):
41             if calFitness(sub[i]) < calFitness(population[j]):
42                 population[j] = sub[i]
43                 break
44     return population

```

- 局部搜索
 - 随机交换两段序列

```

1 def localSearch(population, numOfCity):
2     for i in range(len(population)):
3         if random.random() <= p2:
4             best = population[i][:]
5             for _ in range(100):
6                 first = random.randint(1, numOfCity -
7 2)
8                 second = random.randint(first + 1,
9 numOfCity - 1)
10                population[i][first:second] =
11                population[i][second-1:first-1:-1]
12                if calFitness(best) >
13                calFitness(population[i]):
14                    best = population[i][:]
15                    population[i] = best
16            return population

```

- 随机找到两段连续的序列后交换这两段序列的位置

```

1 def localSearch(population, numOfCity):
2     for i in range(len(population)):
3         if random.random() <= p2:
4             best = population[i][:]
5             for _ in range(100):
6                 first = random.randint(1, numOfCity -
7 3)
8                 second = random.randint(first + 1,
9 numOfCity - 2)
10                third = random.randint(second + 1,
11 numOfCity - 1)
12                population[i][first:third] =
13                population[i][second + 1:third] + population[i]
14                [first:second + 1]
15                if calFitness(best) >
16                calFitness(population[i]):
17                    best = population[i][:]
18                    population[i] = best
19            return population

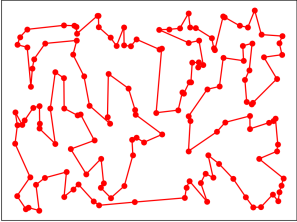
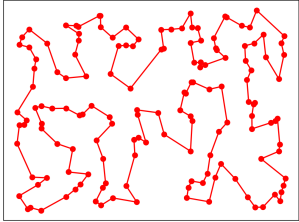
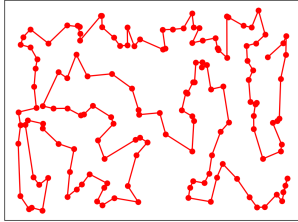
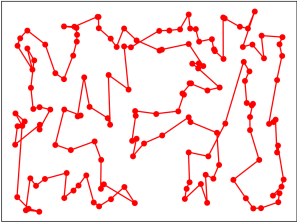
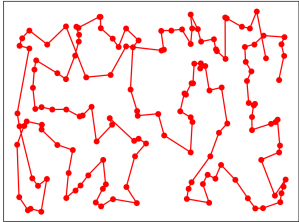
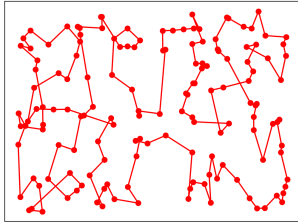
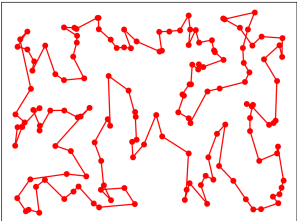
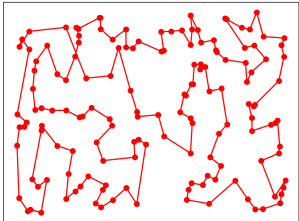
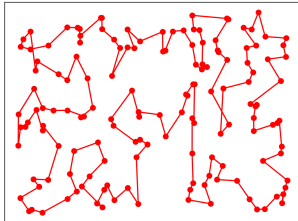
```

- 随机找到两段连续的序列后将后一段序列翻转并交换这两段序列的位置

```
1 def localSearch(population, numOfCity):
2     for i in range(len(population)):
3         if random.random() <= p2:
4             best = population[i][:]
5             for _ in range(100):
6                 first = random.randint(1, numOfCity -
7 3)
8                 second = random.randint(first + 1,
numOfCity - 2)
9                 third = random.randint(second + 1,
numOfCity - 1)
10                population[i][first:third] =
population[i][third - 1:second:-1]+population[i]
[first:second+1]
11                if calFitness(best) >
calFitness(population[i]):
12                    best = population[i][:]
13                    population[i] = best
14            return population
```

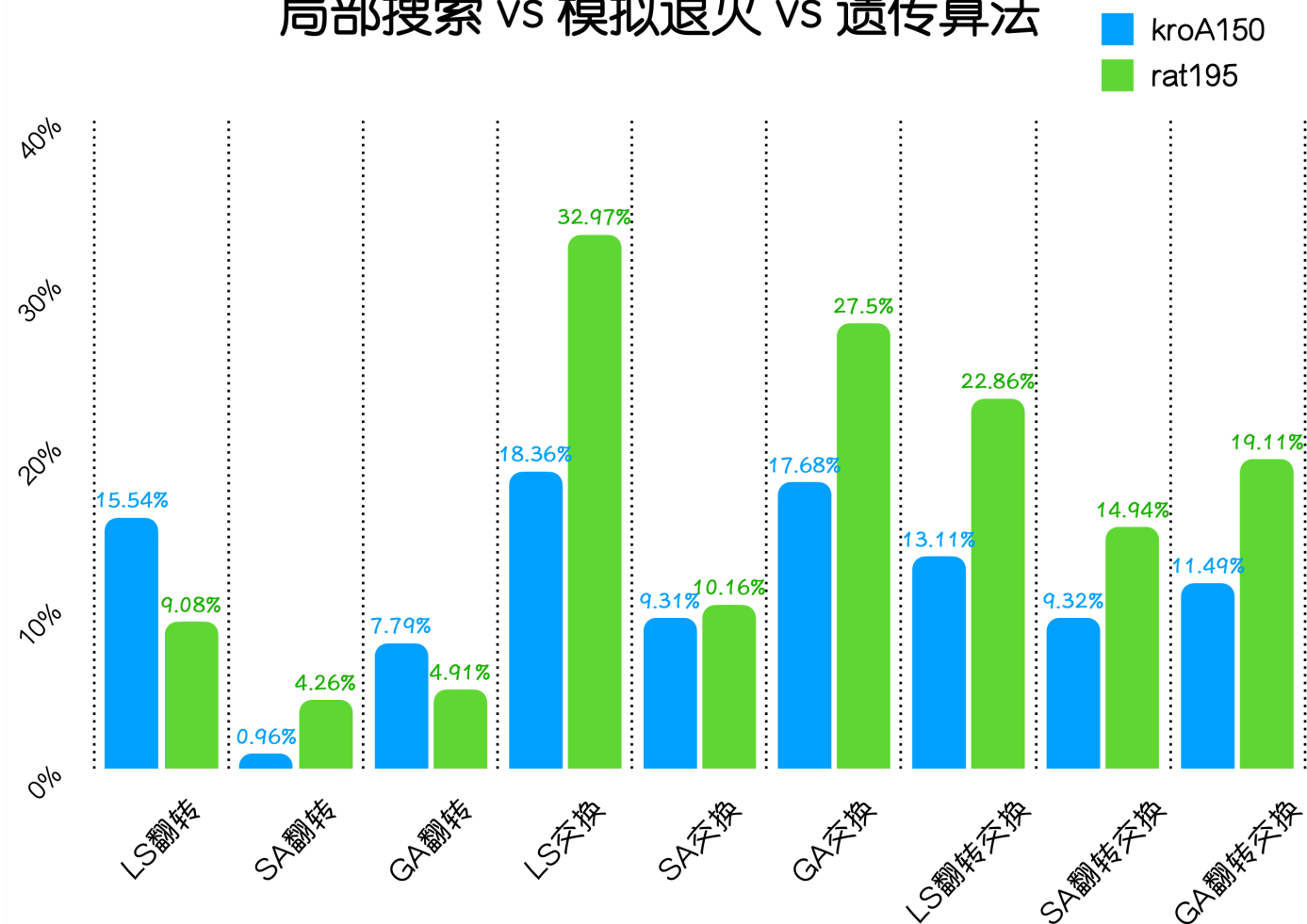
三、实验结果与分析

我使用了kroA150和rat195这两个数据集进行测试，对于三种方法的三种邻域操作在kroA150上的结果图如下所示：

邻域 操作	局部搜索	模拟退火	遗传算法
翻转			
交换			
翻转 交换			

分别计算误差率可得：

局部搜索 vs 模拟退火 vs 遗传算法



从上面的结果图可以得到以下结论：

- 对于三种邻域操作，翻转一段序列的效果最好，将后一段序列翻转并与前一段交换次之
- 对于三种算法，模拟退火是最优的，遗传算法次之，这是因为局部搜索很容易被限制在局部最优，至于遗传算法不如模拟退火的原因可能是交叉算法还不够好；但同时遗传算法的效率也是最低的

以上便是三种算法及三种邻域操作的对比。