

# 实验报告

——GA 算法解决 TSP 问题

姓名：钟文谦

学号：16340309

日期：2019.1.12

**摘要：**在 TSPLIB 中，选择一个大于 100 个城市数的 TSP 问题，使用遗传算法进行求解，设计较好的交叉操作以及和模拟退火类似的局部搜索操作进行变异，经过优化，程序的平均误差在 5% 以内，和我们之前的 SA 类似，比 SA 稍微差一些。

## 1. 引言

### 问题描述，问题背景介绍：

#### 问题描述：

在 TSPLIB 中选一个大于 100 个城市数的 TSP 问题，分别采用多种邻域操作的局部搜索和模拟退火进行求解，比较两种算法。

#### 问题背景：

旅行商问题，即 TSP 问题 (Traveling Salesman Problem) 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。

### 使用的方法

使用的方法即为遗传算法

### 方法的背景介绍

遗传算法 (GA) 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。

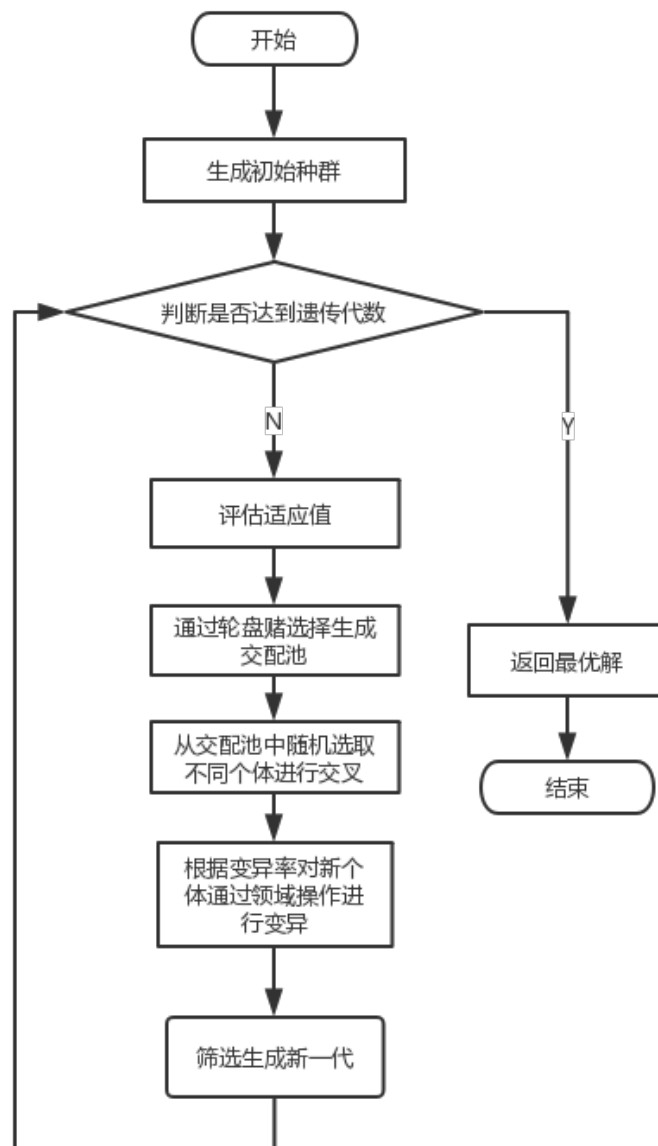
因此，在一开始需要实现从表现型到基因型的映射即编码工作。由于仿照基因编码的工作很复杂，我们需要对我们的问题的解进行简化，如二进制编码，初代种群产生之后，按照适者生存和优胜劣汰的原理，逐代演化产生出越来越好的近似解，在每一代，根据问题域中个体的适应度大小选择个体，并借助于自然遗传学的遗传算子进行组合交叉和变异，产生出代表新的解集

的种群。通过这个过程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可以作为问题近似最优解。

## 2. 实验过程






所用的具体的算法思想流程；

遗传算法的主要流程如下



上面就是程序的基本流程：

下面我们说说程序的具体实现  
程序的结构如下

- ▼  com.zhongwq
  -  City
  -  Convergence
  -  Solution
  - ▶  TourWindow.java
  -  TSP\_GA

City 即为抽象出的城市类，Convergence 是用于展示收敛数据的一个 JFrame，TourWindow 则是用于展示路线的一个 JFrame，Solution 是抽象出的一个解的结果的类，下面具体说说各个类

City.java，这个类较为简单，就是加了一个 getDist 的函数方便距离的计算，以及重载了 toString 函数用于结果的输出

```
public class City {
    int index;
    double x;
    double y;

    // Constructs a randomly placed city
    public City(){
        this.index = -1;
        this.x = 0;
        this.y = 0;
    }

    // Constructs a city at chosen x, y location
    public City(int index, double x, double y){
        this.index = index;
        this.x = x;
        this.y = y;
    }

    public int getIndex() { return index; }

    // Gets city's x coordinate
    public double getX() { return this.x; }

    // Gets city's y coordinate
    public double getY() { return this.y; }

    // Gets the distance to given city
    public double getDist(City city){
        double xDistance = Math.abs(getX() - city.getX());
        double yDistance = Math.abs(getY() - city.getY());
        double distance = Math.sqrt( (xDistance*xDistance) + (yDistance*yDistance) );
        return distance;
    }

    @Override
    public String toString() { return getIndex() + " (" + getX()+", "+getY() + ")"; }
}
```

Solution.java

这个类内容较多，这里只说说比较关键的一些函数

创建贪心初始解，贪心算法的策略是从所有城市中随机选取一个城市 x，放入结果的 ArrayList 中，然后从剩余的结点中选取离 x 最近的结点，继续放入 ArrayList，然后把该节点当做 x，从剩余的结点中选取离 x 最近的结点，不断运行该过程，直到所有点被放到 ArrayList 中，贪心解生成完成，经过测试发现贪心解生成的解平均的误差在 15%左右。

```

public void getGreedySolution() {
    int index = (int) (Math.random() * cities.size());
    boolean visited[] = new boolean[cities.size()];
    ArrayList<City> newCities = new ArrayList<>();
    visited[index] = true;
    double min;
    int minPos = -1;
    newCities.add(cities.get(index));
    while (newCities.size() < cities.size()) {
        min = Double.MAX_VALUE;
        minPos = -1;
        for (int i = 0; i < cities.size(); i++) {
            double dist = cities.get(index).getDist(cities.get(i));
            if (!visited[i] && dist < min) {
                min = dist;
                minPos = i;
            }
        }
        newCities.add(cities.get(minPos));
        visited[minPos] = true;
        index = minPos;
    }
    this.cities = newCities;
    this.cost = getDistance();
    System.out.println("Init: " + this.cost);
}

```

GA 的随机初始解的生成我则是通过直接传入一个经过 shuffle 的城市列表来初始化 Solution 来实现，所以这里没有实现

Convergence 和 TourWindow 则使用到了 java 的 Swing 来进行画图，因为与算法并没有关系，这里就不过多赘述了。

下面主要说说 GA 的各个模块的实现

1. 初始种群生成：初始化种群中，设置 5% 为贪心算法生成的解，设置 95% 为随机生成的解，具体实现如下

```

private void initSolutions(ArrayList<City> cities) { // 初始化种群
    this.solutions = new Solution[this.entity_number];
    for (int i = 0; i < (int) (this.entity_number * 0.95); i++) {
        Collections.shuffle(cities);
        while (isExist(cities, solutions)) {
            Collections.shuffle(cities);
        }
        this.solutions[i] = new Solution(cities);
        this.solutions[i].cost = this.solutions[i].getDistance();
    }
    for (int i = (int) (this.entity_number * 0.95); i < this.entity_number; i++) {
        this.solutions[i] = new Solution(cities);
        this.solutions[i].getGreedySolution();
    }
    this.setReproducePercent();
}

```

2. 适应值计算

```

public void setReproducePercent() { // 进行归一化
    double sumLengthToOne = 0.0;
    for (Solution solution : this.solutions) {
        sumLengthToOne += 1/solution.cost;
    }

    for (Solution solution : this.solutions) {
        solution.reproducePercent = (1 / solution.cost) / sumLengthToOne;
    }
}

```

由于我们是 cost 越小的子代越优秀，所以这里我使用了倒数作为评判期适应值的标准，个体的适应值即为其 cost 倒数除所有个体 cost 倒数之和

### 3. 轮盘赌选择父母亲

```
/**
 * 使用轮盘赌方法选择父母亲
 * @return
 */
private Solution getParentSolution() {
    Random random = new Random();
    double selectPercent = random.nextDouble();
    double distributionPercent = 0.0;
    for(int i = 0; i < this.solutions.length; ++i) {
        distributionPercent += this.solutions[i].reproducePercent;
        if(distributionPercent > selectPercent) {
            return this.solutions[i];
        }
    }
    return null;
}
```

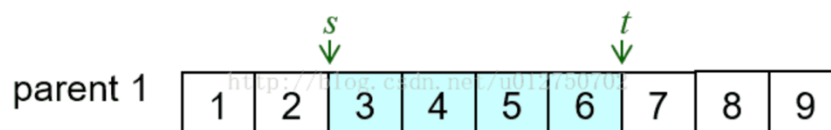
### 4. 交叉算法

这里交叉算法，我在书写程序的过程中用了很多种，最终选择了速度较快的 OrderCrossover，下面我分别说说各种交叉算法的实现

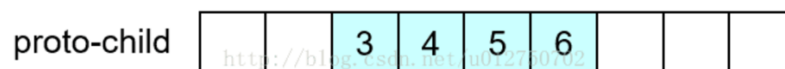
#### OrderCrossover

其交叉过程如下

1. 随机选择一对染色体（父代）中几个基因的起止位置（两染色体被选位置相同）



2. 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：



3. 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：



我们通过该交叉操作产生一个子代

```

private Solution OrderCrossOver(Solution father, Solution mother, int crossStartIndex, int crossEndIndex) {
    Solution result = father.clone();
    List<City> resultCities = new ArrayList<>();
    List<City> motherCities = new ArrayList<>();
    List<City> fatherContainCites = new ArrayList<>(); // 父亲保留下来的基因
    List<City> swapCites = new ArrayList<>();
    resultCities.addAll(result.cities);
    motherCities.addAll(mother.cities);
    for (int i = crossStartIndex; i <= crossEndIndex; i++) {
        fatherContainCites.add(resultCities.get(i));
    }
    int i = 0;
    while (i < city_number) {
        if (!fatherContainCites.contains(motherCities.get(i))) {
            swapCites.add(motherCities.get(i));
        }
        ++i;
    }
    i = 0;
    int j = 0;
    while (i < city_number) {
        if (i < crossStartIndex || i > crossEndIndex) {
            result.cities.set(i, swapCites.get(j++));
        }
        ++i;
    }
    if(Math.random() < this.variable_percent) { // 根据变异率, 对新一代进行变异
        this.oneVariable20pt(result);
    }
    return result;
}

```

## 无 Conflict 的交叉算法

以染色体为例子

染色体 R1 :    1 2 3 | 4 5 6 | 7 8

染色体 R2 :    5 1 8 | 3 4 7 | 6 2

交换后

R1 :    \_ \_ \_ | 3 4 7 | \_ \_    // 横线上依次应是 '1 2 3 7 8'

R2 :    \_ \_ \_ | 4 5 6 | \_ \_    // 横线上依次应是 '5 1 8 6 2'

现在我们把 R1 横线上应存在的数倒序放到 R2 的横线上 (R2 同理)。  
因此, R1、R2 就成了这个样子:

R1 :    2 6 8 | 3 4 7 | 1 5

R2 :    8 7 3 | 4 5 6 | 2 1

无冲突产生

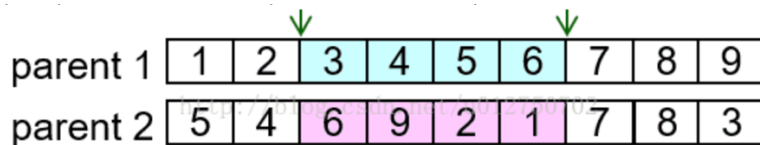
```

private Solution[] swapAndNoConflict(Solution father, Solution mother, int crossStartIndex, int crossEndIndex) {
    Solution[] newSolutions = new Solution[2];
    Solution fatherClone = father.clone();
    Solution motherClone = mother.clone();
    List<City> fatherCities = new ArrayList<>();
    List<City> motherCities = new ArrayList<>();
    fatherCities.addAll(fatherClone.cities);
    motherCities.addAll(motherClone.cities);
    int rightIndex = this.city_number - 1;
    for(int i = 0; i < father.cities.size(); ++i) {
        if(i < crossStartIndex || i > crossEndIndex) {
            while(rightIndex >= crossStartIndex && rightIndex <= crossEndIndex) {
                rightIndex--;
            }
            fatherClone.cities.set(i, motherCities.get(rightIndex));
            motherClone.cities.set(i, fatherCities.get(rightIndex));
            rightIndex--;
        } else {
            City temp = fatherClone.cities.get(i);
            fatherClone.cities.set(i, motherClone.cities.get(i));
            motherClone.cities.set(i, temp);
        }
    }
    newSolutions[0] = fatherClone;
    if(Math.random() < this.variable_percent) { // 根据变异率, 对新子代进行变异
        this.oneVariable20pt(newSolutions[0]);
    }
    newSolutions[1] = motherClone;
    if(Math.random() < this.variable_percent) { // 根据变异率, 对新子代进行变异
        this.oneVariable20pt(newSolutions[1]);
    }
    return newSolutions;
}

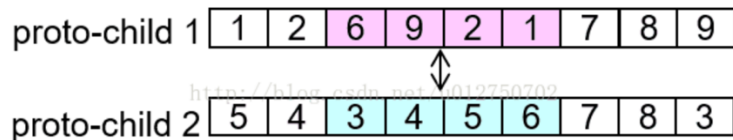
```

## 冲突解决的交叉操作

1. 随机选择一对染色体（父代）中几个基因的起止位置（两染色体被选位置相同）：



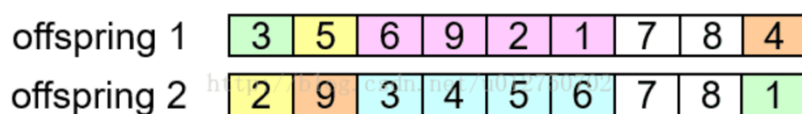
2. 交换这两组基因的位置:



3. 做冲突检测, 根据交换的两组基因建立一个映射关系, 如图所示, 以 1-6-3 这一映射关系为例, 可以看到第二步结果中子代 1 存在两个基因 1, 这时将其通过映射关系转变为基因 3, 以此类推至没有冲突为止。最后所有冲突的基因都会经过映射, 保证形成的新一对子代基因无冲突:



最终结果如下, 可以看到冲突被解决



## 代码实现

```
private Solution[] swapAndHandleConflict(Solution father, Solution mother, int crossStartIndex, int crossEndIndex) {
    Solution[] newEntities = new Solution[2];
    Solution fatherClone = father.clone();
    Solution motherClone = mother.clone();
    Map<Integer, City> fatherCityRelation = new HashMap<>();
    Map<Integer, City> motherCityRelation = new HashMap<>();
    for(int i = 0; i < this.city_number; ++i) {
        if(i >= crossStartIndex && i <= crossEndIndex) {
            City temp = fatherClone.cities.get(i);
            fatherClone.cities.set(i, motherClone.cities.get(i));
            motherClone.cities.set(i, temp); //交换
            fatherCityRelation.put(fatherClone.cities.get(i).index, motherClone.cities.get(i));
            motherCityRelation.put(motherClone.cities.get(i).index, fatherClone.cities.get(i));
        }
    }
    this.handleConflict(fatherClone, fatherCityRelation, crossStartIndex, crossEndIndex);
    this.handleConflict(motherClone, motherCityRelation, crossStartIndex, crossEndIndex);
    newEntities[0] = fatherClone;
    if(Math.random() < this.variable_percent) { // 根据变异率, 对新子代进行变异
        this.oneVariable2Opt(newEntities[0]);
    }
    newEntities[1] = motherClone;
    if(Math.random() < this.variable_percent) { // 根据变异率, 对新子代进行变异
        this.oneVariable2Opt(newEntities[1]);
    }
    return newEntities;
}

/**
 * 解决冲突, 把entity.cities中不在start-end区间内的city换成cityRelation中相同key对应的value
 * @param solution 可能存在冲突的个体
 * @param cityRelation 冲突对应关系
 * @param startIndex 起始位置
 * @param endIndex 截止位置
 */
private void handleConflict(Solution solution, Map<Integer, City> cityRelation, int startIndex, int endIndex) {
    while(!conflictExist(solution, cityRelation, startIndex, endIndex)) {
        for(int i = 0; i < solution.cities.size(); ++i) {
            if(i < startIndex || i > endIndex) {
                int temp = solution.cities.get(i).index;
                if(cityRelation.containsKey(temp)) {
                    solution.cities.set(i, cityRelation.get(temp));
                }
            }
        }
    }
}

/**
 * 是否存在冲突, 即entity.cities中是否出现过cityRelation中keySet中的任一元素
 * @param solution 可能存在冲突的实体
 * @param cityRelation 冲突对应关系
 * @param startIndex 起始位置
 * @param endIndex 截止位置
 * @return true-存在 / false-不存在
 */
private boolean conflictExist(Solution solution, Map<Integer, City> cityRelation, int startIndex, int endIndex) {
    for(int i = 0; i < solution.cities.size(); ++i) {
        if(i < startIndex || i > endIndex) {
            if(cityRelation.containsKey(solution.cities.get(i).index)) {
                return true;
            }
        }
    }
    return false;
}
```

以上就是我使用的三种交叉操作, 后来经过比较, 选择了 OrderCrossover 作为最后的交叉操作。

## 5. 变异

变异操作为了方便我采用的是结合到交叉生成新子代的过程中, 根据概率判断是否对新子代采取变异操作, 变异操作则采取的是 2-opt 操作, 实现如下

```
private void oneVariable2Opt(Solution solution) {
    Random random = new Random();
    int index1 = random.nextInt(this.city_number);
    int index2 = random.nextInt(this.city_number);
    while(index1 == index2) {
        index2 = random.nextInt(this.city_number); //保证突变基因不是同一个
    }
    if (index1 > index2) {
        int tmp = index1;
        index1 = index2;
        index2 = tmp;
    }
    while (index1 < index2) {
        City temp = solution.cities.get(index1);
        solution.cities.set(index1, solution.cities.get(index2));
        solution.cities.set(index2, temp);
        index1++;
        index2--;
    }
}
```



到这里基本的模块就完成了，通过在每一代演化中按照流程图调用它们，就可以完成遗传算法的实现了。

3. 结果分析

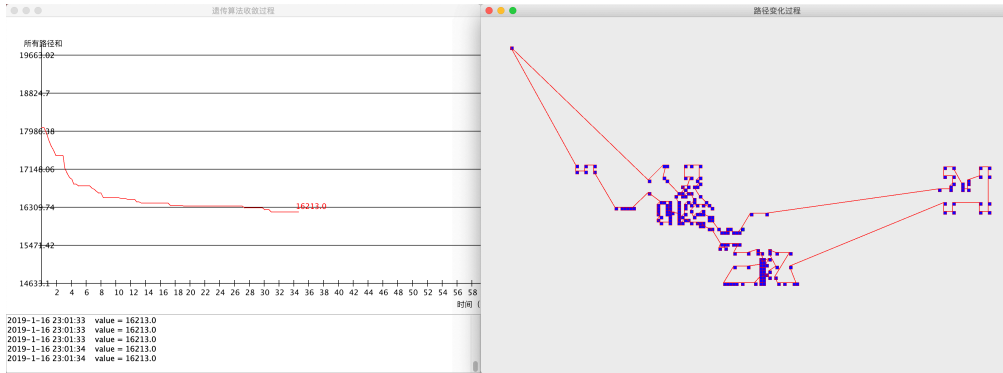
实验环境为：

操作系统：MacOS  
语言环境：java 1.8  
IDE：Intellij IDEA

算法参数

变异率	0.01
遗传代数	10000
种群大小	200

实验运行截图(左边为收敛的过程，右边为生成的路线)



算法运行结果，与模拟退火算法进行比较

算法	遗传算法		模拟退火算法	
	结果	误差	结果	误差
1	16204	2.69%	16086	1.94%
2	16252	2.99%	15874	0.60%
3	16260	3.04%	16193	2.62%
4	16882	6.98%	16108	2.08%
5	16186	2.57%	15998	1.38%
6	16232	2.86%	16165	2.44%
7	16987	7.65%	16075	1.87%
8	16535	4.78%	15911	0.83%
9	16112	2.10%	16585	5.10%
10	16280	3.17%	16020	1.52%
最优解	最好解(误差)	平均误差(标准差)	最好解(误差)	平均误差(标准差)
15780	16112(2.10%)	3.88%(1.94%)	15874(0.60%)	2.04%(1.25%)

首先我们单独对遗传算法得到的结果进行分析，经过优化(使用贪心算法生成 5%初始解)后，种群内一开始就有了比较好的解，误差大约在 15%左右，然后我们再通过遗传算法，这时我们解的误差稳定进入 10%，甚至大部分情况下在 5%以内，效果较好。

### 与模拟退火算法进行比较

1. 经过优化后，解的质量方面还是没有模拟退火算法好
2. 运行时间的话，遗传算法的运行时间更长，

### 设计经验

1. 交叉操作：为了程序的运行效果更好，我们需要针对我们的问题，设计一个较好的交叉操作，为了实现这一点，我们可以实现多种交叉操作，比较它们的实现效果，最终选取一个最为合适的交叉操作。
2. 初始种群生成：我们可以使用贪心算法生成部分初始解，通过贪心算法生成部分初始解就相当于在种群初期我们引入了优良基因，之后我们再在这之上进行遗传算法，就可以得到更为优质的解了，也正因为如此，我们的解稳定进入 10%以内，甚至经常到达 5%以内。
3. 如果想解的效果更佳，我们可以适当调整种群的大小，种群规模越大越可能找到全局解，但运行时间也相对较长，所以，我们需要权衡解的质量以及运行时间。

### 单点搜索和多点搜索的优缺点

#### 多点搜索

优点：有较好的全局搜索性能，更加不容易陷入局部最优

缺点：收敛比较慢，局部搜索能力较弱，运行时间较长，容易受参数影响。

#### 单点搜索

优点：局部搜索能力较强，运行时间较短

缺点：全局搜索能力差，同样容易受参数影响

## 4. 结论

### 通过这次实验

1. 我熟悉了遗传算法的实现原理、运作过程，能够使用遗传算法解决一些实际的问题
2. 了解了如何较好地对实际问题进行建模，了解了如何较好将解进行编码成为染色体。
3. 经过优化算法的过程，我了解到了如何较好地优化我们的交叉操作，选择一个适合我们具体情况的解。
4. 知道了如何通过一些优化操作，使程序可以得到更好的解，就像通过贪心算法生成部分初始解。
5. 知道了如何合理调整变异率、种群大小等参数，使程序运行效果更佳。