实验报告

－－－－LS&SA 解决 TSP 问题

姓名：钟文谦　　学号：16340309　　日期：2019.1.10

摘要：在 TSPLIB 中，选择一个大于 100 个城市数的 TSP 问题，使用多邻域操作的局部搜索 local search 策略求解，和加入模拟退火策略的搜索解决该问题，最终比较两者效果，发现由于模拟退火允许程序跳出局部最优解，所以模拟退火算法的结果更加优秀。

1．导言

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。

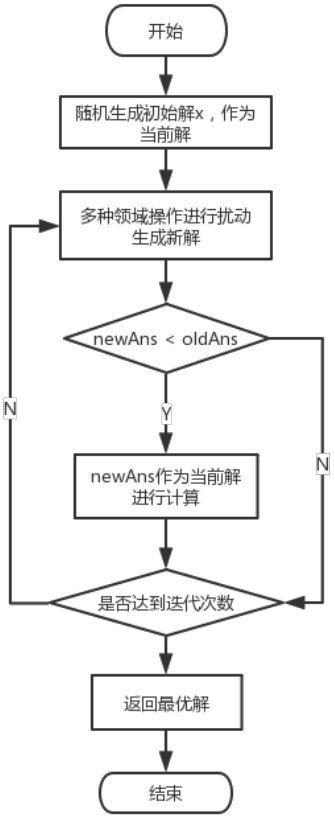
使用的方法，方法的背景介绍：

- 局部搜索算法：每次随机生成领域解，判断是否小于当前解，小于则不接受，大于则接收其作为当前解，所以容易陷入局部最优，又名爬山算法。
- 加入模拟退火策略的随机搜索算法: 加入模拟退火策略后，相比局部搜索算法，其可以一定概率地接收比当前解更差的解，从而拥有跳出局部最优的能力，最终得到一个全局来说的更好的解。

2．实验过程

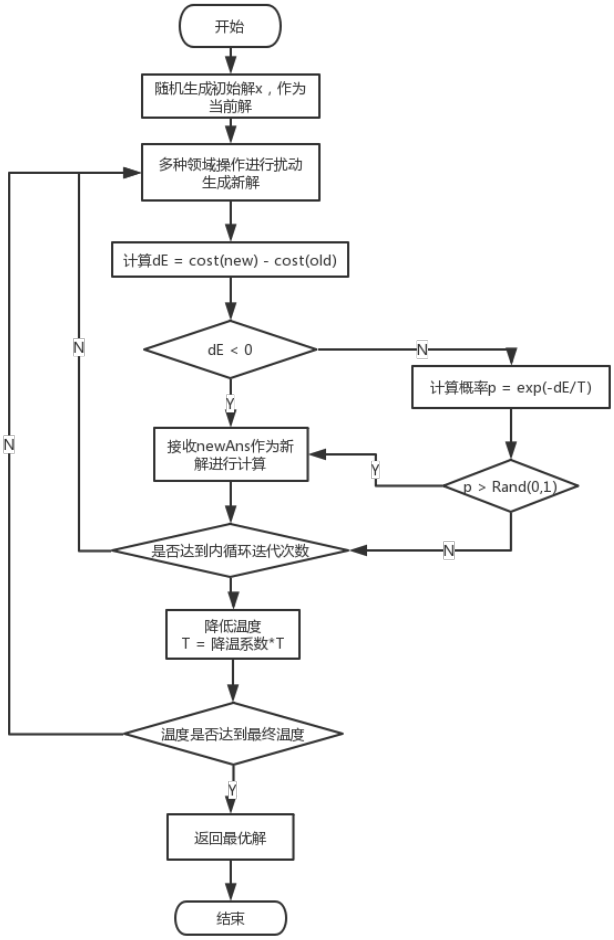
所用的具体的算法思想流程；

局部搜索的算法过程: 其算法流程框图如下



可以看到在局部搜索操作中，算法只接收比当前解更优的解作为当前解进行计算，实验效果在下面说明

模拟退火算法流程框图如下



可以看到模拟退火算法相比局部搜索，加入了温度的控制，以及根据概率接受 dE>0 即生成比当前解更差的解时，根据 p = exp(-dE/T)概率，随机生成 0-1 的浮点数，判断是否接受新解，让算法拥有跳出局部最优解的能力，结果比起刚刚的爬山算法，效果更加优秀。

在具体实现当中，由于为了比较两算法，所以我让他们的迭代次数一致，局部搜索使用温度以及内循环来决定迭代次数，但是只接受比当前解更好的解来实现局部搜索算法。实现两算法的比较。

具体实现:

具体实现中，代码结构如下:

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

创建随机初始解

```
/**
 * 创建一个随机初始解
 */
public Solution getRandomInitial() {
    for (int cityIndex = 0; cityIndex < citiesList.size(); cityIndex++) {
        setCity(cityIndex, this.getCity(cityIndex));
    }
    Collections.shuffle(citiesList);
    return this;
}
```

通过 java 提供的 Collections.shuffle(citiesList)随机打乱城市序列来进行随机初始解的生成

扰动获取邻域解，这里用到了两种邻域操作，分别是 2-opt 和随机交换两个点，每次获取时随机选择一种运行

```
/**
 * 获取邻居
 * @return
 */
public Solution generateNeighour() {
    int choice = (int) (2 * Math.random());
    switch (choice) {
        case 0:
            return generateNeighourTourWith2Opt();
        case 1:
            return generateNeighourTourWithExchange();
    }
    return null;
}
```

领域操作:
交换两个城市

```
/**
 * 领域操作: 交换两个城市
 * */
public Solution generateNeighourTourWithExchange(){
    Solution newSolution = new Solution(this.citiesList);
    int tourPos1 = (int) (newSolution.numberOfCities() * Math.random());
    int tourPos2 = (int) (newSolution.numberOfCities() * Math.random());
    City citySwap1 = newSolution.getCity(tourPos1);
    City citySwap2 = newSolution.getCity(tourPos2);
    newSolution.setCity(tourPos2, citySwap1);
    newSolution.setCity(tourPos1, citySwap2);

    return newSolution;
}
```

2-opt

```
/**
 * 领域操作: 2-opt
 * @return
 */
public Solution generateNeighourTourWith2Opt(){
    Solution newSolution = new Solution(this.citiesList);
    // Get a random positions in the tour
    int tourPos1 = (int) (newSolution.numberOfCities() * Math.random());
    int tourPos2 = (int) (newSolution.numberOfCities() * Math.random());
    while (tourPos1 == tourPos2) {
        tourPos1 = (int) (newSolution.numberOfCities() * Math.random());
        tourPos2 = (int) (newSolution.numberOfCities() * Math.random());
    }
    if (tourPos1 > tourPos2) {
        int tmp = tourPos1;
        tourPos1 = tourPos2;
        tourPos2 = tmp;
    }

    while (tourPos1 < tourPos2) {
        // Get the cities at selected positions in the tour
        City citySwap1 = newSolution.getCity(tourPos1);
        City citySwap2 = newSolution.getCity(tourPos2);

        // Swap them
        newSolution.setCity(tourPos2, citySwap1);
        newSolution.setCity(tourPos1, citySwap2);

        tourPos1++;
        tourPos2--;
    }

    return newSolution;
}
```

其他一些获取距离的简单的函数这里不多进行说明

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。

下面说说主要的 LS 和 SA 的实现，SA.java
这里主要说说核心的算法部分

对于 SA, 关键的两层循环

```
// Loop until system has cooled
while (currentTemperature > minTemperature) {
    for (int i = 0; i < internalLoop; i++) {
        // 通过多邻域操作获取新解
        newSolution = currentSolution.generateNeighour();
        // 获得新解的cost
        int currentEnergy = currentSolution.getDistance();
        int neighbourEnergy = newSolution.getDistance();

        // 根据概率查看是否接收新解作为startPoint
        if (acceptanceProbability(currentEnergy, neighbourEnergy,
                currentTemperature) > Math.random()) {
            currentSolution = new Solution(newSolution.getCitiesList());
        }

        if (currentSolution.getDistance() < bestSolution.getDistance()) {
            bestSolution = new Solution(currentSolution.getCitiesList());
        }
        ++count;
    }
    currentTemperature *= coolingRate; // 降温

    // 显示数据
    long millis = System.currentTimeMillis();
    if (show && millis - tp > 10) {
        tp = millis;
        convergence.addData(millis, bestSolution.getDistance(), currentSolution.getDistance());
        tourWindow.showTour(bestSolution);
    }
}
```

计算接受概率(区别 LS 和 SA)

```
/**
 * 计算接收概率
 **/
private double acceptanceProbability(int energy, int newEnergy, double temperature) {
    // 如果新解更优, 100%接收
    if (newEnergy < energy) {
        return 1.0;
    }
    // 如果新解的cost更高, 返回接收概率
    if (SAsearch)
        return Math.exp((energy - newEnergy) / temperature);  // 模拟退火时
    return 0; // 爬山算法时, 不接受差解
}
```

这里就做了一个条件判断，当程序不接受差解，即面对差解时返回概率 0 时，程序运行的就是局部搜索算法(爬山法)，而当程序面对差解时返回 p = exp(-dE/T)时，此时程序运行的就是模拟退火算法，通过调整创建类时的变量就可以控制该程序运行的方式，这样保证了迭代次数的一致性，也很方便实现了两个算法。

通过这几个类的实现，我们就实现了我们的局部搜索算法以及模拟退火算法了。

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

算法的各个参数: 最终经过调整参数，模拟退火算法的参数如下

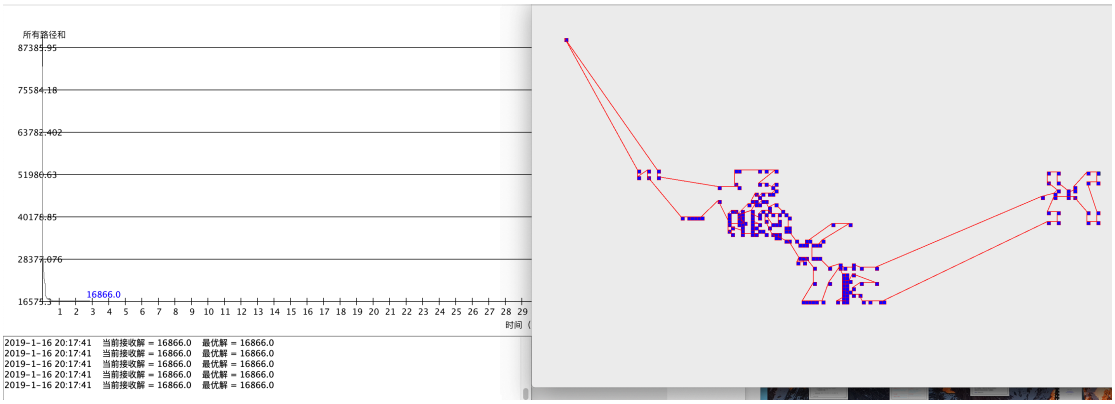
初始温度	100
结束温度	0.01
内循环次数	1000
降温系数	0.99

实验结果的可视化

左边窗口灰色线表示接受解的大小，红色线表示当前的最优解

LS 算法

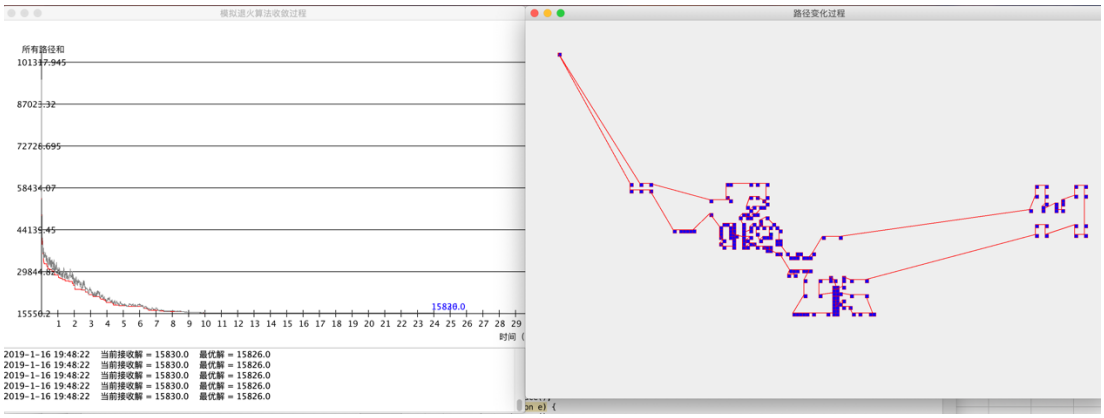
(左边窗口灰色线与红色线重合，说明局部搜索不接受比当前解差的解)



SA 算法

为了方便体现，模拟退火接受差解，这里调大了降温系数，调整为 0.999，让程序收敛的更慢，更能体现模拟退火算法接受差解的特性

灰色代表接受的解的大小，红色代表当前最优解的大小，左图表示程序接受比当前解更差的解，从上面可以看出，随着迭代次数增加，温度降低，程序接受差解的概率降低，到后面几乎不接受差解



算法运行过程分析

收敛速度

阶段	局部搜索	模拟退火算法
迭代初期	收敛速度快	收敛速度慢，温度高时，跳出局部最优概率大，接受恶化解的概率大。
迭代中期	收敛速度逐渐变慢，并且逐渐陷入局部最优，但是我们的 2-opt 和交换两个的邻域操作对其进行了扰动，有可能跳出局部最优，但是概率较小	温度下降，接收恶化解的概率下降，收敛速度开始加快
迭代后期	基本陷入局部最优，解基本稳定	由于到了迭代后期，温度较低，程序不易接受差解，解基本稳定

程序运行若干次(10 次)，统计运行结果，结果如下

算法	局部搜索		模拟退火算法	
	结果	误差	结果	误差
1	16455	4.28%	16086	1.94%
2	16622	5.34%	15874	0.60%
3	16637	5.43%	16193	2.62%
4	16292	3.24%	16108	2.08%
5	16900	7.10%	15998	1.38%

6	16927	7.27%	16165	2.44%
7	16909	7.15%	16075	1.87%
8	16667	5.62%	15911	0.83%
9	16485	4.47%	16585	5.10%
10	16593	5.15%	16020	1.52%
测例最优解	最好解	平均误差(标准差)	最好解	平均误差(标准差)
15780	16292(3.24%)	5.51%(1.34%)	15874(0.60%)	2.04%(1.25%)

程序每次运行会迭代运行 10 次，输出每次的结果并给出最后 10 次的平均距离最优解的误差。

实现效果来说，爬山法稳定在 10%以内，而模拟退火算法基本在 3%以内，少数情况超出 3%, 实现效果还是可以的。

结果分析:

由上面的结果可以看到，无论是局部搜索(爬山法)还是模拟退火算法，它们的解都稳定在 10%以内，但是可以看到，局部搜索得到的解的效果比起模拟退火算法要稍微差一些, 因为局部搜索算法较为容易陷入局部最优且由于其不接受差解较难跳出局部最优。而模拟退火算法通过差解可以跳出局部最优，所以结果相对比局部搜索算法要好上一些。

算法速度来说，爬山算法收敛得更快能在 0.5s 左右达到收敛，模拟退火算法由于其接受差解，能在 1.5s 左右收敛，总体运行时间来说，由于程序循环次数固定，所以运行时间来说都在 3s 左右。

4．结论

通过这次实验

1. 我熟悉了局部搜索算法和模拟退火算法的实现过程，能够通过局部搜索算法和模拟退火算法解决实际的问题
2. 通过比较局部搜索算法以及模拟退火算法，我了解到了局部搜索与模拟退火算法的区别，了解了这个简单的接受概率对于程序运行的影响，也了解到了其在每一个过程(迭代初期、迭代中期、迭代晚期)，两个算法的运行特征，这在我们上面分析的时候已经描述过了。这里不多赘述
3. 懂得了一些模拟退火算法的调整参数的技巧，知道了如何通过合理调整参数，在较短的时间内让程序收敛，获得一个较好的解。