中南大學

CENTRAL SOUTH UNIVERSITY

B 题

基于 K—均值聚类算法的最优站点分布方案

	参赛队员1	参赛队员2	参赛队员3
姓名	张贵	肖涛	赵洪俊
学院	信息科学与工程学院	信息科学与工程学院	信息科学与工程学院
专业班级	通信1601	通信1602	通信1601
学号	0905160120	0905160212	0905160121
电话号码	13117411881	18273415912	18711092134

B 题 基于 K—均值聚类算法的最优站点分布方案

摘要：

Relay无线回传方案可以解决城区、农网等场景下的传统传输方式不可达的问题，同时在部分场景下也可以替代微波，有效降低站高，节省加站费用，本文针对relay无线回传方案，在给定一个地区中候选站点的位置分布的情况下，提出了一个成本最优的部站方案。选取经纬度范围适中的一千个站点分布，并以K均值聚类算法为依据进行最优站点模型的求解。

一千个站点随机分布，为选取合适的地区范围。对于经纬度范围的选取进行评估，在此不考虑各站点覆盖面积。以站点间距离的合理性作为评判依据，以各站点间的距离作为直接衡量指标，在三个不同的有代表性经纬度区域范围内选取最佳的区域范围，根据实际站点分布合理性在首跳距离和之后每跳距离给定的前提下，可判断合理性高，实际意义大的为选取三。

在以上背景下，根据生成的一千个站点，在约束条件

$$\mathrm{Fd} \le 20,\ \mathrm{Nd} \le 10,\ \ \mathrm{Num1} \le 4,\ \mathrm{Num2} \le 2,\ \ \mathrm{Sec} \le 2,\ Max \le 6$$

分别以总成本和回 $f(a)$ 传损耗为目 PL 标函数

$$f(a) = 10\ a_1 + 5a_2 + 50\ a_3,\ \ \mathrm{PL} = 32.5 + 20\lg(D_{ij}) + 20\lg(900)$$

建立了一个最优站点模型并求解了这个模型。

该模型的算法思想核心是 K 均值聚类和图的拓扑算法，先随机生成 1000 个子站点，然后利用聚类生成宿主站，K 均值聚类主要用在分站点为以宿主站为中心包含合适数量的子站的若干个簇；拓扑算法主要用于连接每个簇内的子站和宿主站，对结果集中的簇进行遍历，对每一个簇，先遍历每个子站并得到其与宿主站之间的距离，取距离最小的四个子站作为首跳，然后遍历剩下的子站，对于每一个剩下的子站，得到其与首跳的四个子站的距离，并与距离最小的那个首跳子站建立连接。

该模型基于 K—均值聚类算法和图的拓扑算法，与实际紧密联系，故具有推广意义，在实际应用时，该模型主要适用于人口分布较均匀且密度不宜过大的地区。

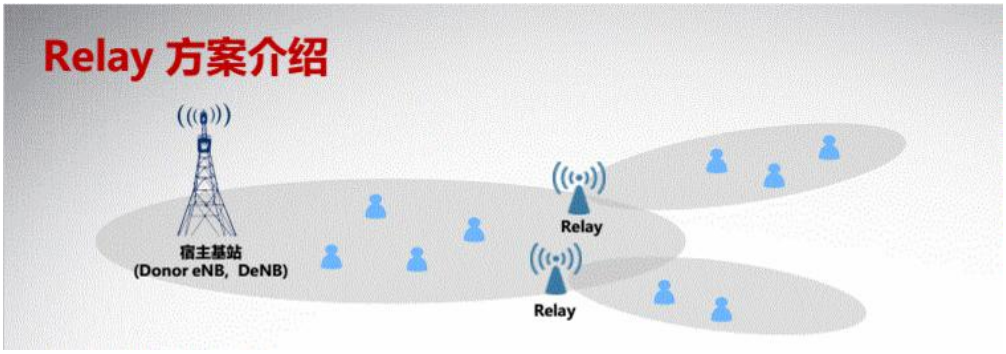
关键词：经纬度范围、拓扑算法、K 均值聚类算法、最优站点模型

一、问题概述

1、问题背景

在城区建设基站，传输光纤部署最后一公里的成本高，光纤到站率低，全球综合来看低于 60%；如果使用微波传输，由于微波只能在 LOS（视距）场景下部署，而城区场景中 LOS 信道比例低于 50%。在农村网建设基站，单站业务量低，收入低，ROI（投资回报率）差，运营商建站对成本较为敏感。卫星传输租金、光纤传输建设费用对于运营商是很大的负担，而如果使用微波传输，对于相当一部分站点需要提升铁塔高度来满足微波的 LOS 场景要求，铁塔费用的增加对于运营商来说同样是不小的负担。

Relay 无线回传方案利用 FDD LTE 或 TDDLTE 制式承载来为站点回传，相对微波有较强的 NLOS（非视距）传输能力，可以解决城区、农网等场景下的传统传输方式不可达的问题，同时在部分场景下也可以替代微波，有效降低站高，节省加站费用。本题是针对 relay 无线回传方案，在给定一个地区中候选站点的位置分布的情况下，根据站点间的相互位置、站点间拓扑关系限制等条件，以及宿主站与子站的距离是否满足某门限来判断是否满足最低回传质量要求的前提下，要求提出一个成本最优的部站方案。



2、问题重述

（1）题目要求：在给定一个地区中候选站点的位置分布的情况下，根据站点间的相互位置、站点间拓扑关系限制等条件，以及宿主站与子站的距离是否满足某门限来判断是否满足最低回传质量要求的前提下，设计成本最优的部站方案。

（2）在该部站方案中应有以下问题解决：

a. 候选站点是安装子站还是宿主站

b. 候选站点间的连接关系

（3）优化目标

a. 更低的总体成本和平均成本

b. 更低的回传路径损耗和系统平均损耗

二、模型假设

1、 假设不考虑各站点覆盖面积及总覆盖面积
2、假设不考虑影响回传质量的因素包括距离，地形阻挡，普通手机接入影响、ReBTS 干扰，相邻基站干扰等多种复杂因素
3、 假设宿主站均为蝴蝶站（以下简称 H 站）

三、符号说明

符号说明如下表 1 所示

符号	说明
D_{ij}	生成的各站点间的距离
Lat	站点纬度
Lng	站点经度
F_d	首跳距离
N_d	之后每跳距离
Num_1	宿主站每个扇区第一级最大接入子站数
Num_2	宿主站每个扇区第二级最大接入子站数
Max	宿主站每个扇区最大接入子站数
Sec	宿主站扇区数
a_1	宿主站数量
a_2	子站数量
a_3	卫星数量
$f(a)$	总体成本目标函数
PL	路径损耗
$Times$	运行时间

（表 1）

四、问题分析

1、问题总要求：在给定一个地区中候选站点的位置分布的情况下，根据站点间的相互位置、站点间拓扑关系限制等条件，在满足一定回传质量的前提下，设计成本最优的部站方案，包括①候选站点类型②候选站点间的连接关系，要求结合现网中对于无线回传拓扑规划问题的具体需求，尽可能使算法收敛速度快、尽可

能覆盖更多的站点。

2、对于问题（2），该题目的核心问题，需要确定一千个站点的站型，连接关系，并满足如下限制条件：

a. 以各站点宿主站与子站连接的首跳距离不大于 20KM，子站与子站间的连接每跳距离不大于 10KM
b. 站点包含 RuralStar 和蝴蝶站两种不同站型；其中，RuralStar 共包含 1 个扇区，蝴蝶站共包含 2 个扇区；若该站点为宿主站，则每个扇区第一级最大接入子站数 4，最大总接入子站数 6；为了简化问题，暂不考虑蝴蝶站的扇区覆盖方向；
c. 宿主站之间采用微波连接，最大通信距离为 50KM
d. 宿主站和子站以及子站之间采用无线回传连接
e. 每个子站最多只能有 2 条无线回传连接；
f. 任意子站只能归属一个宿主站，到达所属宿主站有且只有一条通路，且该通路包含的跳数小于等于 3
g. 任意宿主站都有且只有一颗卫星负责回传，成片连接的宿主站可共享同一颗卫星，但一颗卫星最多只能负担 8 个成片宿主站的回传数据
h. 成片宿主站中，宿主站总数不设上限

4、 对于问题（3），此问题为挑战目标，一是更低的总体成本，二是更低的回传路径损耗。

a.总体成本：宿主站数量*宿主站成本+子站数量*子站成本+卫星数量*卫星成本

平均成本=总体成本/地区内站点总数

①这里，卫星的数量等于 Ceil(宿主站数量/8)，Ceil()表示向上取整。表 2 为各种传输方式的成本，单位：W USD

宿主站成本	10
子站成本	5
卫星成本	50

（表 2）

②为满足成本最低的要求，我们先规定生成站点中有 1000 个子站，这样在确定子站个数的前提下，子站成本固定，然后尽量减少宿主站和卫星的个数以达到减小总成本的目的。

b. 为了简化问题，采用自由空间传播模型估计站点之间的路径损耗，公式如下：

$$\mathrm{PL} = 32.5 + 20\lg(D) + 20\lg(\mathrm{F})$$

系统平均损耗=所有无线回传连接的损耗之和/无线回传连接数

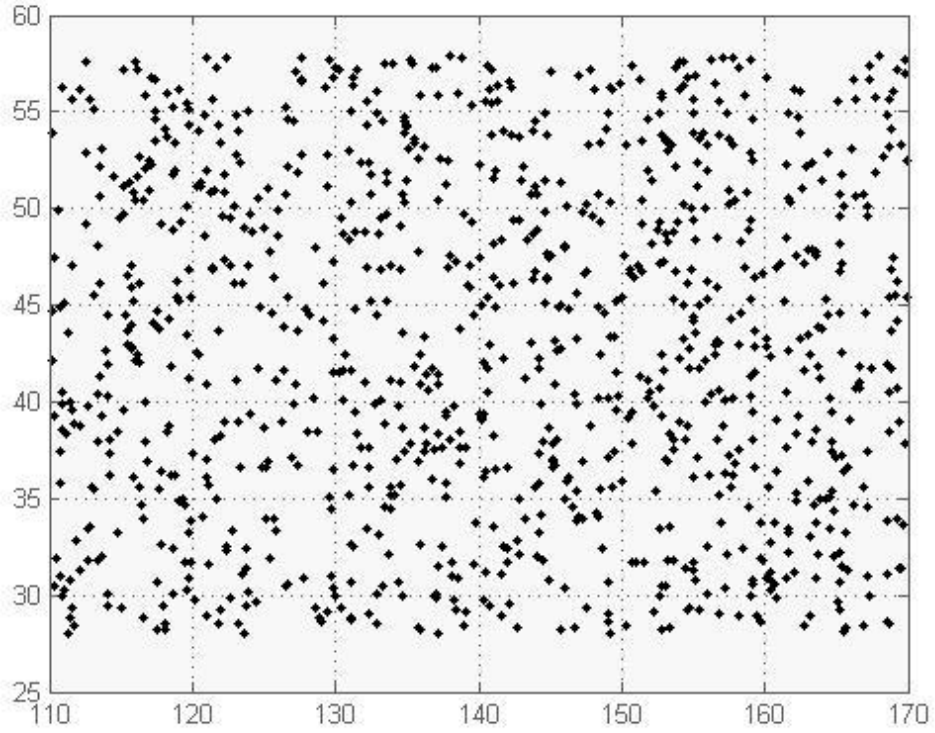
①其中，PL 是路径损耗，是两个站点之间的距离，D 单位为 km，F 是发射频率，默认采用 900MHz；且该路径损耗只考虑子站回传部分，宿主站之间采用微波传输，只需满足距离限制，不计算该损耗。

②为减小回传路径损耗，由于宿主站直接采用微波连接，只要模型能尽可能满足宿主站之间距离满足约束条件，就不考虑它们之间的损耗。那么损耗就只在宿主站与子站及子站与子站间。而回传损耗的唯一决策变量是距离，因此，模型尽可能选取宿主站与子站及子站与子站的距离小的进行连接就能减小回传路径损耗。

五、模型的建立与求解

（一）一千个站点分布范围的确定

一千个站点随机分布，为选取合适的地区范围。对于经纬度范围的选取进行评估，在此不考虑个站点覆盖面积。以站点间距离的合理性作为评判依据，以各站点间的距离作为直接衡量指标，在三个不同的有代表性经纬度区域范围内选取最佳的区域范围，根据实际站点分布合理性在首跳距离和之后每跳距离给定的前提下，选取可判断合理性高，实际意义大的分布范围。



以站点间距离的合理性作为评判依据，以各站点间的距离作为直接衡量指标，在三个不同的有代表性经纬度区域范围内选取最佳的区域范围，根据实际站点分布合理性将站点间距离分为三个等级：

A. 250<=Dij<=650　　超大范围，不合理
B. 30<=Dij<=180　　大范围，合理性低
C. 0<Dij<=45　　适中范围，合理性高

再根据题目提供的球面距离公式，随机抽取一部分靠近的站点间计算两点间距离来判断经纬度的合理性。

1、选取一

（1）纬度范围 30.0，经度范围 60.0

Lat∈（28,58）

Lng∈（70,130）

（2）部分站点距离如下表 3-1：

Dij	距离（KM）	随机站点	距离（KM）
D12	335.0939	D23	445.9998
D13	473.4039	D24	357.1523
D14	288.7309	D25	530.0145
D15	306.8313	D26	346.7938
D16	402.0794	D27	445.9998
D17	389.3069	D28	638.6621
D18	454.8875	D34	497.5586

（表 3-1）

2、 选取二

（1）纬度范围 1.0，经度范围 5.0

Lat∈（27.0,28.0）

Lng∈（112.0,117.0）

（2）部分站点距离如下表 3-2：

Dij	距离（KM）	随机站点	距离（KM）
D12	43.1947	D23	148.3897
D13	65.6223	D24	41.2452
D14	76.8667	D25	32.3092
D15	47.8509	D26	112.7521
D16	120.8751	D27	85.5627
D17	86.6708	D28	52.6257
D18	115.5703	D34	74.0712

（表 3-2）

3、 选取三

（1）纬度范围 0.3，经度范围 1.0

Lat∈（28.1，28.4）

Lng∈（112.9，113.9）

（2）部分站点距离如下表 3-3：

Dij	距离（KM）	随机站点	距离（KM）
D12	13.373	D23	3.6315
D13	29.1703	D24	17.979
D14	14.8433	D25	5.9615
D15	6.1088	D26	16.1875
D16	5.0148	D27	31.0237
D17	4.5979	D28	4.8564
D18	11.1287	D34	12.6999

（表 3-3）

综上，根据表格数据，在选取一，纬度范围 30.0，经度范围 60.0 的情况下，随机 14 个站点距离的平均值为 429.4653；在选取二，纬度范围 1.0，经度范围 5.0，上述指定 14 个站点距离的平均值为 78.8290；在选取三，纬度范围 0.3，经度范围 1.0，上述指定 14 个站点距离的平均值为 12.6125。综合 ABC 等级和三个站点距离表来看，选取经纬度范围应为选取三，该选取满足 C 等级，且对于限制条件首条距离不大于 20KM，之后每跳距离不大于 10KM，都体现出较好的适应性，对于实际来说也较为合理。

（二）基于 Relay 无线回传方案最优站点模型

1、模型的约束条件与目标函数：在以上假设成立情况下，根据实际情况生成的一千个站点，以题目给出的要求为约束条件，分别以总成本 f(a)和回传损耗 PL 为目标函数。用公式表达如下：

①约束条件：

$$\mathrm{Fd} \le 20，\ \mathrm{Nd} \le 10$$

$$\mathrm{Num1} \le 4，\ \mathrm{Num2} \le 2$$

$$\mathrm{Sec} \le 2，\ Max \le 6$$

②目标函数：

$$f(a) = 10a_1 + 5a_2 + 50a_3$$

$$\mathrm{PL} = \sum_{i=1,j=1}^{n} (32.5 + 20\lg(D_{ij}) + 20\lg(900))$$

2、模型的建立

（1）为满足成本最低的要求，我们先规定生成站点中有 1000 个子站，再假设所有宿主站为蝴蝶型站，先不设定分区大小，暂定分区数为 84（1000/12），因为这样能保证相互距离最小的点分在一簇里，然后进行聚类，对聚类的结果进行分析，遍历簇，如果簇满足约束条件，将该簇加入到结果集中；否则，则以该簇为聚类对象再聚类，再分析聚类结果（如，如果簇内站点数量超过 12 个，则构建模型（分区数 N = 站点数量/12（整除） + 1）,类似于神经网络里的训练思想；这样，就能保证所有的簇都能满足约束条件，这样在确定子站个数的前提下，子站成本固定，然后尽量减少宿主站和卫星的个数以达到减小总成本的目的。

（2）为减小回传路径损耗，由于宿主站直接采用微波连接，只要模型能尽可能满足宿主站之间距离满足约束条件，就不考虑它们之间的损耗。那么损耗就只在宿主站与子站及子站与子站间。而回传损耗的唯一决策变量是距离，因此，模型尽可能选取宿主站与子站及子站与子站的距离小的进行连接就能减小回传路径损耗。

3、模型的求解

（1）理论依据：

K 均值聚类算法—K 均值聚类算法是先随机选取 K 个对象作为初始的聚类中心。然后计算每个对象与各个种子聚类中心之间的距离，把每个对象分配给距离它最近的聚类中心。聚类中心以及分配给它们的对象就代表一个聚类。一旦全部对象都被分配了，每个聚类的聚类中心会根据聚类中现有的对象被重新计算。这个过程将不断重复直到满足某个终止条件。终止条件可以是没有（或最小数目）对象被重新分配给不同的聚类，没有（或最小数目）聚类中心再发生变化，误差平方和局部最小。

（2）算法思想

算法的核心是 K 均值聚类和图的拓扑算法,K 均值聚类主要用在分站点为以宿主站为中心包含合适数量的子站的若干个簇，拓扑算法主要用于连接每个簇内的子站和宿主站。

考虑到成本，采用这样的思路聚类：固定每个分区大小为 12，后来输出数据和打印散点图时发现这样不满足首跳距离小于 20Km 等一系列约束条件；最后转而采用这样的思路进行聚类：先不设定分区大小，暂定分区数为 84（1000/12），因为这样能保证相互距离最小的点分在一簇里，然后进行聚类，对聚类的结果进行分析，遍历蔟，如果簇满足约束条件，将该簇加入到结果集中；否则，则以该簇为聚类对象再聚类，再分析聚类结果（如，如果簇内站点数量超过 12 个，则构建模型分区数 N = 站点数量/12（整除） + 1）,类似于神经网络里的训练思想；这样，就能保证所有的簇都能满足约束条件，但是分区数不止 84,因为在初始的 84 个簇里，不满足条件的已经进行裂变。

然后是图的拓扑算法，对结果集中的簇进行遍历，对每一个簇，先

遍历每个子站并得到其与宿主站之间的距离，取距离最小的四个子站作为首跳，然后遍历剩下的子站，对于每一个剩下的子站，得到其与首跳的四个子站的距离，并与距离最小的那个首跳子站建立连接（采用矩阵无向图表示图），以此类推，构图完成。

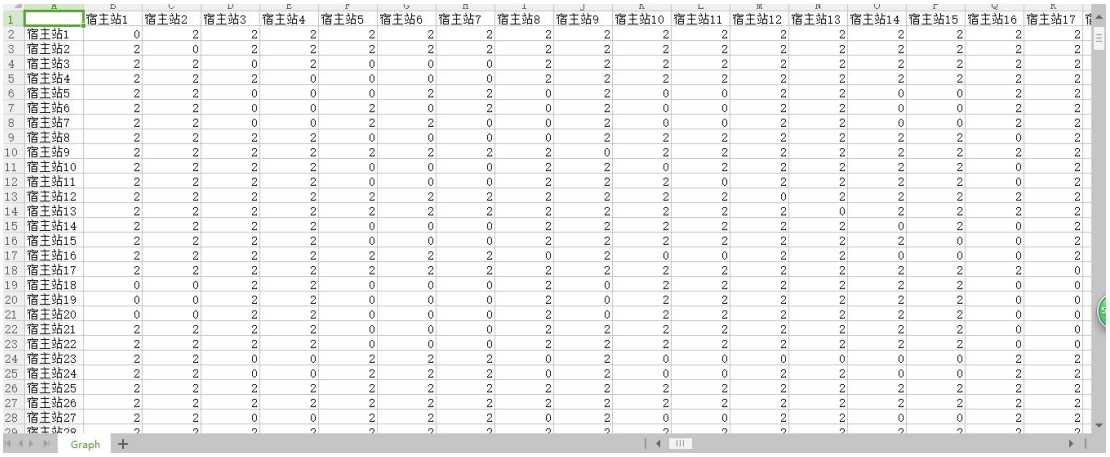
（3）结果展示

采用 java 语言编写算法并运行，得到如下结果：

总体成本:7020.0
平均成本:6.25668449197861
回传路径总体损耗:115249.16443318884
回传路径平均损耗 94.46652822392528
程序运行时间： 728ms
（截图如下：）

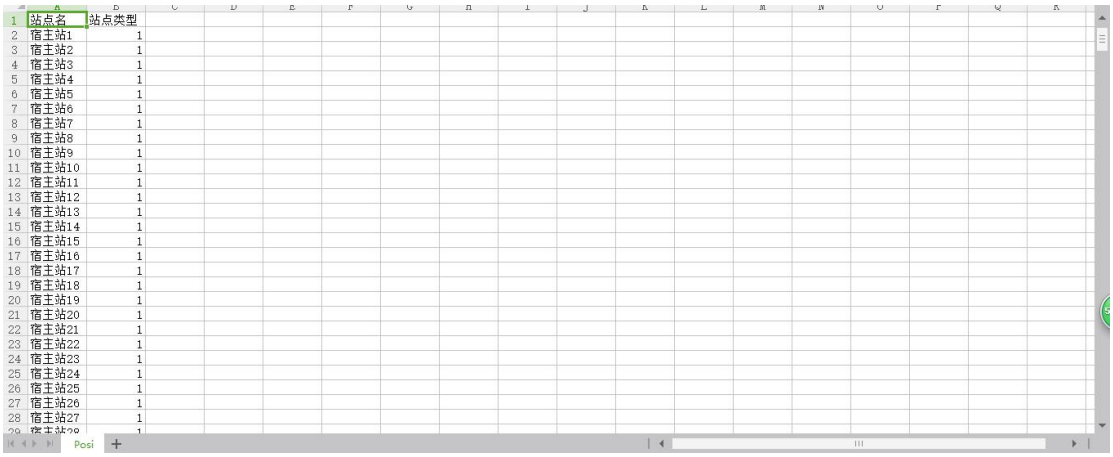
在此次运行中，生成了 1000 个子站和 122 个宿主站，并建立了连接关系，Graph.csv 文件中输出了连接关系：0 表示没有连接关系，1 表示采用无线回传连接（子站与子站，宿主站与子站），2 表示采用微波连接（宿主站与宿主站），Posi.csv 文件中输出了站点类型，0 表示子站，1 表示宿主站。总体成本:7020.0（W USD），平均成本:6.25668449197861(W USD)，均符合模型的预期结果；回传路径总体损耗:115249.16443318884，回传路径平均损耗 94.46652822392528，符合模型预期的结果；而程序运行时间仅为 728ms ，不仅在题目要求的五分钟内，而且可以说是非常迅速的。

（截图如下：）



	宿主站1	宿主站2	宿主站3	宿主站4	宿主站5	宿主站6	宿主站7	宿主站8	宿主站9	宿主站10	宿主站11	宿主站12	宿主站13	宿主站14	宿主站15	宿主站16	宿主站17
宿主站1	0	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
宿主站2	2	0	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
宿主站3	2	2	0	2	0	0	0	2	2	2	2	2	2	2	2	2	2
宿主站4	2	2	2	0	0	0	0	2	2	2	2	2	2	2	2	2	2
宿主站5	2	2	0	0	0	2	2	0	2	0	0	2	2	0	0	2	2
宿主站6	2	2	0	0	2	0	2	0	2	0	0	2	2	0	0	2	2
宿主站7	2	2	0	0	2	2	0	0	2	0	0	2	2	0	0	2	2
宿主站8	2	2	2	2	0	0	0	0	2	2	2	2	2	2	2	0	2
宿主站9	2	2	2	2	2	2	2	2	0	2	2	2	2	2	2	2	2
宿主站10	2	2	2	2	0	0	0	2	2	0	2	2	2	2	2	0	2
宿主站11	2	2	2	2	0	0	0	2	2	2	0	2	2	2	2	0	2
宿主站12	2	2	2	2	2	2	2	2	2	2	2	0	2	2	2	2	2
宿主站13	2	2	2	2	2	2	2	2	2	2	2	2	0	2	2	2	2
宿主站14	2	2	2	2	0	0	0	2	2	2	2	2	2	0	2	0	2
宿主站15	2	2	2	2	0	0	0	2	2	2	2	2	2	2	0	0	2
宿主站16	2	2	2	2	2	2	2	0	2	0	0	2	2	0	0	0	2
宿主站17	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	0
宿主站18	0	0	2	2	0	0	0	2	0	2	2	2	2	2	2	0	0
宿主站19	0	0	2	2	0	0	0	2	0	2	2	2	2	2	2	0	0
宿主站20	0	0	2	2	0	0	0	2	0	2	2	2	2	2	2	0	0
宿主站21	2	2	2	2	0	0	0	2	2	2	2	2	2	2	2	0	0
宿主站22	2	2	2	2	0	0	0	2	2	2	2	2	2	2	2	0	0
宿主站23	2	2	0	0	2	2	2	0	2	0	0	0	0	0	0	2	2
宿主站24	2	2	0	0	2	2	2	0	2	0	0	2	2	0	0	2	2
宿主站25	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
宿主站26	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
宿主站27	2	2	0	0	2	2	2	0	2	0	0	2	2	0	0	2	2

（图 1：Graph.csv 文件截图）



站点名	站点类型
宿主站1	1
宿主站2	1
宿主站3	1
宿主站4	1
宿主站5	1
宿主站6	1
宿主站7	1
宿主站8	1
宿主站9	1
宿主站10	1
宿主站11	1
宿主站12	1
宿主站13	1
宿主站14	1
宿主站15	1
宿主站16	1
宿主站17	1
宿主站18	1
宿主站19	1
宿主站20	1
宿主站21	1
宿主站22	1
宿主站23	1
宿主站24	1
宿主站25	1
宿主站26	1
宿主站27	1

（图 2：Posi.csv 文件截图）

六、模型的评价与改进

（一）模型的评价

无线回传拓扑规划问题是现网中的一个实际课题，本文通过建立最优站点模型，在量化分析的基础上得到了较为合理的，具有实际意义的部站方案。在解决问题的过程中克服了很多困难，模型有优点也有不足。

1、优点：

（1）本文建立的模型对于站点经纬度范围选取具有合理性高的特点，有实际意义；

（2）在实施具体的部站方案时，采用最优站点模型，成功运用 K 均值聚类算法，得到结果；

（3）本文还考虑总体成本及路径损耗方面，并以此为目标函数对站点模型进行优化，得出有实用价值的成本最优的部站方案；

（4）算法收敛速度极快（每次运行时间约 1 秒钟）。

2、缺点：

（1）没有考虑站点的覆盖面积；

（2）对约束条件的考虑还不够全面，还可以再进一步进行优化；

（3）基于站点为随机分布，对实际部署时大密度用户使用情况考虑不够周到；

（4）生成极少数站点重复。

（二）模型的改进

1、将站点的覆盖面积作为约束条件之一，并考虑用户密度及用户分布，更加严谨的建立模型；

2、 对于不同的地形因素，相邻基站间的干扰等影响，使得回传损耗在不同路径上是不同的，需要对具体的情况做具体的路径损耗分析。

七、模型的推广和应用建议

基于K—均值聚类算法的最优站点分布模型是为解决无线relay回传拓扑规划而提出的解决方案，适用于城区、农网等场景下的传统传输方式不可达的问题，同时在部分场景下也可以替代微波，有效降低站高，节省加站费用。该模型基于K—均值聚类算法，与实际紧密联系，故具有推广意义，在实际应用时，该模型主要适用于人口分布较均匀且密度不宜过大的地区，例如中国农村地区，但在中心城区这样的高用户密度地区，该模型并不很适宜。在应用时可将宿主站安装于地区中心地带，以此为中心将子站发散开来，以期接入用户利用率最大化。

八、参考文献

[1] 张然，温向明，路兆明.基于混合遗传算法的无线回传网络部署[D].北京邮电大学
[2] Reinhard Diestel.图论[M].于青林译.高等教育出版社
[3] 黄静静,王爱文.数学建模方法与CUMCM赛题详解[M].机械工业出版社
[4] 李海燕.数学建模竞赛优秀论文选评[C].科学出版社
[5] 郝琳,陈春梅,房茂燕,吴聪伟.全国大学生数学建模竞赛指南[M].西安电子科技大学出版社

九、源代码附录

1、main 函数：

```
package 程序入口;
import java.nio.charset.Charset;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.HashMap;
import com.csvreader.CsvWriter;
import 工具.Util;
import 数据.DataProducer;
import 模型.Station;
import 算法.Kmeans;
import 评价.Judge;

public class Main {

    public static void main(String[] args) throws Exception {
        long startTime = System.currentTimeMillis(); // 获取开始时间
        /*System.out.println("请选择数据输入方式:\n1.计算机模拟产生符
```

```
合条件的数据\n"
                + "2.从文件 dataresouce.");*/
        // 初始化一个 Kmean 对象，将 k 置为 10
        Kmeans k = new Kmeans(84);
        ArrayList<Double> D = new ArrayList<Double>();
        DataProducer dp = new DataProducer();
        // 设置原始数据集
        k.setDataSet(dp.locDataSet);
        k.execute();
        ArrayList<Station> curCenter = new ArrayList<Station>();// 当前
中心链表
        ArrayList<ArrayList<Station>>      curCluster      =      new
ArrayList<ArrayList<Station>>();
        // 得到聚类结果
        ArrayList<ArrayList<Station>> cluster = k.getCluster();
        for (int i = 0; i < cluster.size(); i++) {
            //System.out.println("i=====" + i);
            if (cluster.get(i).size() <= 12) {
                curCluster.add(cluster.get(i));
                curCenter.add(k.getCenter().get(i));
            } else {
                // 这个表达式基于统计平均
                int num = (cluster.get(i).size()) / 12 + 1;
                Kmeans k2 = new Kmeans(num);
                k2.setDataSet(cluster.get(i));
                k2.execute();
                for (int u = 0; u < num; u++) {
                    curCluster.add(k2.getCluster().get(u));
                    curCenter.add(k2.getCenter().get(u));
                }

            }
        }
        // 输出每个簇中最大距离
        for (int i = 0; i < curCluster.size(); i++) {
            double     max     =     Util.getDistance(curCenter.get(i),
curCluster.get(i).get(0));
            ;
            for (int j = 1; j < curCluster.get(i).size(); j++) {
                double curDistance = Util.getDistance(curCenter.get(i),
curCluster.get(i).get(j));
                if (max <= curDistance)
                    max = curDistance;
            }
```

```
            //System.out.println("第" + (i + 1) + "个簇最大距离为" + max);
        }
        // 聚类结果(子站)保存到文件
        StringBuffer sb = new StringBuffer();
        for (int i = 0; i < curCluster.size(); i++) {
            for (int j = 0; j < curCluster.get(i).size(); j++) {
                sb.append(curCluster.get(i).get(j).Longtitude + "," +
curCluster.get(i).get(j).Lattitude + "\r\n");
            }
        }
        Util.writefile(sb.toString(), "zizhan.txt");

        // 聚类结果(宿主站)保存到文件
        sb.delete(0, sb.length());
        for (int i = 0; i < curCenter.size(); i++) {
            sb.append(curCenter.get(i).Longtitude + "," +
curCenter.get(i).Lattitude + "\r\n");
        }
        Util.writefile(sb.toString(), "suzhu.txt");
        // 建立连接,
        int size = 1000 + curCenter.size();
        int[][] connect = new int[size][size];
        for (int i = 0; i < size; i++) {
            for (int j = 0; j < size; j++) {
                connect[i][j] = 0;
            }
        }
        sb.delete(0, sb.length());
        for (int i = 0; i < curCenter.size(); i++) {
            for (int j = i + 1; j < curCenter.size(); j++) {
                double distance = Util.getDistance(curCenter.get(i),
curCenter.get(j));
                if (distance <= 50) {
                    connect[i][j] = 2;
                    connect[j][i] = 2;
                } else {
                    sb.append("From " + i + " To " + j + "distance===" +
distance + "\r\n");
                }
            }
        }
        Util.writefile(sb.toString(), "test.txt");

        for (int i = 0; i < curCenter.size(); i++) {
```

```
            int clustersize = curCluster.get(i).size();

            int count = curCenter.size();
            for (int j = 0; j < i; j++)
                count += curCluster.get(j).size();//

            if (clustersize <= 8) {

                for (int s = 0; s < clustersize; s++) {
                    D.add(Util.getDistance(curCenter.get(i),
curCluster.get(i).get(s)));
                    connect[i][count + s] = 1;
                    connect[count + s][i] = 1;
                }
            } else {
                // keymap
                HashMap<Double,  Integer>  hm  =  new  HashMap<Double,
Integer>();
                double[] distances = new double[clustersize];
                for (int j = 0; j < clustersize; j++) {

                    distances[j]  =  Util.getDistance(curCenter.get(i),
curCluster.get(i).get(j));

                    hm.put(distances[j], j);
                }
                Arrays.sort(distances);
                for (int s = 0; s < 8; s++) {
                    D.add(Util.getDistance(curCenter.get(i),
curCluster.get(i).get(hm.get(distances[s]))));
                    connect[i][count + hm.get(distances[s])] = 1;
                    connect[count + hm.get(distances[s])][i] = 1;
                }

                for (int q = 8; q < clustersize; q++) {
                    double min = 100000000;
                    int minindex = 0;
                    for (int p = 0; p < 8; p++) {
                        double                distance                =
Util.getDistance(curCluster.get(i).get(q), curCluster.get(i).get(p));
                        if (distance < min) {
                            min = distance;
                            minindex = p;
                        }
```

```
                }

                for (int s = 8; s < clustersize; s++) {
                    connect[count + minindex][count + s] = 1;
                    connect[count + s][count + minindex] = 1;

    D.add(Util.getDistance(curCluster.get(i).get(minindex),
curCluster.get(i).get(s)));
                }
            }
        }
    }
    // 输出 csv 文件
    // String path = "D://result.csv";
    // 定义一个 CSV 路径
    String graphCsvFilePath = "Graph.csv";
    // 创建 CSV 写对象 例如:CsvWriter(文件路径，分隔符，编码格式);
    CsvWriter csvWriter = new CsvWriter(graphCsvFilePath, ',',
Charset.forName("GBK"));
    // 写表头
    ArrayList<String> csvHeaders = new ArrayList<String>();
    csvHeaders.add(" ");
    for (int i = 0; i < size; i++) {
        if (i < curCenter.size()) {
            csvHeaders.add("宿主站" + (i + 1));
        } else
            csvHeaders.add("子站" + (i + 1));
    }
    String[] headers = csvHeaders.toArray(new
String[csvHeaders.size()]);
    csvWriter.writeRecord(headers);
    // 写内容
    for (int i = 0; i < size; i++) {
        ArrayList<String> csvContent = new ArrayList<String>();
        if (i < curCenter.size()) {
            csvContent.add("宿主站" + (i + 1));

        } else {
            csvContent.add("子站" + (i + 1 - curCenter.size()));
        }
        for (int j = 0; j < size; j++) {
            csvContent.add(connect[i][j] + "");
        }
        String[] content = csvContent.toArray(new
```

```
String[csvContent.size()]);
            csvWriter.writeRecord(content);
        }

        csvWriter.close();
        System.out.println("--------Graph.csv 文件已经写入--------");

        //
        String posiCsvFilePath = "Posi.csv";
        // 创建 CSV 写对象 例如:CsvWriter(文件路径，分隔符，编码格式);
        CsvWriter csvWriter1 = new CsvWriter(posiCsvFilePath, ',',
Charset.forName("GBK"));
        // 写表头
        ArrayList<String> csvHeaders1 = new ArrayList<String>();

        csvHeaders1.add("站点名" );
         csvHeaders1.add("站点类型" );
        String[]       headers1       =       csvHeaders1.toArray(new
String[csvHeaders1.size()]);
         csvWriter1.writeRecord(headers1);
         for(int i = 0; i<size; i++) {
            ArrayList<String> content = new ArrayList<String>();
            if(i<curCenter.size()) {
                content.add("宿主站"+(i+1));
                content.add(1+"");
            }
            else {
                content.add("子站"+(i-curCenter.size()+1));
                content.add(0+"");
            }
            String    []    writecontent    =    content.toArray(new
String[content.size()]);
            csvWriter1.writeRecord(writecontent);
         }
         csvWriter1.close();
         System.out.println("--------Posi.csv 文件已经写入 --------");
        System.out.println("    总    体    成    本    :"    +
Judge.ChallengeOne(curCenter.size(), 1000));
        System.out.println("    平    均    成    本    :"    +
Judge.ChallengeOne(curCenter.size(),  1000)  /  (curCenter.size()  +
1000));
        System.out.println(" 回 传 路 径 总 体 损 耗 :" +
Judge.ChallengeTwo(D));
        System.out.println("回传路径平均损耗" + Judge.ChallengeTwo(D) /
```

```
D.size());
        long endTime = System.currentTimeMillis(); // 获取结束时间
        System.out.println("程序运行时间： " + (endTime - startTime) +
"ms");
    }
}
```

2、Util 包：

```
package 工具;
import java.io.BufferedWriter;
import java.io.FileOutputStream;
import java.io.OutputStreamWriter;
import 模型.Station;
public class Util {
    private static double EARTH_RADIUS = 6378.137;

    private static double rad(double d) {
        return d * Math.PI / 180.0;
    }
    // longitude 经度 ;“纬度”Latitude
    //求距离工具函数,传入两个Station类参数，返回距离(Km)
    public static double getDistance(Station start,Station end) {
        double radLng1 = rad(start.Longtitude);
        double radLng2 = rad(end.Longtitude);
        double deltaRadLng = radLng1 - radLng2;
        double radLat1 = rad(start.Lattitude);
        double radLat2 = rad(end.Lattitude);
        double deltaRadLat = radLat1 - radLat2;
        double             s             =             (2             *
Math.asin(Math.sqrt(Math.pow(Math.sin(deltaRadLat / 2), 2)
                +     Math.cos(radLat1)     *     Math.cos(radLat2)     *
Math.pow(Math.sin(deltaRadLng / 2), 2))));
        s = s * EARTH_RADIUS;
        s = Math.round(s * 10000d) / 10000d;
        return s;
    }
    //写 txt 文件工具类函数
    public static void writefile(String content, String filename) throws
Exception {
        FileOutputStream fos = new FileOutputStream(filename);
        OutputStreamWriter osw = new OutputStreamWriter(fos, "UTF-8");
        BufferedWriter bw = new BufferedWriter(osw);
        bw.write(content);
        bw.close();
```

```
        osw.close();
        fos.close();
    }
}
```

3、站点包：

```
package 模型;
public class Station {
    //Longitude 经度
    public double Longtitude;
    //Latitude 纬度
    public double Lattitude;
     //构造函数
    public Station(double Longtitude,double Lattitude) {
        this.Longtitude = Longtitude;
        this.Lattitude = Lattitude;
    }
}
```

4、生成站点算法：

```
    package 数据;
    import java.util.ArrayList;
    import java.util.Random;
    import 模型.Station;
    public class DataProducer {
        public ArrayList<Station> locDataSet;
        //模拟产生处于 112.98~113.98;28.12~28.24 之间的 location
        public DataProducer() {
            Random random = new Random();
            locDataSet = new ArrayList<Station>();
            for(int i=0; i<1000; i++) {
                double x = random.nextDouble()+112.98;// 经 度
112.98~113.98
                double y = random.nextDouble()*0.3+28.12;// 纬 度
28.12~28.24
                Station item = new Station(x, y);
    /*          System.out.println(" 经 度 :"+item.Longtitude+"     纬
度:"+item.Lattitude);
    */          locDataSet.add(item);
            }
        }
    }
```

4、Kmeans 算法：

```
    package 算法;
    import java.util.ArrayList;
    import java.util.Random;
```

```
import 工具.Util;
import 模型.Station;
public class Kmeans {
    private int k;// 分成多少簇
    private int m;// 迭代次数
    private int dataSetLength;// 数据集元素个数，即数据集的长度
    private ArrayList<Station> dataSet;// 数据集链表
    private ArrayList<Station> center;// 中心链表
    private ArrayList<ArrayList<Station>> cluster; // 簇
    private  ArrayList<Float>  jc;//  误 差 平 方 和 ， k 越 接 近
dataSetLength，误差越小
    private Random random;

    /**
     * 设置需分组的原始数据集
     *
     * @param dataSet
     */
    public void setDataSet(ArrayList<Station> dataSet) {
        this.dataSet = dataSet;
    }

    /**
     * 获取结果分组
     *
     * @return 结果集
     */

    public ArrayList<ArrayList<Station>> getCluster() {
        return cluster;
    }

    /**
     * 构造函数，传入需要分成的簇数量
     *
     * @param k
     *            簇数量,若 k<=0 时，设置为 1，若 k 大于数据源的长度
时，置为数据源的长度
     */
    public Kmeans(int k) {
        if (k <= 0) {
            k = 1;
        }
        this.k = k;
```

```
    }

    /**
     * 初始化
     */
    private void init() {
        m = 0;
        random = new Random();
        if (dataSet == null || dataSet.size() == 0) {
            initDataSet();
        }
        dataSetLength = dataSet.size();
        if (k > dataSetLength) {
            k = dataSetLength;
        }
        center = initCenters();
        cluster = initCluster();
        jc = new ArrayList<Float>();
    }
    /**
     * get center
     */
    public ArrayList<Station> getCenter() {
        return this.center;
    }
    /**
     * 如果调用者未初始化数据集，则采用内部测试数据集
     */
    private void initDataSet() {
        dataSet = new ArrayList<Station>();

        Station[] dataSetArray = new Station[] { new
Station(113.90652663861538, 28.15692405281289),
                new Station(113.90652663861538, 28.15692405281289)};

        for (int i = 0; i < dataSetArray.length; i++) {
            dataSet.add(dataSetArray[i]);
        }
    }

    /**
     * 初始化中心数据链表，分成多少簇就有多少个中心点
     *
     * @return 中心点集
```

```
     */
    //生成的中心到后面肯定不属于簇了
    private ArrayList<Station> initCenters() {
        ArrayList<Station> center = new ArrayList<Station>();
        int[] randoms = new int[k];
        boolean flag;
        int temp = random.nextInt(dataSetLength);
        randoms[0] = temp;
        for (int i = 1; i < k; i++) {
            flag = true;
            while (flag) {
                temp = random.nextInt(dataSetLength);
                int j = 0;
                while (j < i) {
                    if (temp == randoms[j]) {
                        break;
                    }
                    j++;
                }
                if (j == i) {
                    flag = false;
                }
            }
            randoms[i] = temp;
        }
        for (int i = 0; i < k; i++) {
            center.add(dataSet.get(randoms[i]));// 生成初始化中心链
表
        }
        return center;
    }

    /**
     * 初始化簇集合
     *
     * @return 一个分为k簇的空数据的簇集合
     */
    private ArrayList<ArrayList<Station>> initCluster() {
        ArrayList<ArrayList<Station>>      cluster      =      new
ArrayList<ArrayList<Station>>();
        for (int i = 0; i < k; i++) {
            cluster.add(new ArrayList<Station>());
        }
```

```
        return cluster;
    }

    /**
     * 计算两个点之间的距离
     *
     * @param element
     *            点1
     * @param center
     *            点2
     * @return 距离
     */
    private double distance(Station start, Station end) {
        return Util.getDistance(start, end);
    }

    /**
     * 获取距离集合中最小距离的位置
     *
     * @param distance
     *            距离数组
     * @return 最小距离在距离数组中的位置
     */
    private int minDistance(double[] distance) {
        double minDistance = 10000000000000d;//distance[0];
        int minLocation = -1;//0;

        for (int i = 0; i < distance.length; i++) {

            if (distance[i] < minDistance) {
                minDistance = distance[i];
                minLocation = i;
            } else if (distance[i] == minDistance) // 如果相等，随机
返回一个位置
            {
                if (random.nextInt(10) < 5) {
                    minLocation = i;
                }
            }
        }

        return minLocation;
    }
```

```
        /**
         * 核心，将当前元素放到最小距离中心相关的簇中
         */
        private void clusterSet() {
            double[] distance = new double[k];
            for (int i = 0; i < dataSetLength; i++) {
                for (int j = 0; j < k; j++) {
                    distance[j]        =        distance(dataSet.get(i),
center.get(j));
                    //
System.out.println("test2:"+"dataSet["+i+"],center["+j+"],distance="+
distance[j]);

                }
                //ArrayList<Integer> ignore = new ArrayList<Integer>();
                int minLocation = minDistance(distance);//,ignore);

                cluster.get(minLocation).add(dataSet.get(i));// 核心,将
当前元素放到最小距离中心相关的簇中

            }
        }

        /**
         * 求两点误差平方的方法
         *
         * @param element
         *            点1
         * @param center
         *            点2
         * @return 误差平方
         */
        private double errorSquare(Station element, Station center) {
            double x = element.Longtitude - center.Longtitude;
            double y = element.Lattitude - center.Lattitude;

            double errSquare = x * x + y * y;

            return errSquare;
        }

        /**
         * 计算误差平方和准则函数方法
         */
```

```
        private void countRule() {
            float jcF = 0;
            for (int i = 0; i < cluster.size(); i++) {
                for (int j = 0; j < cluster.get(i).size(); j++) {
                    jcF      +=      errorSquare(cluster.get(i).get(j),
center.get(i));
                }
            }
            jc.add(jcF);
        }

        /**
         * 设置新的簇中心方法
         */
        private void setNewCenter() {
            for (int i = 0; i < k; i++) {
                int n = cluster.get(i).size();
                if (n != 0) {
                    Station newCenter = new Station(0, 0);
                    for (int j = 0; j < n; j++) {
                        newCenter.Longtitude                              +=
cluster.get(i).get(j).Longtitude;
                        newCenter.Lattitude                               +=
cluster.get(i).get(j).Lattitude;
                    }
                    // 设置一个平均值
                    newCenter.Longtitude = newCenter.Longtitude / n;
                    newCenter.Lattitude = newCenter.Lattitude / n;
                    center.set(i, newCenter);
                }
            }
        }

        /**
         * 打印数据，测试用
         *
         * @param dataArray
         *            数据集
         * @param dataArrayName
         *            数据集名称
         */
        public void printDataArray(ArrayList<Station> dataArray, String
dataArrayName) {
            for (int i = 0; i < dataArray.size(); i++) {
```

```
                System.out.println("print:" + dataArrayName + "[" + i +
"]={" + dataArray.get(i).Longtitude + ","
                        + dataArray.get(i).Lattitude + "}");
            }
            System.out.println("==================================");
        }

        /**
         * Kmeans 算法核心过程方法
         */
        private void kmeans() {
            init();
            while (true) {
                clusterSet();
                countRule();
                // 误差不变了，分组完成

                if (m != 0) {
                    if (jc.get(m) - jc.get(m - 1) ==0) {
                        break;
                    }
                }
                setNewCenter();
                m++;
                cluster.clear();
                cluster = initCluster();
            }
        }

        /**
         * 执行算法
         */
        public void execute() {
            kmeans();
        }
    }
```