**Project2实验报告**

张毅恒 2016080104006

**1 实验题目**

* 代码实现Prim实现#4（基于堆）
* 代码实现Kruskal实现#2（基于UNION-FIND）
* 设计实验，针对多组相同实例，比较真实运行时间。

**2 实验思路与设计设计**

**2.1 Prim算法实现#4**

2.1.1 代码结构

由于对于堆的操作种类多，并且每种操作的步骤并不是十分复杂，另外设计调用函数会导致程序复杂且没有必要，所以不单独设计堆操作的函数，整个函数只分为两个部分：

主体部分：负责读取文件，调用最小生成树函数

最小生成树函数：中间不调用函数，输入参数为图以及起始点，函数最后打印出边权之和以及树内的边的构成。

2.1.2 存储方式

图：以邻接链表方式存储，体现在数据结构上为字典套字典，即{point1：{point2：value，point3：value，…}，point2…}，其中value指的是边的权重

堆：使用list存储点的标号，其中list[0]取值为-1，即点从list[1]开始存，在算法中的d(v)为了方便查询，使用dict存储，同时，使用一个dict存储并实时同步点在堆数组的位置，以方便修改点的d值时及时获取点的位置，而不是逐个遍历以至于增加不必要的复杂度，与此同时，维护一个堆的长度变量length

X（已覆盖的点集）、V（所有点的点集）、T（生成树）均使用set存储

P(v)（当前节点的父节点）也使用dict存储。

2.1.3 代码设计

由于本实现方法参考的是实现#3，所以只要区别在于与实现#3的区别，主要在于：

在循环最开始提前建好堆结构；

点进入X之后将其在堆内删除，再将堆的最后一个元素补上，堆的长度自减一，随后维护堆的最小性质；

点的d值变化后在dict内更新，然后根据维护的位置的dict直接找到点的位置，从这个位置开始向上维护堆的最小性质

**2.2 代码实现Kruskal实现#2（基于UNION-FIND）**

2.2.1 代码结构

在代码结构上，由于相同的原因，结构与上一个设计一样，不同的是，最小生成树的读入是三个量：图、点数、边数。

2.2.2 存储结构

图：为了方便读取边的邻接点以参与比较，图的存储方式为字典内的list，即{edge0:[point1,point2,weight]}，其中weight为边的权重值；

Union：union的存储分为几个部分，第一个是leader和size的集合，另一个是一个将点分成集合的list内的list。list内的list将点分为几个集合，当集合合并时，将元素少的那个集合置空（但是不删除），将里面的元素加入元素多的那个集合，leader和size的集合维护每一个点的两个参数：leader和size，leader指的是这个点当前所在的list位于外面的大list的位置（从0开始），作用是与其他点比较判断是否成环。Size指的是这个点当前所在集合的元素的个数，用于合并时判断。

2.2.3 代码设计

生成树的打印输出与上一个程序一样。

排序可采用快排或者归并排序，由于本project的主要内容不是排序，所以选择直接调用sort进行排序，具体方法是将边的编号及其权重值合成一个tuple，然后将所有的tuple合成一个list，将list按照每一个元的第二项从小到大排序。

**3 实验结果**

代码的运行结果、时间结果以及复杂度如下：

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Prim | | Kruskal | |
| 复杂度 | O(mlogn) | | O(mlogn) | |
|  | 执行时间（秒） | 权重和 | 执行时间（秒） | 权重和 |
| Graph6 | 0.004987 | 340 | 0.03989 | 340 |
| Graph7 | 0.010971 | 198 | 0.009973 | 198 |
| Graph8 | 0.012964 | 163 | 0.018949 | 163 |
| Graph9 | 0.017948 | 704 | 0.015956 | 704 |
| Graph10 | 0.034907 | 404 | 0.033912 | 404 |
| Graph11 | 0.028922 | 315 | 0.091751 | 315 |
| Graph12 | 1.191162 | 3886 | 2.260565 | 3886 |

在时间数量级上，两个算法执行时间是近似的，最后一个执行时间区别较大，分析原因可能是第二个算法相比于第一个算法，复杂度更接近O(mn)，虽然已经进行size的优化