**Project3实验报告**

张毅恒 2016080104006

**1 实验题目**

* 代码实现Dijkstra #3（基于堆）
* 代码实现Dijkstra #4（基于循环桶）
* 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 代码实现Dijkstra #3（基于堆）**

由于从算法角度，迪杰斯特拉与prim算法差别不大，且这个project是用堆来实现，所以只需要在project2的基础上稍作修改。（区别部分加粗斜体下划线）

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，***project2输入为图以及起始点，project3由于最理想的是求出所有点到其他所有点的情况，所以输入参数只有图***；***project2函数最后打印出边权之和以及树内的边的构成，project3对于每一个起点，打印从这个点到达所有终点的路径以及长度，并且将所有权重求和以检验，如果没有路径，那么权重为65535，路径为Nopath。***

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。

***路径（path）：使用字典存储，每一个点的路径使用list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

***由于输出的结果可能会过长，所以还使用了一个文件resultHeap.txt单独存储输出结果。***

**2.2 代码实现Dijkstra #4（基于循环桶）**

2.2.2 代码结构

从算法角度，这个实验和上一个实验差别不大，所以代码结构基本一样，只不过把堆换成了循环桶，不过为了建桶的方便，函数的输入除了图，还有一个边的最大值。

2.2.2 存储方式

桶：桶列表使用list存储，这样每一个桶都有一个标号，列表内的每一个桶使用set来存储；

图：字典套字典；

父节点（parent）：字典；

距离：字典，同时距离标号mod桶的数量就是该点所在的桶的编号；

路径：字典套list。

2.2.3 代码设计

最小值：在第一个非空的list内的set随便找一个并且删除

变化值： 变distance并且set内删除增加

指向桶的指针循环，存入桶的元素也循环

读入图的时候搞一个最大value出来

函数输入变量包括图和最大边（主要是比较懒不高兴再算最大边）

要考虑点无法到达的情况，如果不考虑，很容易因为V-X这样的判决进入死循环，所以如果所有桶空那么也提前结束循环。

**3 实验结果**

路径和权重值这样的结果比较多，所以不放在这里讨论，直接放在提交的里面或者使用代码跑。

|  |  |  |
| --- | --- | --- |
|  | Heap | Bucket |
|  | 执行时间（秒） | 执行时间（秒） |
| Graph1 | 0.006981 | 0.039990 |
| Graph2 | 0.008975 | 0.002992 |
| Graph3 | 0.003988 | 0.017950 |
| Graph4 | 0.303324 | 0.265291 |

可以看出，在算法复杂度上，虽然两种实现不好比较（一个是O(mlogn)，一个是O(m+nC)），但是整体情况下，使用桶实现要比使用堆快，只不过代价是占用了更多的内存（尽管使用了循环桶）