中山大学计算机学院本科生实验报告

（2025学年第1学期）

课程名称：数据结构与算法　　实验任课教师：张子臻

年级	2024级	**专业（方向）**	计算机科学与技术（人工智能与大数据）
学号	242325157	**姓名**	梁玮麟
电话	18620062182	**Email**	3371676041@qq.com
开始日期	2025.12.17	**结束日期**	2025.12.22

第一题

1、实验题目

⭐ z15-最大生成森林

题目描述

对连通图进行遍历，过程中所经过的边和顶点的组合可看做是一棵普通树，通常称为生成树。而连通图的所有生成树中边权重和最大的那棵生成树即为最大生成树。

对于非连通图而言，多个连通分量对应的多棵生成树就构成了整个非连通图的生成森林。

而最大生成森林的定义就是在非连通图中，所有连通分量对应的最大生成树之和。

因此我们给定一个无向图 G，它可能是一个非连通图的，你的任务是找到最大生成森林。

输入

可能有多个测试用例。第一行是测试用例数目。

对于每个测试用例，第一行包含两个整数 n, m(1 ≤ n ≤ 1000, 0 ≤ m ≤ n×(n−1)/2)，其中 n 是图中的顶点数，m 是边数。

接下来的 m 行，每行有三个整数 x, y, d(0 ≤ x, y ≤ n − 1, d > 0)，这意味着有一个无向边 (x, y)，代价为 d。

输出
对于每个测试用例，输出最大生成森林的值。

输入样例

```
2
3 3
0 1 1
1 2 2
2 0 3
4 2
0 1 1
2 3 2
```

输出样例

```
5
3
```

2、实验目的

- 理解生成树、最大生成树与最大生成森林的概念。
- 掌握并实现最大生成森林的两种经典算法思路：Kruskal（按边选取）与 Prim（按点扩展）。
- 对比两种实现的复杂度与适用场景，能够根据数据特征选择更合适的算法。

3、算法设计

> 本题提供两种实现（Kruskal / Prim）。下面分别给出两份算法设计与复杂度分析，并在最后对比适用场景。

方案一：Kruskal（最大生成森林）

设计思路

- 最大生成森林 = 对每个连通分量求最大生成树权值之和。
- Kruskal 从**边的角度**出发：将所有边按权重从大到小取出，依次尝试加入森林。
- 使用并查集（DSU）判断一条边的两个端点是否已在同一棵树中：
  - 若不在同一集合，则加入该边并合并集合（不会成环）。
  - 若已在同一集合，跳过该边。
- 因为图可能非连通，Kruskal 不需要显式找连通分量：最终会自然形成多棵树（森林）。

流程图

```
Start
  |
  |--> 输入测试用例数 num
  |
  |--> while num--:
  |         输入 n, m
  |         初始化并查集 root[i] = i
  |         建立最大堆 e（按边权从大到小）
  |
  |         读入 m 条边 (x, y, d):
  |               将 (x, y, d) 加入堆 e
  |
  |         weight = 0
```

```
  |
  |        while e 非空:
  |              取出堆顶边 (u, v, w)
  |              ru = find(u), rv = find(v)
  |              if ru != rv:
  |                    weight += w
  |                    合并 root[ru] = rv
  |              else:
  |                    跳过该边
  |
  |        输出 weight
  |
End
```

复杂度分析(Kruskal)

- **建并查集 root**：初始化 `n` 个元素，时间 `O(n)`，空间 `O(n)`。
- **读入并压入优先队列 m 条边**：每次 `push` 代价 `O(log m)`，共 `O(m log m)`；边存储空间 `O(m)`。
- **主循环弹出边并做并查集合并**：
  - `pop` 共 `m` 次，每次 `O(log m)` → `O(m log m)`；
  - `find/union` 使用路径压缩，均摊近似 `O(α(n))`，总 `O(m α(n))`（通常可视为很小常数）。

总时间复杂度：`O(m log m)`
总空间复杂度：`O(n + m)`

细节注意

- kruskal天然不保证结果联通，联通的结果依赖于图本身的联通，很适合写着一道题。
- 并查集可以按秩合并优化。

具体实现

```
//kruskal
//从边的角度出发。因为所有的权重都来自于边，所以只需要遍历所有边，并跳过某些情况即可
#include<iostream>
#include<vector>
#include<queue>
using namespace std;
struct edge{
    int u;
    int v;
    int weight;
    edge()=default;
    edge(int u,int v,int weight):u(u),v(v),weight(weight){}
    bool operator<(const edge& other) const{
        return this->weight<other.weight;
    }
};
int find(vector<int>& root, int u){
    return root[u]==u ? u : root[u]=find(root, root[u]);
}
int main(){
    int num;
    cin>>num;
    while(num--){
        int n,m;
        cin>>n>>m;
        priority_queue<edge,vector<edge>,less<edge>> e;
        //储存每一个点的根节点，用于判断是否同一棵树
        vector<int> root(n);
        for(int i=0;i<n;i++){
            root[i]=i;
        }
        //初始化数据，用edge类储存边，用priority_queue储存edge
        for(int i=0;i<m;i++){
            int x,y,d;
            cin>>x>>y>>d;
            e.push(edge(x,y,d));
        }
```

```
        int weight=0;
        while(!e.empty()){
            edge crt=e.top();
            e.pop();
            int u=crt.u, v=crt.v, ru, rv;
            ru = find(root, u);
            rv = find(root, v);
            if(ru!=rv){
                weight+=crt.weight;
                root[ru]=rv;
            }else{
                continue;
            }
        }
        cout<<weight<<endl;
    }
    return 0;
}
```

方案二：Prim（对每个连通分量求最大生成树）

设计思路

- Prim 从**点的角度**出发：从某个起点开始，不断把“当前树能连到的最大权边”对应的点加入树。
- 因为图可能非连通：
  - 用 `vst[]` 记录是否访问过；
  - 枚举每个点 `i`，若未访问，则以 `i` 为起点跑一次 Prim，得到该连通分量的最大生成树，最终得到不同的联通分量；
  - 将各分量结果累加就是最大生成森林。
- 使用优先队列维护候选边（在代码中用 `(邻点, 边权)` 表示），每次取出“边权最大的候选”。

流程图

```
Start
  |
  |--> 输入测试用例数 num
  |
  |--> while num--:
  |        输入 n, m
  |        建立邻接表 list
  |        初始化 vst[0..n-1] = false
  |
  |        读入 m 条无向边 (x, y, d):
  |              list[x].push_back(y, d)
  |              list[y].push_back(x, d)
  |
  |        weight = 0
  |
  |        for i = 0..n-1:
  |              if vst[i] == false:
  |                    建立最大堆 q
  |                    q.push(i, 0)
  |
  |                    while q 非空:
  |                          取出堆顶 (u, w)
  |                          if vst[u] == true:
  |                                continue
  |                          vst[u] = true
  |                          weight += w
  |                          for 每个 (to, d) ∈ list[u]:
  |                                q.push(to, d)
  |
  |        输出 weight
  |
End
```

明白 👆

那我给你一版**短、干净、但逻辑完整**的复杂度分析，只解释**你这份 Prim 实现本身**，不提教科书、不做额外展开，适合直接放实验报告。

你可以**整段替换**你现在的那一段。

复杂度分析（Prim）

- **建邻接表**：无向边在邻接表中各存一份，共插入 `2m` 次，时间复杂度为 `O(m)`，空间复杂度为 `O(n + m)`。
- **外层枚举顶点**：由于图可能非连通，需要遍历所有顶点以寻找未访问的连通分量起点，时间复杂度为 `O(n)`。
- **Prim 主过程（优先队列操作）**：
  - 每个连通分量会将其起点压入优先队列一次，最坏情况下共 `O(n)` 次；
  - 当顶点首次被访问时，其所有邻接边都会被压入优先队列，邻接表总长度为 `2m`，因此边相关的入队次数为 `O(m)`；
  - 所有入队元素最终都会被弹出一次，故堆相关操作总次数为 `O(m + n)`；
  - 优先队列中元素规模最坏可达 `O(m)`，单次 `push/pop` 的时间复杂度为 `O(log m)`。

总时间复杂度：`O((m + n) log m)`
总空间复杂度：`O(n + m)`

细节注意

- 相对kruskal，prim算法需要在顶层加一个遍历，才能保证得出不同的连通分量。
- 当优先队列中可以访问相同的节点，优先队列会把大的权重的节点优先弹出，然后再次碰到相同节点但是权重更小的情况，会因为已经被访问过直接跳过。
- 标准的prim中存的是顶点，但是我这里存的是<顶点，weight>，且容许冗余项存在。所以严格上界是`O(( m + n ) log m)`，并非`O(( m + n ) log n)`

具体实现

```
//prim
#include<iostream>
#include<vector>
#include<queue>
using namespace std;
```

```
struct cmp{
    bool operator()(const pair<int,int>& a,const pair<int,int>& b){
        return a.second<b.second;
    }
};
int main(){
    int num;
    cin>>num;
    while(num--){
        int n,m;
        cin>>n>>m;
        //邻接表
        vector<vector<pair<int,int>>> list(n);
        //访问记录
        vector<bool> vst(n,false);
        for(int i=0;i<m;i++){
            int x,y,d;
            cin>>x>>y>>d;
            list[x].push_back(make_pair(y,d));
            list[y].push_back(make_pair(x,d));
        }
        int weight=0;
        for(int i=0;i<n;i++){
            if (vst[i]==false){
                //找到一个连通分支，构建最大生成树
                priority_queue<pair<int,int>,vector<pair<int,int>>, cmp> q;
                q.push(make_pair(i,0));
                while(!q.empty()){
                    //先把当前节点取出来
                    pair<int,int> crt = q.top();
                    q.pop();
                    //如果当前节点已经被访问过，需要跳过
                    if(vst[crt.first]==true) continue;
                    //否则
                    vst[crt.first]=true;
                    weight+=crt.second;
                    for(auto it:list[crt.first]){
                        //将所有邻接节点压入队列
                        q.push(it);
```

```
                    }
                }
            }
        }
        cout<<weight<<endl;
    }
}
```

两种实现对比：什么情况下哪一份代码更好（从复杂度角度）

- **总体复杂度对照（本题实现方式）**
  - **Kruskal**：需要对所有边进行全局优先级处理（相当于排序/堆取最大边），时间复杂度为

$$O(m \log m); (+; m\alpha(n))$$

    其中并查集操作均摊近似常数，整体由 `m log m` 主导。
  - **Prim（本实现：边入堆、出堆时用 vst 过滤 + 可能多连通分量）**：
    堆操作次数为 `O(m+n)`（`2m` 次邻接边入堆 + 最多 `n` 次分量起点入堆），堆规模最坏可达 `O(m)`，因此时间复杂度为
    [
    O((m+n)\log m)
    ]
  - **结论**：两者最坏情况下都带有 `log m`，但 **Kruskal 的堆/排序操作是“对所有边一次性全局处理”**；而 **Prim 的堆操作是“遍历过程中不断把邻接边反复入堆”**，在常数与实际堆操作次数上可能更大。
- **稀疏图（m 与 n 同阶，例如 m = O(n)）**
  - Kruskal：`O(m log m)` ≈ `O(n log n)`
  - Prim：`O((m+n) log m)` ≈ `O(n log n)`
  - **结论**：两者渐进复杂度接近；但 Kruskal 的“每条边最多考虑一次是否合并”比较直接，而 Prim 会产生大量“重复入堆但被 vst 跳过”的元素（虽然仍在同阶），实际常数可能更大。因此**稀疏图下两者都可用，但 Kruskal 更稳定**。
- **稠密图（m 接近 n^2）**
  - Kruskal：`O(m log m)`（必须全局处理所有边，规模很大）

- Prim：`O((m+n) log m)` ≈ `O(m log m)`（同阶）
- **结论**：两者同阶；但在稠密图中，Prim 的堆可能长期维持大量候选边，且会产生更多无效 pop（被 `vst` 跳过），因此**稠密图下 Kruskal 往往更“可预期”**（但两者都可能较慢）。

- **非连通程度很高（例如 m 很小甚至 m=0）**
  - Kruskal：堆里几乎没边，主要是读取与少量操作，时间近似 `O(m log m)`（很小）。
  - Prim：仍然需要外层遍历所有顶点，并对每个连通分量执行一次 `q.push(i,0)`，因此会出现明显的 `+n` 项：
    [
    O((m+n)\log m)
    ]
    虽然当 `m=0` 时 `log m` 需要用“堆规模按常数处理”的理解，但从操作次数上看 Prim 仍然要处理 `n` 次启动分量。
  - **结论**：在**高度非连通**场景，Kruskal 几乎只处理边集，代价更小；Prim 必须逐点启动分量，额外开销更明显，因此**Kruskal 更优**。
- **总结选择建议（只从复杂度/操作次数角度）**
  - 若关注“全局一次性处理边集”的稳定复杂度：**优先 Kruskal（`m log m` 主导）**。
  - 若图连通性较强、且希望按分量扩展（但注意本实现会产生较多无效堆元素）：**Prim 也可用，但其代价更贴近 `((m+n) log m)`**。
  - 若图非常不连通或边很少：**Kruskal 通常更划算**。

程序运行与测试

测试样例一

- 标准输入：

```
2
1 0
2 1
0 1 1
```

- 实际输出：

```
0
1
```

- 期望输出：

```
0
1
```

测试样例二

- 标准输入：

```
1
9 7
0 3 52
0 5 70
0 8 28
2 3 49
2 8 50
3 6 1
5 6 54
```

- 实际输出：

```
275
```

- 期望输出：

```
275
```

实验总结与心得

- 通过把“最大生成森林”拆解为“对每个连通分量求最大生成树并求和”，从而能直接套用 MST 的经典算法思想。
- Kruskal 与 Prim 的差异本质是“按边全局选择”与“按点逐步扩展”的差别：
  - Kruskal 更像全局筛选边集；
  - Prim 更像在连通分量内部逐步扩张边界。
- 实现层面，非连通图的处理方式也体现了两者风格：Kruskal 天然得到森林；Prim 需要用访问数组对每个分量重复启动。
- 进一步加深了对并查集（路径压缩）、优先队列（最大堆/自定义比较器）以及复杂度估算（分段分析→总复杂度归纳）的理解。