Prim VS Kruskal

2018211302班 2018210074 熊宇

一、引入

关于集合的一些基本运算可用于实现Kruskal算法。

按权的递增顺序查看等价于对优先队列执行DeleteMin运算。可以用堆实现这个优先队列。

对一个由连通分支组成的集合不断进行修改，需要用到抽象数据类型并查集UnionFind所支持的基本运算。

当图的边数为e时，Kruskal算法所需的计算时间是$O(eloge)$。当$e=\Omega(n^2)$时，Kruskal比Prim算法差，但当$e=O(n^2)$时，Kruskal算法却比Prim算法好得多。（e为边数，n为结点数）

二、最小生成树

一个有 n 个结点的连通图的生成树是原图的极小连通子图，且包含原图中的所有 n 个结点，并且有保持图连通的最少的边。简单来说就是有且仅有n个点n-1条边的连通图。

而最小生成树就是最小权重生成树的简称，即所有边的权值之和最小的生成树。最小生成树问题一般有两种求解方式：Prim算法和Kruskal算法。

三、Prim算法

1.算法内核

Prim算法通常以邻接矩阵作为储存结构。它的基本思想是以顶点为主导地位，从起始顶点出发，通过选择当前可用的最小权值边把顶点加入到生成树当中来，具体步骤为:

1.从连通网络$N=\{V,E\}$中的某一顶点U_0出发，选择与它关联的具有最小权值的边（U_0,V）,将其顶点加入到生成树的顶点集合U中。

2.以后每一步从一个顶点在U中，而另一个顶点不在U中的各条边中选择权值最小的边(U,V),把它的顶点加入到集合U中。如此继续下去，直到网络中的所有顶点都加入到生成树顶点集合U中为止。

2.算法可行性证明

设prim生成的树为G_0，假设存在$Gmin$使得$cost(Gmin)<cost(G_0)$，则在Gmin中存在(u,v)不属于G_0，将(u,v)加入G_0中可得一个环，且(u,v)不是该环的最长边，这与prim每次生成最短边矛盾，故假设不成立，得证。

3.算法实现$O(n^2)$

```
#include <iostream>
#include <cstdio>
#include <algorithm>
#include <cstring>
using namespace std;
const int N = 500+10;
int n,m;
int g[N][N],dis[N],vis[N];

void prim()
{
    memset(dis,0x1f,sizeof dis);
    dis[1]=0;
    for(int j=1;j<=n;j++)
    {
        int min_len=2e+9,k;
        for(int i=1;i<=n;i++)
        {
            if(!vis[i]&&dis[i]<min_len)
            {
                min_len=dis[i];
                k=i;
            }
        }
        vis[k]=1;
        for(int i=1;i<=n;i++)
        {
```

```
            if(!vis[i]&&dis[i]>g[k][i])
                dis[i]=g[k][i];
        }
    }

}

int main()
{
    scanf("%d%d",&n,&m);
    memset(g,0x1f,sizeof g);
    for(int i=1;i<=m;i++)
    {
        int u,v,w;scanf("%d%d%d",&u,&v,&w);
        g[u][v]=g[v][u]=min(g[u][v],w);  //因为有重边，所以取min
    }
    prim();
    int ans=0;
    for(int i=1;i<=n;i++)ans+=dis[i];
    if(ans>1e7)printf("impossible\n");
    else printf("%d\n",ans);
    return 0;
}
```

4.堆优化$O(elogn)$

```
void Prim_heap(int point)
{
    memset(dis,0x1f,sizeof(dis));
    priority_queue<pair<int,int> > q;

    dis[point]=0;
    q.push(make_pair(0,1));
    while(!q.empty())
    {
        int k=q.top().second;
        q.pop();
        v[k]=1;
        for(int i=h[k];i!=-1;i=edge[i].next)
        {
            int to=edge[i].to,w=edge[i].w;
            if(!v[to]&&dis[to]>w)
            {
                dis[to]=w;
                q.push(make_pair(-dis[to],to));  //优先队列大根堆变小
根堆小骚操作：只需一个‘-’号；
            }
        }
    }
    for(int i=1;i<=n;i++)if(dis[i]==0x1f1f1f1f)flag=false;  //判断是
否不存在最小生成树
```

```
    return ;
}
```

四、Kruskal算法

1.算法内核

先构造一个只含 n 个顶点、而边集为空的子图，把子图中各个顶点看成各棵树上的根结点，之后，从网的边集 E 中选取一条权值最小的边，若该条边的两个顶点分属不同的树，则将其加入子图，即把两棵树合成一棵树，反之，若该条边的两个顶点已落在同一棵树上，则不可取，而应该取下一条权值最小的边再试之。依次类推，直到森林中只有一棵树，也即子图中含有 n-1 条边为止。

简单来说就是以边为主导地位，每次选择权值最小的边，判断该边连接的两点是否连通，若不连通，则合并两点（合并操作以并查集实现）。记录合并的次数，当次数等于n-1时结束。具体步骤如下：

1.新建图G，G中拥有原图中相同的节点，但没有边。

2.将原图中所有的边按权值从小到大排序。

3.从权值最小的边开始，如果这条边连接的两个节点于图G中不在同一个连通分量中，则添加这条边到图G中。

4.重复3，直至图G中所有的节点都在同一个连通分量中。

2.算法可行性证明

由Kruskal算法构成的任何生成树$T* = G[\{e_1, e_2, ..., en - 1\}]$都是最小生成树，这里n为赋权图G的顶点数。使用反证法证明。

1.假设存在异于T*的生成树T，他的权值更小。

2.定义函数f(T)表示不在T中的最小权值i的边e_i。假设T*不是最小树，T才是真正的最小树，显然T会使f(T)尽可能大的，即T本身权重则会尽可能小。

3.设f(T)=k,表示存在一个不在T中的最小权值边$e_k = k$，也就是说$e_1, e_2, ..., ek - 1$同时在T和T*中，$e_k = k$不在T中。

4.$T + e_k$包含唯一圈C。设e'_k 是C的一条边，他在T中而不在T*中。（想象圈C中至少有e_k和e'_k ，其中e_k是由Kruskal算法得出的最小权边）。

5.令$T' = W(T) + w(e_i) - w(e'_i)$，Kruskal算法选出的是最小权边$e_k$，（而$e'_k$是T自己根据f(T)选出来的边）有$w(e'_k) >= w(e_k)$ 且$W(T') = W(T*)$（T ' 也是一个最小生成树）。

6.但是$f(T') > k = f(T)$，即T并没有做到使得f(T)尽可能大，他根本不是真正的最小树，所以T>=T*,从而T*确实是一棵最小树。

3.算法实现$O(eloge)$

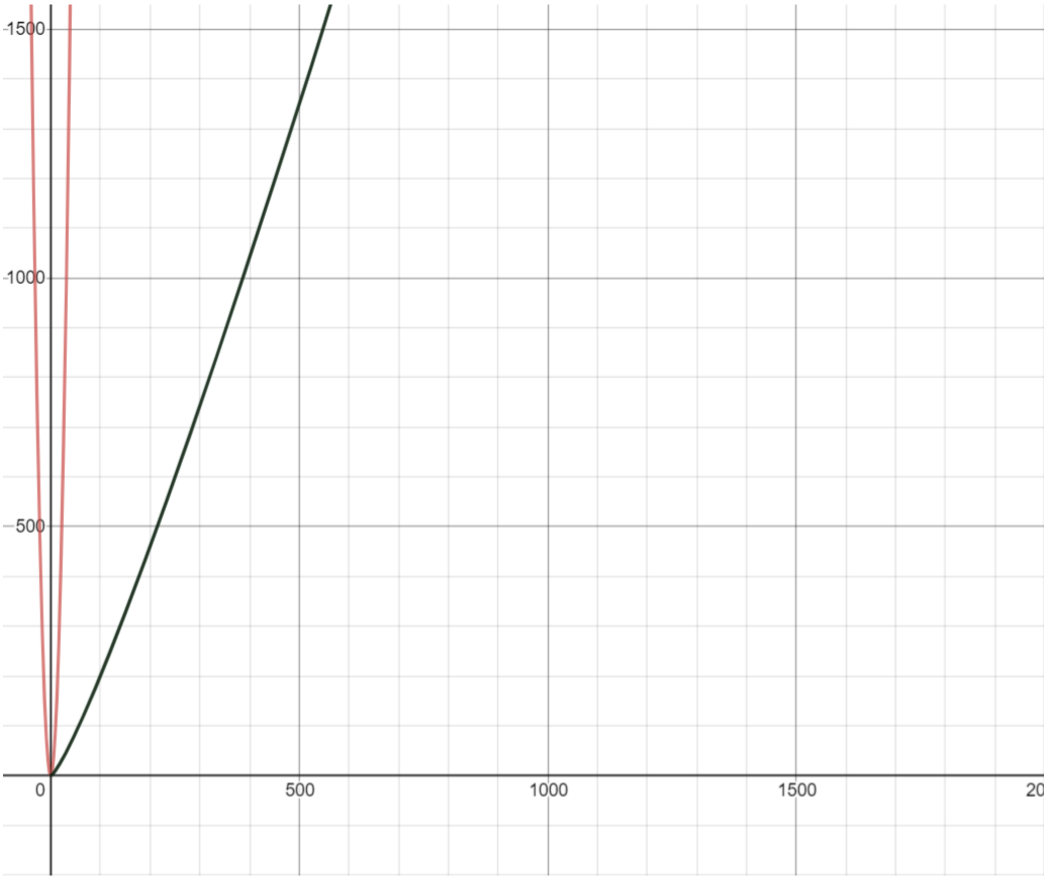
```
#include <iostream>
#include <algorithm>
#include <cstdio>
#include <cstring>
using namespace std;
const int N = 100000+10, M = 200000+10;

struct Edge{
    int u,v,w;
    bool operator < (const Edge &E)const
    {
        return w<E.w;
    }
}edge[M];
int fa[N];
int n,m,cnt,ans;

int find(int x)
{
    if(fa[x]==x)return x;
    else return fa[x]=find(fa[x]);
}

int main()
{
    scanf("%d%d",&n,&m);
    for(int i=1;i<=n;i++)fa[i]=i;
    for(int i=1;i<=m;i++)
    {
        int a,b,c;scanf("%d%d%d",&a,&b,&c);
        edge[i].u=a;edge[i].v=b;edge[i].w=c;
    }
    sort(edge+1,edge+m+1);
    for(int i=1;i<=m;i++)
    {
        int u=find(edge[i].u),v=find(edge[i].v),w=edge[i].w;
        if(u!=v)
        {
            cnt++;
            fa[u]=v;
            ans+=w;
        }
    }
    if(cnt==n-1)printf("%d\n",ans);
    else printf("impossible\n");
    return 0;
}
```

五、复杂度对比

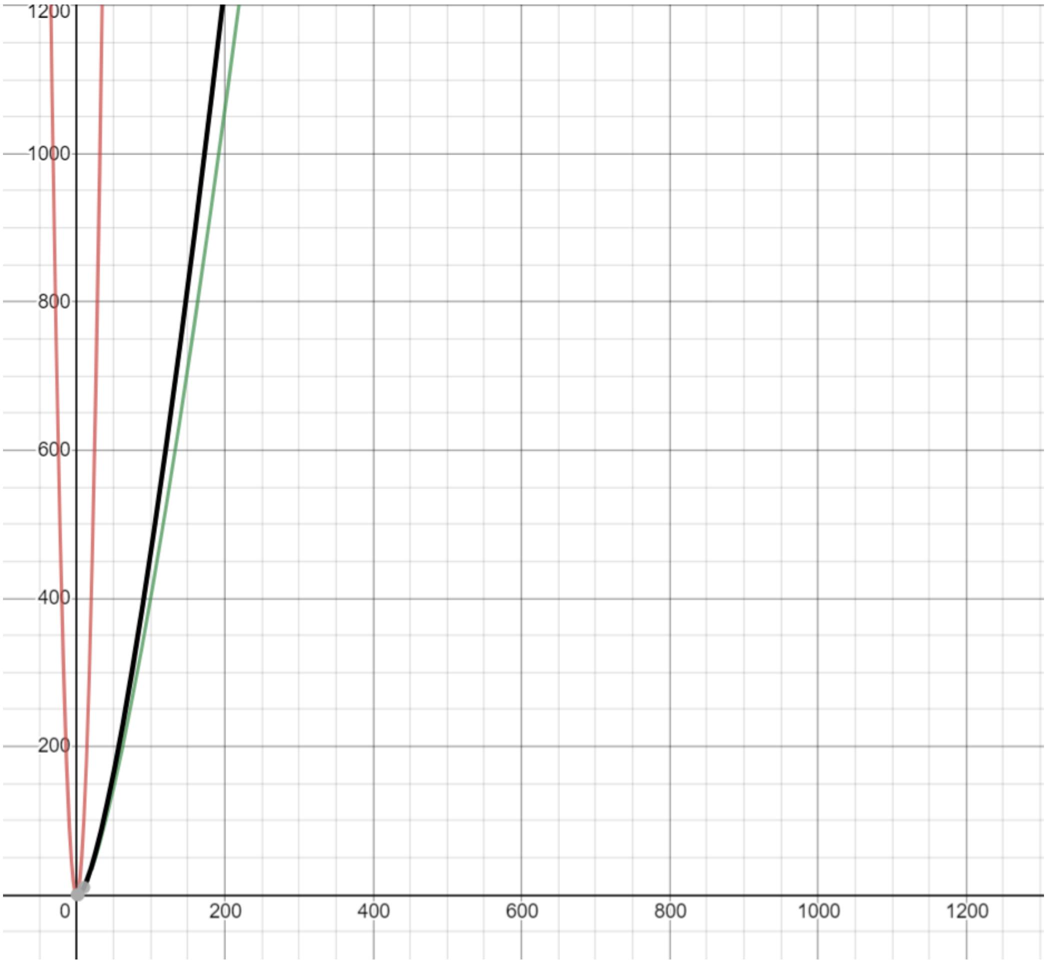
（下图红色为Prim算法n^2，绿色为Prime + heap算法elogn，黑色为Kruskal算法eloge）

我使用的绘图工具

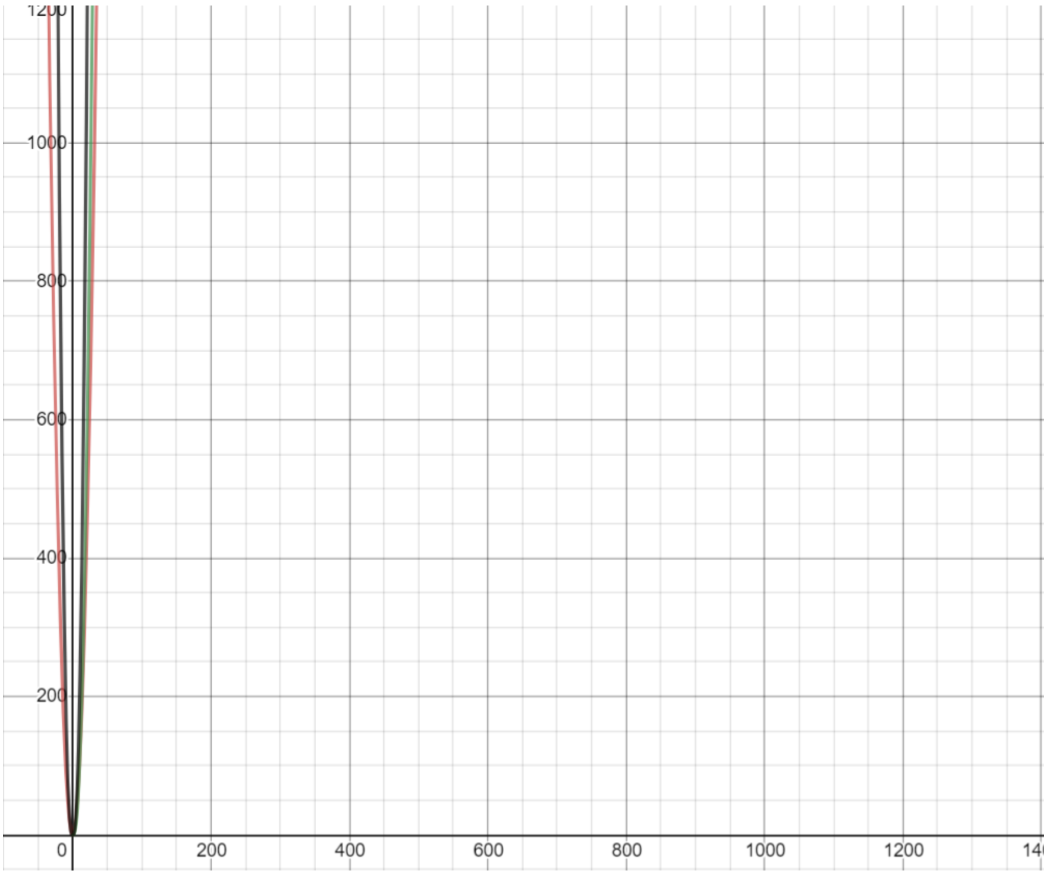
首先是e=n



接着是e=nlogn



然后是e=n^2



1.Prim在稠密图中比Kruskal优，在稀疏图中比Kruskal劣。

2.Prim+Heap在任何时候都有令人满意的的时间复杂度，但是代价是空间消耗极大。