算法基础（十四）： 图 - 最小生成树算法

最小生成树算法看下图：

如果是稠密图的话`m`的量级是`n`的大约平方倍，用`Kruskal`算法没有优势，就用朴素Prim算法，代码短，思路清晰

如果是稀疏图的话`m`和`n`是一个量级，就用克鲁斯卡尔算法

堆优化版本的Prim算法不常用，这里不再赘述

朴素Prim(朴素普利姆算法)

基本思想：

伪码如下：

```
dsit[i] <- INF;
for(i = 0; i < n; i++)
{
    t <- 找到集合外距离集合最近的点;
    用t更新其他点到集合的距离;
    st[t] = true;
}
```

基本过程就是找到集合外距离集合最近的点，这里一个点到集合的距离指的是这个点到集合的内部所有边中最短的那条边

找到这个点后将这个点以及最短距离对应的边加入到集合S，作为最小生成树的一个结点以及一条边

然后用这个点再更新集合外的所有的点到集合的距离

重复上述操作`n`次即可，两重循环`n ^ 2`的复杂度

注意几个问题：

1. 最小生成树问题都是无向图
2. 一开始是随机加入一个点进入集合S，然后更新其他的点

代码实现：

给定一个 n 个点 m 条边的无向图，图中可能存在重边和自环，边权可能为负数。

求最小生成树的树边权重之和，如果最小生成树不存在则输出 `impossible`。

给定一张边带权的无向图 $G = (V, E)$，其中 V 表示图中点的集合，E 表示图中边的集合，$n = |V|$，$m = |E|$。

由 V 中的全部 n 个顶点和 E 中 $n - 1$ 条边构成的无向连通子图被称为 G 的一棵生成树，其中边的权值之和最小的生成树被称为无向图 G 的最小生成树。

输入格式

第一行包含两个整数 n 和 m。

接下来 m 行，每行包含三个整数 u, v, w，表示点 u 和点 v 之间存在一条权值为 w 的边。

输出格式

共一行，若存在最小生成树，则输出一个整数，表示最小生成树的树边权重之和，如果最小生成树不存在则输出 `impossible`。

数据范围

$1 \leq n \leq 500$,
$1 \leq m \leq 10^5$,
图中涉及边的边权的绝对值均不超过 10000。

输入样例：

```
4 5
1 2 1
1 3 2
1 4 3
2 3 2
3 4 4
```

输出样例：

```
6
```

```cpp
#include<iostream>
#include<cstring>
#include<algorithm>
using namespace std;
const int N = 510, INF = 0x3f3f3f3f;

int n, m;
int g[N][N];//邻接矩阵用来存图中的所有的边
int dist[N];//dist[i]表示点i到集合S的距离
bool st[N];//状态数组用来表示点是否在集合S中


int prim()
{
    memset(dist, 0x3f, sizeof dist);

    int res = 0;
    //迭代n次
    for(int i = 0; i < n; i ++)
```

```
    {
        int t = -1;
        //找到集合外的到集合距离最短的一个点
        for(int j = 1; j <= n; j ++)
        {
            //这个点在集合外
            if(!st[j])
            {
                //t = -1的时候表示第一次for循环
                //这时显然得将j赋给t
                //然后剩下的循环找到集合外距离集合最近的点
                if(t == -1 || dist[t] > dist[j])
                {
                    t = j;
                }
            }
        }
        //如果不是第一个点，并且最近的点到集合的距离为正无穷
        //说明这个图不连通，返回
        if(i != 0 && dist[t] == INF) return INF;
        //否则，在不是第一个点的情况下
        //我们将这个最短距离的点加入到最小生成树的权值中
        else if(i != 0)
        {
            res += dist[t];
            //cout << "jj" << dist[t];
        }
        //把点加入到集合中
        st[t] = true;
        //更新其他的点到集合的距离
        //注意我们统一更新所有的点，不用区分是否在集合S中
        //对于已经在集合S中的点(包括点t)，哪怕更改了dist也没关系
        //因为已经加进res中了
        //对于不再S中的点，在没有更新之前这些点到集合的距离是点到集合中的点的边的值
        //如果发生更改，那必然是同t与这些点的边进行比较
        for(int j = 1; j <= n; j ++)
        {
            dist[j] = min(dist[j], g[t][j]);
            //if(t == 1) cout << "sss:" << dist[j];
        }


    }
    return res;



}


int main()
{
```

```
    scanf("%d%d", &n, &m);
    memset(g, 0x3f, sizeof g);

    while(m --)
    {
        int a, b, c;
        scanf("%d%d%d", &a, &b, &c);
        //无向图等价于有向图的一条边有两个方向
        //对于重边的处理与图的前面的几个算法相似，都是存储最短的那条边
        g[a][b] = g[b][a] = min(g[a][b], c);
    }

    int t = prim();

    //最小生成树不存在的情况
    //当所有的点不连通的时候最小生成树不存在
    if(t == INF) puts("impossible");
    else cout << t << endl;


    return 0;
}

```

需要注意的几个问题：

1. 重边的处理问题，上面代码中有详解，不再赘述
2. 自环的问题，自环并不影响算法的正确性，假设算法运行到中间的某个状态
    1. 在寻找不在集合`S`中的点到集合的距离的时候始终比较的是`dist`，与自环无关
    2. 在将点`t`加入到集合`S`中后更新其他点的距离的时候，自环只会对当前`t`的`dsit`有更新效果，但是哪怕更新过了也没有影响，因为**在更新之前`res`已经加上了`dist[t]`**，注意这点，更新`res`要在更新其他的`dist[]`之前做
    3. 自环也不会影响初始状态，因为初始点加入集合`S`后更新其他的点，也只是跟点与点之间的边有关，跟自环没有关系
3. 对于负环，这个算法跟有没有负环没有关系，更新`dsit`以及进行比较的时候都是至于点与点之间的边有关
4. 对于无向图来讲，他跟有向图相比多出来的性质就是一条边的两个点都互相可达，所以在存储无向图的时候我们将一条边存入两个方向就等价于存储无向图了

如果我们使用堆优化的话，找集合外距离最近的点的复杂度在一次循环中是`O(1)`，`n`次循环就是`O(n)`

用`t`更新堆的时候，每次外循环我们都会得到一个点`t`，然后遍历`t`的所有`g[t][j]`， 所以所有的外循环加起来，我们是遍历了所有的边，总的复杂度是`m`，而进行堆优化的话，更新一堆中的一个数据的复杂度是`O(logn)`，所以总的复杂度是`mlog(n)`，复杂度分析与`Dijkstra`的情况完全一样

Kruskal算法（克鲁斯卡尔算法）

基本思想

1. 先将所有的边按权重从小到大排序（算法瓶颈，实现复杂度`O(mlogm)`，并且注意的是，`kmlogm`的常数部分`k`很小
2. 一开始是集合中只有点，没有边，所有的点都不连通
3. 然后枚举每条边`a - b(w:c)`
    1. 如果`a`，`b`不连通，就把这条边加入到集合里面去

使用的思想是并查集和快排

并查集的查找和合并的复杂度都是`O(1)`，所以时间复杂度主要在排序上面，也就是`mlog(m)`

代码实现

给定一个 n 个点 m 条边的无向图，图中可能存在重边和自环，边权可能为负数。

求最小生成树的树边权重之和，如果最小生成树不存在则输出 `impossible`。

给定一张边带权的无向图 $G = (V, E)$，其中 V 表示图中点的集合，E 表示图中边的集合，$n = |V|$，$m = |E|$。

由 V 中的全部 n 个顶点和 E 中 $n - 1$ 条边构成的无向连通子图被称为 G 的一棵生成树，其中边的权值之和最小的生成树被称为无向图 G 的最小生成树。

输入格式

第一行包含两个整数 n 和 m。

接下来 m 行，每行包含三个整数 u, v, w，表示点 u 和点 v 之间存在一条权值为 w 的边。

输出格式

共一行，若存在最小生成树，则输出一个整数，表示最小生成树的树边权重之和，如果最小生成树不存在则输出 `impossible`。

数据范围

$1 \leq n \leq 10^5$,
$1 \leq m \leq 2 * 10^5$,
图中涉及边的边权的绝对值均不超过 1000。

输入样例：

```
4 5
1 2 1
1 3 2
1 4 3
2 3 2
3 4 4
```

输出样例：

```
6
```

```cpp
#include<iostream>
#include<algorithm>
using namespace std;
```

```cpp
const int N = 200010;

int n, m;

//p[N]用来初始化并查集
//p[x] = k表示x节点的父结点是k
//经过路径压缩之后路径上的点的p[x]都是对应的集合的根结点
int p[N];

//用结构体来存储边，不需要邻接表或是邻接矩阵
struct Edge
{
    int a, b, w;
    //重载一下小于号
    //重载之后就可以在下面的代码中使用edge[i] < edge[k]这样的语句了
    bool operator< (const Edge &W) const
    {
        return w < W.w;
    }
}edges[N];


//并查集路径压缩方法寻找根结点
int find(int x)
{
    if(x != p[x]) p[x] = find(p[x]);
    return p[x];
}


int main()
{
    scanf("%d%d", &n, &m);

    //读入所有的边
    for(int i = 0; i < m; i ++)
    {
        int a, b, w;
        scanf("%d%d%d", &a, &b, &w);
        edges[i] = {a, b, w};
    }
    //对边进行排序，sort函数是左闭右开的，左边是第一个元素，右边是尾元素的下一个元素
    //sort函数默认的是从小到大排序
    sort(edges, edges + m);

    //初始化并查集
    for(int i = 1; i <= n; i ++) p[i] = i;

    //res存放最小生成树所有边的权值之和
    //cnt存放是当前我们加了多少条边
    int res = 0, cnt = 0;

```

```
    //从小到大枚举所有的边
    for(int i = 0; i < m; i ++)
    {
        int a = edges[i].a;
        int b = edges[i].b;
        int w = edges[i].w;

        //a的祖宗节点
        a = find(a);

        //b的祖宗结点
        b = find(b);
        //如果祖宗结点不连通，就表示a和b不连通
        if(a != b)
        {
            //将边加入到最小生成树中
            res += w;
            //边的个数++
            cnt ++;
            //合并集合，a的祖宗结点的父结点指向b的祖宗结点
            //需要注意的是我们合并集合的时候是用a所在的树的根结点的父结点指向b所在的树的父结点
            //但在kruskal算法中两个连通块实际合并时时因为边才变得连通
            //我们用并查集只是用来表明两个连通块合并，是用来标记的，与实际有所不同
            p[a] = b;
        }

    }
    //如果最后的边数小于n - 1说明这样的树不存在
    //也就是图原本就不连通
    if(cnt < n - 1) puts("impossible");
    else printf("%d\n", res);

    return 0;
}
```

因为`Kruskal`是先对所有边从小到大排序的，这样子就和之前读入边的时候取最小值是一个道理，先把最小边连进去了，后面的重边再连的时候两个点已经再一个连通块中了自然就没有影响了。关于`Kruskal`中的自环，即`a=b`的情况，那么很容易就可以被并查集的`if`判断给过掉，即显然两个点在一个连通块中，自然也没有影响