图的建立及遍历

这是写给刚刚接触图论的朋友们看的~

有向图和无向图

~~这决定这我们建图的方式~~

无向边：边是双向的

有向边：单向边，有箭头

无向图：只有无向边的图

有向图：只有有向边的图

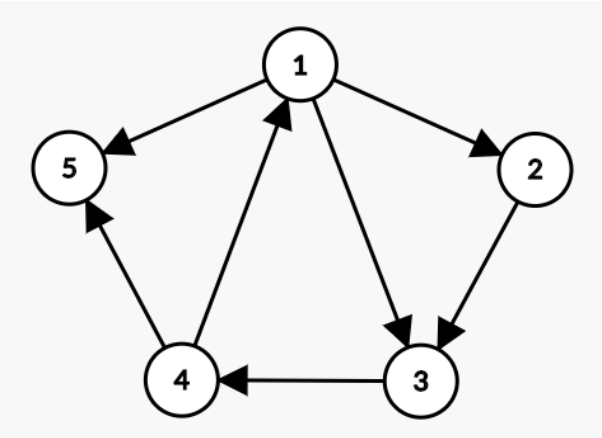
（写在前面的话：下文中所有的n代表的是结点的个数，m代表的是边的条数。$maxn$代表结点的最多个数，$maxm$代表最多有多少条边)

图的存储方式

1.邻接矩阵

这个就是用一个最直观的二维数组来存储，当然理解起来也最为容易，它可以很直接明了的看出两点之间的连边关系，但是它的局限性很多呀，因为用二维数组来存对于空间的消耗是很大的（空间复杂度是$O(n^2)$），这其中大量的空间都没有用到，当数据大一点的时候，是会爆炸的，一般int型的数据当n到10^4左右的数据就不能用这种方式存图了

其存图方式就是如果$i \to j$有边的话，就将$mp[i][j]$标记为1



$$\begin{pmatrix} 0 & 1 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

如果是双向图的话也很简单，双向图如果i、j之间有边，则一定是$i \leftrightarrow j$，所以$mp[i][j]$和$mp[j][i]$都要标记为1

可以发现双向图的邻接矩阵是个对称矩阵。

这个画一画就知道啦，图就不放了

~~毕竟邻接矩阵不是我们主要要了解的方法QAQ~~

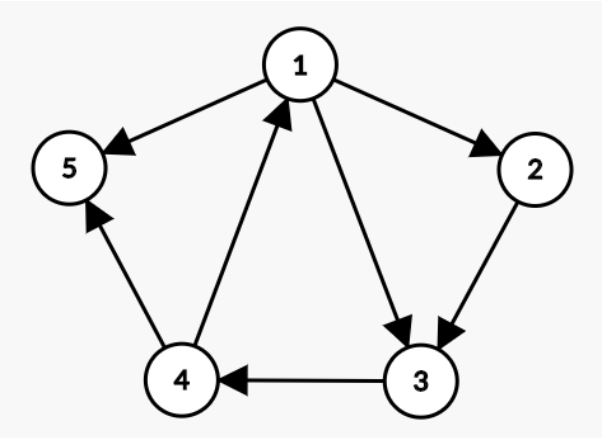
2.邻接表

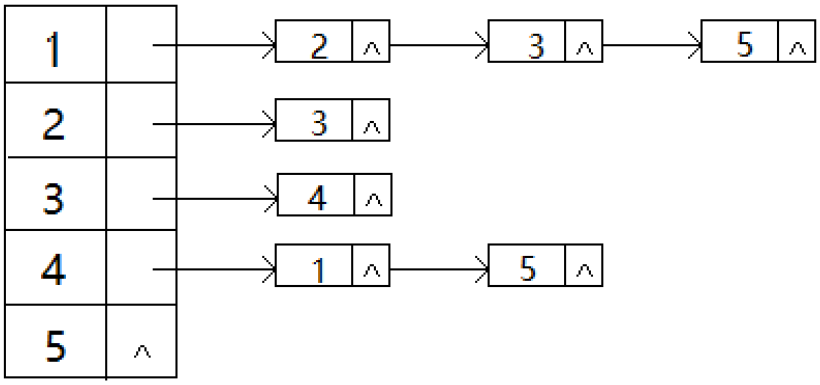
这个就是对图中的每一个点建一个单链表，每个点的链表中存的是这个点的所有邻接节点。

其优点是可以节省空间（~~相比邻接表来说节省的不是一点点~~），很适合稀疏图，而大部分图其实都比较稀疏，所以这个方法需要好好掌握。

> 数据结构中对于稀疏图的定义为：有很少条边或弧（边的条数|E|远小于|V|²）的图称为稀疏图，反之边的条数|E|接近|V|²，称为稠密图

举个栗子：





其建图方式就是将某点的所有临接结点都加到该点的链表中

比方说1的出度为3，分别是$1 \rightarrow 2$、$1 \rightarrow 3$、$1 \rightarrow 5$，所以其邻接节点有2、3、5，将2、3、5顺次加入1的单链表中。

这样建图基本上是有多少条边就会存多少进去，大大的减少了开辟无用的空间。

所以这个方法其实对于平时碰到的数据都是够用了的。

理想的空间复杂度：

有向图：$O(n+m)$　　**无向图：**$O(n+2\times m)$

这个也很好理解，可以想象成一个点有多少条向外的连边，那么它对应的单链表后面就有多少个结点。固定的对于有n个点的图来说，一定有每个都建一个单链表的头结点，固定空间消耗$O(n)$

- 对于有向图来说，m条有向边，每一条边都会通向一个结点，所以额外的有m个结点。即$O(n+m)$
- 对于无向图来说，可以将每条无向边看成是两条有向边，即$i \leftrightarrow j$可以理解为既有$i \rightarrow j$、又有$j \rightarrow i$，所以每条边产生的额外结点有$2\times m$个。即$O(n+2\times m)$

建图方式

一般这个单链表我们都是用数组模拟的，但是我们在存边的时候会不停的向后面加点，那每个点的单链表开多大合适呢，这个就很麻烦。这时候STL大法就要出场啦，里面有个特别好用的东西叫vector，这是一个动态数组，就是你不需要像定义我们的普通数组一样，一开始要预定好所有的内存，这个是你加进去的空间会随着你加进去的数据进行增长的，所以很方便。

```
vector<int> e[maxn];   //对每个点建一个单链表
//存边
for(i=1;i<=m;i++){
    scanf("%d%d",&u,&v);
    e[u].push_back(v);     //u->v有一条有向边，所以将v加入到u的单链表中
    //如果是双向边的话，就将u也加到v的单链表中就好了，表示v->u也有一条有向边。也就是下面这一句
    e[v].push_back(u);
}
```

但是这个方法不太好存$u \to v$之间的边权，除非用$pair$类型封装一下，但是这个用的也不是很多。用的很多的还是下面的方法。

3.链式前向星

压轴出场的，就是它啦~ 这个要重点把握，因为以后遇到图论的题目，基本上很少用上面那两种方法，这个是用的最多的，但同时也比较难理解，所以要耐心的看。

链式前向星是一种特殊的**边集数组**,我们将同起点的边存在一起，将**起点相同**的边按照边的输入顺序**逆着存储**，这样在遍历时是倒着遍历的,也就是说与输入顺序是相反的,不过这样不影响结果的正确性（~~因为反正每条边都会遍历到~~）

先来感受一下链式前向星的基本存储方式：

```
int head[maxn],cnt;  //head[i]保存的是以i为起点的所有边中编号最大的边    cnt是当前总共存了多少条边
struct edge{
    int v;     //【终点】
    int w;     //【边的权值】这个有没有要视情况而定，因为有些题只是表示u->v有边，而没有边权
    int next;  //【下一条边的编号】
};
edge e[maxm];
```

其中$e[i].v$表示第i条边的终点,$e[i].next$表示与第i条边同起点的下一条边的存储位置,$e[i].w$为边权值.

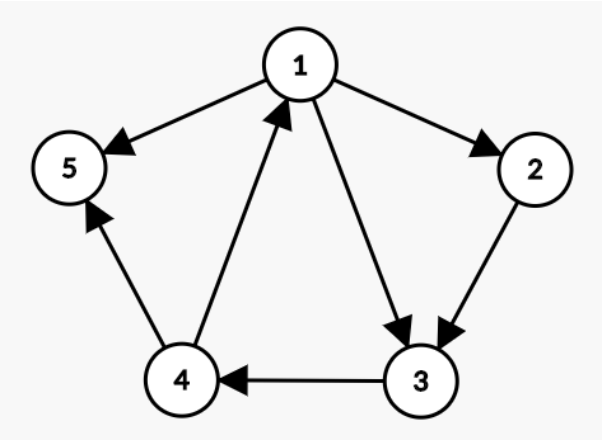
还有一个数组$head[]$,它是用来表示以i为起点的第一条边存储的位置,实际上这里的第一条边存储的位置其实在以i为起点的所有边的最后输入的那个编号.因为之前说过了，链式前向星本质上是逆着将边存储进去的。

$head[]$数组一般初始化为-1或0，对于加边的add函数是这样的:

```
void add(int u,int v,int w){  //u：起点  v：终点   w：边权
    ++cnt;
    e[cnt].v=v;
    e[cnt].w=w;    //这一步有没有也视题目而定，有些题没有边权就不需要这一步
    e[cnt].next=head[u];
    head[u]=cnt;
}
```

初始时cnt为0，我们每加一条边，都将$cnt++$。首先将终点和权值都添加上去，这两步都很好理解。

下面那两步我们来模拟一下：



还是这张图，有边输入顺序如下：

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

我们来模拟一下：($head[i]$都初始化为-1)

$$e[1].v=2;e[1].next=-1;head[1]=1;$$

$$e[2].v=3;e[2].next=-1;head[2]=2;$$

$$e[3].v=4;e[3].next=-1;head[3]=3;$$

$$e[4].v=3;e[4].next=1;\quad head[1]=4;$$

$$e[5].v=1;e[5].next=-1;head[4]=5;$$

$$e[6].v=5;e[6].next=4;\quad head[1]=6;$$

$$e[7].v=5;e[7].next=5;\quad head[4]=7;$$

很明显，当遇到相同起点的边的时候，都用之前的$head$来更新当前边的$next$，其实就是在这两条同起点的边之间建立联系，使得后输入的边可以通过$next$访问到之前的边，这样就可以将同起点的边全部访问到了。

$head[i]$保存的是以i为起点的所有边中编号最大的那个,而把这个当作顶点i的第一条起始边的位置。往后遇到的每一个$next$，其实都是以i为起点的前一条边，我们是将它逆着存储。同样的，这样在遍历时是倒着遍历的,也就是说与输入顺序是相反的。

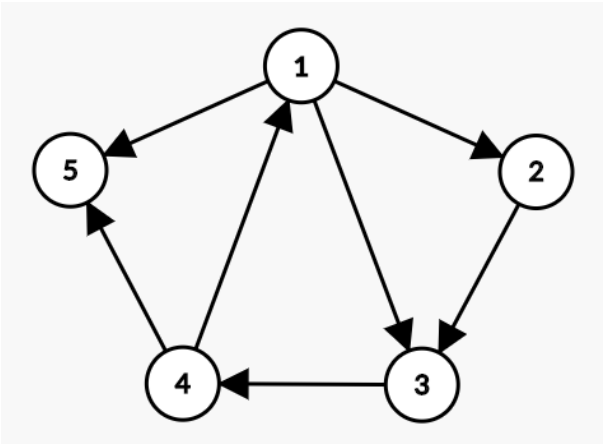
这里有个问题：为什么**边集数组里面不用存储u**呢?

因为我们之前说了，链式前向星是以某个点i为相同起点的边才会存储在一起，而这些边通过$next$可以逆向找到，我们又将最后一条边的编号存储在$head[i]$，这样通过$head[i]$，然后依次访问通过的每一个$next$，就可以找到所有以i为起点的边了。

那么这种存储方式要怎么遍历呢

```
//将head[i]初始化为-1的遍历
for(int i=head[u];~i;i=e[i].next){  //遍历以u为起点的所有边
    int v=e[i].v;
    //下面是要实施的操作...
}
```

```
//将head[i]初始化为0的遍历
for(int i=head[u];i;i=e[i].next){  //遍历以u为起点的所有边
    int v=e[i].v;
    //下面是要实施的操作...
}
```



再放一次图，方便观察

比如以上图为例,以节点1为起点的边有3条,它们的编号(上文的输入顺序)分别是1,4,6 而$head[1]=6$

那么就是说先遍历编号为6的边,也就是$head[1]$,然后就是$e[5].next$,也就是编号4的边,然后继续$e[3].next$,也

就是编号1的边,可以看出是逆序的.

至此，链式前向星就讲完啦~

例题讲解

1.最小生成树

题目：P3366 【模板】最小生成树

题目大意：给出一个无向图，求出最小生成树，如果该图不连通，则输出 `orz`。

下面的两种方法都是用prim算法写的。这个不了解的可以去题解里学习一下。这里主要是展示一下不同的建图方式怎么遍历，然后思路也给了很详细的注释。

这里用了两种方法建图：一是万能的链式前向星，二是邻接矩阵（~~找这个题就是看中它数据范围小，可以演示一下邻接矩阵的用法QAQ~~）

法一：链式前向星存图

```
//P3366 【模板】最小生成树     链式前向星
#include<iostream>
#include<cstdio>
using namespace std;
#define maxn 5005
#define maxm 200005
#define INF 1e9

int n,m;
int dis[maxn];
bool vis[maxn];   //vis标记点i是否已经加入所选的点集中
int ans;
int head[maxn],cnt;
struct edge{
    int v;
    int w;    //这里每条边会有边权，所以要定义w变量
    int next;
}e[maxm*2];   //这里注意哦。因为题目说是双向边，所以要开两倍的空间
void add(int u,int v,int w){
    e[++cnt].v=v;
    e[cnt].w=w;
    e[cnt].next=head[u];
    head[u]=cnt;
}

bool prim(){
    int i;
    //先将所有点的距离初始化为无穷
    for(i=1;i<=n;i++){
        dis[i]=INF;
    }
    vis[1]=1;
    //将1作为第一个点，用它去更新其他的点
    for(i=head[1];i;i=e[i].next){  //这里head数组初始化是0
        int v=e[i].v;
        dis[v]=min(dis[v],e[i].w);   //这是为了防止1和v之间有重边，我们取其中边权最小的
    }
    //每次我们都要选一个最小的dis出来，加入已选点的集合，然后用它去更新其他未选点的dis
    int min_path=INF,now=1;
    for(int tot=2;tot<=n;tot++){  //tot记录个数，一共要选n个点，刚刚已经选了一个，现在要选第2个了
        min_path=INF;
        for(i=1;i<=n;i++){
            if(!vis[i]&&dis[i]<min_path){//如果这个点还没有被选出来，并且到它的距离比到其他点距离更短，就更新
                min_path=dis[i];
                now=i;   //记录下这个点，之后还要用去更新别的点
            }
        }
        if(min_path==INF) return false;  //说明没有更新过，此时没有最小生成树
        //我们将刚刚上面选好的now加入到已选则的点集里去，并且用它去更新和它有连边的点的dis
        ans+=min_path;
        vis[now]=1;
        for(i=head[now];i;i=e[i].next){
            int v=e[i].v;
```

```
            if(!vis[v]&&dis[v]>e[i].w){
                dis[v]=e[i].w;
            }
        }
    }
    return true;
}

int main(){
    cin>>n>>m;
    int i;
    int u,v,w;
    for(i=1;i<=m;i++){
        scanf("%d%d%d",&u,&v,&w);
        add(u,v,w);//这里我们简单的提过，如果是双向边，可以看成是u->v和v->u都有边，所以反着也要加一次
        add(v,u,w);    //这也是无向图要开两倍空间的原因
    }
    if(!prim()) cout<<"orz";
    else cout<<ans;
    return 0;
}
```

法二：邻接矩阵存图

```
//P3366 【模板】最小生成树     邻接矩阵
#include<iostream>
#include<cstdio>
using namespace std;
#define maxn 5005
#define maxm 200005
#define INF 1e9

int n,m;
int mp[maxn][maxn];
int dis[maxn];
bool vis[maxn];   //vis标记点i是否已经加入所选的点集中
int ans;

void init(){
    int i,j;
    //初始化mp数组
    for(i=1;i<=n;i++){
        for(j=1;j<=n;j++){
            mp[i][j]=INF;
        }
    }
    //初始化dis数组
    for(i=1;i<=n;i++){
        dis[i]=INF;
    }
}

bool prim(){
    //还是将1作为起始点拖出来更新其他的点
    vis[1]=1;
    int i;
```

```
    for(i=1;i<=n;i++){
        dis[i]=mp[1][i];
    }
    int min_path,now=1;
    for(int tot=2;tot<=n;tot++){
        min_path=INF;
        for(i=1;i<=n;i++){
            if(!vis[i]&&dis[i]<min_path){  //选出当前dis最小的点，然后用它去更新其他的
点
                min_path=dis[i];
                now=i;
            }
        }
        if(min_path==INF) return false;

        //将刚刚选出的now拖出来标记已选，并且去更新和它有连边的点的dis
        ans+=min_path;
        vis[now]=1;
        for(i=1;i<=n;i++){
            if(dis[i]>mp[now][i]) dis[i]=mp[now][i];
        }
    }
    return true;
}

int main(){
    cin>>n>>m;
    init();
    int i;
    int u,v,w;
    for(i=1;i<=m;i++){
        scanf("%d%d%d",&u,&v,&w);
        mp[u][v]=min(mp[u][v],w);
        mp[v][u]=min(mp[v][u],w);
    }
    if(!prim()) cout<<"orz"<<endl;
    else cout<<ans;
    return 0;
}
```

2.查找文献

题目：P5318 【深基18.例3】查找文献

题目大意：每篇文章可能会有若干个（也有可能没有）参考文献的链接指向别的博客文章。如果小k看了某篇文章，那么他一定会去看这篇文章的参考文献（如果他之前已经看过这篇参考文献的话就不用再看它了）。目前已经打开了编号为 1 的一篇文章，请 设计一种方法，使小 K 可以不重复、不遗漏的看完所有他能看到的文章。请对这个图分别进行 DFS 和 BFS，并输出遍历结果。如果有很多篇文章可以参阅，请先看编号较小的那篇(因此你可能需要先排序)。

思路：这个题就是很基础的图上dfs和bfs了。需要注意的地方题目里也说了，就是要编号小的尽量先输出，所以要先排序。

```
//P5318 【深基18.例3】查找文献     邻接表建图
#include<iostream>
```

```cpp
#include<cstdio>
#include<queue>
#include<cstring>
#include<algorithm>
using namespace std;
#define maxn 100005
#define maxm 1000005

int n,m;
bool vis[maxn];
vector<int> e[maxn];   //对每个点建一个单链表

void dfs(int now){
    cout<<now<<" ";    //每遍历到一个点就将它输出来
    for(int i=0;i<e[now].size();i++){
        int v=e[now][i];
        if(vis[v]) continue;
        vis[v]=1;
        dfs(v);
    }
}

void bfs(int s){  //s为起始点
    queue<int> q;   //bfs要借助队列来辅助，因为其特性是先进先出
    q.push(s);
    vis[s]=1;
    while(!q.empty()){
        int now=q.front();
        q.pop();
        cout<<now<<" ";    //每遍历到一个点就将它输出来
        for(int i=0;i<e[now].size();i++){
            int v=e[now][i];
            if(vis[v]) continue;
            vis[v]=1;
            q.push(v);
        }
    }
}

int main(){
    cin>>n>>m;
    int i;
    int u,v;
    for(i=1;i<=m;i++){
        scanf("%d%d",&u,&v);
        e[u].push_back(v);    //u->v有一条有向边，所以将v加入到u的单链表中
    }
    //因为要将编号小的尽量先输出，所以就将每个点的单链表里的点排个序
    for(i=1;i<=n;i++){
        sort(e[i].begin(),e[i].end());
    }
    vis[1]=1;   //标记
    dfs(1);
    cout<<endl;
    memset(vis,0,sizeof(vis));  //将vis数组清零，因为dfs用过之后有些点有标记会影响后面的
bfs的遍历
    bfs(1);
    cout<<endl;
```

```
    return 0;
}
```