图论

最短路

Dijkstra

```
//使用邻接表存储稀疏图
int n,m,idx;
int h[MAXN],e[MAXN],ne[MAXN],w[MAXN];//h[]存储每个邻接表上的头结点；ne[]存的是每个节点的下一个节点，即next；w存储权重
int dis[MAXN];
bool st[MAXN];
//pair的first存的是距离，second存的是编号
void add(int a,int b,int c){
    e[idx]=b,w[idx]=c;
    ne[idx]=h[a],h[a]=idx++;
}
int dijkstra(){
    memset(dis,0x3f,sizeof(dis));
    dis[1]=0;

    priority_queue<pii,vector<pii>,greater<pii>> heap; //存储一个小根堆
    heap.push({0,1});

    while(heap.size()){
        auto u = heap.top();heap.pop();

        int ver = u.second,distance = u.first;  //ver存储点的序号，distance存储距离
        if(st[ver])
            continue;
        st[ver]=true; //这nm千万别忘了啊
        for(int i=h[ver];i!=-1;i=ne[i]){
            int j=e[i];
            if(dis[j]>distance+w[i]){
                dis[j]=distance+w[i];
                heap.push({dis[j],j});
            }
        }
    }
    if(dis[n]==0x3f3f3f3f) return -1;
    return dis[n];
}
```

SPFA

```
#include<iostream>
#include<string>
#include<algorithm>
#include<string.h>
#include<queue>
```

```
#define MAXN 160010
using namespace std;

typedef pair<int,int> pii;

//使用邻接表存储稀疏图
int n,m,idx;
int h[MAXN],e[MAXN],ne[MAXN],w[MAXN];//h[]存储每个邻接表上的头结点；ne[]存的是每个节点的下一个节点，即next；w存储权重
int dis[MAXN];
bool st[MAXN];
//pair的first存的是距离，second存的是编号
int spfa(){
    memset(dis,0x3f,sizeof(dis));
    dis[1]=0;

    queue<int> Q;
    Q.push(1);
    st[1]=true;
    while(Q.size()){
        int t=Q.front();Q.pop();
        st[t]=false;

        for(int i=h[t];i!=-1;i=ne[i]){
            int j=e[i];
            if(dis[j]>dis[t]+w[i]){
                dis[j]=dis[t]+w[i];
                if(!st[j]){
                    Q.push(j);
                    st[j]=true;
                }
            }
        }
    }
    if(dis[n]==0x3f3f3f3f) return -1;
    return dis[n];
}
```

求负环

```
bool spfa(){
    //memset(dis,0x3f,sizeof(dis));
    //dis[1]=0;
    queue<int> Q;
    for(int i=1;i<=n;i++){
        st[i]=true;
        Q.push(i);
    }
    while(Q.size()){
        int t=Q.front();Q.pop();
        st[t]=false;

        for(int i=h[t];i!=-1;i=ne[i]){
            int j=e[i];
            if(dis[j]>dis[t]+w[i]){
                dis[j]=dis[t]+w[i];
                cnt[j] = cnt[t]+1;
```

```
                if(cnt[j]>n) return true;
                if(!st[j]){
                    Q.push(j);
                    st[j]=true;
                }
            }
        }
    }
    return false;
}
```

Floyd

```
for(k=1~n)
    for(i=1~n)
        for(j=1~n)
            d[i][j]=min(d[i][j],d[i][k]+d[k][j];
```

- 传递闭包
- 恰好经过k条边的最短路径--------改进Floyd

bitset优化传递闭包

```
void solve(){
    n=read(),m=read();
    rep(i,1,m) {
        int x=read(),y=read();
        a[x][y]=1;
    }
    rep(k,1,n) rep(i,1,n) if(a[i][k]) a[i]|=a[k];
    LL ans = 0;
    rep(i,1,n) ans += a[i].count();
    print(((n*(n-1))>>1)-ans);
}
```

最小生成树

Prim算法

```
#define MAXN 510
#define INF 0x3f3f3f3f
int n,m;
int g[MAXN][MAXN],dis[MAXN];
bool st[MAXN];

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

    int res=0;
    //n次迭代
    for(int i=1;i<=n;i++){
        int t=-1;
```

```
        for(int j=1;j<=n;j++)
            if(!st[j]&&(t==-1||dis[t]>dis[j]))
                t=j;

        if(i!=1&&dis[t]==INF) return INF;
        if(i!=1) res+=dis[t];

        for(int j=1;j<=n;j++)
            dis[j]=min(dis[j],g[t][j]);

        st[t]=true;
    }
    return res;
}
```

二分图

性质

二分图不存在奇数环，染色法不存在矛盾问题

- 如果一个图中存在奇数环，那么这个图一定不是二分图；这一点显然成立。
- 如果一个图中不存在奇数环，那么这个图一定是二分图;

König定理（柯尼定理）：

> 匹配：即没有公共点的边
>
> 最大匹配：边数最多的匹配
>
> 匹配点：在匹配中的点
>
> 增广路径：从一个非匹配点走，依次走非匹配边与匹配边，直到通过非匹配边走到一个非匹配点

最小点覆盖，最大独立集，最小路径点覆盖，最小路径重复点覆盖

最大匹配数=最小点覆盖=总点数-最大独立集=总点数-最小路径覆盖

> 最小点覆盖:从一个图中选出最少的点使得使得每一条边至少有一个端点被选出来

> 一个棋盘可以用1*2或者2*1的方块来覆盖，但是有一些点可以不用覆盖，每个格子可以重复被覆盖。问最少需要多少个方块可以覆盖所有需要覆盖的点；
>
> 算法：先做一遍二分图的最大匹配，染色方法为每个格子向相邻的四个格子染色，然后得到的最大匹配数为最大匹配的两倍，因此还需要总需要覆盖点数-最大匹配数个点需要一个方块覆盖一个点。

证明:最小点覆盖数=最大匹配数
[1]最小点覆盖数>=最大匹配数:
    每一条边都选出一个点即可
[2]最小覆盖数与最大匹配数之间可取等号:
将一个图分成两半，构造:
    从左边每个非匹配点做一遍匈牙利算法，并标记所有经过的点。最终选出左边未被标记的点和右边被标记的点

上述构造满足三个性质:
    1)左边所有未被标记的点都是匹配点
    2)右边所有被标记的点都是匹配点
    3)对于匹配边，要么左右全被标记，要么全不被标记

因此选出来的点恰好可以满足最小覆盖，因为每个匹配边有且只有一个点被选

- *最大独立集：从一个图中选出最多的点使得选出的点之间无边<=>去掉最少的点将所有边都破坏掉*
- *最大团：选出最大的点使得任意两点之间都有边*
- *最小路径点覆盖：在DAG中用最少的互不相交的路径（从起点连连连到终点）将所有点覆盖*，拆点将每个点分成出点和入点，使得原图变成一个二分图
- *最小路径重复点覆盖*:
  - [1]求传递闭包(如果一个点间接连向另外一个点的话就直接加一条边)得到G'
  - [2]在G'上求最小路径覆盖

染色法

```
bool dfs(int x,int k){
    color[x]=k;
    for(int i=h[x];i!=-1;i=ne[i]){
        int j=e[i];
        if(!color[j]){
            if(!dfs(j,3-k)) return false;
        }
        else if(color[j]==k) return false;
    }
    return true;
}
//之后的main函数内:
for(int i=1;i<=n;i++){
    if(!color[i]){
        if(!dfs(i,1)){
            flag=false;
            break;
        }
    }
}
if(!flag) puts("No");
else puts("Yes");
```

匈牙利算法

```
bool find(int x){
    for(int i=h[x];i!=-1;i=ne[i]){
        int j=e[i];
        if(st[j]) continue;
```

```
        st[j]=true;
        if(match[j]==0||find(match[j])){
            match[j]=x;
            return true;
        }
    }
    return false;
}

int main()
{
    for(int i=1;i<=n;i++)
    {
        memset(st,0,sizeof st);
        if(find(i)) cnt ++;  //cnt为二分图最大匹配数
    }
}
```

KM二分图最优匹配

```
#include<cstdio>
#include<cstring>
#include<algorithm>
#include<iostream>
#include<string>
#include<vector>
#include<stack>
#include<bitset>
#include<cstdlib>
#include<cmath>
#include<set>
#include<list>
#include<deque>
#include<queue>
#include<map>
#define ll long long
#define pb push_back
#define rep(x,a,b) for (int x=a;x<=b;x++)
#define repp(x,a,b) for (int x=a;x<b;x++)
#define W(x) printf("%d\n",x)
#define WW(x) printf("%lld\n",x)
#define pi 3.14159265358979323846
#define mem(a,x) memset(a,x,sizeof a)
#define lson rt<<1,l,mid
#define rson rt<<1|1,mid+1,r
using namespace std;
const int maxn=3e2+7;
const int INF=1e9;
const ll INFF=1e18;
int mapp[maxn][maxn];//二分图权值描述
int linker[maxn],lx[maxn],ly[maxn];//y中匹配状态，x，y中的点标号
int slack[maxn],n,nx,ny;//两边的点数
bool visx[maxn],visy[maxn];
bool dfs(int x)
{
    visx[x]=true;
    rep(y,1,ny)
```

```
    {
        if (visy[y])continue;
        int tmp=lx[x]+ly[y]-mapp[x][y];
        if (tmp==0)
        {
            visy[y]=true;
            if (linker[y]==-1||dfs(linker[y]))
            {
                linker[y]=x;
                return true;
            }
        }
        else if (slack[y]>tmp)slack[y]=tmp;
    }
    return false;
}
int KM()
{
    mem(linker,-1);
    mem(ly,0);
    rep(i,1,nx)
    {
        lx[i]=-INF;
        rep(j,1,ny)
        {
            if (mapp[i][j]>lx[i])lx[i]=mapp[i][j];
        }
    }
    rep(x,1,nx)
    {
        rep(i,1,ny)slack[i]=INF;
        while(1)
        {
            mem(visx,false);
            mem(visy,false);
            if (dfs(x))break;
            int d=INF;
            rep(i,1,ny)
            {
                if (!visy[i]&&d>slack[i])d=slack[i];
            }
            rep(i,1,nx)
            {
                if (visx[i])lx[i]-=d;
            }
            rep(i,1,ny)
            {
                if (visy[i])ly[i]+=d;
                else slack[i]-=d;
            }
        }
    }
    int res=0;
    rep(i,1,ny)
        if (linker[i]!=-1)
            res+=mapp[linker[i]][i];
    return res;
}
```

```
int main()
{
    //建图

    /*输出答案
    W(KM());
    */
    return 0;
}
```

```
#include<iostream>//二分图最大权匹配
#include<cstdio>
#include<queue>
#include<algorithm>
#include<cstring>
#define N 505
#define M 250005
#define INF 9990365505
#define ll long long
using namespace std;

int n,m,x,y,z,tot,tim;
int visx[N],visy[N];//visx[i]表示左侧第i个节点第几轮访问
int matchx[N],matchy[N];//matchy[i]表示左边第i个节点与右边第matchy[i]个节点匹配
ll ex[N],ey[N],slack[N];//ex表示左边节点值,ey表示右边节点的值,slack用于维护最小的的点权之和减去边权
int e[M],ne[M],h[M],w[M],idx=0,pre[N];//记录x的先驱节点

void add(int x,int y,int z){//链式前向星存边
    e[idx]=y,w[idx]=z;
    ne[idx]=h[x],h[x]=idx++;
}

void modify(int cur){//修改之前的 匹配方式
    for (int last,x=cur;x;x=last){
        last=matchx[pre[x]];
        matchx[pre[x]]=x;
        matchy[x]=pre[x];
    }
}
void bfs(int cur){
    for(int i=1;i<=n;i++)
        slack[i]=INF,pre[i]=0;//初始化
    queue<int>q;
    q.push(cur);
    ++tim;
    while(1){
        while (!q.empty()){//bfs交错树
            int u=q.front();
            q.pop();
            visx[u]=tim;//u是第tim轮被访问的
            for (int i=h[u];~i;i=ne[i]){

                int v=e[i],cost=w[i];//访问u的相邻节点
```

```
                if(visy[v]==tim)
                    continue;//本轮已经被访问过的不需要再次被访问

                ll mincost=ex[u]+ey[v]-cost;//记录

                if (mincost<slack[v]){//维护最小点权之和减边权
                    slack[v]=mincost;
                    pre[v]=u;//v的先驱节点记为u
                    if (!mincost){//mincost==0 则连边
                        visy[v]=tim;//这一轮也访问到了v
                        if (!matchy[v]){//左侧第v个节点没有与右侧节点匹配
                            modify(v);//修改之前的匹配方式,并终止
                            return;
                        }
                        else q.push(matchy[v]);//否则入队
                    }
                }
            }
        }
        ll mincost=INF;
        for(int i=1;i<=n;++i){
            if(visy[i]!=tim){ //本轮没有被访问过
                mincost=min(mincost,slack[i]);//在交错树的边中寻找顶标和与边权之差最小的边
            }
        }
        for(int i=1;i<=n;++i){
            if (visx[i]==tim)//左侧节点减去这个值
                ex[i]-=mincost;
            if (visy[i]==tim)
                ey[i]+=mincost;//右侧节点加这个值
            else
                slack[i]-=mincost;//维护更新
        }
        for(int i=1;i<=n;++i){
            if( visy[i]!=tim && !slack[i]){//发生冲突并解决冲突之后，继续匹配
                visy[i]=tim;//标记为本轮访问的
                if (!matchy[i]){//没有匹配过
                    modify(i);//修改
                    return;//返回
                }
                else q.push(matchy[i]);//否则入队
            }
        }
    }
}

void KM(){
    for(int i=1;i<=n;++i)
        bfs(i);//对每个点做一遍bfs
    //做完之后的匹配值就是左右两端节点数的总和
    ll ans=0;//
    for (int i=1;i<=n;++i)
        ans+=ex[i]+ey[i];
    printf("%lld\n",ans);
    for (int i=1;i<=n;++i)
        printf("%d ",matchy[i]);
    printf("\n");
}
```

```
int main(){
    memset(h,-1,sizeof h);
    scanf("%d%d",&n,&m);
    for (int i=1;i<=m;++i){//读入数据
        scanf("%d%d%d",&x,&y,&z);
        add(x,y,z);
        ex[x]=max(ex[x],(ll)z);
    }
    KM();//km算法求解
    return 0;
}
```

强连通分量SCC

- 性质
  - 将一个有向图转化为强连通分量所需要加的边的最小个数$min\{cnt_{出度为0},cnt_{入度为0}\}$
  - 强连通分量的逆序为拓扑排序
- 作用：将一个有向图缩点成有向无环图(DAG)

```
const int N=200010,M=N*2,mod=1e9+7;
int n,m,k,e[M],ne[M],h[N],idx=0,id[N];
int timestamp,low[N],dfn[N],stk[N],top,scc_cnt;
bool ins[N];

void clr(){
    timestamp=scc_cnt=0;
    rep(i,1,n) ins[i]=dfn[i]=low[i]=0;
}

void add(int a,int b){
    e[idx]=b,ne[idx]=h[a],h[a]=idx++;
}

void tarjan(int u){
    dfn[u]=low[u]=++timestamp;
    stk[++top]=u,ins[u]=1;
    for(int i=h[u];~i;i=ne[i]){
        int j=e[i];
        if(!dfn[j]){
            tarjan(j);
            low[u]=min(low[u], low[j]);
        }
        else if(ins[j]){
            low[u]=min(low[u], dfn[j]);
        }
    }
    if(low[u]==dfn[u]){
        int y;
        scc_cnt ++;
        do{
            y=stk[top--];
            ins[y]=0;
```

```
            id[y]=scc_cnt;
        }while(y!=u);
    }
}
```

双连通分量

E-DCC 边的双连通分量

> 桥：是一个无向边。对于一个无向连通图，如果删除某一条边会变得不连通，那么称这条边为桥

定义：极大的，不含有桥的连通区域被称为边的双连通分量

双连通分量性质：

[1] 删去任意一条边仍然是连通图

[2] 任意两点之间一定包含两条不相交的路径

[3] 将一个无向图转化为边的双连通分量最小需要加的边的个数是

$$[\frac{cnt+1}{2}]\ \text{其中}cnt\text{表示度为1的点的个数}$$

无向图中存在类似于有向图中的三种边：

```
[1]树枝边  (x,y)
[2]前向边  (a,b)
[3]后向边  (m,n)
```

E-DCC的缩点方法:

```
[1]类似于SCC，首先引入时间戳预处理出：
    dfn[x]:遍历到x节点的时间戳
    low[x]:x所能遍历到的最小的时间戳
[2]找到桥<=>找到dfn[x]<low[y]  //y在x下方，y无论如何也走不到x
[3]找到所有边的双连通分量有两种方法：
    1)将所有桥删除掉，剩下的每一个连通块都是一个连通分量
    2)类似于有向图，借助stack来判断dfn[x]==low[x]
```

找桥的方法：

```
void tarjan(int u,int from){
    dfn[u]=low[u]=++timestamp;

    for(int i=h[u];~i;i=ne[i]){
        int j=e[i];
        if(!dfn[j]){
            tarjan(j,i);  //from是反向边，此处为i
            low[u]=min(low[u],low[j]);

            if(dfn[u] < low[j]){    //如果满足桥的性质
                is_bridge[i]=is_bridge[i^1]=true;  //加边的时候是一偶一奇加的
            }
```

```
        }
        else if(i!=(from^1))  //如果不是反向边
            low[u]=min(low[u],dfn[j]);
    }
}
```

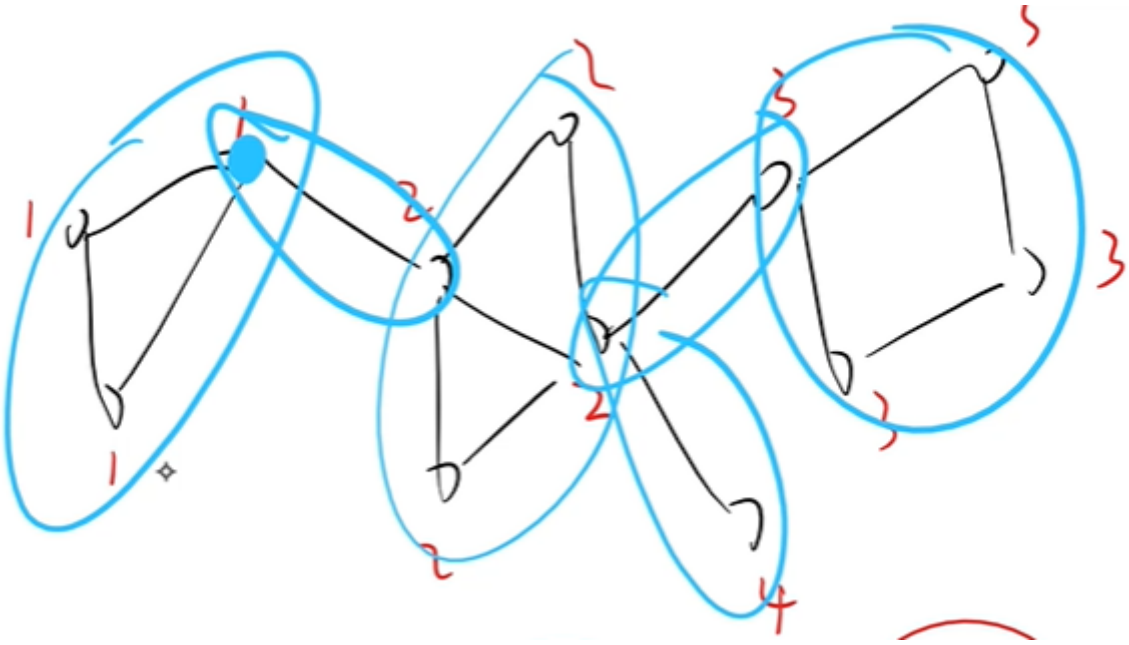
找到桥之后缩点

```
int scc_cnt, id[N];
void dfs(int u){
    id[u] = scc_cnt;
    for(int i=h[u];~i;i=ne[i]){
        int j=e[i];
        if(id[j] || bridge[i]) continue;
        dfs(j);
    }
}
int main(){
    rep(i,1,n) if(!id[i]) scc_cnt ++, dfs(i);
}
```

E-DCC的缩点：循环所有边

```
for(int i=0;i<idx;++i){
    int x = e[i], y = e[i^1];
    if(id[x] == id[y]) continue;
    add(id[x],id[y]);
}
```

V-DCC点的双连通分量



> 割点：在一个无向图中如果删去某一个点使得整个图变得不连通，则称此点为此无向图的割点

定义：极大的，不包含个点的连通块被称为点的双连通分量

性质：每一个割点至少属于两个连通分量

求割点：
[1] 满足low[y]>=dfn[x]后需要分类讨论
[2] 如果x不是根节点，那么x是割点
[3] 如果x是根节点，则其至少有两个子节点yi都满足low[yi]>=dfn[x],此时x才能算割点

求点的双连通分量思路：

[1] 记录时间戳，当前点入栈
[2] 特判,如果是孤立点就单独记录进对应连通块的数组
[3] 遍历所有邻边，并更新。如果没有更新过
当找到了dfn[x]<=dfn[y]之后要对其讨论是否是割点：
if(dfn[x]<=low[y]){
cnt++;//对于记录当前有多少个分支+1
if(x!=root||cnt>1) x是割点
将栈中元素弹出直至弹出y为止
将x也放入当前双连通分量中
}

V-DCC的缩点方式：

割点的判定：

```
void tarjan(int u){
    int cnt = 0;
    dfn[u] = low[u] = ++timestamp;

    for(int i=h[u];~i;i=ne[i]){
        int j = e[i];
        if(!dfn[j]){
            tarjan(j);
            low[u] = min(low[u], low[j]);

            if(dfn[u] <= low[j]) {
                cnt++;
                if(u!=root || cnt > 1)  cut[u] = true;
            }
        }
        else low[u] = min(low[u], dfn[j]);
    }
}
```

点的双连通分量求解

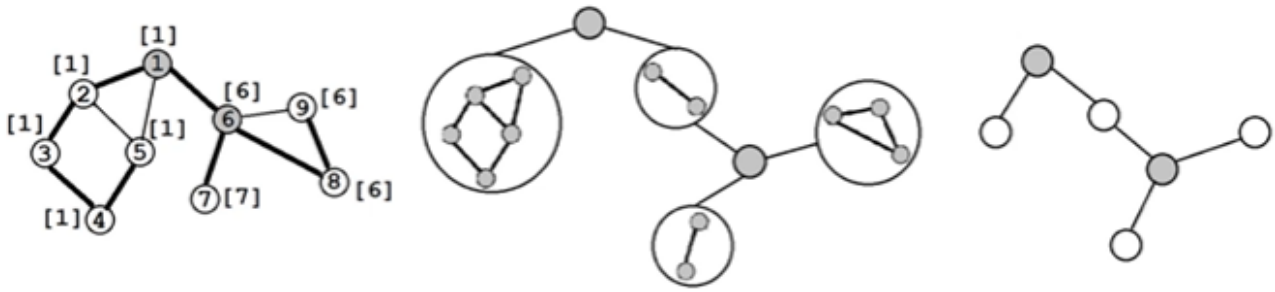
求V-DCC需要借助于栈来实现

```
void tarjan(int u){
    dfn[u] = low[u] = ++timestamp;
    stk[++ top] = u;

    if(u == root && h[u] == -1){ //孤立点
        dcc_cnt ++;
        dcc[dcc_cnt].push_back(u);
        return ;
    }
```

```
    int cnt = 0;
    for(int i=h[u];~i;i=ne[i]){
        int j = e[i];
        if(!dfn[j]){
            tarjan(j);
            low[u] = min(low[u], low[j]);

            if(dfn[u] <= low[j]) {
                cnt++;
                if(u!=root || cnt > 1)  cut[u] = true;
                dcc_cnt ++;
                int y;
                do{
                    y = stk[top --];
                    id[y] = dcc_cnt;
                    dcc[dcc_cnt].push_back(y);
                }while(y!=j);
                dcc[dcc_cnt].push_back(u);
            }
        }
        else low[u] = min(low[u], dfn[j]);
    }
}
```

注意当dfn[u] < dfn[j]的时候仍然要把u加入栈，因为此时x---y这个仍然是一个两个点的双连通分量

V-DCC缩点




```
int num=dcc_cnt;
for(int i=1;i<=dcc_cnt;++i) {
    for(auto u:dcc[i]){
        if(cut[u]) {
            if(!from[u]) from[u]=++num;
            edge[from[u]].push_back(i);
            edge[i].push_back(from[u]);
            d[i] ++, d[from[u]] ++;
        }
        else from[u]=i;
    }
}
```

例题：电力

> n个点m条边的无向图，求删除一个点后，连通块最多有多少即求一个割点使得删去此割点后的连通块的个数最多0

思路：先统计所有连通块的个数，然后枚举每一个连通块求割点的同时统计删除每个割点会形成多少个新的联通块
root=1 !root=2

```
//=================================
const int N = 20010, M = 30010;
int n, m, k, timestamp, top, idx, scc_cnt, ans, root;
int e[M], ne[M], h[N], stk[M], dfn[N], low[N], deg[N];
bool ins[M], is_bridge[M];

void add(int a,int b){
    e[idx] = b, ne[idx] = h[a], h[a] = idx ++;
}

void tarjan(int u){
    int cnt = 0;
    dfn[u] = low[u] = ++timestamp;

    for(int i=h[u];~i;i=ne[i]){
        int j = e[i];
        if(!dfn[j]){
            tarjan(j);
            low[u] = min(low[u], low[j]);

            if(dfn[u] <= low[j]) cnt++;
        }
        else low[u] = min(low[u], dfn[j]);
    }
    if(u!=root && cnt) cnt ++;
    ans = max(ans, cnt);
}
//=================================
int main(){
    n = read(), m = read();
    while(n || m){
        int cnt = 0;timestamp = ans = idx = 0;
        memset(h, -1, sizeof h);
        memset(dfn,0,sizeof dfn);
        memset(low,0,sizeof low);

        rep(i,1,m){
            int a = read(), b = read();
            add(a, b), add(b, a);
        }
        for(root=0;root<n;root++) if(!dfn[root])
            tarjan(root), cnt ++;

        print(ans+cnt-1);

        n = read(), m = read();
    }
    return 0;
}
```

欧拉路径与欧拉回路

大前提所有点都是连通的

欧拉路径：

- 是否存在一条路径每个边之走一遍

对于无向图：

- 欧拉路径（一笔画）问题的解决方案：所有奇数路径的点（奇点）的个数只能是0或者2个。如果是2个只能位于起点或者终点

对于有向图：

- 欧拉路径
  - [1]所有点的入度等于出度
  - [2]除了起点和终点之外的所有点的出度等于入度。起点的出度比入度多1，终点的入度比出度多1

欧拉回路：

- 是否存在一个环路，每个边恰好走一次之后又回到原来的地方

对于无向图

- ***欧拉回路的度数为奇数的点只能由0个***

对于有向图

- 所有的点出度等于入度

算法：

```
dfs(int u){
    for(int i=h[u];~i;i=ne[i]){
        int j=e[i];
        if(!st[j])
            dfs(i);
    }
    queue.push(u);
}
```

如果是有向图，则使用之后删去此边

如果是无向图，经过一条边之后应该标记反向边，标记方法&1（因为加入的时候是01，23..加入的）

板子：(优化)

```
void dfs(int u){
    for(int &i=h[u];~i;){
        if(st[i]){   //如果已经标记过来就直接删除
            i=ne[i];
            continue;
        }
        st[i]=true;  //标记当前边
        if(T==1) st[i^1]=true;  //如果是无向图就加双向边
```

```
        int t;
        //先算出要加的边
        if(T==1){
            t=i/2+1;
            if(i&1) t=-t;
        }
        else t=i+1;
        //先把当前边删除再遍历，这样能保证每个边只被遍历依次
        int j=e[i];
        i=ne[i];
        dfs(j);
        Q.push(t);

    }
}
```

或者

```
void dfs(int u)
{
    while(~h[u])
    {
        int i = h[u];
        if(used[i]) {
            h[u] = ne[i];
            continue;
        }

        h[u] = ne[i];
        used[i] = true;
        if(type == 1) used[i ^ 1] = true;

        dfs(e[i]);
        if(type == 1) {
            int t = i / 2 + 1;
            if(i & 1) t *= -1;
            ans[ ++ cnt] = t;
        }
        else ans[ ++ cnt] = i + 1;
    }
}
```

2-SAT

一般SAT问题 (NP问题)

> 给定n个命题$x_1, x_2 \dots x_n$
>
> 给出n个形式的条件，形如$x_1 \vee \neg x_2 \dots \vee x_n$ 表示如图形式中至少有一个为真
>
> 求给出$x_1, x_2 \dots x_n$的一组取值使得所有上述条件都为真。

做法：

首先将每个数x_i看作两个点x_1为真和$\neg x_i$ 为真，即拆成要么正变量成立要么取反后成立

然后根据数理逻辑的原理，有推论

$a \vee b$等价于$\neg b \rightarrow a$和$\neg a \rightarrow b$ (a或者b成立等价于a不成立时b必然成立且b不成立时a必然成立)。因此我们从$\neg a$向b连边add(2*a,2*b+1),从$\neg b$向a连边add(2*b+1,2*a)，这表征$\neg b$与a在同一逻辑域(同时成立)。按照此形式建立出来的图中，如果a和$\neg a$在同一个连通分量中，说明存在矛盾。否则，其中一可行解为：先对原图进行tarjan缩点，之后对于变量a和$\neg a$谁的拓扑序在更后面，我们就选取那一个变量对应的值。

> 但是，由于我们之前使用了tarjan缩点，根据结论tarjan缩点后每个点的便后就是每个点的拓扑序的逆序，因此又对于每个点
>
> ```
> if(id[i*2]<id[i*2+1]) cout << 0 << " ";
> else cout << 1 << " ";
> ```

板子没什么特别的，就是建图+tarjan,关键还是第一步

```
void tarjan(int u){
    dfn[u]=low[u]=++timestamp;
    ins[u]=true; stk[++top]=u;

    for(int i=h[u];~i;i=ne[i]){
        int j=e[i];
        if(!dfn[j]){
            tarjan(j);
            low[u]=min(low[u],low[j]);
        }
        else if(ins[j]) low[u]=min(low[u],dfn[j]);
    }
    if(low[u]==dfn[u]){
        int y;
        scc_cnt ++;
        do{
            y=stk[top --];
            ins[y]=false;
            id[y]=scc_cnt;
        }while(y!=u);
    }
}

void solve(){
    n=read(),m=read();
    rep(i,1,m){
        int x,a,y,b;
        x=read(),a=read(),y=read(),b=read();
        x--,y--;
        add(2*x+!a,2*y+b); add(2*y+!b,2*x+a);
    }

    rep(i,0,2*n-1) if(!dfn[i]) tarjan(i);

    bool ok=true;
    rep(i,0,n-1){
```

```
        if(id[i*2]==id[2*i+1]) {
            // debug(i);
            ok=false;
            break;
        }
    }
    if(!ok) puts("IMPOSSIBLE");
    else{
        puts("POSSIBLE");
        rep(i,0,n-1){
            if(id[2*i]<id[2*i+1]) printf("0 ");
            else printf("1 ");
        }
    }
}
```

Prufer编码

Prufer编码转树 O(n)

可以把一棵无根树变成一个序列，也可以将一个序列变成一棵无根树。

- 每次找到无根树编号最小的度数为1的点
- 然后将此点的父节点加入序列

O(n)时间求Prufer编码

对于一个有n个节点的树，其prufer编码只有n-2个数。

树转Prufer编码

一个数在prufer编码中出现多少次，就说明其有几个儿子

Step:

> void tree2prufer()的操作是顺序遍历，然后从1开始遍历，当找到出度为0的点时，将其父节点加入prufer序列，然后递归其父节点，如果其父节点的出度为0且父节点小于当亲遍历到的j的值，父节点的父节点也加入prufer序列。
>
> void pruffer2tree()的操作是顺序遍历prufer序列的同时遍历1~n，如果1~n遍历的时候有出度为0的点，则其父亲就是当前遍历到的prufer序列，然后prufer序列当前数的出度--，如果为0且小于当前遍历的树，则其父亲节点为下一个prufer序列中的树。

一道板子，如果m=1，给出给定树的prufer编码，否则给出给定prufer编码的树

```
const int N = 1e5+10;
int n,m;
int fa[N],p[N],d[N],idx=0; //d表示出度

void tree2prufer(){
    rep(i,1,n-1) fa[i]=read(),d[fa[i]]++;

    for(int i=0,j=1;i<n-2;j++){
        while(d[j]) j++;
        p[i++]=fa[j];
```

```
        while(i<n-2&&--d[p[i-1]]==0&&p[i-1]<j) p[i++]=fa[p[i-1]];
    }
    rep(i,0,n-3) printf("%d ",p[i]); puts("");
}

void prufer2tree(){
    rep(i,1,n-2) p[i]=read(),d[p[i]]++;
    p[n-1]=n;

    for(int i=1,j=1;i<n;++i,j++){
        while(d[j]) j++;
        fa[j]=p[i];
        while(i<n-1&&--d[p[i]]==0&&p[i]<j) fa[p[i]]=p[i+1],i++;
    }
    rep(i,1,n-1) printf("%d ",fa[i]);
}

void solve(){
    n=read(),m=read();
    if(m==1)
        tree2prufer();
    else prufer2tree();
}
```

应用：

Cayley定理

- 对于一个n个点的无向完全图，其生成树的个数为n^{n-2}

证明：由prufer编码一共n-2位，每一位有n个选择，因此为n^{n-2}

朱刘算法

有向图的类Prim算法，找有向图的最小生成树

最小树形图

树形图：

- 无有向环
- 除了根节点外，每个点入度为1

以某个点为根的一棵有向树，其边权之和为图中所有树形图中是最小的称为最小树形图。

朱刘算法 $O(nm)$

(1) 除了根节点外对每个点选取一条边权最小的入边

(2)判断当前(选出的边)组成的图中有无环

1.若无环：则说明当前图已经为构造好的最小生成树，算法结束

2.若有环：进行第(3)步

(3)将构造的图进行强连通分量缩点，得到新图G',对于G'中的所有边

1.如果是环中的边：直接删去

2.如果终点在环内(即新缩的点)：更新此边权权值为$W-W_{环内}$

3.其他边：不变

然后继续从(1)开始迭代

> 当迭代完成后,所有选择的边的边权之和就是最终的答案。

邻接矩阵版本：

> 由于复杂度是 O(nm)，因此在存储图的时候不需要背邻接表的板子，直接背邻接矩阵的即可。

板子：

```
const int N = 110,M=2e4+10,INF=1e8;
int n,m,r;
int d[N][N],bd[N][N],g[N][N];
int pre[N],bpre[N];
int dfn[N],low[N],timestamp,stk[N],top;
int id[N],scc_cnt;
bool st[N],ins[N];

void dfs(int u){
    st[u]=true;
    for(int i=1;i<=n;++i)
        if(d[u][i]<INF&&!st[i])
            dfs(i);
}

bool check_con(){
    memset(st,0,sizeof st);
    dfs(r);
    rep(i,1,n)
        if(!st[i])
            return false;
    return true;
}

void tarjan(int u){
    dfn[u]=low[u]=++timestamp;
    stk[++top]=u;ins[u]=true;

    int j=pre[u];
    if(!dfn[j]){
        tarjan(j);
        low[u]=min(low[u],low[j]);
    }
    else if(ins[j]) low[u]=min(low[u],dfn[j]);

    if(low[u]==dfn[u]){
        int y;
```

```
        scc_cnt ++;
        do{
            y=stk[top --];
            ins[y]=false;
            id[y]=scc_cnt;
        }while(y!=u);
    }
}

int zhuliu(){
    int ans = 0;

    while(true){
        for(int i=1;i<=n;++i){
            pre[i]=i;
            for(int j=1;j<=n;++j)
                if(d[pre[i]][i] > d[j][i])
                    pre[i]=j;
        }

        memset(dfn,0,sizeof dfn);
        timestamp=scc_cnt=0;
        for(int i=1;i<=n;++i)
            if(!dfn[i])
                tarjan(i);


        if(scc_cnt == n){
            for(int i=1;i<=n;++i) {
                if(i==r) continue;
                ans += d[pre[i]][i];
            }
            break ;
        }

        for(int i=1;i<=n;++i){
            if(i==r) continue;
            if(id[pre[i]] == id[i]){
                ans += d[pre[i]][i];
            }
        }

        for(int i=1;i<=scc_cnt;++i)
            for(int j=1;j<=scc_cnt;++j)
                bd[i][j]=INF;

        for(int i=1;i<=n;++i){
            for(int j=1;j<=n;++j){
                if(d[i][j] < INF && id[i]!=id[j]){
                    int a=id[i],b=id[j];
                    if(id[pre[j]]==id[j])
                        bd[a][b]=min(bd[a][b],d[i][j]-d[pre[j]][j]);
                    else bd[a][b]=min(bd[a][b],d[i][j]);
                }
            }
        }
        r=id[r];
        n = scc_cnt;
```

```
        memcpy(d,bd,sizeof d);
    }
    return ans;
}

void solve(){
    n=read(),m=read(),r=read();
    rep(i,1,m) {
        int u=read(),v=read(),w=read();
        d[u][v]=min(d[u][v],w);
    }
    // rep(i,1,n) rep(j,1,n) debug(d[i][j]);
    if(!check_con()) puts("-1");
    else print(zhuliu());
}
```

模板题

在一个二维平面中求 一个最小树形图

```
const int N = 110, M = 10010, INF=1e8;
int n,m,k;
pdd q[N];
bool g[N][N];
double d[N][N],bd[N][N]; //d数组用来存距离，bd数组用来存储备份
int pre[N],bkppre[N];  //pre数组用来存储备份，bkppre数组用来存储前去的备份
int dfn[N],low[N],timestamp,stk[N],top;
int id[N],scc_cnt=0;
bool st[N],ins[N];

void dfs(int u){
    st[u]=true;
    rep(i,1,n) if(g[u][i]&&!st[i]) dfs(i);
}

bool check_con(){
    memset(st,0,sizeof st);
    dfs(1);
    rep(i,1,n) if(!st[i]) return false;
    return true;
}

double get_dist(int a,int b){
    double dx = q[a].x-q[b].x;
    double dy = q[a].y-q[b].y;
    return sqrt(dx*dx+dy*dy);
}

void tarjan(int u){
    dfn[u]=low[u]=++timestamp;
    stk[++top]=u,ins[u]=true;

    int j=pre[u];
    if(!dfn[j]){
        tarjan(j);
```

```
        low[u]=min(low[u],low[j]);
    }
    else if(ins[j]) low[u]=min(low[u],dfn[j]);

    if(low[u]==dfn[u]){
        int y;
        scc_cnt ++;
        do{
            y=stk[top --];
            ins[y]=false;
            id[y]=scc_cnt;
        }while(y!=u);
    }
}

double zhuliu(){
    double ans= 0;
    for(int i=1;i<=n;++i)
        for(int j=1;j<=n;++j)
            if(g[i][j]) d[i][j] = get_dist(i,j);
            else d[i][j] = INF;

    while(true) {
        //找所有点的入点的最短边
        for(int i=1;i<=n;++i){
            pre[i]=i;
            for(int j=1;j<=n;++j){
                if(d[pre[i]][i] > d[j][i])
                    pre[i]=j;
            }
        }

        //tarjan找环
        memset(dfn,0,sizeof dfn);
        timestamp=scc_cnt=0;
        for(int i=1;i<=n;++i)
            if(!dfn[i])
                tarjan(i);

        //缩点后无环,累加答案后算法结束
        if(scc_cnt == n) {
            for(int i=2;i<=n;++i) ans += d[pre[i]][i];
            break;
        }

        //累加所有环内的边
        for(int i=2;i<=n;++i)
            if(id[pre[i]]==id[i])
                ans += d[pre[i]][i];

        //清空bd数组,准备存储缩点完更新的边权
        for(int i=1;i<=scc_cnt;++i)
            for(int j=1;j<=scc_cnt;j++)
                bd[i][j] = INF;

        //遍历每一个点，然后根据缩点后的结果对边进行操作
        for(int i=1;i<=n;++i)
            for(int j=1;j<=n;++j)
```

```
                if(d[i][j] < INF && id[i] != id[j]){
                    int a=id[i],b=id[j];
                    if(id[pre[j]] == id[j])
                        bd[a][b]=min(bd[a][b],d[i][j]-d[pre[j]][j]);
                    else bd[a][b] = min(bd[a][b], d[i][j]);
                }
        n = scc_cnt;
        memcpy(d,bd,sizeof d);
    }

    return ans;
}

void solve(){
    rep(i,1,n) scanf("%lf%lf",&q[i].x,&q[i].y);

    memset(g,0,sizeof g);
    while(m--){
        int a=read(),b=read();
        if(a!=b&&b!=1) g[a][b]=true;
    }
    if(!check_con()) puts("poor snoopy");
    else printf("%.2lf\n",zhuliu());
}
//================================
int main(){
    while(~scanf("%d%d",&n,&m)){
        solve();
    }
    return 0;
}
```