0.1 图的存储

0.1.1 链式前向星

```
struct road{int b,c,nex;}r[2000005];
void add(int a,int b,int c)
{
    r[num].b=b;r[num].c=c;
    r[num].nex=head[a];head[a]=num++;
}
```

0.2 树

0.2.1 树的直径

求法：

1.两次DFS，第一次从任意点出发，找到距离最远的点x。第二次从x出发，找到的最远点y。此方法不适用于有负权边的树。

2.树形DP，求出x子树中最深的深度和次深的深度，最大的和即为直径。

性质：

1.从树上任一点出发，在树上的最长路径距离一定是到直径两端点中的一个。

2.若树上所有边边权均为正，则树的所有直径中点重合。

方法1

```
void dfs(int u,int fa)
{
    for(int i=0;i<v[u].size();i++)
    {
        int nex=v[u][i];
        if(nex==fa) continue;
        dep[nex]=dep[u]+1;
        if(dep[nex]>dep[t]) t=nex;
        dfs(nex,u);
    }
}
```

方法2

```cpp
void dfs(int u,int fa)
{
    d1[u]=d2[u]=0;
    for(auto nex:v[u])
    {
        if(nex==fa) continue;
        dfs(nex,u);
        int t=d1[nex]+1;
        if(t>d1[u]) d2[u]=d1[u],d1[u]=t;
        else if(t>d2[u]) d2[u]=t;
    }
    d=max(d,d1[u]+d2[u]);
}
```

0.2.2 树的重心

将树的某点删掉后，会形成有多棵树的森林，可以使森林中最大的树最小的点就是树的重心。

性质：

1.以树的重心为根时，所有子树的大小都不超过整棵树大小的一半。

2.树中所有点到某个点的距离和中，到重心的距离和是最小的；如果有两个重心，那么到它们的距离和一样。

3.树中所有点到某个点的距离和中，到重心的距离和是最小的；如果有两个重心，那么到它们的距离和一样。

4.在一棵树上添加或删除一个叶子，那么它的重心最多只移动一条边的距离。

5.一棵树最多有两个重心，且相邻。

求法：

DFS计算每个子树的大小，记录“向下”的子树的最大大小，利用总点数-当前子树（这里的子树指有根树的子树）的大小得到“向上”的子树的大小，然后就可以依据定义找到重心了。

```cpp
void dfs(int u,int fa)
{
    sz[u]=1;
    for(int i=0;i<v[u].size();i++)
    {
        int nex=v[u][i];
        if(nex==fa) continue;
        dfs(nex,u);
```

```
        sz[u]+=sz[nex];
        w[u]=max(w[u],sz[nex]);
    }
    w[u]=max(w[u],n-sz[u]);
}
```

带权树重心

树上点具有点权，重心到所有点的距离乘各点点权的乘积之和最小。

```
//w[i]代表i点的权值，f[i]代表以i点为跟产生的重量大小
void dfs(int u,int fa)
{
    sz[u]+=w[u];
    for(int i=0;i<v[u].size();i++)
    {
        int nex=v[u][i];
        if(nex==fa) continue;
        dep[nex]=dep[u]+1;
        dfs(nex,u);
        sz[u]+=sz[nex];
    }
    f[1]+=w[u]*dep[u];
}
void dfs2(int u,int fa)
{
    for(int i=0;i<v[u].size();i++)
    {
        int nex=v[u][i];
        if(nex==fa) continue;
        f[nex]=f[u]+sz[1]-sz[nex]*2;
        dfs2(nex,u);
    }
}
```

0.2.3 最近公共祖先

```
void dfs(int u,int fa)//fa=0
{
    f[u][0]=fa;dep[u]=dep[fa]+1;
    for(int i=1;i<=20;i++) f[u][i]=f[f[u][i-1]][i-1];
    for(int i=head[u];~i;i=r[i].nex)
```

```
    {
        int nx=r[i].b;
        if(nx==fa) continue;
        dfs(nx,u);
    }
}
int lca(int a,int b)
{
    if(dep[a]<dep[b]) swap(a,b);
    for(int i=20;i>=0;i--)
    {
        if(dep[f[a][i]]>=dep[b]) a=f[a][i];
    }
    if(a==b) return a;
    for(int i=20;i>=0;i--)
    {
        if(f[a][i]!=f[b][i]) a=f[a][i],b=f[b][i];
    }
    return f[a][0];
}
```

0.2.4 树上启发式合并

时间复杂度证明：

一个有n个结点的树，根节点到任意一点经过的轻边数量小于$\log n$ 条。一个节点的被遍历的次数等于它到根节点路径上的轻边数+1，所以一个节点的被遍历次数$\log n+1$，总时间复杂度则为$O(n(\log n+1))=O(n\log n)$。

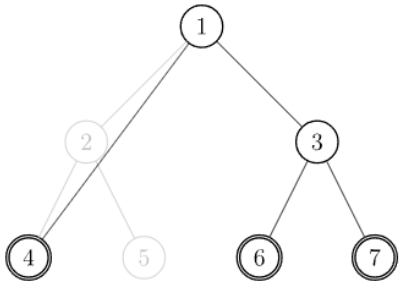
树上启发式合并求子树中颜色种类

```
void add(int u){if(vis[col[u]]==0) num++;vis[col[u]]++;}
void del(int u){if(vis[col[u]]==1) num--;vis[col[u]]--;}
void dfs(int u,int fa)
{
    sz[u]=1;L[u]=++dfn;pos[dfn]=u;R[u]=dfn;
    int mx=0;
    for(auto nex:v[u])
    {
        if(nex==fa) continue;
        dfs(nex,u);
        sz[u]+=sz[nex];R[u]=max(R[u],R[nex]);
        if(sz[nex]>mx) mx=sz[nex],big[u]=nex;
```

```
    }
}
void dfs2(int u,int fa,int lable)
{
    for(auto nex:v[u])
    {
        if(nex==fa || nex==big[u]) continue;
        dfs2(nex,u,0);
    }
    if(big[u]) dfs2(big[u],u,1);
    add(u);
    for(auto nex:v[u])
    {
        if(nex==fa || nex==big[u]) continue;
        for(int i=L[nex];i<=R[nex];i++) add(pos[i]);
    }
    ans[u]=num;
    if(lable==0)
    {
        for(int i=L[u];i<=R[u];i++) del(pos[i]);
    }
}
```

0.2.5 虚树

[4, 6, 7]三点构成的虚树如图。使用虚树可以缩小树形DP遍历的范围。




```
bool cmp(int a,int b){return dfn[a]<dfn[b];};
void dfs(int u,int fa)
{
    dfn[u]=++tim;f[u][0]=fa;dep[u]=dep[fa]+1;
    for(int i=1;i<20;i++) f[u][i]=f[f[u][i-1]][i-1];
    for(int i=head[u];~i;i=r[i].nex)
    {
```

```
8           int nex=r[i].b;
9           if(nex==fa) continue;
10          dfs(nex,u);
11      }
12  }
13  int lca(int a,int b)
14  {
15      if(dep[a]<dep[b]) swap(a,b);
16      for(int i=19;i>=0;i--)
17      {
18          if(dep[f[a][i]]>=dep[b])
19          a=f[a][i];
20      }
21      if(a==b) return a;
22      for(int i=19;i>=0;i--)
23      {
24          if(f[a][i]!=f[b][i])
25          a=f[a][i],b=f[b][i];
26      }
27      return f[a][0];
28  }
29  void build_Virtual_Tree(int rt)
30  {
31      sort(h.begin(),h.end(),cmp);
32      //切记不要拿出来清空，警钟敲烂
33      sta[top=1]=rt;vtree[rt].clear();
34      for(int i=0;i<h.size();i++)
35      {
36          if(h[i]==rt) continue;
37          int l=lca(sta[top],h[i]);
38          //LCA与栈顶不同，说明当前节点与栈中节点不在同一条树链上
39          if(l!=sta[top])
40          {
41              //找到栈中与当前LCA在同一树链上的位置
42              while(dfn[sta[top-1]]>dfn[l])
43              {
44                  vtree[sta[top-1]].push_back(sta[top]);
45                  top--;
46              }
47              //这个位置不是LCA,将LCA入栈
48              if(dfn[l]>dfn[sta[top-1]])
49              {
```

```
                vtree[l].clear();
                vtree[l].push_back(sta[top]);sta[top]=l;
            }
            else //LCA就在栈中
            {
                vtree[l].push_back(sta[top--]);
            }
        }
        sta[++top]=h[i];vtree[h[i]].clear();
    }
    for(int i=1;i<top;i++)
    {
        vtree[sta[i]].push_back(sta[i+1]);
    }
}
```

0.2.6 树分治

点分治

求出树上两点距离小于等于k的点对数量

```
#include<bits/stdc++.h>
using namespace std;
const int maxn=4e4+7;
int num,head[maxn],n,m,sz[maxn],w[maxn],rt,vis[maxn],dis[
    maxn],sum,k;
struct road{int b,c,nex;}r[80005];
void add(int a,int b,int c)
{
    r[num].b=b;r[num].c=c;r[num].nex=head[a];head[a]=num++;
}
void getrt(int u,int fa)//找u子树中的重心
{
    sz[u]=1;w[u]=0;
    for(int i=head[u];~i;i=r[i].nex)
    {
        int nex=r[i].b;
        if(nex==fa||vis[nex]) continue;
        getrt(nex,u);
        sz[u]+=sz[nex];
        w[u]=max(w[u],sz[nex]);
```

```
20      }
21      w[u]=max(w[u],sum-sz[u]);
22      if(w[u]<w[rt]) rt=u;
23  }
24  int ans,cnt,a[maxn],b[maxn];
25  void dfs(int u,int fa,int dis,int from)
26  {
27      b[u]=dis;a[++cnt]=b[u];
28      for(int i=head[u];~i;i=r[i].nex)
29      {
30          int nex=r[i].b;
31          if(nex==fa||vis[nex]) continue;
32          dfs(nex,u,dis+r[i].c,from);
33      }
34  }
35  int caldis(int u,int dis)
36  {
37      cnt=0;b[u]=dis;a[++cnt]=b[u];
38      for(int i=head[u];~i;i=r[i].nex)
39      {
40          int nex=r[i].b;
41          if(vis[nex]) continue;
42          dfs(nex,u,r[i].c+dis,nex);
43      }
44      int res=0;
45      sort(a+1,a+1+cnt);
46      for(int i=1,j=cnt;i<j;i++)
47      {
48          while((j-1)>i&&a[j]+a[i]>k) j--;
49          if(j>i&&a[j]+a[i]<=k) res+=(j-i);
50      }
51      return res;
52  }
53  void solve(int u)
54  {
55      vis[u]=1;ans+=caldis(u,0);
56      for(int i=head[u];~i;i=r[i].nex)
57      {
58          int nex=r[i].b;
59          if(vis[nex]) continue;
60          ans-=caldis(nex,r[i].c);
61          dis[nex]=r[i].c;
```

```
62          rt=0;sum=sz[nex];
63          getrt(nex,u);solve(rt);
64      }
65  }
66  int main()
67  {
68      memset(head,-1,sizeof(head));
69      scanf("%d",&n);
70      for(int i=1;i<n;i++)
71      {
72          int a,b,c;scanf("%d%d%d",&a,&b,&c);
73          add(a,b,c);add(b,a,c);
74      }
75      scanf("%d",&k);
76      w[rt]=1e9;sum=n;getrt(1,0);solve(rt);
77      printf("%d\n",ans);
78  }
```

0.3 生成树

0.3.1 次小生成树

最小生成树转变为次小生成树，只需要插入一条新边，删去一条原边即可。枚举可以插入的新边，在新边LCA的路径上找到可以删去的最长的边，可以用倍增实现。

```
1  #include<bits/stdc++.h>
2  #define int long long
3  #define pb push_back
4  using namespace std;
5  const int maxn=3e5+7;
6  struct road{int x,y,c;}r[maxn];
7  bool cmp(road a,road b){return a.c<b.c;}
8  int n,m,vis[maxn],fa[maxn];
9  int find(int x){return fa[x]==x?x:fa[x]=find(fa[x])}
10 vector<pair<int,int>>v[maxn];
11 int f[maxn][20],mx1[maxn][20],mx2[maxn][20],depth[maxn];
12 void dfs(int u,int fa)
13 {
14     depth[u]=depth[fa]+1;
15     f[u][0]=fa;
```

```
16      for(int i=0;i<v[u].size();i++)
17      {
18          auto nex=v[u][i];
19          if(nex.first==fa) continue;
20          mx1[nex.first][0]=nex.second;
21          mx2[nex.first][0]=-1e18;
22          dfs(nex.first,u);
23      }
24  }
25  void ST()
26  {
27      for(int i=1;i<20;i++)
28      {
29          for(int j=1;j<=n;j++)
30          {
31              f[j][i]=f[f[j][i-1]][i-1];
32              mx1[j][i]=max(mx1[j][i-1],mx1[f[j][i-1]][i-1]);
33              mx2[j][i]=max(mx2[j][i-1],mx2[f[j][i-1]][i-1]);
34              if(mx1[j][i-1]!=mx1[f[j][i-1]][i-1])
35              mx2[j][i]=max(mx2[j][i],min(mx1[j][i-1],mx1[f[j
                    ][i-1]][i-1]));
36          }
37      }
38  }
39  int LCA(int a,int b)
40  {
41      if(depth[a]<depth[b]) swap(a,b);
42      for(int i=19;i>=0;i--)
43      {
44          if(depth[f[a][i]]>=depth[b]) a=f[a][i];
45      }
46      if(a==b) return a;
47      for(int i=19;i>=0;i--)
48      {
49          if(f[a][i]!=f[b][i]) a=f[a][i],b=f[b][i];
50      }
51      return f[a][0];
52  }
53  int findmx(int x,int pa,int val)
54  {
55      int len=-1e18;
56      for(int i=19;i>=0;i--)
```

```
57      {
58          if(depth[f[x][i]]<depth[pa]) continue;
59          if(mx1[x][i]!=val) len=max(len,mx1[x][i]);
60          else len=max(len,mx2[x][i]);
61          x=f[x][i];
62      }
63      return len;
64  }
65  signed main()
66  {
67      scanf("%lld%lld",&n,&m);
68      for(int i=1;i<=n;i++) fa[i]=i;
69      for(int i=1;i<=m;i++)
70      {
71          scanf("%lld%lld%lld",&r[i].x,&r[i].y,&r[i].c);
72      }
73      sort(r+1,r+1+m,cmp);
74      int sum=0;
75      for(int i=1;i<=m;i++)
76      {
77          int px=find(r[i].x),py=find(r[i].y);
78          if(px==py) continue;
79          vis[i]=1;fa[px]=py;sum+=r[i].c;
80          v[r[i].x].pb({r[i].y,r[i].c});v[r[i].y].pb({r[i].x,r
                [i].c});
81      }
82      dfs(1,0);
83      ST();
84      int ans=1e18;
85      for(int i=1;i<=m;i++)
86      {
87          if(vis[i]) continue;
88          int lca=LCA(r[i].x,r[i].y);
89          int lmx=findmx(r[i].x,lca,r[i].c),rmx=findmx(r[i].y,
                lca,r[i].c);
90          ans=min(ans,sum-max(lmx,rmx)+r[i].c);
91      }
92      printf("%lld\n",ans);
93  }
```

0.3.2 瓶颈生成树

定义：

无向图G的瓶颈生成树是这样的一个生成树，它的最大的边权值在G的所有生成树中最小。

性质：

最小生成树是瓶颈生成树的充分不必要条件。即最小生成树一定是瓶颈生成树，而瓶颈生成树不一定是最小生成树。

最小瓶颈路

无向图G中x到y的最小瓶颈路是这样的一类简单路径，满足这条路径上的最大的边权在所有x到y的简单路径中是最小的。

性质：

x到y的最小瓶颈路上的最大边权等于最小生成树上x到y路径上的最大边权。

0.3.3 Kruskal重构树

构造方法：

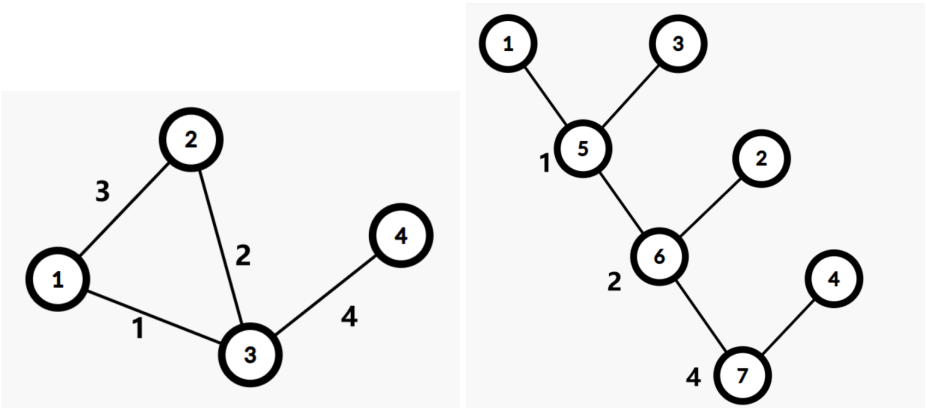
在跑Kruskal的过程中我们会从小到大加入若干条边,现在我们仍然按照这个顺序。每一次加边会合并两个集合，我们可以新建一个点，点权为加入边的边权，同时将两个集合的根节点分别设为新建点的左儿子和右儿子。然后我们将两个集合和新建点合并成一个集合,将新建点设为根。

不难发现，在进行$n-1$轮之后我们得到了一棵恰有$2n-1$个节点的二叉树，同时每个非叶子节点恰好有两个儿子。这棵树就叫Kruskal重构树。

性质：

两点的LCA的点权为原图中最大值最小的路径上的最大值（最小瓶颈路）。

是一颗二叉树,任意点的权值大于左右儿子的权值，是一个大根堆。



左为原图，右为该图的Kruskal重构树

```
void Kruskal()
{
    sort(r+1,r+1+m,cmp);
    for(int i=1;i<=m;i++)
    {
        int px=find(r[i].x),py=find(r[i].y);
        if(px==py) continue;
        fa[px]=fa[py]=++idx;fa[idx]=idx;
        val[idx]=r[i].c;
        add(idx,px);add(idx,py);
    }
}
```

0.4 最小树形图

有向图上的最小生成树称为最小树形图。

0.4.1 朱刘算法

求DAG上最小树形图：

对于一个DAG，只要我们对于每一个点选出最小的入边，那么这一定是个树形图。

求环上最小树形图：

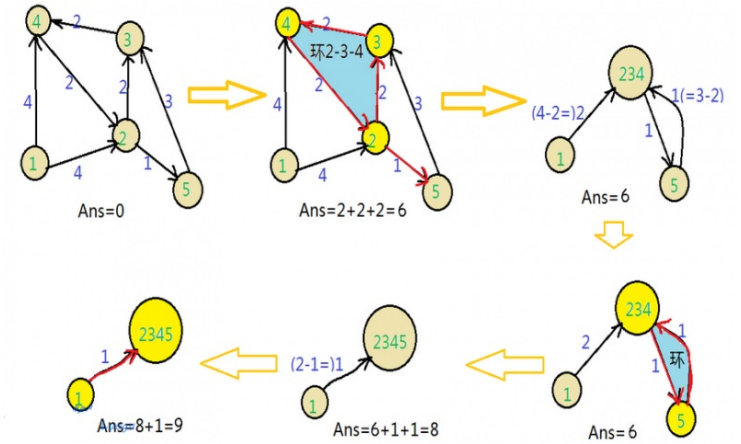
只需要将环上最长的一条边去掉即可。

求有向图最小树形图：

对图上所有点直接选取其最小的入边，判断这些边是组成环。

若无环，说明已经找到了最小树形图，直接结束即可。

若有环，肯定要把其中一条边换成环外边。我们就把贪心算出来的那个所谓的“树形图”上的环**缩成一个点**，环外边指向这个所称的点，为了方便统计，由于我们确信环外边的长度≥环内边的长度，那么我们ans先加上这个环的边权和，然后指向环的边边权设为：**自己的长度−所连向的环内点在环中指向的点的边的长度**。



这样我们一直加边，找环，缩点循环，直到找到的是一个DAG。至此结束，时间复杂度$O(nm)$。

```
int Edmonds(int root)
{
    int ans=0,cnt=0;
    while(1)
    {   //i点的父亲是fa[i]，祖先是top[i]，属于第loop[i]个环，其选择的入边权值为mn[i]
        for(int i=1;i<=n;i++) fa[i]=top[i]=loop[i]=0,mn[i]=1e9;
        for(int i=1;i<=m;i++) //找每个点最小的入边
        {
            int u=e[i].u,v=e[i].v,w=e[i].w;
            if(u!=v&&w<mn[v]) mn[v]=w,fa[v]=u;
        }
        mn[root]=0;
        for(int i=1;i<=n;i++)
        {
            if(mn[i]==1e9) return -1; //有点无边连接，无解
            ans+=mn[i];
        }
        for(int i=1,j=1;i<=n;i++,j=i)
        {
```

```
20              while(j!=root&&top[j]!=i&&!loop[j]) top[j]=i,j=
                    fa[j];
21              if(j!=root&&!loop[j]) //找到环，标号
22              {
23                  loop[j]=++cnt;
24                  for(int k=fa[j];k!=j;k=fa[k]) loop[k]=cnt;
25              }
26          }
27          if(!cnt) return ans; //如果无环，结束
28          for(int i=1;i<=n;i++) if(!loop[i]) loop[i]=++cnt;//将
                不在环内的点设置为独立环
29          for(int i=1;i<=m;i++) //缩点并且重新设置新边权
30              e[i].w-=mn[e[i].v],e[i].u=loop[e[i].u],e[i].v=
                    loop[e[i].v];
31          n=cnt;root=loop[root];cnt=0; //初始化。注意会更改n的值!!!!
32      }
33  }
```

0.5 斯坦纳树

最小斯坦纳树

给定连通图G中的n个点与k个关键点，连接k个关键点，使得生成树的所有边的权值和最小。

我们使用状态压缩动态规划来求解。用$f(i,S)$表示以i为根的一棵树，包含集合S中所有点的最小边权值和。

1. 对于i的度数为1的情况，可以考虑枚举树上与i相邻的点j，则：$f(i,S)=f(j,S)+w(j,i)$。
2. 对于i的度数大于1的情况，可以划分成几个子树考虑，即：$f(i,S)=f(i,T)+f(i,S-T)$。

时间复杂度为$O(n\times 3^k+m\log m\times 2^k)$。

```
1   #include<bits/stdc++.h>
2   using namespace std;
3   const int maxn=105;
4   typedef pair<int,int> PII;
5   int head[maxn],num,f[maxn][(1<<10)+5],key[10],vis[maxn];
6   struct road{int b,c,nex;}r[10005];
7   void add(int a,int b,int c)
```

```
8  {r[num].b=b;r[num].c=c;r[num].nex=head[a];head[a]=num++;}
9  priority_queue<PII,vector<PII>,greater<PII>>q;
10 void dijkstra(int s)
11 {
12     memset(vis,0,sizeof(vis));
13     while(!q.empty())
14     {
15         auto [val,u]=q.top();q.pop();
16         if(vis[u]) continue;
17         vis[u]=1;
18         for(int i=head[u];~i;i=r[i].nex)
19         {
20             int v=r[i].b;
21             if(f[v][s]>f[u][s]+r[i].c)
22             {
23                 f[v][s]=f[u][s]+r[i].c;
24                 q.push({f[v][s],v});
25             }
26         }
27     }
28 }
29 int main()
30 {
31     memset(f,0x3f,sizeof(f));
32     memset(head,-1,sizeof(head));
33     int n,m,k;scanf("%d%d%d",&n,&m,&k);
34     for(int i=1;i<=m;i++)
35     {
36         int a,b,c;scanf("%d%d%d",&a,&b,&c);
37         add(a,b,c);add(b,a,c);
38     }
39     for(int i=1;i<=k;i++)
40     {
41         scanf("%d",&key[i]);f[key[i]][1<<(i-1)]=0;
42     }
43     for(int i=1;i<(1<<k);i++)
44     {
45         for(int j=1;j<=n;j++)
46         {
47             for(int sub=i&(i-1);sub;sub=i&(sub-1))
48             {
49                 f[j][i]=min(f[j][i],f[j][sub]+f[j][i^sub]);
```

```
50              }
51              if(f[j][i]!=0x3f3f3f3f) q.push({f[j][i],j});
52          }
53          dijkstra(i);
54      }
55      printf("%d\n",f[key[1]][(1<<k)-1]);
56  }
```

0.6 基环树

如果一张无向连通图包含恰好一个环，则称它是一棵基环树。

如果一张有向弱连通图每个点的入度都为1，则称它是一棵基环外向树。

如果一张有向弱连通图每个点的出度都为1，则称它是一棵基环内向树。

1. 先处理环上每个点支出来的子树部分，然后把贡献记录在环上的点，最后问题就成了计算一个环的答案了。
2. 先删去环上的一条边，让基环树变成真正的树计算后考虑加上之前删去那条边的影响即可（使用并查集找到环上的两个点，以两个点为树根分别树形DP）。

基环树直径

基环树的直径显然有下面两种可能：1.在“根节点”的某一棵子树中。2.经过“根节点”，在“根节点”的某两棵子树中。

先将基环树中的环剔出来，对于环上的每棵树，使用动态规划法求出树的直径，即可解决1的可能。对于2，我们需要找到环上的两个节点i,j，使得$d_i+d_j+dist_{i,j}$最大。这需要我们将长度为n的环拓展为长度为$2*n$的链，使用单调队列维护最大的$d_j+dist_{i,j}$即可。

IOI2008-Island

```
1  #include<bits/stdc++.h>
2  #define int long long
3  using namespace std;
4  const int maxn=1e6+7;
5  int head[maxn],num,n,vis[maxn],loop[2*maxn],cnt,lable[maxn],
       s[2*maxn];
6  struct road{int b,c,nex;}r[2000005];
```

```
7  void add(int a,int b,int c)
8  {
9      r[num].b=b;r[num].c=c;r[num].nex=head[a];head[a]=num++;
10 }
11 bool dfs(int u,int f)
12 {
13     if(vis[u]==1)
14     {
15         vis[u]=2,loop[++cnt]=u,lable[u]=1;
16         return 1;
17     }//找到衔接点
18     vis[u]=1; //维护访问数组
19     for(int i=head[u];~i;i=r[i].nex)
20     {
21         if(i!=(f^1)&&dfs(r[i].b,i))//如果当前边不是上一条边并且当
               前节点在环上
22         {
23             if(vis[u]!=2)//当前节点不是衔接点
24             {
25                 loop[++cnt]=u,lable[u]=1,s[cnt]=r[i].c;
26                 return 1;
27             }
28             else//是衔接点
29             {
30                 s[1]=r[i].c;
31                 return 0;
32             }
33         }
34     }
35     return 0;
36 }
37 int d1[maxn],d2[maxn],d;
38 void dfs2(int u,int fa)
39 {
40     d1[u]=d2[u]=0;vis[u]=1;
41     for(int i=head[u];~i;i=r[i].nex)
42     {
43         int nex=r[i].b;
44         if(nex==fa||lable[nex]) continue;
45         dfs2(nex,u);
46         int t=d1[nex]+r[i].c;
47         if(t>d1[u]) d2[u]=d1[u],d1[u]=t;
48         else if(t>d2[u]) d2[u]=t;
```

```
49      }
50      d=max(d,d1[u]+d2[u]);
51  }
52  signed main()
53  {
54      memset(head,-1,sizeof(head));
55      scanf("%lld",&n);
56      for(int i=1;i<=n;i++)
57      {
58          int a,b;scanf("%lld%lld",&a,&b);
59          assert(a!=i);
60          add(i,a,b);add(a,i,b);
61      }
62      int ans=0;
63      for(int i=1;i<=n;i++)
64      {
65          if(vis[i]) continue;
66          cnt=0;dfs(i,-1);
67          int sum=0;
68          for(int j=1;j<=cnt;j++)
69          {
70              d=0;dfs2(loop[j],-1);
71              sum=max(sum,d);
72              s[j+cnt]=s[j];loop[j+cnt]=loop[j];
73          }
74          s[1]=0;
75          for(int j=2;j<=2*cnt;j++) s[j]=s[j-1]+s[j];
76          deque<int>q;
77          for(int j=1;j<=2*cnt;j++)
78          {
79              while(!q.empty()&&q.front()<=j-cnt+1) q.
                    pop_front();
80              while(!q.empty()&&s[q.back()]+d1[loop[q.back()
                    ]]<=s[j]+d1[loop[j]])
81              q.pop_back();
82              q.push_back(j);
83              if(j>=cnt)
84              {
85                  int u=q.front();
86                  sum=max(sum,s[u]-s[j-cnt+1]+d1[loop[u]]+d1[
                        loop[j-cnt+1]]);
87              }
```

```
88          }
89          ans+=sum;
90      }
91      printf("%lld\n",ans);
92  }
```

0.7 搜索

0.7.1 双向搜索

双向同时搜索

双向同时搜索的基本思路是从状态图上的起点和终点同时开始进行广搜或深搜。将起点和终点位置分别打上不同的标记，并使用搜索拓展，若某次拓展过程中遇到了不同的标记，则搜索的两端相遇了，那么可以认为是获得了可行解。

meet in the middle

主要思想是将整个搜索过程分成两半，分别搜索，最后将两半的结果合并。

暴力搜索的时间复杂度往往是指数级别的，而改用meet in the middle算法后复杂度的指数可以减半，即让复杂度从$O(a^b)$降低成$O(a^{b/2})$。

使用DFS同时维护模式，状态，步数基本可以较为简单的实现。

0.7.2 0-1BFS

本质是dijkstra算法，图上的边权有$0,1$两种。使用双端队列进行BFS，队首一定是队列中距离起点距离最短的点，走距离为$0/1$的边时分别将新的位置加入队首/队尾，以此来实现最短路中的松弛操作。

0.7.3 A*

A*算法是一种以BFS为基础的优化算法，在起点到终点的广阔搜索范围中，我们可以通过定义合理的启发式函数，缩小搜索的范围，从而加速搜索。

定义起点为s，终点为t，从起点（初始状态）开始的真实距离为$d(x)$，到终点（最终状态）的真实距离$h(x)$，估计距离为$h^*(x)$，必须满足$h^*(x) \leq h(x)$，以及每个点的估价函数$f(x) = d(x) + h^*(x)$。

A*算法每次从优先队列中取出一个f最小的元素，然后更新相邻的状态。在启发式函数的约束下，搜索效率得以提升，当**终点第一次离开队列时**，此时记录的便是最优答案。

当$h=0$时，A*算法变为 Dijkstra，当$h=0$并且边权为1时变为BFS。

八数码问题

题目大意：在3×3的棋盘上，摆有八个棋子，每个棋子上标有1至8的某一数字。棋盘中留有一个空格，空格用0来表示。空格周围的棋子可以移到空格中，这样原来的位置就会变成空格。给出一种初始布局和目标布局（为了使题目简单，设目标状态如下），找到一种从初始布局到目标布局最少步骤的移动方法。

启发式函数h定义为，不在应该在的位置的数字个数。

```
#include<bits/stdc++.h>
using namespace std;
string ed="123804765",st;
int h(string s)
{
    int cnt=0;
    for(int i=0;i<9;i++) cnt+=(s[i]!=ed[i]);
    return cnt;
}
map<string,int>vis;
int nex[4]={-3,3,-1,1};
struct node
{
    string str;int v;
    bool operator<(node x)const{return x.v+h(x.str)<v+h(str)
        ;}
};
int main()
{
    cin>>st;
    priority_queue<node>q;
    q.push({st,0});
    int ans;
    while(!q.empty())
    {
        string sta=q.top().str;
        int step=q.top().v;q.pop();
        if(sta==ed)
```

```
        {
            ans=step;break;
        }
        int pos;
        for(int i=0;i<9;i++) if(sta[i]=='0') pos=i;
        for(int i=0;i<4;i++)
        {
            if((i==0&&pos/3==0)||(i==1&&pos/3==2)||(i==2&&
                pos%3==0)||(i==3&&pos%3==2))
            continue;
            string temp=sta;swap(temp[pos],temp[pos+nex[i]])
                ;
            if(!vis.count(temp))
            {
                vis[temp]=1;q.push({temp,step+1});
            }
        }
    }
    printf("%d\n",ans);
}
```

K短路

启发式函数为$f(x) = d(x) + dis(x)$，$dis(x)$为x点到终点的最短路距离，当终点第一次出队时为最短路，第k次出队时的距离即为第k短路。使用A*算法时间复杂度为$O(nk \log n)$，存在使用可持久化可并堆的算法可以做到在$O((n+m)\log n + k \log k)$的时间复杂度解决$k$短路问题。

```
#include<bits/stdc++.h>
using namespace std;
typedef pair<double,int> PDI;
const int maxn=5005;
int n,m,num,h[maxn],h1[maxn],vis[maxn];
double sum,dis[maxn];
struct road{int b,nex;double c;}r[400005];
void add(int a,int b,double c){r[num].b=b;r[num].c=c;r[num].
    nex=h[a];h[a]=num++;}
void add1(int a,int b,double c){r[num].b=b;r[num].c=c;r[num
    ].nex=h1[a];h1[a]=num++;}
void dijkstra(int num)
{
    for(int i=1;i<=n;i++) dis[i]=1e9;
```

```
13      priority_queue<PDI,vector<PDI>,greater<PDI>>q;
14      dis[num]=0;q.push({0,num});
15      while(!q.empty())
16      {
17          int d=q.top().second;double v=q.top().first;q.pop();
18          if(vis[d]) continue;vis[d]=1;
19          for(int i=h1[d];~i;i=r[i].nex)
20          {
21              int nex=r[i].b;double val=r[i].c;
22              if(dis[nex]>dis[d]+val)
23              {
24                  dis[nex]=dis[d]+val;
25                  q.push({dis[nex],nex});
26              }
27          }
28      }
29  }
30  struct node
31  {
32      int p;double s;
33      bool operator<(node x)const{return x.s+dis[x.p]<s+dis[p
            ];}
34  };
35  int astar()
36  {
37      priority_queue<node>q;q.push({1,0});
38      int ans=0;
39      while(!q.empty())
40      {
41          int pos=q.top().p;double step=q.top().s;q.pop();
42          if(pos==n)
43          {
44              if(sum>=step) sum-=step,ans++;
45              else break;
46          }
47          for(int i=h[pos];~i;i=r[i].nex)
48          {
49              int nex=r[i].b;q.push({nex,r[i].c+step});
50          }
51      }
52      return ans;
53  }
```

```
54 int main()
55 {
56     memset(h,-1,sizeof(h));
57     memset(h1,-1,sizeof(h1));
58     scanf("%d%d%lf",&n,&m,&sum);
59     for(int i=1;i<=m;i++)
60     {
61         int a,b;double w;
62         scanf("%d%d%lf",&a,&b,&w);
63         add(a,b,w);add1(b,a,w);
64     }
65     dijkstra(n);
66     printf("%d\n",astar());
67 }
```

0.7.4 迭代加深

迭代加深是一种**每次限制搜索深度的**深度优先搜索。迭代加深在搜索的同时带上了一个深度d，当d达到设定的深度时就返回，一般用于找最优解。如果一次搜索没有找到合法的解，就让设定的深度加一，重新从根开始。

可以认为迭代加深是一种使用DFS实现BFS的过程，相较于BFS队列记录信息较多时会占用较大的内存，迭代加深空间复杂度相对较小。

UVA529 Addition Chains

```
1  #include<bits/stdc++.h>
2  using namespace std;
3  int n,ans[100],lim;
4  bool dfs(int u)
5  {
6      if(u>lim) return ans[u-1]==n;
7      if(ans[u-1]*(1<<(lim-u+1))<n) return false;
8      for(int i=1;i<u;i++)
9      {
10         for(int j=i;j<u;j++)
11         {
12             if(ans[i]+ans[j]>n) break;
13             ans[u]=ans[i]+ans[j];
14             if(dfs(u+1)) return true;
15         }
16     }
```

```
    return false;
}
int main()
{
    while(scanf("%d",&n)!=EOF&&n)
    {
        ans[1]=1;lim=1;
        while(!dfs(2)) lim++;
        for(int i=1;i<=lim;i++) printf("%d ",ans[i]);
        puts("");
    }
}
```

0.7.5 IDA*

本质上是迭代加深DFS和A*的结合。整个过程较为简单，构造合适的启发式函数，在使用迭代加深搜素时，用启发式函数判断距离，从而快速减枝。

0.8 拓扑排序

拓扑排序找环

```
bool topsort()
{
    int cnt=0;queue<int>q;
    for(int i=1;i<=n;i++) if(d[i]==0) q.push(i),cnt++;

    while(!q.empty())
    {
        int k=q.front();q.pop();
        for(int i=head[k];~i;i=r[i].nex)
        {
            int v=r[i].b;
            if(--d[v]==0)
            {
                q.push(v);cnt++;
            }
        }
    }
    return cnt==n;
```

```
}
```

0.9 最短路

0.9.1 Dijkstra

```
void dijkstra(int num)
{
    priority_queue<PII,vector<PII>,greater<PII> >q;
    dis[num]=0;
    q.push({0,num});
    while(!q.empty())
    {
        PII p=q.top(); q.pop();
        int d=p.second,v=p.first;
        if(vis[d]) continue;
        vis[d]=1;
        for(int i=head[d];~i;i=r[i].nex)
        {
            int next=r[i].b,val=r[i].c;
            if(dis[next]>dis[d]+val)
            {
                dis[next]=dis[d]+val;
                q.push({dis[next],next});
            }
        }
    }
}
```

0.9.2 SPFA

```
bool spfa(int num)
{
    queue<int>q;dis[num]=0;
    q.push(num);vis[num]=1;
    while(!q.empty())
    {
        int k=q.front();q.pop();vis[k]=0;
        for(int i=head[k];~i;i=r[i].nex)
        {
```

```
10              if(dis[r[i].b]>dis[k]+r[i].c)
11              {
12                  dis[r[i].b]=dis[k]+r[i].c;
13                  cnt[r[i].b]=cnt[k]+1;
14                  if(cnt[r[i].b]>n) return false;
15                  if(vis[r[i].b]==0)
16                  {
17                      q.push(r[i].b);
18                      vis[r[i].b]=1;
19                  }
20              }
21          }
22      }
23      return true;
24  }
```

SPFA其余形式优化

普通SPFA是非常好卡的，只需要一个随机网格图（在网格图中走错一次路可能导致很高的额外开销），或者一个构造过的链套菊花（使得队列更新菊花的次数非常高）即可。很多奇怪写法的SPFA都只能通过两者中的至多一种，因此你只需要将图构造为网格套菊花即可。

堆优化：将队列换成堆，与Dijkstra的区别是允许一个点多次入队。在有负权边的图可能被卡成指数级复杂度。

栈优化：将队列换成栈（即将原来的BFS过程变成 DFS），在寻找负环时可能具有更高效率，但最坏时间复杂度仍然为指数级。

LLL优化：将普通队列换成双端队列，每次将入队结点距离和队内距离平均值比较，如果更大则插入至队尾，否则插入队首。

SLF优化：将普通队列换成双端队列，每次将入队结点距离和队首比较，如果更大则插入至队尾，否则插入队首。

D´Esopo-Pape 算法：将普通队列换成双端队列，如果一个节点之前没有入队，则将其插入队尾，否则插入队首。

0.9.3 差分约束

求最大值：

按照$B-A\leq W$进行不等式的转化add(a,b,w)，求出图中的最短路，即为最大值。

求最小值：

按照$B-A\geq w$进行不等式的转化add(a,b,w)，最长路即为最小值。如果图中存在负环/正环，则不等式无法成立。

差分约束找环求最值不一定非要使用SPFA求最短路求解，对于边权均为正/负的图，可以先用Tarjan判环缩点为DAG后使用拓扑排序求解。

0.9.4 同余最短路

当出现形如“给定n个整数，求这n个整数能拼凑出多少的其他整数（n个整数可以重复取）”，以及“给定n个整数，求这n个整数不能拼凑出的最小（最大）的整数”，或者“至少要拼几次才能拼出模K余p的数”的问题时可以使用同余最短路的方法。

对于x,y,z三个数字能够拼凑出来的小于h的数字，令dis_i为只使用y和z，能得到的模x下与i同余的最小数，用来计算该同余类满足条件的数个数。

于是可以有以下转移：$\begin{cases} i \xrightarrow{y} (i+y) \bmod x \\ i \xrightarrow{z} (i+z) \bmod x \end{cases}$

相当于在在图上建边$\begin{cases} add(i,(i+y)\%x,y) \\ add(i,(i+z)\%x,z) \end{cases}$

接下来只需要对整个图跑最短路，得到所有的dis，最后的答案即为：

$$\sum_{i=0}^{x-1}\left(\frac{h-d_i}{x}+1\right)$$

```cpp
#include<bits/stdc++.h>
using namespace std;
typedef pair<long long,int> PII;
const int maxn=1e5+7;
int x,y,z,num,head[maxn],vis[maxn];
long long h,dis[maxn];
struct road{int b,c,nex;}r[2000005];
void add(int a,int b,int c)
{r[num].b=b;r[num].c=c;r[num].nex=head[a];head[a]=num++;}
void dijkstra(int num)
{
    memset(dis,0x3f3f,sizeof(dis));
    priority_queue<PII,vector<PII>,greater<PII> >q;
    dis[num]=0;q.push({0,num});
    while(!q.empty())
    {
        PII p=q.top(); q.pop();
        int d=p.second,v=p.first;
```

```
19          if(vis[d]) continue;
20          vis[d]=1;
21          for(int i=head[d];~i;i=r[i].nex)
22          {
23              int next=r[i].b,val=r[i].c;
24              if(dis[next]>dis[d]+val)
25              {
26                  dis[next]=dis[d]+val;
27                  q.push({dis[next],next});
28              }
29          }
30      }
31  }
32  int main()
33  {
34      memset(head,-1,sizeof(head));
35      scanf("%lld%d%d%d",&h,&x,&y,&z);
36      if(x==1)//注意特判
37      {
38          printf("%lld\n",h);
39          return 0;
40      }
41      for(int i=0;i<x;i++)
42      {
43          add(i,(i+y)%x,y);add(i,(i+z)%x,z);
44      }
45      dijkstra(0);
46      long long ans=0;
47      for(int i=0;i<x;i++)
48      {
49          if(h>=dis[i]) ans+=((h-dis[i])/x+1);
50      }
51      printf("%lld\n",ans);
52  }
```

0.10 欧拉图

定义：

通过图中所有边恰好一次的通路称为欧拉通路。

通过图中所有边恰好一次的回路称为欧拉回路。

具有欧拉回路的无向图或有向图称为欧拉图。

具有欧拉通路但不具有欧拉回路的无向图或有向图称为半欧拉图。

非形式化地讲，欧拉图就是从任意一个点开始都可以一笔画完整个图，半欧拉图必须从某个点开始才能一笔画完整个图。

性质：

欧拉图中所有顶点的度数都是偶数。

若G是欧拉图，则它为若干个环的并，且每条边被包含在奇数个环内。

辨别法：

对于无向图（图是连通图）：

欧拉通路的充要条件：度数为奇数的点只能有0个或2个。

欧拉回路的充要条件：度数为奇数的点只能有0个。

对于有向图（图是连通图）：

欧拉通路的充要条件：要么所有点的入度等于出度，要么除了两个点外所有点的入度等于出度，这两个点一个出度比入度多1（起点），一个入度比出度多1（终点）。

欧拉回路的充要条件：所有点的入度等于出度。

0.10.1 Fleury算法

也称避桥法，是一个偏暴力的算法。

算法流程为每次选择下一条边的时候优先选择不是桥的边。

一个广泛使用但是错误的实现方式是先Tarjan预处理桥边，然后再DFS避免走桥。但是由于走图过程中边会被删去，一些非桥边会变为桥边导致错误。最简单的实现方法是每次删除一条边之后暴力跑一遍Tarjan找桥，时间复杂度是$O(m^2)$。复杂的实现方法要用到动态图等，实用价值不高。

0.10.2 Hierholzer算法

无向图欧拉回路字典序最小输出方案

```
struct node
{
    int to,exit,rev;
    bool operator<(node x)const{return to<x.to;}
};
vector<node>v[maxn];
vector<int>ans;
void dfs(int u)
{
    for(int i=head[u];i<v[u].size();i=head[u])
```

```
11      {
12          if(v[u][i].exit)
13          {
14              node e=v[u][i];
15              v[u][i].exit=0;v[e.to][e.rev].exit=0;
16              head[u]++;
17              dfs(e.to);
18          }
19          else head[u]++;
20      }
21      ans.push_back(u);
22  }
23  int revtop[maxn];
24  int main()
25  {
26      scanf("%d",&m);
27      for(int i=1;i<=m;i++)
28      {
29          int a,b;scanf("%d%d",&a,&b);
30          v[a].push_back({b,1,0});v[b].push_back({a,1,0});
31          d[a]++;d[b]++;
32      }
33      for(int i=1;i<=n;i++) sort(v[i].begin(),v[i].end());
34      for(int i=1;i<=n;i++)
35      {
36          for(int j=0;j<v[i].size();j++) v[i][j].rev=revtop[v[
                i][j].to]++;
37      }
38      int st=1;
39      for(int i=1;i<=n;i++) if(d[i]%2){st=i;break;}
40      dfs(st);
41      reverse(ans.begin(),ans.end());
42      for(auto i:ans) printf("%d\n",i);
43  }
```

0.10.3 混合图欧拉回路

将所有无向边任意规定一个方向，统计各点的度数，若某点入度出度和为奇数，则不存在欧拉回路。

满足上述条件后，删除原图中的有向边，用网络流建图。

对出度大于入度的点，用源点向其连接大小为需要改变的边数的边。

对入度大于出度的点，用汇点向其连接大小为需要改变的边数的边，其余规定了方向的无向边在网络流图上流量为1。跑网络流，如果可以满流，则说明存在欧拉回路，满流的边意味着该无向边需要反转方向。

0.11 哈密顿图

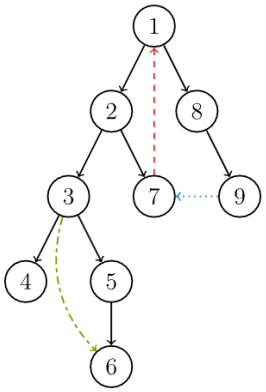
定义：

通过图中所有顶点一次且仅一次的通路称为哈密顿通路。

通过图中所有顶点一次且仅一次的回路称为哈密顿回路。

具有哈密顿回路的图称为哈密顿图。具有哈密顿通路而不具有哈密顿回路的图称为半哈密顿图。

0.12 连通图



一颗DFS生成树包含四种类型的边，**树边，返祖边，横叉边，前向边**。

定义DFN(u)为节点u搜索的次序编号(时间戳)，Low(u)为u或u的子树能够追溯到的最早的**栈中**节点的次序号。

0.12.1 强连通分量

```
1 void tarjan(int u)
2 {
3     dfn[u]=low[u]=++tim;
4     atack.push(u);inst[u]=1;
5     for(int i=head[u];~i;i=r[i].nex)
6     {
7         int v=r[i].b;
8         if(dfn[v]==0)
```

```
        {
            tarjan(v);low[u]=min(low[u],low[v]);
        }
        else if(inst[v]==1) low[u]=min(low[u],dfn[v]);
    }
    if(dfn[u]==low[u])
    {
        numb++;int q;
        do
        {
            q=atack.top();inst[q]=0;atack.pop();
            bl[q]=numb;nums[numb]++;
        }while(q!=u);
    }
}
```

将多个强连通分量加边联通为一个强连通分量，最少需要$\max(num_{ru=0}, num_{chu=0})$条边,只有一个强连通分量需要特判。

0.12.2 双连通分量

只要删去桥还能够保持联通，就是边双连通。

边双连通分量

```
void tarjan(int u,int f)//是路线编号f
{
    atack.push(u);
    dfn[u]=low[u]=++tim;
    for(int i=head[u];~i;i=r[i].nex)
    {
        int v=r[i].b;
        if(!dfn[v])
        {
            tarjan(v,i);low[u]=min(low[u],low[v]);
        }
        else if(i!=(f^1)) low[u]=min(low[u],dfn[v]);
    }
    if(dfn[u]==low[u])
    {
        numb++;int t;
        do
        {
```

```
            t=atack.top();atack.pop();
            bl[t]=numb;nums[numb]++;
        }while(t!=u);
    }
}
```

只要删去割点还能保持联通，就是点双连通，但是要注意，一个割点可能同时属于多个点双连通分量。**注意特判自环!!!!**

点双连通分量

```
void tarjan(int u,int root)//有没有其实无所谓fa
{
    dfn[u]=low[u]=++tim;
    atack.push(u);
    if(u==root && head[u]==-1)//特判孤立点
    {
        dcc[++numb].push_back(u);return;
    }
    int child=0;
    for(int i=head[u];~i;i=r[i].nex)
    {
        int v=r[i].b;
        if(!dfn[v])
        {
            tarjan(v,root);
            low[u]=min(low[u],low[v]);
            if(dfn[u]==low[v])
            {
                child++;
                if(u!=root||child>1) isgd[u]=true;
                numb++;
                int y;
                do{
                    y=atack.top();atack.pop();
                    dcc[numb].push_back(y);
                }while(y!=v);
                dcc[numb].push_back(u);
            }
        }
        else low[u]=min(low[u],dfn[v]);
    }
}
```

0.12.3 割点和桥

将某点及其连边删去，图无法继续连通。

割点

```
void tarjan(int u,int root)
{
    dfn[u]=low[u]=++tim;
    int child=0;
    for(int i=head[u];~i;i=r[i].nex)
    {
        int v=r[i].b;
        if(!dfn[v])
        {
            tarjan(v,root);
            low[u]=min(low[u],low[v]);
            if(low[v]==dfn[u])
            {
                child++;
                if(u!=root || child>1)  isgd[u]=1;
            }
        }
        else
        low[u]=min(low[u],dfn[v]);
    }
}
```

删去图中的某个边，图不再连通，无向图只包含树边和非树边，没有非树边覆盖的树边就是桥。

桥

```
void tarjan(int u,int f)//是路线编号f
{
    dfn[u]=low[u]=++tim;
    for(int i=head[u];~i;i=r[i].nex)
    {
        int v=r[i].b;
        if(!dfn[v])
        {
            tarjan(v,i);
            low[u]=min(low[u],low[v]);
            if(dfn[u]<low[v])//和割点的区别
            {
```

```
                isbri[i]=isbri[i^1]=1;
            }
        }
        else if(i!=(f^1))
        low[u]=min(low[u],dfn[v]);
    }
}
```

一棵树上有n个度数为1的点，最少连接$(n+1)/2$条边即可变为双连通分量。

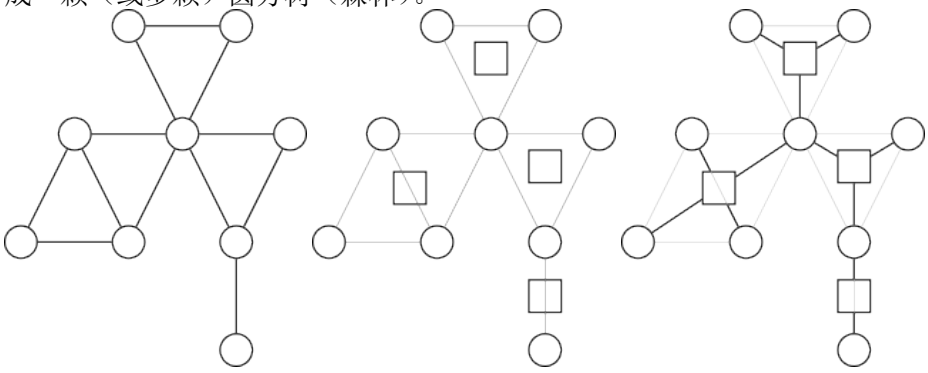
0.12.4 圆方树

众所周知，树（或森林）有很好的性质，并且容易通过很多常见数据结构维护。

而一般图则没有那么好的性质，所幸有时我们可以把一般图上的某些问题转化到树上考虑。

顾名思义，圆方树即有圆点有方点的树。

在一张无向图中，将图中原有的点视为圆点，将图中的点双连通分量视为一个方点，将方点对应的圆点用菊花图的连接方式连接起来。就会形成一颗（或多颗）圆方树（森林）。



技巧：

1.对圆方树上圆点权值初始化为-1，方点权值初始化为度数大小。任意两圆点在树上路径权值之和即为原无向图两点路径途经点集的数量和。

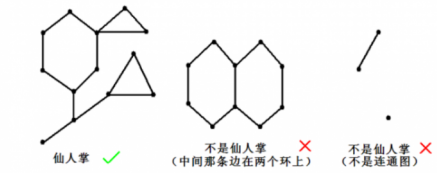
2.无向图任意两点间的割点数，等价于圆方树上两点树上路径上的圆点数量。

```
void tarjan(int u)
{
    dfn[u]=low[u]=++tim;
    stk.push(u);
    for(int i=head[u];~i;i=r[i].nex)
```

```
    {
        int v=r[i].b;
        if(!dfn[v])
        {
            tarjan(v);low[u]=min(low[u],low[v]);
            if(low[v]==dfn[u])
            {
                idx++;int x;
                do
                {
                    x=stk.top();stk.pop();
                    rst[idx].push_back(x);
                    rst[x].push_back(idx);
                }while(x!=v);
                rst[u].push_back(idx);
                rst[idx].push_back(u);
            }
        }
        else low[u]=min(low[u],dfn[v]);
    }
}
```

0.13 仙人掌

仙人掌图（cactus）是一种无向连通图，它的每条边最多只能出现在一个简单回路里面。从直观上说，可以把仙人掌图理解为允许存在回路的树。如果一个图不包含**偶数环**那么这个图一定是仙人掌。



仙人掌圆方树

上述圆方树在某种意义上被认为是广义圆方树，即适用于一般无向图的圆方树。对于仙人掌，我们对其使用狭义圆方树。两种圆方树一个明显的区别就是，一个只有两个点的点双联通分量要不要建方点。因此除了广义圆方树上仅有的圆方边，在此类圆方树中还包含圆圆边，但两种圆方树中都不存在方方边。

仙人掌图本质上是一个树上开花（环）的树，因此我们对其做边双连通分量，对于树边（桥），继续用原图上的边对其连接（圆圆边），如果遇到了环，对环上节点依次与其方点连接。建好的树即为仙人掌圆方树。

仙人掌上最短路

对于圆圆边，边权为原边边权，圆方边的边权为0，方圆边的边权为原图上圆点到方点父亲的最短路。

建好树后，记$dis(x)$为x点到树根的距离，对于u到v的最短路，若其LCA为圆点，最短路即$dis(u)+dis(v)-2*dis(LCA)$。若为方点，找出LCA的两个儿子A,B，分别为u,v的祖先，此时最短路即$dis(A,B)+dis(u,A)+dis(v,B)$。其中$dis(A,B)$可以通过预处理环长计算。

```
#include<bits/stdc++.h>
using namespace std;
typedef long long ll;
const int maxn=20005;
int n,m,q,f[maxn][16],head[maxn],dep[maxn],num,dfn[maxn],low
    [maxn],tim,idx;
int sum[maxn],len[maxn],fa[maxn],dis[maxn],A,B;
struct road{int b,c,nex;}r[2000005];
void add(int a,int b,int c){r[num].b=b;r[num].c=c;r[num].nex
    =head[a];head[a]=num++;}
stack<int>stk;
vector<pair<int,int>>rst[maxn];
void build_tree(int u,int v,int val)
{
    int pre=val,x=v;
    while(x!=fa[u])
    {
        sum[x]=pre;pre+=len[x];x=fa[x];
    }
    sum[++idx]=sum[u];
    x=v;rst[u].push_back({idx,0});
    while(x!=u)
    {
        int mn=min(sum[idx]-sum[x],sum[x]);
        rst[idx].push_back({x,mn});x=fa[x];
    }
}
void tarjan(int u,int from)
```

```
{
    dfn[u]=low[u]=++tim;
    stk.push(u);
    for(int i=head[u];~i;i=r[i].nex)
    {
        int v=r[i].b;
        if(!dfn[v])
        {
            len[v]=r[i].c;fa[v]=u;
            tarjan(v,i);
            low[u]=min(low[u],low[v]);
        }
        else if(i!=(from^1)) low[u]=min(low[u],dfn[v]);
        if(low[v]>dfn[u]) //桥圆圆边=
        {
            rst[u].push_back({v,r[i].c});
        }
    }
    for(int i=head[u];~i;i=r[i].nex)
    {
        int v=r[i].b;
        if(fa[v]==u||dfn[v]<=dfn[u]) continue;
        build_tree(u,v,r[i].c);
    }
}
void dfs(int u,int fa)
{
    f[u][0]=fa;dep[u]=dep[fa]+1;
    for(int i=1;i<=15;i++) f[u][i]=f[f[u][i-1]][i-1];
    for(auto [nex,w]:rst[u])
    {
        dis[nex]=dis[u]+w;
        dfs(nex,u);
    }
}
int lca(int a,int b)
{
    if(dep[a]<dep[b]) swap(a,b);
    for(int i=15;i>=0;i--)
    {
        if(dep[f[a][i]]>=dep[b]) a=f[a][i];
    }
```

```
69      if(a==b) return a;
70      for(int i=15;i>=0;i--)
71      {
72          if(f[a][i]!=f[b][i]) a=f[a][i],b=f[b][i];
73      }
74      A=a,B=b;
75      return f[a][0];
76  }
77  int main()
78  {
79      memset(head,-1,sizeof(head));
80      scanf("%d%d%d",&n,&m,&q);idx=n;
81      for(int i=1;i<=m;i++)
82      {
83          int u,v,w;scanf("%d%d%d",&u,&v,&w);
84          add(u,v,w);add(v,u,w);
85      }
86      tarjan(1,-1);
87      dfs(1,0);
88      while(q--)
89      {
90          int a,b;scanf("%d%d",&a,&b);
91          int LCA=lca(a,b);
92          if(LCA<=n) printf("%d\n",dis[a]+dis[b]-2*dis[LCA]);
93          else printf("%d\n",dis[a]+dis[b]-dis[A]-dis[B]+min(
                abs(sum[A]-sum[B]),sum[LCA]-abs(sum[A]-sum[B])));
94      }
95  }
```

0.14 2-SAT

个人认为2-SAT跟强连通的关系有点像差分约束跟最短路的关系。

k-SAT问题：有n个$bool$类型的事件，有m种约束，每种约束都有k个事件的关系。找出一种符合约束的n个事件的取值。（NP完全问题）

将n个事件拆分为$2*n$个点，分别代表$0,1$两种情况。对于所有的约束，我们根据题意建立有向图。若在有向图中有$(a \Rightarrow b)$，则意为选择了a一定要同时选择b才可以满足约束条件。

根据题意建好图后，应首先判断事件约束是否合法。对其进行**强连通分量缩点**，若a,b在同一个强连通分量内，则意味着a,b绑定在了一起，必须同时选取才能满足约束。这种情况下，如果某个事件的$0,1$情况在同一个强

连通分量下，该事件就又要为1又要为0，显然这是矛盾的。

在判断完上述情况后，就一定存在满足约束要求的合法方案。优先选择强连通分量标号较小的情况，因为在Tarjan结束后的出栈顺序决定了缩点后DAG的拓扑序。

原式	构图
$a \vee b = 1$	$(\neg a \Rightarrow b) \wedge (\neg b \Rightarrow a)$
$a\&b = 1$	$(\neg a \Rightarrow a) \wedge (\neg b \Rightarrow b)$
$a\&b = 0$	$(a \Rightarrow \neg b) \wedge (b \Rightarrow \neg a)$
$a \mid b = 1$	$(\neg a \Rightarrow b) \wedge (\neg b \Rightarrow a)$
$a \mid b = 0$	$(a \Rightarrow \neg a) \wedge (b \Rightarrow \neg b)$
$a \oplus b = 1$	$(a \Rightarrow \neg b) \wedge (\neg a \Rightarrow b) \wedge (b \Rightarrow \neg a) \wedge (\neg b \Rightarrow a)$
$a \oplus b = 0$	$(a \Rightarrow b) \wedge (\neg a \Rightarrow \neg b) \wedge (b \Rightarrow a) \wedge (\neg b \Rightarrow \neg a)$

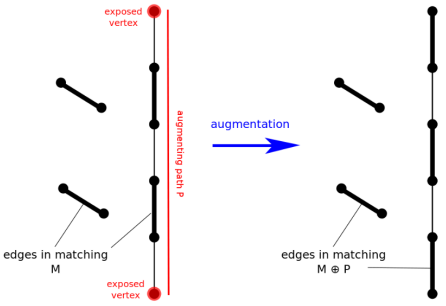
```
for(int i=1;i<=n;i++)
{
    if(bl[i]==bl[i+n]){puts("IMPOSSIBLE");return 0;}
}
puts("POSSIBLE");
for(int i=1;i<=n;i++)
{
    if(bl[i]<bl[i+n]) printf("1 ");
    else printf("0 ");
}
```

0.15 图的匹配

- 最大匹配: 匹配数最多的匹配。
- 完美匹配: 所有点都属于匹配，同时也符合最大匹配。
- 近完美匹配: 发生在图的点数为奇数，刚好只有一个点不在匹配中。

增广路定理

交错路：始于非匹配点且由匹配边与非匹配边交错而成。
增广路：是始于非匹配点且终于非匹配点的交错路。



增广路上非匹配边比匹配边数量多一，如果将匹配边改为未匹配边，反之亦然，则匹配大小会增加一且依然是交错路。

0.15.1 二分图

染色法判断二分图

```
bool dfs(int u,int color)
{
    col[u]=color;
    for(int i=head[u];~i;i=r[i].nex)
    {
        int next=r[i].b;
        if(col[next]==0)
        {
            if(!dfs(next,3-color)) return false;
        }
        else
        {
            if(col[next]==color) return false;
        }
    }
    return true;
}
```

二分图完美匹配

设$G =< V_1, V_2, E >$为二分图，$|V_1| \leq |V_2|$，M为G中一个最大匹配，且$M = |V_1|$，则称M为V_1到V_2的完美匹配。

霍尔定理

对于一个二分图，如果对于左边任意子集S，其对应边连接了一个右边

的边集T，都有$|S| <= |T|$，那么这个二分图有完美匹配（充要）。

0.15.2 二分图最大匹配

匈牙利算法

邻接矩阵$O(n^3)$

```
bool match(int x)
{
    for(int i=1;i<=n2;i++)
    {
        if(side[x][i]&&!vis[i])
        {
            vis[i]=1; //将加入增广路上i
            if(!mat[i]||match(mat[i]))
            {
                mat[i]=x;return true;
            }
        }
    }
    return false;
}
int Hungarian()
{
    int ans=0;
    for(int i=1;i<=n1;i++)
    {
        memset(vis,0,sizeof(vis));
        if(match(i)) ans++;
    }
    return ans;
}
```

- 换用邻接表，可将时间优化至$O(nm)$，使用网络流建图跑dinic，时间复杂度$O(m\sqrt{n})$。

- 若同侧匹配，记得反向记录mat，且第一次开始递归的节点x标记vis[x]=1。

二分图最小点覆盖（König 定理）

最小点覆盖：选最少的点，满足每条边至少有一个端点被选。

二分图中，最小点覆盖=最大匹配。

二分图最大独立集

最大独立集：选最多的点，满足两两之间没有边相连。

因为在最小点覆盖中，任意一条边都被至少选了一个顶点，所以对于其点集的补集，任意一条边都被至多选了一个顶点，所以不存在边连接两个点集中的点，且该点集最大。因此二分图中，最大独立集=总点−最小点覆盖。

0.15.3 二分图最大权匹配

KM算法

本质上是匈牙利算法+贪心。

将两个集合中点数比较少的补点，使得两边点数相同，再将不存在的边权重设为0，这种情况下，问题就转换成求**最大权完美匹配问题**，从而能应用KM算法求解。

可行顶标：给每个节点i分配一个权值$l(i)$，对于所有边$w(u,v)$满足$w(u,v) \leq l(u)+l(v)$。

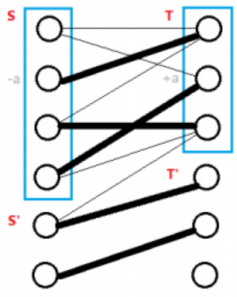
相等子图：在一组可行顶标下原图的生成子图，包含所有点但只包含满足$w(u,v) \leq l(u)+l(v)$的边$w(u,v)$。

我们的目标就是透过不断的调整可行顶标，使得相等子图是完美匹配。

在一开始，我们让图中左侧的点的相等子图都贪心的连向最大的匹配边，初始化顶标$lx(i)=\max_{1\leq j\leq n}\{w(i,j)\}, ly(i)=0$。

然后我们开始在图上增广，找到增广路就增广，否则，会得到一个交错树。

令S，T表示二分图左边右边在交错树中的点S'，T'表示不在交错树中的点。



此时无法继续修改，这时我们只能修改一些顶标使得相等子图发生变化从而可以继续匹配下去，这就意味着S中的某点必须做出让步，去匹配一些权值稍微低一点的边。我们找出$S \sim T'$中权值最小的边，权值$a = \min\{lx(u) + ly(v) - w(u,v) | u \in S, v \in T'\}$，使得损失尽可能降低。

使S中的顶标$-a$，T中的顶标$+a$，可以发现：$S \sim T$边依然在相等子图中。$S' \sim T'$没有变化。$S \sim T'$的$lx + ly$减少，有可能加入相等子图。$S' \sim T$的边不可能加入相等子图。

相等子图发生了变化，观察是否可以匹配，否则重复上述过程。由于原图一定存在完美匹配，所以最多n次一定可以找到增广路。

一开始枚举n个点找增广路，为了找增广路需要延伸n次交错树，每次延伸需要n次维护，共$O(n^3)$。

```
bool check(int v)
{
    visy[v]=1;
    if(maty[v]!=-1)
    {
        q.push(maty[v]);visx[maty[v]]=1;
        return false;
    }
    while(v!=-1){maty[v]=pre[v];swap(v,matx[pre[v]]);}
    return true;
}
void bfs(int x)
{
    while(!q.empty()) q.pop();
    q.push(x);visx[x]=1;
    while(1)
    {
        while(!q.empty())//使用BFS进行匹配
        {
            int u=q.front();q.pop();
            for(int v=1;v<=n;v++)
```

```
22              {
23                  if(visy[v]) continue;
24                  int delta=lx[u]+ly[v]-side[u][v];
25                  if(slack[v]<delta) continue;
26                  pre[v]=u; //delta=0意为改边属于相等子图
27                  if(delta) slack[v]=delta;
28                  else if(check(v)) return;
29              }
30          }
31          int mn=1e9;//匹配失败
32          for(int i=1;i<=n;i++) if(!visy[i]) mn=min(mn,slack[i
                ]);
33          for(int i=1;i<=n;i++)
34          {
35              if(visx[i]) lx[i]-=mn;
36              if(visy[i]) ly[i]+=mn;
37              else slack[i]-=mn;
38          }
39          for(int i=1;i<=n;i++)
40          {
41              if (!visy[i]&&slack[i]==0&&check(i)) return;
42          }
43      }
44  }
45  void km()
46  {
47      memset(lx,0,sizeof(lx));
48      memset(ly,0,sizeof(ly));
49      memset(matx,-1,sizeof(matx));
50      memset(maty,-1,sizeof(maty));
51      for(int i=1;i<=n;i++)
52      {
53          for(int j=1;j<=n;j++) lx[i]=max(lx[i],side[i][j]);
54      }
55      for(int i=1;i<=n;i++)
56      {
57          memset(visx,0,sizeof(visx));
58          memset(visy,0,sizeof(visy));
59          memset(slack,0x3f,sizeof(slack));
60          bfs(i);//slack[i]的值为右边第i个点和左边的点的顶标的最大值
61      }
62      int ans=0;
```

```
63      for(int i=1;i<=n;i++) ans+=side[i][matx[i]];
64      printf("%d\n",ans);
65      for(int i=1;i<=n;i++) printf("%d ",maty[i]);
66  }
```

转化为费用流模型

在图中新增一个源点和一个汇点。

从源点向二分图的每个左部点连一条流量为1，费用为0的边，从二分图的每个右部点向汇点连一条流量为1，费用为0的边。

接下来对于二分图中每一条连接左部点u和右部v，边权为w的边，则连一条从u到v，流量为1，费用为w的边。

求这个网络的**最大费用最大流**即可得到答案。

0.15.4 一般图最大匹配

带花树

一般图匹配和二分图匹配不同的是，图可能存在奇环。时间复杂度$O(n^3)$

```
1   int find(int x){return fa[x]==x?x:fa[x]=find(fa[x]);}
2   int LCA(int u,int v)
3   {
4       ++tim;u=find(u);v=find(v);
5       while(dfn[u]!=tim)//u,v轮流上跳，直到跳到环顶
6       {
7           dfn[u]=tim;
8           u=find(pre[mat[u]]);
9           if(v) swap(u,v);
10      }
11      return u;
12  }
13  void Blossom(int x,int y,int w)
14  {//对奇环进行缩花，同时将图上所有白点涂黑，向环外增广
15      while(find(x)!=w)
16      {
17          pre[x]=y,y=mat[x];
18          if(vis[y]==2) vis[y]=1,q.push(y);
19          if(find(x)==x) fa[x]=w;
20          if(find(y)==y) fa[y]=w;
21          x=pre[y];
```

```
22      }
23  }
24  int bfs(int x)
25  {
26      for(int i=1;i<=n;i++) fa[i]=i,vis[i]=pre[i]=0;
27      while(!q.empty()) q.pop(); q.push(x);vis[x]=1;
28      while(!q.empty())
29      {
30          int u=q.front();q.pop();
31          for(int i=head[u];~i;i=r[i].nex)
32          {
33              int v=r[i].b;
34              //u,v在同一朵花中或v是白色（偶环）
35              if(find(u)==find(v) || vis[v]==2) continue;
36              if(!vis[v]) //如果v尚未染色
37              {
38                  vis[v]=2;pre[v]=u;
39                  if(!mat[v])//增广成功
40                  {
41                      for(int j=v,last;j;j=last)
42                      last=mat[pre[j]],mat[j]=pre[j],mat[pre[j
                            ]]=j;
43                      return 1;
44                  }
45                  vis[mat[v]]=1;q.push(mat[v]);
46              }
47              else //如果v是黑色，出现奇环，开花
48              {
49                  int w=LCA(v,u);//第一次进入奇环的黑点
50                  Blossom(u,v,w);Blossom(v,u,w);
51              }
52          }
53      }
54      return 0;
55  }
56  void match()
57  {
58      int ans=0;
59      for(int i=1;i<=n;i++) if(!mat[i] && bfs(i)) ans++;
60      printf("%d\n",ans);
61      for(int i=1;i<=n;i++) printf("%d ",mat[i]);
62      puts("");
```

```
63 }
```

0.16 网络流

0.16.1 最大流

EK算法

EK算法就是BFS找增广路，然后对其进行增广,时间复杂度$O(nm^2)$。

```
1  int bfs(int s,int t)
2  {
3      memset(vis,0,sizeof(vis));
4      queue<int>q;q.push(s);
5      vis[s]=1;incf[s]=1e18;
6      while(!q.empty())
7      {
8          int u=q.front();q.pop();
9          for(int i=head[u];~i;i=r[i].nex)
10         {
11             int v=r[i].b;
12             if(vis[v] || r[i].c==0) continue;
13             q.push(v);vis[v]=1;pre[v]=i^1;
14             incf[v]=min(incf[u],r[i].c);
15             if(v==t) return 1;
16         }
17     }
18     return 0;
19 }
20 int EK(int s,int t)
21 {
22     int p=t;
23     while(p!=s)
24     {
25         r[pre[p]].c+=incf[t];
26         r[pre[p]^1].c-=incf[t];
27         p=r[pre[p]].b;
28     }
29     return incf[t];
30 }
```

Dinic算法

Dinic算法的过程是这样的：每次增广前，我们先用BFS来将图分层。设源点的层数为0，那么一个点的层数便是它离源点的最近距离。

然后使用DFS对流量进行增广，每次找增广路的时候，都只找比当前点层数多1的点进行增广（这样就可以确保我们找到的增广路是最短的），时间复杂度$O(n^2m)$。

多路增广优化：每次找到一条增广路的时候，如果残余流量没有用完怎么办呢？我们可以利用残余部分流量，再找出一条增广路。这样就可以在一次DFS中找出多条增广路，大大提高了算法的效率。

当前弧优化：如果一条边已经被增广过，那么它就没有可能被增广第二次。那么，我们下一次进行增广的时候，就可以不必再走那些已经被增广过的边。

```
int make_level()
{
    memset(depth,-1,sizeof(depth));
    queue<int>q;q.push(s);
    depth[s]=1;now[s]=head[s];//当前弧优化
    while(!q.empty())
    {
        int u=q.front();q.pop();
        for(int i=head[u];~i;i=r[i].nex)
        {
            int v=r[i].b;
            if(depth[v]!=-1 || r[i].c<=0) continue;
            now[v]=head[v];
            depth[v]=depth[u]+1;
            q.push(v);
        }
    }
    return depth[t]!=-1;
}
int dinic(int u,int flow)
{
    if(u==t) return flow;
    int sum=0;
    for(int i=now[u];~i;i=r[i].nex)//多路增广
    {
        now[u]=i;
        int v=r[i].b;
        if(depth[v]!=depth[u]+1 || r[i].c<=0) continue;
```

```
29          int use=dinic(v,min(flow-sum,r[i].c));
30          if(use)
31          {
32              r[i].c-=use;r[i^1].c+=use;
33              sum+=use;
34          }
35          if(sum==flow) return flow;
36      }
37      if(sum==0) depth[u]=-1;
38      return sum;
39  }
```

ISAP

在Dinic算法中，我们每次求完增广路后都要跑BFS来分层，有没有更高效的方法呢？ISAP是一种只需要一次BFS就可以不断增广的写法。

和Dinic算法一样，我们还是先跑BFS对图上的点进行分层，不过与Dinic略有不同的是，我们选择在反图上，从t点向s点进行BFS。

设i号点的层为d_i，当我们结束在i号点的增广过程后，我们遍历残量网络上i的所有出边，找到层最小的出点j，随后令$d_i = d_j + 1$。特别地，若残量网络上i无出边，则$d_i = n$。时间复杂度$O(n^3)$

GAP优化：记录每一层的点的数量，若某层为空，即出现了断层，直接结束。

```
1   void make_level()
2   {
3       queue<int>q;q.push(t);
4       depth[t]=1;gap[1]++;
5       while(!q.empty())
6       {
7           int u=q.front();q.pop();
8           for(int i=head[u];~i;i=r[i].nex)
9           {
10              int v=r[i].b;
11              if(depth[v]) continue;
12              depth[v]=depth[u]+1;
13              gap[depth[v]]++;
14              q.push(v);
15          }
16      }
17  }
```

```
ll dfs(ll u,ll flow)
{
    if(u==t) return flow;
    ll sum=0;
    for(int i=head[u];~i;i=r[i].nex)
    {
        ll v=r[i].b;
        if(depth[v]+1!=depth[u] || r[i].c==0) continue;
        ll use=dfs(v,min(flow-sum,r[i].c));
        if(use)
        {
            r[i].c-=use;
            r[i^1].c+=use;
            sum+=use;
        }
        if(sum==flow) return flow;
    }
    gap[depth[u]]--;
    if(gap[depth[u]==0]) depth[s]=n+1;//优化GAP
    depth[u]++;gap[depth[u]]++;
    return sum;
}
void ISAP()
{
    make_level();
    while(depth[s]<=n)
    {
        ans+=dfs(s,1e18);
    }
}
```

HPLL

时间复杂度$O(n^2\sqrt{m})$

```
#include<bits/stdc++.h>
using namespace std;
const int N = 1200, M = 120000, INF = 0x3f3f3f3f;
int n, m, s, t;
struct qxx {int nex, t, v;};
qxx e[M * 2 + 1];
int h[N + 1], cnt = 1;
```

```
8  void add_path(int f, int t, int v) {
9      e[++cnt] = (qxx){h[f], t, v}, h[f] = cnt; }
10 void add_flow(int f, int t, int v) {
11     add_path(f, t, v);
12     add_path(t, f, 0);
13 }
14 int ht[N + 1], ex[N + 1],
15     gap[N];  // 高度; 超额流; gap 优化 gap[i] 为高度为 i 的节点的数量
16 stack<int> B[N];  // 桶 B[i] 中记录所有 ht[v]==i 的v
17 int level = 0;    // 溢出节点的最高高度
18 int push(int u) {       // 尽可能通过能够推送的边推送超额流
19     bool init = u == s;  // 是否在初始化
20     for (int i = h[u]; i; i = e[i].nex) {
21         const int &v = e[i].t, &w = e[i].v;
22         if (!w || init == false && ht[u] != ht[v] + 1)  // 初始化时不考虑高度差为1
23             continue;
24         int k = init ? w : min(w, ex[u]);
25         // 取到剩余容量和超额流的最小值，初始化时可以使源的溢出量为负数。
26         if(v!=s && v!=t && !ex[v]) B[ht[v]].push(v),level=max(level, ht[v]);
27         ex[u] -= k, ex[v] += k, e[i].v -= k, e[i ^ 1].v += k;  // push
28         if (!ex[u]) return 0;  // 如果已经推送完就返回
29     }
30     return 1;
31 }
32 void relabel(int u) {  // 重贴标签（高度）
33     ht[u] = INF;
34     for (int i = h[u]; i; i = e[i].nex)
35         if (e[i].v) ht[u] = min(ht[u], ht[e[i].t]);
36     if (++ht[u] < n) {  // 只处理高度小于 n 的节点
37         B[ht[u]].push(u);
38         level = max(level, ht[u]);
39         ++gap[ht[u]];  // 新的高度，更新 gap
40     }
41 }
42 bool bfs_init() {
43     memset(ht, 0x3f, sizeof(ht));
44     queue<int> q;
45     q.push(t), ht[t] = 0;
46     while (q.size()) {  // 反向 BFS，遇到没有访问过的结点就入队
```

```
47          int u = q.front();
48          q.pop();
49          for (int i = h[u]; i; i = e[i].nex) {
50          const int &v = e[i].t;
51          if (e[i ^ 1].v && ht[v] > ht[u] + 1) ht[v] = ht[u] +
                1, q.push(v);
52          }
53      }
54      return ht[s] != INF;  // 如果图不连通，返回 0
55  }
56  // 选出当前高度最大的节点之一，如果已经没有溢出节点返回 0
57      int select() {
58      while (B[level].size() == 0 && level > -1) level--;
59      return level == -1 ? 0 : B[level].top();
60  }
61  int hlpp() {                    // 返回最大流
62      if (!bfs_init()) return 0;  // 图不连通
63      memset(gap, 0, sizeof(gap));
64      for (int i = 1; i <= n; i++)
65          if (ht[i] != INF) gap[ht[i]]++;  // 初始化 gap
66      ht[s] = n;
67      push(s);  // 初始化预流
68      int u;
69      while ((u = select())) {
70          B[level].pop();
71          if (push(u)) {  // 仍然溢出
72          if (!--gap[ht[u]])
73              for (int i = 1; i <= n; i++)
74              if (i != s && i != t && ht[i] > ht[u] && ht[i] <
                    n + 1)
75                  ht[i] = n + 1;  // 这里重贴成 n+1 的节点都不是溢
                        出节点
76          relabel(u);
77          }
78      }
79      return ex[t];
80  }
81  int main() {
82      scanf("%d%d%d%d", &n, &m, &s, &t);
83      for (int i = 1, u, v, w; i <= m; i++) {
84          scanf("%d%d%d", &u, &v, &w);
85          add_flow(u, v, w);
86      }
```

```
87      printf("%d", hlpp());
88      return 0;
89  }
```

0.16.2 最小割

对于一个网络流图$G=(V,E)$，其割的定义为一种点的划分方式：将所有的点划分为S和$T=V-S$两个集合，其中源点$s\in S$，汇点$t\in T$。

方案

我们可以通过从源点s开始DFS，每次走残量大于0的边，找到所有S点集内的点。

```
1  void dfs(int u)
2  {
3      vis[u]=1;
4      for(int i=head[u];~i;i=r[i].nex)
5      {
6          int v=r[i].b;
7          if(!vis[v]&&r[i].c) dfs(v);
8      }
9  }
```

割边数量

如果需要在最小割的前提下最小化割边数量，那么先求出最小割，把没有满流的边容量改成inf，满流的边容量改成1，重新跑一遍最小割就可求出最小割边数量；如果没有最小割的前提，直接把所有边的容量设成1，求一遍最小割就好了。

问题模型1：二者选其一

有n个物品和两个集合A,B，如果一个物品没有放入A集合会花费a_i，没有放入B集合会花费b_i；还有若干个形如u_i,v_i,w_i限制条件，表示如果u_i和v_i同时不在一个集合会花费w_i。每个物品必须且只能属于一个集合，求最小的代价。

我们对于每个集合设置源点s和汇点t，第i个点由s连一条容量为a_i的边、向t连一条容量为b_i的边。对于限制条件u,v,w，我们在u,v之间连容量

为w的双向边。

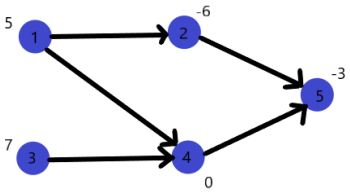
最小割就是最小花费。

问题模型2：最大权闭合图

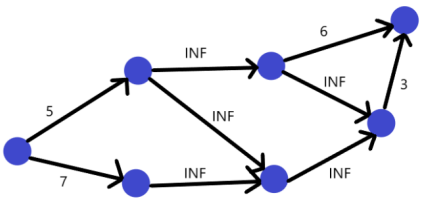
最大权值闭合图，即给定一张有向图，每个点都有一个权值（可以为正或负或0），你需要选择一个权值和最大的子图，使得子图中每个点的后继都在子图中。

做法：建立超级源点s和超级汇点t，若节点u权值为正，则s向u连一条有向边，边权即为该点点权；若节点u权值为负，则由u向t连一条有向边，边权即为该点点权的相反数。原图上所有边权改为inf。跑网络最大流，将所有正权值之和减去最大流，即为答案。

此外，该图应该是一个拓扑图，如果无法满足则应该先拓扑排序找出合法的边。



最大权闭合子图为{3, 4, 5}, 权值为4



判断一条边是否被割，应该通过最后一次增广后，depth是否为−1来判断。

0.16.3 费用流

只需将EK算法或Dinic算法中找增广路的过程，替换为用最短路/最长路算法寻找单位费用最小的增广路即可。

最小费用最大流

```
1 bool spfa()
2 {
3     for(int i=1;i<=n;i++)
4     {
```

```
5           pre[i]=-1;vis[i]=0;dis[i]=inf;
6       }
7       incf[s]=inf;
8       queue<int>q;q.push(s);
9       dis[s]=0;vis[s]=1;
10      while(!q.empty())
11      {
12          int u=q.front();q.pop();
13          vis[u]=0;
14          for(int i=head[u];~i;i=r[i].nex)
15          {
16              int v=r[i].b;
17              if(r[i].c && dis[v]>dis[u]+r[i].d)
18              {
19                  dis[v]=dis[u]+r[i].d;
20                  incf[v]=min(incf[u],r[i].c);
21                  pre[v]=i;
22                  if(vis[v]==0)
23                  {
24                      vis[v]=1;
25                      q.push(v);
26                  }
27              }
28          }
29      }
30      if(pre[t]==-1) return false;
31      return true;
32  }
33  void MCMF()
34  {
35      while(spfa())
36      {
37          int x=t;
38          maxflow+=incf[t];
39          mincost+=incf[t]*dis[t];
40          while(x!=s)
41          {
42              r[pre[x]].c-=incf[t];
43              r[1^pre[x]].c+=incf[t];
44              x=r[1^pre[x]].b;
45          }
46      }
```

```
47 }
```

0.16.4 上下界网络流

无源汇上下界可行流

给定无源汇流量网络G。询问是否存在一种标定每条边流量的方式，使得每条边流量满足上下界同时每一个点流量平衡（循环环流）。一个满足流量平衡的点意味着该点的流入流量等于流出流量。

首先每条边的流量应该大于其下界流量。假设每条边当前都只流出了其下界，用下界流量记录各点的流量平衡状态。在这个条件的基础上建立一个新图，新图每条边的流量为：流量上界−流量下界，也就是图上剩余的流量值。

建立虚拟源汇点S',T'。若点u流出量大于流入量，差值为M，我们就从S'出发向u连接一条值为M的附加边。若点u流出量小于流入量，差值为M，我们就从u出发向T'连接一条值为M的附加边。

如果附加边满流，说明这一个点的流量平衡条件可以满足，否则这个点的流量平衡条件不满足。在建图完毕之后跑S'到T'的最大流，若S'连出去的边全部满流，则存在可行流，否则不存在。

有源汇上下界可行流

给定有源汇流量网络G。询问是否存在一种标定每条边流量的方式，使得每条边流量满足上下界同时除了源点和汇点每一个点流量平衡。

假设源点为S，汇点为T。则我们可以加入一条T到S的上界为∞，下界为0的边转化为无源汇上下界可行流问题。若有解，则S到T的可行流流量等于T到S的附加边的流量。

有源汇上下界最大流

首先，先判断是否存在有源汇上下界可行流。如果找不到解就可以直接结束。

记计算出来的上下界可行流为$flow1$。否则，我们删去图上所有的附加边，在残量网络上算出S到T计算出最大流$flow2$。有源汇上下界最大流$=flow1+flow2$。

有源汇上下界最大流

```
1 #include<bits/stdc++.h>
2 #define inf 0x3f3f3f3f
```

```
using namespace std;
int n,m,in[205],out[205],depth[205],now[205],head[205],s,t,
    num,low[200005];
struct road{int b,c,nex;}r[200005];
void add(int a,int b,int c){
    r[num].b=b;r[num].c=c;r[num].nex=head[a];head[a]=num++;}
int make_level()
{
    memset(depth,-1,sizeof(depth));
    queue<int>q;q.push(s);
    depth[s]=1;now[s]=head[s];
    while(!q.empty())
    {
        int u=q.front();q.pop();
        for(int i=head[u];~i;i=r[i].nex)
        {
            int v=r[i].b;
            if(depth[v]!=-1 || r[i].c<=0) continue;
            now[v]=head[v];
            depth[v]=depth[u]+1;
            q.push(v);
        }
    }
    return depth[t]!=-1;
}
int dinic(int u,int flow)
{
    if(u==t) return flow;
    int sum=0;
    for(int i=now[u];~i;i=r[i].nex)
    {
        now[u]=i;
        int v=r[i].b;
        if(depth[v]!=depth[u]+1 || r[i].c<=0) continue;
        int use=dinic(v,min(flow-sum,r[i].c));
        if(use)
        {
            r[i].c-=use;r[i^1].c+=use;
            sum+=use;
        }
        if(sum==flow) return flow;
    }
```

```
44      if(sum==0) depth[u]=-1;
45      return sum;
46  }
47  int main()
48  {
49      memset(head,-1,sizeof(head));
50      scanf("%d%d%d%d",&n,&m,&s,&t);
51      for(int i=1;i<=m;i++)
52      {
53          int a,b,u;scanf("%d%d%d%d",&a,&b,&low[i],&u);
54          add(a,b,u-low[i]);add(b,a,0);
55          out[a]+=low[i];in[b]+=low[i];//记录各点的流量进出状况
56      }
57      int s1=n+1,t1=s1+1;
58      int sum=0,st=num;
59      for(int i=1;i<=n;i++)
60      {
61          if(in[i]==out[i]) continue;//建立附加边
62          if(in[i]>out[i]) add(s1,i,in[i]-out[i]),add(i,s1,0);
63          else add(i,t1,out[i]-in[i]),add(t1,i,0),sum+=abs(in[
                i]-out[i]);
64      }
65      add(t,s,inf);add(s,t,0);//建立附加边
66      int ans=0;
67      swap(s,s1);swap(t,t1);
68      while(make_level()) ans+=dinic(s,1e9);
69      if(ans==sum)//存在可行流
70      {
71          int flow1=r[num-1].c;
72          for(int i=st;i<num;i++) r[i].c=0; //删除图上所有的附加边
73          swap(s,s1);swap(t,t1);
74          while(make_level()) flow1+=dinic(s,inf); //flow1+
                flow2
75          printf("%d\n",flow1);
76      }
77      else//不存在可行流
78          puts("please go home to sleep");
79  }
```

有源汇上下界最小流

首先，先判断是否存在有源汇上下界可行流。如果找不到解就可以直接结束。

记计算出来的上下界可行流为$flow1$。否则，我们删去图上所有的附加边，在残量网络上算出T到S计算出最大流$flow2$。意为将多余的无用流量退还回去，有源汇上下界最大流$= flow1 - flow2$。

0.17 Stoer-Wagner算法

Stoer-Wagner算法是一种解决无向正权图上的**全局最小割**问题的算法。算法复杂度$O(nm + n^2 \log |n|)$一般可近似看作$O(n^3)$。

算法过程:

1.在图G中任意指定两点s, t，并且以这两点作为源汇点求出最小割，更新当前答案。

2.将t合并入s变为同一点。合并过程：删除s, t之间的连边，对于$G/s, t$中任意一点k，删除t, k，并将其边权$d(t, k)$加到$d(s, k)$上。

3.输出所有最小割的最小值。

若选择的割边会将两点s, t分为两个连通块，则该割边的大小即以s, t为源汇的最小割。否则，s, t将绑定在一起，共享所有的边。因此，处理完一对s, t之间的最小割后，就只有它们处于同一连通块的情况了，也就是做完一对以后就合并一对点，如是进行次$n - 1$即合并成一个点，算法完成。

最小割求法:

假设进行若干次合并以后，当前图$G' = (V', E')$，我们构造一个集合A，初始时令$A = \varnothing$。

我们每次将G'中所有点中，满足$i \notin A$，且权值函数$w(i)$最大的节点加入集合A，直到$|A| = |V'|$。

其中$w(i) = \sum_{j \in A} d(i, j)$。

令$ord(i)$表示第i个加入A的点，令s为$ord(|V' - 1|)$，t为$ord(|V'|)$，则此时的$w(t)$就是s到t的最小割。

```
int contract(int x)
{
    memset(vis,0,sizeof(vis));
    memset(w,0,sizeof(w));
    w[0]=-1;
    for(int i=1;i<=n-x+1;i++)
    {
```

```
8           int mx=0;
9           for (int j=1;j<=n;j++)
10          {   //dap[j]=1表示j点已经与其他点合并
11              if(!dap[j]&&!vis[j]&&w[j]>w[mx]) mx=j;
12          }
13          vis[mx]=1,ord[i]=mx; //第i个加入集合A的点是ord[i]
14          for(int j=1;j<=n;j++)
15          {
16              if(!dap[j]&&!vis[j]) w[j]+=side[mx][j]; //更新维
                    护w函数
17          }
18      }
19      s=ord[n-x],t=ord[n-x+1]; //这样的s和t的最小割一定是w[t]
20      return w[t];
21  }
22  int Stoer_Wagner()
23  {
24      int res=INF;
25      for (int i=1;i<n;i++)
26      {
27          res=min(res,contract(i));
28          dap[t]=1; //将t与s合并
29          for (int j=1;j<=n;j++)
30          {
31              side[s][j]+=side[t][j];
32              side[j][s]+=side[j][t];
33          }
34      }
35      return res;
36  }
```

0.18 特殊的图

0.18.1 竞赛图

竞赛图也叫有向完全图。每对顶点之间都有一条边相连的有向图称为竞赛图。

- 竞赛图没有自环，没有二元环；若竞赛图存在环，则一定存在三元环。（如果存在一个环大于三元，那么一定存在另一个三元的小环。）
- 任意竞赛图都有哈密顿路径（经过每个点一次的路径，不要求回到出

发点）。

- 图存在哈密顿回路的充要条件是强联通。

兰道定理

兰道定理是用来判定竞赛图的定理。将一个竞赛图的每一个点的出度从小到大排序后得到长度为n的序列称为竞赛图的比分序列$s = s_1 \leq s_2 \leq \cdots \leq s_n$是合法的比分序列当且仅当：

$$\forall 1 \leq k \leq n, \sum_{i=1}^{k} s_i \geq \binom{k}{2}$$

且$k = n$时一定相等。

0.18.2 平面图

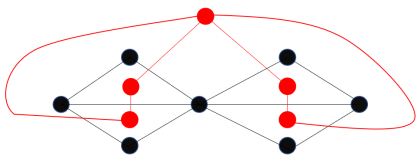
如果可以将一个图画在二维平面上，可以存在一种画法使任意两条边都不相交，该图就是平面图。

如何判断一个图是不是平面图？

拓扑学欧拉公式：对于一个连通平面图$G = (V, E, F), |V| - |E| + |F| = 2$则该图为平面图，其中$|V|, |E|, |F|$分别为点数，边数，和形成的面数（面内不应该有其余的边，且最外面的无限大的区域也算一个面）。
证明：对于一棵树，一定是平面图，且满足$|V| - |E| + |F| = 2$，此后，每增加一条边，都会多一条边和一个面。平面图边数和点数的关系：$m \leq 3 * n - 6$。

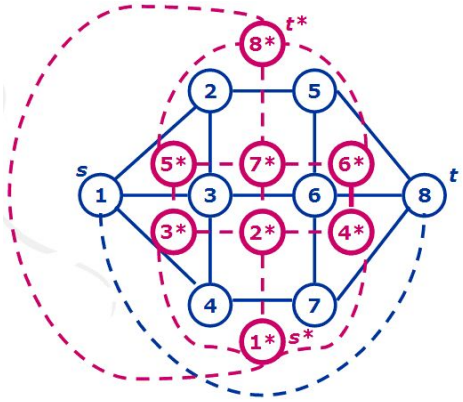
0.18.3 对偶图

对偶图是与平面图相伴的一种图。将平面图中的每个面都变为一个点，将各个面割开的边变为新的边，权值不变，连接在新的点上。对偶图的对偶图是原图。



平面图的最小割等于其对偶图的最短路

对于一个$s-t$平面图，我们对其进行如下改造：首先连接$s-t$得到一个附加面，求该图的对偶图G，令附加面对应的点为s，无界面对应的点为t，然后对图G进行连边，此时$s-t$最短路就是最小割。



0.19 最小环

给出一个图，问其中的有n个节点构成的边权和最小的环$(n \geq 3)$是多大。二元化较为特殊，处理方法也比较简单。

Dijkstra

设u和v之间有一条边长为w的边，$dis(u,v)$表示删除u和v之间的连边之后，两点之间的最短路。那么图中的最小环是$dis(u,v)+w$。时间复杂度$O(m(n+m)\log n)$。

floyd

```
int val[maxn+1][maxn+1]; //原图的邻接矩阵
int floyd(int &n)
{
    int dis[maxn+1][maxn+1]; //最短路矩阵
    for(int i=1;i<=n;++i)
        for(int j=1;j<=n;++j) dis[i][j]=val[i][j]; //初始化最短路矩阵
    int ans=inf;
    for(int k=1;k<=n;++k)
    {
        for(int i=1;i<k;++i)
```

```
11              for(int j=1;j<i;++j)
12              ans=min(ans,dis[i][j]+val[i][k]+val[k][j]); //更
                    新答案
13          for(int i=1;i<=n;++i)
14              for(int j=1;j<=n;++j)
15              dis[i][j]=min(dis[i][j],dis[i][k]+dis[k][j]);
16      }
17      return ans;
18  }
```