图论模板

- 并查集操作
- bitset求可达性
- 最小生成树
  - Kruskal O(elog(e))
- 最短路
  - Dijkstra
  - 堆优化Dijkstra
  - Bellman-Ford （O(V*E)）（可以处理负权，但不能有负权回路）
  - SPFA链式前向星(可能被卡时长)
  - Floyd （需要没有权值小于0的回路）
- 图的连通性
  - 求联通分支数(Tarjan算法) 链接
    （链式前向星版）
- 最短路与联通性结合
- 欧拉回路
  - 判断欧拉回路模板题
- 二分图
  - 判断是否是二分图
  - 匈牙利算法 O(V*E)
  - HK算法 O(sqrt(V)*E)
  - KM算法(二分图带权最优匹配)(O(n^3))
- 网络流
  - EK算法（最多增广次数为$O(VE)$）
  - DINIC算法
- 带花树 O(n^3)
- 差分约束
  - 最小值问题
  - 不等式的标准化
- 哈密顿回路

小结论

图论模板

并查集操作

```
int father[1001];
//并查集找父亲的操作
int findFather(int x){
    while(x!=father[x]){
        x = father[x];
    }
    return x;
}
//更新的并查集查找
int find(int x)
{
    if(x==fa[x])
        return x;
    return fa[x]=find(fa[x]);
}
//并查集合并的操作
void Union(int a, int b){
    int af = findFather(a);
    int bf = findFather(b);
    father[bf] = af;
}
```

bitset求可达性

O（$n^2/64$）

```
#include <iostream>
#include <cstring>
#include <algorithm>
#include <stack>
#include<vector>
#include<cstdio>
#include<bitset>
#define ll long long
using namespace std;
const int maxn=2e3+10;
bitset<maxn> f[maxn]; //相当于邻接矩阵存
int ans;
char s[maxn];
int main()
{
  int n;
```

```
  scanf("%d",&n);
  for(register int i=1;i<=n;i++)
  {
    scanf("%s",s);
    for(int j=1;j<=n;j++)
    {
      if(s[j-1]=='1'||i==j)
        f[i][j]=1;
    }
  }
  for(register int i=1;i<=n;i++)
    for(register int j=1;j<=n;j++)
  {
    if(f[j].test(i))
      f[j]|=f[i];
    //如果j可以到达i,那么点i能到的点肯定能被点j所达到
  }
  ll ans=0;
  for(register int i=1;i<=n;i++)
    ans+=f[i].count();
  cout<<ans;
}
```

最小生成树

Kruskal O(elog(e))

为完全图时，边数有n^2个，复杂度为$O(n^2logn)$

```
int n,r;
int fa[maxn];
int find(int x)
{
  if(fa[x]==x)return x;
  return fa[x]=find(fa[x]);
}
void merge(int a,int b)
{
  fa[find(a)]=find(b);
}

struct edge{
  int u,v,w;
}all[maxn*maxn];

bool cmp(edge a ,edge b)
{
  return a.w<b.w;
}

int Kruskal()
{
  int ans=0;
  cin>>n>>r;  //点数和边数
  for(int i=1;i<=n;i++)
    fa[i]=i;
  for(int i=1;i<=r;i++)
    cin>>all[i].u>>all[i].v>>all[i].w;
  sort(all+1,all+1+r,cmp);
  int now=1;
  for(int _=1;_<=n-1;_++){  //只需选n-1条边
    while(find(all[now].u)==find(all[now].v))now++;
    ans+=all[now].w;
    merge(all[now].u,all[now].v);
  }
  return ans;
}
```

最短路

Dijkstra

$O(n^2+e)$（需要所有边均为正权）

```
const int N=110;
const int INF=0x3f3f3f3f;
int Map[N][N];  //邻接矩阵存储,需要将不通的路置为INF(!!!需要初始化为INF)
bool vis[N];
int dj[N];
void Dijkstra() //有暇疵,仅计算从0点出发的最短路径长,最终结果保存于dj数组
{
    memset(vis,0,sizeof(vis));
    for(int i=0;i<=n;++i)
        dj[i] = Map[0][i];
    vis[0] = 1;
    for(int i=1;i<=n;++i)
    {
        int mindj = INF;
        int pos;
        for(int j=1;j<=n;++j)
        {
            if(dj[j]<mindj&&!vis[j])
            {
                mindj = dj[j];
                pos = j;
            }
        }
        vis[pos] = 1;
        for(int j=1;j<=n;++j)
        {
            if(!vis[j] && dj[j] > dj[pos] + Map[pos][j])
                dj[j] = dj[pos] + Map[pos][j];
        }
    }
}
```

堆优化Dijkstra

O(E*log(V))

```
//堆优化，优先级队列 复杂度O(E*log(E))
#include<iostream>
#include<vector>
#include<queue>
#include<cstdio>
#include<cstring>
```

```cpp
#define ll long long
const ll N=110;
const ll inf=0x3f3f3f3f;
using namespace std;
struct node
{
    ll to,w;
    bool operator <(const node & a) const {return w>a.w;}
    //优先队列默认按降序排列，因此要使小权重路线的优先级高
};
vector<node> adja[N]; //邻接表存储，注意存储单向边还是双向边
bool vis[N];
ll dis[N];
ll Dijkstra(ll start,ll end)
{
    memset(vis, false, sizeof(vis));
    for(ll i=0;i<N;i++)
    {
        dis[i]=inf;
    }
    priority_queue<node> q;
    dis[start]=0;
    q.push({start,0});
    while(!q.empty())
    {
        node now=q.top();
        q.pop();
        ll mid=now.to;
        if(vis[mid])
            continue;
        vis[mid]=true;
        ll len=adja[mid].size();
        for(ll i=0;i<len;i++)
        {
            node tar=adja[mid][i];
            ll ter=tar.to;
            ll cost=tar.w;
            if(!vis[ter]&&dis[ter]>dis[mid]+cost)
            {
                dis[ter]=dis[mid]+cost;
                q.push({ter,dis[ter]});
            }
        }
    }
    return dis[end];
}
```

Bellman-Ford （O(V*E)）（可以处理负权，但不能有负权回路）

```
#include<cstring>
#define ll long long
const ll N=110,M=10010;
const ll inf=0x3f3f3f3f;
using namespace std;
struct Edge
{
    ll from,to,weight;
}E[M];  //位置0不使用
ll dis[N];
ll nodenum,edgenum;
void init(ll start)
{
    for(int i=1;i<=nodenum;i++)
    {
        dis[i]=inf;
    }
    for(ll i=1;i<=edgenum;i++)    //该处初始化依照情况，可调整至输入时初始化
    {
        if(E[i].from==start)
            dis[E[i].to]=E[i].weight;
    }
    dis[start]=0;
}
void relax(ll from,ll to,ll weight)
{
    if(dis[to]>dis[from]+weight)
        dis[to]=dis[from]+weight;
}
ll bellman_ford(ll start,ll end)
{
    init(start);
    for(ll i=1;i<nodenum;i++)    //V-1次
    for(ll j=1;j<=edgenum;j++)
    {
        relax(E[j].from,E[j].to,E[j].weight);
    }
    for(ll i=1;i<=edgenum;i++)
    {
        if(dis[E[i].to]>dis[E[i].from]+E[i].weight)
            return -1;
    }
    return dis[end];
}
```

SPFA链式前向星(可能被卡时长)

```
#include <algorithm>
#include <iostream>
#include <cstring>
#include <vector>
#include <cstdio>
#include <queue>
#include <cmath>
#define LL long long
using namespace std;

const int N=1e6+100;
const LL M=(1<<31)-1,MM=0x3f;
struct sb
{
  int uu,ww,to,next;
}a[N];  //数组的大小需要开2*N
int u,v,w,n,m,s,t,tot,f[N],dis[N],last[N];
queue <int> q;
void add (int u,int v,int w)
{
  a[++tot].ww=w;
  a[tot].to=v;
  a[tot].next=last[u];
  last[u]=tot;
}
void spfa ()
{
    memset (dis,0x3f,sizeof (dis));
    dis[s]=0;
    q.push (s);
  while (!q.empty ())
  {
    int h=q.front ();
    q.pop ();
    for (int i=last[h];i;i=a[i].next)
    {
      int fg=a[i].to;
      if (dis[fg]>dis[h]+a[i].ww)
      {
        dis[fg]=dis[h]+a[i].ww;
        q.push (fg);
      }
    }
  }
}

int main ()
```

```
{
  scanf ("%d %d %d",&m,&n,&s);
  for (int i=1;i<=n;i++) scanf ("%d %d %d",&u,&v,&w),add (u,v,w);//add (v,u,w);
  spfa ();
  for (int i=1;i<=m;i++)
  {
    if (dis[i]>N) printf ("%d ",M);
    else printf ("%d ",dis[i]);
  }
  return 0;
}
```

Floyd （需要没有权值小于0的回路）

```
#include <stdio.h>
#define inf 0x3f3f3f3f
int map[1000][1000];    //邻接矩阵
int main()
{
    int k,i,j,n,m;
    //读入n和m，n表示顶点个数，m表示边的条数
    scanf("%d %d",&n,&m);

    //初始化
    for(i=1; i<=n; i++)
        for(j=1; j<=n; j++)
            if(i==j)
                map[i][j]=0;
            else
                map[i][j]=inf;
    int a,b,c;
    //读入边
    for(i=1; i<=m; i++)
    {
        scanf("%d %d %d",&a,&b,&c);
        map[a][b]=c;//这是一个有向图
    }

    //Floyd-Warshall算法核心语句
    for(k=1; k<=n; k++)
        for(i=1; i<=n; i++)
            for(j=1; j<=n; j++)
                if(map[i][j]>map[i][k]+map[k][j] )
                    map[i][j]=map[i][k]+map[k][j];

    //输出最终的结果,最终二维数组中存的即使两点之间的最短距离
    for(i=1; i<=n; i++)
    {
```

```
        for(j=1; j<=n; j++)
        {
            printf("%10d",map[i][j]);
        }
        printf("\n");
    }
    return 0;
}
```

图的连通性

求联通分支数(Tarjan算法) 链接

题目传送门（样例数据是错的）

> 其他要点：添加最少的边使图变成强联通，

> 边数=强联通分量缩点后出度为0的点数与入读为0的点数的较大值

> 一个出度为0的点和入度为0的点之间一定能添加一条边，使这两个点联通

> 即一个出度为0的点和入度为0的点能相消，相消后多出来的那部分点不联通，因此得再多加边，使这些点与其他点联通

```
#include <iostream>
#include <cstring>
#include <algorithm>
#include <stack>
#include<vector>
using namespace std;
const int maxn=3e5+10;
vector<int> G[maxn];
stack<int> s;
int n,sum;
int tot;
bool on_stack[maxn];     //判断是否在栈
int dfn[maxn],low_link[maxn];
int sz[maxn],belong[maxn];  //sz为联通分支的点个数，belong为n号点归属于哪个联通分支
int out[maxn],in[maxn];    //统计联通分支缩点后的入度和出度
void tarjan(int u)
{
  dfn[u]=low_link[u]=++sum;
  s.push(u);
  on_stack[u]=1;
  int len=G[u].size();
  for(int i=0;i<len;i++)
  {
    int v=G[u][i];
    if(!dfn[v])
    {
      tarjan(v);
      low_link[u]=min(low_link[v],low_link[u]);
    }
    else if(on_stack[v])
    {
      low_link[u]=min(low_link[v],low_link[u]);
    }
  }
```

```
  if(low_link[u]==dfn[u])
  {
    int y=-99999;
    tot++;
    do
    {
      y=s.top();
      s.pop();
      belong[y]=tot;
      sz[tot]++;
      on_stack[y]=0;
    }while(y!=u);
  }
}
pair<int,int> E[maxn];
int main()
{
  int m;
  cin>>n>>m;
  for(int i=0;i<m;i++)
  {
    int from,to;
    cin>>from>>to;
    G[from].push_back(to);
    E[i]=make_pair(from, to);
  }
  for(int i=1;i<=n;i++)
    if(!dfn[i])      //遍历图，保证每个联通分支都dfs到
      tarjan(i);
  for(int i=0;i<m;i++)
  {
      int bu=belong[E[i].first],bv=belong[E[i].second];
      if(bu!=bv)
      {
        out[bu]++;
        in[bv]++;
      }
  }
  int ans1=0,ans2=0;
  for(int i=1;i<=tot;i++)
  {
    if(!in[i]) ans1++;
    if(!out[i]) ans2++;
  }
  cout<<*max_element(sz+1,sz+tot+1)<<'\n'<<max(ans1,ans2);
}
```

（链式前向星版）

```
#include<cstdio>
#include<cstring>
#include<iostream>
#include<algorithm>
using namespace std;
#define M 500010
inline int Readint(void){
  int s=0,f=1;char c=getchar();
  while(!isdigit(c)){if(c=='-')f=-1;c=getchar();}
  while(isdigit(c)){s=s*10+c-48;c=getchar();}
  return s*f;
}
struct Node{
  int nex,u,v,w;
}Edge[M];
int n,m,u,v;
int head[M],cnt;
int stack[M],top;
int belong[M],idx,size[M];
int dfn[M],low[M],tot;
int in[M],out[M];
int Max,ans1,ans2;
void addEdge(int u,int v,int w){
  Edge[++cnt]=(Node){head[u],u,v,w};
  head[u]=cnt;
  return;
}
void Tarjan(int u){
  dfn[u]=low[u]=++tot;
  stack[++top]=u;
  for(int i=head[u];i;i=Edge[i].nex){
    int v=Edge[i].v;
    if(!dfn[v]){
      Tarjan(v);
      low[u]=min(low[u],low[v]);
    }
    else if(!belong[v]) low[u]=min(low[u],dfn[v]);
  }
  if(dfn[u]==low[u]){
    idx++;
    while(stack[top+1]!=u){
      int v=stack[top--];
      belong[v]=idx;
      size[idx]++;
      Max=max(Max,size[idx]);
    }
  }
```

```
  return;
}
signed main(void){
  n=Readint();m=Readint();
  for(int i=1;i<=m;i++){
    u=Readint();v=Readint();
    addEdge(u,v,1);
  }
  for(int i=1;i<=n;i++)
    if(!dfn[i])
      Tarjan(i);
  printf("%d\n",Max);
  for(int i=1;i<=m;i++){
    int Bu=belong[Edge[i].u],Bv=belong[Edge[i].v];
    if(Bu!=Bv){
      out[Bu]++;
      in[Bv]++;
    }
  }
  for(int i=1;i<=idx;i++){
    if(!in[i]) ans1++;
    if(!out[i]) ans2++;
  }
  printf("%d\n",max(ans1,ans2));
  return 0;
}
```

最短路与联通性结合

题目描述 链接

小胖和ZYR要去ESQMS森林采蘑菇。

ESQMS森林间有N个小树丛，M条小径，每条小径都是单向的，连接两个小树丛，上面都有一定数量的蘑菇。小胖和ZYR经过某条小径一次，可以采走这条路上所有的蘑菇。由于ESQMS森林是一片神奇的沃土，所以一条路上的蘑菇被采过后，又会长出一些新的蘑菇，数量为原来蘑菇的数量乘上这条路的“恢复系数”，再下取整。

比如，一条路上有4个蘑菇，这条路的“恢复系数”为0.7，则第一~四次经过这条路径所能采到的蘑菇数量分别为4,2,1,0.

现在，小胖和ZYR从S号小树丛出发，求他们最多能采到多少蘑菇。

对于30%的数据，N<=7，M<=15

另有30%的数据，满足所有“恢复系数”为0

对于100%的数据，N<=80,000，M<=200,000，0.1<=恢复系数<=0.8且仅有一位小数，1<=S<=N.

输入格式

第一行，N和M

第2……M+1行，每行4个数字，分别表示一条小路的起点，终点，初始蘑菇数，恢复系数。

第M+2行，一个数字S

输出格式

一个数字，表示最多能采到多少蘑菇，在int32范围内。

做法其一 （Tarjan缩点+SPFA求最长路）

> 边权乘-1后跑SPFA求最短路，最终结果再乘-1可得最长路

```
#include<iostream>
#include<vector>
#include<queue>
#define ll long long
using namespace std;
const int maxn=2e5+10;
const int inf=0x3f3f3f3f;
struct edge
{
  int u,v,w,nex;
}E[maxn],E2[maxn];
int cnt,cnt2,head[maxn],head2[maxn];
void add(int u,int v,int w)
{
  E[++cnt]={u,v,w,head[u]};
  head[u]=cnt;
  return;
```

```
}
void add2(int u,int v,int w)
{
  E2[++cnt2]={u,v,w,head2[u]};
  head2[u]=cnt2;
  return;
}

int dfn[maxn],low[maxn],top,stack[maxn],tot;
int belong[maxn],idx,size[maxn];
void Tarjan(int u){
  dfn[u]=low[u]=++tot;
  stack[++top]=u;
  for(int i=head[u];i;i=E[i].nex){
    int v=E[i].v;
    if(!dfn[v]){
      Tarjan(v);
      low[u]=min(low[u],low[v]);
    }
    else if(!belong[v]) low[u]=min(low[u],dfn[v]);
  }
  if(dfn[u]==low[u]){
    idx++;
    while(stack[top+1]!=u){
      int v=stack[top--];
      belong[v]=idx;
      size[idx]++;
    }
  }
  return;
}

int dis[maxn];
bool vis[maxn];
double coef[maxn];

int ans=inf;
int point[maxn];

int main()
{
  int n,m;
  cin>>n>>m;
  for(int i=1;i<=m;i++)
  {
    int from,to,w;
    cin>>from>>to>>w>>coef[i];
    add(from,to,w);
  }
```

```
  int st;
  cin>>st;
  for(int i=1;i<=n;i++)
  {
    if(!dfn[i])
      Tarjan(i);
  }

  for(int i=1;i<=m;i++)
  {
    int bu=belong[E[i].u],bv=belong[E[i].v];
    if(bu==bv)
    {
      int num=E[i].w;
      while(num!=0)
      {
        point[bu]+=num;
        num=(int)(num*coef[i]);
      }
    }
  }

//    for(int i=1;i<=tot;i++)
//      cout<<belong[i]<<" ";
//    cout<<endl;

  for(int i=1;i<=m;i++)
    {
      int bu=belong[E[i].u],bv=belong[E[i].v];
      if(bu!=bv)
        {
          add2(bu,bv,-(point[bv]+E[i].w));
        }
    }


  for(int i=0;i<maxn;i++)
  {
    dis[i]=inf;
  }

  queue<int> q;
  q.push(belong[st]);
  dis[belong[st]]=-point[belong[st]];
  while (!q.empty()){
    int i=head2[q.front()];
    while (i){
      if (dis[E2[i].v]>dis[q.front()]+E2[i].w){
        dis[E2[i].v]=dis[q.front()]+E2[i].w;
```

```
        if (!vis[E2[i].v]){
          q.push(E2[i].v);
          vis[E2[i].v]=true;
        }
      }
      i=E2[i].nex;
    }
    vis[q.front()]=false;
    q.pop();
  }


  for(int i=1;i<=tot;i++)
  {
    if(dis[i]!=inf)
    ans=min(ans,dis[i]);
  }
  ans*=-1;
  cout<<ans;
}
```

欧拉回路

欧拉回路：图G，若存在一条路，经过G中每条边有且仅有一次，称这条路为欧拉路，

如果存在一条回路经过G每条边有且仅有一次，称这条回路为欧拉回路。具有欧拉回路的图成为欧拉图。

判断欧拉路是否存在的方法

有向图：图连通，有一个顶点出度大入度1，有一个顶点入度大出度1，其余都是出度=入度。

无向图：图连通，只有两个顶点是奇数度，其余都是偶数度的。

判断欧拉回路是否存在的方法

有向图：图连通，所有的顶点出度=入度。

无向图：图连通，所有顶点都是偶数度。

（不能有多个连通分量，但是孤立点可以有）

```
//求欧拉回路或欧拉路，邻接阵形式，复杂度o（n^2）
//返回路径长度，path返回路径(有向图是得到的是反向路径)
//传入图的大小n和邻接阵mat，不相交邻点边权0
//可以有自环与重边，分为无向图和有向图
#define MAXN 100
//u为无向图
void find_path_u(int n,int mat[][MAXN],int now,int& step,int* path){
    int i;
    for (i=n-1;i>=0;i--)
        while (mat[now][i])       {
         mat[now][i]--,mat[i][now]--;
         find_path_u(n,mat,i,step,path);
        }
    path[step++]=now;
}
//d为有向图
void find_path_d(int n,int mat[][MAXN],int now,int& step,int* path){
    int i;
    for (i=n-1;i>=0;i--)
        while (mat[now][i])       {
         mat[now][i]--;
         find_path_d(n,mat,i,step,path);
        }
    path[step++]=now;
}
int euclid_path(int n,int mat[][MAXN],int start,int* path){
    int ret=0;
    find_path_u(n,mat,start,ret,path);
    // find_path_d(n,mat,start,ret,path);
    return ret;
}
```

判断欧拉回路模板题

```
//确定无向图欧拉回路的充要条件：除孤立节点外，其它节点满足 1.连通 2.度为偶数

#include <cstdio>
#include <algorithm>

using namespace std;

int father[1001];
//并查集找父亲的操作
int findFather(int x){
    while(x!=father[x]){
        x = father[x];
    }
    return x;
}
//并查集合并的操作
void Union(int a, int b){
    int af = findFather(a);
    int bf = findFather(b);
    father[bf] = af;
}

int main(){
    int n,m;
    while(scanf("%d%d",&n,&m)!=EOF){
        int d[1001]; //节点度
        fill(d,d+1001,0);
        for(int i=0;i<=n;i++) father[i]=i; //初始化father数组
        for(int i=0;i<m;i++){
            int a,b;
            scanf("%d%d",&a,&b);
            d[a]++;
            d[b]++;
            Union(a,b);
        }
        int tmp=0;
        for(int i=1;i<=n;i++){
            //有奇数度，应打印0
            if(d[i]%2!=0){
                tmp++;
                break;
            }
        }
        if(tmp>0){
            printf("0\n");
            continue;
        }
```

```
        int t = 1;
        for(int i=0;i<=n;i++){  //寻找一个非孤立节点，存入t
            if(d[i]!=0){
                t = i;
                break;
            }
        }
        int f = findFather(t);
        bool flag = false;
        for(int i=2;i<=n;i++){
            //既不是孤立节点，也不连通，应打印0
            if(findFather(i)!=f && findFather(i)!=i){
                flag = true;
                break;
            }
        }
        if(flag){
            printf("0\n");
            continue;
        }
        if(n!=0){
            printf("1\n");
        }

    }
    return 0;
}
```

二分图

判断是否是二分图

```
#include<iostream>
#include<cmath>
#include<vector>
#include<cstdio>
#include<queue>
#include<cstring>
#include<algorithm>
using namespace std;
vector<int> G[100];
int color[100]; //染色bfs
bool bfs(int st)
{
    queue<int> q;
    q.push(st);
    while(!q.empty())
    {
        int now=q.front();
        q.pop();
        if(color[now]==1)
        {
            int len=G[now].size();
            for(int i=0;i<len;i++)
            {
                if(color[G[now][i]]==1)
                    return 0;
                if(!color[G[now][i]])
                {
                    color[G[now][i]]=2;
                    q.push(G[now][i]);
                    continue;
                }
            }
        }
        if(color[now]==2)
        {
            int len=G[now].size();
            for(int i=0;i<len;i++)
            {
                if(color[G[now][i]]==2)
                    return 0;
                if(!color[G[now][i]])
                {
                    color[G[now][i]]=1;
                    q.push(G[now][i]);
```

```
                    continue;
                }
            }
        }
        if(color[now]==0)
        {
            color[now]=1;
            q.push(now);
        }
    }
    return 1;
}
int main()
{
    int n,m;
    while(cin>>n>>m)
    {
        memset(color,0,sizeof(color));
        for(int i=0;i<=n;i++)
        {
            G[i].clear();
        }
        for(int i=0;i<m;i++)
        {
            int to,from;
            cin>>to>>from;
            G[to].push_back(from);
            G[from].push_back(to);
        }
        int ans=0;
        for(int i=1;i<=n;i++)
        {
            if(!bfs(i))
            {
                ans=1;
                break;
            }
        }
        if(ans)
        {
            cout<<"No\n";
        }
        else cout<<"Yes\n";
    }
}
```

匈牙利算法 O(V*E)

```
#include<iostream>           //最小点覆盖==最大匹配
#include<cstring>          //最大独立集==点的总数 - 最小点覆盖。
using namespace std;
const int maxn=510,maxm=510;     //左右点集中点的数量
bool G[maxn][maxm];          //只需建G[x][y]=1,从0开始存
int x[maxn],y[maxm];
bool chk[maxm];
int SearchPath(int k)
{
    for(int i=0;i<maxm;i++)
    {
        if(G[k][i]&&!chk[i])
        {
            chk[i]=1;
            if(y[i]==-1||SearchPath(y[i]))
            {
                y[i]=k;
                x[k]=i;
                return 1;
            }
        }
    }
    return 0;
}
int hungarian()
{
    int ans=0;
    memset(x, -1, sizeof(x));
    memset(y, -1, sizeof(y));
    for(int i=0;i<maxn;i++)
    {
        memset(chk, false , sizeof(chk));
        ans+=SearchPath(i);
    }
    return ans;
}
```

HK算法 O(sqrt(V)*E)

```
#include<cstdio>
#include<cstring>
#include<queue>
#define INF 0x3f3f3f3f
using namespace std;
```

```
const int MAXN = 100+5;
const int MAXM = 300+5;

int p, n;
int a[MAXN][MAXM];

int dis;
int cx[MAXN], cy[MAXM];
int dx[MAXN], dy[MAXM];
bool vis[MAXM];

bool bfs_findPath() {
     queue<int> q;
     memset(dx, -1, sizeof(dx));
     memset(dy, -1, sizeof(dy));
     // 使用BFS遍历对图的点进行分层，从X中找出一个未匹配点v
     // (所有v)组成第一层，接下来的层都是这样形成——每次查找
     // 匹配点(增广路性质)，直到在Y中找到未匹配点才停止查找，
     // 对X其他未匹配点同样进行查找增广路径(BFS只分层不标记
     // 是否匹配点)
     // 找出X中的所有未匹配点组成BFS的第一层
     dis = INF;
     for(int i = 1; i <= p; ++i) {
         if(cx[i] == -1) {
             q.push(i);
             dx[i] = 0;
         }
     }
     while(!q.empty()) {
         int u = q.front();
         q.pop();
         if(dx[u] > dis) break;// 该路径长度大于dis，等待下一次BFS扩充
         for(int v = 1; v <= n; ++v) {
             if(a[u][v] && dy[v] == -1) {// (u,v)之间有边且v还没有分层
                 dy[v] = dx[u] + 1;
                 if(cy[v] == -1) dis = dy[v];// v是未匹配点，停止延伸（查找）,得到本次BFS的最
大遍历层次
                 else {// v是已匹配点，继续延伸
                     dx[cy[v]] = dy[v] + 1;
                     q.push(cy[v]);
                 }
             }
         }
     }
     return dis != INF;// 若dis为INF说明Y中没有未匹配点，也就是没有增广路径了
 }

 bool dfs(int u) {
```

```
    for(int v = 1; v <= n; ++v) {
        if(!vis[v] && a[u][v] && dy[v] == dx[u] + 1) {
            vis[v] = 1;
            // 层次（也就是增广路径的长度）大于本次查找的dis
            // 是bfs中被break的情况，也就是还不确定是否是增广路
            // 只有等再次调用bfs再判断(每次只找最小增广路集)
            if(cy[v] != -1 && dy[v] == dis) continue;
            if(cy[v] == -1 || dfs(cy[v])) {// 是增广路径，更新匹配集
                cy[v] = u;
                cx[u] = v;
                return true;
            }
        }
    }
    return false;
}

int HK() {
    int ans = 0;
    memset(cx, -1, sizeof(cx));
    memset(cy, -1, sizeof(cy));
    while(bfs_findPath()) {// 有增广路
        memset(vis, 0, sizeof(vis));
        for(int i = 1; i <= p; ++i) {
            // 用DFS查找增广路径，增广路径一定从未匹配点开始
            // 如果查找到一个增广路径，匹配数加一
            if(cx[i] == -1 && dfs(i)) ++ans;
        }
    }
    return ans;
}

int main() {
    int T;
    scanf("%d", &T);
    while(T--) {
        scanf("%d%d", &p, &n);
        memset(a, 0, sizeof(a));

        for(int i = 1; i <= p; ++i) {
            int nm, x;
            scanf("%d", &nm);
            for(int j = 0; j != nm; ++j) {
                scanf("%d", &x);
                a[i][x] = 1;
            }
        }
        printf("%s\n", HK() == p ? "YES" : "NO");
    }
```

```
    return 0;
}
```

KM算法(二分图带权最优匹配)(O(n^3))

```
//Data
const int N=500;
int n,m,e[N+7][N+7];

//KM
int mb[N+7],vb[N+7],ka[N+7],kb[N+7],p[N+7],c[N+7];
int qf,qb,q[N+7];
void Bfs(int u){
    int a,v=0,vl=0,d;
    for(int i=1;i<=n;i++) p[i]=0,c[i]=inf;
    mb[v]=u;
    do {
        a=mb[v],d=inf,vb[v]=1;
        for(int b=1;b<=n;b++)if(!vb[b]){
            if(c[b]>ka[a]+kb[b]-e[a][b])
                c[b]=ka[a]+kb[b]-e[a][b],p[b]=v;
            if(c[b]<d) d=c[b],vl=b;
        }
        for(int b=0;b<=n;b++)
            if(vb[b]) ka[mb[b]]-=d,kb[b]+=d;
            else c[b]-=d;
        v=vl;
    } while(mb[v]);
    while(v) mb[v]=mb[p[v]],v=p[v];
}
ll KM(){
    for(int i=1;i<=n;i++) mb[i]=ka[i]=kb[i]=0;
    for(int a=1;a<=n;a++){
      for(int b=1;b<=n;b++) vb[b]=0;
    Bfs(a);
  }
  ll res=0;
  for(int b=1;b<=n;b++) res+=e[mb[b]][b];
  return res;
}

//Main
int main(){
  n=ri,m=ri;
  for(int a=1;a<=n;a++)
    for(int b=1;b<=n;b++) e[a][b]=-inf;
  for(int i=1;i<=m;i++){
    int u=ri,v=ri,w=ri;
```

```
    e[u][v]=max(e[u][v],w);
  }
  printf("%lld\n",KM());
  for(int u=1;u<=n;u++) printf("%d ",mb[u]);puts("");
  return 0;
}
```

以上求的是最大权值匹配

若要求最小权值匹配，可以建负边，然后ans= - KM();

网络流

EK算法（最多增广次数为$O(VE)$）

```
#include <bits/stdc++.h>
//最大流EK算法：复杂度:n*m^2(n是点数,m是边数)
//如果遇到稠密图用Dinic

using namespace std;
const int maxn=220,inf=0x7f7f7f7f;
int g[maxn][maxn],flow[maxn],pre[maxn],n,m;
queue<int> q;

inline int bfs(int s,int t)
{
    while(!q.empty())q.pop();
    memset(pre,-1,sizeof(pre));
    pre[s]=0,flow[s]=inf;
    q.push(s);
    while(!q.empty())
    {
        int p=q.front();q.pop();
        if(p==t)break;
        for(int u=1;u<=n;u++)
        {
            if(u!=s&&g[p][u]>0&&pre[u]==-1)
            {
                pre[u]=p;
                flow[u]=min(flow[p],g[p][u]);
                q.push(u);
            }
        }
    }
    if(pre[t]==-1)return -1;
    return flow[t];
}

inline int ek(int s,int t)
{
    int delta=0,tot=0;
    while(1)
    {
        delta=bfs(s,t);
        if(delta==-1)break;
        int p=t;
        while(p!=s)//更新增广路
        {
            g[pre[p]][p]-=delta;
```

```
            g[p][pre[p]]+=delta;
            p=pre[p];
        }
        tot+=delta;
    }
    return tot;
}

int main()
{
    ios::sync_with_stdio(0);
    int u,v,w;
    scanf("%d %d",&m,&n);
    memset(g,0,sizeof(g));
    memset(flow,0,sizeof(flow));
    for(int i=0;i<m;i++)
    {
        scanf("%d %d %d",&u,&v,&w);
        g[u][v]+=w;
    }
    printf("%d\n",ek(1,n));
    return 0;
}

/*
5 4
1 2 40
1 4 20
2 4 20
2 3 30
3 4 10
50
*/
```

DINIC算法

```
#include <stdio.h>
#include <cstdlib>
#include <cstring>
#include <cmath>
#include <iostream>
#include <algorithm>
#include <vector>
#include <queue>
#define maxn 10010
#define maxm 100010
#define inf 0x7fffffff
```

```cpp
using namespace std;

int n,m,s,t;
int sum,ans;
int d[maxn];

struct edge{
    int to,val,rev; //rev表示反边在to的vector当中下标是几
    edge (int _to,int _val,int _rev) //构造函数
    {
        to=_to;
        val=_val;
        rev=_rev;
    }
};

vector<edge> e[maxn];

void addedge(int x,int y,int w)
{
    e[x].push_back(edge(y,w,e[y].size()));
    e[y].push_back(edge(x,0,e[x].size()-1));
    /*
     或者 也可以这样写
     e[x].push_back((edge){y,w,e[y].size()});
     e[y].push_back((edge){x,0,e[x].size()-1});
    这样写不需要?的构造函数
     */
}

bool bfs()
{
    memset(d, -1, sizeof(d));
    queue<int> q;
    q.push(s);
    d[s]=0;
    while(!q.empty())
    {
        int x=q.front();
        q.pop();
        for(int i=0;i<e[x].size();i++)
        {
            int y=e[x][i].to;
            if(d[y]==-1 && e[x][i].val)
            {
                q.push(y);
                d[y]=d[x]+1;
            }
        }
```

```
    }
    if(d[t]==-1)
        return 0;
    else
        return 1;
}

int dfs(int x,int low) //x表示当前节点，low表示当前流到x的残量
{
    if(x==t || low==0)
        return low;
    int totflow=0; //totflow表示x节点总共流出的流量
    for(int i=0;i<e[x].size();i++)
    {
        int y=e[x][i].to;
        int rev=e[x][i].rev;
        if(d[y]==d[x]+1 && e[x][i].val) //y是x的下一层 且 当前边有残量>0
        {
            int a=dfs(y,min(low,e[x][i].val)); //a表示当前边流出的流量
            e[x][i].val-=a;
            e[y][rev].val+=a;
            low-=a;
            totflow+=a;
            if(low==0) //流到x的流量流完了
                return totflow;
        }
    }
    if(low!=0) //优化，直观理解：流到x的流量会有冗余，这一轮dfs中就再也不到x了
        d[x]=-1;
    return totflow;
}

void dinic()
{
    while(bfs())
        ans+=dfs(s,inf);
}

int main()
{
    scanf("%d%d%d%d",&n,&m,&s,&t);
    for(int i=1;i<=m;i++)
    {
        int x,y,w;
        scanf("%d%d%d",&x,&y,&w);
        addedge(x, y, w);
    }
    dinic();
    printf("%d\n",ans);
```

```
    return 0;
}
```

带花树 O(n^3)

```
#include <bits/stdc++.h>
#ifndef ONLINE_JUDGE
#define debug(x) cout << #x << ": " << (x) << endl
#else
#define debug(x)
#endif
using namespace std;
typedef long long ll;
typedef vector<int> vi;
const int maxn=5e3+7,inf=0x3f3f3f3f,mod=1e9+7;

vector<int>G[maxn];
namespace Blossom
{ ///0~n-1
    const int N=1000+5;
    bool ban[N];
    int mate[N],n,ret;
    int nxt[N],dsu[N],mark[N],vis[N];
    queue<int> Q;
    int get(int x)
    {
        return (x==dsu[x]) ? x: (dsu[x]=get(dsu[x]));
    }
    void merge(int a,int b)
    {
        dsu[get(a)]=get(b);
    }
    int lca(int x,int y)
    {
        static int t=0;
        ++t;
        for(;;swap(x,y))
            if(x!=-1)
            {
                if(vis[x=get(x)]==t) return x;
                vis[x]=t;
                x=(mate[x]!=-1)?nxt[mate[x]]:-1;
            }
    }
    void group(int a,int p)
    {
        for(int b,c;a!=p;merge(a,b),merge(b,c),a=c)
        {
```

```
            b=mate[a],c=nxt[b];
            if(get(c)!=p)nxt[c]=b;
            if(mark[b]==2)mark[b]=1,Q.push(b);
            if(mark[c]==2)mark[c]=1,Q.push(c);
        }
    }
    void aug(int s,const vector<int> G[]) ///start from s ,do augment
    {
        for(int i=0;i<n;++i) nxt[i]=vis[i]=-1,dsu[i]=i,mark[i]=0;
        while(!Q.empty()) Q.pop();
        Q.push(s);
        mark[s]=1;
        while(mate[s]==-1 && !Q.empty())
        {
            int x=Q.front(); Q.pop();
            for(auto &y:G[x])
            if(!ban[y] && y!=mate[x]&& get(x)!=get(y) && mark[y]!=2)
            {
                if(mark[y]==1)
                {
                    int p=lca(x,y);
                    if(get(x)!=p) nxt[x]=y;
                    if(get(y)!=p) nxt[y]=x;
                    group(x,p),group(y,p);
                }
                else if(mate[y]==-1)
                {
                    nxt[y]=x;
                    for(int j=y,k,l;j!=-1;j=l) k=nxt[j],l=mate[k],mate[j]=k,mate[k]=j;
                    break;
                }
                else nxt[y]=x,Q.push(mate[y]),mark[mate[y]]=1,mark[y]=2;
            }
        }
    }
    int solve(int n,const vector<int> G[])
    {
        for(int i=0;i<n;++i) mate[i]=-1;
        for(int i=0;i<n;++i) if(mate[i]==-1) aug(i,G);
        int ans=0;
        for(int i=0;i<n;++i) if(mate[i]!=-1) ans++;
        return ans;
    }
};

int tot;
map<int,int>prime;
int getid(int x)
{
```

```
    if(x==1) return tot++;
    if(prime.count(x)) return prime[x];
    return prime[x]=tot++;
}

int main()
{
    ios::sync_with_stdio(false);
    cin.tie(0);
    int n;
    cin>>n;
    for(int i=0,x;i<n;++i)
    {
        cin>>x;
        bool f=0;
        for(int j=2;j*j<=x;++j)
        {
            if(x%j==0)
            {
                int a=getid(j),b=getid(x/j);
                if(a==b) b=getid(1);
                G[a].push_back(b),G[b].push_back(a);
                f=1;
                break;
            }
        }
        if(!f)
        {
            int a=getid(x),b=getid(1);
            G[a].push_back(b),G[b].push_back(a);
        }
    }
    Blossom::n=tot;
    int res=Blossom::solve(tot,G)/2;
    cout<<res<<'\n';
    return 0;
}
```

差分约束

给定n个变量和m个不等式，每个不等式形如 x[i] - x[j] <= a[k]，求 x[n-1] - x[0] 的最大值。
(0 <= i, j < n)
例：n=4
x1−x0<=2①
x2−x0<=7②
x3−x0<=8③
x2−x1<=3④
x3−x2<=2⑤

然后经过认真的瞎搞计算就变成了这个鸭子：
1、③　　x3−x0<=8
2、②+⑤　　x3−x0<=9
3、①+④+⑤　　x3−x0<=7

对于每个不等式 xi−xj<=ak，对结点 j 和 i 建立一条 j−>i的有向边，
边权为ak，求x_{n-1}−x0 的最大值就是求 0 到n−1的最短路。

差分约束系统中的每个约束条件 xi-xj<=ck 都可以变形成 xi<=xj+ck ，这与单源最短路中的三角形不等式 dist[y]<=dist[x]+z (x为出发点，经过z到达y)非常相似。因此，我们可以把每个变量 xi 看做图中的一个结点，对于每个约束条件 xi-xj<=ck，从结点 j 向结点 i 连一条长度为 ck 的**有向边**。

注意到，如果 {a1,a2,...,an} 是该差分约束系统的一组解，那么对于任意的常数 d ，{a1+d,a2+d,...,an+d} 显然也是该差分约束系统的一组解，因为这样做差后 d 刚好被消掉。

设 dist[0]=0 并向每一个点连一条权重为 0 边，跑单源最短路，若图中存在负环，则给定的差分约束系统无解，否则，xi=dist[i] 为该差分约束系统的一组解。

一般使用 Bellman-Ford 或队列优化的 Bellman-Ford（俗称 SPFA，在某些随机图跑得很快）判断图中是否存在负环，最坏时间复杂度为 O(nm)。 （n个变量，即n个点，m个约束条件，即m条边）

最小值问题

如果我们把不等式的符号改变一下那么求得最短路那不就是变成了最长路问题吗，
那么我们在做题的时候只需要把spfa中的松弛操作的符号变一下就行了....

不等式的标准化

> 1.如果给出的不等式有"<="也有">="：

如果需要求的是两个变量差的最大值，那么需要将所有不等式转变成"<="的形式，建图后求最短路；
相反，如果需要求的是两个变量差的最小值，那么需要将所有不等式转化成">="，建图后求最长路。

> 2.如果有形如：A - B = c 这样的等式呢?

我们可以将它转化成以下两个不等式：
A−B>=c ①
A−B<=c ②
再通过上面的方法将其中一种不等号反向，建图即可。

> 3.如果这些变量都是整数域上的，那么遇到A - B < c这样的不带等号的不等式怎么办呢?

我们可以将它转化成"<="或者">="的形式，即 A−B<=c−1形式就可以做了...

例题

哈密顿回路

```
#include<stdio.h>
```

```
#define V 5

void printSolution(int path[]);

/* A utility function to check if the vertex v can be added at
   index 'pos' in the Hamiltonian Cycle constructed so far (stored
   in 'path[]')

用来检测当前顶点个数为pos，走到顶点v是否可以加入当前回路中
*/
bool isSafe(int v, bool graph[V][V], int path[], int pos)
{
    /* Check if this vertex is an adjacent vertex of the previously
       added vertex.
       上一次加入的顶点和这个顶点v是否相连
     */
    if (graph [ path[pos-1] ][ v ] == 0)
        return false;

    /* Check if the vertex has already been included.
      This step can be optimized by creating an array of size V
      判断顶点是否已经在回路中，这里可以用一个visited数组来优化。
      */
    for (int i = 0; i < pos; i++)
        if (path[i] == v)
            return false;

    return true;
}

/* A recursive utility function to solve hamiltonian cycle problem

*/
bool hamCycleUtil(bool graph[V][V], int path[], int pos)
{
    /* base case: If all vertices are included in Hamiltonian Cycle
    顶点数够了
    */
    if (pos == V)
    {
        // And if there is an edge from the last included vertex to the
        // first vertex
        //看回路第一个顶点和最后一个顶点是否相连，相连则满足条件
        if ( graph[ path[pos-1] ][ path[0] ] == 1 )
           return true;
        else
          return false;
    }
```

```
    // Try different vertices as a next candidate in Hamiltonian Cycle.
    // We don't try for 0 as we included 0 as starting point in in hamCycle()

    //0作为回路初始顶点
    for (int v = 1; v < V; v++)
    {
        /* Check if this vertex can be added to Hamiltonian Cycle */

        //判断1到v顶点是否可以加入回路
        if (isSafe(v, graph, path, pos))
        {
            path[pos] = v;  //可以，就加入回路

            /* recur to construct rest of the path */
            if (hamCycleUtil (graph, path, pos+1) == true)  //递归剩下的路径
                return true;

            /* If adding vertex v doesn't lead to a solution,
               then remove it
                如果顶点v加入后不构成回路，则弹出来。
            */
            path[pos] = -1;
        }
    }

    /* If no vertex can be added to Hamiltonian Cycle constructed so far,
       then return false */
    return false;
}

/* This function solves the Hamiltonian Cycle problem using Backtracking.
  It mainly uses hamCycleUtil() to solve the problem. It returns false
  if there is no Hamiltonian Cycle possible, otherwise return true and
  prints the path. Please note that there may be more than one solutions,
  this function prints one of the feasible solutions. */

/*回溯法求汉密尔顿回路，如果不存在返回false，如果存在返回true并打印回路，可能有多重方案，只打印一种*/
bool hamCycle(bool graph[V][V])
{
    int *path = new int[V];
    for (int i = 0; i < V; i++)
        path[i] = -1; //初始化path数组

    /* Let us put vertex 0 as the first vertex in the path. If there is
       a Hamiltonian Cycle, then the path can be started from any point
       of the cycle as the graph is undirected */
    path[0] = 0; //加入0号顶点
    if ( hamCycleUtil(graph, path, 1) == false )
```

```
    {
        printf("\nSolution does not exist");
        return false;
    }

    printSolution(path);
    return true;
}

/* A utility function to print solution
   打印此回路
*/

void printSolution(int path[])
{
    printf ("Solution Exists:"
            " Following is one Hamiltonian Cycle \n");
    for (int i = 0; i < V; i++)
        printf(" %d ", path[i]);

    // Let us print the first vertex again to show the complete cycle
    printf(" %d ", path[0]);
    printf("\n");
}

// driver program to test above function
int main()
{
   /* Let us create the following graph
      (0)--(1)--(2)
       |   / \   |
       |  /   \  |
       | /     \ |
      (3)-------(4)    */
   bool graph1[V][V] = {{0, 1, 0, 1, 0},
                        {1, 0, 1, 1, 1},
                        {0, 1, 0, 0, 1},
                        {1, 1, 0, 0, 1},
                        {0, 1, 1, 1, 0},
                       };

    // Print the solution
    hamCycle(graph1);

   /* Let us create the following graph
      (0)--(1)--(2)
       |   / \   |
       |  /   \  |
       | /     \ |
```

```
       (3)       (4)    */
    bool graph2[V][V] = {{0, 1, 0, 1, 0},
                         {1, 0, 1, 1, 1},
                         {0, 1, 0, 0, 1},
                         {1, 1, 0, 0, 0},
                         {0, 1, 1, 0, 0},
                        };

    // Print the solution
    hamCycle(graph2);

    return 0;
}
```

小结论

1. 开队列进行拓扑排序可以去除有向图中的环外节点
2. 霍尔定理：二分图G中的两部分顶点集合分别为X, Y,假设有 |X|<=|Y|。G图存在完美匹配的充分必要条件是：X中的任意k个点至少与Y中的k个点相邻，即对于X中的一个点集W ，令N(W)为W所连的点， 霍尔定理即对于任意W有 |W|<=|N(W)|

霍尔定理推论：二分图的最大匹配为：|X|−max{|W|−|N(W)|}，其中W为X的任意子集。（不一定是完美匹配，并且要把不符合题意的情况排除）