网络流

算法简介

几个概念

容量

```
  每条边$(u,v)$都有一个权值$c(u,v)$□被称为容量，而当边不属于这个图时，容量为0
```

流

流满足以下几个条件：

容量限制：流经过边的流量不能超过该边的容量，即 $f(u,v)≤c(u,v)$

斜对称性：每条边的流量与其相反边的流量之和为0，即 $f(u,v)=-f(v,u)$

流守恒性：从源点流出的流量等于汇点流入的流量

剩余容量

```
  表示一条边的容量与流量之差□ $c_{f}(u,v)=c(u,v)-f(u,v)$
```

残量网络

```
    对于流函数 $f$ □残存网络 $G_{f}$ 是网络 $G$ 中所有结点和剩余容量大于$0$的边构成的子
图。注意，剩余容量大于 0 的边可能不在原图中。可以理解为，残量网络中包括了那些还剩了流量空间
的边构成的图，也包括虚边（即反向边）。
```

增广路

```
   在原图中若一条从源点到汇点的路径上所有边的剩余容量都大于0，这条路叫做增广路。
```

对于网络流，现在主流的有 $EK,Dinic,SAP,ISAP$ 算法

算法思想

我们先设点数为 n □边数为 m □

EK

使用 BFS 进行增广。

具体来说就是从源点一直 BFS 走来走去，碰到汇点就停，然后进行增广，要注意一下流量合不合法。

增广方式：把找到的增广路再走一遍，走的时候把这条路的能够成的最大流量减下去，然后给答案加上最小流量。

增广的时候要注意建造反向边，原因是这条路不一定是最优的，这样子程序可以进行反悔。假如我们对这条路进行增广了，那么其中的每一条边的反向边的流量就是它的流量。

时间复杂度为 $O(nm^{2})$ 效率较低。

$Dinic$

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。

何时停止：如果不存在到汇点的增广路（即汇点的层数不存在），我们即可停止增广。

优化：

多路增广：在一次 DFS 中找出多条增广路。

当前弧优化：因为一条边最多可以被增广一次，所以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。

时间复杂度为 $O(n^{2}m)$ 在稀疏图上效率和 EK 算法相当，但在稠密图上效率要比EK算法高很多。在求解二分图最大匹配问题时Dinic 算法的时间复杂度是 $O(m\sqrt{n})$

我们要确保找到的增广路是最短的，所以我们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）。

```
void addEdge(int i,int a,int b,ll c,int d) {
    u[i]=a;
    v[i]=b;
    w[i]=c;
    rev[i]=d;
    nex[i]=first[a];
    first[a]=i;
}
bool bfs(ll s,ll t) {
    memset(d,0x7fffffff,sizeof(d));
    memset(vis,0,sizeof(vis));
    Q.push(s);
    vis[s]=1;
    d[s]=0;
    while(Q.size()) {
```

```
        int p=Q.front();
        Q.pop();
        for(int i=first[p]; i!=-1; i=nex[i])
            if(!vis[v[i]] && w[i]>0) {
                vis[v[i]]=1;
                d[v[i]]=d[p]+1;
                Q.push(v[i]);
            }
    }
    return vis[t];
}
ll dfs(ll x,ll t,ll a) {
    if(x==t || a==0)return a;
    ll flow=0,f;
    for(ll& i=cur[x]; i!=-1; i=nex[i])
        if(d[x]+1==d[v[i]] && (f=dfs(v[i],t,min(a,w[i])))>0) {
            w[i]-=f;
            w[rev[i]]+=f;
            a-=f;
            flow+=f;
            if(a==0)break;
        }
    return flow;
}
ll Dinic(int s,int t) {
    ll flow=0;
    while(bfs(s,t)) {
        for(int i=1; i<=n; i++)cur[i]=first[i];
        flow+=dfs(s,t,oo);
    }
    return flow;
}



        //加边操作

        for(int i=0; i<m; i++) {
            a=read();
            b=read();
            c=read();
            addEdge(2*i,a,b,c,2*i+1);
            addEdge(2*i+1,b,a,0,2*i);
        }
```

$ISAP$

$Dinic$ 算法中，每次求完增广路之后要跑一遍 BFS 分层，ISAP的策略是在反图中，从 t 到 s 点进行 BFS 。

增广过程和 $Dinic$ 类似，选择比当前点层数少 1 的点来增广，所以也存在当前弧优化。

但不同的是， $ISAP$ 在找增广路的途中会完成下一步的分层，比如现在到了 i 号点，其层数为 d_{i} ，结束这个点的增广过程后，遍历残量网络上 i 的所有出边，找到层最小的出点 j ，然后令 $d_{i}=d_{j}+1$ ，当无出边时，则 $d_{i}=n$ ，则当 $d_{s}≥n$ 时，图上不存在增广路，此时即可终止算法。

$ISAP$ 还存在 GAP 优化，记录层数为 i 的点的数量为 $num[i]$ ，每当将一个点的层数从 x 更新到 y 时，要同时更行 num 数组的值，当某次更新之后，如果 $num[x]==0$ 则图中出现断层，必然找不到增广路，则可以直接终止算法(将 d_{s} 标记为 n)。

时间复杂度与 $Dinic$ 等同，但是实际时间上优于 $Dinic$ 。

```
inline void addedge(int u,int v,int val){
    node[++cnt].v=v;
    node[cnt].val=val;
    node[cnt].next=head[u];
    head[u]=cnt;
}

void bfs(){
    memset(dep,-1,sizeof(dep));
    memset(gap,0,sizeof(gap));
    dep[t]=0;
    gap[0]=1;
    queue<ll>q;
    q.push(t);
    while(!q.empty()){
        int u=q.front();
        q.pop();
        for(int i=head[u];i;i=node[i].next) {
            int v=node[i].v;
            if(dep[v]!=-1) continue;
            q.push(v);
            dep[v]=dep[u]+1;
            gap[dep[v]]++;
        }
    }
    return;
}
long long maxflow;
ll dfs(ll u,ll flow){
    if(u==t){
        maxflow+=flow;
        return flow;
    }
    ll used=0;
    for(int i=cur[u];i;i=node[i].next){
        cur[u]=i;
        ll d=node[i].v;
```

```
        if(node[i].val&&dep[d]+1==dep[u]){
            int mi=dfs(d,min(node[i].val,flow-used));
            if(mi){
                node[i].val-=mi;
                node[i^1].val+=mi;
                used+=mi;
            }
            if(used==flow)return used;
        }
    }
    --gap[dep[u]];
    if(gap[dep[u]]==0)dep[s]=n+1;
    dep[u]++;
    gap[dep[u]]++;
    return used;
}
long long ISAP(){
    maxflow=0;
    bfs();
    while(dep[s]<n)    memcpy(cur,head,sizeof(head)),dfs(s,inf);
    return maxflow;
}



        //加边操作

        for(int i=1;i<=m;i++){
            u=Read();v=Read();w=Read();
            addedge(u,v,w);addedge(v,u,0);
        }
```

预流推进

思想：从源点疯狂“灌水”，水不停流向汇点，能流多少流多少，最后汇点的水的量，就是最大流。

余流：每个点当前有多少水。

步骤：

    假装源点有无限的水，并向周围的点推流，推的流量不能超过自身的余流，也不能超过边的容量，并让周围的点入队(s 和 t 不能入队)。
    不断取队首的元素，对队首元素进行推流。
    队列为空的时候结束算法，此时汇点的余流即为最大流。

但是有可能会存在两个点之间来回推流，导致死循环。

于是我们需要给每个点设置一个高度，让水只从搞出往低处走，在算法进行的时候，不断地对有余流的点更改高度（更新为与它相邻（算反向边）且最低的点的高度+ 1），直到这些点全部没有余流为止。 而此时，因为两个点来回推流的时候，高度会不断上升，而超过 s 的高度之后，他们自动会把流还给 s 这样就不会死循环了，而 s 的初始高度我们设为 n 即可。

奇慢无比，但是是下一个算法的基础。

$HLPP$

步骤：

从 t 到 s 反向 BFS 使得每个点有一个初始高度。

从 s 开始推流，将有余流的点放进优先队列，

不断从优先队列中取出高度最高的点进行推流操作，

如果还有余流，更新高度，重新放入优先队列

优先队列为空时结束算法t的余流是最大流。

为什么相较于普通的预流推进变快了呢？首先我们用BFS预处理了高度，并且因为有优先队列，所以我们可以每次选用高度最高的点，这样大大减少了推流的次数。

优化：

    GAP 优化，同 $ISAP$ 如果某个高度不存在，将所有比该高度高的节点标记为不可到达。（使它的高度为 $n+1$ 这样就会直接向 s 推流了）。

  时间复杂度：理论上是 $O(n^{2}\sqrt{m})$ 但是有较大常数，实际状况下，一般 $ISAP$ 已经足够用。

```
inline void addEdge(const ll u,const ll v,const ll f) {
    a[u].push_back(edge(v,f,a[v].size()));
    a[v].push_back(edge(u,0,a[u].size()-1));
}
inline void relabel(ll n,ll t) {
    h.assign(n,n);
    h[t]=0;
    cnt.assign(n,0);
    que.clear();
    que.resize(n+1);
    ll qh=0,qt=0;
    for(que[qt++]=t; qh<qt;) {
        ll u=que[qh++],het=h[u]+1;
        for(Iterator p=a[u].begin(); p!=a[u].end(); ++p) {
            if(h[p->to]==n&&a[p->to][p->next].flow>0) {
                cnt[h[p->to]=het]++;
                que[qt++]=p->to;
            }
        }
    }
    for(ll i=0; i<=n; ++i) {
        llist[i].clear();
        dlist[i].clear();
```

```
    }
    for(ll u=0; u<n; ++u) {
        if(h[u]<n) {
            iter[u]=dlist[h[u]].insert(dlist[h[u]].begin(),u);
            if(e[u]>0)llist[h[u]].push_back(u);
        }
    }
    hst=(nowh=h[que[qt-1]]);
}
inline void push(ll u,edge &ed) {
    ll v=ed.to;
    ll df=min(e[u],ed.flow);
    ed.flow-=df;
    a[v][ed.next].flow+=df;
    e[u]-=df;
    e[v]+=df;
    if(0<e[v]&&e[v]<=df)llist[h[v]].push_back(v);
}
inline void push(ll n,ll u) {
    ll nh=n;
    for(Iterator p=a[u].begin(); p!=a[u].end(); ++p) {
        if(p->flow>0) {
            if(h[u]==h[p->to]+1) {
                push(u,*p);
                if(e[u]==0)return;
            } else nh=min(nh,h[p->to]+1);
        }
    }
    ll het=h[u];
    if(cnt[het]==1) {
        for(ll i=het; i<=hst; ++i) {
            for(List::iterator it=dlist[i].begin(); it!=dlist[i].end();
++it) {
                cnt[h[*it]]--;
                h[*it]=n;
            }
            dlist[i].clear();
        }
        hst=het-1;
    } else {
        cnt[het]--;
        iter[u]=dlist[het].erase(iter[u]);
        h[u]=nh;
        if(nh==n)return;
        cnt[nh]++;
        iter[u]=dlist[nh].insert(dlist[nh].begin(),u);
        hst=max(hst,nowh=nh);
        llist[nh].push_back(u);
    }
}
inline ll hlpp(ll n,ll s,ll t) {
```

```
    if(s==t)return 0;
    nowh=0;
    hst=0;
    h.assign(n,0);
    h[s]=n;
    iter.resize(n);
    for(ll i=0; i<n;
++i)if(i!=s)iter[i]=dlist[h[i]].insert(dlist[h[i]].begin(),i);
    cnt.assign(n,0);
    cnt[0]=n-1;
    e.assign(n,0);
    e[s]=INF;
    e[t]=-INF;
    for(ll i=0; i<(ll)a[s].size(); ++i)push(s,a[s][i]);
    relabel(n,t);
    for(ll u; nowh>=0;) {
        if(llist[nowh].empty()) {
            nowh--;
            continue;
        }
        u=llist[nowh].back();
        llist[nowh].pop_back();
        push(n,u);
    }
    return e[t]+INF;
}



        //加边操作

        for(register int i=m; i>0; --i) {
            u=read(),v=read(),f=read();
            addEdge(u,v,f);
        }
```

至此洛谷的两道模板题（基础版和加强版）结束

补充：最小割

由最大流最小割定理知，最小割等于最大流，如果求割边数量，则要将每条边的容量变为 1，重新跑最大流模板即可。

经典问题

现在有 n 个物品， 2 个集合 A 和 B 如果将一个物品放入 A 集合产生了 a_{i} 的代价，放入 B 集合会产生 b_{i} 的代价，还有 m 个限制条件形如 u_{i},v_{i},w_{i} 其表示如果

u_{i},v_{i} 不在同一集合，会产生 w_{i} 的代价，每个物品必须且只能属于一个集合，求总共最小代价。

对于放在集合 A 产生的代价，我们可以从源点 s 向这个点连一条容量为 a_{i} 的边，对于放在集合 B 产生的代价，我们可以从这个点向汇点 t 连一条容量为 b_{i} 的边，于是利用最小割的思想，因为一个点只能属于一个集合，所以两条边至少要有一条被选中，不然会有从 s 到 t 的流。

而对于最后一种情况，不妨设 u_{i} 在 A 集合里, v_{i} 在 B 集合里，所以现在是 $s→u_{i} v_{i}→t$ □而想要承担 w_{i} 的代价，我们只需要在 u_{i} , v_{i} 之间连一条 w_{i} 的边，因为另一种情况也有可能，所以这条边要是双向的，于是图就构建完毕了。

代码练习

1.https://www.luogu.com.cn/problem/P1345

给定点总数 n □边总数 m □源点 s □汇点 t □边是双向边，问最少割掉多少个点才能让源点汇点失去联系

很明显是一个最小割，但这里割的是点不是边，于是需要拆点（太长时间不做连基本操作都忘了的 wzb 是屑…），1 到 n 是出点，分别对应坐标 $+n$ 的 $n+1$ 到 $2×n$ 是入点，对于正常的 m 条边，我们从 u 到 $v+n$ 连一条权值为 inf 的边，反向权值为 0。因为是双向的，所以 v 到 $u+n$ 的边也是 inf 的，反向权值为 0。为 inf 的原因是我们不要割掉这种边，我们的目的是割掉入点和出点之间的边，表示割掉这个点。而源点和汇点显然不需要割掉，所以对于其他的点我们都从入点到出点连一条正向为 1 反向为 0 的边。

```
while(~scanf("%d %d %d %d",&n,&m,&s,&t)) {
        cnt=1;
        memset(head,0,sizeof(head));
        ll u,v;
        for(int i=1;i<=n;i++){
            if(i==s||i==t){
                addedge(i+n,i,inf);
                addedge(i,i+n,0);
            }else{
                addedge(i+n,i,1);
                addedge(i,i+n,0);
            }
        }
        for(int i=1; i<=m; i++) {
            u=Read();
            v=Read();
            addedge(u,v+n,inf);
            addedge(v+n,u,0);
            addedge(v,u+n,inf);
            addedge(u+n,v,0);
        }
        n<<=1;
        printf("%lld\n",ISAP());
    }
```

From:
https://wiki.cvbbacm.com/ - **CVBB ACM Team**

Permanent link:
https://wiki.cvbbacm.com/doku.php?id=2020-2021:teams:legal_string:%E7%8E%8B%E6%99%BA%E5%BD%AA:%E7%BD%91%E7%BB%9C%E6%B5%81%E5%85%A5%E9%97%A8&rev=1626082682

Last update: **2021/07/12 17:38**