14.网络流入门

最大流

1.基本概念

容量网络

如果有向图 G = (V, E)中，满足下列性质：

（1）有唯一的一个源点 S（入度为 0）；

（2）有唯一的一个汇点 T（出度为 0）；

（3）图中每条弧 <u, v> 都有一非负容量 c (u, v)>=0。

满足上述条件的图 G 称为，也称容量网络（也称流网络），记为： G = (V,E,C)。

如图容量网络图 a，边上的权值是相应弧的容量。

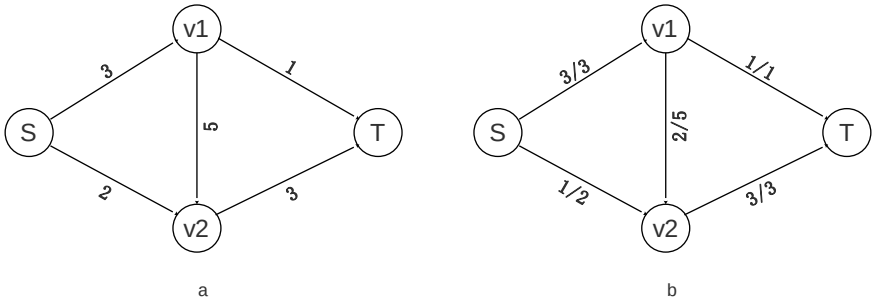

图 1

弧的流量

容量网络 G 中，每条弧< u, v >上 给定一个实数 f(u,v),满足：有 0 <= f (u, v) <= c(u, v),则 f(u,v)称为弧< u, v>上的实际流量，简称流量。如 f(s,v1)=3。

如图 b 中弧上注明的是 f(u,v)/c(u,v)。

网络流

所有弧上的流量的集合称为该容量网络的一个网络流，简称流。

可行流

在容量网络中满足以下条件的流称 f 为可行流：

（1）容量限制：弧<u,v>的流量 f(u,v)满足 0<=f(u,v)<=c(u,v；

（2）反对称性：f(u,v)=- f(v,u)；

（2）流量守恒：除了源点 s 和汇点 t 的任意结点，满足流出流量等于流入流量，|f|是可行流 f 的流量。

$$\sum_{v\in V} f(u,v)-\sum_{v\in V} f(v,u)=\begin{cases}|f| & 当u=s\\ 0 & \\ -|f| & 当u=t\end{cases}$$

容量网络至少有一个|f|=0 的可行流，成为零流。

流量|f|的定义：

$$|f|=\sum_{v\in V} f(s,v)-\sum_{v\in V} f(v,s)$$

可行流 f 的流量|f|就是从源点 s 流出的总流量-流入源点 s 的总流量。图 b 中的|f|=4。

最大流

在所有的可行流中， 流量|f|最大的一个可行流称为网络最大流，简称最大流。最大流不是唯一的。

求最大流的两类算法：增广路算法和预流推进算法

2.增广路算法（Augmenting Path Algorithm）

残余容量

给定容量网络 G 和可行流 f，弧<u,v>上的残余容量记为 r(u,v)=c(u,v)-f(u,v)，说明弧<u,v>还可以再有 r(u,v)的流量经过。

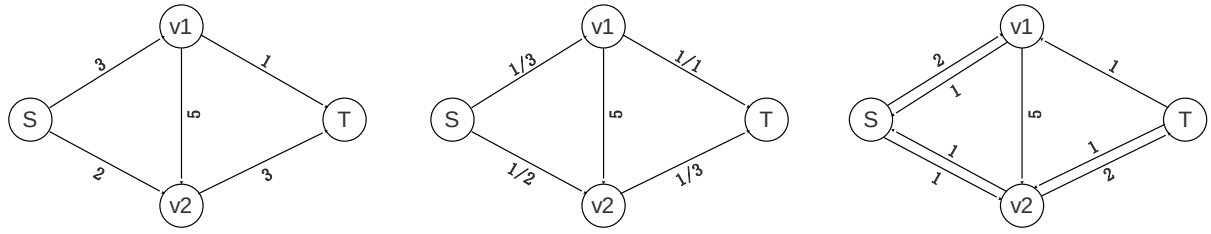

图 2

如上图 2，r(s,v1)=c(s,v1)-f(s,v1)=3-1=2，r(v1,v2)=c(v1,v2)-f(v1,v2)=5-0=5。

残余网络

有一个容量网络 G(V,E)及其上的网络流 f，G 关于 f 的残留网络记为 G'=(V',E')，其中 V'=V，E'有两类：

（1）对 G 中的每一条弧 <u,v>，如果 f(u,v)<c(u,v)，则 G'中存在一条弧 <u,v>∈E'，其容量为 r<u,v>=c(u,v)-f(u,v)。如图 2-c 中，r(s,v1)=2，r(s,v2)=1，r(v2,t)=2。

（2）对 G 中的每一条弧 <u,v>，而且如果 f(u,v)>0，则在 G'中再添加一条弧 <v,u>，其容量为 r(v,u)=c(v,u)-f(v,u)=0-(-f(u,v)=f(u,v)，称为反向弧。如图 2-c 中，r(v1,s)=1，r(v2,s)=1，r(t,v2)=1。

增广路

设 f 是容量网络 G 中的一个可行流，p 是**残留网络 G'**中从 Vs 到 Vt 的一条简单路径，若 P 满足条件：

（1）所有正向弧（方向与路径 p 方向一致）<u,v>，0≤f(u,v)＜c(u,c)，即这些弧都有余量 r(u,v)>0。

（2）所有反向弧（方向与路径 p 方向相反）<u,v>，f(v.u)>0，即这些弧上都有非 0 的流量。

如图 2-c 中，S-v1-v2-T 是一条增广路，沿着这条路还能增加流量 2。增广路 S-v2-T 能增加流量 1。

上述两条增广路上的弧都是正向弧。

下面看一具有反向弧的例子：

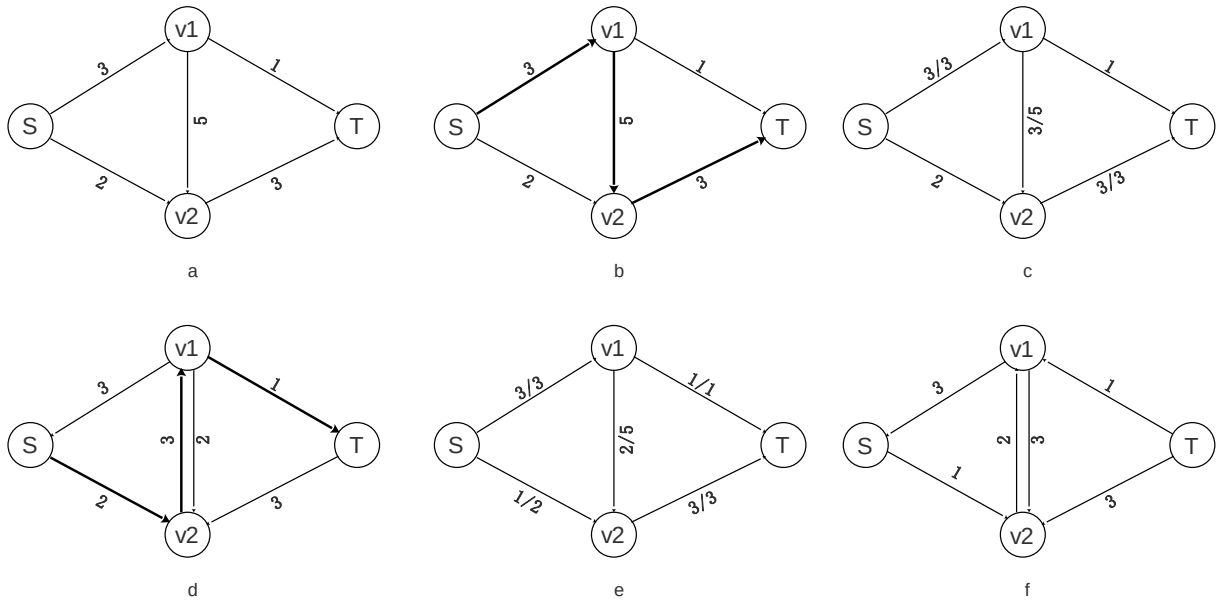

图 3

图 3-a 中，按 b 中黑色加粗增广路 s-v1-v2-t，增加流量 w1=min{r(sv1),r(v1,v2),r(v2,t)}=3，流量/容量图变为 c，残余网络为 d，有增广路 s-v2-v1-t，其中弧<v2,v1>是反向弧，r(v2,v1)=f(v1,v2)=3，增加流量 w2=min{r(s,v2),r(v2,v1),r(v1,t)}=1，流量/容量图变为 e，残留网络图变为 f，无增广路，|f|=4 是最大流。

如果没有反向弧<v2,v1>，就无法再继续增加流量。

所以说反向边是网络流的精髓所在。

如果不建立反向弧<v2,v1>，就发现不了那条增广路径 s-v2-v1-t。本质上看，建立反向弧就是为算法修改先前犯错误的可能，等价来拟补所犯的错误。有时叫悔流或退流，送回去改道。这条增广路的作用就是，把原来弧<v1,v2>的流量 3，推送回去 1，改道走弧<v1,t>。原来弧<v2,t>少的 1 由弧<s,v2>的流量 1 来替换。

所以反向弧的作用，就是将原来流量推送回去一部分，也称为抵消操作，是求最大流的最关键的操作。

残余网络的容量就是弧的余量，后面的网络流算法都是在残余网络上进行的。

每找到一条从源点到汇点的一条增广路径（f(u,v)>0）后，都要修改残余网络：

r(u,v)-=f(u,v)；r(v,u)+=f(u,v)。

增广路算法求最大流，从人一个可行流开始（第一次可行流为零流），每次在残余网络上找增广路对网络流进行增广，找出这条路上能增加的最大流量 minf=min{所有增广路上弧的剩余容量}，然后修改增广路上弧的剩余容量，正向弧-minf，反向弧+minf；然后继续在残余网络上找增广路，直到没有增广路为止。

2.1 一般增广路算法（Generic Augmenting Path Algorithm）-FF（Ford-Fulkerson）

这里指的是求最大流的 Ford-Fulkerson 方法：不断的在找增广路增加最大流的值。方法的实现有多种不同是实现算法。

我们从 0 流量出发（此时残存网络就是原图），找到增广路径（注意增广路径一定是在残存网络里找），接着把流更新，修改残余网络，直到残存网络中没有增广路径（就是没有路径从 s 到 t）为止。

上图 3-a 找增广方法有多种，图 3 所示的一种方法，下列图 4 描述了另外一种增广方法。

下图 4 的是一种增广方法：

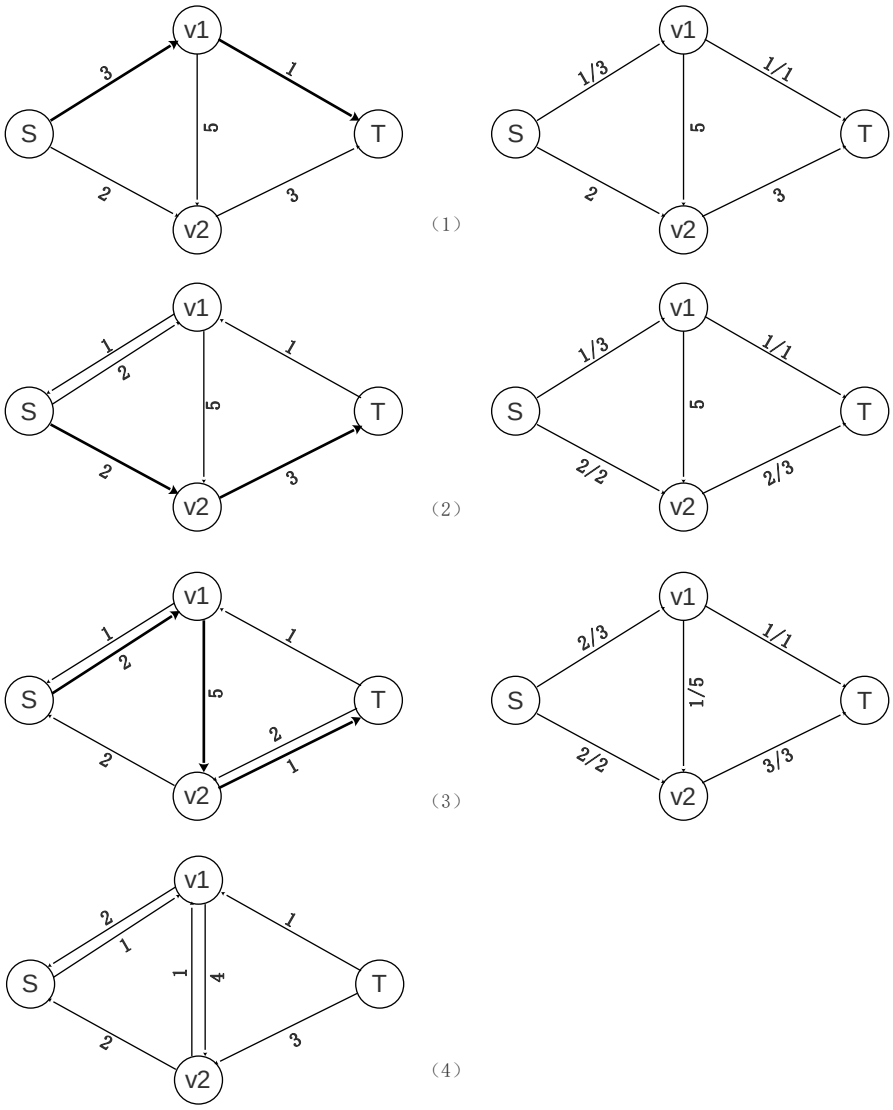

图 4

主要思路和操作是，找到一条增广路，计算可以增加的最大流量 f，关键一步是对应增广路上的每一天弧 <u,v>，要修改剩余流量 r(u,c)-=f；r(v,u)+=f。

这里讲一种用递归找增广路的算法:

```
//p3376
const int INF=0x7fffffff;
const int N=10000+10;
const int E=100000+100;
struct Edge{
    int u,v,w,next;//w 是弧<u,v>的剩余容量
};
Edge e[2*E];
int Flow[N];
int h[N];
int vis[N];
int n,m,s,t,tot=-1;
```

```
int Maxflow=0;
inline void Add(int u,int v,int w){
    e[++tot].u=u;
    e[tot].v=v;
    e[tot].w=w;
    e[tot].next=h[u];
    h[u]=tot;
}
int dfs(int u,int minf){
    if(u==t||minf==0)return minf;
    vis[u]=1;
    for(int i=h[u];i!=-1;i=e[i].next){
        int v=e[i].v;
        int w=e[i].w;
        if(w&&!vis[v]){
            int f=dfs(v,min(minf,w));
            if(f>0){
                e[i].w-=f;
                e[i^1].w+=f;
                return f;
            }
        }
    }
    return 0;
}
void FF(){
    Maxflow=0;
    while(1){
        memset(vis,0,sizeof(vis));
        int f=dfs(s,INF);
        if(f==0)return;
        Maxflow+=f;
    }
}
int main(){
    scanf("%d%d%d%d",&n,&m,&s,&t);
    memset(h,-1,sizeof(h));
    for(int i=0;i<m;i++){
        int u,v,w;
        scanf("%d%d%d",&u,&v,&w);
        Add(u,v,w);
        Add(v,u,0);
    }
    FF();
    printf("%d\n",Maxflow);
```

```
    return 0;
}
```

算法的时间复杂度，每次可能增加流量 1，与最大流有关，O(m*Maxflow)。

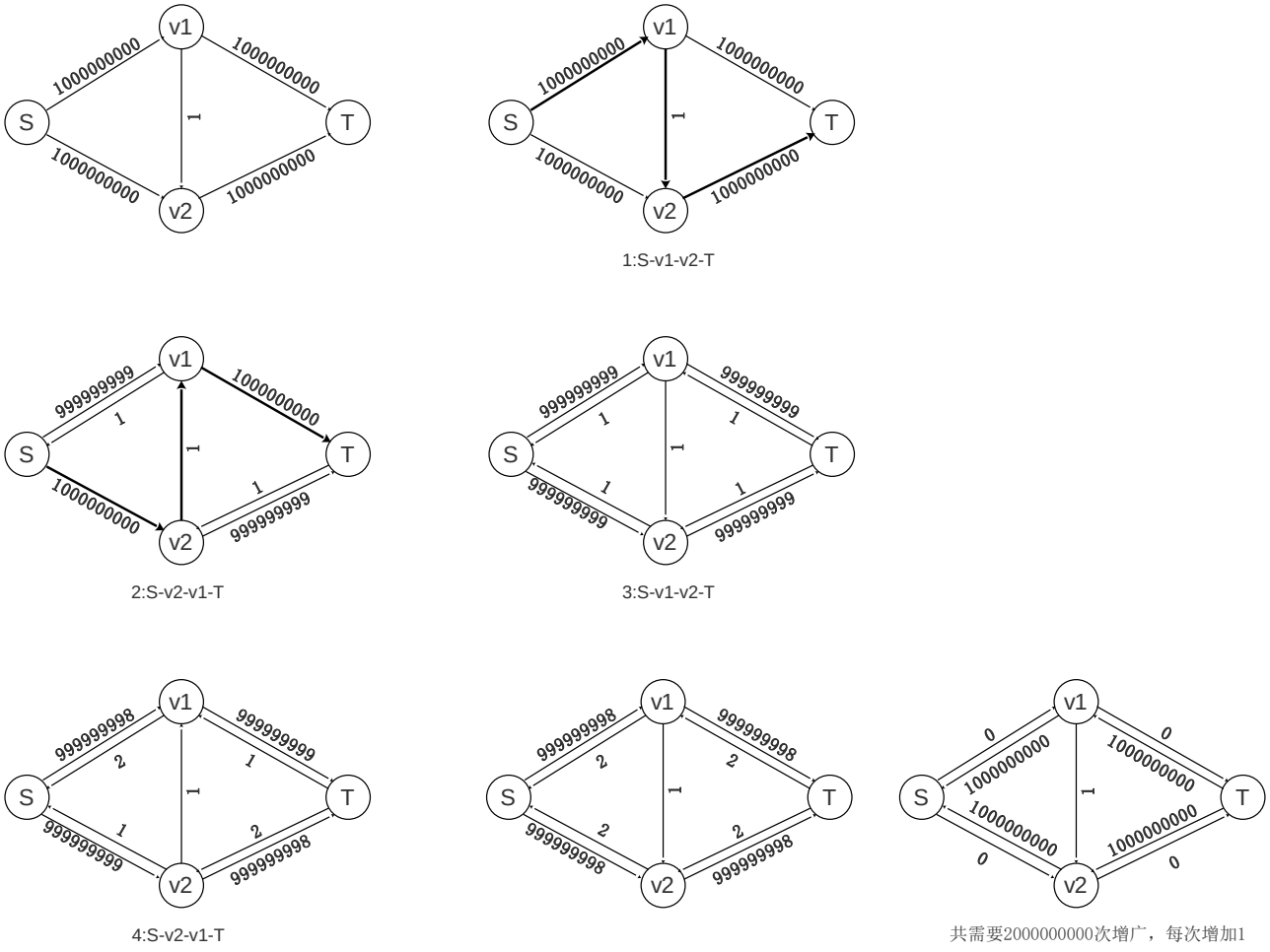

图 5

2.2 最短增广路算法（Shortest Augmenting Path）-EK（Edmonds-Karp）

FF（Ford-Fulkerson）方法中，用 dfs 找增广路效率比较低，我们用 bfs 找增广路来改进 FF 效率，就是每次在残余网络中找一条含弧数量最少的增广路径进行增广，也就是从源点 s 到汇点 t 一条最短路（边权为 1），这种算法就是 EK（Edmonds-Karp）算法。

bfs 找到的增广路其实就是边权长度看做长度为 1 的最短路。

bfs 先找到一条增广路，计算出可以增加的最大流量 f，然后倒序修改增广路的剩余容量，需要 bfs 时记录增广路径的父亲结点即可。类似 bfs 记录最短路径的方法。

bfs 的过程，等于在残余网络上从源点开始建立了一个网络层次图，d[s]=0 开始，增广路上的弧<u,v>要满足 d[v]=d[u]+1。这里没有真正的建立层次图，就是 bfs 时顺序而已。

下图 6 中的（1）到（6）是 bfs 找增广路的增广过程。

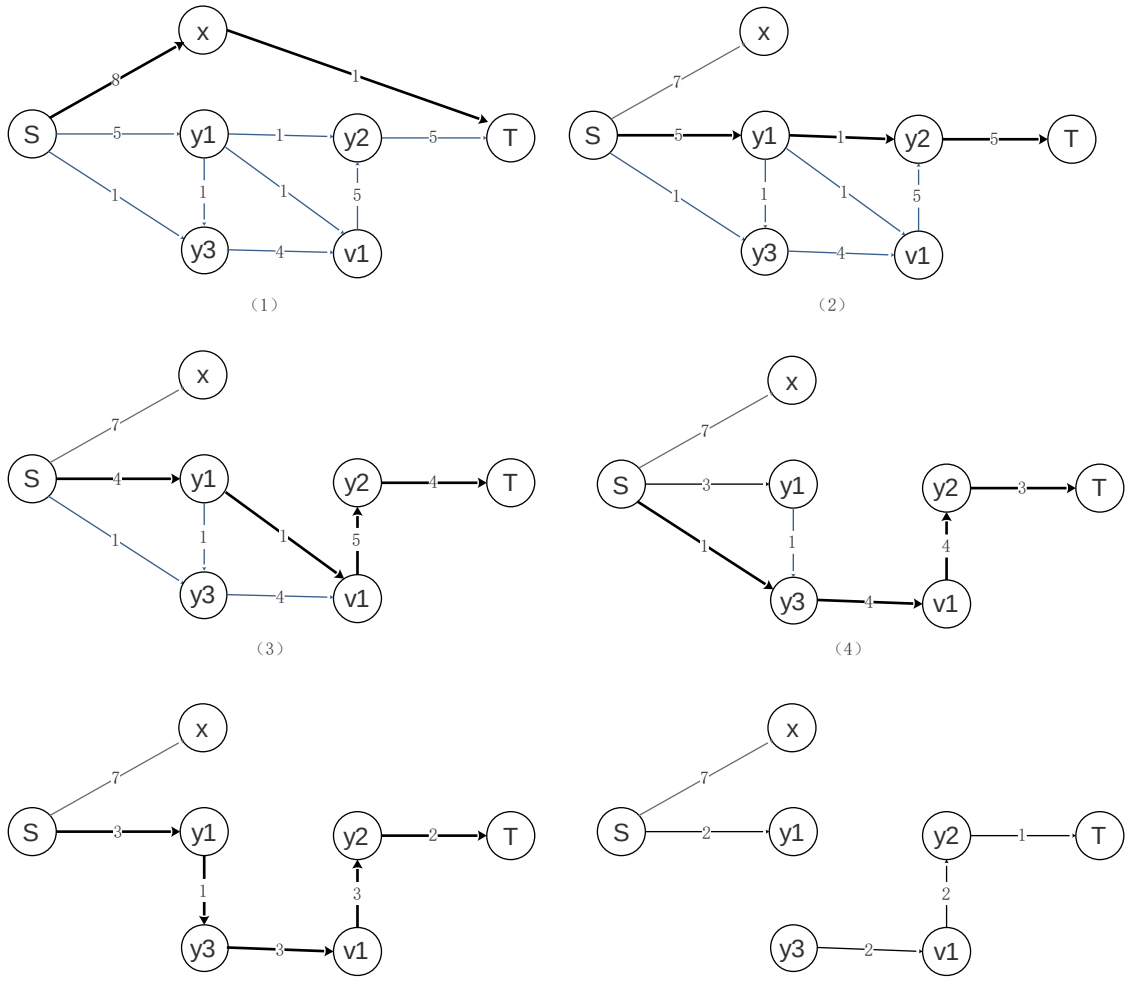

图 6

```
const int INF=0x7fffffff;
const int N=10000+10;
const int E=100000+100;
struct Edge{
    int u,v,w,next;
};
Edge e[2*E];
int Flow[N];
int pre[N];
int h[N];
int vis[N];
int n,m,s,t,tot=-1;
int Maxflow=0;
inline void Add(int u,int v,int w){
    e[++tot].u=u;
    e[tot].v=v;
    e[tot].w=w;
    e[tot].next=h[u];
    h[u]=tot;
}
```

```
int bfs(){
    memset(vis,0,sizeof(vis));
    memset(pre,0,sizeof(pre));
    queue<int>q;
    q.push(s);
    vis[s]=1;
    Flow[s]=INF;
    while(!q.empty()){
        int u=q.front();q.pop();
        for(int i=h[u];i!=-1;i=e[i].next){
            int v=e[i].v;
            int w=e[i].w;
            if(w&&!vis[v]){
                Flow[v]=min(Flow[u],w);
                pre[v]=i;
                q.push(v);
                vis[v]=1;
                if(v==t)return Flow[v];
            }
        }
    }
    return 0;
}
void EK(){
    int f=0;
    while(f=bfs()){
        int u=t;
        while(u!=s){
            int i=pre[u];
            e[i].w-=f;
            e[i^1].w+=f;
            u=e[i].u;//u=e[i^1].v;//也可以
        }
        Maxflow+=f;
    }
}
int main(){
    scanf("%d%d%d%d",&n,&m,&s,&t);
    memset(h,-1,sizeof(h));
    for(int i=0;i<m;i++){
        int u,v,w;
        scanf("%d%d%d",&u,&v,&w);
        Add(u,v,w);
        Add(v,u,0);
    }
```

```
    EK();
    printf("%d\n",Maxflow);
    return 0;
}
```

复杂度：

建立层次网络和找增广路径。

每次 bfs，就是构造层次网络的过程，时间 O(m)，最多 n 个层次网络，建立层次网络的时间是 O(n*m)。

我们最坏情况下每次只少增广一条边（饱和弧），最多需要增广 m-1 次。

EK 算法的时间复杂度是 $O(n*m^2)$。

最多 O（n*m）次增广，每次 bfs 时间是 O(m)，所以时间是 O(n*m2)。

实际远远达不到上限，能过 1000 到 10000 的网络。

2.3 连续最短增广路算法（Successive Shortest Augmenting Path）-Dinic

Dinic 算法可以认为是 EK 算法的优化，又和 FF 算法类似（dfs 找增广路），EK 算法是执行完一次 BFS 后增广后一次，重新再 BFS 从源点到汇点找增广路。 Dinic 每次通过 BFS 构造分层图后，只需要一次 dfs 达到多路增广的效果。Dinic 算法是比较优秀的求最大流算法，一般的网络流题目会卡 FF 和 EK，但很少卡 Dinic。

Dinic 算法流程：

（1）根据残留网络，利用 BFS 构造层次网络。

（2）在层次网络中利用一次 dfs 进行多路增广。

（3）重复（1）和（2），直到层次网络中没有汇点，增广结束。

层次网络，就是把残余网络中的点按照到源点的距离分“层”，只保留不同层之间的边的图。分层过程也叫做标号过程，就是给每个点标一个标号，就是这个点的层次（深度）。源点深度为 0，根据 bfs 过程往后依次给后面的点标号。

增广时，当前点 u 能否沿着弧<u,v>往后走要满足两个条件：一个是弧<u,v>的残余容量>0，二是按层次网络的编号递增，即要满足 d[v]=d[u]+1。这条弧<u,v>也称允许弧。

参考上面图 6 的增广过程，其中（3）和（4）是一次 bfs 后就能完成的增广。d(T)=4。

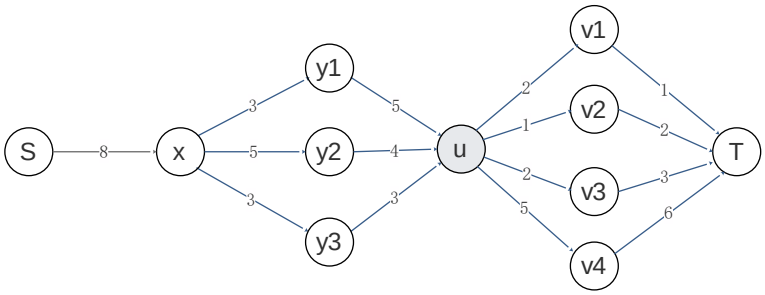

图 7

为了实现多路增广，这里用到一个优化：

每次增广一条路后可以看做“榨干”了这条路，既然榨干了就没有再增广的可能了。但如果后面每次都再扫描这些“枯萎的”边是很浪费时间的。于是我们就记录一下“榨取”到哪条边了，然后下一次直接从这条边开始增广，就可以节省大量的时间，这个非常有效的优化称为当前弧优化。

具体的实现方法，就是每次 dfs 增广前，邻接表的 h 数组复制一份，存进 cur 数组，然后在 cur 数组中每次记录“榨取”到哪条边了。

图 7 中，如果 dfs 增广止 s-x-y1-u，当前到达 u 位置，当前的残余 minf=3，u 沿着弧<u,v1>往后能增加流量 1，然后 u 再沿着弧<u,v2>增广增加流量 1，这是 minf 省选 1 了，继续沿着弧 <u,v3>只能增加流量 1 了，尽管沿着弧<u,v3>往后还能增加 1。这时候 u 回溯，下一次沿着 s-x-y2-u,第二次到达 u，由于第一次访问 u 时，弧 <u,v1)和弧<u,v2>往后已经饱和了，所以没有必要再尝试了，直接从上次的弧 <u,v3>开始尝试就行了。

参考代码：

```
const int INF=0x7fffffff;
const int N=10000+10;
const int E=100000+100;
struct Edge{
    int u,v,w,next;
};
Edge e[2*E];
int d[N];
int h[N];
int cur[N];
int vis[N];
int n,m,s,t,tot=-1;
int Maxflow=0;
inline void Add(int u,int v,int w){
    e[++tot].u=u;
    e[tot].v=v;
    e[tot].w=w;
    e[tot].next=h[u];
    h[u]=tot;
}
int bfs(int s,int t){
    memset(d,-1,sizeof(d));
    queue<int>q;
    q.push(s);
    d[s]=0;
    while(!q.empty()){
        int u=q.front();q.pop();
        for(int i=h[u];i!=-1;i=e[i].next){
            int v=e[i].v;
            int w=e[i].w;
            if(d[v]==-1&&w){
                d[v]=d[u]+1;
                q.push(v);
            }
        }
    }
    return d[t]!=-1;
}
int dfs(int u,int t,int minf){//minf 是到目前为止所有弧上最小的残余容量
```

```
    if(u==t||minf==0)return minf;
    int f,flow=0;
    for(int i=cur[u];i!=-1;i=e[i].next){
        cur[u]=i;//记录当前弧，下次从这个开始
        int v=e[i].v;
        int w=e[i].w;
        if(d[v]==d[u]+1&&(f=dfs(v,t,min(minf,w)))){
            minf-=f;
            flow+=f;
            e[i].w-=f;
            e[i^1].w+=f;
            if(minf==0)return flow;//u 出发的后面的弧没有残余容量可供分配了
        }
    }
    return flow;
}
void Dinic(int s,int t){
    Maxflow=0;
    while(bfs(s,t)){
        for(int i=1;i<=n;i++)cur[i]=h[i];
        Maxflow+=dfs(s,t,INF);
    }
}
int main(){
    scanf("%d%d%d%d",&n,&m,&s,&t);
    memset(h,-1,sizeof(h));
    for(int i=0;i<m;i++){
        int u,v,w;
        scanf("%d%d%d",&u,&v,&w);
        Add(u,v,w);
        Add(v,u,0);
    }
    Dinic(s,t);
    printf("%d\n",Maxflow);
    return 0;
}
```

弧优化是可以采取地址引用方法，到达更新 cur 的作用。

```
for(int &i=cur[u];i!=-1;i=e[i].next)
```

Dinic 的时间复杂度分析：（参考百度）

因为在 Dinic 的执行过程中，每次重新分层，汇点所在的层次是严格递增的，而 n 个点的层次图最多有 n 层，所以最多重新分层 n 次。在同一个层次图中，因为每条增广路都有一个瓶颈，而两次增广的瓶颈不可能相同，所以增广路最多 m 条。搜索每一条增广路时，前进和回溯都最多 n 次，所以这两者造成的时间复杂度是 O(nm)；而沿着同一条边(i,j)不可能枚举两次，因为第一次枚举时要么这条边的容量已经用

尽，要么点 j 到汇不存在通路从而可将其从这一层次图中删除。综上所述，Dinic 算法时间复杂度的理论上界是 O(n^2*m)。

Dinic 已经足够高效，一般能解决 10000 的网络流。

2.4 改进的最短增广路算法(Improved Shortest Augumenting Path）-ISAP

在 Dinic 中我们每次增广前都进行了一次 bfs 建立分层图对结点进行标号，虽然一次标号能进行多路增广，但是可能要进行多次 bfs。那有没有什么办法只跑一次 bfs 呢?那就是 ISAP 算法了!

ISAP 运行过程：

（1）从 t 到 s 跑一遍 bfs（这里从汇点反向跑），标记深度（d[t]=0）;

（2）从 s 到 t 跑 dfs，和 Dinic 类似，只是当一个点 u 跑完后（u 出发的所有允许弧递归结束），如果从上一个点传过来的最小残余 minf 还有剩余（对于该点 u 当前的深度来说，该点在该点前面走过的路上以后就无意义了），则把它的深度加 1（这样和刚才已经走过的下一层点就断开了），如果整个网络出现断层（某个深度没有点了，则 s 无法到达 t，于是将 d[s]=n+1,保证 s 无法到达 t），结束算法；

（3）如果（2）没有结束算法，重复操作（2）。

ISAP 其实与 Dinic 差不多，但是它只跑一遍 bfs，但是每个点的层数随着 dfs 的进行而不断提高（这样就不用反复跑 bfs 重新分层了），当 s 的深度大于 n 时（这就是为什么 bfs 要从 t 到 s 反向分层标号），结束算法。

下图 8 的增广过程：

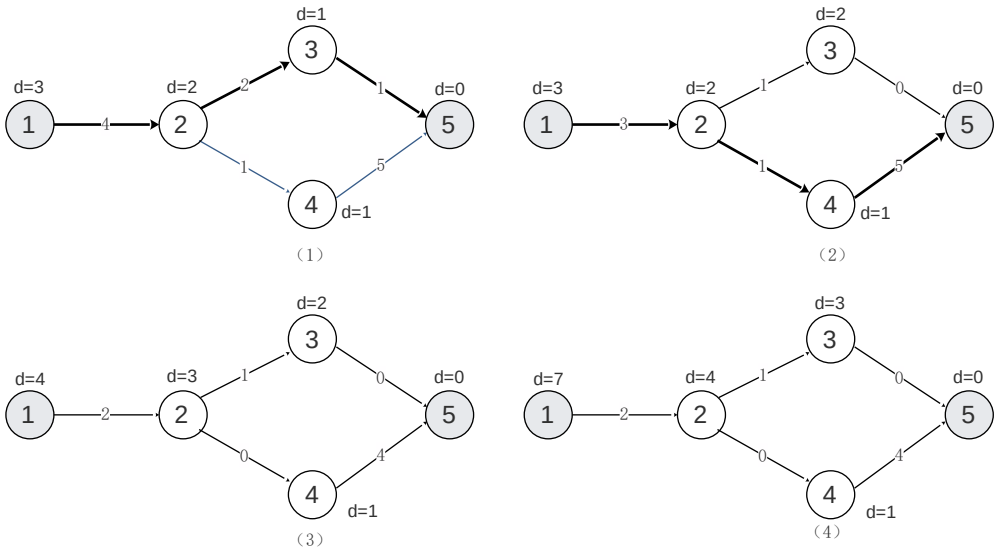

图 8

下图 9 的增广过程：

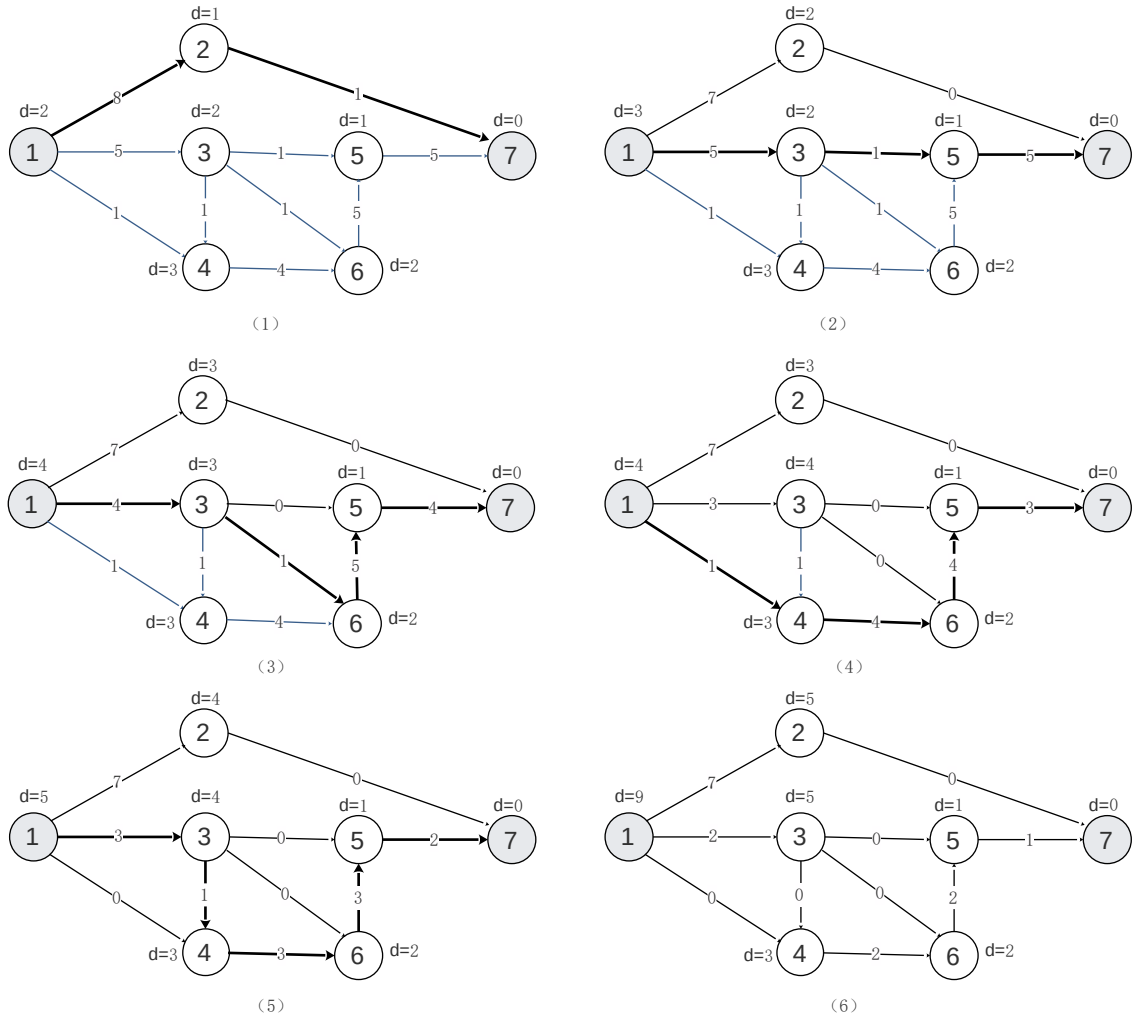

图 9

参考代码：

```
const int INF=2e9;
const int N=10000+10;
const int E=100000+100;
struct Edge{
    int u,v,w,next;
};
Edge e[2*E];
int h[N];
int cur[N];
int vis[N];
int d[N];
int gap[N];//gap[i]:记录深度为 i 的结点的数量
int n,m,s,t,tot=-1;
inline void Add(int u,int v,int w){
    e[++tot].u=u;
    e[tot].v=v;
    e[tot].w=w;
    e[tot].next=h[u];
```

```
    h[u]=tot;
}
void bfs(int s,int t){
    memset(d,-1,sizeof(d));
    memset(gap,0,sizeof(gap));
    queue<int>Q;
    Q.push(t);
    d[t]=0;
    gap[0]=1;
    while(!Q.empty()){
        int u=Q.front();Q.pop();
        for(int i=h[u];i!=-1;i=e[i].next){
            int v=e[i].v;
            int w=e[i].w;
            if(d[v]==-1&&e[i^1].w){
                d[v]=d[u]+1;
                gap[d[v]]++;
                Q.push(v);
            }
        }
    }
}
int dfs(int u,int t,int minf){
    if(u==t||minf==0)return minf;
    int f,flow=0;
    for(int &i=cur[u];i!=-1;i=e[i].next){
        int v=e[i].v;
        int w=e[i].w;
       if(d[v]==d[u]-1&&(f=dfs(v,t,min(minf,w)))){//注意这里深度要求和 dinic 不一样，这里是反向标号的
            minf-=f;//分掉 f，留个后面 v 的兄弟可分的减少了
            flow+=f;
            e[i].w-=f;
            e[i^1].w+=f;
            if(minf==0)return flow;//u 上边没有可向下分的流量了。
        }
    }
    //前面和 Dinci 基本是一样的
    //到这里了，说明 u 之前的 minf 有剩余，u 后面的都已经饱和了
    gap[d[u]]--;
    if(gap[d[u]]==0)d[s]=n+1;//断层了，s 无法到达 t 了，结束即可
    d[u]++;//层数+1
    gap[d[u]]++;//层数对应点的个数+1
    return flow;
}
```

```
int ISAP(int s,int t){
    int ans=0;
    bfs(s,t);//一次分层标号即可，不管剩余容量
    while(d[s]<n){
        for(int i=1;i<=n;i++)cur[i]=h[i];
        ans+=dfs(s,t,INF);
    }
    return ans;
}
int main(){
    scanf("%d%d%d%d",&n,&m,&s,&t);
    memset(h,-1,sizeof(h));//初值为-1 不要忘记
    for(int i=0;i<m;i++){
        int u,v,w;
        scanf("%d%d%d",&u,&v,&w);
        Add(u,v,w);
        Add(v,u,0);
    }
    int Maxflow=ISAP(s,t);
    printf("%d\n",Maxflow);
    return 0;
}
```

这里用到了一个叫 gap 优化：

统计每个深度对应点数只为了这句话：

`if(gap[d[u]]==0)d[s]=n+1;//`出现断层了算法结束，让 `d[s]=n+1`,这样保证 s 一定无法达到 t 了。

因为我们是按照深度来往前走的，路径上的点的深度一定是连续的，而 t 的深度为 0，如果某个深度的点不存在，那么我们就无法到达 t 了。因为增广路径的深度一定是连续的。

最后结束后 `d[s]=n+2`,思考为什么？最后回溯的 s 时，前面出现了 `n+1`,最后 s 结束时又加了个 1。

3. 最高标号预流推进 HLPP（Highest-Label Preflow-Push Algorithm）

增广路算法是对整个残余网络进行检查，每次找出一条增广路，然后沿着增广路对弧进行操作，最多对 n-1 条弧（n 个点的最短路最多 n-1 条弧），每次增广的时间是 O(n)。

预流推进算法考虑的是对每条弧的操作和处理，而不必每次要处理一条完整的增广路径。

图 10：采取增广路算法：有 3 条增广路，依次流量是 3,4,5。

采用预流推进算法：通俗的描述，把每个点看做一个水库，可以储存水。从源点开始，猛灌水，能往前流就流，不能流的想法退回到源点。保证最后，除了源点和起点，中间结点不能后库存，满足流量守恒。最后汇点水库里的水就是最大流。

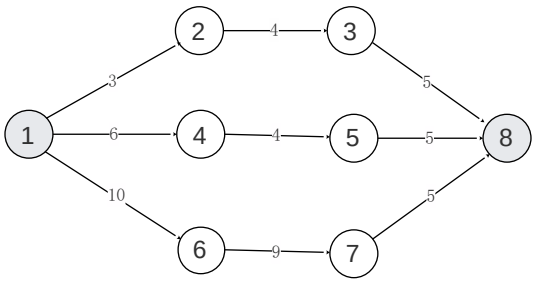

图 10

几个概念：

超额流（盈余）：对结点 u 而已，流入的流量减去流出的流量称为 u 的超额流 e(u)。

$$e(u)=\sum_{v\in V} f(v,u)-\sum_{v\in V} f(u,v)$$

活跃点：如果 u∈V-{s,t}，e(u)>0，u 称为活跃点。

预流：对于容量网络 G 的一个流 f，每天弧都满足：

0<=f(u,v)<=c(u,v)，<u,v>∈E

e(u)>0，u∈V-{s,t}

则该流 f 称为 G 的一个预流。

对于容量网络 G 的一个预流 f，如果存在活跃结点，则该预流 f 不是可行流。预流推进算法就是选择一个活跃点，把他的盈余推送到他的邻接点，争取使 e(u)=0。如果多个邻接点，尽量选择距离 t 最近的点，因为我们的目的是将流水推送到汇点 t。

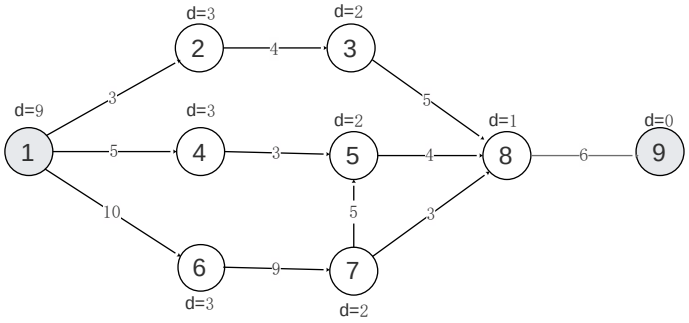

图 11

步骤：

（1）从汇点倒着贴标签编号为结点的高度，d[t]=0,bfs 顺序往前依次编号。我们规定水只能从高处往下一层流，d[u]=d[v]+1; 开始时 d[s]=n，保证能流出水。

（2）开始时，从源点根据发出弧的容量，送出尽量多的水，发出弧的容量和。

（3）推送时，选高度最高的点 u，推送时，规定水往下一层流:d[u]=d[v]+1，弧(u,v)残余量 w>0,推送流量为 f=min(w,e[u])。

e[i].w-=f;e[i^1].w+=f;

flow[u]-=f;flow[v]+=f; //库存

如果所有的可行弧推送完了，u 还有库存,flow[u]>0，我们抬高 u 的高度，给 u 重新贴标签，让 u 的高度为 u 发出可行弧的最低点的高度+1，保证能流，并且尽量流向低的结点。

（4）一个优化：如果一个点 u 再被重贴标签以后，如果它原来的高度已经没有其它点（出现断层），那么高于它的点一定不能将流量推送到 t 了。所以我们可以将那些中间点中高度大于 d[u]且小于 n+1 的点高度设置为 n+1，如果有流量以便尽快将流量推送回 s（d[s]=n）。和 Dinic 一样，采用 gap 优化。

参考：

预流推进算法的思想就：

把每个点看作一个临时蓄水的水库（正常的网络流不允许这样），允许水在（源点 s 和汇点 t 以外）的结点中暂时存储（称作这个点的超额流或盈余 Excess Flow），同时找机会把结点的超额流通过可行弧推送到邻接点。如果能保证算法结束后所有（源点 s 和汇点 t 除外）结点的超额流都为 0（反对称性），那么这种算法就是正确的。

为了避免反复推送而出现死循环的问题，给每个节点定义一个高度，结点的高度决定推荐方向：只允许从高度较高的结点向高度较低的结点进行推送。d[t]=0, d[s]=n, 开始其他各点的高度都为 0，开始时从源点发出尽量多的流（源点发出弧的容量和），尽量往汇点推送。当把流推送到中间结点时，先暂时收集在该结点的水库中，然后再沿着弧往后面推送。如果一个节点因为受到高度的限制而不能推送自身的超额流，那么我们就抬高这个节点的高度，我们把这个操作叫做重贴标签。

算法的两个操作就是：推送预流；结点重贴标签。

一般预流推进算法的时间是 $O(n^2*m)$。

下面的改进能把时间优化到 $O(n^2*m^{1/2})$。

最高预流推进算法：

（1）从具有最大距离标号的活跃点开始预流推进（用优先队列实现），这样标号小的尽量接收更多的标号大的结点的推进流量，可能减少推进次数。

（2）通过一遍 bfs 将每个点的初始高度设置为它到汇点 t 的最短距离，这样就节省了大量重贴标签操作。源点 s 的高度还是应该设置为 n。

（3）u 推送时，规定水往下一层流：d[u]=d[v]+1，弧(u, v)残余量 w>0, 推送流量为 min(w, e[u])。

u 推送过程执行完毕后，如果我们发现 e(u)>0，说明当前的高度 d[u]不够，因此我们对 u 重贴标签。我们找到有流量而且边的另一端 v 高度最小的边，将 d[u]=d[v]+1，这样就保证了下次 u 一定可以推送 v。将重贴标签后的 u 加入优先队列中。

（4）如果一个点 u 再被重贴标签以后，如果它原来的高度已经没有其它点，那么高于它的点一定不能将流量推送到 t 了。所以我们可以将高度大于 d[u]且小于 n+1 的点高度设置为 n+1，以便尽快将流量推送回 s。和 Dinic 一样，采用 gap 优化。

如图所示：

参考代码：

```
const int INF=0x7fffffff;
const int N=1200+10;
const int E=120000+100;
struct Edge{
    int u,v,w,next;
};
struct node{
    int p;
    int h;
    bool operator < (const node &x)const{return h<x.h;};
};
Edge e[2*E];
priority_queue<node>Q;
int h[N];
int inq[N];
int d[N];
```

```
int flow[N];//结点的盈余量
int gap[2*N];//gap[i]:记录深度为 i 的结点的数量 ;可能达到 2*n-1,后面的点需要依次送回 s
int n,m,s,t,tot=-1;
inline void Add(int u,int v,int w){
    e[++tot].u=u;e[tot].v=v;e[tot].w=w;e[tot].next=h[u];h[u]=tot;
}
inline int bfs(int s,int t){//从汇点 t 反向贴标签编号,高度 d[i]
    queue<int>q;
    memset(d,-1,sizeof(d));
    d[t]=0;
    q.push(t);
    while(!q.empty()){
        int u=q.front();q.pop();
        for(int i=h[u];i!=-1;i=e[i].next){
            int v=e[i].v;
            int w=e[i^1].w;
            if(w&&d[v]==-1){
                d[v]=d[u]+1;
                q.push(v);
            }
        }
    }
    return d[s]!=-1;
}
inline void PUSH(int u){//推进有容量且高度少 1 的邻接点
    for(int i=h[u];i!=-1;i=e[i].next){
        int v=e[i].v;
        int w=e[i].w;
        if(w&&d[v]+1==d[u]){//往下一层，容量大于 0 的结点推进
            int f=min(flow[u],w);//推送流量
            e[i].w-=f;e[i^1].w+=f;
            flow[u]-=f;flow[v]+=f;
            if(v!=s&&v!=t&&inq[v]==0){
                Q.push((node){v,d[v]});
                inq[v]=1;
            }
            if(flow[u]==0)break;
        }
    }
}
inline void Gap(int u){//把比 u 高的那些点高度变为 n+1,以便早流回源点 s
    for(int i=1;i<=n;i++)
        if(i!=s&&i!=t&&d[i]>d[u]&&d[i]<=n)d[i]=n+1;
}
inline void Relabel(int u){//重新贴标签，增加高度 =比周围最低的高度+1
```

```
    d[u]=INF;
    for(int i=h[u];i!=-1;i=e[i].next){
        int v=e[i].v;
        int w=e[i].w;
        if(w>0&&d[v]<d[u])d[u]=d[v];//别用 min 函数
    }
    d[u]++;
}
inline void Init(){//把源点 s 出发的点先推满再说,进入初始队列
    for(int i=h[s];i!=-1;i=e[i].next){
        int v=e[i].v;
        int w=e[i].w;
        if(w){
            e[i].w-=w;/*=0*/e[i^1].w+=w;
            flow[s]-=w;flow[v]+=w;
            if(v!=s&&v!=t&&inq[v]==0){
                Q.push((node){v,d[v]});
                inq[v]=1;
            }
        }
    }
}
inline int HLPP(int s,int t){
    if(!bfs(s,t))return 0;//初次贴标签，初始化高度
    d[s]=n;//源点最高，保证能往后流
    memset(gap,0,sizeof(gap));//gap[i]:高度为 i 的结点数量
    for(int i=1;i<=n;i++)if(d[i]!=-1)gap[d[i]]++;
    memset(inq,0,sizeof(inq));
    Init();//源点 s 出发的点先推满再说,进入初始队列：s 和 t 是不进入队列的
    while(!Q.empty()){
        int u=Q.top().p;Q.pop();
        inq[u]=0;
        PUSH(u);//u 向后推进流
        if(flow[u]){//如果 u 还有库存，需要重新贴标签
            gap[d[u]]--;
            if(gap[d[u]]==0)Gap(u);//断层优化
            Relabel(u);//重新贴标签
            gap[d[u]]++;
            Q.push((node){u,d[u]});
            inq[u]=1;
        }
    }
    return flow[t];
}
int main(){
```

```
    scanf("%d%d%d%d",&n,&m,&s,&t);
    memset(h,-1,sizeof(h));//初值为-1 不要忘记
    for(int i=0;i<m;i++){
        int u,v,w;
        scanf("%d%d%d",&u,&v,&w);
        Add(u,v,w);Add(v,u,0);
    }
    int Maxflow=HLPP(s,t);
    printf("%d\n",Maxflow);
    return 0;
}
```

最大流算法：

两类算法	算法名称	复杂度	算法概述
增广路算法（ Augmenting Path Algorithm）	1.一般增广路算法（Generic Augmenting Path Algorithm）FF（Ford-Fulkerson）	O(m*Maxflow)	每次在残余网络中利用 dfs 任意找一条从源点 s 到汇点 t 的增广路，直至不存增广路为止
	2.最短增广路算法（Shortest Augmenting Path）EK（Edmonds-Karp）	$O(n*m^2)$	在层次网络中，每次利用 bfs 找最短增广路，指导汇点 t 不在层次网络中
	3.连续最短增广路算法-Dinic（Successive Shortest Augmenting Path）	$O(n^2*m)$	利用 dfs 实现多路增广，每次 bfs 分层标号
	4.改进的最短增广路算法 ISAP(Improved Shortest Augumenting Path）	$O(n^2*m)$	一次 bfs 标号。然后 dfs 多路增广，同时修改编号，利用 gap 优化
预流推进算法（ Preflow-Push Algorithm）	5.一般预流推进算法（Generic Preflow-Push Algorithm）	$O(n^2*m)$	维护一个预流，利用队列，不断对活跃点进行推进和重标号来调整预流。
	6.最高标号预流推进 HLPP（Highest-Label Preflow-Push Algorithm）	$O(n^2*m^{1/2})$	利用优先队列，每次检查最高标号的活跃点进行推进和重标号。gap 优化。

最大流最小割定理

割（CUT）是网络 G=（V，E）中顶点的一个划分，它把网络中的所有顶点划分成两个顶点集合 S 和 T=V-S，其中源点 s∈S，汇点 t∈T。记为 CUT（S,T）。

如果一条弧的两个顶点分别属于顶点集 S 和 T（一个顶点在 S，另一个在 T），那么这条弧称为割 CUT（S,T）的一条割边。

从 S 指向 T 的割边是正向割边；

从 T 指向 S 的割边是逆向割边。

割 CUT（S,T）中所有正向割边的容量和称为割 CUT（S,T）的容量。不同割的容量不同。

割的净流量=正向割边容量和-逆向割边的容量和。任何割的净流量等于流量|f|。

定理一：

如果 f 是网络中的一个流，CUT（S,T）是任意一个割，那么 f 的值等于正向割边的流量与负向割边的流量之差。

证明：

设 X 和 Y 是网络中的两个顶点集合，用 f（X,Y）表示从 X 中的一个顶点指向 Y 的一个顶点的所有弧（弧尾在 X 中，弧头在 Y 中：X Y）的流量和.

只需证明：f=f(S,T)-f(T,S) 即可。

下列结论成立：

如果 X∩Y= ，那么：

f(X,(Y1∪Y2))=f(X,Y1)+f(X,Y2)

f((X1∪X2),Y)=f(X1,Y)+f(X2,Y) 成立。

根据网络流的特点：

如果 V 既不是源点也不是汇点，那么：

f({V},S∪T)-f(S∪T,{V})=0;

任何一个点，流入的与流出的量相等。

如果 V 是源，那么：

f({V},S∪T)-f(S∪T,{V})=f

对于 S 中的所有点 V 都有上述关系式，相加得到：

f(S,S∪T)-f(S∪T,S)=f

f= f(S,T)- f(T,S)<=f(S,T)<=割 CUT（S,T）的容量

推论 1：

如果 f 是网络中的一个流，CUT（S,T）是一个割，那么 f 的值不超过割 CUT（S,T）的容量。

推论 2：

网络中的最大流不超过任何割的容量。

定量 2：

在任何网络中，如果 f 是一个流，CUT（S,T）是一个割，且 f 的值等于割 CUT（S,T）的容量，那么 f 是一个最大流，CUT（S,T）是一个最小割（容量最小的割）。

证明：

令割 CUT（S,T）的容量为 C，所以流 f 的流量也为 C。

假设另外的任意流 f1，流量为 c1，根据流量不超过割的容量，则 c1<=c,所以 f 是最大流。

假设另外的任意割 CUT（S1,T1），容量为 c1，根据流量不超过割的容量，所以有 c1>=c,故，CUT（S,T）是最小割。

定量 3：最大流最小割定量：

在任何的网络中，最大流的值等于最小割的容量。

结论 1：

最大流时，最小割 cut（S，T）中，正向割边的流量=容量，逆向割边的流量为 0。否则还可以增广。

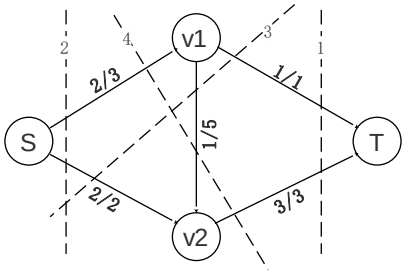

图 12

费用流

最小费用最大流、最大费用最大流

对于网络流图：G=（V，E，C，W）：

每条弧<u,v>，除了有容量限制 c(u,v)，还有单位费用 w(u,v)，当弧 <u,v>流过的流量为 f(u,v),需要花费为 f(u,v)*w(u,v)。

费用也满足反对称性 w(u,v)=-w(v,u)。

网络中总费用最小的最大流称为最小费用最大流，首先保证流最大，然后再保证费用最小。

总花费最大的最大流称为最大费用最大流。

在最大流的 EK 算法求解最大流的基础上，把 用 BFS 求解任意增广路 改为 用 SPFA 求解单位费用之和最小的增广路 即可。

需要注意的是在求最短路时，把单位费用看做边权。弧<u,v>的边权为 w(u,v),反向弧的边权为-w(u,v)。

模板题：p3381

参考代码

```
const int INF=0x7f7f7f7f;
const int N=5000+10;
const int E=50000+100;
struct Edge{
    int u,v,w,c,next;
};
Edge e[2*E];
int h[N];
int d[N];
int pre[N];//记录边
int flow[N];
int vis[N];
int n,m,s,t,Maxflow=0,Mincost=0;
int tot=-1;//可以初始化为 1，这样从 2 开始存储，0 和 1 不用，如果从 0 开始，不要忘了 h[]=-1。
inline void Add(int u,int v,int w,int c){
    e[++tot].u=u;
    e[tot].v=v;
    e[tot].w=w;
    e[tot].c=c;
    e[tot].next=h[u];
```

```
    h[u]=tot;
}
inline int spfa(){
    queue<int>Q;
    memset(d,0x7f,sizeof(d));
    memset(vis,0,sizeof(vis));
    d[s]=0;
    vis[s]=1;
    pre[s]=-1;
    flow[s]=INF;
    Q.push(s);
    while(!Q.empty()){
        int u=Q.front();Q.pop();
        vis[u]=0;
        for(int i=h[u];i!=-1;i=e[i].next){
            int v=e[i].v;
            int w=e[i].w;
            int c=e[i].c;
            if(w&&d[v]>d[u]+c){
                d[v]=d[u]+c;
                flow[v]=min(flow[u],w);
                pre[v]=i;
                if(!vis[v]){
                    Q.push(v);
                    vis[v]=1;
                }
            }
        }
    }
    return d[t]!=INF;
}
void MincMaxf(){
    Maxflow=0;
    Mincost=0;
    while(spfa()){
        Maxflow+=flow[t];
        Mincost+=flow[t]*d[t];
        int u=t;
        while(u!=s){
            int i=pre[u];
            e[i].w-=flow[t];
            e[i^1].w+=flow[t];
            u=e[i].u;//u=e[i^1].v；如果弧没存起点，就用反向弧表示
        }
    }
```

```
}
int main(){
    scanf("%d%d%d%d",&n,&m,&s,&t);
    memset(h,-1,sizeof(h));//又忘了
    for(int i=0;i<m;i++){
        int u,v,w,c;
        scanf("%d%d%d%d",&u,&v,&w,&c);
        Add(u,v,w,c);
        Add(v,u,0,-c);
    }
    MincMaxf();
    printf("%d %d\n",Maxflow,Mincost);
    return 0;
}
```

练习：5-P2045 方格取数加强版(最大费用最大流)