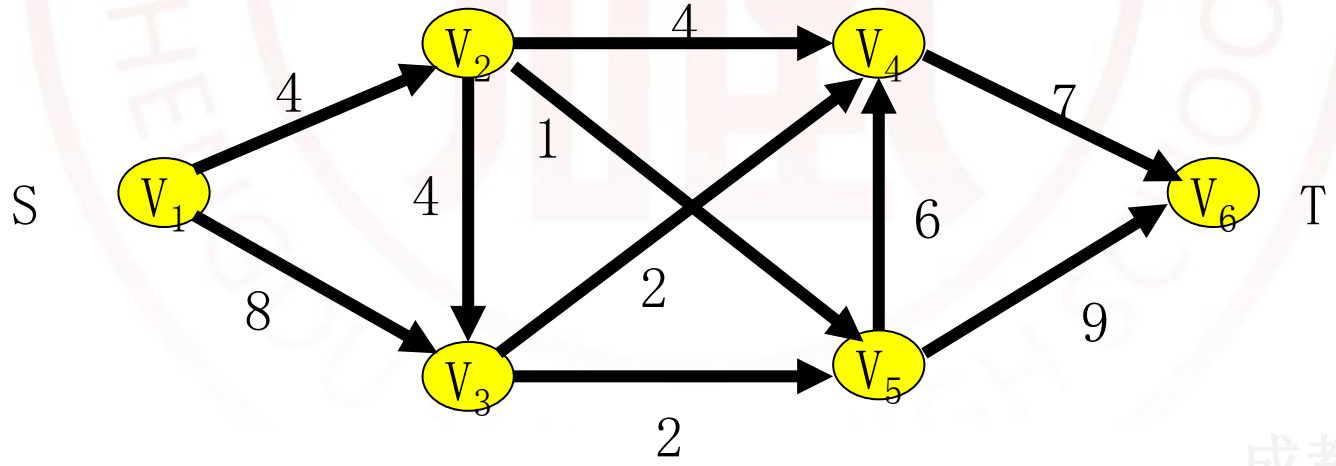
网络流问题

石室中学：梁德伟

运输网络

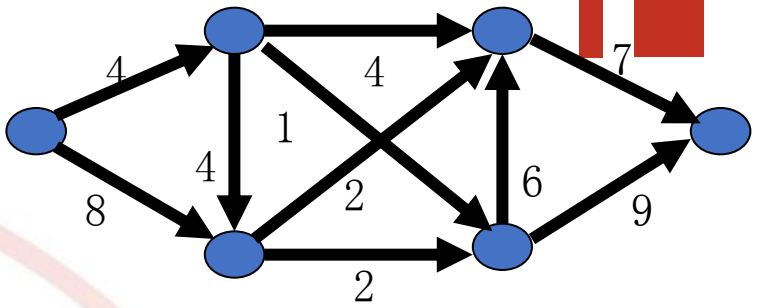
- 现在想将一些物资从S运抵T，必须经过一些中转站。连接中转站的是公路，每条公路都有最大运载量。
- 每条弧代表一条公路，弧上的数表示该公路的最大运载量。最多能将多少货物从S运抵T？

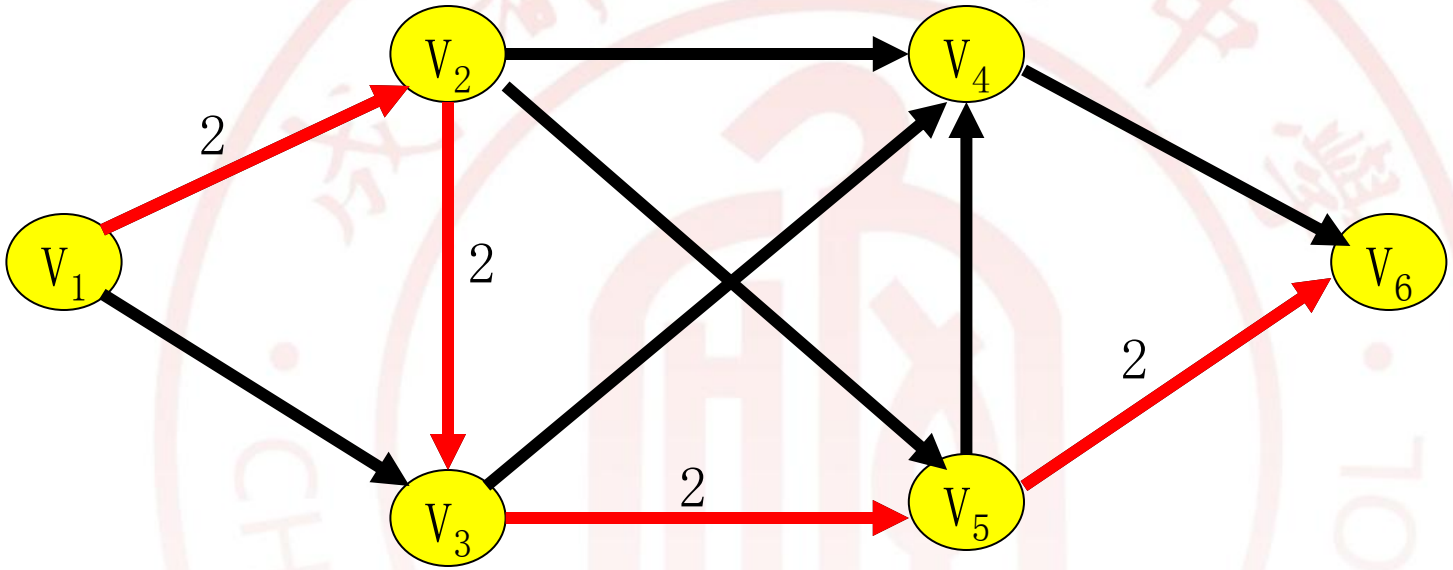


基本概念

- 这是一个典型的网络流模型。为了解答此题，我们先了解网络流的有关定义和概念。
- 若有向图G=(V,E)满足下列条件：
  1. 有且仅有一个顶点S，它的入度为零，即d-(S) = 0，这个顶点S便称为源点，或称为发点。
  2. 有且仅有一个顶点T，它的出度为零，即d+(T) = 0，这个顶点T便称为汇点，或称为收点。
  3. 每一条弧都有非负数，叫做该边的容量。边(vi, vj)的容量用cij表示。
- 则称之为网络流图，记为G = (V, E, C)

网络流





P2: $V_1 \rightarrow V_2 \rightarrow V_3 \rightarrow V_5 \rightarrow V_6$

可行流

可行流

对于网络流图G，每一条弧(i,j)都给定一个非负数fij，这一组数满足下列三条件时称为这网络的可行流，用f表示它。

1. 容量限制：每一条弧(i,j)有$f_{ij} \leqslant C_{ij}$

2. 流量平衡

除源点S和汇点T以外的所有的点vi，恒有：

$$\Sigma_j(f_{ij}) = \Sigma_k(f_{jk})$$

该等式说明中间点vi的流量守恒，输入与输出量相等。

3. 反对称性：对所有的u，v∈V，要求f(u,v)=-f(v,u)。

若给一个可行流$F=(F_{ij})$

饱和弧：网络中$F_{ij}=C_{ij}$的弧
非饱和弧：网络中$F_{ij}<C_{ij}$的弧
非零流弧：网络中$F_{ij}>0$的弧
零流弧：网络中$F_{ij}=0$的弧

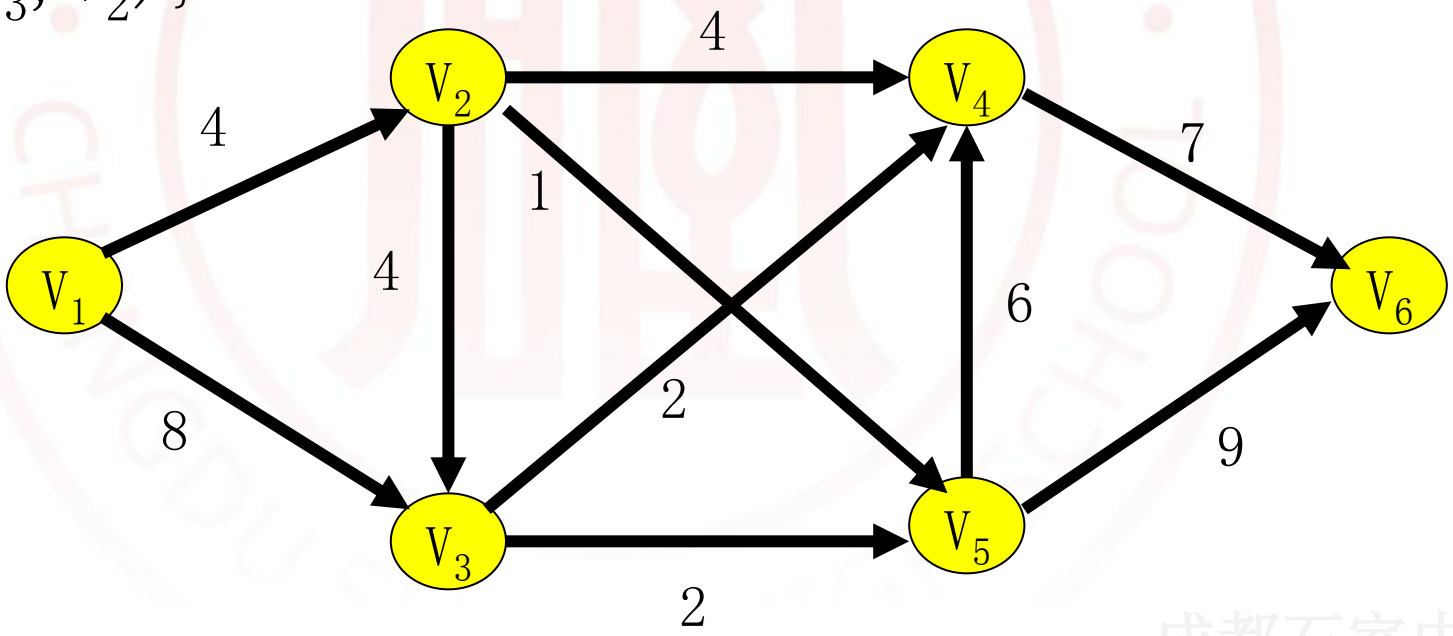
若P是网络中联结源点V_s和汇点V_t的一条路，我们定义路的方向是从V_s到V_t，则路上的弧被分成两类：

一类是弧的方向与路的方向一致，叫做前向弧。前向弧的全体记为P+。另一类弧与路的方向相反，叫做后向弧。后向弧的全体记为P-。

如图，在路$P=\{V_1, V_3, V_2, V_4, V_6\}$中

$P+=\{(V_1, V_3), (V_2, V_4), (V_4, V_6)\}$

$P-=\{(V_3, V_2)\}$



在图G中，一个由不同的边组成的序列$e_1, e_2, \cdots, e_g$，如果e_i是连接V_{i-1}与V_i($i=1,2,\cdots,g$)的，我们就称这个序列为从V_0到V_g的一条道路，数g称为路长，V_0与V_g称为这条道路的两个端点，V_i($1<=i<=g-1$)叫做道路的内点。

对于$V_{ij} \in A$　　$V_{ji} \notin A$

V_{ji}　　是边吗？

残量网络

- 为了更方便算法的实现，一般根据原网络定义一个残量网络。其中r(u, v)为残量网络的容量。
- r(u, v) = c(u, v) – f(u, v)
- 通俗地讲：就是对于某一条边（也称弧），还能再有多少流量经过。
- G_f残量网络,E_f表示残量网络的边集.

例1

原网络 (a, b)表示(流量f, 容量c)

图1

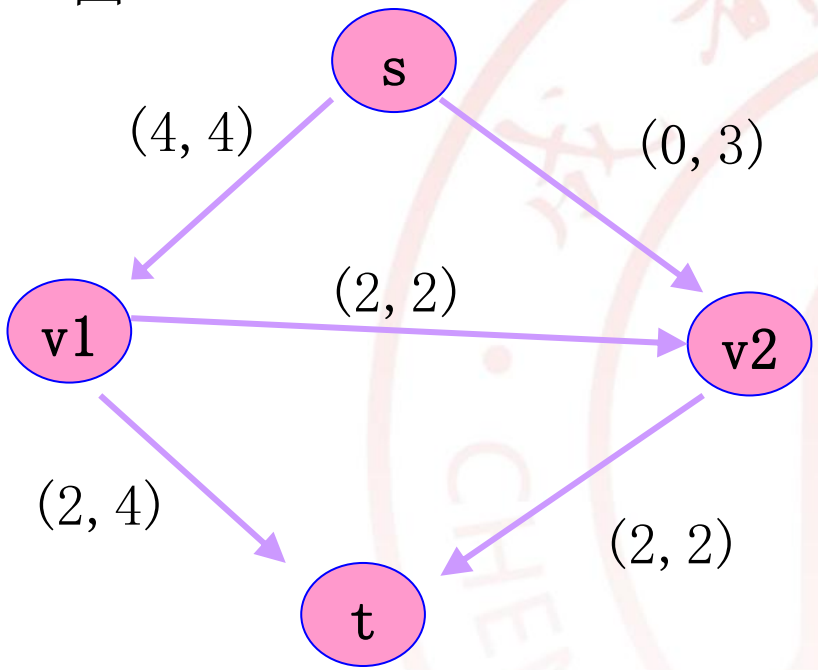
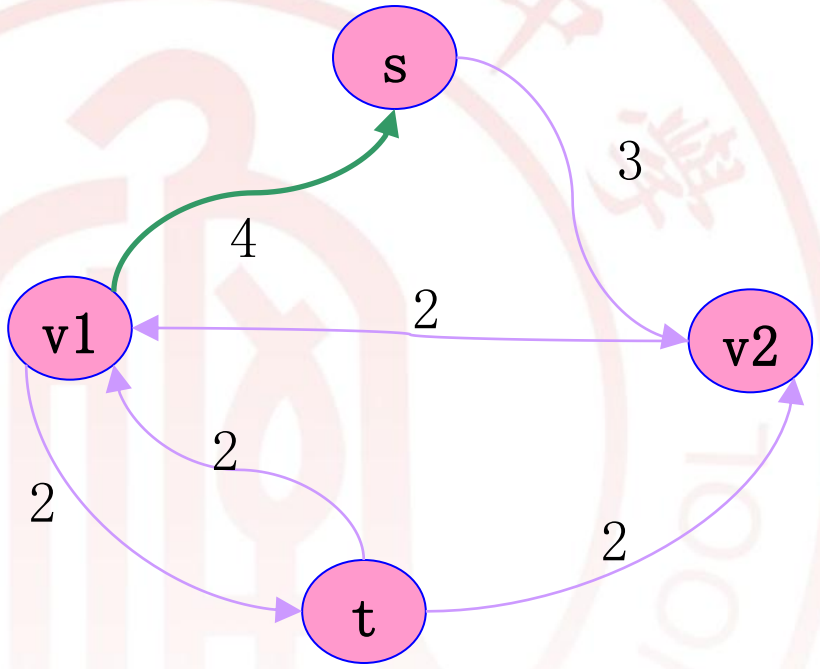
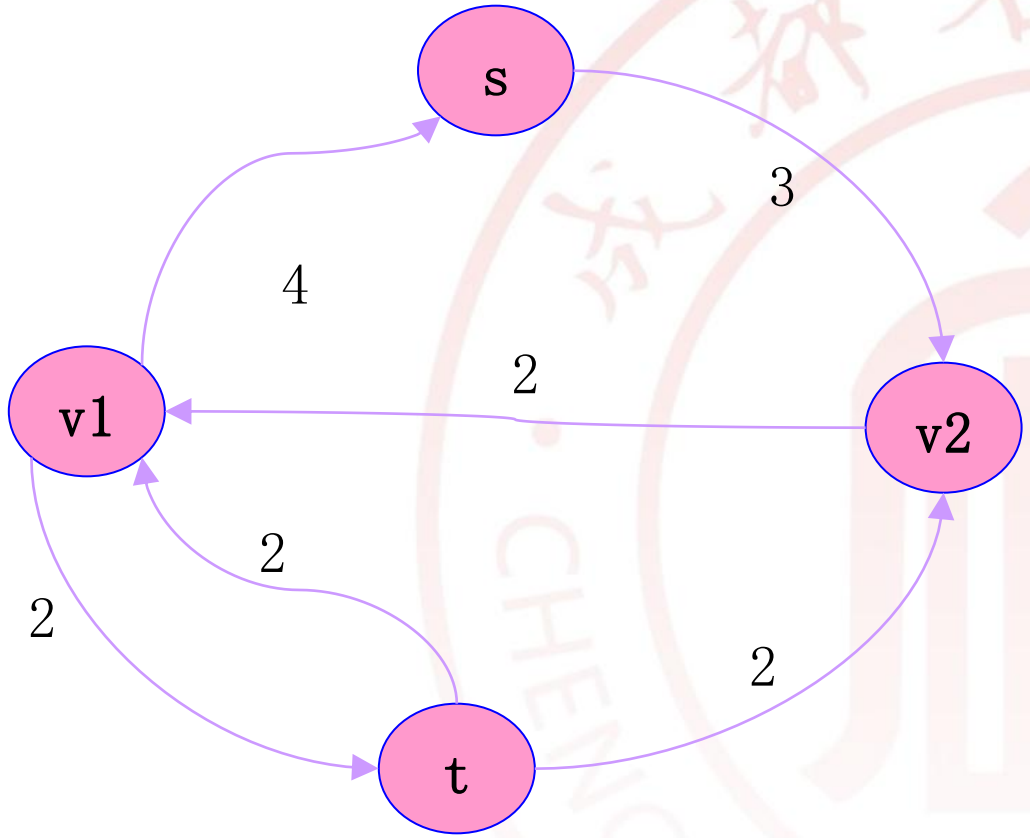


图2



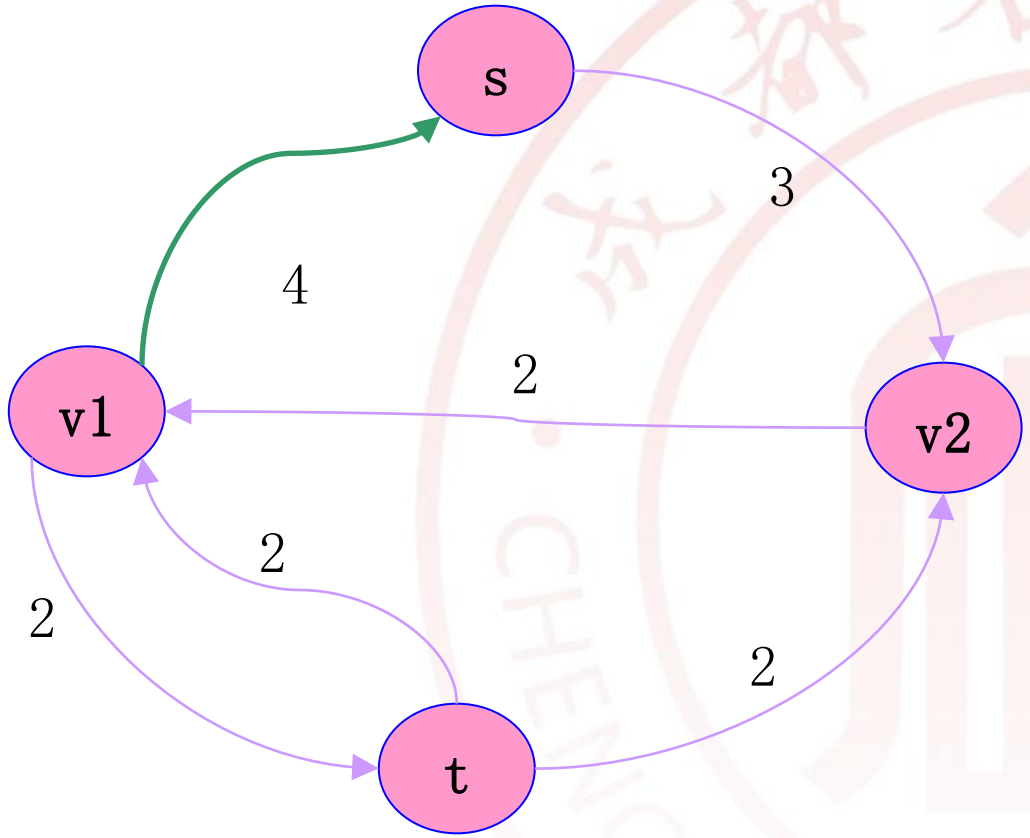
残量网络（如果网络中一条边的容量为0, 则认为这条边不在残量网络中。r(s, v1)=0, 所以就不画出来了。另外举个例子：r(v1, s) = c(v1, s) - f(v1, s) = 0 - (-f(s, v1)) = f(s, v1) = 4.

例1



- 从残量网络中可以清楚地看到：
- 因为存在边(s, v2) = 3, 我们知道从S到v2还可以再增加2单位的流量；
- 因为存在边(v1, t) = 2, 我们知道从v1到t还可以再增加2单位的流量。

后向弧



- 其中像(v1, s)这样的边称为后向弧，它表示从v1到s还可以增加4单位的流量。
- 但是从v1到s不是和原网络中的弧的方向相反吗？显然“从v1到s还可以增加4单位流量”这条信息毫无意义。那么，有必要建立这些后向弧吗？

为什么要建立后向弧

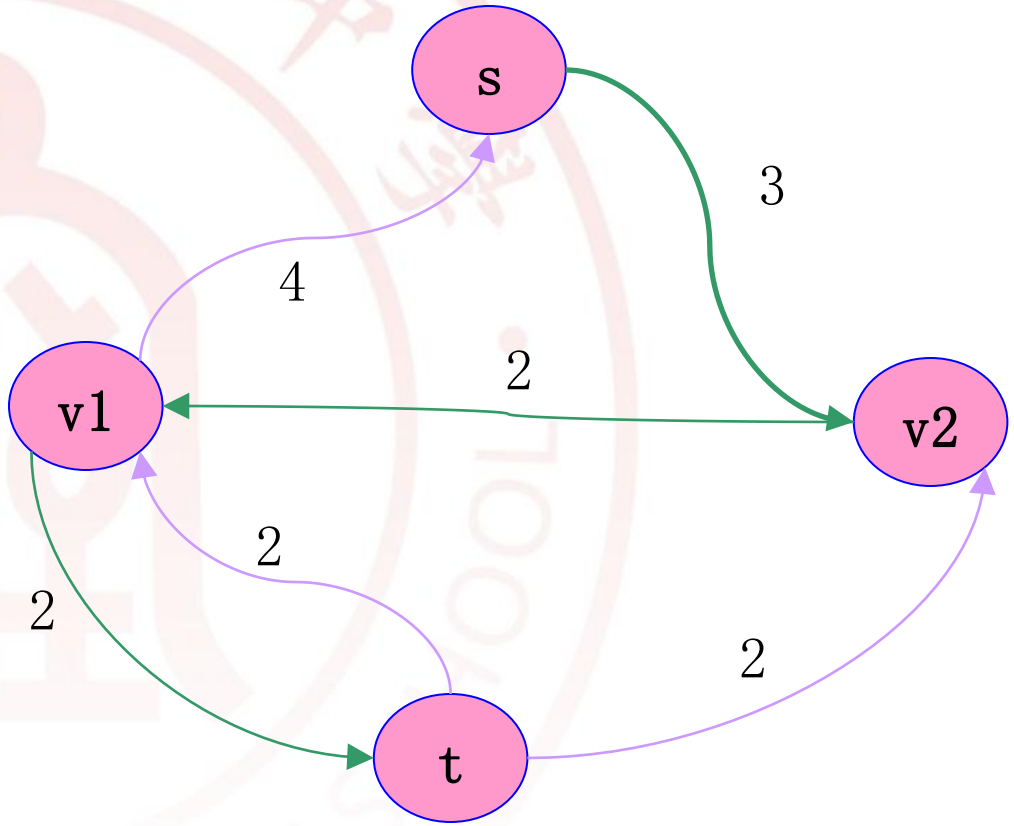
- 显然，例1中的画出来的不是一个最大流。
- 但是，如果我们把s -> v2 -> v1 -> t这条路径经过的弧的流量都增加2,就得到了该网络的最大流。
- 注意到这条路径经过了一条后向弧:(v2,v1)。
- 如果不设立后向弧，算法就不能发现这条路径。
- 从本质上说，后向弧为算法纠正自己所犯的错误提供了可能性，它允许算法取消先前的错误的行为（让2单位的流从v1流到v2)

为什么要建立后向弧

- 当然,可以把上面说的情况当成特殊情况来处理。但使用后向弧可以使编程简单许多.
- 注意,后向弧只是概念上的,在程序中后向弧与前向弧并无区别.
- 我们为所有边设置反向边，并将其流量与正向边同步。为了保持T->S边流量为0，每当由于增广操作，T->S边流量上升，其反向边也同步上涨，而显然在残量网络中反向边是沿S->T方向的，这就又出现了一条增广路，计算机对此无法容忍，下调它的流量，我们的T->S边流量也随之回到了0，达到了目的。

增广路

- 增广路定义：在残量网络中的一条从s通往t的路径，其中任意一条弧(u, v)，都有r[u, v]>0。
- 绿色的即为一条增广路。

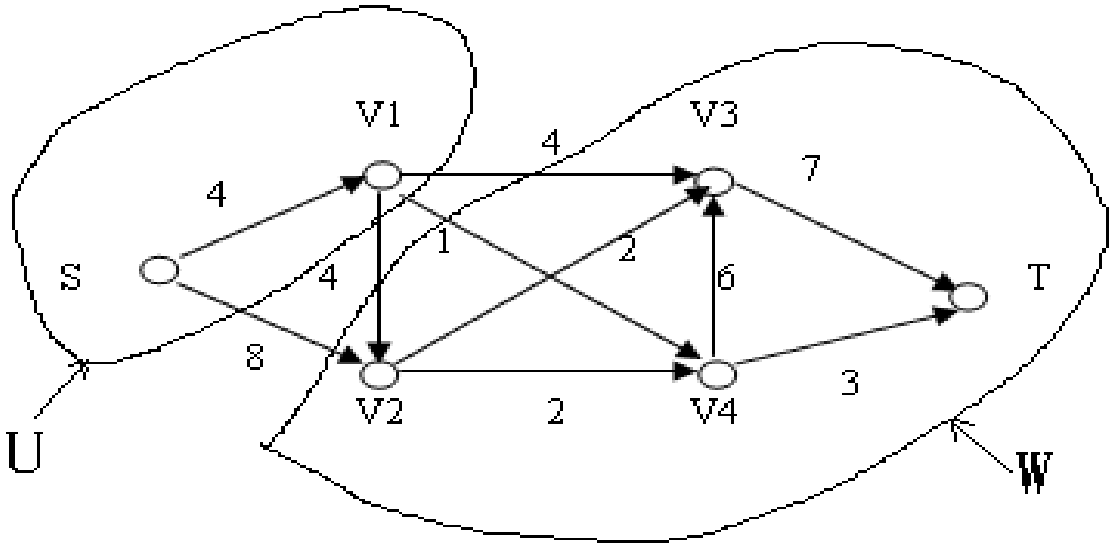


割

- G = (V, E, C)是已知的网络流图，设U是V的一个子集，W = V\U，满足S ∈ U，T∈W。即U、W把V分成两个不相交的集合，且源点和汇点分属不同的集合。
- 对于弧尾在U，弧头在W的弧所构成的集合称之为**割切**，用（U，W）表示。把割切（U，W）中所有弧的容量之和叫做此割切的容量，记为C（U，W），即：

$$C(U,W)=\sum_{\substack{i\in U\\ j\in W}} c_{ij}$$

割



- **例中，令**U = {S, V1}**，则**W = {V2, V3, V4, T}**，那么，**
- C(U, W) = <S, V2> + <V1, V2> + <V1, V3>+<V1, V4>
  =8+4+4+1=17

增广路算法

- 增广路算法：每次用BFS找一条最短的增广路径，然后沿着这条路径修改流量值（实际修改的是残量网络的边权）。当没有增广路时，算法停止，此时的流就是最大流。

找增广路可以用DFS和BFS，DFS容易被卡，所以采取BFS比较好

流量算法的基本理论

- 定理1：对于已知的网络流图，设任意一可行流为f，任意一割切为(U, W)，必有：V(f) ≤ C(U, W)。
- 定理2：可行流f是最大流的充分必要条件是：f中不存在增广路。
- 定理3：整流定理。

  如果网络中所有的弧的容量是整数，则存在整数值的最大流。
- 定理4：最大流最小割定理。

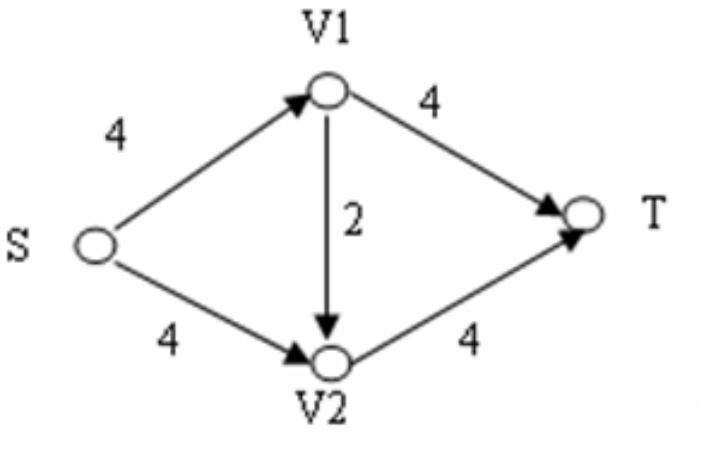
  最大流等于最小割，即max V(f) = min C(U, W)。

最大流算法Edmonds-Karp

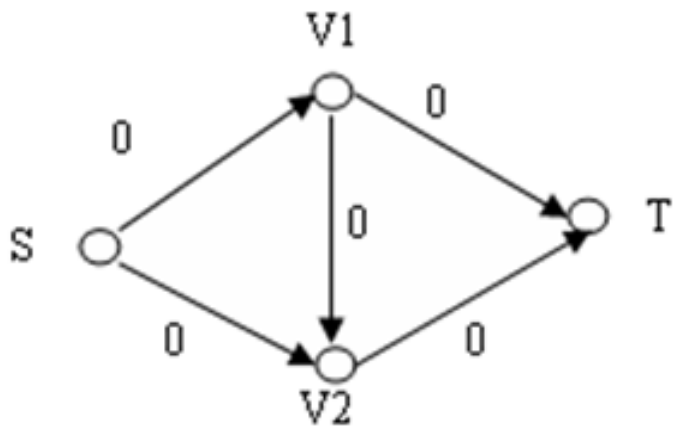
- 第1步，令$x=(x_{ij})$是任意整数可行流，可能是零流，给s一个永久标号$(-, \infty)$。
- 第2步(找增广路)，如果所有标号都已经被检查，转到第4步。

  找到一个标号但未检查的点i，并做如下检查，
  - 对每一个弧(i, j)，如果$x_{ij}<C_{ij}$，且j未标号，则给j一个标号$(+i, \delta(j))$，其中，$\delta(j)=\min\{C_{ij}-x_{ij}, \delta(i)\}$
  - 对每一个弧(j, i)，如果$x_{ji}>0$，且j未标号，则给j一个标号$(-i, \delta(j))$，其中，$\delta(j)=\min\{x_{ji}, \delta(i)\}$
- 第三步(增广)，由点t开始，使用指示标号构造一个增广路，指示标号的正负则表示通过增加还是减少弧流量来增加还是减少弧流量来增大流量，抹去s点以外的所有标号，转第二步继续找增广轨。
- 第四步(构造最小割)，这时现行流是最大的，若把所有标号的集合记为S，所有未标号点的集合记为T，便得到最小容量割(S, T)。

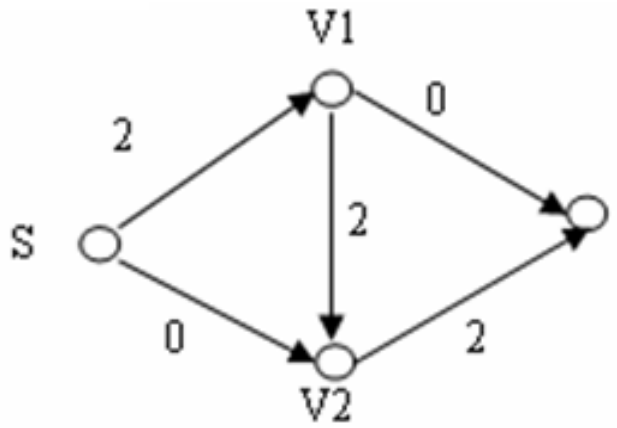
实例



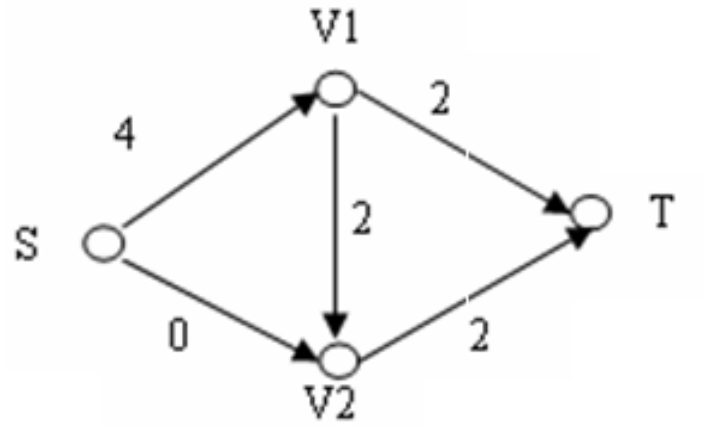
一个简单的网络流图



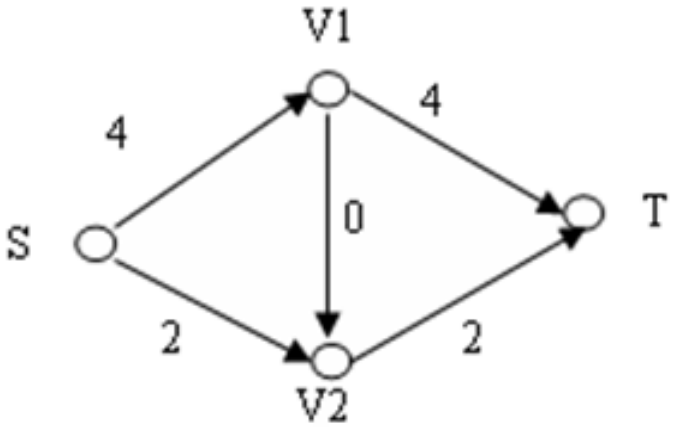
取一个初始可能流（零流）



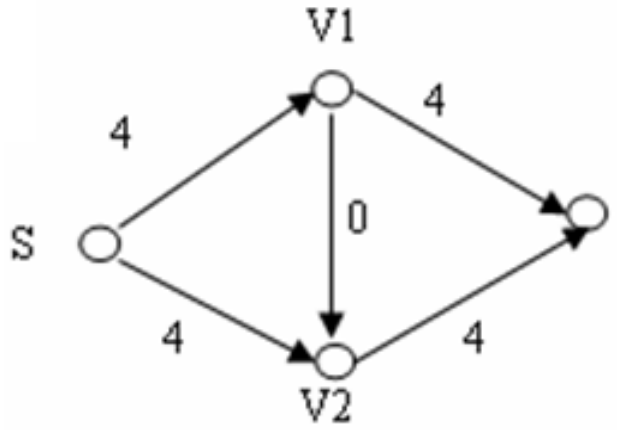
第1次增广(S, V1, V2, T)



第二次增广(S, V1, T)



第三增广(S, V2, V1, T)



第四次增广(S, V2, T)

复杂度分析

- 设图中弧数为m,每找一条增广轨最多需要进行2m次弧的检查。如果所有弧的容量为整数，则最多需要v(其中v为最大流)次增广，因此总的计算量为O(mv)。

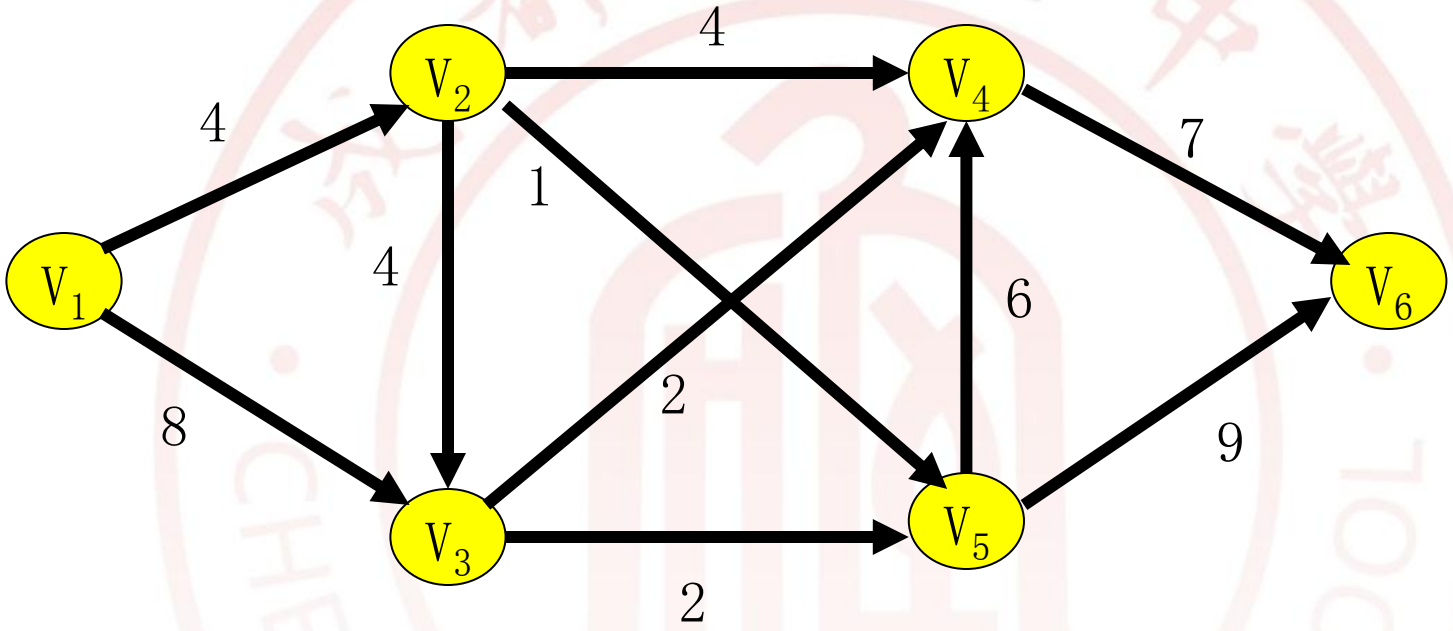
2634模板题

- 给定 n 个点，m 条边，给定每条边的容量，求从点 s 到点 t的最大流。

```
int bfs(int s,int t){
    int a[N];
    memset(a,0,sizeof(a));
    memset(path,-1,sizeof(path));
    queue<int>q;    q.push(s);
    a[s]=INF;
    while(!q.empty()){
        int u=q.front();q.pop();
        for(int i=first[u];i;i=nxt[i]){
            Edge&x=e[i];
            if(!a[x.v] && x.f < x.c){//没有访问，且有残余
                path[x.v]=i;
                a[x.v]=min(a[u],x.c-x.f);
                q.push(x.v);
            }
        }
        if(a[t])return a[t];
    }
    return 0;
}
```

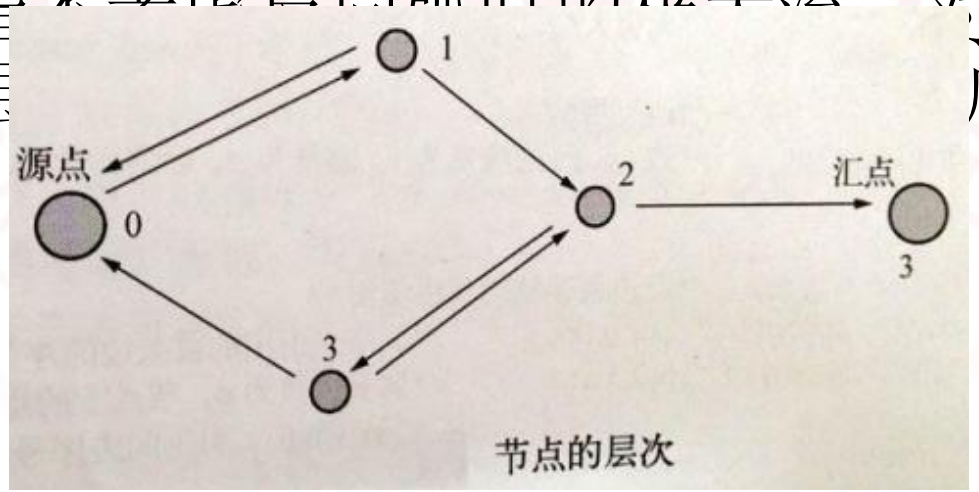
```
int EK(int s,int t){
    int flow=0;
    while(1){
        int tmp=bfs(s,t);
        if (!tmp)break;
        for(int i=t;i!=s;i=e[path[i]].u){
            e[path[i]].f+=tmp;
            e[path[i]+1].f-=tmp;//反向边取相反
        }
        flow+= tmp;
    }
    return flow;
}
```

```
const int INF=1e9;
#define N 100010
struct Edge{
    int u,v,c,f;
}e[N<<1];
int first[N]={0},nxt[N<<1],cnt=0;
int n,m,st,ed;
int path[N];
void add(int u,int v, int c){
    e[++cnt].u=u;e[cnt].v=v;e[cnt].c=c;
    nxt[cnt]=first[u];first[u]=cnt;
}int main(){
    scanf("%d%d%d%d",&n,&m,&st,&ed);
    for(int i=1;i<=m;i++){
        int from,to,cl;
        scanf("%d%d%d",&from,&to,&cl);
        add(from,to,cl);
        add(to,from,0);//反向边
    }
    cout<<EK(st,ed);
    return 0;
}
```



Dinic算法——按层次计算最大流

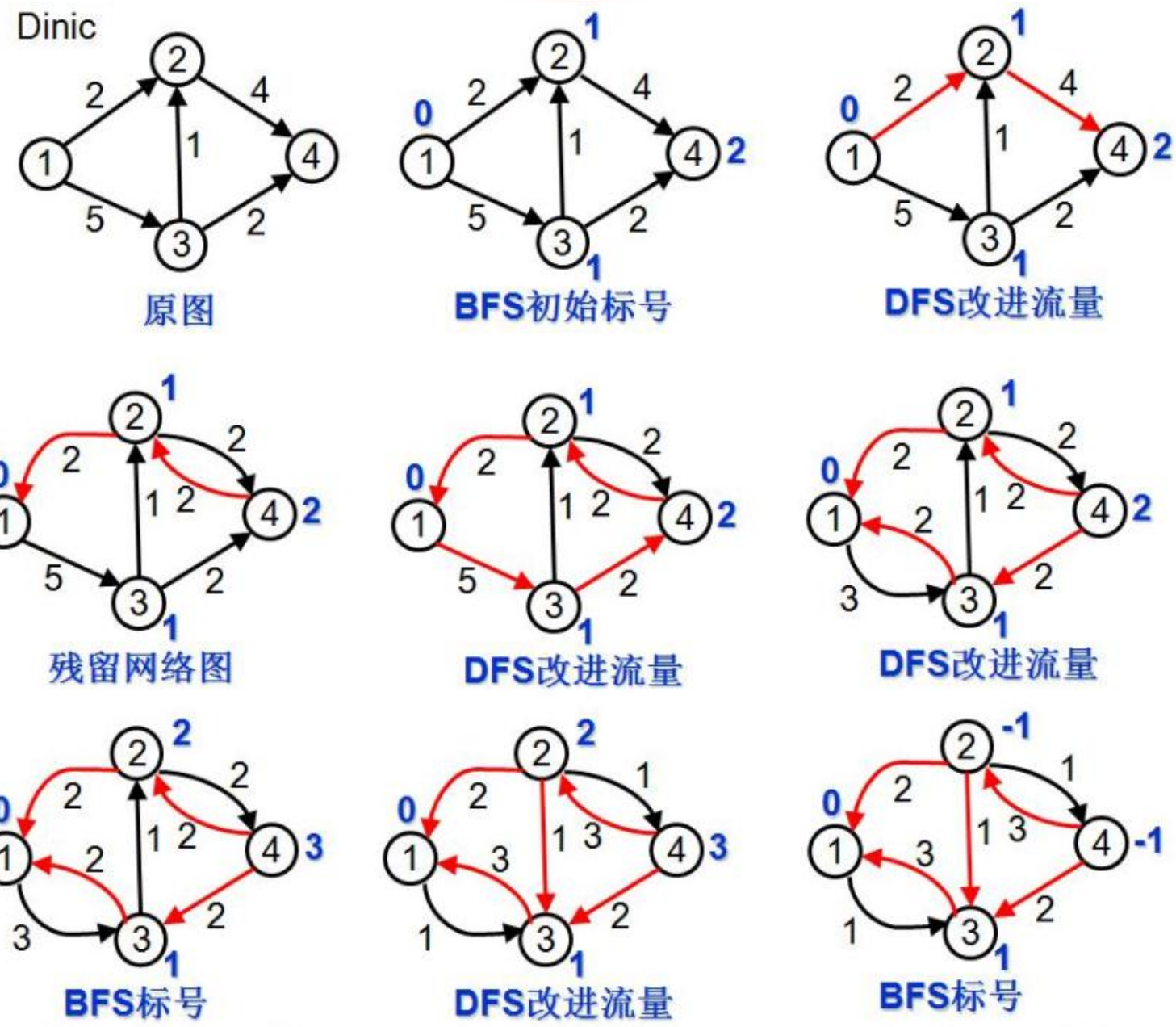
- **基本思想**：不停的用BFS构建层次图，然后用阻塞流来增广。
- **层次图**：假设在残留网络中，起点到结点u的距离是dist[u]，就把dist[u]看做是结点u的“层次”。保留每个点出发到下个层次的弧，即只保留dist[u]+1=dist[v]的边(u,v),得到的图就是层次图。层次图上任意路径都是：起点->层次1->层次2->…->T　的顺序，而且每条这样的路都是S-T的最短路。
- **阻塞流**：就是指不考虑反向弧时的极大流，对应到程序就是从起点开始在层[illegible]广。



Dinic算法——按层次计算最大流

Dinic算法的基本流程：

Dinic算法——按层次计算最大流



```cpp
int bfs(){
    queue<int>q;
    q.push(S);
    memset(dis+1,-1,n<<2);
    dis[S]=0;
    while(!q.empty()){
        int u=q.front();q.pop();
        for(int i=first[u];~i;i=e[i].nxt){
            int &v=e[i].v;
            if(e[i].c>0&&dis[v]==-1){
                dis[v]=dis[u]+1;
                q.push(v);
                if(v==T)return 1;
            }
        }
    }
    return 0;
}
```

```cpp
int dfs(int x,int f){//f最小残量
    if(x==T||f==0)return f;
    int used=0;//当前点已经流出的流量
    for(int& i=cur[x];~i;i=e[i].nxt){//
        if(e[i].c && dis[e[i].v]==dis[x]+1){//如果有容量且是下个层次
            int w=dfs(e[i].v,min(f,e[i].c));//
            if(!w)continue;
            used+=w;f-=w;
            e[i].c-=w;e[i^1].c+=w; //正反边
            if(f==0)break;//如果流量用完就不搜 程序及时结束会提高效率
        }
    }
    if(!used)dis[x]=-1;
    return used;
}
int dinic(){
    int mflow=0;
    while(bfs()){
        memcpy(cur+1,first+1,n<<2);
        mflow+=dfs(S,INF);
    }
    return mflow;
}
```

时间复杂度

- 对于n个点，m条边。
- 最多有n个阶段，即最多构建n个层次，每个层次用bfs一遍即可得到，1次bfs是M，所有构建层次图总时间O(NM)
- 一次dfs是O（nm），最多n次dfs，所以找可增广需O(N*N*M)也是整个算法的复杂度
- Dinic算法是基于距离标号算法

SAP算法（shortest Augmenting Paths）

- **距离标号**：即某个点到会点最少边的数量。
- 设点i的距离标号是dis[i]，那么将满足dis[i]=dis[j]+1的边(i,j)叫**允许弧**。
- 找增广的过程就是从S开始沿着“允许弧”向前走。如果走不动了呢？边增广边修改。
- 具体是：在从结点i往回走的时候，把dis(i)修改为min{dis[j]|(i,j)是残留网络的弧}+1即可。
- 注意，如果残留网络中从i出发没有弧，则设dis[i]=n

SAP算法（shortest Augmenting Paths）

GAP优化：

用gap[]数组维护每个距离标号下的结点总数

gap[x]=y:表示残留网路中dis[i]=x的个数为y，若重标号使得gap[]中原标号变为0，则停止算法。

由于残留网路的修改只会使dis[i]越来越大(因为修改前dis[i]<dis[j]+1，而修改后会存在dis[i]=dis[j]+1，因此变大了)，所有dis[i]是单调递增的，这就提示我们，如果gap[k]=0,出现了“断层”，即没有dis[j]=k,而有dis[j]=k±1，这时候必定无法再找到增广路。可以这样思考，现在的i满足dis[i]=k+1，发现没有一个dis[j]=k，因此就会尝试去调整dis[i]，但是dis[i]是单调递增的，只会越来越大，所有k这个空缺便永远不会被补上，也就是说无法再找到增广路。

SAP算法（shortest Augmenting Paths）

算法步骤：

1、初始化dis[i]=0,gap[0]=N

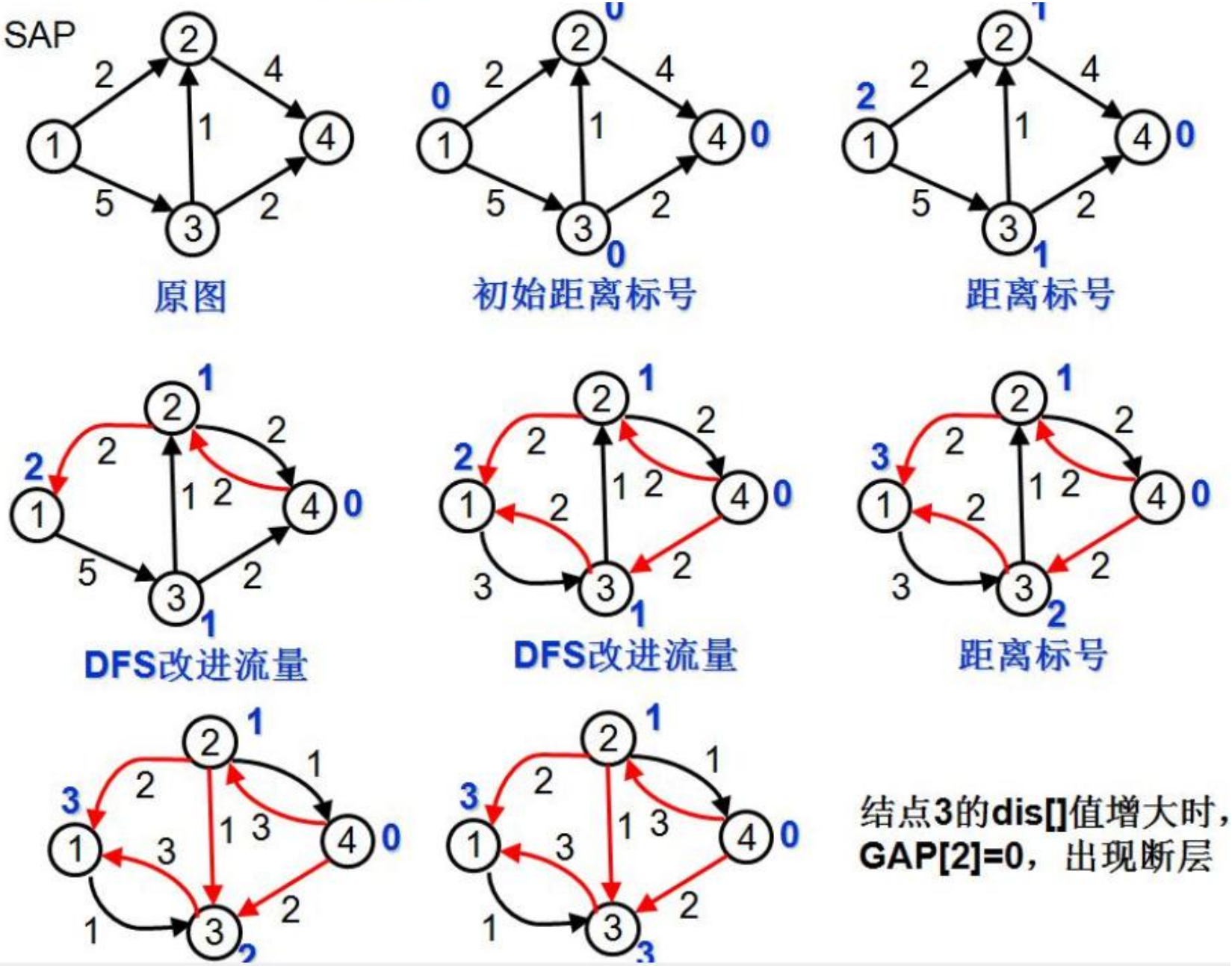
2、在初始标号上，不断沿可行弧找增广路，一般用dfs，对于可行弧(I,j)，有dis[i]=dis[j]+1

3、遍历完成当前节点后，为了使下次再来的时候有路（要满足距离标号的性质：不超过真实的距离），重新对当前节点标号为：dis[i]=min{dis[j]|(i,j)}+1

4、当dis[s]>=总节点个数N时，算法结束。

SAP算法（shortest Augmenting Paths）

- 示例图



```
void bfs(int s,int t){
    queue<int>q;q.push(t);
    memset(dis,0,sizeof(dis));memset(gap,0,sizeof(gap));
    memcpy(cur,first,sizeof(cur));
    dis[t]=1;gap[1]++;
    while(!q.empty()){
        int now=q.front();q.pop();
        for(int i=first[now];i;i=e[i].nxt){
            if(dis[e[i].v])continue;
            dis[e[i].v]=dis[now]+1;
            q.push(e[i].v);
            gap[dis[now]+1]++;
        }
    }
}
```

```
int dfs(int s,int t,int x,int f){
    if(x==t)return f;
    int used=0,w;
    for(int &i=cur[x];i;i=e[i].nxt){
        if(e[i].w && dis[e[i].v]+1 == dis[x]){
            w=dfs(s,t,e[i].v,min(f-used,e[i].w));
            used+=w;e[i].w-=w;e[i^1].w+=w;
            if(used==f)return f;
        }
    }
    if((--gap[dis[x]])==0)dis[s]=n+1;//如果当前x这一层的标号数量为0，说明出现了断层，则直接结束算法
    ++gap[++dis[x]];//流量没有用完，且无法找出增广路，说明当前距离标号过时，则提高标号
    cur[x]=first[x];//当前弧恢复为first
    return used;
}
int isap(int s,int t){
    bfs(s,t);
    int ret=0;
    while(dis[s]<=n){
        ret+=dfs(s,t,s,inf);
    }
    return ret;
}
```

1668 Drainage Ditches

题意：n个点m条边，每条边有最大容量，求从1到n的最大流量

2303 「网络流 24 题」搭配飞行员

题意：n个点的二分图，求最大匹配

最大流求二分图最大匹配建模方法

增加源S，汇T

1、S向所有X点连一条容量为1的边

2、所有Y点向T连一条容量为1的边

3、原X到Y的边为容量为1的边

4、求最大流，最大流即最大匹配数，所有满流边是一组可行解

理解：S向X连一条容量为1的边，即在一条从S->T的可行流中，X只能被选一次

2351 [USACO07OPEN]吃饭Dining

- 有N头牛，F种食物和D种饮料，每头牛有多种喜欢的食物和饮料，每头牛只可以吃一种食物和饮料，且每种食物和饮料都只能被一头牛吃掉。一头牛满意当且仅当它吃到满意的食物并且喝到想喝的饮料，问最多可能让多少头牛满意。

分析

- 本题是一个三分匹配问题。对于一般的二分匹配，通常是左边一个点集表示供应与源点连接，右边点集表示需求与汇点连接。
- 在本题中，供应有两个资源，需求只有一个点集。
- 考虑最大流模型的建模原理，他的正确性依赖于一条S->T的流代表一种实际方案。 在本题中需要用S-T流将一头牛与它喜欢的 食物与饮料 连 起来，而食物与饮料没有直接联系，自然想到了将牛放中间，两边是食物与饮料，有S,T串起来构成一种分配方案，于是有如下建模：
- 每种食物 i 作为一个点并连边(s, i, 1)，每种饮料 j 作为一个点并连边(j, t, 1)，将每头牛 k拆成两个点 k’，k’’ 并连边(k’，k’’，1)，(i, k’，1)，(k’’，j, 1)，其中 i, j 均是牛 k 喜欢的食物或饮料。求一次最大流即为结果。
- **注：一头牛拆分为两个点，中间一条容量为1的边可以保证每头牛只满足一次，如果不拆分，每头牛可以被多次满足。**

1243 蜥蜴

- 在一个r行c列的网格地图中有一些高度不同的石柱，一些石柱上站着一些蜥蜴，你的任务是让尽量多的蜥蜴逃到边界外。
- 每行每列中相邻石柱的距离为1，蜥蜴的跳跃距离是d，即蜥蜴可以跳到平面距离不超过d的任何一个石柱上。石柱都不稳定，每次当蜥蜴跳跃时，所离开的石柱高度减1（如果仍然落在地图内部，则到达的石柱高度不变），如果该石柱原来高度为1，则蜥蜴离开后消失。以后其他蜥蜴不能落脚。任何时刻不能有两只蜥蜴在同一个石柱上。

分析

- 对于每根石柱，采取一分为二的想法，即把一个点分为两个点（可抽象为石柱底部到顶部），其连线容量限制为石柱高度。
- 超级源与所有有蜥蜴的点相连，容量为1。
- 超级汇与地图内所有能跳出的点相连，容量为INF。
- 对于地图内任意两个石柱，如果间距小于d，就将其中一根石柱的顶部与另一根石柱的底部相连，其连线容量为INF。
- 构图完成，剩下就是跑一遍最大流，然后用蜥蜴数量减去最大流就是最终结果。

- 3090
- 4077