最大流入门教程

永远的魔灵

Contents

Part I
前言

随着OI日新月异的发展，原来只有高手才能掌握的网络流的各种算法和应用现在早就飞入寻常百姓家中。平时和同学开玩笑，都说现在随便找一个OIer，他可能不会搜八皇后，但是他绝对知道网络流。

虽然如此，我觉得更有写一篇介绍网络流的文章，介绍我所知道的网络流的冰山一角，以及在信息学竞赛中的一些应用，权当是抛砖引玉，同时也可以给正在学习或者希望学习相关问题的同学一点参考资料。

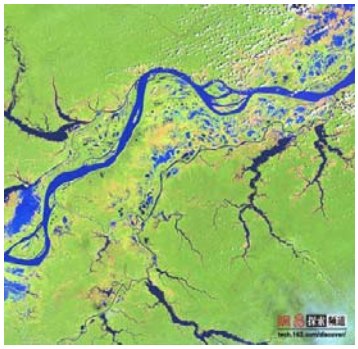
本人水平有限，若有遗漏或者错误，希望各位大牛不吝赐教。谢谢。

Mail:g201513@gmail.com
QQ:195724606

Part II
网络流的定义

1 初识网络流

网络流(network flow)，顾名思义就是在网络中的流。流，可以形象地理解为水流，当然也可能是其他的类似物质，比如痰，为了挽救我刚吃下的晚饭，我们还是将它想象为水吧。网络流问题，可以形象地理解为在一个管道网络中（当然也不一定是水管，比如水槽，河道或者尿道），一些固定的管道连接了一些管道结点，每个管道所能通过的水流速度有限制，在这些约束条件下所能规划的问题便是网络流问题。



在简单的网络流模型中，这些结点中有一个管道结点被称为源(source)，可以想象有无穷无尽的水从这里流入，还有一个结点被称为汇(sink)，也可以想象汇就像一个黑洞流到这里的水都被它吸收殆尽。

所谓最大流问题，就是通过指派每个管道中的水的流量，使得最后流入汇的水流量最大。你可以想象你是那个贪吃的黑洞(汇)，那么自然是吃得越多身体越壮，如果你正在减肥，请忽略这个比喻。

最小费用最大流问题，则是每条水管都有一个费用，当然是流得越多，交费就越多，你希望在保证流量最大的前提下，保证自己的腰包重量，那么就需要研究研究这个问题了。不过在本文中我们不讨论这个问题，你的腰包你自己去管吧。自然，也有笨蛋想去求最大费用最大流，这人不是神志不清，就是家财万贯没地方花钱，我们也就不提了。

2 定义

更加形式化地，给出下列定义：

Definition 2.1 *流网络(flow network)*$G = (V, E)$*为一个有向图，其中每条边*$e < u, v > \in E$*都有容量*$c(u, v) \geq 0$*或*$c(e) \geq 0$*为叙述方便，下文都以二元组(u,v)代指e¡u,v¿，且默认图为完全图且没有重边，若边(u,v)在原图中不存在，则定义*$c(u, v) = 0$*，若(u,v)在原图中出现重边，则定义c(u,v)为其所有边容量和。*

V中包含两不同点源和汇。

这是说，还是那个管道系统，如果结点u到结点v存在一条管道，则一个结点到另外一个结点的容量定义为管道的容量，若不存在，我们假设他们之间有一条容量为0的，也就是无限细的管道，这不会影响任何问题。如果一

个结点到另外一个结点有多条管道，那么我们把多条管道并起来，换成一条大管道，也没有任何的问题。

Definition 2.2 *流网络的流(flow)为一个函数$f : E \to R$，满足容量限制和流量平衡。为了叙述和算法方便，我们还定义流量满足反对称性。*
(1)容量限制：对于所有(u,v)，满足$f(u,v) \leq c(u,v)$;
(2)反对称性：对于所有(u,v)，满足$f(u,v) = -f(v,u)$。
(3)流量守恒：对于所有点$u \in V$ and $u \neq source$ and $u \neq sink$，满足f(u,V)=0;*

容量限制是说管道里面的水流自然不能超过管道的限制，否则水管暴了，可没有好日子过。

流量守恒便是不允许哪个结点有汇的权利，私自藏水，否则汇就没喝的了，全被其他结点抢跑啦！

Definition 2.3 *流f的值*

$$|f| = f(V, sink)$$

说白了，流的值就是汇喝了多少水了，最大流即为值最大的流。

自然，汇喝多少，源就送出了多少：

Theorem 2.4

$$|f| = f(V, sink) = f(source, V)$$

我们还可以定理流的运算，加法和标量积：[†]

Definition 2.5 *f=f1+f2，则表示对于所有(u,v)，都有f(u,v)=f1(u,v)+f2(u,v)，f在满足容量限制的条件下成立;*

Definition 2.6 *$f = \alpha g$，则表示对于所有(u,v)，都有$f(u,v) = \alpha g(u,v)$，同样f在满足容量限制的条件下才成立。*

可以知道，若$f1+f2$存在，$|f1+f2| = |f1|+|f2|$，若αf存在，$|\alpha f| = \alpha|f|$

Definition 2.7 *对于流网络G，和G的流f，可以定义他的残量网络[‡]G_f(residual network)：*

对于(u,v)，$c_f(u,v) = c(u,v) - f(u,v)$

---

[*]为叙述方便，若函数参数为定义类型的集合，即为求和符号，如f(u,V)则表示$\sum_{v \in V} f(u,v)$

[†]在本文中没啥应用，好玩而已

[‡]又称剩余网络，残余网络

根据定义，残量网络仍然是流网络。并且为原来的流网络减去已经有流的部分，故名残量，即为剩下的网络。

在残量网络中，最重要的定义为增广路：

Definition 2.8 *残量网络G_f中，长度为k的路径$P = \{u_1, u_2, u_3 \dots u_k\}$为增广路的充要条件为$u_1$为源，$u_k$为汇且对于所有$i \geq 1$且$i + 1 \leq k$都有$c_f(u_i, u_{i+1}) \geq 0$*

可以想象，增广路就是一条从源到汇的空旷的管道通路，我们可以沿着增广路将流量增大，这个过程叫增广。

Theorem 2.9 *增广路定理：对于流f，若残量网络G_f不存在增广路，则f为流网络G的最大流。*

这个定理是很多算法的理论依据，可以非常直观地想象当已经没法找到空隙继续增大流量的时候，整个流就是最大流了。当然若需要具体证明，请参见section4。

Definition 2.10 *定义流网络G的切割(cut)，为流网络的点集V的一个划分$[S,T]$，且$source \in S, sink \in T$。从S到T的边称为割边。*

切割的容量为所有割边容量之和。切割的流量定义为$f(S,T)$，即为所有从S到T的边的流量之和。

切割，形象地，就是说一刀将整个辛苦搭建的管道系统切成两半的切痕。看到这里不要忙着难过，只要这一刀不切下来，比划比划还是可以的，实际上切这一刀的方式和网络中的流可有重大的联系。

Theorem 2.11 *最小切割最大流定理：在网络G中$[S,T]$为最小的切割，f为最大的流，则有$|S| = |f|$*

这个定理也及其重要，是证明增广路定理的有利武器，并且也是很多模型中经常用到的工具。

直观地理解为，切这一刀所花的最小力气，和这个网络的最大流，实际上是相等的。更直观地，如果将整个网络想象成一个大管子，那么大管子的流量自然是最细的部分，砍一刀花的力气最小的自然也是最细的部分。当然，具体证明还是参见section4。

3 部分证明

3.1 Theorem 2.4

Theorem 2.4的证明如下：[§]

由反对称性，可知：

Lemma 3.1

$$\sum_{u\in V, v\in V} f(u,v) = 0$$

又由流量守恒，可知：

Lemma 3.2

$$\sum_{u\in V, v\in V \ and \ u\neq source \ and \ v\neq sink} f(u,v) = 0$$

根据上述引理，Theorem 2.4可得证。

3.2 Theorem 2.9 and Theorem 2.11

Theorem 2.9和Theorem 2.11的证明如下：

Lemma 3.3 *对于任意一个切割，* $f(S,T) = |f|$

Lemma的证明：$f(S,T) = f(S,V) - f(S,S) = f(S,V) = f(s,V) + f(S - \{s\}, V) = f(s,V) = |f|$

下面，我们只需要证明以下三个命题等价：

- f为最大流；
- (2)残量网络中无增广路；
- (3)存在一个切割[S,T]，$|f| = |[S,T]|$

证明：

$1 \to 2$	反证法。若存在，则可以沿他增广，得到更大的流。
$2 \to 3$	在残量网络中不存在$s-t$通路。定义s可达点集为S，T为V-S，则$\|f\| = \|[S,T]\|$。
$3 \to 1$	可知$\|[S,T]\| \geq \|f\|$，又由于令等号成立的$[S,T]$存在，则得证。

综上所述，Theorem 2.9和Theorem 2.11得证。

[§]本文的证明可能不够科学严谨，望见谅，证明亦可略去不看

Part III
网络最大流的算法

4 最大流算法概述

好啦，上面的那么多都是废话，下面才是关键的。具体怎么求网络最大流呢，自然有多种算法可供选择，在这里主要介绍的三类算法：枚举算法、增广路算法和预流推进算法。三种算法基于不同的理论和性质，不过都获得了最大流的解法。在本文中我都附了各种算法的C++源程序，大家可以作为参考。

5 枚举算法

说道枚举人人都会，可是如何设计简单高效的枚举算法求最大流仍然不是一个十分容易的事情，在这里提供一个思路，将最大流问题转化为他的等效问题，最小割。枚举所有割来求最小割便可以在相当短的代码内完成任务，在数据规模不大的情况下相当方便，算法复杂度为$O(2^{|V|}|E|)$。

参考代码横空出世：

```
1  int maxflow()//calculate network max-flow by enumeration source=0 and sink=n-1
2  {
3      int flow=2147483647,tmp;
4      for(int i=0; i<(1<<(n-2)); i++)//enumerate the set S
5      {
6          s=(i<<1)|1; tmp=0;
7          for(int u=0; u<n; u++)
8              for(int v=0; v<n; v++)
9                  if(((s>>u)&1)==1&&((s>>v)&1)==0)
10                     tmp+=c[u][v];
11         if(flow>tmp)flow=tmp;
12     }
13     return flow;
14 }
```

在这个参考代码中s为枚举的S集合，用二进制的方式存储了每个点的状态，然后每枚举一个割的状态，就求一次容量，然后更新答案。

本算法最大的好处就是简单易行，只要学过编程都可以写出代码，最大流代码令人望而生畏的时代一去不复返啦！

6 增广路算法

6.1 增广路算法

枚举算法虽然正确，但在处理大规模数据时就显得力不从心，于是便需要更快的方法。

增广路算法为目前信息学竞赛中应用最广的算法，基本思路为不断在残量网络中寻找增广路进行增广，直到找不到增广路时算法结束，这时所求得的流为最大流。根据Theorem 2.9，正确性显然。

根据寻找增广路的策略不同，算法的实现和效率也有不同。所以在这里分别介绍了三种增广路算法，给与参考。

6.2 深度优先搜索

使用深度优先搜索策略实现最简单，算法复杂度上，若为整数网络，每进行一次搜索，至少可以使得答案增加1，每次搜索的复杂度为$O(|E|)$，故若最大流为c，总复杂度为$O(|E|c)$。

无辜的代码飘过：

```
1  int dfs(int x,int low) {
2      if(x==sink)return low;
3      if(vis[x])return 0;
4      vis[x]=true;
5      for(int i=0,flow;i<n;i++)
6          if(c[x][i]&&(flow=dfs(x,low<?c[x][i])))
7          {
8              c[x][i]-=flow; c[i][x]+=flow;
9              return flow;
10         }
11     return 0;
12 }
13
14 int maxflow() {
15     int ans=0,flow; memset(vis,0,sizeof(vis));
16     while(flow=dfs(s,2147483647)){memset(vis,0,sizeof(vis));ans+=flow;}
17     return ans;
18 }
```

这个参考代码以low记录可增广量，它为残量网络整个增广路中容量最小的边的容量。

用深度优先搜索的方式求最大流有一个非常有效的优化：当前弧(current arc)。即记录每个结点当前已经枚举到了哪一个位置，下一次搜索时从这

个位置开始继续枚举。

在代码实现时只需要另外记录一个当前弧信息cur[]即可，穿了马甲再次飘过：

```
int dfs(int x,int low) {
    if(x==sink)return low;
    if(vis[x])return 0;
    vis[x]=true;
    for(int ii=0,i=cur[x],flow;ii<n;ii++,i=(i+1==n?0:i+1))
        if(c[x][i]&&(flow=dfs(x,low<?c[x][i])))
        {
            cur[x]=i; c[x][i]-=flow; c[i][x]+=flow;
            return flow;
        }
    return 0;
}

int maxflow() {
    int ans=0,flow; memset(vis,0,sizeof(vis));
    while(flow=dfs(source,2147483647)){memset(vis,0,sizeof(vis));ans+=flow;}
    return ans;
}
```

据我了解这个优化在OIer中传播首先始于CQF在OIBH中的某个帖子¶，给出了使用当前弧优化通过NOI2006的PROFIT题的程序。经过一些实验得知本优化在某些特殊图中效率相当不错，但是在某些更为一般的图中时间表现不甚突出。

6.3 广度优先搜索

使用深度优先搜索算法求最大流已经得到了很不错的伪多项式算法，但是如果我们以更加有效的策略来进行增广，将会影响算法的渐进时间复杂度吗？答案是肯定的，经过证明，若每次从最短的增广路开始增广，则最多只用进行$O(|V||E|)$次增广，这比盲目进行增广所需要的增广次数大有改进。

那么如果求最短的增广路呢？最朴素的想法是使用广度优先搜索求出增广路，那么每次增广的复杂度为$O(|E|)$，算法总复杂度就为$O(|V||E|^2)$。此算法名为Edmond-Karp算法。

参考代码：

```
int maxflow() {
```

¶http://www.oibh.org/bbs/viewthread.php?tid=14263&page=6#pid151645

```
    int u,v,l,r,flow=0;
    do
    {
        memset(vis,0,sizeof(vis));vis[source]=1;
        memset(a,0,sizeof(a));low[s]=2147483647;
        q[0]=source;l=0;r=1;
        while(l<r)
        {
            u=q[l++];
            for(v=1; v<=n; v++) if(!vis[v] && c[u][v]>f[u][v])
            {
                vis[v]=1; pre[v]=u; q[r++]=v;
                low[v]=c[u][v]-f[u][v];  if(low[v]>low[u])low[v]=low[u];
            }
        }
        if(low[sink]>0)
        {
            u=t;
            do{f[pre[u]][u]+=low[sink]; f[u][pre[u]]-=low[sink]; u=pre[u];}while(u!=s
        );
            flow+=low[sink];
        }
    }
    while(low[sink]>0);
    return flow;
}
```

同样本代码low[]记录可增广量，同时pre[]记录结点的前驱，q[]作为广搜队列。每次使用广搜搜到一个解以后，进行增广。重复此过程直到找不到增广路。

好了，如果你想应付一般的网络最大流题目，看到这里就可以了。用Edmond-Karp可以解决大部分OI中遇到的最大流题目，就算遇到了时间上的问题，至多挂掉30%的分数，知足了吧。当然，也可以选择继续看下去，学会一些更快当然也更容易写错的算法，然后极有可能发生的结果是在考场上代码写错导致爆零，各位别怪我没提醒你了。

6.4 标号法

可以很容易地知道，最短增广路的算法复杂度下界为$O(|V|^2|E|)$，因为每次增广至少需要$O(|V|)$的时间，而一共需要增广$O(|V||E|)$次。那么这个下界如何得到，便要使用标号法来求最短增广路。这个算法被很多OIBH坛友称作SAP，即Shortest Argument Path，不过我更愿意叫它标号法，因为SAP这个名字实在是太宽泛了。

算法对每个结点使用了距离标号$d[i]$。对于残量网络G_f满足$d[sink] = 0$且对于所有$c_f(u,v) \geq 0$都有$d[u] \leq d[v]+1$。如果一条弧(u,v)满足$d[u] = d[v]+1$，我们则称这种弧为允许弧，由允许弧组成的从源到汇的路径则为允许路径。容易知道所有允许路径都是从源到汇的最短路。于是我们只需要不断修改距离标号找出增广路，即可完成算法。

在具体的算法中，可以证明，如果一条弧(u,v)不是允许弧，则它重新成为允许弧的必要条件为$d[u]$改变。所以对于每个点我们使用当前弧方法，记录每个点当前扫描到的位置，每次执行这个点时就从当前弧的位置开始扫描，直到扫描到最后也没找到增广路的话，则需要对$d[u]$进行重新标号为$\min_{(u,v)\in G_f} d[v]+1$。在利用了当前弧性质以后，可以证明整个算法复杂度为$O(|V|^2|E|)$。

这是首次展示完整代码，大家鼓掌：

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

const int maxn=1<<11; const int maxm=maxn*maxn; const int inf=2147483647;

int g[maxn],cap[maxm],to[maxm],np[maxm],cp;
int m,n,x,y,z,u,v,now,s,t,flow;
int d[maxn],a[maxn],p[maxn],cur[maxn],cnt[maxn];

int main() {
    freopen("ditch.in","r",stdin); freopen("ditch.out","w",stdout);
    scanf("%d%d",&m,&n); cp=2; s=1; t=n;
    while(m--)
    {
        scanf("%d%d%d",&x,&y,&z);
        cap[cp]=z;to[cp]=y;np[cp]=g[x];g[x]=cp++;
        cap[cp]=0;to[cp]=x;np[cp]=g[y];g[y]=cp++;
    }
    for(u=1;u<=n;u++)cur[u]=g[u]; flow=0; a[u=s]=inf; cnt[0]=n;
    while(d[s]<n)
    {
        for(now=cur[u];now;now=np[now])if(cap[now]&&d[v=to[now]]+1==d[u])break
         ; cur[u]=now;
        if(now)
        {
            p[v]=now; a[v]=cap[now]; if(a[v]>a[u])a[v]=a[u];
            if((u=v)==t)
            {
                do{cap[p[u]]-=a[t]; cap[p[u]^1]+=a[t]; u=to[p[u]^1];}while(u!=s);
                flow+=a[t]; memset(a,0,sizeof(a)); a[s]=inf;
            }
```

```
31          }
32          else
33          {
34              if(--cnt[d[u]]==0)break; d[u]=n; cur[u]=g[u];
35              for(now=g[u];now;now=np[now]) if(cap[now] && d[u]>d[to[now]]+1) d[u]=
            d[to[now]]+1;
36              cnt[d[u]]++;
37              if(u!=s)u=to[p[u]^1];
38          }
39      }
40      printf("%d\n",flow);
41      return 0;
42  }
```

这个程序在记录图的时候使用了链表，其中g[]记录了链表头，cap[]为容量，to[]为指向的结点，np[]为链表的下一个元素的地址。为了方便，将边以及与其对应的逆边分别记录在相邻的位置，要访问他的逆弧的时候只需要将自己的地址异或1即可。

另外在程序34行，有一句相当不起眼的break，实际上这是标号法相当重要的一个优化，叫做间隙优化(Gap Heuristic)。其内容是，如果当前没有任何一个结点的距离标号为k，且存在距离标号大于k的结点，则整个网络流算法结束。这是一个相当有效的优化，他的原理也相当简单，即是整个图已经被分开，当然不用继续费心寻找增广路了。

标号法使用了间隙优化后在实际应用中总有很好的表现，而且实现比较简单，所以深受很多OIer青睐。

6.5 层次图中的网络流算法

关于最短增广路的各种实现，另有一类算法另辟蹊径，重新开创了一类不同于每次找到一条最短增广路的思路，通过构建网络层次图，每次可以检查出很多条增广路，这就是所谓的多路增广算法，也就是在层次图中找增广路的算法。

层次图的网络最大流算法在wxs的论文中有比较详尽的阐述‖，在论文中一共阐述了三种利用层次图进行的网络流算法：MPLA，Dinic和MPM。

所谓层次图，即是在残量网络中所有最短增广路所组成的网络，所有的层次图中的算法，都是先建立层次图，然后在层次图上进行增广，无法增广以后重新建立层次图，直到算法结束。一般层次图的建立使用从汇开始在残量网络中的广度优先遍历，给每个点进行距离标号，若边(u,v)有容量且$d[u] = d[v] + 1$，则称边(u,v)为允许弧，允许弧组成的图便为层次图。在

‖《浅谈基于分层思想的网络流算法》，王欣上，2007年国家集训队论文

实际实现的时候，层次图也不用重新存储，只用在遍历的时候检查边是否满足距离标号的条件即可。

MPLA算法使用广度优先搜索在层次图中寻找增广路并增广，最终复杂度为$O(|V||E|^2)$。为最朴素的层次图中的网络最大流算法。由于它和直接广搜的差距不大，所以在这里就不给出代码了。

Dinic算法则使用了深度优先搜索在层次图中寻找增广路，并且记录了每个结点的当前弧信息。由于每个结点的距离标号不改变的情况下，任何弧不可能从非允许弧重新变为允许弧，所以在结点访问完自己所有相邻的边以后，不用从第一条边开始继续访问，可以直接退出，所以算法复杂度有了本质的改变。整个算法复杂度为$O(|V|^2|E|)$，并且在实际应用中通常有更好的效果。

下面是一个使用邻接矩阵实现的Dinic算法，在实际应用中应该使用邻接表或者前向星以获得需要的时间效果。

```
1  bool bfs() {
2      int l=0,r=0,u,v; q[r++]=t; d[t]=0; memset(b,0,sizeof(b)); b[t]=1;
3      while(l<r)
4      {
5          u=q[l++];
6          for(v=1;v<=n;v++) if(f[v][u]<c[v][u]&&!b[v]) {d[v]=d[u]+1; b[v]=1; q[r++]=v
           ;}
7          if(b[s])return b[s];
8      }
9      return false;
10 }
11
12 int maxflow() {
13     int u,v,flow=0;
14     while(bfs())
15     {
16         a[u=s]=inf; memset(cur,0,sizeof(cur));
17         for (;;)
18         {
19             int ok=0;
20             for(v = cur[u]; v <= n; v++) if(f[u][v]<c[u][v] && d[u]==d[v]+1) { ok
           =1; break; }
21             if(ok){
22                 cur[u]=v+1; p[v]=u; a[v]=c[u][v]-f[u][v]; if(a[v]>a[u]) a[v]=a[u];
23                 if((u=v) == t){
24                     do{cur[p[u]]=u; f[p[u]][u]+=a[t]; f[u][p[u]]-=a[t]; u=p[u];}while
           (u!=s);
25                     flow+=a[t]; a[s]=inf;
26                 }
```

```
27              }else if(u!=s){cur[u]=v;u=p[u];}
28              else break;
29          }
30      }
31      return flow;
32  }
```

在程序中使用了非递归的算法来实现深度优先遍历。可以给想要学习非递归进行深度优先遍历的同学一些参考。

MPM算法则没有使用通常的方法利用遍历图寻找增广路，而是另辟蹊径，在层次图中定义了每个非源非汇点的通过量，即是每个点的入边容量和与每个点的出边容量和的较小值。每次选择一个通过量最小的结点作为关键结点进行增广，增广的过程则为从关键结点开始利用一次遍历将流量从源流到关键点，然后在利用一次遍历将流量从关键点流到汇，在这同时更新访问到的点的通过量。由于关键点的通过量是最小的，所以不会存在不能增广的可能性。将关键点的流量置满以后，将这个点从图中删去，并且更新相邻点的通过量。然后重复此过程直到所有边都被删去。然后再重新建立层次图进行下一次过程。这样做整个算法复杂度可以证明为$O(|V|^3)$。

MPM程序的实现较为繁琐，不过具有一定的理论价值，具体代码如下**：

7 预流推进算法

7.1 预流推进算法

除了增广路算法以外，另外一个流传甚广的，并且常年以来一直被我认为牛不可及的网络最大流算法便是预流推进算法(Preflow Push)。这个算法被我束之高阁主要归功于《算法艺术》††，里面描述的预流推进算法让我只能模糊地看到算法的轮廓，却没法将它用简单的程序语言表述出来。

当然，事实上预流推进算法并不是我想象中的那么困难。相对于增广路算法来说，预流推进算法更加“局部”。如果说增广路算法是扮演一个将军调兵遣将指挥整个战斗，那么预流推进算法更像是一个一个士兵在完成自己的使命。

预流推进的数据组织和操作都是以结点为单位进行的。对于每个结点，可以想象为一个蓄水池，蓄水池中的水，称为盈余，有盈余的非源非汇结

**To be announced

††《算法艺术与信息学竞赛》，刘汝佳，黄亮，相当厚道地只卖45元

点称为活跃结点(active node)，每个结点还有一个高度(height)，其实也就是前面所介绍的距离函数，引入高度是为了限制水只能从高的结点流向低的结点而不能倒流，所以可以定义所有容量不为0且满足$d[u] = d[v] + 1$的弧(u,v)为允许弧，所有操作都只能在允许弧上进行。在算法开始时将所有从源出发的弧全部满载，并且修改对应的结点的盈余，然后将源的高度设为$|V|$，之后开始主算法。

主算法一共有两种基本操作，推进(Push)和重标号(Relabel)。

对于一个活跃结点，检查所有以这个结点出发有没有允许弧，如果有，则可以进行推进操作，将这个结点的盈余推到与他相邻的其他结点，知道结点没有盈余或者所有允许弧都满载了为止。

如果一个活跃结点出发没有允许弧，则需要对他进行重标号操作，即将这个结点的高度标号重新标为$d[u] = \min_{c(u,v)>0} d[v] + 1$。

预流推进算法即为不停地对活跃结点进行上述两种操作，直到没有活跃结点存在。整个过程相当于模拟水流从源开始慢慢流向汇然后再将多余的水流流回源的过程，开始源的高度设为$|V|$可以保证所有盈余不会一开始就流回源然后假惺惺地告诉你算法已经结束了，而你还会惊叹于他怎么做到如此的迅速。可以注意到在预流推进算法的进行过程中没有满足流量平衡条件，所以在这个时候的流称作“预流”(Preflow)，而最短增广路算法进行过程中时时都满足了流量平衡条件的。算法结束时汇中的盈余即为答案。

预流推进算法的操作执行的顺序直接影响了算法的复杂度。在这里有一个经验性的优化是利用队列保存所有活跃结点，每次从队首取出一个活跃结点，然后对其进行推进操作，推进结束后若还有盈余，则对这个结点进行重标号后将结点放回队末[‡‡]。

队列优化的参考程序如下：

```
1  int maxflow(int s, int t){
2    int u, v, flow=0, l=0, r=0;
3     memset(d, 0, sizeof(d)); memset(e, 0, sizeof(e)); d[s]=n;
4     for(u=1;u<=n;u++){e[u]=f[s][u]=c[s][u]; f[u][s]=-f[s][u];  if(e[u]>0&&u!=t){q[r
            ++]=u; cnt[d[u]]++;} }
5     while(l != r){
6       int ok = 0;  u = q[l++]; l%=n;
7       for(v = 1; v <= n; v++)
8         if(c[u][v]>f[u][v]  && d[u]==d[v]+1 && e[u]>0){
9           ok = 1; int a = c[u][v]-f[u][v];  if(a>e[u]) a = e[u];
10          f[u][v]+=a; f[v][u]-=a; e[u]-=a; e[v]+=a;
11          if(e[v] == a && v!=s && v!=t){ q[r++] = v; r%=n; }
```

[‡‡]不过据说更加正规的做法是不断推进与重标号这个点直到这个结点不活跃为止，我也不是很清楚，希望大家提出自己的意见

```
12          }
13        if(e[u]>0 && u!=s && u!=t){
14          int k=d[u]; cnt[k]--; d[u]=2*n+1;
15          for(v = 1; v <= n; v++) if(c[u][v]>f[u][v]  && d[u]>d[v]+1) d[u]=d[v]+1;
16          q[r++] = u; r%=n; cnt[d[u]]++;
17          if(cnt[k]==0) for(v = 1 ; v <= n; v++) if(d[v]>k && d[v]<n){ cnt[d[v]]--; d[
              v]=n; cnt[n]++; }
18        }
19      }
20      for(u = 1; u <= n; u++) flow+=f[s][u];
21      return flow;
22    }
```

在这个这个预流推进程序中，也使用了同标号法一样的间隙优化，不同于标号法的是，在间隙优化条件满足的时候将所有高度大于k小于等于$|V|$的结点的高度置于$|V|+1$，然后继续执行算法直到正常结束。

可以证明，预流推进算法的算法复杂度为$O(|V|^2|E|)$。而使用了队列优化的预流推进的算法复杂度为$O(|V|^3)$。

7.2 最高标号预流推进算法

Part IV
网络最大流的应用

8 直接应用的例子

9 各种变形

10 最小割问题

11 更多的例子