Lecture 10. 网络流及其应用

吴步娇

2016 年 1 月 1 日

1 上节回顾

1.1 求解最小割问题的算法的发展史

年	提出者	时间复杂度
1956	Ford and Fulkerson	$O(mC)$ and $O(m^2 \log C)$
1972	Edmonds and Karp	$O(m^2 n)$
1970	Dinitz	$O(n^2 m)$
1974	Karzanov	$O(n^3)$
1983	Sleator and Tarjan	$O(nm \log n)$
1988	Goldberg and Tarjan	$O(n^2 m \log(\frac{n^2}{m}))$
2012	Orlin	$O(nm)$

1.2 最大流问题

问题描述:有向图$G =< V, E >$，边e上有大小为C_e的容量限制，G中有两个特殊的点：起点s 和目的地t。现给每条边$e = (u, v)$指定流值$f(u, v)$，问如何指定可使得总的流值$\sum_{u,(s,u)\in E} f(s, u)$ 最大。例如下图（图1.1）：

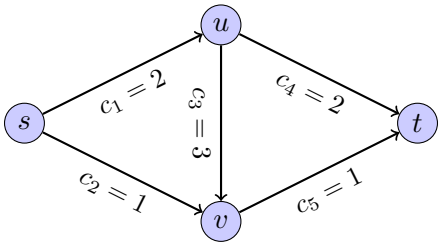


Figure 1: 图1.1 求解$s \to t$可运输的最多货物

目标：从起点s运输尽可能多的货物到目的地t。

2 Ford-Fulkerson算法[1956]

Figure 2: Lester Randolph Ford Jr. and Delbert Ray Fulkerson

2.1 动态规划求解

遗憾的是，该问题并不存在动态规划的多项式时间算法。但在第8节中也可以看出，该问题是P 类的，因为可以用线性规划求解。

2.2 Improvement策略

(1)Improvement策略大致思想

先确定一个初始可行解，然后再改进，直到不能改进。该问题用Improve- ment 策略实现的伪代码如下：

IMPROVEMENT(f)

```
1: x = x0; //starting from an initial solution;
2: while TRUE do
3:    x =IMPROVE(x); //move one step towards optimum;
4:    if STOPPING(x, f) then
5:       break;
6:    end if
7: end while
8: return x;
```

(2)算法需要考虑的3个关键因素：

a.如何构造初始可行解? 一个很直接的方法是初始化为0流，每条边均有$f(e) = 0$

b.如何对接进行改进

c.何时需要停止?

(3)一个失败的尝试

对于上节课所讲的一个笨拙的改进算法，随机选择可行流并在图上进行改进所得到的流，可能最终并不能终止于最优解。例如如下左图（图2.2）的可行流，该可行流无法通过增加流值再得到改进，但该可行流并不是最大流，因为右图（图2.3）是一个流值为2 的可行流。

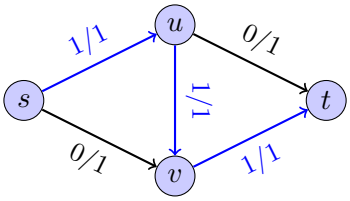


图2.2 $V(f)=1$

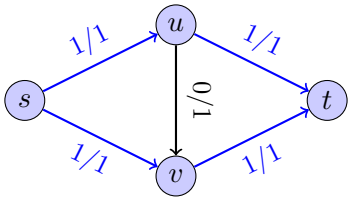


图2.3 $V(f)=2$

(4)Ford-Fulkerson算法引入反向边，让运出去的货物还可以有机会退回。将加入了退货边的图称为剩余图，构造剩余图的大致思路为，若$u \to v$运了f吨货物，且容量限制为$C(u,v)$，则剩余图中u到v 的正向弧$u \to v$权值为$C(u,v)-f$，表示最多还可再运$C(u,v)-f$ 吨货，并构造权值为f的反向弧，表示最多可退大小为f的货物。例如，下面两个图分别代表网络的流图（图2.4）及剩余图（图2.5）。

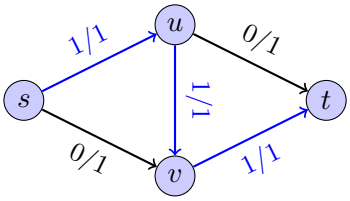


图2.4 Flow f

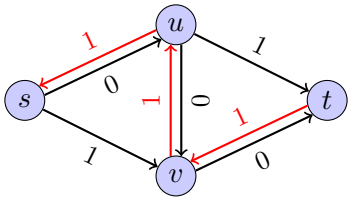
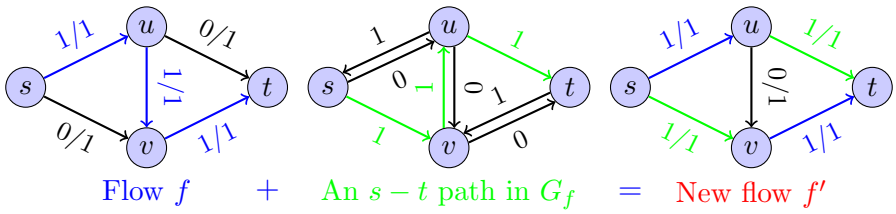


图2.5 Backward edges

Ford-Fulkerson算法的实现过程可以简单地用如下所示的图来刻画：



简单来说，就是原始流+ 剩余图中的可行路= 原始流的一个改进。

令p表示剩余图G_f中的一条简单路径，称为增广路径。并定义$bottleneck(p,f)$为路径p 上的最小容量边。则Ford-Fulkerson算法可以描述为：

FORD-FULKERSON algorithm

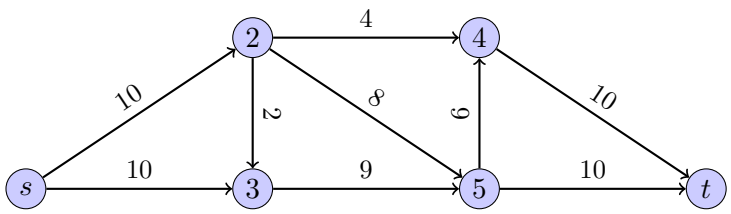
1: Initialize $f(e)=0$ for all e.
2: **while** there is an $s-t$ path in residual graph G_f **do**
3: **arbitrarily** choose an $s-t$ path p in G_f;
4: f =AUGMENT(p,f);
5: **end while**
6: **return** f;

与上节课所讲的随机行走的算法的唯一区别是剩余图有起点到终点的路而不是原网络图有起点到终点的路径。但这一小小的修改却可以使算法最终终止于最优解（网络的最大流）。

下面是一个Ford-Fulkerson算法的实现。

Ford-Fulkerson算法实例

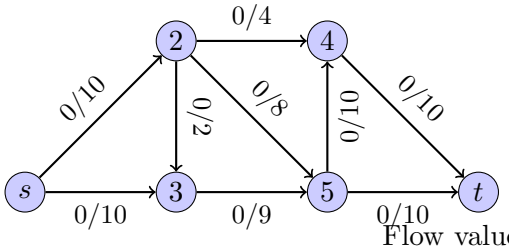
图G



Flow $value = 0$

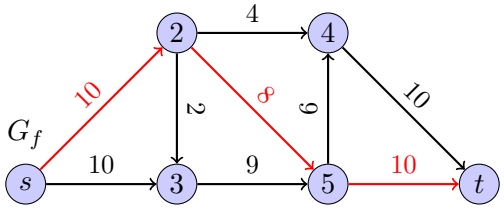
step1

G



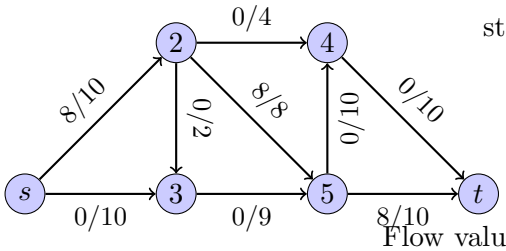
Flow value = 0

G_f



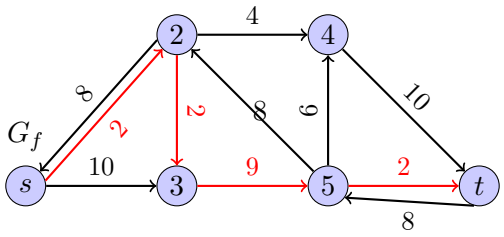
step2

G



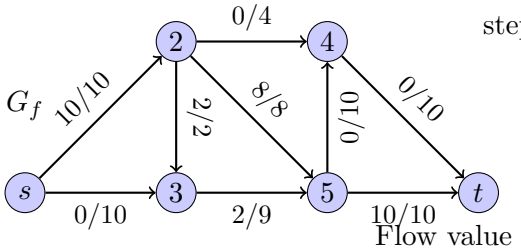
Flow value = 8

G_f

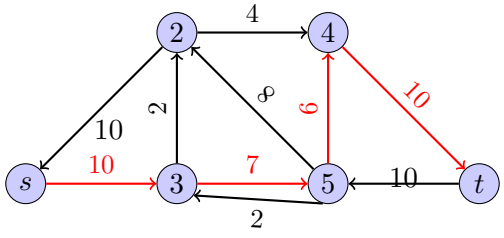


step3

G

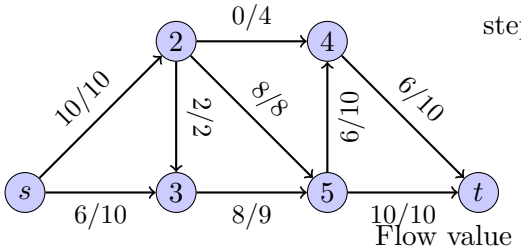


Flow value = 10

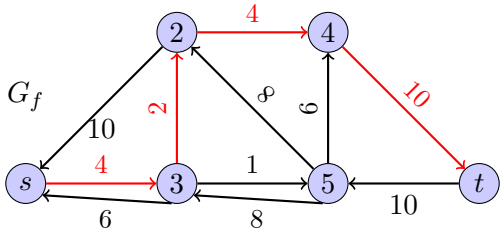


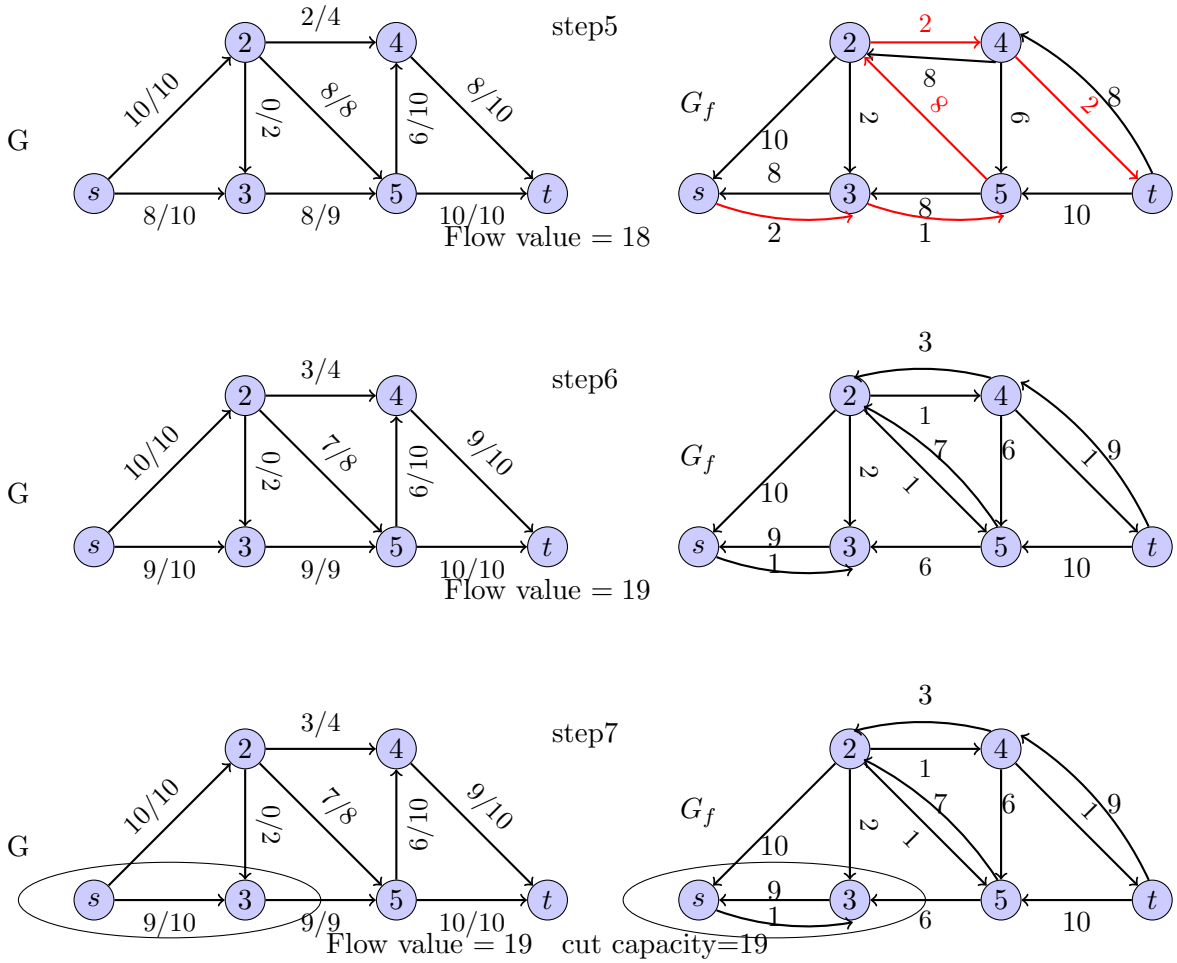
step4

G



Flow value = 16





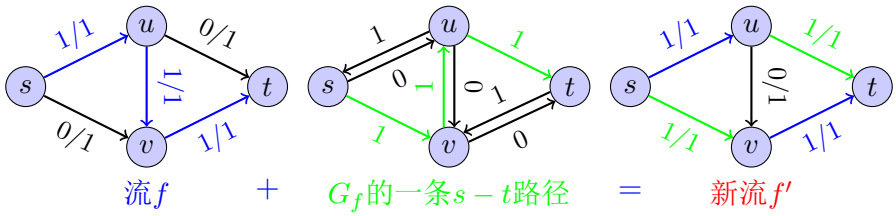
上图中step7中的流即为最大流，G_f中阴影部分无可到达外面的有向边，称阴影部分与外面所形成的割为最小割。

2.3 正确性证明及时间复杂性分析

2.3.1 增广操作将产生一个新的流

定理1. 操作$f' = AUGMENT(p, f)$将生成G的一个新流f'。

如下图所示：



证明. • 检查f'是否满足容量限制(capacity constraints)$0 \le f'(e) \le C(e)$，对于路径p上的边$e = (u, v)$, 可以分为以下两种类型

(a) (u,v)是前向边($(u, v) \in E$)：

$0 \le f(e) \le f'(e) = f(e) + bottleneck(p, f) \le f(e) + (C(e) - f(e)) \le C(e)$

(b) (u,v)是反向边($(v,u) \in E$):
$C(e) \geq f(e) \geq f'(e) = f(e) - bottleneck(p,f) \geq f(e) - f(e) = 0$

- 检查储存限制(conservation constraints)(流入=流出)
  对f'中的每个中间节点v(除去起点s，终点t)，流入v的流的总和等于流出v 的总和。

□

2.3.2 单调递增

定理2. （单调递增）$V(f') > V(f)$

证明. $V(f') = V(f) + bottleneck(p,f) > V(f)$ □

2.3.3 一个平凡的上界

定理3. $C = \sum_{e\ out\ of\ s} C(e)$是$V(f)$的一个上界。

证明. 显然，$V(f) \leq f^{out}(s) = C$。 □

2.3.4 增广次数

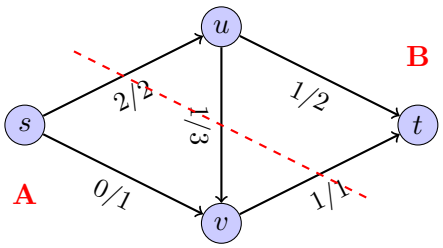
定理4. 假定所有的边的容量值都是整数，则执行*Ford-Fulkerson*算法时，每次迭代的流值及容量值都是整数。因此，$bottleneck(p,f) \geq 1$，且循环至多进行C次。

证明. 在假定容量值为整数的前提下，循环的每步均有$bottleneck(p,k) \geq 1$，因此，$V(f') \geq V(f) + 1$，则循环至多进行C 次。
时间复杂性为$O(mC)$.(C次循环，每次循环需要使用DFS或BFS寻找一条$s-t$的路径，花费时间为$O(m+n)$) □

2.3.5 一个更紧确的上界

定理5 (紧确的上界). 给定流f，对于任意$s-t$割$cut(A,B)$，有$V(f) \leq C(A,B)$。

下图给出了一个例子：



$$V(f) = 2 \leq C(A,B) = 3$$

证明.

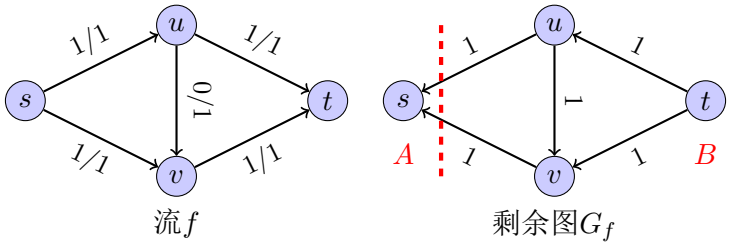
$$\begin{aligned}
V(f) &= f^{out}(A) - f^{in}(A) && \text{(由流值引理)} \\
&\leq f^{out}(A) && \text{(由} f^{in}(A) \geq 0\text{)} \\
&= \sum_{e \in A \to B} f(e) \\
&\leq \sum_{e \in A \to B} C(e) && \text{(由} f(e) \leq C(e)\text{)} \\
&= C(A,B)
\end{aligned}$$

□

2.3.6 正确性证明

定理6. *Ford-Fulkerson算法终止于最大流f，也即为最小割$cut(A,B)$。*

如图所示：



证明. • 当剩余图G_f中无$s-t$的路径时，Ford-Fulkerson算法终止

- 令A为G_f中从s可达的顶点集，并令$B=V-A$，则(A,B)形成了一个$s-t$割$(A\neq\phi, B\neq\phi)$。
- 位于割$cut(A,B)$之间的边可分为以下两种类型:
  1. $u\in A, v\in B$：则有$f(e)=C(e)$。否则，由$(u,v)\in G_f$ 可得$v\in A$，
  2. $u\in B, v\in A$：则有$f(e)=0$。否则，由$(v,u)\in G_f$ 可得$u\in A$
- 于是有最大流等于最小割

$$\begin{aligned} V(f) &= f^{out}(A)-f^{in}(A) \\ &= f^{out}(A) \qquad (\text{由} f^{in}(A)=0) \\ &= \sum_{e\in A\to B} f(e) \\ &= \sum_{e\in A\to B} C(e) \qquad (\text{由} f(e)=C(e)) \\ &= C(A,B) \end{aligned}$$

□

2.4 FORD-FULKERSON算法的缺点

2.4.1 整数约束

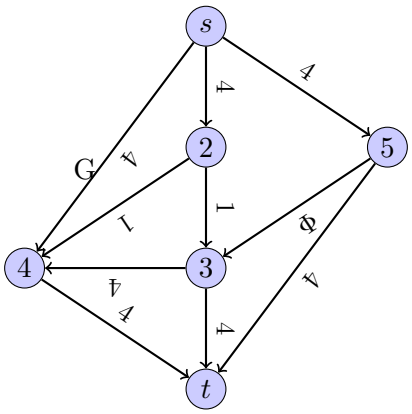
容量的整数限制是该算法不可或缺的条件，因为这可使得边的瓶颈(bottleneck)增量每次至少增加1。

但当容量限制为无理数(有理数总可以通过乘以某个数转化为整数)，该算法有可能会处于无限循环中，而无法终止于最大流。

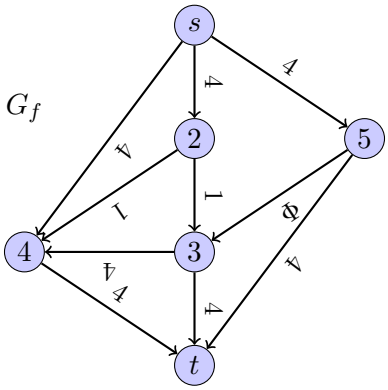
下面的例子显示了当容量约束条件存在无理数时，会出现无限循环而无法停止的结果：

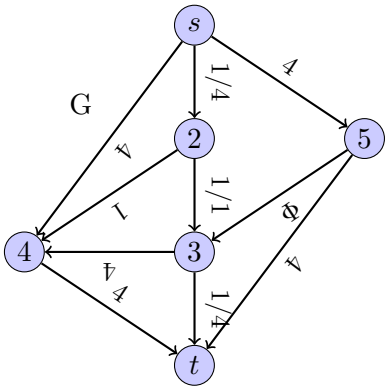
其中，$1-\Phi=\Phi^2$

最大流实例



Step 1

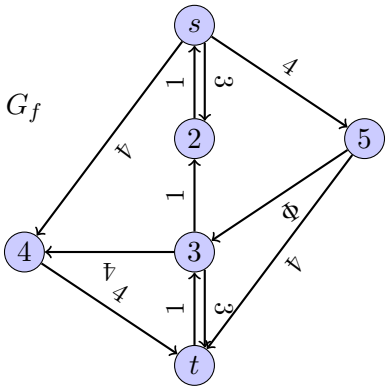


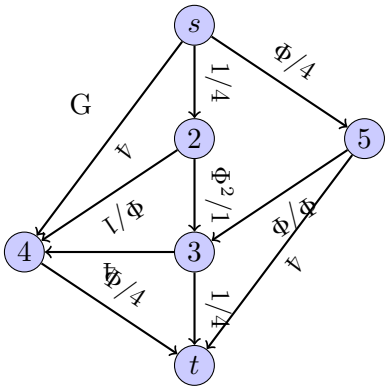


流值= 1

1

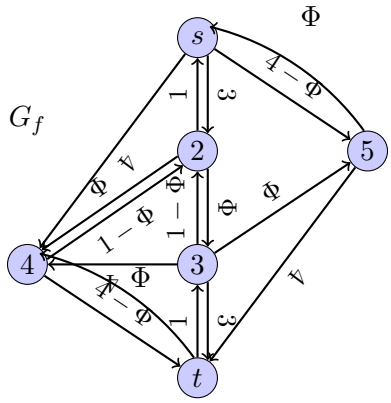
Step 2

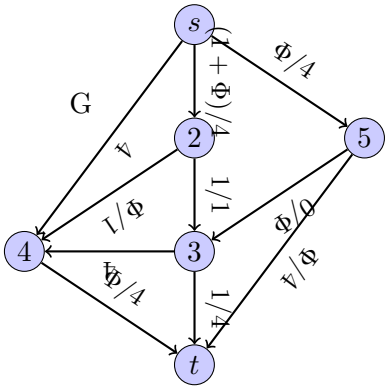




流值= $1 + \Phi$

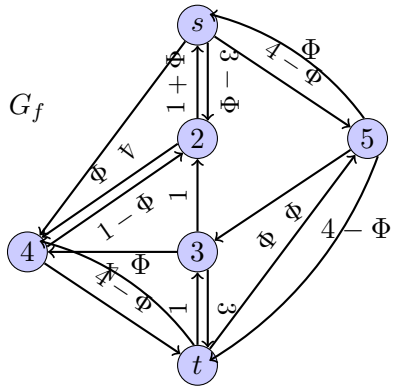
Step 3

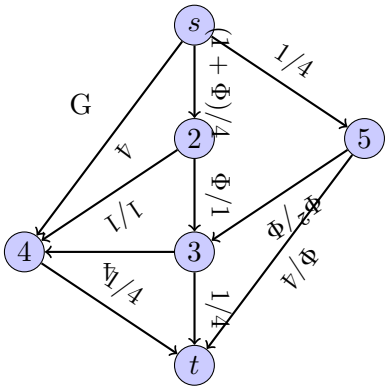




流值$= 1 + \Phi + \Phi$

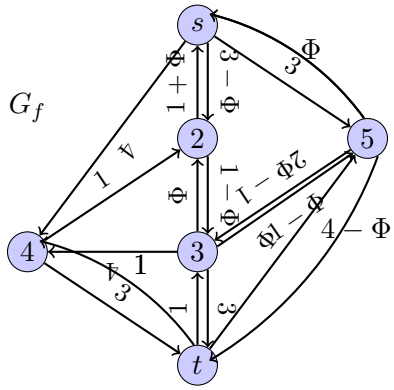
Step 4

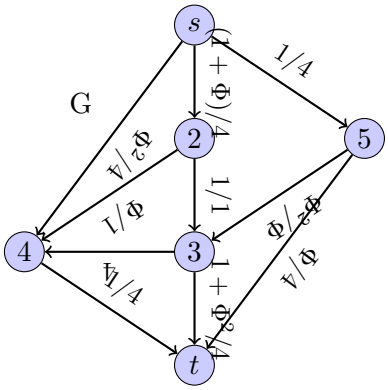




流値$= 1 + \Phi + \Phi + \Phi^2$

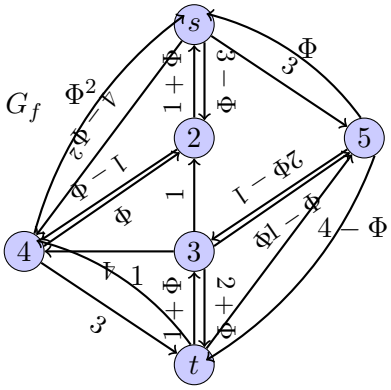
Step 5

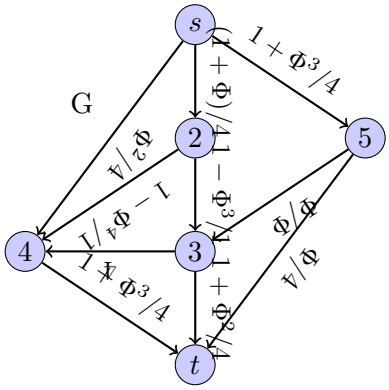




流値$= 1 + \Phi + \Phi + \Phi^2 + \Phi^2$

Step 6



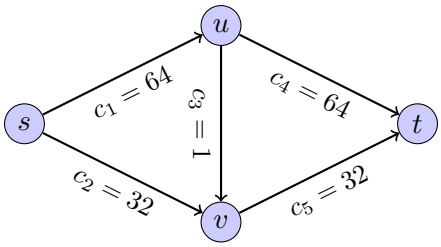


流值$= 1 + \Phi + +\Phi + \Phi^2 + \Phi^2 + \Phi^3$

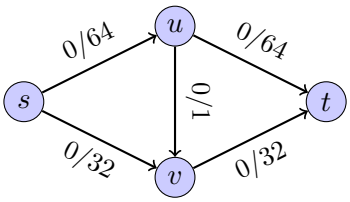
1: **if** FLOW$(2, 4) == 1$ **then**
2: Choose Step 5 as the path.;
3: **else**
4: **if** FLOW $(5, 3) == \Phi$ **then**
5: Choose Step 3 as the path.;
6: **end if**
7: **if** FLOW $(2, 3) == 1$ **then**
8: Choose Step 4(6) as the path.;
9: **end if**
10: **end if**

2.4.2 一个时间复杂度很高的实例

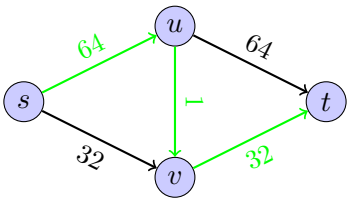
由于FORD-FULKERSON算法并未指定如何选取路径，算法运行时间依赖于每次所选的路径，比如下面的例子，当选取的增广路径不太好时，运行时间就会大大的增大。



FORD-FULKERSON算法: Step 1

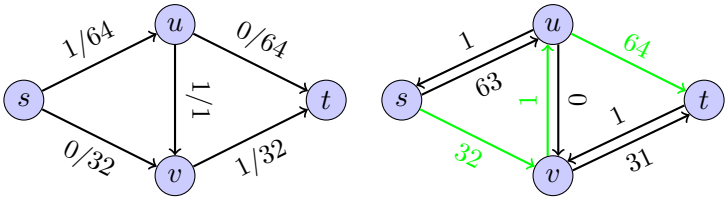


流$f : V(f) = 0$



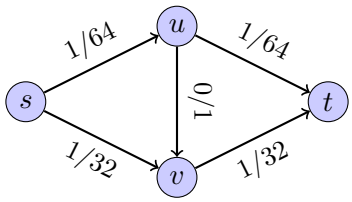
G_f的一个$s-t$路径

Ford-Fulkerson算法: Step 2



流$f: V(f) = 1$　　G_f的一个$s-t$路径

Ford-Fulkerson 算法：Step 3



流$f: V(f) = 2$

经过两次迭代后，该问题类似于原问题，只需修改原问题的容量限制，使得每条边的容量限制减1即可。

因此，利用FORD-FULKERSON算法，经过$32+32+1$次迭代就可以得到最大流。但实际上该问题若选取合适的扩展路，只需要$2+1$次迭代即可。

2.5 FORD-FULKERSON算法的改进思想

由于FORD-FULKERSON算法并未确定如何选择增广路径，故可能会使得每次的瓶颈容量值选取的很小。

改进思路

（1）尽量走大路

每次选择具有最大瓶颈容量的增广路径(实际应用较少)。

Scaling技术：一种可以寻找增广路径的有效的方法，每次可以得到大的改进。

（2）走最短路径

Edmonds-Karp: 在BFS树中选择最短s-t路径。

Dinitz算法：在分层网络中寻找一条路径，并执行

3 改进策略一：Scaling技术

问题：如何选取一个大的增广路径？$tottleneck(p,f)$越大，迭代次数越少。

1) 可以通过二进制搜索，或者以$O(m+nlogn)$ 时间对Dijkstra算法稍加修改得到最大的$bottleneck(p,f)$。但从某种程度来说（速度很慢），这种策略是不可行的。

b) 一种朴素的思想：放松条件，将“最大”放松为“足够大”。

借助该思想，可以建立$bottleneck(p,f)$的一个下界△：删除小边。即：删除容量小于△的边，得到的图记为$G_f(\triangle)$。

3.1 Scaling Ford-Fulkerson 算法

1: Initialize $f(e) = 0$ for all e.
2: Let $\Delta = C$;
3: **while** $\Delta \geq 1$ **do**
4: **while** there is an $s - t$ path in $G_f(\Delta)$ **do**
5: choose an $s - t$ path p;
6: f =AUGMENT(p, f);
7: **end while**
8: $\Delta = \Delta/2$;
9: **end while**
10: **return** f;

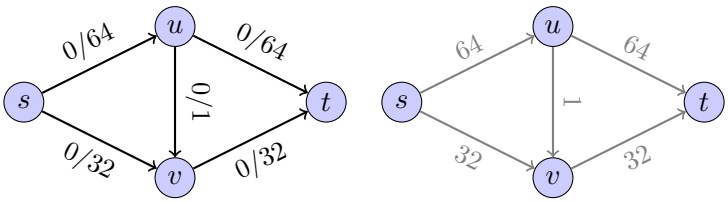
除了step2,3,4,8外其余与Ford-Fulkerson算法步骤相同。

由于在G_f中删去了小于△的边，因此每次增流的规模都很大。

△最终必须减为1，这样才能保证剩余图中无反向边(不漏掉容量较小的边)。

用该改进算法求上述实例：

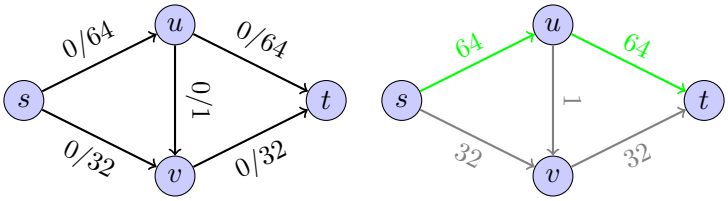
实例: Step 1



流$f : V(f) = 0$　　G_f中无$s - t$ 路径

构造0流，△ = 96，由于G_f(△)图中无$s - t$的路径，令△ = △/2 = 48。

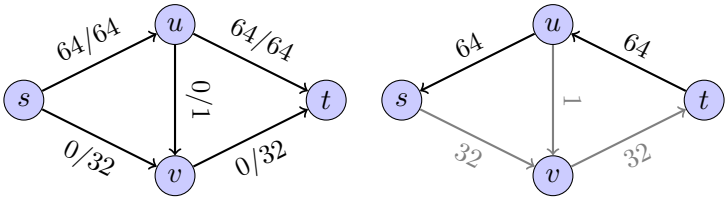
实例: Step 2



流$f : V(f) = 0$　　G_f中存在$s - t$路径

0流，△ = 48，存在$s - t$的路径(s-u-t)，执行增广操作。

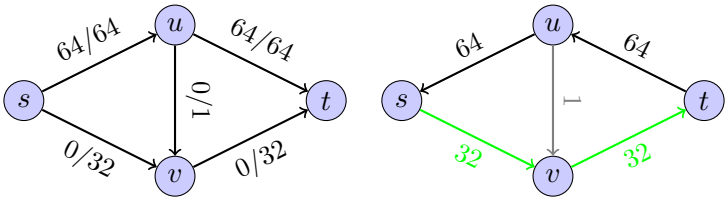
实例: Step 3



流$f : V(f) = 64$　　G_f中不存在$s - t$路径

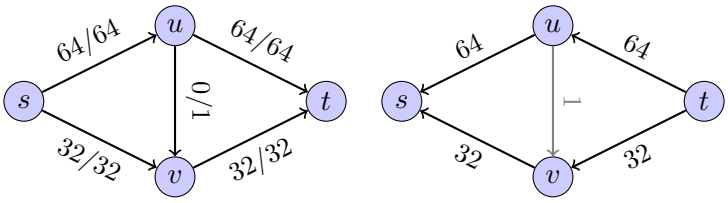
流：$V(f) = 64$，△ = 48，不存在$s - t$路径，令△ = △/2 = 24。

实例: Step 4



流：64，$\triangle = 224$，存在$s-t$路径：$s-v-t$，执行增广操作。

实例: Step 5



流：96，已得到最大流。$\triangle = 24$，不存在$s-t$路径。

3.2 Scaling技术的时间复杂度

ScalingFF的时间复杂度为$O(m^2 \log C)$，因为算法依赖于C的长度，因此该算法并不是多项式算法。

定理7 (外循环次数). *while迭代次数至多进行*$1 + log_2 C$。

定理8 (内循环次数). *Scaling阶段，增广操作的次数至多进行*$2m$ *次。*

证明. **1)** 令f表示$\triangle$ Scaling 阶段结束时的流，f^*表示最大流。则有$V(f) \geq V(f^*) - m\triangle$。

2) 在下个$\frac{\triangle}{2}$ Scaling阶段，每次增广，$V(f)$至少增加$\frac{\triangle}{2}$。

则在$\frac{\triangle}{2}$ Scaling至多进行$2m$次增广。

□

下面来证明$V(f) \geq V(f^*) - m\triangle$。

证明. 令A表示剩余图中从s可到达的顶点集，$B = V - A$，则(A, B)构成了最小割。

考察$e = (u, v) \in E$：

1) $u \in A, v \in B$: 则有$f(e) \geq C(e) - \Delta$。否则，A应该包含v，若G_f 中正向边的流值$\geq \triangle$，$(u, v) \in G_f(\triangle)$

2) $v \in A, u \in B$: 则有$f(e) \leq \Delta$。否则，A应该包含v，若G_f中反向边的流值大于$\triangle$ ，$(u, v) \in G_f(\triangle)$。

因此：

$$\begin{aligned} V(f) &= \sum_{e \in A \to B} f(e) - \sum_{e \in B \to A} f(e) \\ &\geq \sum_{e \in A \to B} (C(e) - \Delta) - \sum_{e \in B \to A} \Delta \\ &\geq \sum_{e \in A \to B} C(e) - m\Delta \\ &= C(A, B) - m\Delta \\ &\geq V(f^*) - m\Delta \end{aligned}$$

□

Scaling技巧对以后的发展很有用，后面在近似算法的背包问题也会提到。

4 改进策略二：EDMONDS-KARP $O(m^2n)$算法

4.1 算法的创造者


Figure 3: Jack Edmonds, and Richard Karp

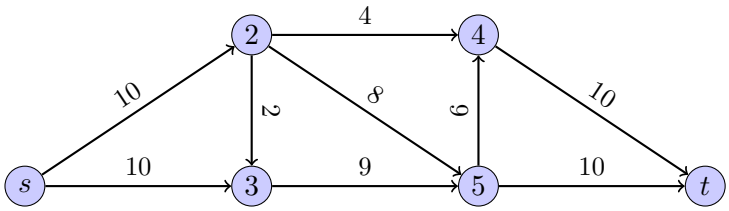
4.2 EDMONDS-KARP算法

1: Initialize $f(e) = 0$ for all e.
2: **while** there is a $s - t$ path in G_f **do**
3: choose **the shortest** $s - t$ path p in G_f using BFS;
4: f =AUGMENT(p, f);
5: **end while**
6: **return** f;

下面是该算法的一个运行实例：

Edmonds-Karp算法实例

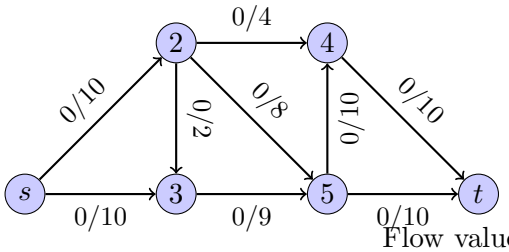
图G



Flow $value = 0$

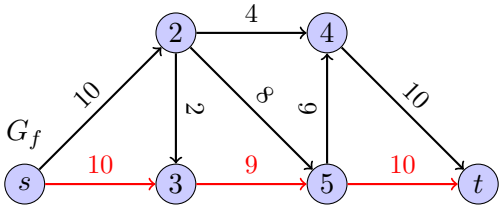
step1

G



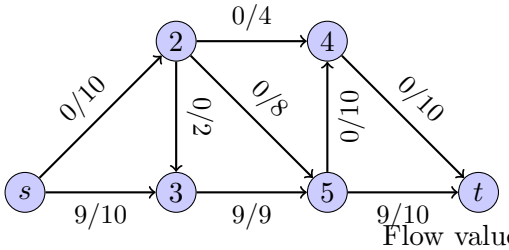
Flow value = 0

G_f



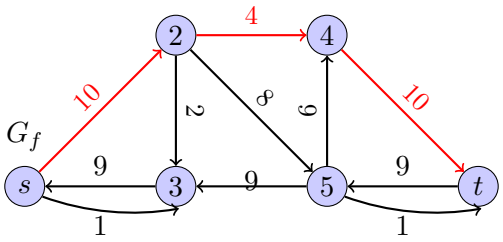
step2

G



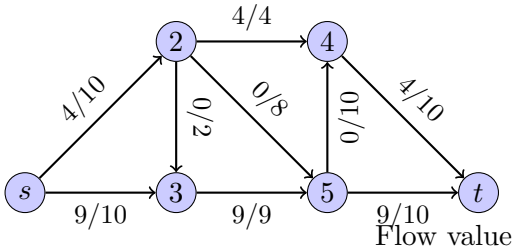
Flow value = 9

G_f



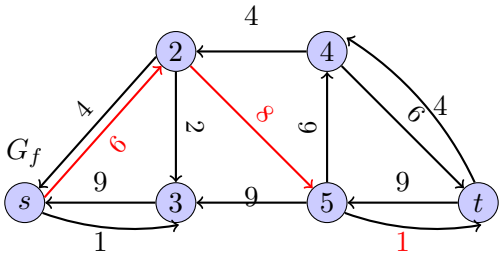
step3

G



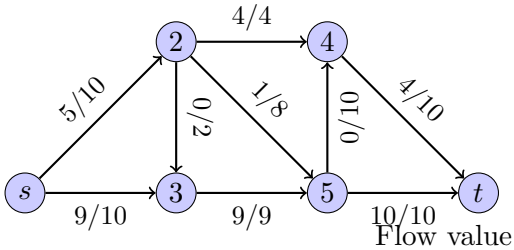
Flow value = 13

G_f



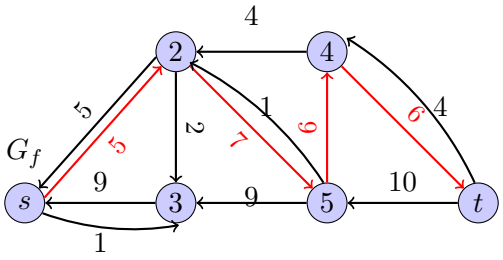
step4

G

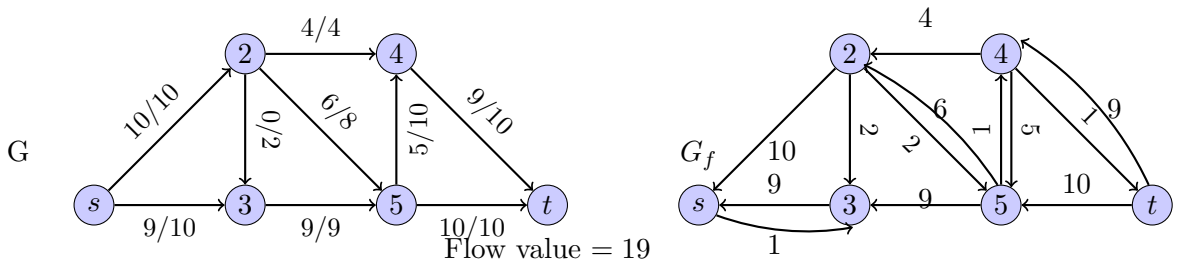


Flow value = 14

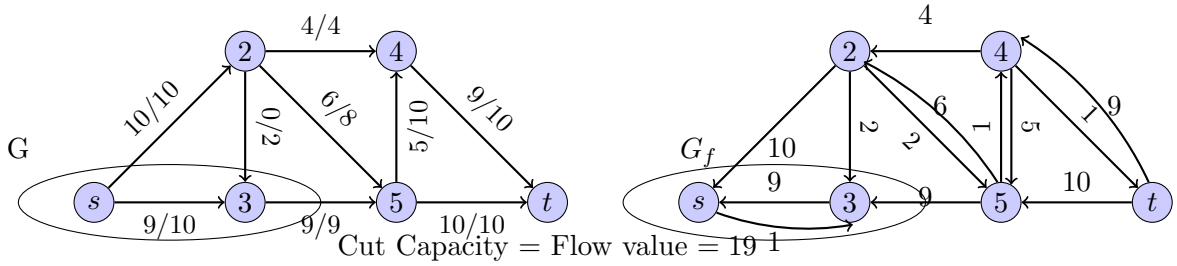
G_f



step5



step6



4.3 时间复杂性分析

引理1. Edmonds-Karp算法中，一条边$e=(u,v)$作为*bottleneck*边的次数至多为$\frac{n}{2}$ 次。

证明. • 将剩余图G_f的节点分层为$L_0, L_1, ...$，其中$L_0 = s$，L_i表示从s 出发，最短路径为i的所有点的集合，用$L(u)$表示节点u的层数。

- 假设一条边$e=(u,v)$在G_f 中连续两次作为bottleneck，依次记为while循环中的第k步，第k'''步。
- 在第k步:$L(v) = L(u) + 1$，经过这一步流的增广，bottleneck 边$e = (u,v)$的反向边将出现在G_f 中（只有反向边$e' = (v,u)$没有正向边，因为该边是瓶颈(bottleneck) 边）。
- 在第k'''中，$e = (u,v)$又成为bottleneck边。这意味着中间某一次循环，记为$k''(k < k'' < k''')$中，$e' = (v,u)$出现在最短路径上（此次增广后，$e = (u,v)$ 将出现在G_f 中）。因此有$L''(u) = L''(v) + 1$，即$L''(u) = L''(v) + 1 > L(v) + 1 = L(u) + 2$。
- 又由于对任意节点，层数至多为n，因此有引理成立。

□

定理9. Edmonds-Karp算法运行时间为$O(m^2n)$。

证明. • 由引理1知，每条边$e = (u,v)$ 至多会被作为bottleneck边$\frac{n}{2}$次。因此，while 循环至多进行$\frac{n*m}{2}$次(总共有m条边)。

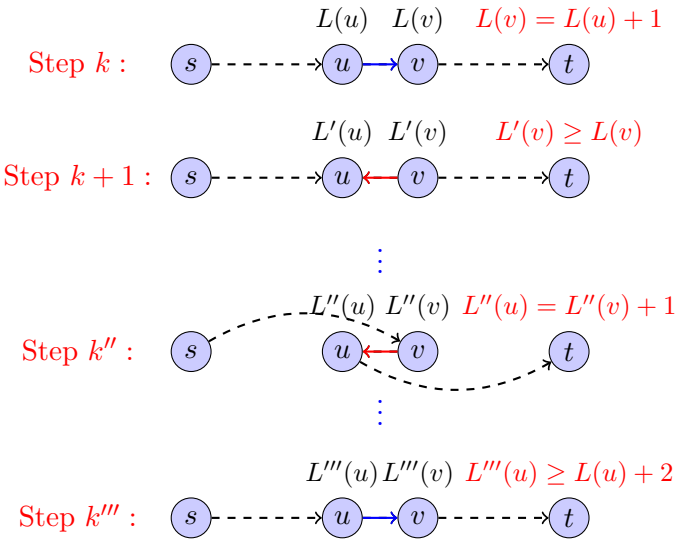
- 使用BFS寻找最短路径耗时为$O(m)$。

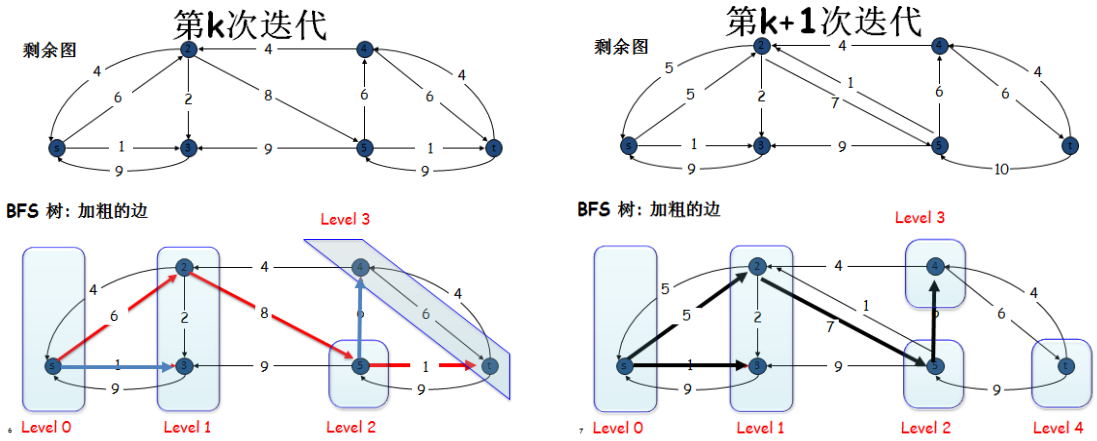
  则有总的时间复杂度为$O(m^2n)$。

□

以下为该定理的图形化解释

还有一个很关键的点是层数不会下降，如下图所示：





5 改进策略三：Dinitz算法及Dinic算法

5.1 原始的Dinitz算法

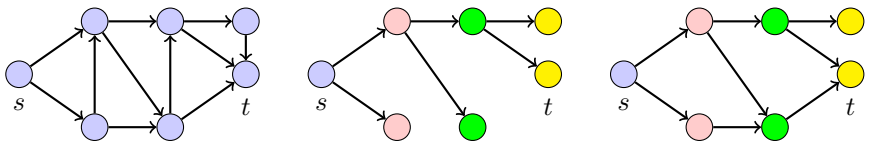
算法思想与FORD-FULKERSON算法类似，区别在于借助了一个数据结构。

在FORD-FULKERSON算法中寻找一条增广路径需要$O(m)$的时间，但利用BFS tree就可以存储子序列的迭代过程，就可很快的进行搜索。

Dinitz算法就是将BFS tree的思想加入到分层网络中：

- BFS tree: 仅包含到达v的第一条边
- Layered network: 包含剩余图中$s - v$的所有最短路径上的边。

如下图所示：



Residual graph G_f BFS tree Layered network G_L

操作较为繁琐，因此应用不太广泛。

5.2 Dinic算法

5.2.1 算法简述

Shimon Even和Alon Itai在Y.Dinitz的基础上又融合了A.Karzanov的思想。主要修改了以下两部分：

- 阻塞流（A.Karzanov首先提出）：用$O(m)$的时间构造分层网络G_L，每得到一个G_L，不断地进行增广，直到没有路径。
- 用DFS搜索增广路径。

5.2.2 算法实现

Dinic's algorithm

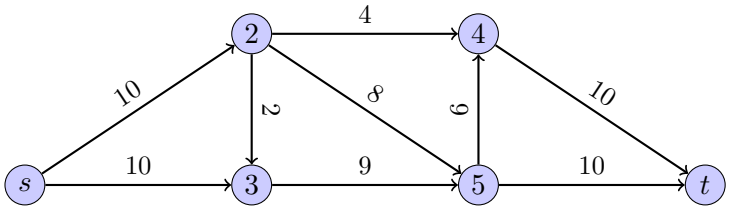
```
Initialize f(e) = 0 for all e.
while TRUE do
    Construct layered network G_L from residual graph G_f;
    if dist(s, t) = ∞ then
        break;
    end if
    find a blocking flow f' in G_L using DFS technique;
    augment flow f by f';
end while
return f;
```

- 这里，阻塞流就代表了在G_L网络中经过该流的增广，不再存在$s-t$的路径。
- 而在使用$O(m)$时间得到一个分层网络后，EDMONDS-KAPR每次只增广一条路径（效率较低）。

算法实例：

Dinic算法实例

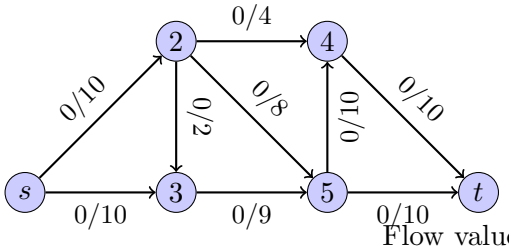
图G



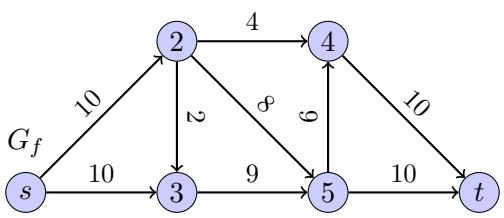
Flow $value = 0$

step1

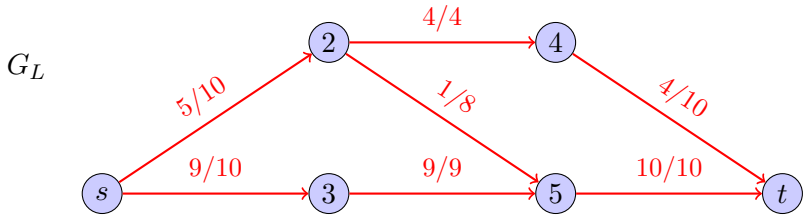
G



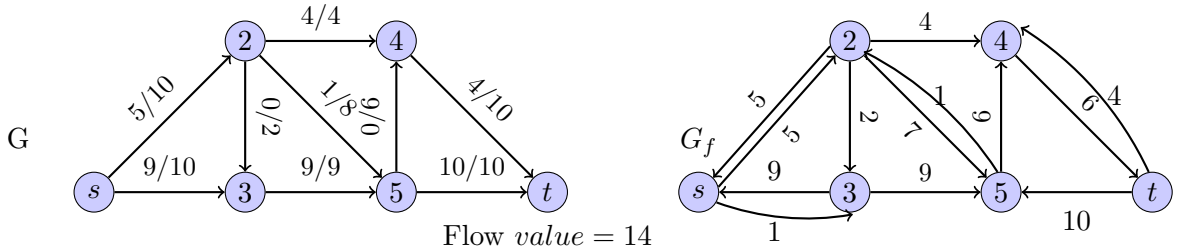
G_f



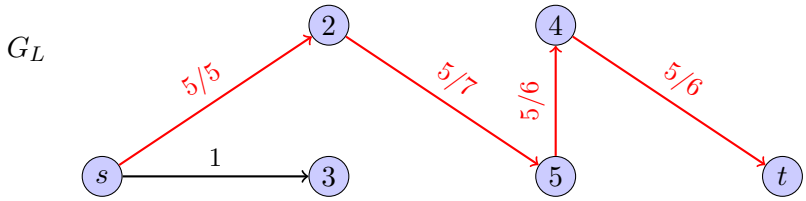
Flow value = 0



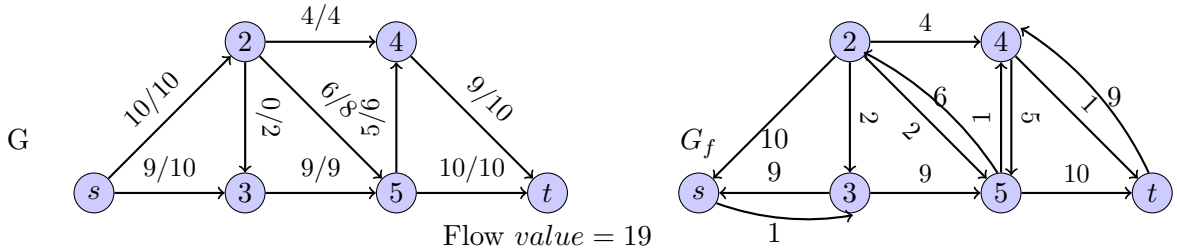
step2



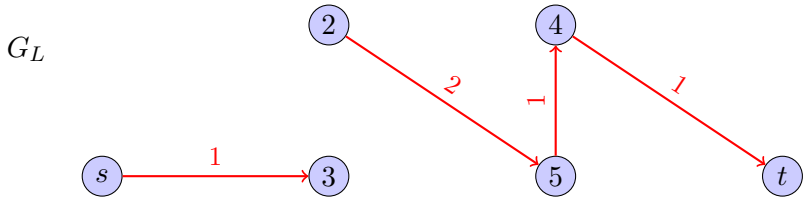
Flow *value* = 14



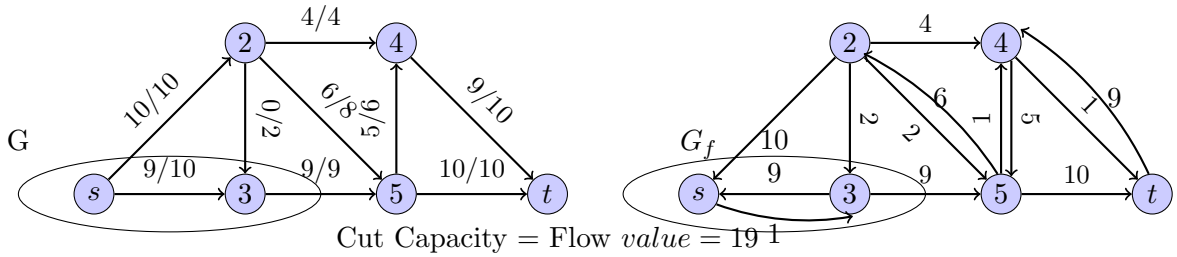
step3



Flow *value* = 19



step4



5.2.3 算法分析

总的时间复杂度为$O(mn^2)$

- 构造分层网络：$O(m)$（扩展的BFS）
- 寻找阻塞流：$O(mn)$。其中，

1) 用DFS在分层网络中寻找一条$s-t$路径需要$O(n)$时间。

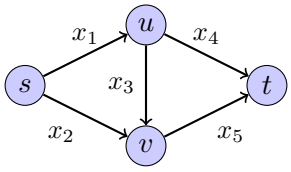
2) 每条路径至少饱和一个bottleneck边。因此，寻找一条阻塞流至多需要m 次迭代。

- $\#while = O(n)$(与EDMOND-KARP增广过程分析相同)
  因此总的时间复杂度为$O(mn^2)$

6 从对偶的角度理解网络流

另一个较为简单的思想是利用线性规划求解。

6.1 最大流-最小割：对偶问题



DUAL: 给边设置变量(x_i 表示边i的$flow$)

$$
\begin{array}{lccccccl}
\max & & & & & & f & \\
s.t. & x_1 & +x_2 & & & & -f & = 0 \text{ vertex } s \\
 & & & & -x_4 & -x_5 & +f & = 0 \text{ vertex } t \\
 & -x_1 & & +x_3 & +x_4 & & & = 0 \text{ vertex } u \\
 & & -x_2 & -x_3 & & +x_5 & & = 0 \text{ vertex } v \\
 & x_1 & & & & & & \le C_1 \\
 & & x_2 & & & & & \le C_2 \\
 & & & x_3 & & & & \le C_3 \\
 & & & & x_4 & & & \le C_4 \\
 & & & & & x_5 & & \le C_5 \\
 & x_1, & x_2, & x_3, & x_4, & x_5 & & \ge 0
\end{array}
$$

则可通过单纯形法得到得到最大流，但由于该问题较为特殊，因此还有其他解决方法。以下是一个等价的线性规划表示版本：

$$
\begin{array}{lccccccccc}
\max & & & & & & & f & & \\
s.t. & x_1 & +x_2 & & & & -f & \leq 0 & \text{vertex } s \\
 & & & & -x_4 & -x_5 & +f & \leq 0 & \text{vertex } t \\
 & -x_1 & & +x_3 & +x_4 & & & \leq 0 & \text{vertex } u \\
 & & -x_2 & -x_3 & & +x_5 & & \leq 0 & \text{vertex } v \\
 & x_1 & & & & & & \leq C_1 & \\
 & & x_2 & & & & & \leq C_2 & \\
 & & & x_3 & & & & \leq C_3 & \\
 & & & & x_4 & & & \leq C_4 & \\
 & & & & & x_5 & & \leq C_5 & \\
 & x_1, & x_2, & x_3, & x_4, & x_5 & & \geq 0 &
\end{array}
$$

由于约束(1),(2),(3),(4)中任意三个不等式相加，所得到的不等式与第四个不等式联立，即为第四个等式，所以这两组线性规划是等价的。

6.2 原始问题

PRIMAL：给节点设定变量

$$
\begin{array}{lccccccccccc}
\min & & & & & C_1z_1 & +C_2z_2 & +C_3z_3 & +C_4z_4 & +C_5z_5 & & \\
s.t. & y_s & & -y_u & & +z_1 & & & & & & \geq 0 \\
 & y_s & & & -y_v & & +z_2 & & & & & \geq 0 \\
 & & & y_u & -y_v & & & +z_3 & & & & \geq 0 \\
 & & -y_t & +y_u & & & & & +z_4 & & & \geq 0 \\
 & & -y_t & & +y_v & & & & & +z_5 & & \geq 0 \\
 & -y_s & +y_t & & & & & & & & & \geq 1 \\
 & y_s, & y_t, & y_u, & y_v, & z_1, & z_2, & z_3, & z_4, & z_5 & & \geq 0
\end{array}
$$

对该线性规划的分析：

- 因为约束条件中y_i的出现是相对的，因此可以设定其中一个y_i的值，不妨令$y_s = 0$，则由约束条件(6)可得$y_t \geq 1$。
- 由约束条件(4)$z_4 \geq y_t - y_u$，又目标函数是使得C_4z_4尽可能的小，因此有$y_t = 1$。
- 由约束条件(1)$z_1 \geq y_u$，又目标函数是使得C_1z_1尽可能的小，因此有$z_1 = y_u$。

因此原问题可化为下述等价版本：

$$
\begin{array}{lccccccccc}
\min & & & & C_1z_1 & +C_2z_2 & +C_3z_3 & +C_4z_4 & +C_5z_5 & \\
s.t. & & -y_u & & +z_1 & & & & & = 0 \\
 & & & -y_v & & +z_2 & & & & = 0 \\
 & & y_u & -y_v & & & +z_3 & & & \geq 0 \\
 & & y_u & & & & & +z_4 & & \geq 1 \\
 & & & y_v & & & & & +z_5 & \geq 1 \\
 & y_s & & & & & & & & = 0 \\
 & & y_t & & & & & & & = 1 \\
 & & y_u, & y_v, & z_1, & z_2, & z_3, & z_4, & z_5 & \geq 0
\end{array}
$$

由于约束的系数矩阵是全单模的，因此最优解一定是整数解。所以变量是0/1变量。原问题又可以等价为：

$$
\begin{array}{lccccccccc}
\min & & & & C_1z_1 & +C_2z_2 & +C_3z_3 & +C_4z_4 & +C_5z_5 & \\
s.t. & & -y_u & & +z_1 & & & & & = 0 \\
& & & -y_v & & +z_2 & & & & = 0 \\
& & y_u & -y_v & & & +z_3 & & & \geq 0 \\
& & y_u & & & & & +z_4 & & \geq 1 \\
& & & y_v & & & & & +z_5 & \geq 1 \\
& y_s & & & & & & & & = 0 \\
& y_t & & & & & & & & = 1 \\
& & y_u, & y_v, & z_1, & z_2, & z_3, & z_4, & z_5 & = 0/1
\end{array}
$$

直观的理解，z_i表示第i条边，$z_i = 1$表示选择第i条边作为割边，则最终目标函数就是求最小割。即最大流的对偶问题是最小割问题。

- 如果节点i在A中，则$y_i = 0$(由$z_i = 1$可得)，否则$y_i = 1$。
- 由弱对偶性，$f \leq c$（流≤ 割），即最大流最小割定理。

6.3 FORD-FULKERSON算法本质上是一个原始对偶算法

$$
\begin{array}{lccccccc}
\max & & & & & & f & \\
s.t. & x_1 & +x_2 & & & & -f & \leq 0 \text{ vertex } s \\
& & & & -x_4 & -x_5 & +f & \leq 0 \text{ vertex } t \\
& -x_1 & & +x_3 & +x_4 & & & \leq 0 \text{ vertex } u \\
& & -x_2 & -x_3 & & +x_5 & & \leq 0 \text{ vertex } v \\
& x_1 & & & & & & \leq C_1 \\
& & x_2 & & & & & \leq C_2 \\
& & & x_3 & & & & \leq C_3 \\
& & & & x_4 & & & \leq C_4 \\
& & & & & x_5 & & \leq C_5 \\
& x_1, & x_2, & x_3, & x_4, & x_5 & & \geq 0
\end{array}
$$

对该对偶问题转化为DRP形式

- 将约束右边的C_i换成0。
- 增加约束$x_i \leq 1, f \leq 1$。
- 将J中的约束条件分为两类，$J = J^S \bigcup J^E$，其中，$J^S = i|x_i = C_i$在对偶D中，$J^E = j|x_j = C_j$在对偶D 中，直观上来看，J^S表示饱和边，而J^E表示空边(边上流值为0)。

DRP描述：

- DRP:

$$
\begin{array}{lccccccc}
\max & & & & & & f & \\
s.t. & x_1 & +x_2 & & & & -f & = 0 \text{ vertex } s \\
& & & & -x_4 & -x_5 & +f & = 0 \text{ vertex } t \\
& -x_1 & & +x_3 & +x_4 & & & = 0 \text{ vertex } u \\
& & -x_2 & -x_3 & & +x_5 & & = 0 \text{ vertex } v \\
& & & x_i & & & & \leq 0 \quad i \in J^S \\
& & & x_j & & & & \geq 0 \quad j \in J^E \\
& x_1, & x_2, & x_3, & x_4, & x_5, & f & \leq 1
\end{array}
$$

- Ford-Fulkerson 算法本质上是原始对偶算法。
- 由于$\omega_{OPT} \leq 1$，又全单模系数矩阵使得解的值为整数，因此ω_{OPT}只有以下两种情况
  - $\omega_{OPT} = 0$ 意味着已找到最优解。
  - $\omega_{OPT} = 1$ 代表在剩余图G_f中的一条$s-t$路径。
- 为什么会构成剩余图G_f？

  $x_i \leq 0, i \in J^S$ 代表反向边, $x_j \geq 0, j \in J^E$ 代表正向边，对于其他的边x_i没有约束。

7 Push-relabel算法

Push-relabel算法是求最大流的一个更高效的算法。初始版本的复杂性为$O(n^2m)$(与Dinic算法时间复杂性相当).而选用如下数据结构可使得算法复杂性明显提升。

- 选用FIFO顶点选择规则　　$O(n^3)$，
- 选用最高活跃顶点选择规则　　$O(n^2\sqrt{m})$，
- 选用Sleator,Tarjan动态树数据结构　　$O(mn\log(n/m))$

7.1 Push-relabel算法简介

基本思想：增广流的基本思想是每次维持对偶线性规划的可行性，而push-relabel方法每次维持原问题的可行性。

先来介绍几个定义。

7.1.1 预流(pre-flow)

定义1. *若f满足以下两条，称f是预流*

- *容量条件(Capacity condition)*：$f(e) \leq C(e)$
- *超额条件(Excess condition)*：任意节点$v \neq s, E_f(v) = \sum_{eintov} f(e) - \sum_{eoutofv} f(e) \geq 0$

  若所有仓库都是空的($\forall v, E_f(v) = 0$)，则称预流f是流。

7.1.2 标号(label)

定义2. *（有效标号）对每个节点$v \in V$和一个预流f，其高度$h(v)$满足：*

- $h(t) = 0, h(s) = n$
- *对于剩余图G_f的每条边(u,v)，均有$h(u) \leq h(v) + 1$(G_f中的有向边(u,v),v 比u最多低1)*

7.2 Push-relabel算法描述

- Ford-Fulkerson: 对边设置变量，每次对G_f中的边更新流，直到G_f 中不存在$s-t$路径。
- Push-relabel: 对节点设置变量，每次更新预流f，维持G_f中午$s-t$路径的性质，直到f是一个流。

与Ford-Fulkerson算法的区别形式化语言描述：

```
Ford-Fulkerson
f is a flow
while(exist a s-t path){
    sustain:f to a flow
}
```

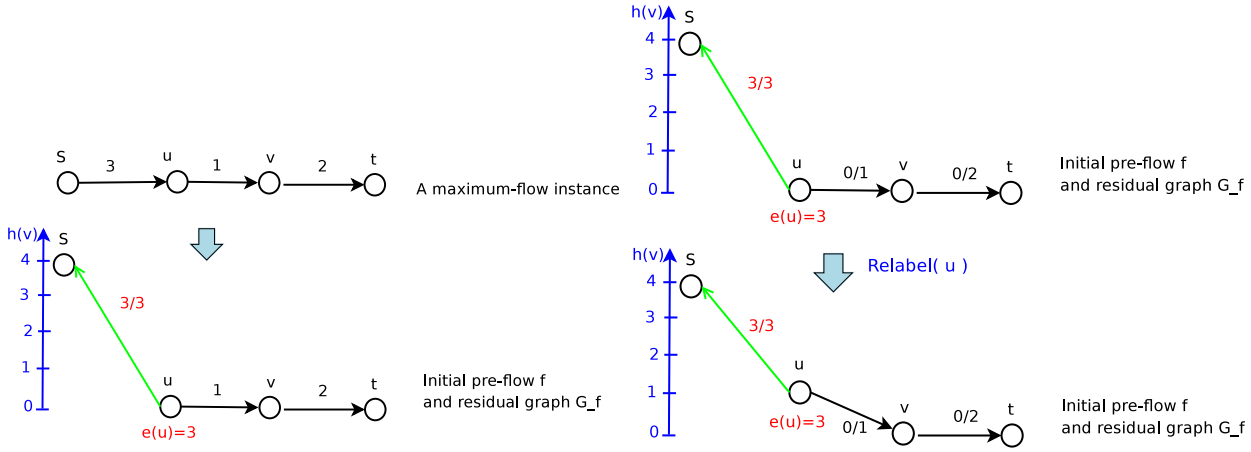
```
Push-relabel
f is a preflow
while(f is not a flow){
    sustain: not exist s-t path
}
```

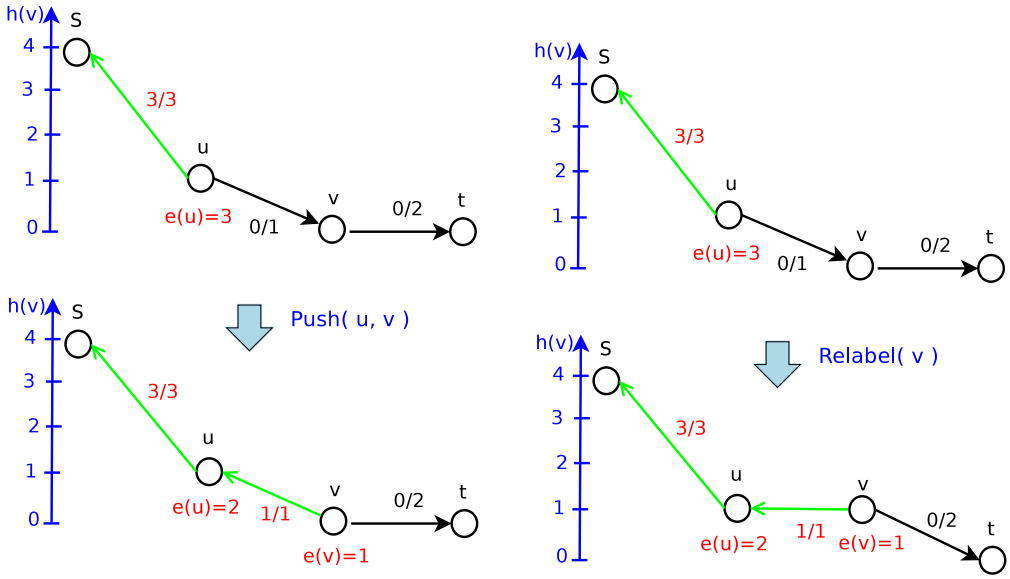
还剩下一个问题，如何标识G_f中无$s-t$ 路径？标号法

定理10. *若在G_f中存在有效标号(valid label)，则G_f中不存在$s-t$路径。*

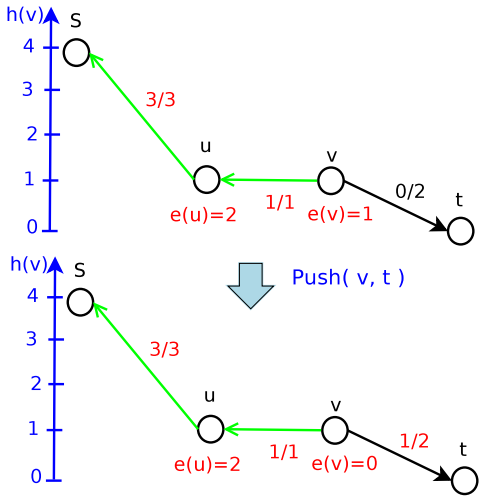
证明. 假设G_f中存在$s-t$路径，则$s-t$路径包含至多$n-1$条边。由于$h(s)=n$且$h(u)\le h(v)+1$，则t的高度应大于0，与$h(t)=0$矛盾。 □

下面用图形来描述，这里只描述了网络中的其中一条路径。

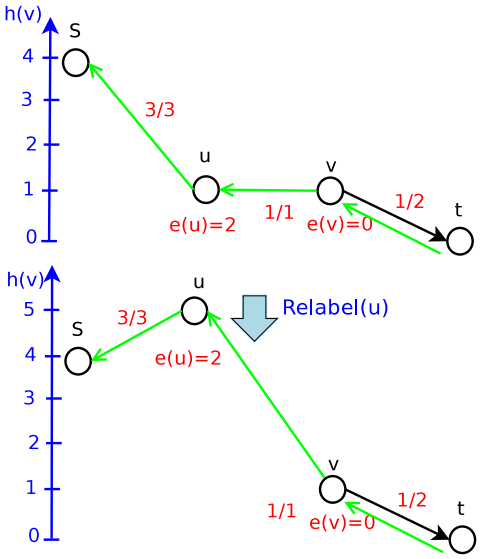




Step2

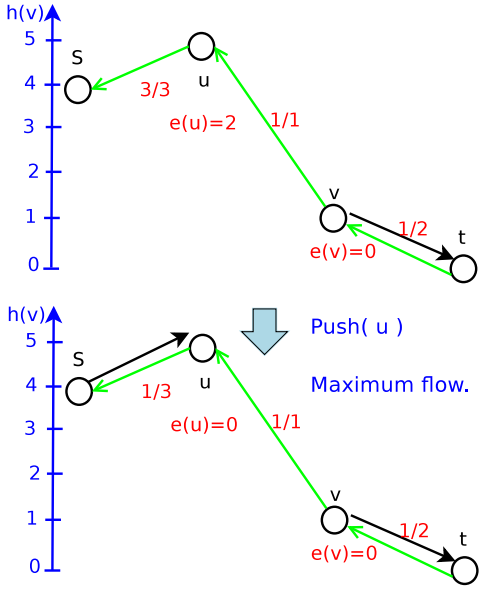


Step3



Step4

Step5



Step6

7.3 Push-relabel算法实现

High-level算法：

初始化：

- 预流设置：初始时将所有货物均运出，$f(s,u)=C(s,u)$，且对其他边有$f(u,v)=0$。
- 标号设置：$h(s)=n$，其他顶点有$h(v)=0$。

迭代过程：每一步，对于$E(f)>0$的节点v

- 若存在一个标号小于v的邻居节点u，则增加v指向u的流。

- 否则，在标号合理的前提下，增加其高度$h(v)$。

Push-relabel algorithm:

1: $h(s) = n$;
2: $h(v) = 0$; for any $v \neq s$;
3: $f(e) = C(e)$ for all $e = (s, u)$;
4: $f(e) = 0$; for other edges;
5: **while** there exists a node v with $E_f(v) > 0$ **do**
6: **if** there exists an edge $(v, w) \in G_f$ s.t. $h(v) > h(w)$; **then**
7: //Push excess from v to w;
8: **if** (v, w) is a forward edge; **then**
9: $e = (v, w)$;
10: $bottleneck = \min\{E_f(v), C(e) - f(e)\}$;
11: $f(e) += bottleneck$;
12: **else**
13: $e = (w, v)$;
14: $bottleneck = \min\{E_f(v), f(e)\}$;
15: $f(e) -= bottleneck$;
16: **end if**
17: **else**
18: $h(v) = h(v) + 1$; //Relabel node v;
19: **end if**
20: **end while**

时间复杂性：$T = O(n^2m)$