

# Advanced Graph Algorithms

(Based on [Manber 1989])

Yih-Kuen Tsay

Department of Information Management  
National Taiwan University

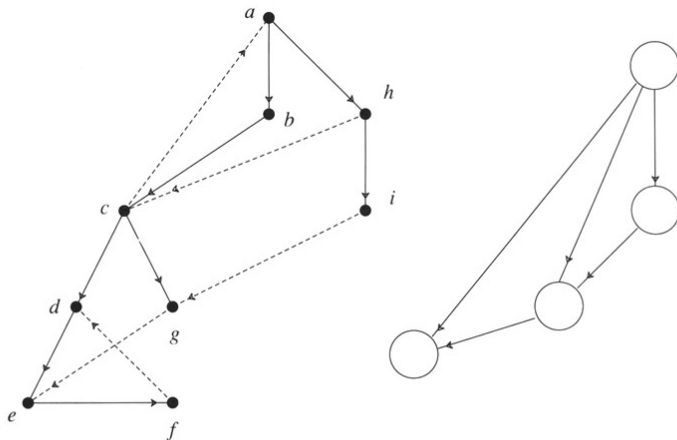
# Strongly Connected Components

- 🌐 A directed graph is *strongly connected* if there is a directed path from every vertex to every other vertex.

# Strongly Connected Components

- 🌐 A directed graph is *strongly connected* if there is a directed path from every vertex to every other vertex.
- 🌐 A *strongly connected component* (SCC) is a maximal subset of the vertices such that its induced subgraph is strongly connected (namely, there is no other subset that contains it and induces a strongly connected graph).

# Strongly Connected Components (cont.)



**Figure 7.30** A directed graph and its strongly connected component graph.

Source: [Manber 1989].

## Lemma (7.11)

*Two distinct vertices belong to the same SCC if and only if there is a circuit containing both of them.*

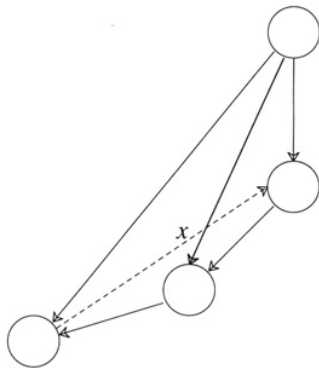
## Lemma (7.11)

*Two distinct vertices belong to the same SCC if and only if there is a circuit containing both of them.*

## Lemma (7.12)

*Each vertex belongs to exactly one SCC.*

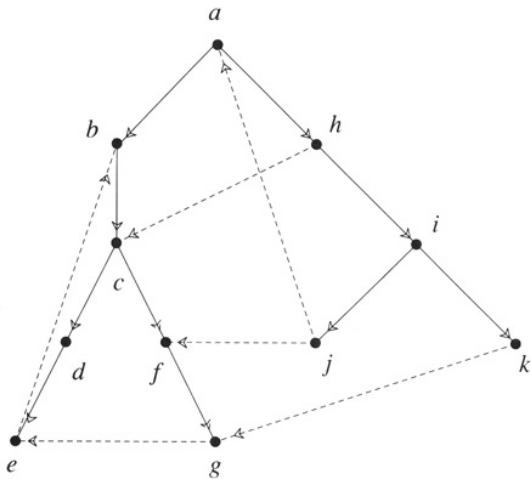
# Strongly Connected Components (cont.)



**Figure 7.31** Adding an edge connecting two different strongly connected components.

Source: [Manber 1989].

# Strongly Connected Components (cont.)



**Figure 7.32** The effect of cross edges.

Source: [Manber 1989].

Yih-Kuen Tsay (IM.NTU)

Advanced Graph Algorithms

Algorithms 2018

6 / 31



## Strongly Connected Components (cont.)

**Algorithm** `Strongly_Connected_Components( $G, n$ )`;

**begin**

**for** every vertex  $v$  of  $G$  **do**

$v.DFS\_Number := 0$ ;

$v.Component := 0$ ;

$Current\_Component := 0$ ;  $DFS\_N := n$ ;

**while**  $v.DFS\_Number = 0$  for some  $v$  **do**

$SCC(v)$

**end**

**procedure**  $SCC(v)$ ;

**begin**

$v.DFS\_Number := DFS\_N$ ;

$DFS\_N := DFS\_N - 1$ ;

    insert  $v$  into  $Stack$ ;

$v.High := v.DFS\_Number$ ;

## Strongly Connected Components (cont.)

```
for all edges  $(v, w)$  do  
    if  $w.DFS\_Number = 0$  then  
         $SCC(w)$ ;  
         $v.High := \max(v.High, w.High)$   
    else if  $w.DFS\_Number > v.DFS\_Number$   
        and  $w.Component = 0$  then  
             $v.High := \max(v.High, w.DFS\_Number)$   
            //  $\max(v.High, w.High)$  also works  
if  $v.High = v.DFS\_Number$  then  
     $Current\_Component := Current\_Component + 1$ ;  
    repeat  
        remove  $x$  from the top of  $Stack$ ;  
         $x.component := Current\_Component$   
    until  $x = v$   
end
```

## Strongly Connected Components (cont.)

```
for all edges  $(v, w)$  do
  if  $w.DFS\_Number = 0$  then
     $SCC(w)$ ;
     $v.High := \max(v.High, w.High)$ 
  else if  $w.DFS\_Number > v.DFS\_Number$ 
    and  $w.Component = 0$  then
     $v.High := \max(v.High, w.DFS\_Number)$ 
    //  $\max(v.High, w.High)$  also works
if  $v.High = v.DFS\_Number$  then
   $Current\_Component := Current\_Component + 1$ ;
  repeat
    remove  $x$  from the top of  $Stack$ ;
     $x.component := Current\_Component$ 
  until  $x = v$ 
end
```

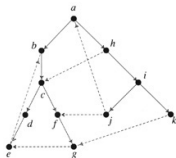
Time complexity:

## Strongly Connected Components (cont.)

```
for all edges  $(v, w)$  do
  if  $w.DFS\_Number = 0$  then
     $SCC(w)$ ;
     $v.High := \max(v.High, w.High)$ 
  else if  $w.DFS\_Number > v.DFS\_Number$ 
    and  $w.Component = 0$  then
     $v.High := \max(v.High, w.DFS\_Number)$ 
    //  $\max(v.High, w.High)$  also works
  if  $v.High = v.DFS\_Number$  then
     $Current\_Component := Current\_Component + 1$ ;
    repeat
      remove  $x$  from the top of  $Stack$ ;
       $x.component := Current\_Component$ 
    until  $x = v$ 
end
```

Time complexity:  $O(|E| + |V|)$ .

# Strongly Connected Components (cont.)



	a	b	c	d	e	f	g	h	i	j	k
High	11	10	9	8	7	6	5	4	3	2	1
a	11	-	-	-	-	-	-	-	-	-	-
b	11	10	-	-	-	-	-	-	-	-	-
c	11	10	9	-	-	-	-	-	-	-	-
d	11	10	9	8	-	-	-	-	-	-	-
e	11	10	9	8	10	-	-	-	-	-	-
f	11	10	9	10	10	10	-	-	-	-	-
g	11	10	10	10	10	6	-	-	-	-	-
h	11	10	10	10	10	6	7	-	-	-	-
i	11	10	10	10	10	7	7	-	-	-	-
j	11	10	10	10	10	7	7	-	-	-	-
k	11	10	10	10	10	7	7	4	-	-	-
a	11	10	10	10	10	7	7	4	3	-	-
b	11	10	10	10	10	7	7	4	3	11	-
c	11	10	10	10	10	7	7	4	3	11	11
d	11	10	10	10	10	7	7	4	3	11	11
e	11	10	10	10	10	7	7	4	3	11	11
f	11	10	10	10	10	7	7	4	3	11	11
g	11	10	10	10	10	7	7	4	3	11	11
h	11	10	10	10	10	7	7	4	3	11	11
i	11	10	10	10	10	7	7	4	3	11	11
j	11	10	10	10	10	7	7	4	3	11	11
k	11	10	10	10	10	7	7	4	3	11	11

Figure 7.34 An example of computing High values and strongly connected components.

# Odd-Length Cycles

## Problem

*Given a directed graph  $G = (V, E)$ , determine whether it contains a (directed) cycle of odd length.*


# Odd-Length Cycles

## Problem

Given a directed graph  $G = (V, E)$ , determine whether it contains a (directed) cycle of odd length.

- 🌐 A cycle must reside completely within a strongly connected component (SCC), so we exam each SCC separately.
- 🌐 Mark the nodes of an SCC with “even” or “odd” using DFS.
- 🌐 If we have to mark a node that is already marked in the opposite, then we have found an odd-length cycle.

# Biconnected Components

-  An undirected graph is *biconnected* if there are at least two vertex-disjoint paths from every vertex to every other vertex.



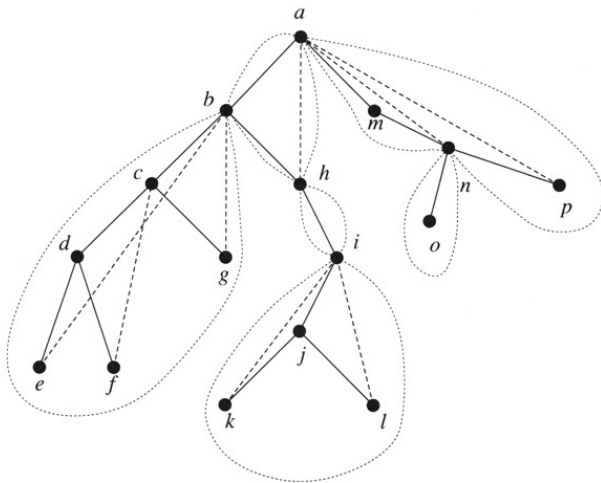
# Biconnected Components

- 🌐 An undirected graph is *biconnected* if there are at least two vertex-disjoint paths from every vertex to every other vertex.
- 🌐 A graph is *not* biconnected if and only if there is a vertex whose removal disconnects the graph. Such a vertex is called an *articulation point*.

# Biconnected Components

- 🌐 An undirected graph is *biconnected* if there are at least two vertex-disjoint paths from every vertex to every other vertex.
- 🌐 A graph is *not* biconnected if and only if there is a vertex whose removal disconnects the graph. Such a vertex is called an *articulation point*.
- 🌐 A *biconnected component* (BCC) is a *maximal* subset of the edges such that its induced subgraph is biconnected (namely, there is no other subset that contains it and induces a biconnected graph).

# Biconnected Components (cont.)



**Figure 7.25** The structure of a nonbiconnected graph.

Source: [Manber 1989].

Yih-Kuen Tsay (IM.NTU)

## Lemma (7.9)

*Two distinct edges  $e$  and  $f$  belong to the same BCC if and only if there is a cycle containing both of them.*

# Biconnected Components (cont.)

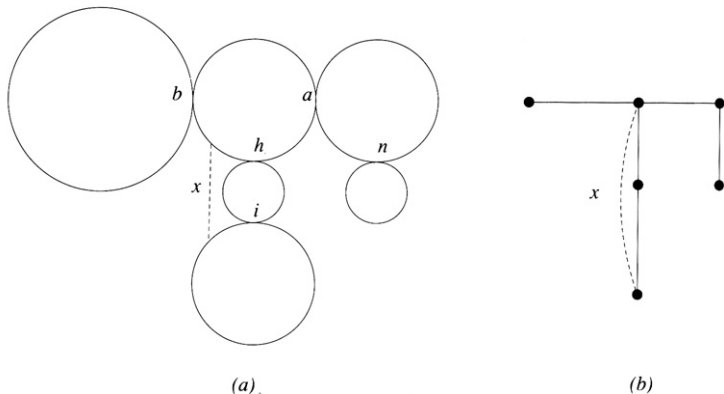
## Lemma (7.9)

*Two distinct edges  $e$  and  $f$  belong to the same BCC if and only if there is a cycle containing both of them.*

## Lemma (7.10)

*Each edge belongs to exactly one BCC.*

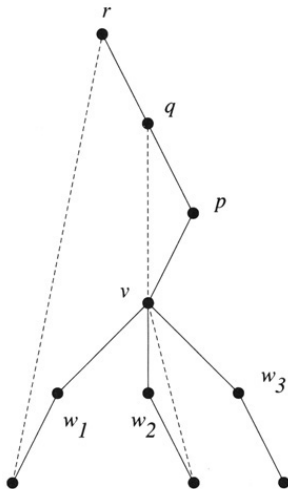
# Biconnected Components (cont.)



**Figure 7.26** An edge that connects two different biconnected components. (a) The components corresponding to the graph of Fig. 7.25 with the articulation points indicated. (b) The biconnected component tree.

Source: [Manber 1989].

# Biconnected Components (cont.)



**Figure 7.27** Computing the *High* values.

Source: [Manber 1989].

Yih-Kuen Tsay (IM.NTU)

# Biconnected Components (cont.)

```
Algorithm Biconnected_Components( $G, v, n$ );  
begin  
  for every vertex  $w$  do  $w.DFS\_Number := 0$ ;  
   $DFS\_N := n$ ;  
   $BC(v)$   
end
```

```
procedure  $BC(v)$ ;  
begin  
   $v.DFS\_Number := DFS\_N$ ;  
   $DFS\_N := DFS\_N - 1$ ;  
  insert  $v$  into  $Stack$ ;  
   $v.High := v.DFS\_Number$ ;
```



# Biconnected Components (cont.)

```
for all edges  $(v, w)$  do  
    insert  $(v, w)$  into Stack;  
    if  $w$  is not the parent of  $v$  then  
        if  $w.DFS\_Number = 0$  then  
             $BC(w)$ ;  
        if  $w.High \leq v.DFS\_Number$  then  
            remove all edges and vertices  
            from Stack until  $v$  is reached;  
            insert  $v$  back into Stack;  
             $v.High := \max(v.High, w.High)$   
        else  
             $v.High := \max(v.High, w.DFS\_Number)$   
            //  $\max(v.High, w.High)$  would not work, unlike in SCC  
    end
```

## Biconnected Components (cont.)

procedure **BC**( $v$ );

**begin**

$v.DFS\_Number := DFS\_N$ ;

$DFS\_N := DFS\_N - 1$ ;

$v.High := v.DFS\_Number$ ;

**for** all edges  $(v, w)$  **do**

**if**  $w$  is not the parent of  $v$  **then**

    insert  $(v, w)$  into *Stack*;

**if**  $w.DFS\_Number = 0$  **then**

$BC(w)$ ;

**if**  $w.high \leq v.DFS\_Number$  **then**

      remove all edges from *Stack*

      until  $(v, w)$  is reached;

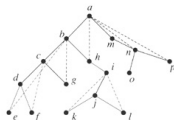
$v.High := \max(v.High, w.High)$

**else**

$v.High := \max(v.High, w.DFS\_Number)$

**end**

# Biconnected Components (cont.)



	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	
16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1		
a	16	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
b	16	15	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
c	16	15	14	-	-	-	-	-	-	-	-	-	-	-	-	-	-
d	16	15	14	13	-	-	-	-	-	-	-	-	-	-	-	-	-
e	16	15	14	13	15	-	-	-	-	-	-	-	-	-	-	-	-
f	16	15	14	15	15	14	-	-	-	-	-	-	-	-	-	-	-
g	16	15	14	15	15	14	-	-	-	-	-	-	-	-	-	-	-
h	16	15	15	15	15	14	15	-	-	-	-	-	-	-	-	-	-
i	16	15	15	15	15	14	15	16	-	-	-	-	-	-	-	-	-
j	16	15	15	15	15	14	15	16	8	-	-	-	-	-	-	-	-
k	16	15	15	15	15	14	15	16	8	7	-	-	-	-	-	-	-
l	16	15	15	15	15	14	15	16	8	7	8	-	-	-	-	-	-
m	16	15	15	15	15	14	15	16	8	8	8	-	-	-	-	-	-
n	16	15	15	15	15	14	15	16	8	8	8	8	-	-	-	-	-
o	16	15	15	15	15	14	15	16	8	8	8	8	4	-	-	-	-
p	16	15	15	15	15	14	15	16	8	8	8	8	4	16	-	-	-
a	16	16	15	15	15	14	15	16	8	8	8	8	4	16	2	-	-
b	16	16	15	15	15	14	15	16	8	8	8	8	4	16	2	-	-
c	16	16	15	15	15	14	15	16	8	8	8	8	4	16	2	-	-
d	16	16	15	15	15	14	15	16	8	8	8	8	4	16	2	16	-
e	16	16	15	15	15	14	15	16	8	8	8	8	4	16	2	16	-
f	16	16	15	15	15	14	15	16	8	8	8	8	4	16	2	16	-
g	16	16	15	15	15	14	15	16	8	8	8	8	4	16	2	16	-
h	16	16	15	15	15	14	15	16	8	8	8	8	4	16	2	16	-
i	16	16	15	15	15	14	15	16	8	8	8	8	4	16	2	16	-
j	16	16	15	15	15	14	15	16	8	8	8	8	4	16	2	16	-
k	16	16	15	15	15	14	15	16	8	8	8	8	4	16	2	16	-
l	16	16	15	15	15	14	15	16	8	8	8	8	4	16	2	16	-
m	16	16	15	15	15	14	15	16	8	8	8	8	4	16	2	16	-
n	16	16	15	15	15	14	15	16	8	8	8	8	4	16	2	16	-
o	16	16	15	15	15	14	15	16	8	8	8	8	4	16	2	16	-
p	16	16	15	15	15	14	15	16	8	8	8	8	4	16	2	16	-

Figure 7.29 An example of computing *High* values and biconnected components.

Source: [Manber 1989].

Yih-Kuen Tsay (IM.NTU)

# Even-Length Cycles

## Problem

*Given a connected undirected graph  $G = (V, E)$ , determine whether it contains a cycle of even length.*

# Even-Length Cycles

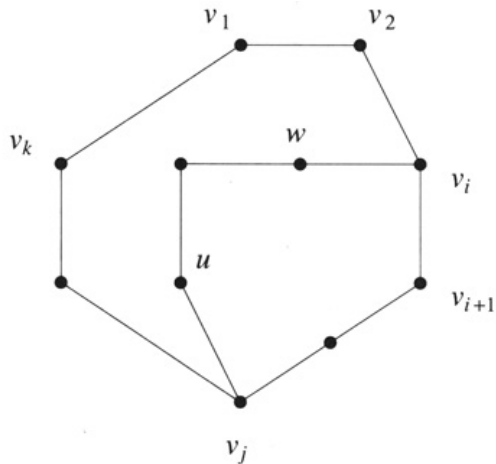
## Problem

*Given a connected undirected graph  $G = (V, E)$ , determine whether it contains a cycle of even length.*

## Theorem

*Every biconnected graph that has more than one edge and is not merely an odd-length cycle contains an even-length cycle.*

# Even-Length Cycles (cont.)



**Figure 7.35** Finding an even-length cycle.

# Network Flows

- Consider a directed graph, or network,  $G = (V, E)$  with two distinguished vertices:  $s$  (the **source**) with indegree 0 and  $t$  (the **sink**) with outdegree 0.
- Each edge  $e$  in  $E$  has an associated positive weight  $c(e)$ , called the *capacity* of  $e$ .

# Network Flows (cont.)

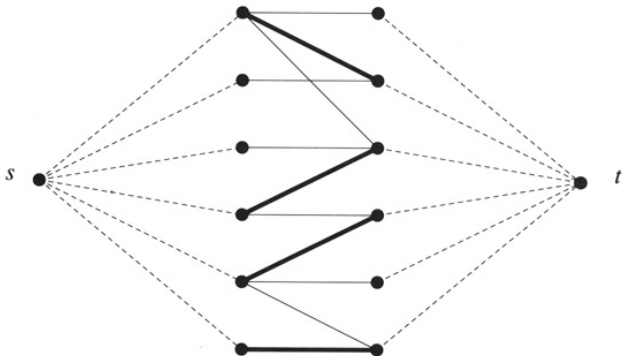
🌐 A **flow** is a function  $f$  on  $E$  that satisfies the following two conditions:

1.  $0 \leq f(e) \leq c(e)$ .

2.  $\sum_u f(u, v) = \sum_w f(v, w)$ , for all  $v \in V - \{s, t\}$ .

🌐 The **network flow problem** is to maximize the flow  $f$  for a given network  $G$ .







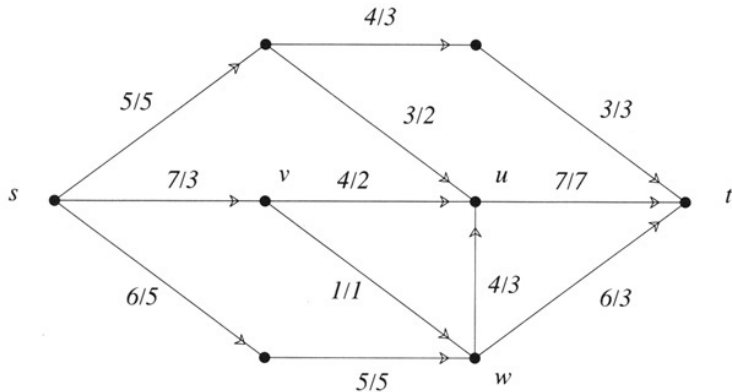
**Figure 7.39** Reducing bipartite matching to network flow (the directions of all the edges are from left to right).

Source: [Manber 1989].

# Augmenting Paths

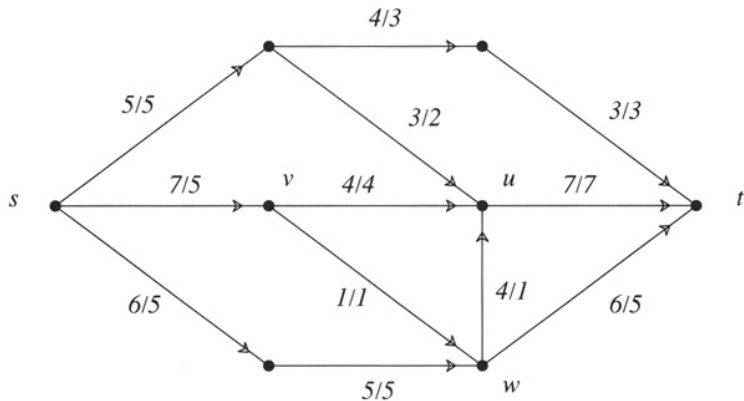
-  An **augmenting path** w.r.t. a given flow  $f$  (of a network  $G$ ) is a directed path from  $s$  to  $t$  consisting of edges from  $G$ , but not necessarily in the same direction; each of these edges  $(v, u)$  satisfies exactly one of:
1.  $(v, u)$  is in the same direction as it is in  $G$ , and  $f(v, u) < c(v, u)$ . (*forward edge*)
  2.  $(v, u)$  is in the opposite direction in  $G$  (namely,  $(u, v) \in E$ ), and  $f(u, v) > 0$ . (*backward edge*)
-  If there exists an augmenting path w.r.t. a flow  $f$  ( $f$  admits an augmenting path), then  $f$  is not maximum.

# Augmenting Paths (cont.)



**Figure 7.40** An example of a network with a (nonmaximum) flow.

# Augmenting Paths (cont.)



**Figure 7.41** The result of augmenting the flow of Fig. 7.40.

# Properties of Network Flows

## Theorem (Augmenting-Path)

*A flow  $f$  is maximum if and only if it admits no augmenting path.*

A *cut* is a set of edges that separate  $s$  from  $t$ , or more precisely a set of the form  $\{(v, w) \in E \mid v \in A \text{ and } w \in B\}$ , where  $B = V - A$  such that  $s \in A$  and  $t \in B$ .

## Theorem (Max-Flow Min-Cut)



*The value of a maximum flow in a network is equal to the minimum capacity of a cut.*

# Properties of Network Flows (cont.)

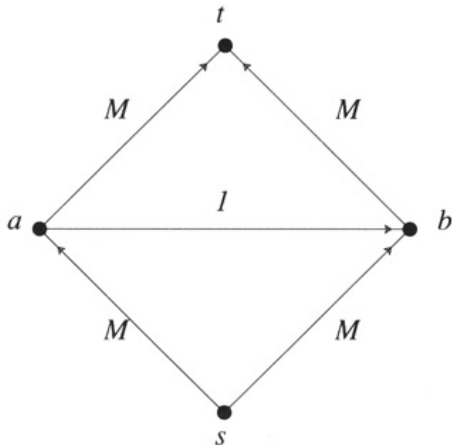
## Theorem (Integral-Flow)

*If the capacities of all edges in the network are integers, then there is a maximum flow whose value is an integer.*

# Residual Graphs

-  The **residual graph** with respect to a network  $G = (V, E)$  and a flow  $f$  is the network  $R = (V, F)$ , where  $F$  consists of all forward and backward edges and their capacities are given as follows:
1.  $c_R(v, w) = c(v, w) - f(v, w)$  if  $(v, w)$  is a forward edge and
  2.  $c_R(v, w) = f(w, v)$  if  $(v, w)$  is a backward edge.
-  An augmenting path is thus a regular directed path from  $s$  to  $t$  in the residual graph.

# Residual Graphs (cont.)



**Figure 7.42** A bad example of network flow.

Source: [Manber 1989].