Graph Algorithms

Yih-Kuen Tsay

Dept. of Information Management

National Taiwan University

The Königsberg Bridges Problem

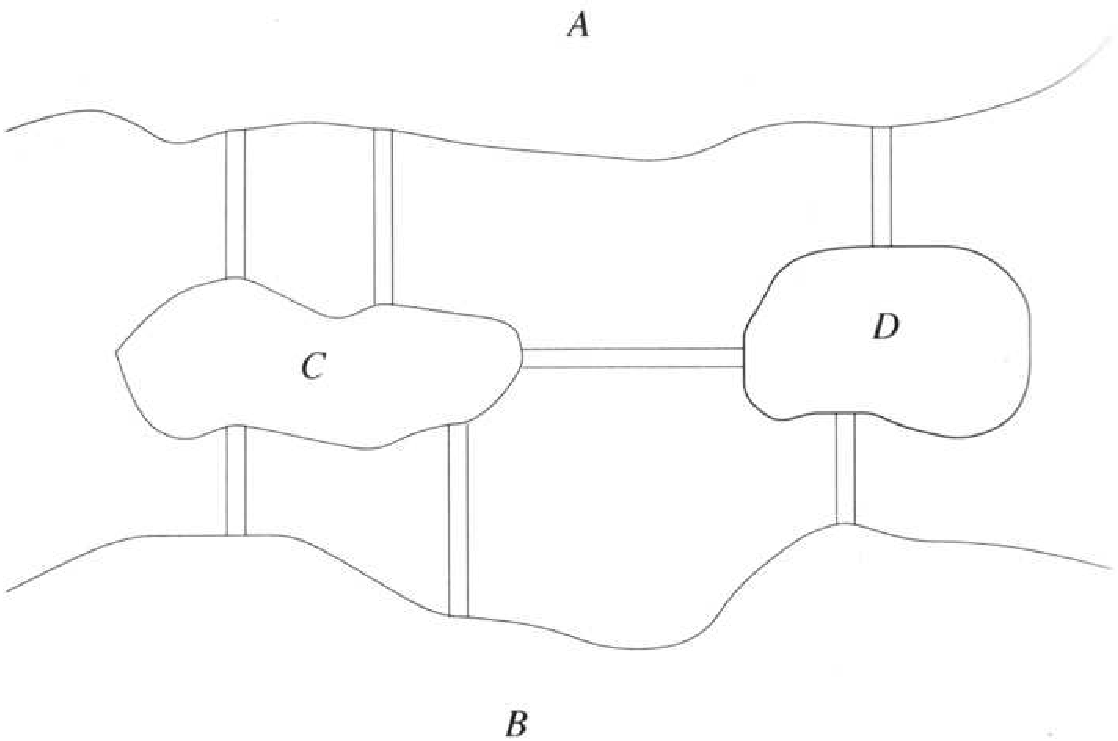


Figure 7.1 The Königsberg bridges problem.

Source: Manber 1989

Can one start from one of the lands, cross every bridge exactly once, and return to the origin?

The Königsberg Bridges Problem (cont.)

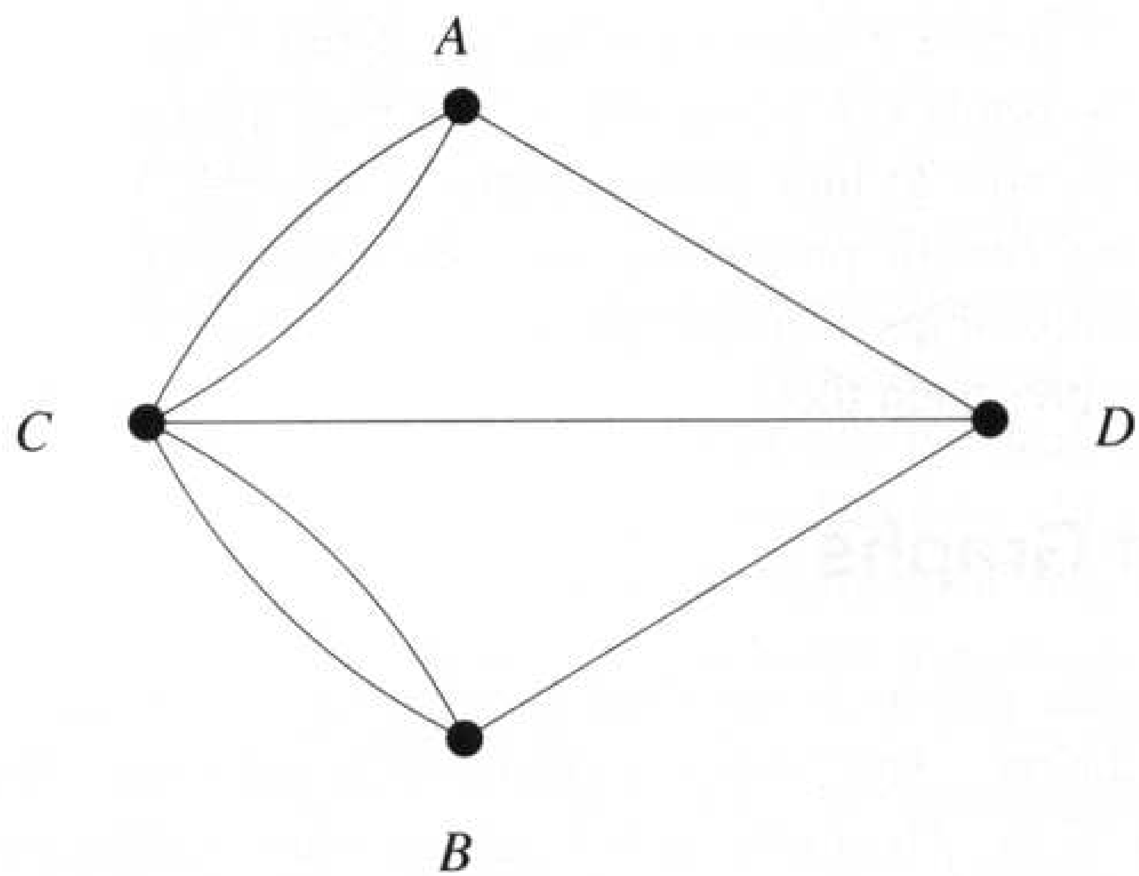


Figure 7.2 The graph corresponding to the Königsberg bridges problem.

Source: Manber 1989

Graphs

A graph consists of a set of vertices (or nodes) and a set of edges (or links, each normally connecting two vertices) and is commonly denoted as $G(V, E)$, where

- G is the name of the graph,
- V is the set of vertices, and
- E is the set of edges.

Modeling with Graphs

- Reachability
- Shortest Routes
- Scheduling

Graphs (cont.)

- Undirected vs. Directed Graphs
- Paths, Simple Paths, Trails
- Circuits, Cycles
- Degrees, In-Degrees, Out-Degrees
- Connected Graphs, Trees
- Subgraphs, Induced Subgraphs, Spanning Trees

Eulerian Graphs

The Problem Given an undirected connected graph $G = (V, E)$ such that all the vertices have even degrees, find a circuit P such that each edge of E appears in P exactly once.

The circuit P in the problem statement is called an *Eulerian circuit*.

Theorem
An undirected connected graph has an Eulerian circuit if and only if all of its vertices have even degrees.

Depth First Search

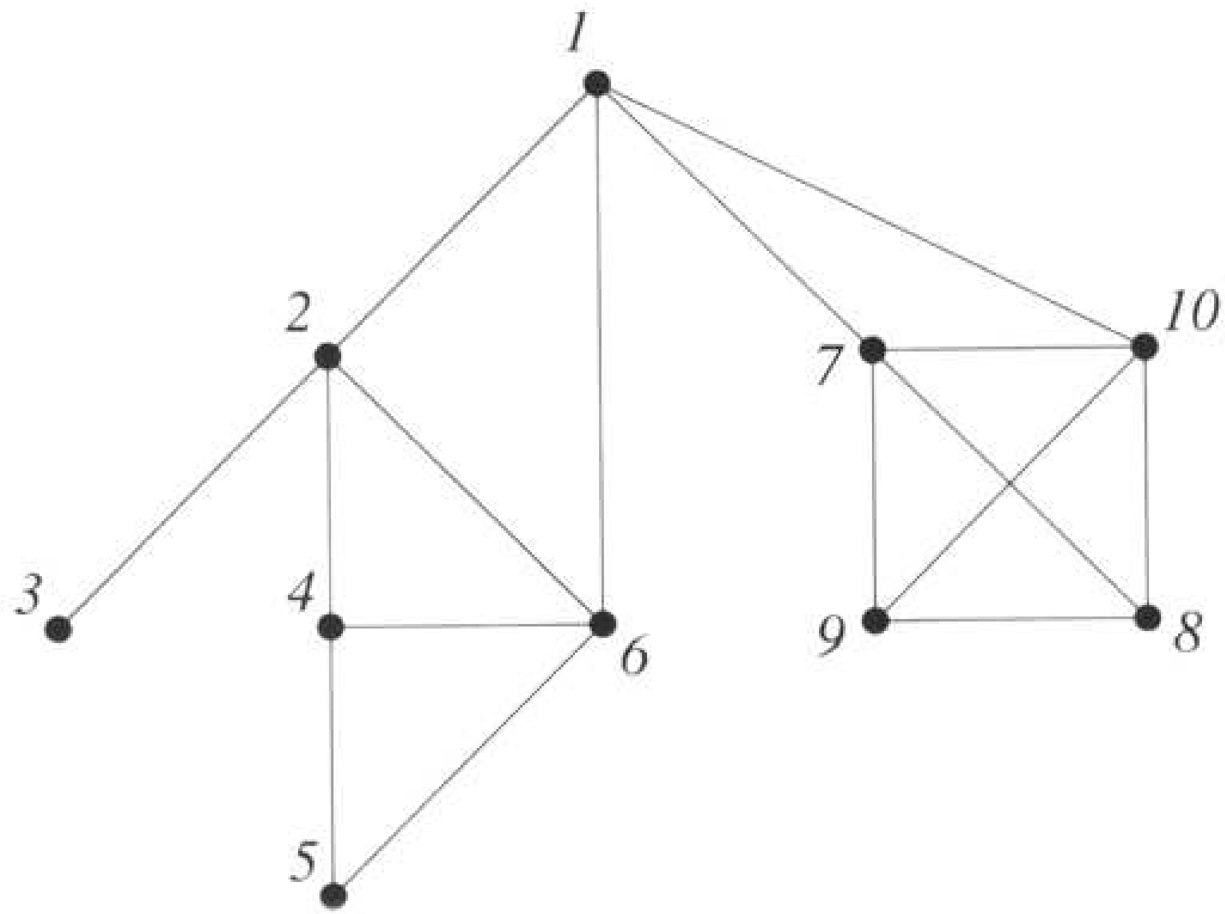


Figure 7.4 A DFS for an undirected graph.

Source: Manber 1989

Depth First Search (cont.)

Algorithm Depth_First_Search(G, v);
begin
    mark v;
    perform preWORK on v;
    for all edges (v, w) **do**
        if w is unmarked **then**
            $Depth\_First\_Search(G, w)$;
        perform postWORK for (v, w)
end

Depth First Search (cont.)

Algorithm Refined_DFS(G, v);
begin
    mark v;
    perform preWORK on v;
    for all edges (v, w) **do**
        if w is unmarked **then**
            $Refined\_DFS(G, w)$;
        perform postWORK for (v, w);
    perform postWORK_II on v
end

Connected Components

Algorithm Connected_Components(G);
begin
$Component\_Number$:= 1;
while there is an unmarked vertex v **do**
$Depth\_First\_Search(G, v)$
(preWORK:
$v.Component$:= $Component\_Number$);
$Component\_Number$:= $Component\_Number + 1$
end

DFS Numbers

Algorithm DFS_Numbering(G, v);
begin

DFS_Number := 1;
$Depth\_First\_Search(G, v)$
(preWORK:
$v.DFS$:= DFS_Number;
DFS_Number := $DFS\_Number + 1$)

end

The DFS Tree

Algorithm Build_DFS_Tree(G, v);
begin
    $Depth\_First\_Search(G, v)$
    (postWORK:
        if w was unmarked **then**
            add the edge (v, w) to T);
end

The DFS Tree (cont.)

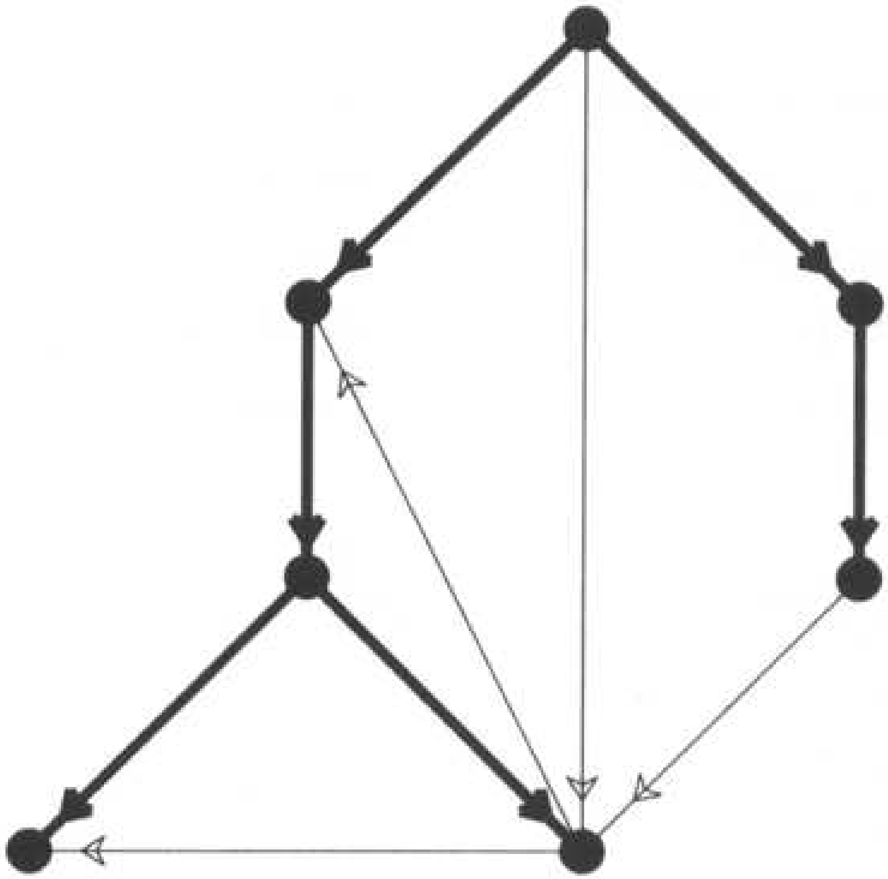


Figure 7.9 A DFS tree for a directed graph.

Source: Manber 1989

The DFS Tree (cont.)

Lemma 7.2
For an undirected graph $G = (V, E)$, every edge $e \in E$ either belongs to the DFS tree T, or connects two vertices of G, one of which is the ancestor of the other in T.

For undirected graphs, DFS avoids cross edges.

Lemma 7.3
For a directed graph $G = (V, E)$, if (v, w) is an edge in E such that $v.DFS\_Number < w.DFS\_Number$, then w is a descendant of v in the DFS tree T.

For directed graphs, cross edges must go "from right to left".

Directed Cycles

The Problem Given a directed graph $G = (V, E)$, determine whether it contains a (directed) cycle.

Lemma 7.4
G contains a directed cycle if and only if G contains a back edge (relative to the DFS tree).

Directed Cycles (cont.)

Algorithm Find_a_Cycle(G);
begin
  $Depth\_First\_Search(G, v)$ /* arbitrary v */
  (preWORK:
    $v.on\_the\_path$:= $true$;
  postWORK:
    if $w.on\_the\_path$ **then**
      $Find\_a\_Cycle$:= $true$;
      halt;
    if w is the last vertex on v's list **then**
      $v.on\_the\_path$:= $false$;)
end

Directed Cycles (cont.)

Algorithm Refined_Find_a_Cycle(G);
begin
    $Refined\_DFS(G, v)$ /* arbitrary v */
    (preWORK:
        $v.on\_the\_path$:= $true$;
    postWORK:
        if $w.on\_the\_path$ **then**
            $Refined\_Find\_a\_Cycle$:= $true$;
            halt;
    postWORK_II:
        $v.on\_the\_path$:= $false$)
end

Breadth-First Search

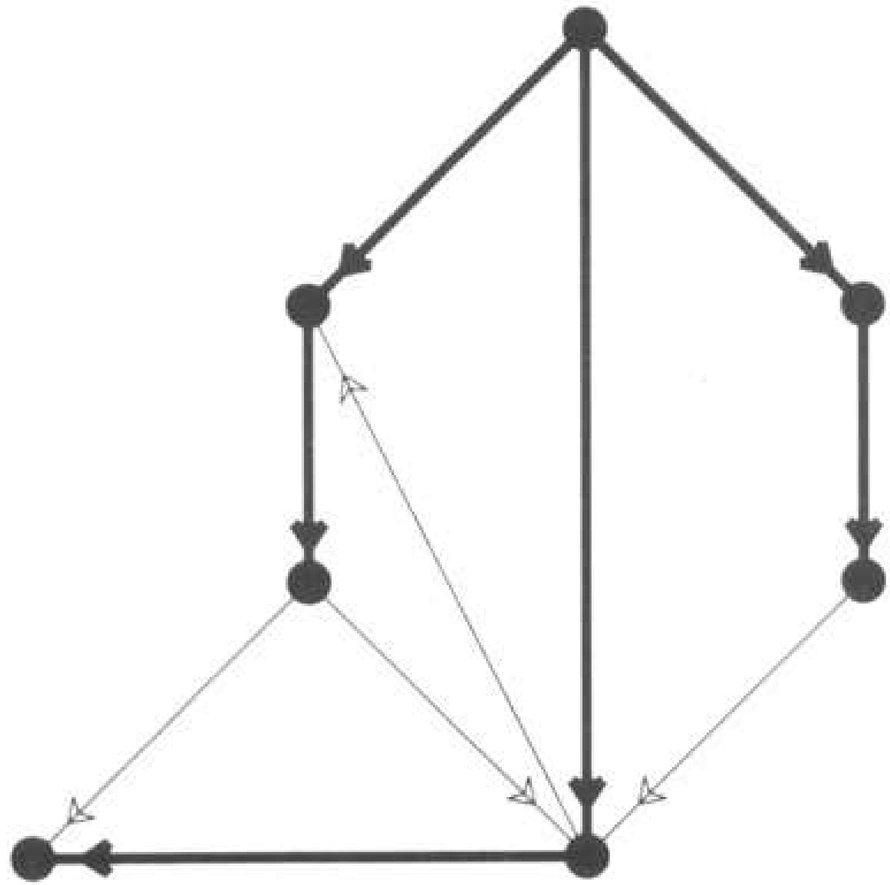


Figure 7.12 A BFS tree for a directed graph.

Source: Manber 1989

Breadth-First Search (cont.)

Algorithm Breadth_First_Search(G, v);
begin
  mark v;
  put v in a queue;
  while the queue is not empty **do**
    remove vertex w from the queue;
    perform preWORK on w;
    for all edges (w, x) with x unmarked **do**
      mark x;
      add (w, x) to the BFS tree T;
      put x in the queue
end

Breadth-First Search (cont.)

Lemma 7.5
If an edge (u, w) belongs to a BFS tree such that u is a parent of w, then u has the minimal BFS number among vertices with edges leading to w.

Lemma 7.6
For each vertex w, the path from the root to w in T is a shortest path from the root to w in G.

Lemma 7.7
If an edge (v, w) in E does not belong to T and w is on a larger level, then the level numbers of w and v differ by at most 1.

Breadth-First Search (cont.)

Algorithm Simple_BFS(G, v);
begin
    put v in a queue;
    while the queue is not empty **do**
        remove vertex w from the queue;
        if w is unmarked **then**
            mark w;
            perform preWORK on w;
            for all edges (w, x) with x unmarked **do**
                put x in the queue
end

Breadth-First Search (cont.)

Algorithm Simple_Nonrecursive_DFS(G, v);
begin
    push v to $Stack$;
    while $Stack$ is not empty **do**
        pop vertex w from $Stack$;
        if w is unmarked **then**
            mark w;
            perform preWORK on w;
            for all edges (w, x) with x unmarked **do**
                push x to $Stack$
end

Topological Sorting

The Problem Given a directed acyclic graph $G = (V, E)$ with n vertices, label the vertices from 1 to n such that, if v is labeled k, then all vertices that can be reached from v by a directed path are labeled with labels $> k$.

Lemma 7.8
A directed acyclic graph always contains a vertex with indegree 0.

Topological Sorting (cont.)

Algorithm Topological_Sorting(G);
    initialize $v.indegree$ for all vertices; /* by DFS */
    G_label := 0;
    for i := 1 to n **do**
        if $v_i.indegree = 0$ **then** put v_i in $Queue$;
    repeat
        remove vertex v from $Queue$;
        G_label := $G\_label + 1$;
        $v.label$:= G_label;
        for all edges (v, w) **do**
            $w.indegree$:= $w.indegree - 1$;
            if $w.indegree = 0$ **then** put w in $Queue$
    until $Queue$ is empty

Single-Source Shortest Paths

The Problem Given a directed graph $G = (V, E)$ and a vertex v, find shortest paths from v to all other vertices of G.

Shorted Paths: The Acyclic Case

Algorithm Acyclic_Shortest_Paths(G, v, n);
{After performing a topological sort on G, $\ldots$}
begin
  let z be the vertex labeled n;
  if $z \neq v$ **then**
    $Acyclic\_Shortest\_Paths(G - z, v, n - 1)$;
    for all w such that $(w, z) \in E$ **do**
      if $w.SP + length(w, z) < z.SP$ **then**
        $z.SP$:= $w.SP + length(w, z)$
  else $v.SP$:= 0
end

The Acyclic Case (cont.)

Algorithm Imp_Acyclic_Shortest_Paths(G, v);

    for all vertices w **do** $w.SP$:= ∞;

    initialize $v.indegree$ for all vertices;

    for i := 1 to n **do**

        if $v_i.indegree = 0$ **then** put v_i in $Queue$;

    $v.SP$:= 0;

    repeat

        remove vertex w from $Queue$;

        for all edges (w, z) **do**

            if $w.SP + length(w, z) < z.SP$ **then**

                $z.SP$:= $w.SP + length(w, z)$;

            $z.indegree$:= $z.indegree - 1$;

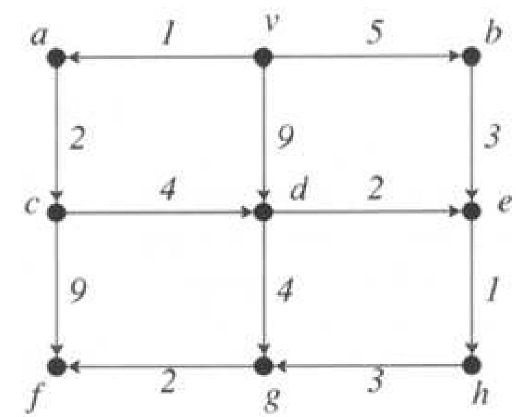
            if $z.indegree = 0$ **then** put z in $Queue$

    until $Queue$ is empty

Shortest Paths: The General Case

Algorithm Single_Source_Shortest_Paths(G, v);
begin
  for all vertices w **do**
    $w.mark$:= $false$;
    $w.SP$:= ∞;
  $v.SP$:= 0;
  while there exists an unmarked vertex **do**
    let w be an unmarked vertex s.t. $w.SP$ is minimal;
    $w.mark$:= $true$;
    for all edges (w, z) such that z is unmarked **do**
      if $w.SP + length(w, z) < z.SP$ **then**
        $z.SP$:= $w.SP + length(w, z)$
end

The General Case (cont.)



	v	a	b	c	d	e	f	g	h
a	0	1	5	∞	9	∞	∞	∞	∞
c	0	1	5	3	9	∞	∞	∞	∞
b	0	1	5	3	7	∞	12	∞	∞
d	0	1	5	3	7	8	12	∞	∞
e	0	1	5	3	7	8	12	11	∞
h	0	1	5	3	7	8	12	11	9
g	0	1	5	3	7	8	12	11	9
f	0	1	5	3	7	8	12	11	9

Figure 7.18 An example of the single-source shortest-paths algorithm.

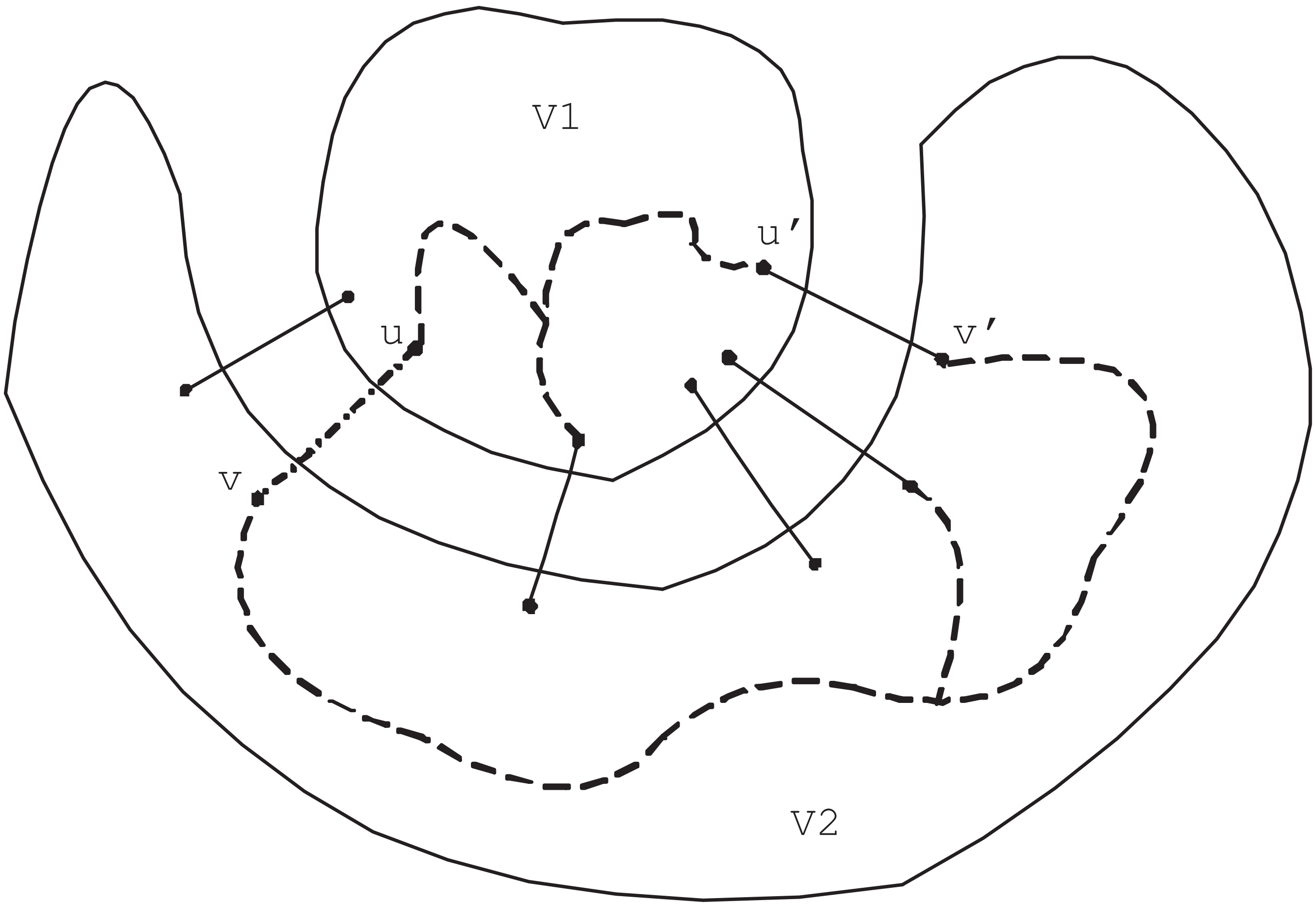
Source: Manber 1989

Minimum-Weight Spanning Trees

The Problem Given an undirected connected weighted graph $G = (V, E)$, find a spanning tree T of G of minimum weight.

Theorem
Let V_1 and V_2 be a partition of V and $E(V_1, V_2)$ be the set of edges connecting nodes in V_1 to nodes in V_2. The edge with the minimum weight in $E(V_1, V_2)$ must be in the minimum-cost spanning tree of G.

Minimum-Weight Spanning Trees (cont.)



- If $cost(u,v)$ is the smallest among $E(V_1, V_2)$, then $\{u,v\}$ must be in the minimum spanning tree.

Minimum-Weight Spanning Trees (cont.)

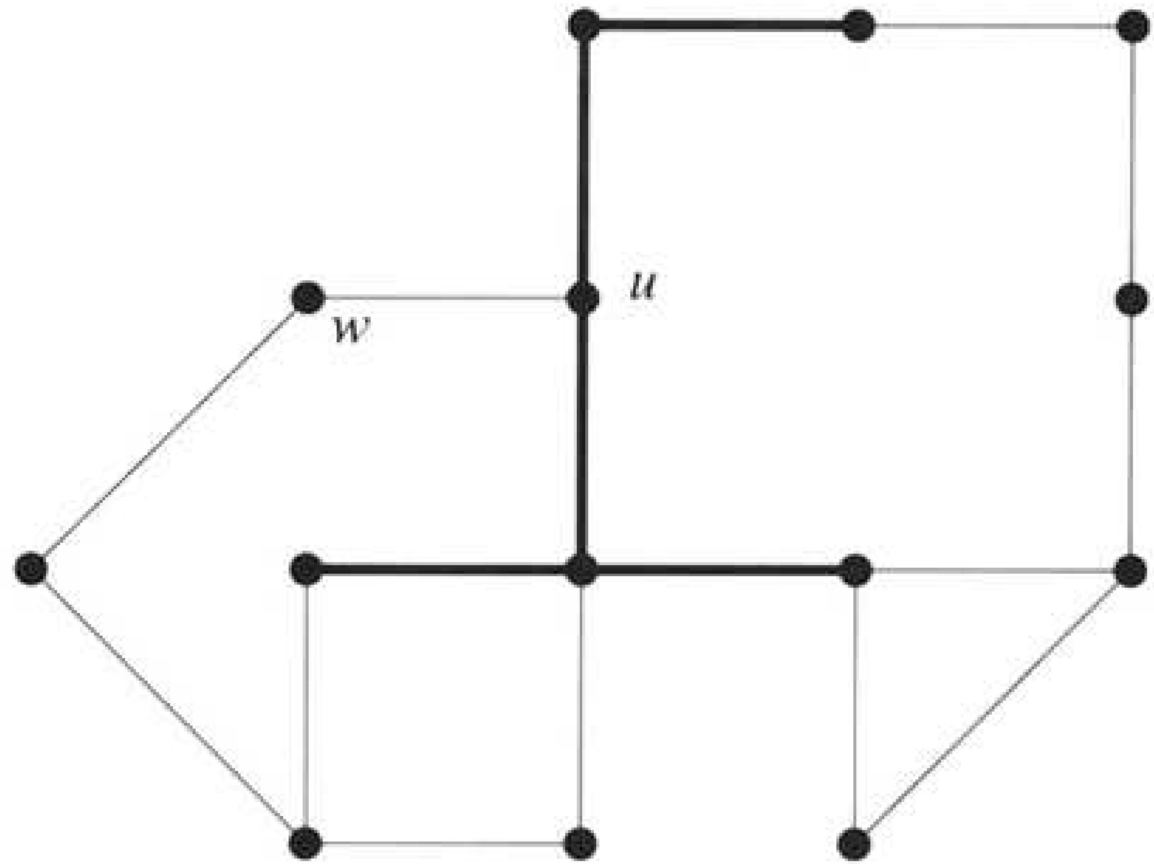


Figure 7.19 Finding the next edge of the MCST.

Source: Manber 1989

Minimum-Weight Spanning Trees (cont.)

Algorithm MST(G);
begin
  initially T is the empty set;
  for all vertices w **do**
    $w.mark$:= $false$; $w.cost$:= ∞;
  let (x, y) be a minimum cost edge in G;
  $x.mark$:= $true$;
  for all edges (x, z) **do**
    $z.edge$:= (x, z); $z.cost$:= $cost(x, z)$;

Minimum-Weight Spanning Trees (cont.)

while there exists an unmarked vertex **do**
  let w be an unmarked vertex with minimal $w.cost$;
  if $w.cost = \infty$ **then**
    print "G is not connected"; halt
  else
    $w.mark$:= $true$;
    add $w.edge$ to T;
    for all edges (w, z) **do**
      if not $z.mark$ **then**
        if $cost(w, z) < z.cost$ **then**
          $z.edge$:= (w, z); $z.cost$:= $cost(w, z)$
end

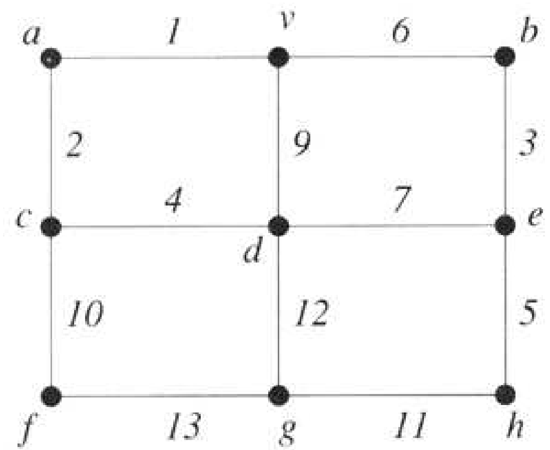
Minimum-Weight Spanning Trees (cont.)

Algorithm Another_MST(G);
begin
  initially T is the empty set;
  for all vertices w **do**
    $w.mark$:= $false$; $w.cost$:= ∞;
  $x.mark$:= $true$; /* x is an arbitrary vertex */
  for all edges (x, z) **do**
    $z.edge$:= (x, z); $z.cost$:= $cost(x, z)$;

Minimum-Weight Spanning Trees (cont.)

while there exists an unmarked vertex **do**
  let w be an unmarked vertex with minimal $w.cost$;
  if $w.cost = \infty$ **then**
    print "G is not connected"; halt
  else
    $w.mark$:= $true$;
    add $w.edge$ to T;
    for all edges (w, z) **do**
      if not $z.mark$ **then**
        if $cost(w, z) < z.cost$ **then**
          $z.edge$:= (w, z);
          $z.cost$:= $cost(w, z)$
end

Minimum-Weight Spanning Trees (cont.)



	v	a	b	c	d	e	f	g	h
v	-	$v(1)$	$v(6)$	∞	$v(9)$	∞	∞	∞	∞
a	-	-	$v(6)$	$a(2)$	$v(9)$	∞	∞	∞	∞
c	-	-	$v(6)$	-	$c(4)$	∞	$c(10)$	∞	∞
d	-	-	$v(6)$	-	-	$d(7)$	$c(10)$	$d(12)$	∞
b	-	-	-	-	-	$b(3)$	$c(10)$	$d(12)$	∞
e	-	-	-	-	-	-	$c(10)$	$d(12)$	$e(5)$
h	-	-	-	-	-	-	$c(10)$	$h(11)$	-
f	-	-	-	-	-	-	-	$h(11)$	-
g	-	-	-	-	-	-	-	-	-

Figure 7.21 An example of the minimum-cost spanning-tree algorithm.

Source: Manber 1989

All Shortest Paths

The Problem Given a weighted graph $G = (V, E)$ (directed or undirected) with nonnegative weights, find the minimum-length paths between all pairs of vertices.

Algorithm All_Pairs_Shortest_Paths(W);
begin
  {initialization omitted}
  for m := 1 to n **do** {the induction sequence}
    for x := 1 to n **do**
      for y := 1 to n **do**
        if $W[x,m] + W[m,y] < W[x,y]$ **then**
          $W[x,y]$:= $W[x,m] + W[m,y]$
end

Transitive Closure

The Problem Given a directed graph $G = (V, E)$, find its transitive closure.

Algorithm Transitive_Closure(A);
begin
  {initialization omitted}
  for m := 1 to n **do**
    for x := 1 to n **do**
      for y := 1 to n **do**
        if $A[x, m]$ and $A[m, y]$ **then**
          $A[x, y]$:= $true$
end

Transitive Closure (cont.)

```
Algorithm Improved_Transitive_Closure(A);
begin
    {initialization omitted}
    for m := 1 to n do
        for x := 1 to n do
            if A[x, m] then
                for y := 1 to n do
                    if A[m, y] then
                        A[x, y] := true
end
```

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* 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.)

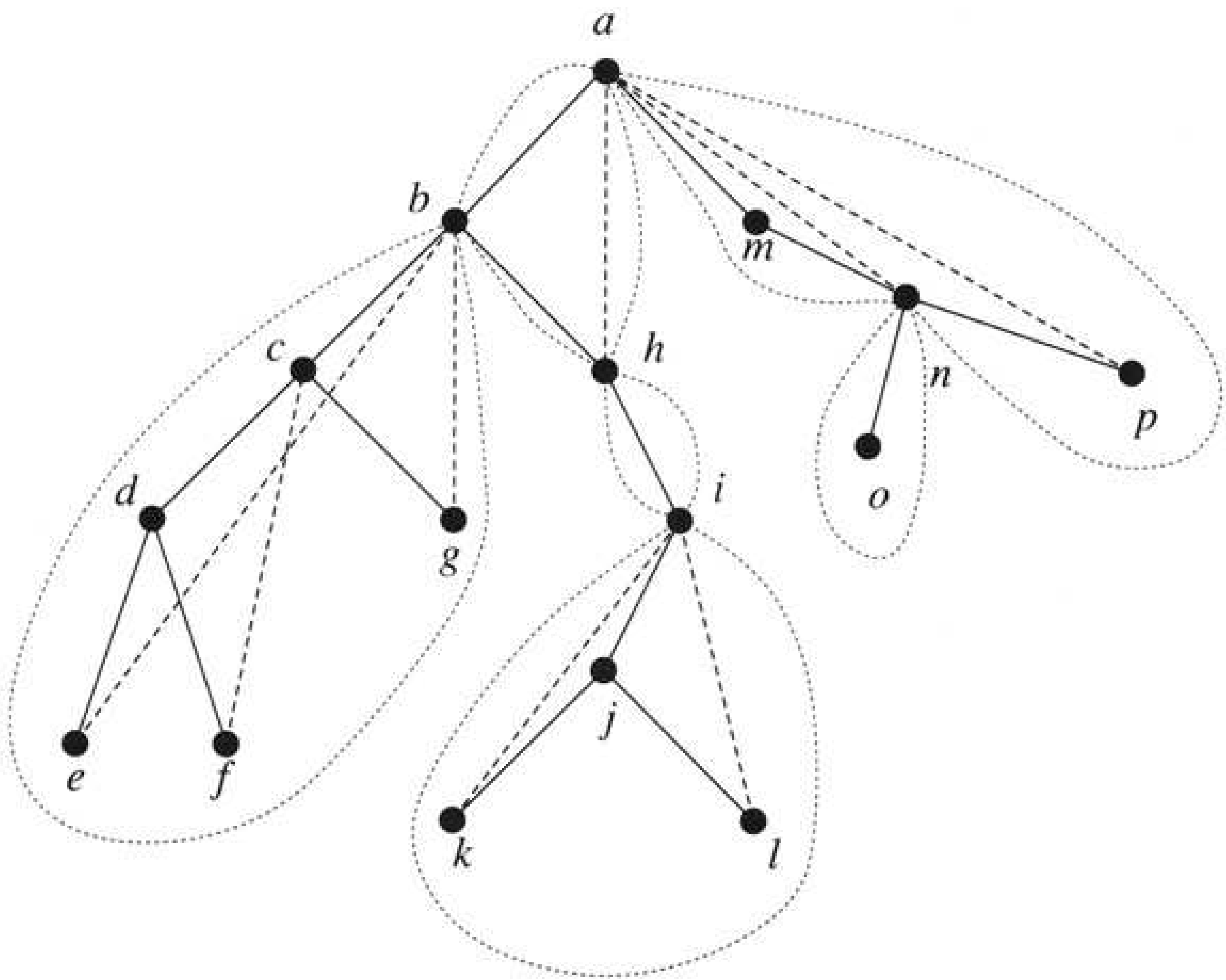


Figure 7.25 The structure of a nonbiconnected graph.

Source: Manber 1989

Biconnected Components (cont.)

Lemma 7.9
Two distinct edges e and f belong to the same biconnected component if and only if there is a cycle containing both of them.

Lemma 7.10
Each edge belongs to exactly one biconnected component.

Biconnected Components (cont.)

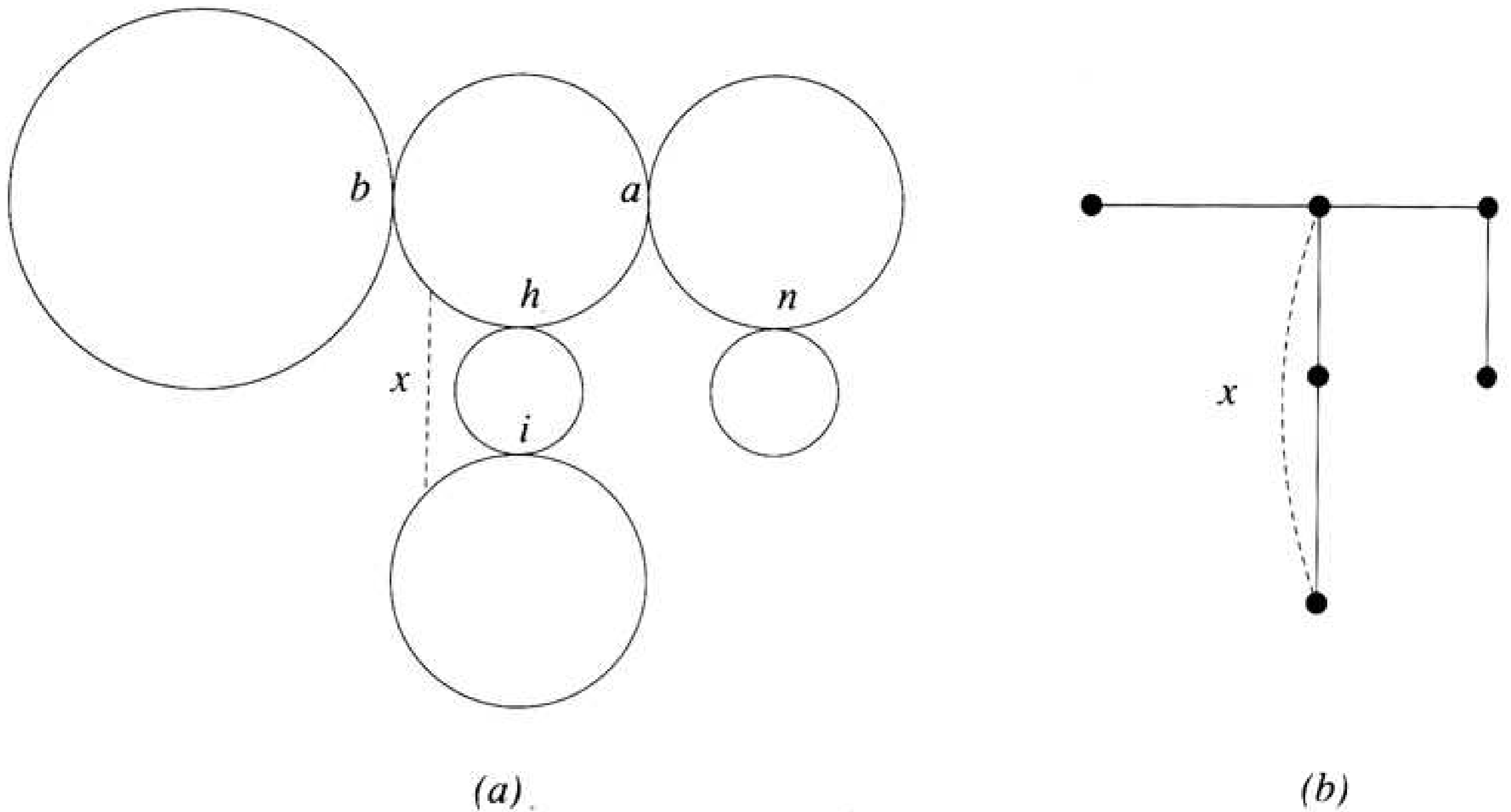


Figure 7.26 An edge that connects two different biconnected components. (a) The components corresponding to the graph of Fig. 7.25 with the articulation points indicated. (b) The biconnected component tree.

Source: Manber 1989

Biconnected Components (cont.)

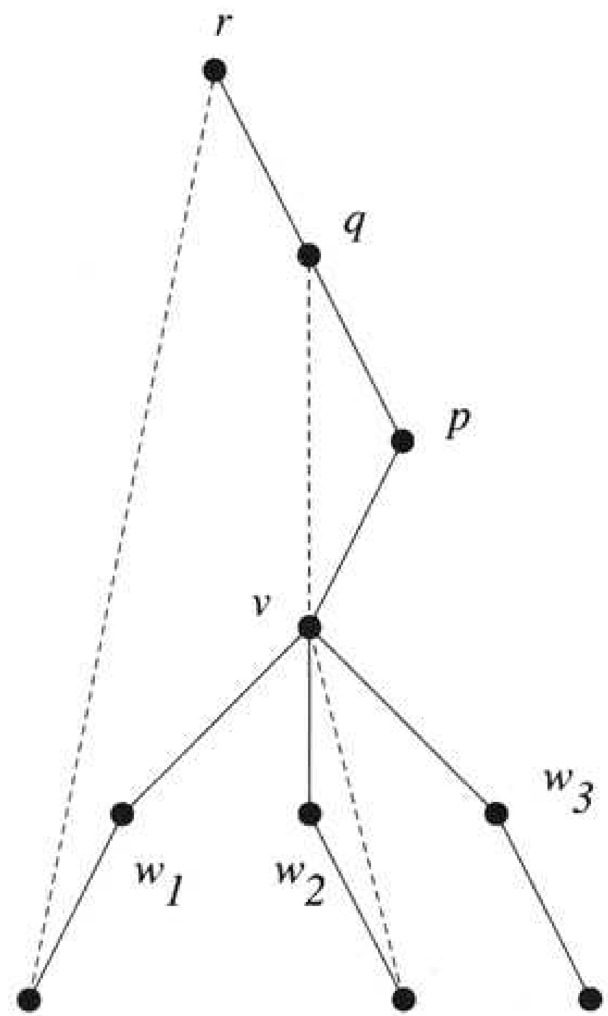


Figure 7.27 Computing the *High* values.

Source: Manber 1989

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 $Stack$;
  $v.high$:= $v.DFS\_Number$;

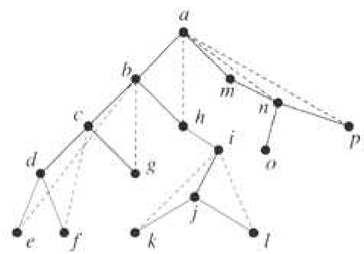
Biconnected Components (cont.)

```
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 ≤ 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)
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 ≤ 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	-	-	-	-	-	-	-	-	-	-	-
d	16	15	14	15	15	-	-	-	-	-	-	-	-	-	-	-
f	16	15	14	15	15	14	-	-	-	-	-	-	-	-	-	-
d	16	15	14	15	15	14	-	-	-	-	-	-	-	-	-	-
c	16	15	15	15	15	14	-	-	-	-	-	-	-	-	-	-
g	16	15	15	15	15	14	15	-	-	-	-	-	-	-	-	-
c	16	15	15	15	15	14	15	-	-	-	-	-	-	-	-	-
b	16	15	15	15	14	13	15	-	-	-	-	-	-	-	-	-
h	16	15	15	15	15	14	15	16	-	-	-	-	-	-	-	-
i	16	15	15	15	15	14	15	16	8	-	-	-	-	-	-	-
j	16	15	15	15	15	14	15	16	8	7	-	-	-	-	-	-
k	16	15	15	15	15	14	15	16	8	7	8	-	-	-	-	-
j	16	15	15	15	15	14	15	16	8	8	8	-	-	-	-	-
l	16	15	15	15	15	14	15	16	8	8	8	8	-	-	-	-
j	16	15	15	15	15	14	15	16	8	8	8	8	-	-	-	-
i	16	15	15	15	15	14	15	16	8	8	8	8	-	-	-	-
h	16	15	15	15	15	14	15	16	8	8	8	8	-	-	-	-
b	16	16	15	15	15	14	15	16	8	8	8	8	-	-	-	-
a	16	16	15	15	15	14	15	16	8	8	8	8	-	-	-	-
m	16	16	15	15	15	14	15	16	8	8	8	8	4	-	-	-
n	16	16	15	15	15	14	15	16	8	8	8	8	4	16	-	-
o	16	16	15	15	15	14	15	16	8	8	8	8	4	16	2	-
n	16	16	15	15	15	14	15	16	8	8	8	8	4	16	2	-
p	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
m	16	16	15	15	15	14	15	16	8	8	8	8	16	16	2	16
a	16	16	15	15	15	14	15	16	8	8	8	8	16	16	2	16

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

 Source: Manber 1989

Even-Length Cycles

The 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.)

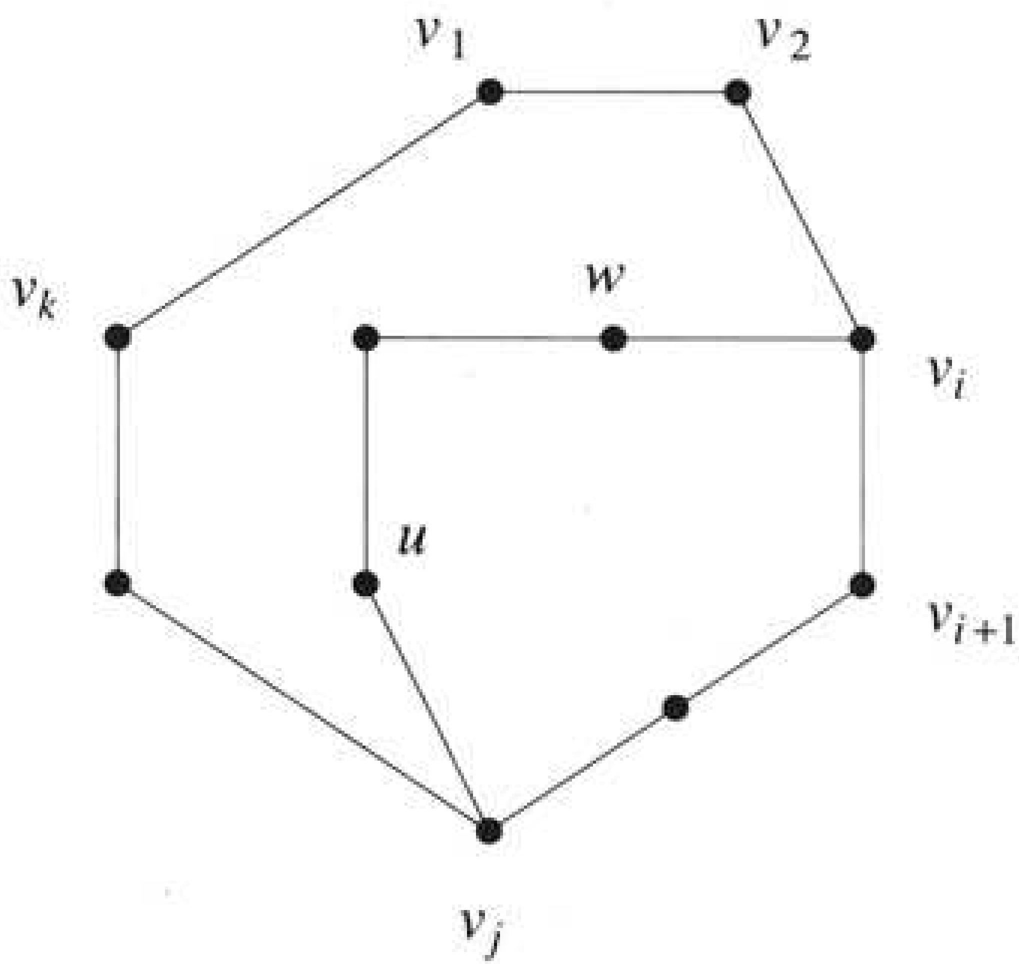


Figure 7.35 Finding an even-length cycle.

Source: Manber 1989

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* 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.)

Lemma 7.11
Two distinct vertices belong to the same strongly connected component if and only if there is a circuit containing both of them.

Lemma 7.12
Each vertex belongs to exactly one strongly connected component.

Strongly Connected Components (cont.)

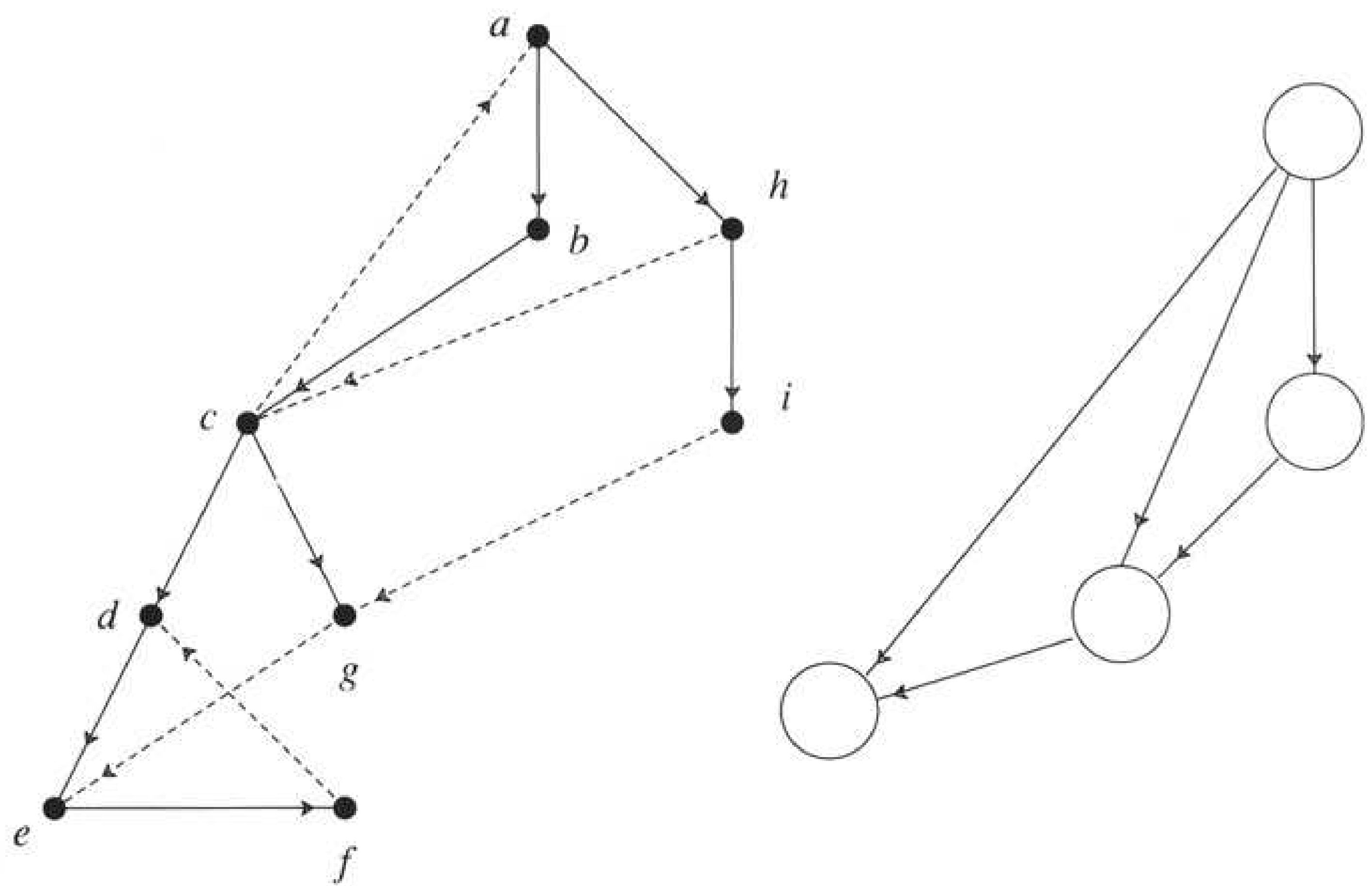


Figure 7.30 A directed graph and its strongly connected component graph.

Source: Manber 1989

Strongly Connected Components (cont.)

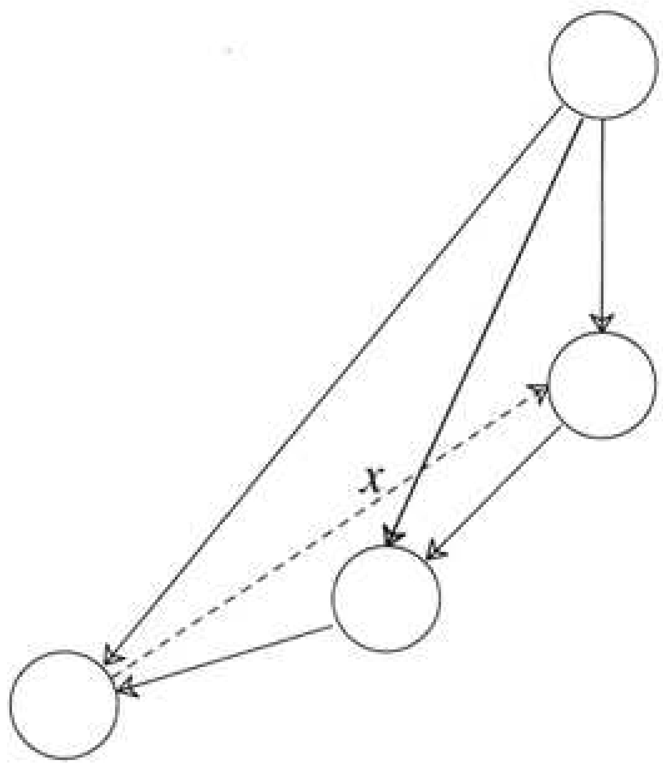


Figure 7.31 Adding an edge connecting two different strongly connected components.

Source: Manber 1989

Strongly Connected Components (cont.)

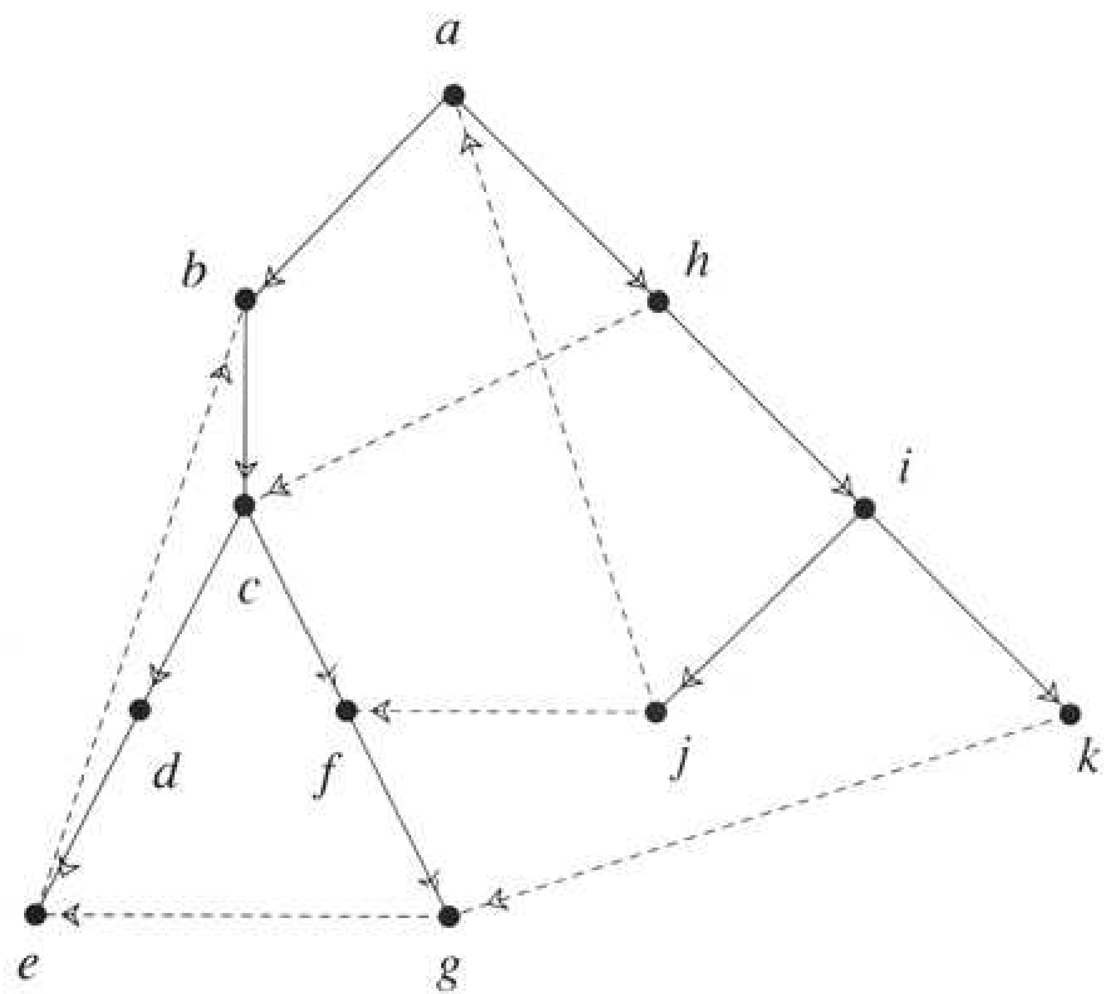


Figure 7.32 The effect of cross edges.

Source: Manber 1989

Strongly Connected Components (cont.)

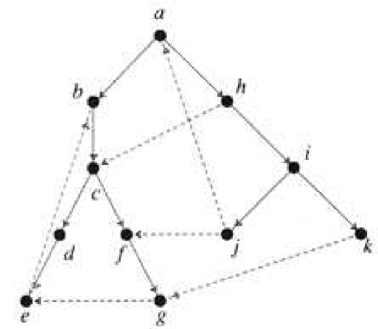
Algorithm Strongly_Connected_Components(G, n);
begin
    for every vertex v of G **do**
        $v.DFS\_Number$:= 0;
        $v.component$:= 0;
    $Current\_Component$:= 0; DFS_N := n;
    while $v.DFS\_Number = 0$ for some v **do**
        $SCC(v)$
end

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)
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.)



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

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

Source: Manber 1989

Odd-Length Cycles

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

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.

Network Flows (cont.)

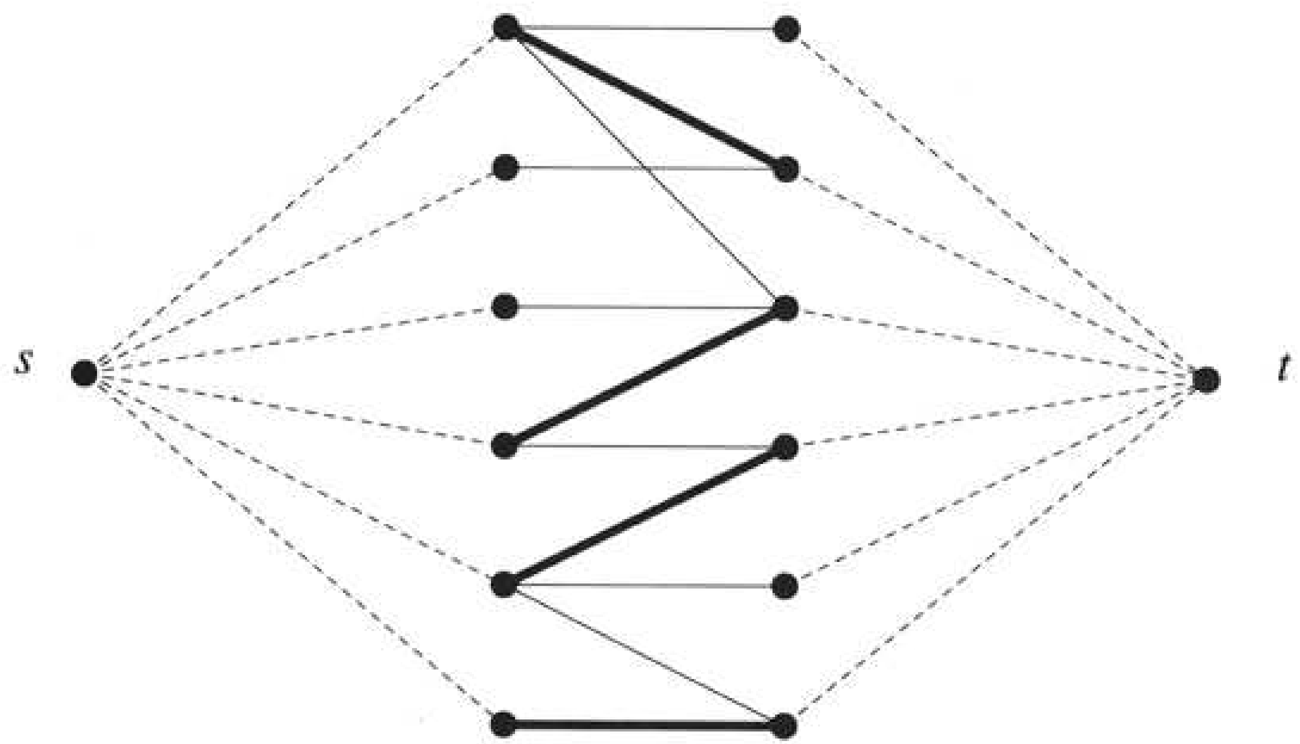


Figure 7.39 Reducing bipartite matching to network flow (the directions of all the edges are from left to right).

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 diretion; 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.)

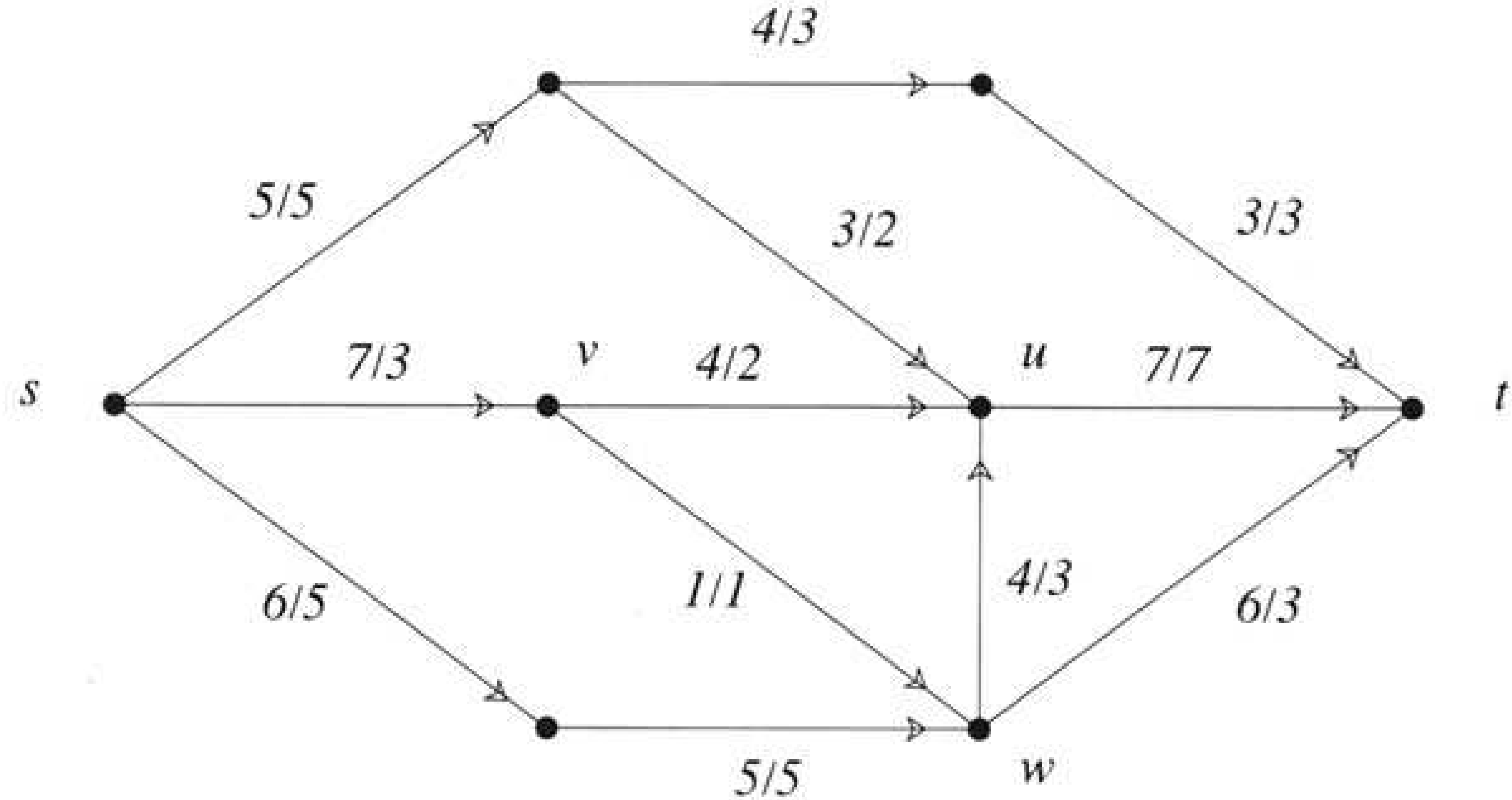


Figure 7.40 An example of a network with a (nonmaximum) flow.

Source: Manber 1989

Augmenting Paths (cont.)

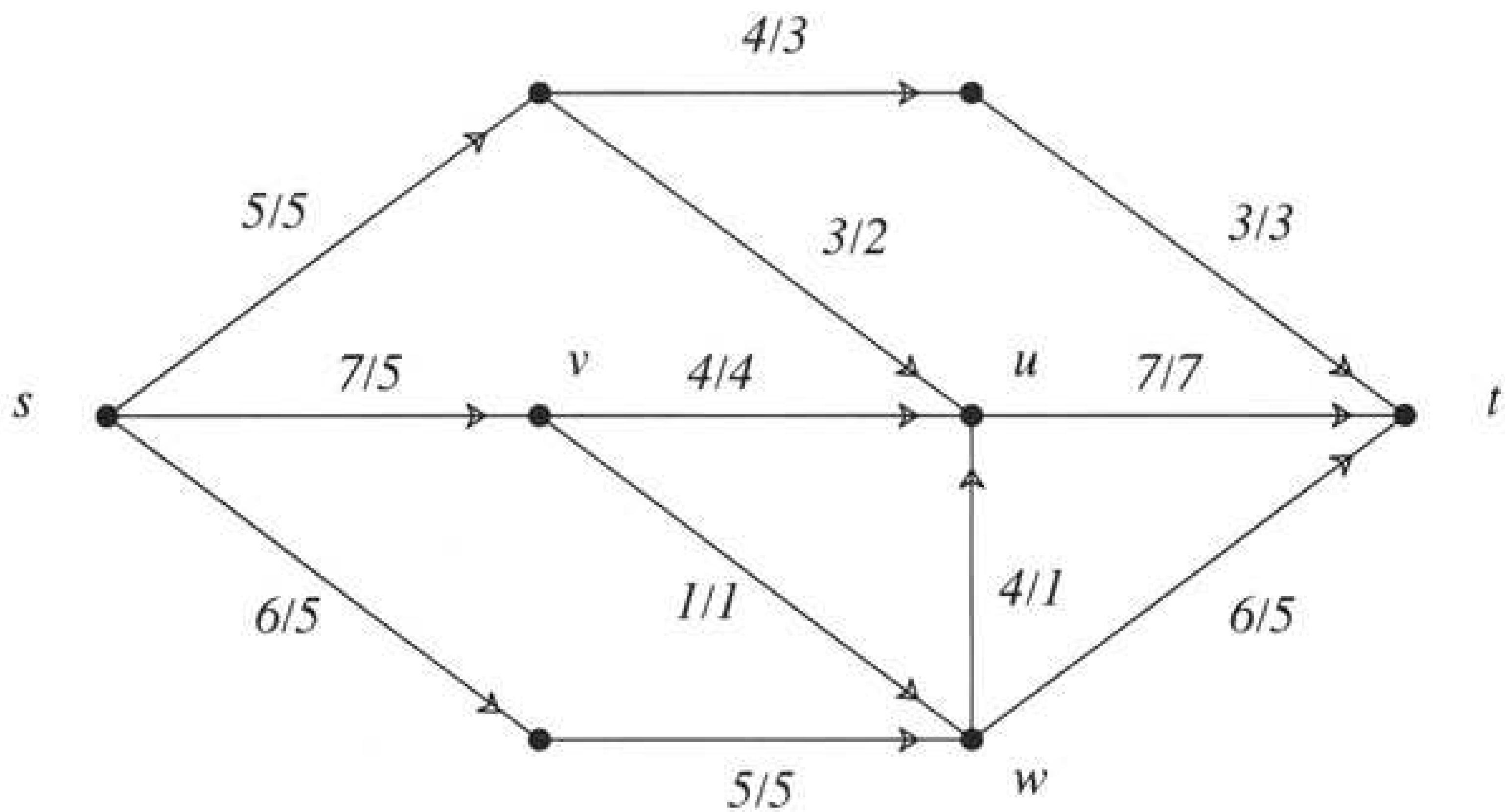


Figure 7.41 The result of augmenting the flow of Fig. 7.40.

Source: Manber 1989

Properties of Network Flows

The Augmenting-Path Theorem 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 $t \in B$.

Max-Flow Min-Cut Theorem The value of a maximum flow in a network is equal to the minimum capacity of a cut.

Properties of Network Flows (cont.)

The Integral-Flow Theorem 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.)

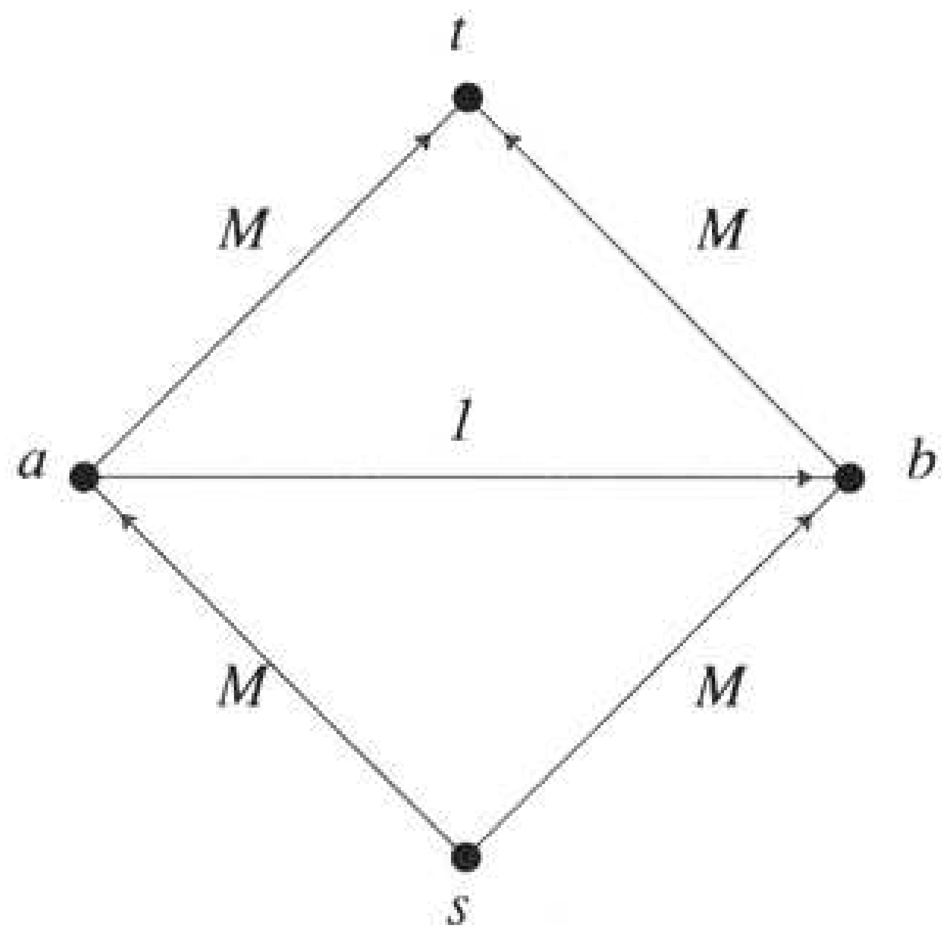


Figure 7.42 A bad example of network flow.

Source: Manber 1989