

Basic Graph Algorithms

(Based on [Manber 1989])

Yih-Kuen Tsay

Department of Information Management
National Taiwan University

The Königsberg Bridges Problem

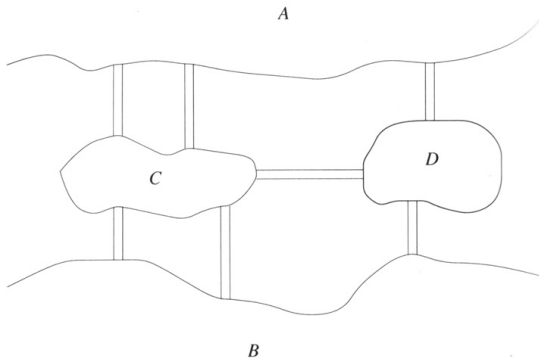


Figure 7.1 The Königsberg bridges problem.

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.)

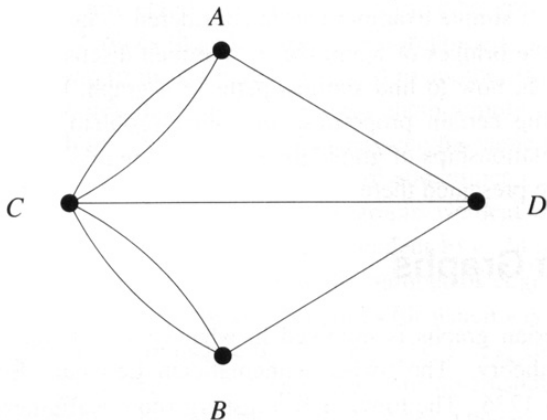


Figure 7.2 The graph corresponding to the Königsberg bridges problem.

Source: [Manber 1989].



- 🌐 A graph consists of a set of **vertices** (or nodes) and a set of **edges** (or links, each normally connecting two vertices).
- 🌐 A graph 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.

Graphs (cont.)



- 🌐 Undirected vs. Directed Graph
- 🌐 Simple Graph vs. Multigraph
- 🌐 Path, Simple Path, Trail
- 🌐 Circuit, Cycle
- 🌐 Degree, In-Degree, Out-Degree
- 🌐 Connected Graph, Connected Components
- 🌐 Tree, Forest
- 🌐 Subgraph, Induced Subgraph
- 🌐 Spanning Tree, Spanning Forest
- 🌐 Weighted Graph

Modeling with Graphs



Reachability

-  Finding program errors
-  Solving sliding tile puzzles

Shortest Paths

-  Finding the fastest route to a place
-  Routing messages in networks

Graph Coloring

-  Coloring maps
-  Scheduling classes

Eulerian Graphs

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

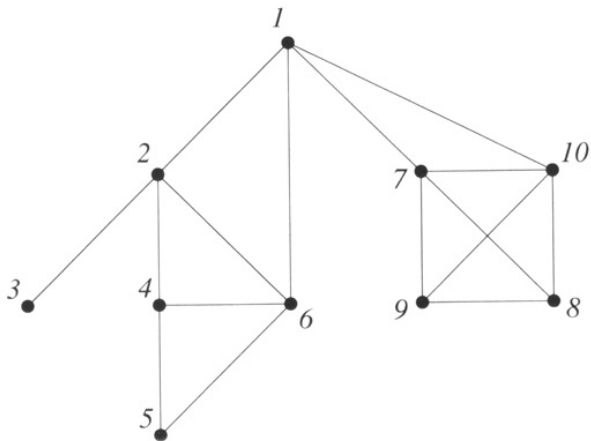


Figure 7.4 A DFS for an undirected graph.

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

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

Time complexity:

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

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

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

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
```

Time complexity: $O(|E|)$ (assuming the input graph is connected).

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.)

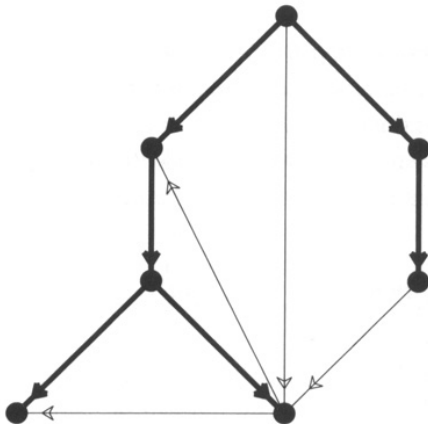


Figure 7.9 A DFS tree for a directed graph.

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

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

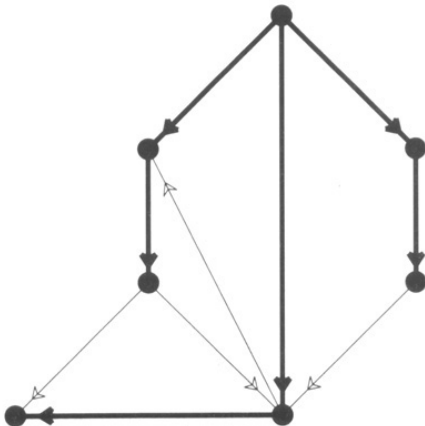


Figure 7.12 A BFS tree for a directed graph.

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 *Queue*;

while *Queue* is not empty **do**

remove vertex w from *Queue*;

if w is unmarked **then**

mark w ;

perform **preWORK** on w ;

for all edges (w, x) with x unmarked **do**

put x in *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

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

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);
{Initially, $w.SP = \infty$, for every node w .}
{A topological sort has been performed on G, \dots }

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 $w.SP + \text{length}(w, z) < z.SP$ **then**
 $z.SP := w.SP + \text{length}(w, z)$
 else $v.SP := 0$
end

The Acyclic Case (cont.)

```
Algorithm Imp_Acyclic_Shortest_Paths( $G, v$ );  
  for all vertices  $w$  do  $w.SP := \infty$ ;  
  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( $G, v$ );  
// Dijkstra's algorithm  
begin  
  for all vertices  $w$  do  
     $w.mark := false$ ;  
     $w.SP := \infty$ ;  
   $v.SP := 0$ ;  
  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
```


Shortest Paths: The General Case

```
Algorithm Single_Source_Shortest_Paths( $G, v$ );  
// Dijkstra's algorithm  
begin  
  for all vertices  $w$  do  
     $w.mark := false$ ;  
     $w.SP := \infty$ ;  
 $v.SP := 0$ ;  
  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
```

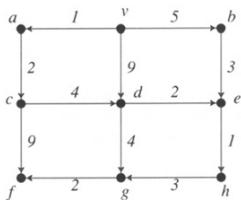
Time complexity:

Shortest Paths: The General Case

```
Algorithm Single_Source_Shortest_Paths( $G, v$ );  
// Dijkstra's algorithm  
begin  
  for all vertices  $w$  do  
     $w.mark := false$ ;  
     $w.SP := \infty$ ;  
 $v.SP := 0$ ;  
  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
```

Time complexity: $O((|E| + |V|) \log |V|)$ (using a min heap).

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.

Minimum-Weight Spanning Trees

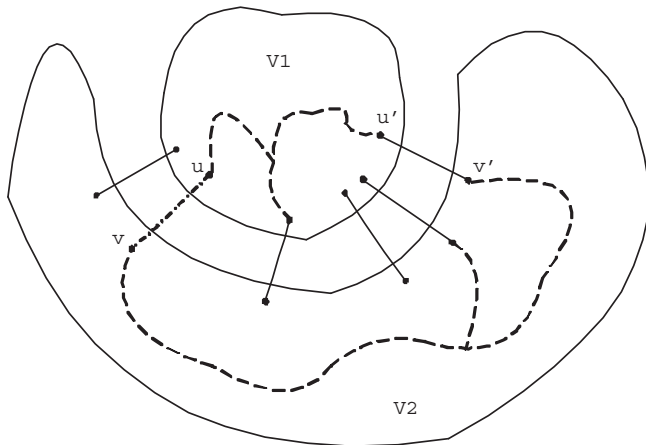
Problem

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.)

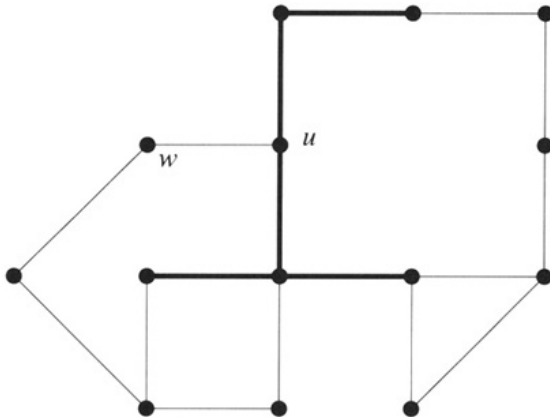


Figure 7.19 Finding the next edge of the MCST.

Source: [Manber 1989].

Minimum-Weight Spanning Trees (cont.)

Algorithm MST(G);

// A variant of Prim's algorithm

begin

initially T is the empty set;

for all vertices w **do**

$w.mark := false$; $w.cost := \infty$;

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);

// Prim's algorithm

begin

initially T is the empty set;

for all vertices w **do**

$w.mark := false$; $w.cost := \infty$;

$x.mark := true$; /* 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.)

```
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
  end
```

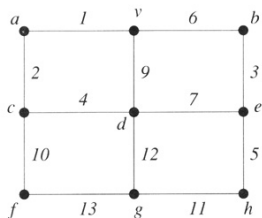
Time complexity:

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
```

Time complexity: same as that of Dijkstra's algorithm.

Minimum-Weight Spanning Trees (cont.)



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

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

All Shortest Paths

Problem

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

Floyd's Algorithm

Algorithm All_Pairs_Shortest_Paths(W);

begin

 {initialization}

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

for $j := 1$ to n **do**

if $(i, j) \in E$ **then** $W[i, j] := \text{length}(i, j)$

else $W[i, j] := \infty$;

for $i := 1$ to n **do** $W[i, i] := 0$;

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

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