Basic Graph Algorithms

Yu-Fang Chen (陳郁方)

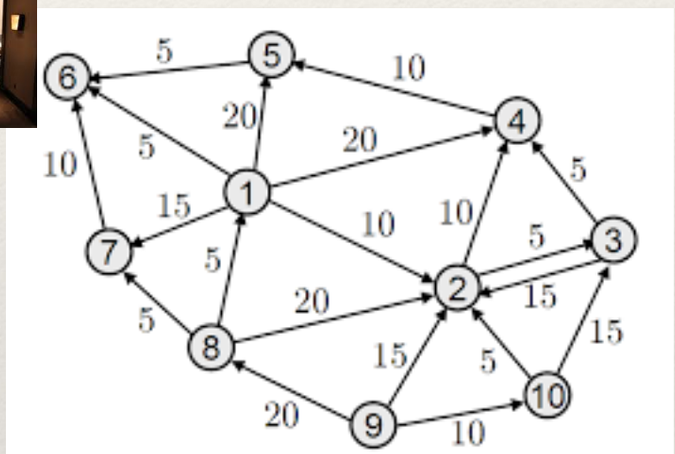
Department of Information Management
National Taiwan University

Graph Algorithm

- In the previous lectures, algorithms over lists/strings have been discussed.
- The relation between objects are relatively simple: ordering, overlapping…
- Today we will discuss algorithms over a more involved relation between objects: ***graph***

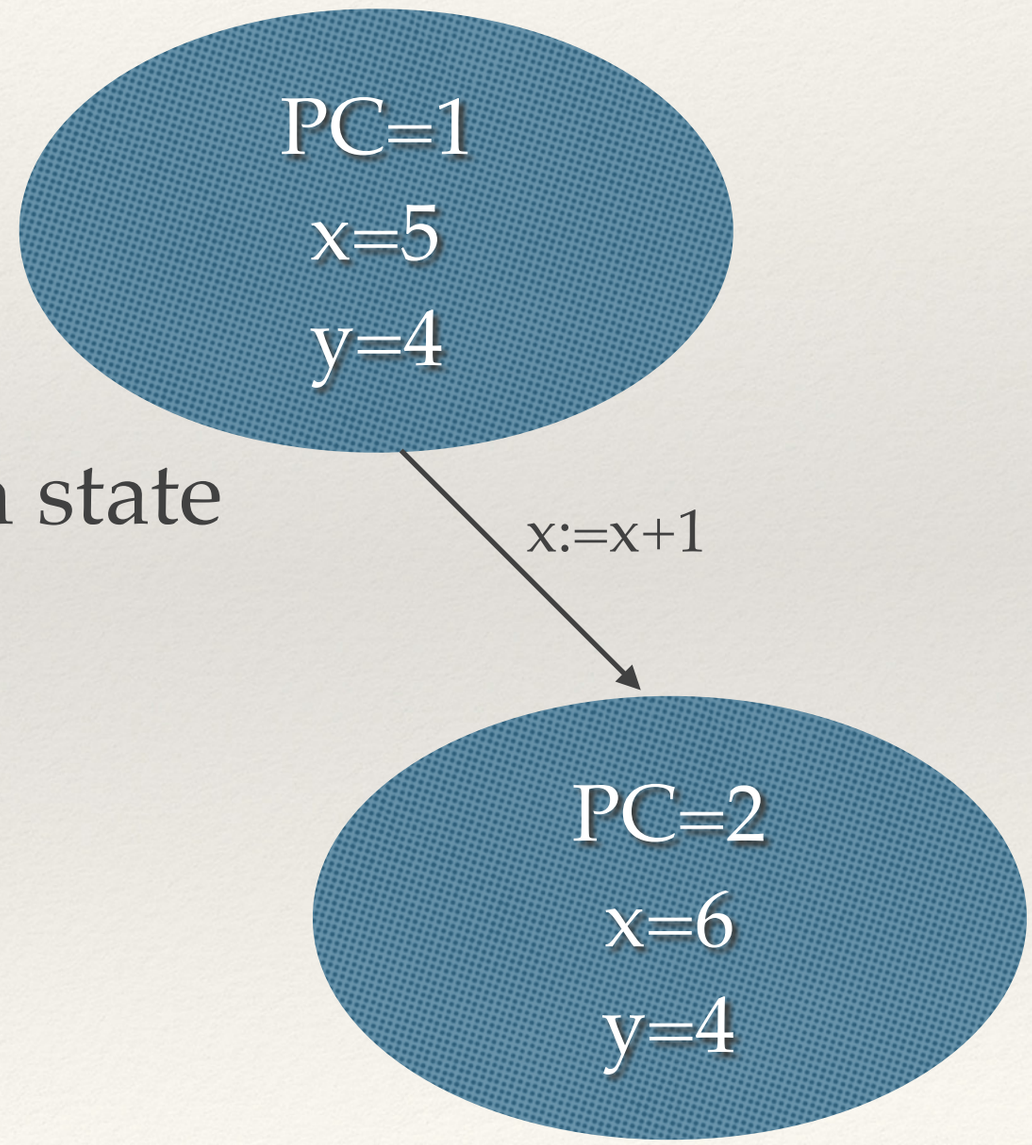
Examples of Modeling by Graphs

- Finding a good route to a restaurant in a city:
  - Edges: streets
  - Vertices: intersections
  - Directed edge to model one way
  - Expected time delay is associated to each edge or vertex

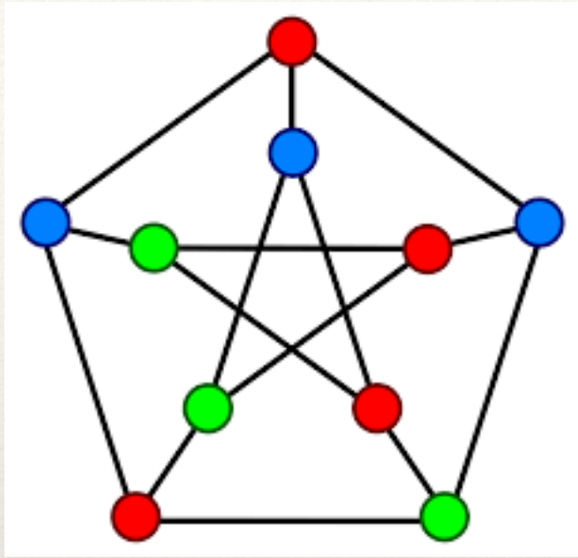


Examples of Modeling by Graphs (cont.)

- Program's Behavior
  - Vertices: program states
  - Edges: program statements
- We can ask if an error program state is reachable
- **We can do a <u>demo</u> :-)**



Examples of Modeling by Graphs (Cont.)



- Scheduling classes:
  - Vertices: classes
  - Undirected edges: two classes taught by the same instructor
  - Colors: time slots

The Königsberg Bridges Problem

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

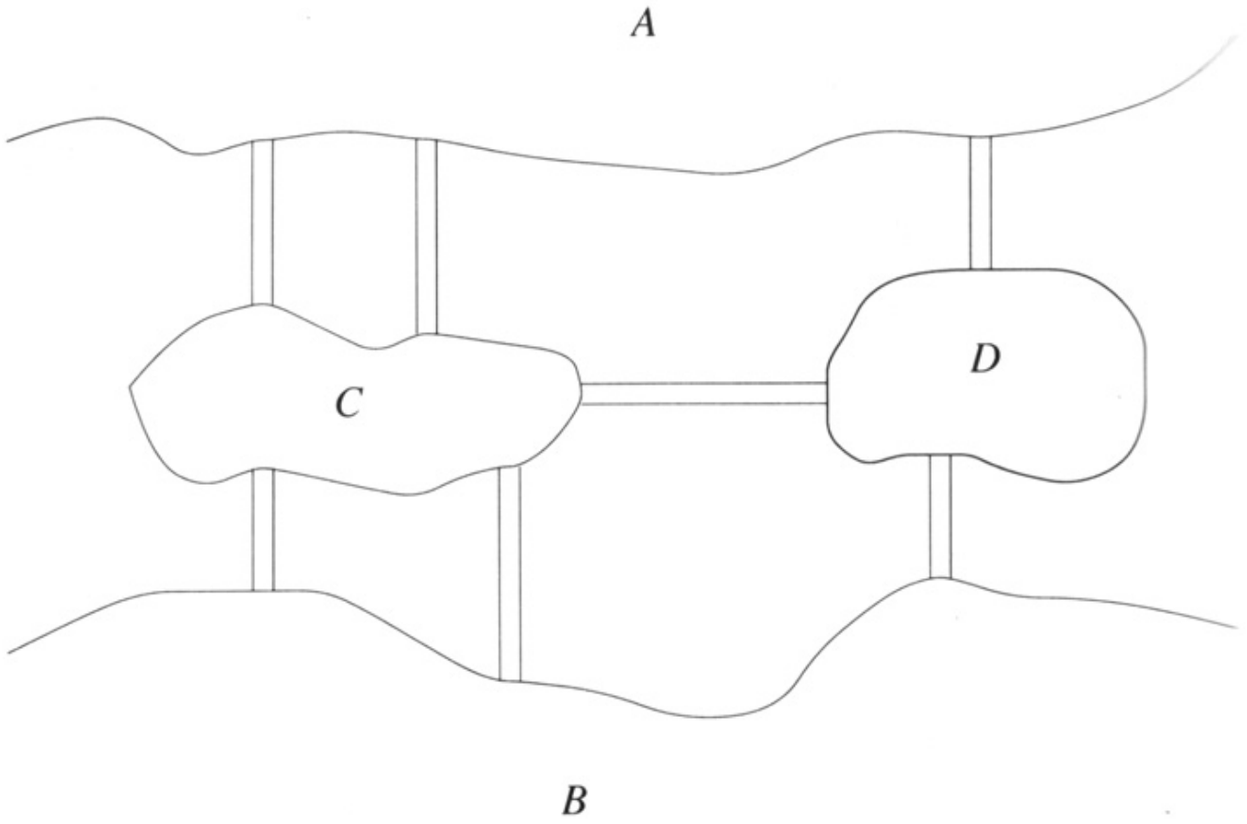


Figure 7.1 The Königsberg bridges problem.

Source: [Manber 1989]

The Königsberg Bridges Problem

The graph corresponding to the problem.

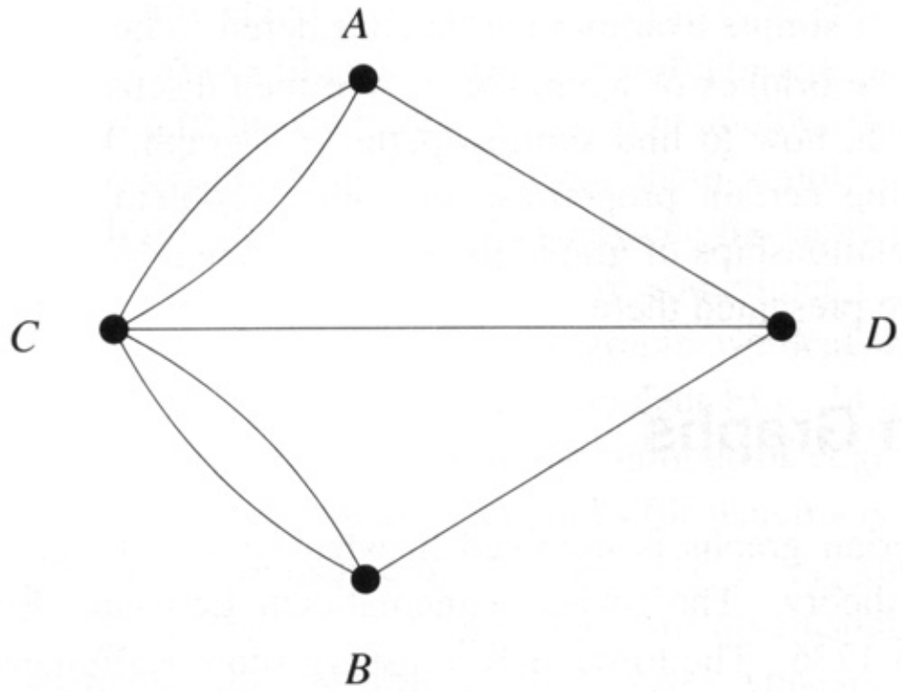


Figure 7.2 The graph corresponding to the Königsberg bridges problem.

Source: [Manber 1989]

Graphs

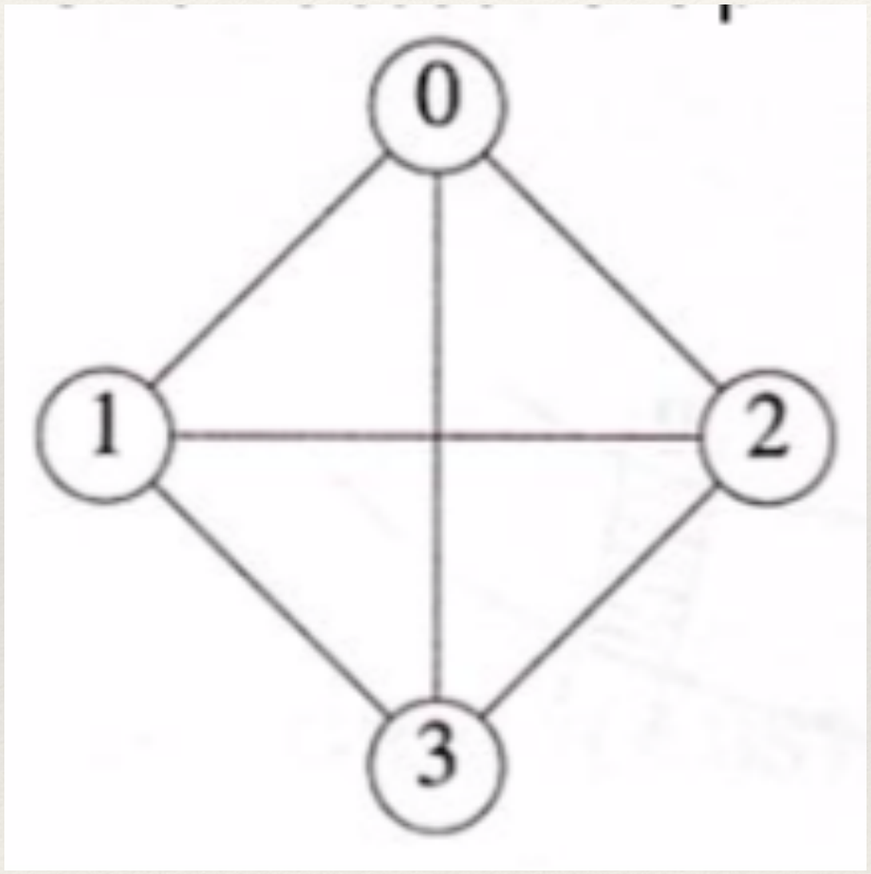
- A graph consists of a set of vertices (or nodes) and a set of edges (or links).
- A graph is commonly denoted as (V, E), where
  - V is the set of vertices, and
  - E is the set of edges

Terminology

- **Undirected Graphs:**
  - (v_0,v_1) and (v_1,v_0) represent the same edge.
  - An undirected graph with n vertices has at most $n(n-1)/2$ edges.
- **Directed Graphs:**
  - (v_0,v_1) and (v_1,v_0) represent different edges.
  - An undirected graph with n vertices has at most $n(n-1)$ edges.

Terminology (cont.)

Undirected Graph

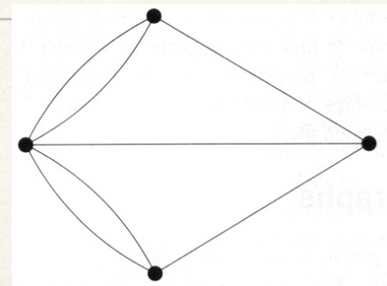


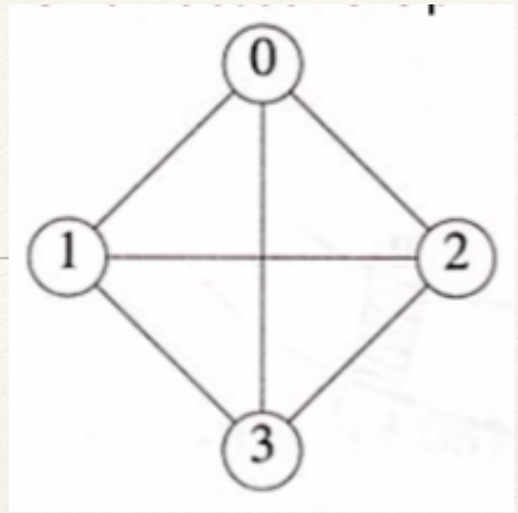
Are (0,1) and (1,0) the same edge?

Directed Graph

Are (0,1) and (1,0) the same edge?

Terminology (cont.)





- **Simple graph vs Multigraph**
  - A pair of vertices may be connected with multiple edges; the edges are not labeled (and thus cannot be distinguished).
- **Path:**
  - A sequence of edges that begins at one vertex and ends at another.
  - May pass the same vertex multiple times.
  - E.g., (0,1)(1,2)(2,0)(0,3) is a path from 0 to 3.
- **Cycle or Circuit:**
  - A path whose first and last vertices are the same.

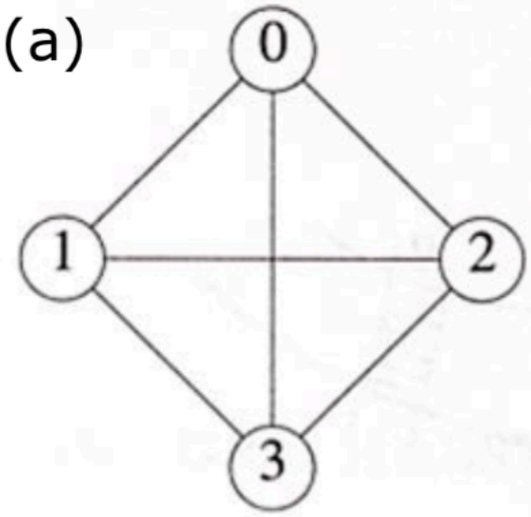
Terminology (cont.)

- **Degree, In-Degree, Out-Degree**
- **Connected Graph:**
  - Has a path between each pair of distinct vertices
- **Subgraph, Induced Subgraph:**
  - A vertex-induced subgraph must include every edge edge in the original graph that connects a pair of selected vertices.

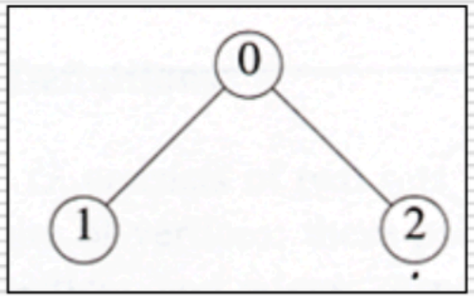
Terminology (cont.)

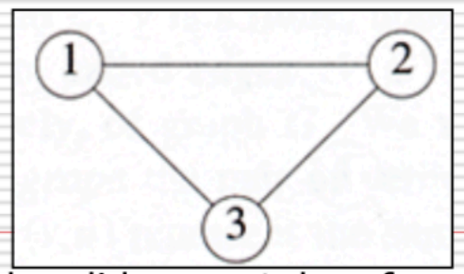
Example

(a)

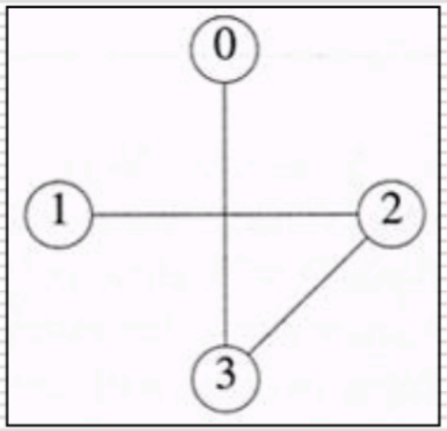


Subgraphs of (a)

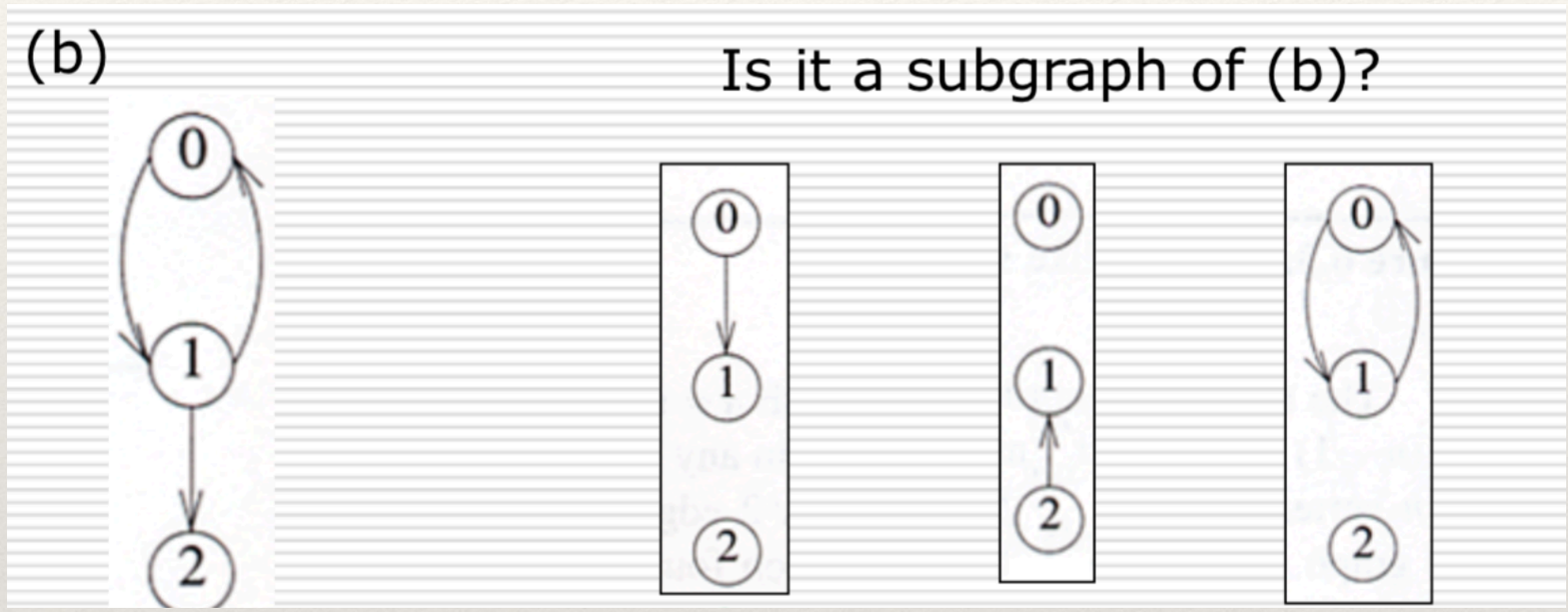




Is it a subgraph of (a)?

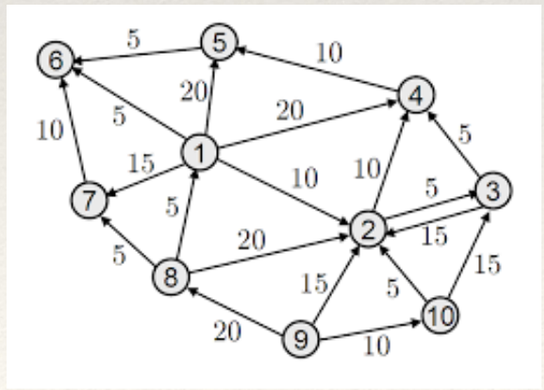


Terminology (cont.)



Terminology (cont.)

- **Tree, Forest**
  - Tree: A connected graph with no cycle
  - Forest: a graph with no cycle
- **Spanning Tree, Spanning Forest**
- **Weighted Graph**



Eulerian Graph

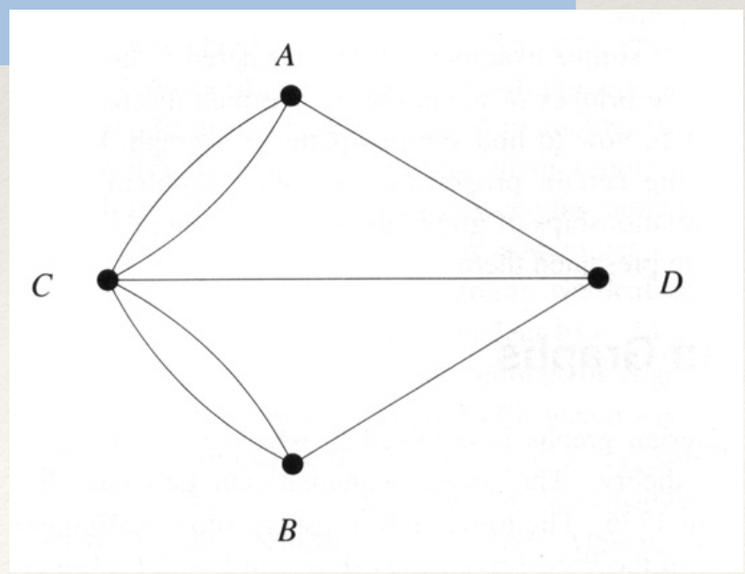
Problem

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.



Eulerian Graph (cont.)

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

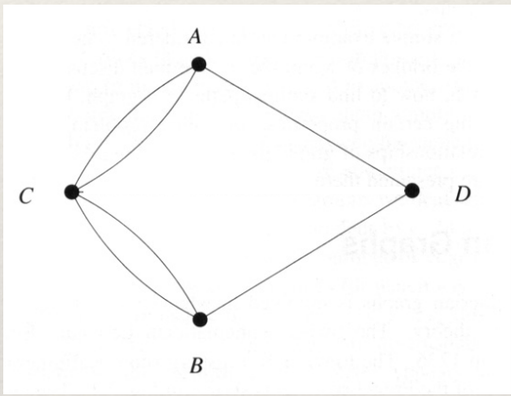
An undirected connected graph has an Eulerian circuit if and only if all of its vertices have even degrees.

The "only if" direction is easy:
When traveling a circuit, we enter and leave each vertex the same number of times. Each edge is used exactly once, the number of vertices adjacent to each vertex must be even.

Eulerian Graph (cont.)

Theorem

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



- We want to prove the "if" direction by induction:
  - Which parameter to apply induction?
  - Reduce the problem without changing it.

- Let's try to remove an edge or a vertex
  - Does not work—the resulting graph might not have the even-degree property

- We should remove a set of edges S such that for each vertex v in the graph, the number of edges from S adjunct to v is even.

Eulerian Graph (cont.)

Theorem

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

- We should remove a set of edges S such that for each vertex v in the graph, the number of edges from S adjunct to v is even.
- Any circuit satisfies this requirement.
- Next question: does even-degree graph always contains a circuit?
  - Hint: one can always leave a newly visited vertex after entering it

Eulerian Graph (cont.)

Theorem

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

Induction hypothesis

A connected graph with $< m$ edges, all of whose vertices have even degrees, contains a circuit that includes each edge exactly once, and we know how to find that circuit.

Note: it is easier to state the induction hypothesis in terms of number of edges rather than the number of circuits, even though the induction is performed on paths.

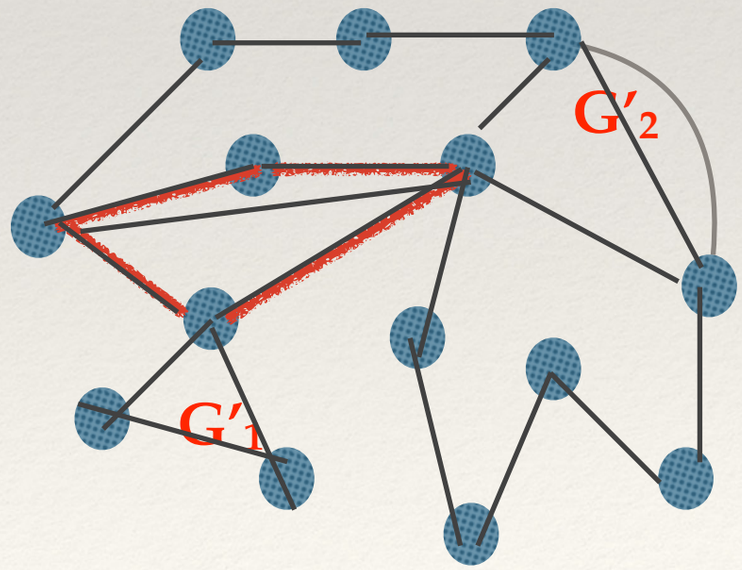
Eulerian Graph (cont.)

Theorem

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

Induction hypothesis

A connected graph with < m edges, all of whose vertices have even degrees, contains a circuit that includes each edge exactly once, and we know how to find that circuit.



Graph Traversal

- The algorithm for Eulerian circuit is not completed yet.
- We still need efficient algorithm to (1) identify connected components, and (2) traverse the graph.
- Traverse all involved objects is a trivial problem in previous lectures—the structure is only linear.

- Two traversal algorithms for graph will be presented: (1) depth-first search and (2) breadth-first search

We need to know not only how it works, but also why it is correct?

Depth-First Search

- Edges: corridors
- Vertices: intersections
- The graph might not be Eulerian

- We want to find a way to see all the paintings in the gallery
- Assume I have unlimited amount of pebbles in my pocket.

Depth-First Search (cont.)

- Edges: corridors
- Vertices: intersections

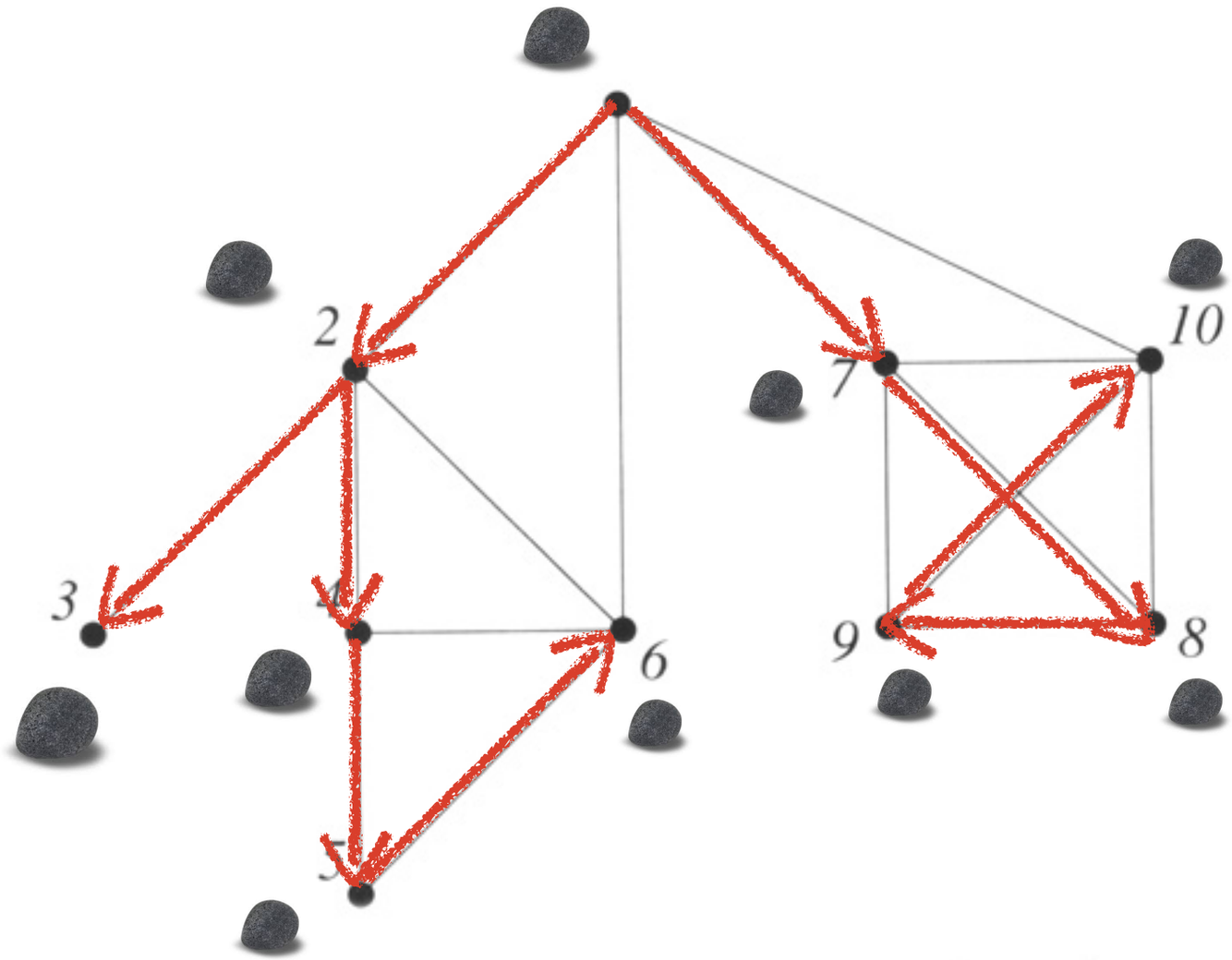


Figure 7.4 A DFS for an undirected graph.

Depth-First Search (cont.)

- DFS for undirected connected graph

```
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
```

Complexity:

- Each edge is visited exactly twice (once from each end).
- The running time is $O(|E|)$.

Depth-First Search (cont.)

Lemma

If G is connected, then all its vertices will be marked by the algorithm Depth_First_Search, and all its edges will be looked at at least once during the execution of the algorithm.

Proof by contradiction:

- Let U be the set of unmarked vertices at the end of the algorithm
- G is connected, at least one vertex in U is connected to a marked vertex in G (contradiction).
- So all vertices are marked. The algorithm visits all edges connected to a visited vertex. It follows that all edges are also visited.

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
```

Time Complexity (assume G is connected): **O(|E|)**

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
```

DFS Tree (cont.)

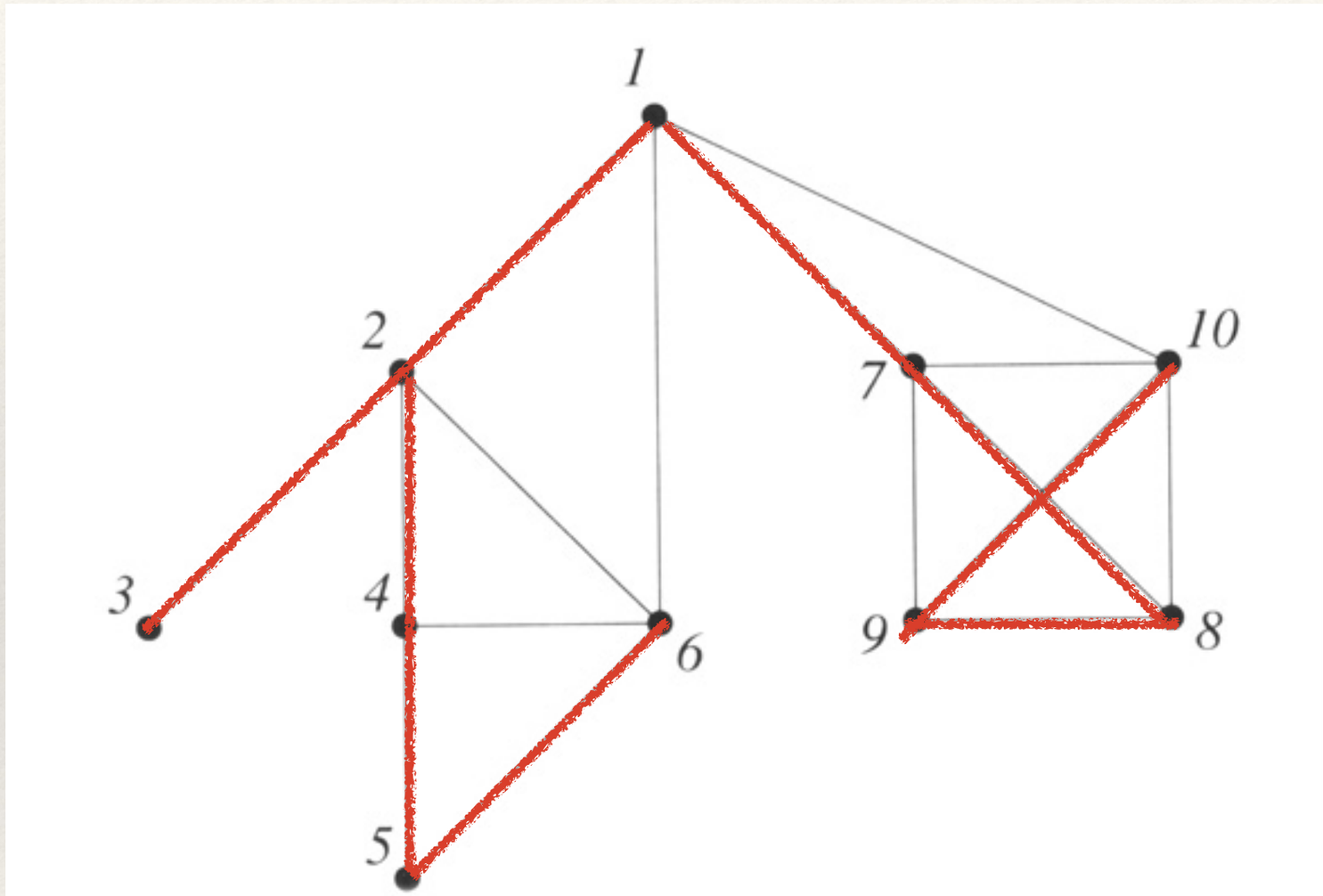


Figure 7.4 A DFS for an undirected graph.

The DFS Tree (cont.)

Lemma (The main property of undirected DFS tree)

For an undirected graph G=(V,E), every edge e in E either (1) belongs to the DFS tree T, or (2) connects two vertices of G, one of which is the **ancestor** of the other in T.

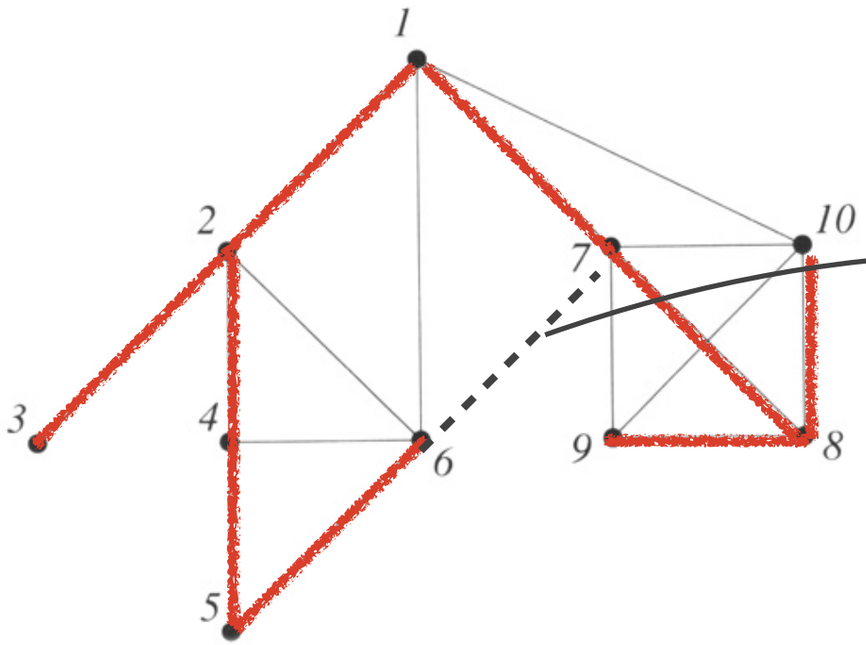


Figure 7.4 A DFS for an undirected graph.

In other words, **cross edges** will not occur

v is called **ancestor** of w in a tree T with root r if v is on the unique path from w to r in T, in this case w is called the **descendant** of v

The DFS Tree (cont.)

Lemma (The main property of undirected DFS tree)

For an undirected graph G=(V,E), every edge e in E either (1) belongs to the DFS tree T, or (2) connects two vertices of G, one of which is the **ancestor** of the other in T.

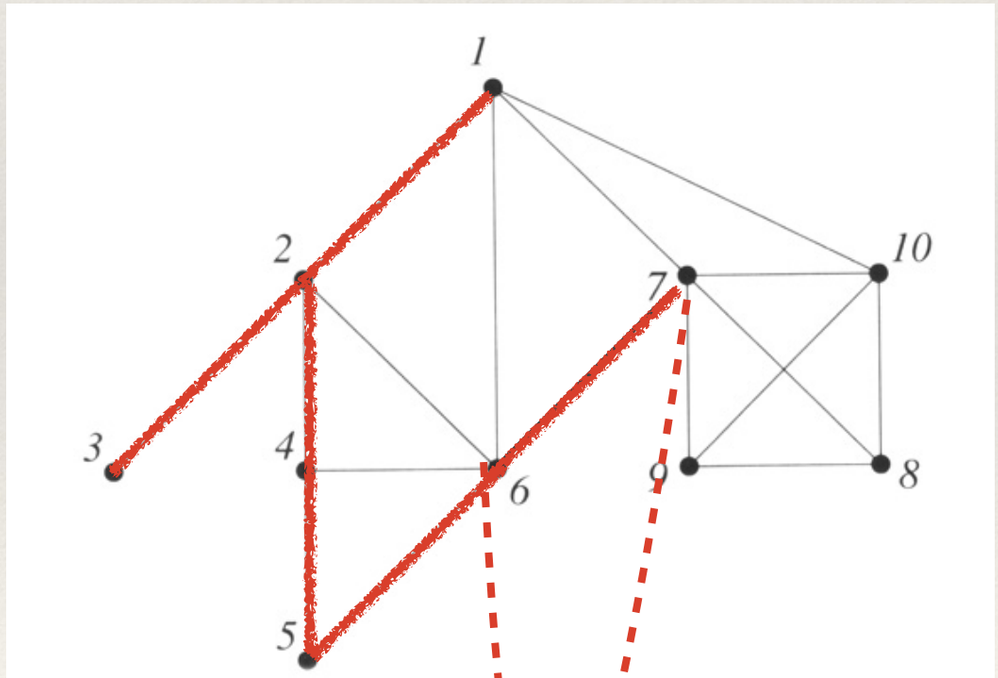


Figure 7.4 A DFS for an undirected graph.

Non-Recursive Version of DFS

- Any recursive program can be rewritten to a non-recursive ones using loop and stack.
- This might be an overkill (we do not need to remember everything).

```
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
```

DFS for Directed Graph

- The DFS procedure for DFS is identical for both directed and undirected graph.
- However, the properties are quite different!

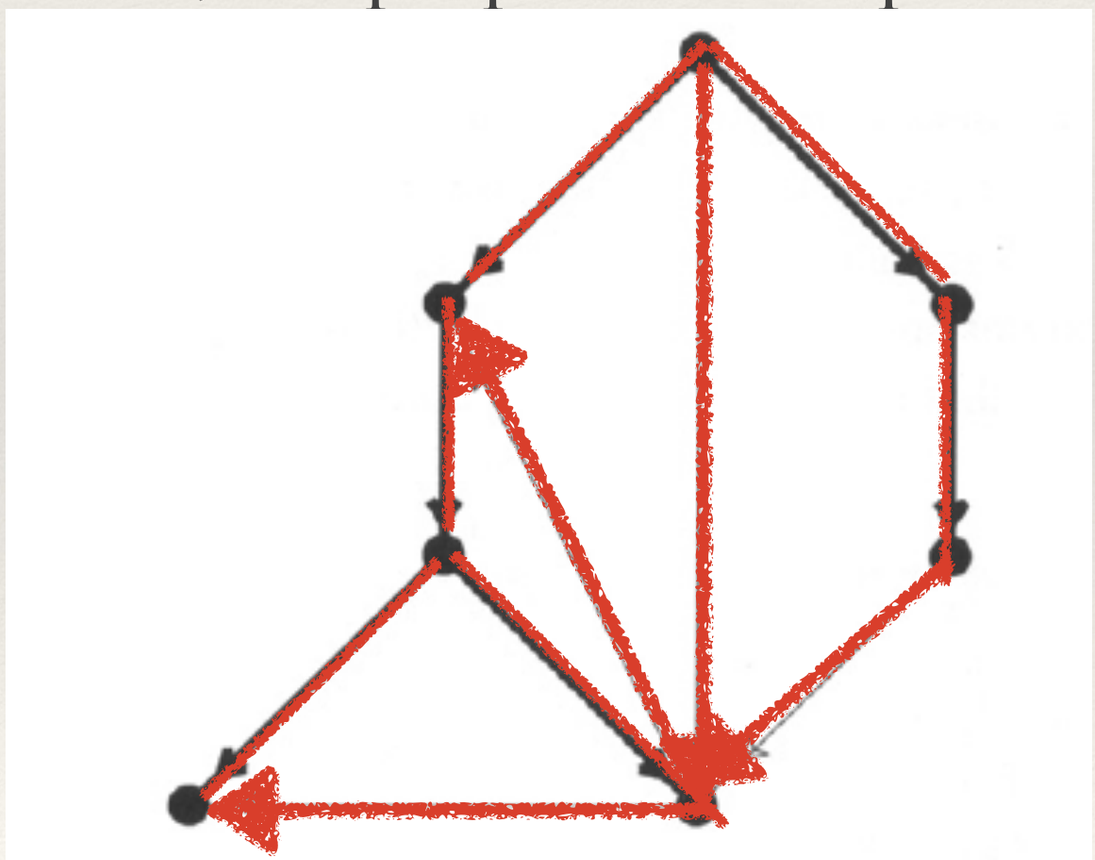


Figure 7.9 A DFS tree for a directed graph.

Tree edges

Back edges

Forward edges

Cross edges

DFS for Directed Graph (cont.)

Lemma (The main property of directed DFS tree)

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.

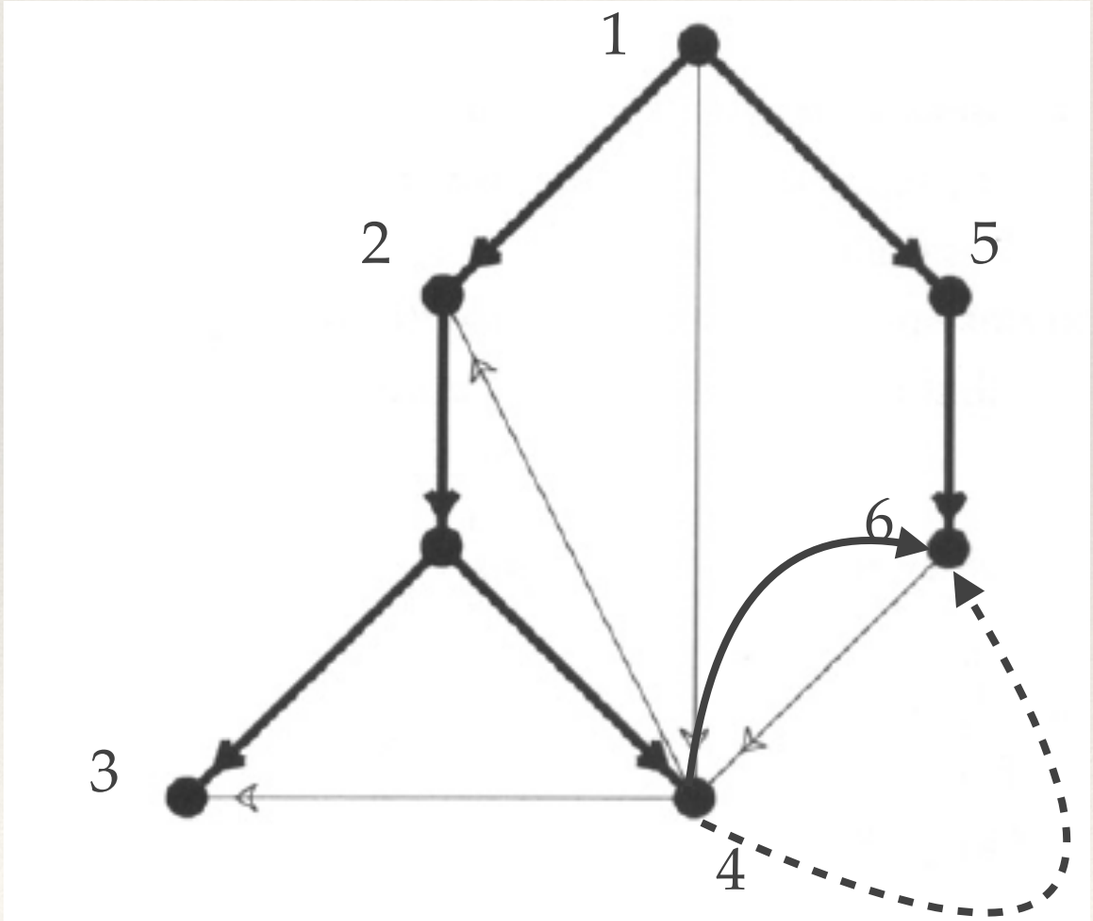


Figure 7.9 A DFS tree for a directed graph.

Cross edge must go from "high to low"

Directed Cycles

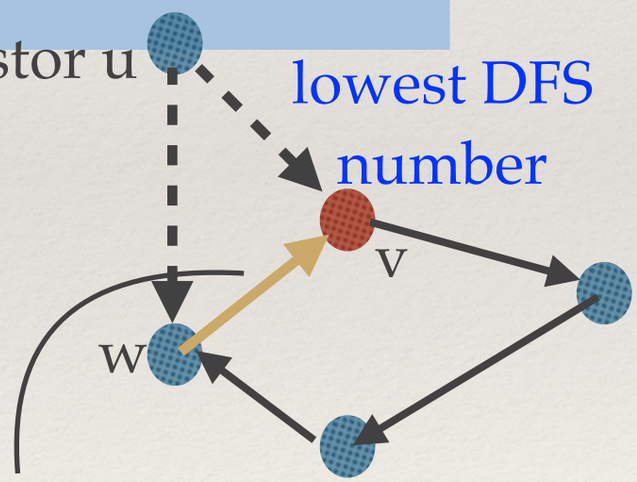
Problem

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

Lemma

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

"only if" direction is more interesting.



1. Goes from high to low: cannot be tree or forward edge
2. If v is an ancestor of w, (w,v) is a back edge
3. Otherwise, v and w are in different subtree of u, the only way to reach w from v if through u or an ancestor of u (cannot go from low to high).
4. But v can reach w following the cycle (contradiction)

That’s the end of the DFS part

Breadth-First Search

```
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
```

E.g. BFS numbers

Breadth-First Search (cont.)

Lemma

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

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

If (v,w) is an edge in E that does not belong to T, then it connects two vertices whose **level** number differ by at most one.

Topological Sorting (cont.)

small to large

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.

Induction Hypothesis

We know how to label all directed acyclic graphs with <n vertices according to the condition above

Can we solve the problem?

There must be a node without incoming transition, i.e., indegree =0

Topological Sorting (cont.)

Can we do it from vertices with zero out degree?

```
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

Problem

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

Let's begin with a simpler case where G is acyclic.

Induction Hypothesis

Given a topological ordering, we know how to find the lengths of the shortest paths from v to the first n-1 vertices.

Hint: the largest vertex according the topological order does not have outgoing edge.

Single-Source Shortest Paths (cont.)

```
Algorithm Acyclic_Shortest_Paths(G, v, n);
{Initially, w.SP = ∞, for every node w.}
{A topological sort has been performed on G, ...}
begin
    let z be the vertex labeled n;
    if z ≠ v then
        Acyclic_Shortest_Paths(G − z, v, n − 1);
        for all w such that (w, z) ∈ E do
            if w.SP + length(w, z) < z.SP then
                z.SP := w.SP + length(w, z)
    else v.SP := 0
end
```

Single-Source Shortest Paths (cont.)

- In the acyclic case, if z is a vertex with label k, then
  - there are no paths from z to vertices with labels <k
  - there are no paths from vertices to z with label >k
- How about the general case?
- Can we define an order with similar property?
  - How about the distance to v?

Single-Source Shortest Paths (cont.)

- How to find the vertex x with shortest path to v?

- How to find the vertex y with 2nd shortest path to v?
  - **other paths from v:** length(v,y)
  - **paths via x:** length(v,x) + length(x,y)

Single-Source Shortest Paths (cont.)

Induction Hypothesis

Given a graph and a vertex v, we know the k vertices that are closest to v and the lengths of the shortest paths to them.

Idea: find a new vertex w with minimal $\min_{u \in V_k}(u\,.\,SP + length(u, w))$

Single-Source Shortest Paths (cont.)

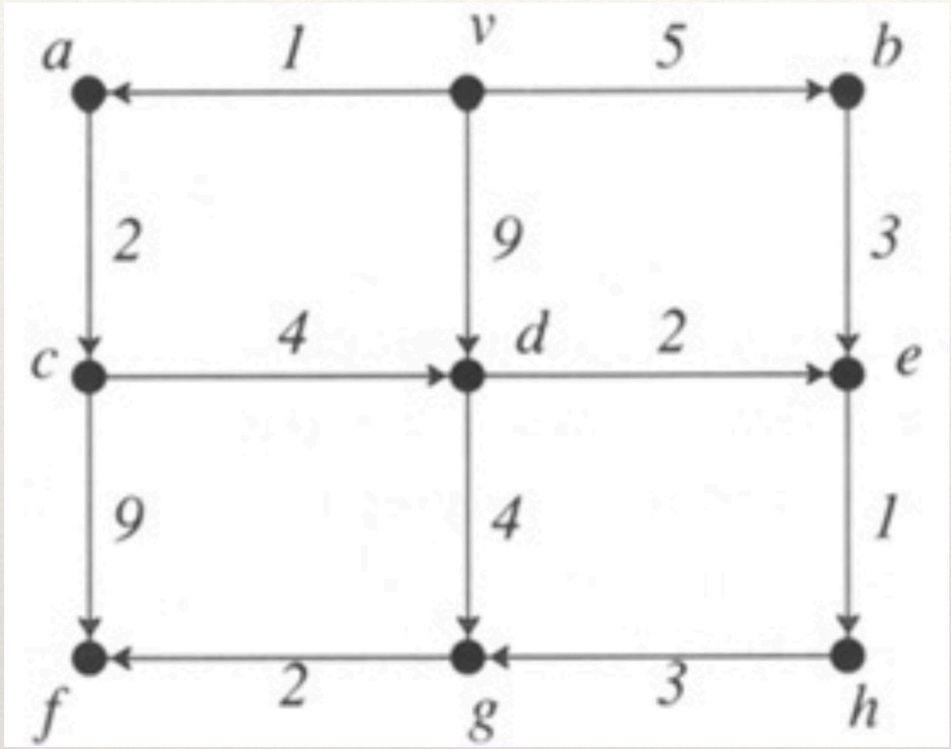
Idea: remember the current minimal and only update the neighbors of the newly added vertex

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

How to design data structure for this algorithm?

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

Single-Source Shortest Paths (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)

Minimum-Cost Spanning Tree

Problem

Given an undirected connected weighted graph G=(V,E), find a spanning tree T of G of minimum total cost.

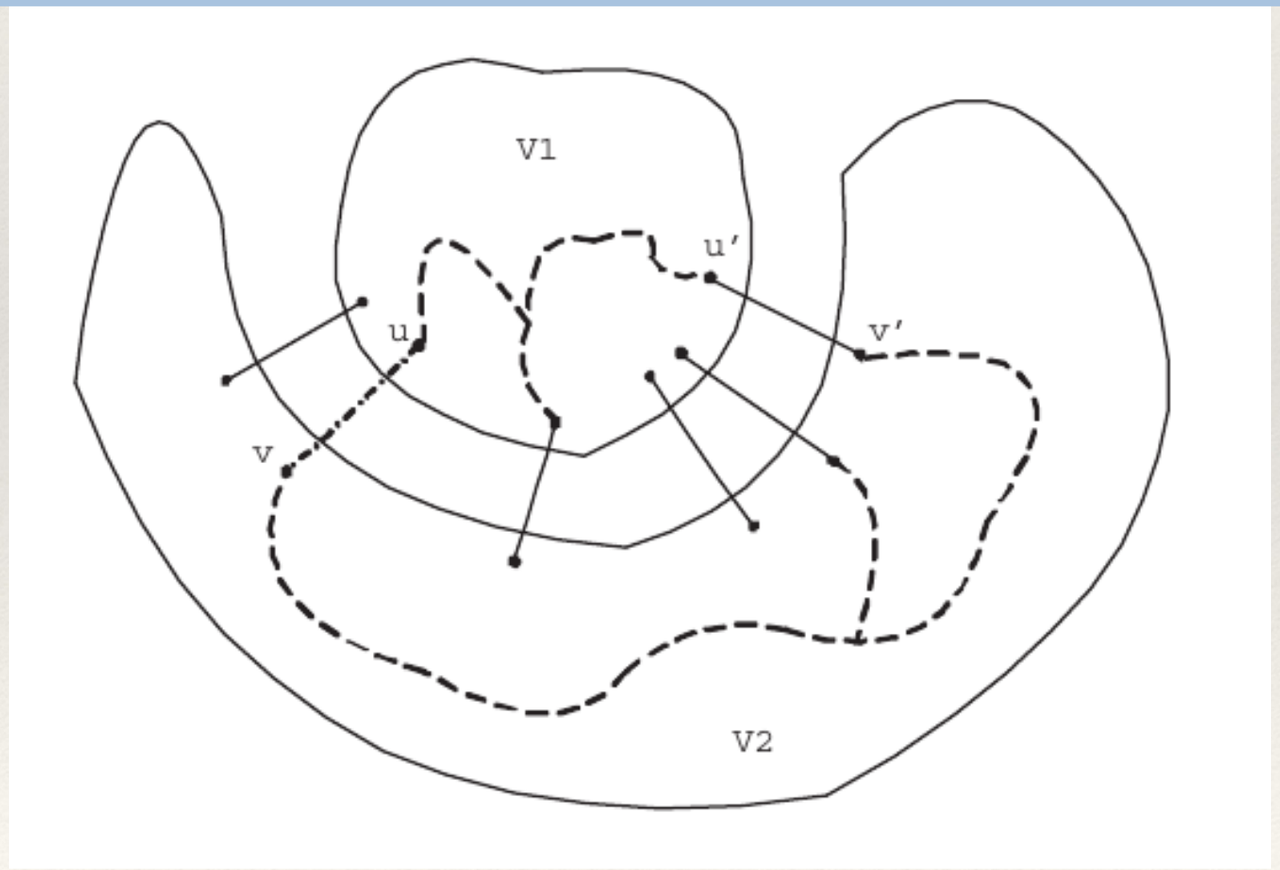
Theorem

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-Cost Spanning Tree (cont.)

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-Cost Spanning Tree (cont.)

```
Algorithm Single_Source_Shortest_Paths(G, v);
// Dijkstra's algorithm
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
```

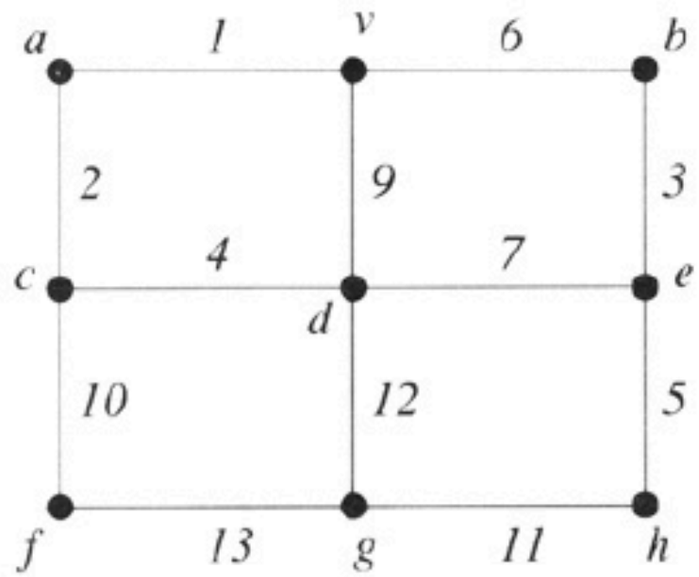
```
if length(w, z) < z.Cost then
    z.Cost := length(w, z)
```

Minimum-Cost Spanning Tree (cont.)

Time complexity: $O((|E| + |V|) \log |V|)$

```
Algorithm MCST(G);
// Prim's algorithm
begin
    for all vertices w do
        w.mark := false;
        w.Cost := ∞;
    x.Cost := 0;
    while there exists an unmarked vertex do
        let w be an unmarked vertex s.t. w.Cost is minimal;
        w.mark := true;
        for all edges (w, z) such that z is unmarked do
            if length(w, z) < z.Cost then
                z.Cost := length(w, z)
end
```

Minimum-Cost Spanning Tree (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	-	-	-	-	-	-	-	-	-

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.

Induction Hypothesis

We know the lengths of the shortest paths between all pairs of vertices such that only k-paths are considered.

Definition

A path from u to w is called a k-path if, **except u and w,** the highest labeled vertex on the path is smaller than k.

All Shortest Paths (cont.)

Problem

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

Induction Hypothesis

We know the lengths of the shortest paths between all pairs of vertices such that only k-paths are considered.

$$\textbf{if } W[x,m] + W[m,y] < W[x,y] \textbf{ then}$$
$$W[x,y] := W[x,m] + W[m,y]$$

All Shortest Paths (cont.)

```
Algorithm All_Pairs_Shortest_Paths(W);
begin
    {initialization}
    for i := 1 to n do
        for j := 1 to n do
            if (i,j) ∈ E then W[i,j] := length(i,j)
            else W[i,j] := ∞;
    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
```

Floyd's Algorithm
Time Complexity = $O(|V|^3)$

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
```

By a reduction to all shortest path. If there is an edge, we set its cost to 0. Otherwise we set the cost to 1.

Transitive Closure (cont.)

Problem

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

```
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
```