GIUSEPPE TURINI

CS-102: COMPUTING AND ALGORITHMS 2
LESSON 08
GRAPHS

Kettering
UNIVERSITY

HIGHLIGHTS

Introduction to Graphs: Definition, Undirected/Directed, Notation/Visualization
Paths and Cycles, Connected/Disconnected/Complete, Dense/Sparse
Weighted and Multigraphs, Subgraphs and Connected Components, Acyclic

ADT Graph: Operations

Representations: Adj Matrix / Adj List / Edge List, Comparison
Example Implementation

Graph Traversals: Introduction

DFS: Depth-First Search, Code, Efficiency, Forest, Applications, Iterator

BFS: Breadth-First Search, Code, Efficiency, Forest, Applications, Iterator

DFS-BFS Comparison

Topological Sort: Introduction, DFS-Based, Source Removal

Other Applications: Spanning Trees, and Minimum Spanning Trees
Shortest Paths, Circuits, Difficult Problems on Graphs

STUDY GUIDE

STUDY MATERIAL

- **This slides.**

SELECTED EXERCISES

- **Set 1:** ex 1-4, ex. 6, ex. 8-9.
- **Set 2:** ex. 1-11, ex. 15-16, ex. 23-26, ex. 31-37.

ADDITIONAL RESOURCES

- **"Object-Oriented Data Structures Using Java (4th Ed.)", chap. 10.**
- "Data Abstraction and Problem Solving with Java (3rd Ed.)", chap. 14.
- visualgo.net/en/graphs
- visualgo.net/en/dfsbfs

INTRO TO GRAPHS - DEFINITION

A **graph** is a set of **vertices** (nodes) with some vertex pairs are connected by **edges**.

Formally, a **graph G** is defined by a pair of 2 sets: $G = \langle V, E \rangle$

- a finite non-empty set of **vertices V**, and
- a set of vertex pairs called **edges E**.

A **subgraph G'** of a **graph G**, consists of a subset **V'** of **V** and a subset **E'** of **E**.

Two vertices **a** and **b** are **adjacent**, if their graph includes an edge connecting them.

INTRO TO GRAPHS - UNDIRECTED / DIRECTED

If **edges E** are unordered vertex pairs, **(a, b)=(b, a)**, vertices **a** and **b** are connected by the **undirected edge (a, b)**, and call **a** and **b endpoints** of the **edge (a, b)**.

A **graph G** whose every edge is undirected is called an **undirected graph**.

If **edges E** are ordered vertex pairs, **(a, b)!=(b, a)**, vertices **a** and **b** are connected by the **directed edge (a, b)**, and call **a** the edge **tail** and **b** the edge **head**.

A graph G whose every edge is directed is called a **directed graph**, or **digraph**.

INTRO TO GRAPHS - NOTATION AND VISUALIZATION

$$G_a = \left\langle \begin{array}{l} V = \{\, a, b, c, d, e, f \,\}, \\ E = \{\, (a,c), (a,d), (b,c), (b,f), (c,e), (d,e), (e,f) \,\} \end{array} \right\rangle$$

$$G_b = \left\langle \begin{array}{l} V = \{\, a, b, c, d, e, f \,\}, \\ E = \{\, (a,c), (b,c), (b,f), (c,e), (d,a), (d,e), (e,c), (e,f) \,\} \end{array} \right\rangle$$

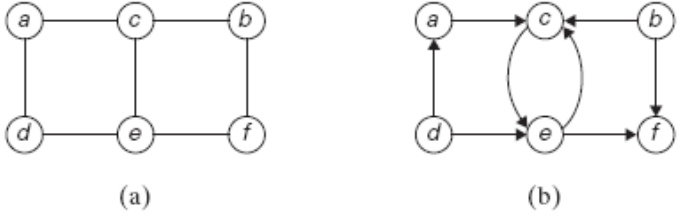


FIGURE 1.6 (a) Undirected graph. (b) Digraph.

INTRO TO GRAPHS - PATHS AND CYCLES

A **path** between vertices **a** and **b** is a sequence of edges starting at **a** and ending at **b** (may pass through the same path vertex more than once).
A **simple path** is a path that passes through each path vertex only once.

A **directed path** is a sequence of vertices, each pair connected by a **directed edge** from first vertex to second vertex.

The **path length** is the total number of edges in the path.

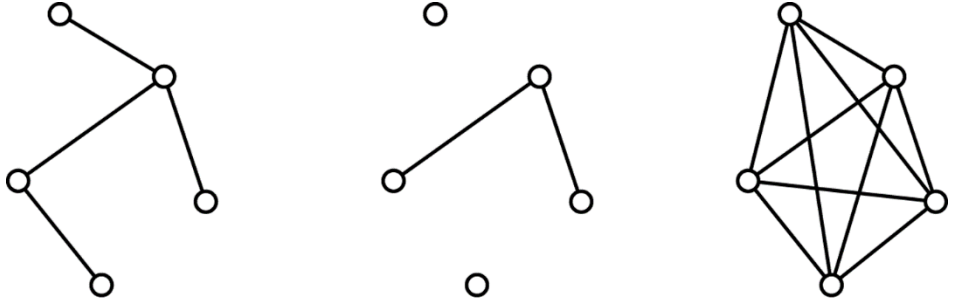
A **cycle** is a path that begins and ends at the same vertex (may pass through the same cycle vertex more than once).
A **simple cycle** is a cycle that passes through a each cycle vertex only once.

INTRO TO GRAPHS - CONNECTED AND COMPLETE

A **connected graph (a)** has a **path** between each pair of distinct vertices.

A **disconnected graph (b)** has at least 2 vertices without a **path** between them.

A **complete graph (c)** has an **edge** between each pair of distinct vertices.



INTRO TO GRAPHS - DENSE / SPARSE

Unless stated otherwise, **self-edges are not allowed** in a graph.
Multiple edges are not allowed (between the same vertex pair) in a graph.

So, we have the following inequality for an **undirected graph with no loops**:

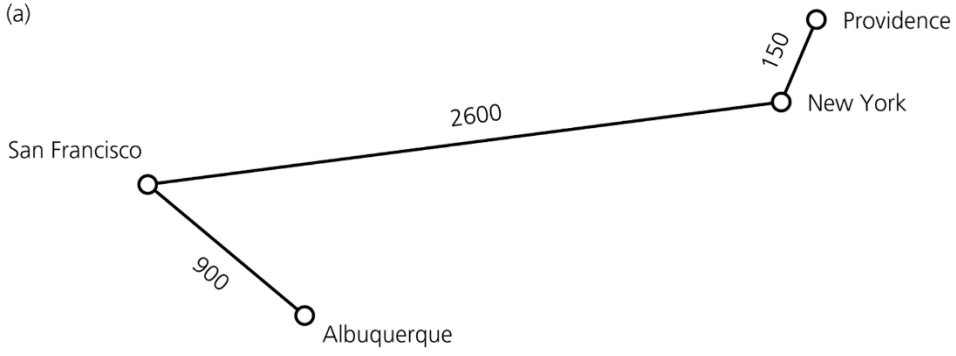
$$0 \leq |E| \leq |V| \left(|V| - 1 \right) / 2$$

A **dense graph** has few missing edges (in respect to a complete graph with same **V**).

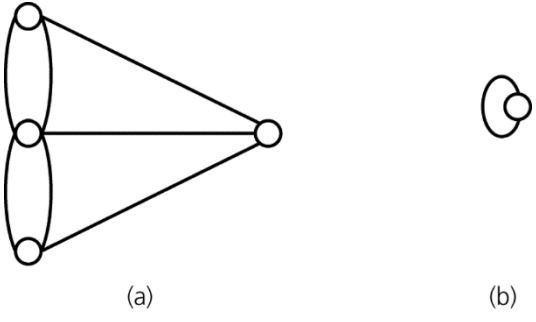
A **sparse graph** has few edges (in respect to a complete graph with same **V**).

INTRO TO GRAPHS - WEIGHTED AND MULTIGRAPHS

A **weighted graph** has numbers (**weights** or costs) assigned to each of its edges.



A **multigraph is not a graph** because it allows:
multiple edges between vertices, and self-edges,
both not allowed in a graph.



INTRO TO GRAPHS - SUBGRAPHS AND COMPONENTS

SUBGRAPHS AND CONNECTED COMPONENTS

A **subgraph** G' of a given graph G is:

$$\begin{cases} G = \langle V, E \rangle \\ G' = \langle V', E' \rangle, \quad V' \subseteq V \wedge E' \subseteq E \end{cases}$$

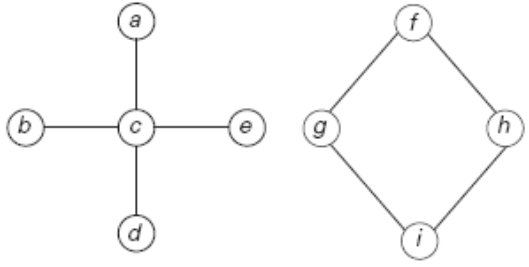


FIGURE 1.9 Graph that is not connected.

A **connected graph** is a graph where for every pair of vertices **a** and **b** there is a path from **a** to **b**. Otherwise the graph is not connected, and in this case the graph will consist of several connected subgraphs called **connected components**.

A **connected component** of a graph, is a maximal (i.e. not expandable by including another vertex and an edge) connected subgraph of a given graph.

INTRO TO GRAPHS - ACYCLIC GRAPHS

Many applications need to know if a graph under consideration has cycles.

A graph with no cycles is said to be **acyclic**.

Note: A **tree** is a connected acyclic graph.

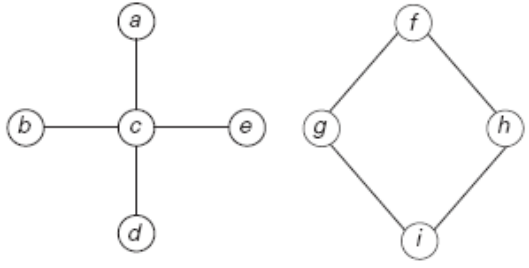


FIGURE 1.9 Graph that is not connected.

ADT GRAPH - OPERATIONS

GRAPH OPERATIONS

Note: In defining graphs, the vertices may or may not contain values.
Operations of the ADT Graph:

- **create** an empty graph;
- determine whether a graph **is empty**;
- determine the **number of vertices** and the **number of edges** in a graph;
- determine whether an **edge exists** between two given vertices;
- return the **edge weight** (only for weighted graphs);
- **insert a vertex** (with unique key) in a graph with keyed vertices;
- **insert an edge** between two given vertices in a graph;
- **delete a vertex** from a graph and any incident edge;
- **delete the edge** between two given vertices in a graph;
- **retrieve a vertex** from a graph given its unique search key.

ADT GRAPH - REPRESENTATIONS 1

Graphs for computer algorithms are usually represented in one of two ways:

- The **adjacency matrix** of a graph with **n** vertices is a **n**x**n** matrix with 1 row and 1 column for each graph vertex, in which element **[i, j]** in row **i** and column **j** is equal to **1** if there is an edge from vertex **i** to vertex **j**, or **0** otherwise.

- The **adjacency list** of a graph is a collection of linked lists, 1 for each vertex, that contains all the vertices adjacent to the list vertex. Usually, such lists start with a header identifying the vertex for which the list is compiled. So, adjacenty lists indicate columns of the adjacency matrix that, for a given vertex, contain **1**.

- The **edge list** is a list of the graph edges, each including its properties (start and end vertices, and weight).

ADT GRAPH - REPRESENTATIONS 2

Question: Why the adjacency matrix of undirected graphs is always symmetric?

Answer: Because in an undirected graph each edge (u, v) = (v, u).

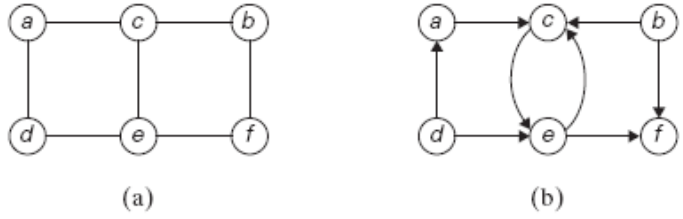


FIGURE 1.6 (a) Undirected graph. (b) Digraph.

Question: What is the best way to represent sparse and dense graphs?

Answer: For sparse graphs usually the adjacency list uses less memory than the adjacency matrix, whereas for dense graphs usually the adjacency matrix is the best representation.

	a	b	c	d	e	f
a	0	0	1	1	0	0
b	0	0	1	0	0	1
c	1	1	0	0	1	0
d	1	0	0	0	1	0
e	0	0	1	1	0	1
f	0	1	0	0	1	0

(a)

a	→	c	→	d			
b	→	c	→	f			
c	→	a	→	b	→	e	
d	→	a	→	e			
e	→	c	→	d	→	f	
f	→	b	→	e			

(b)

FIGURE 1.7 (a) Adjacency matrix and (b) adjacency lists of the graph in Figure 1.6a.

ADT GRAPH - WEIGHTED GRAPH REPRESENTATIONS

A **weighted graph** can be represented by an **adjacency matrix** A, where each element A[i, j] contains the weight of the edge (i, j), if there is such an edge, and a special symbol (e.g. ∞) otherwise. This matrix is called **weight matrix** or **cost matrix**.

A **weighted graph** can be represented by an **adjacency list** including in its nodes both the name of the adjacent vertex and the weight of the corresponding edge.

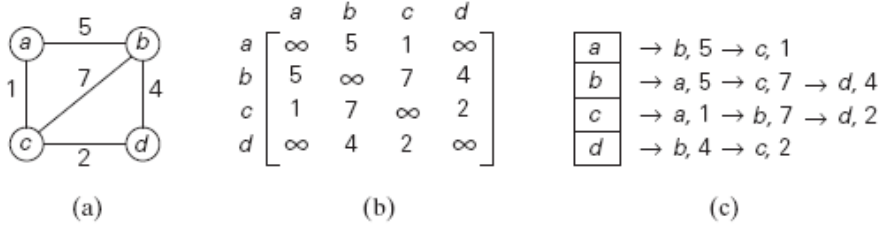
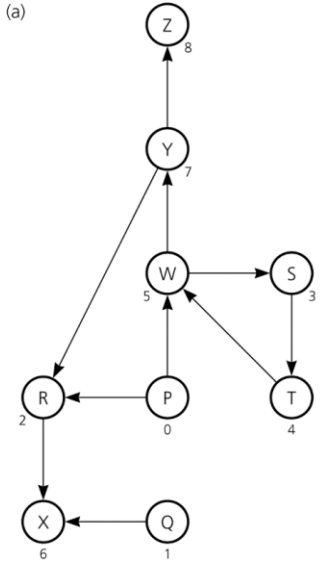


FIGURE 1.8 (a) Weighted graph. (b) Its weight matrix. (c) Its adjacency lists.

ADT GRAPH - DIRECTED GRAPH REPRESENTATIONS

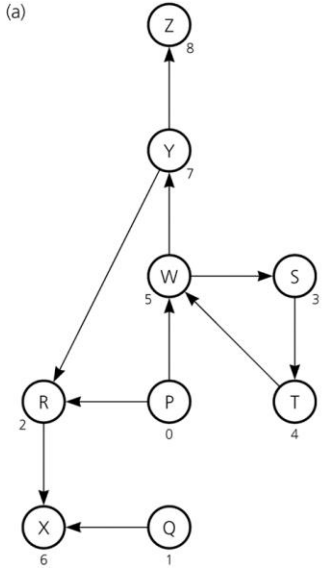
Left: A **directed graph (a)** and its **adjacency matrix (b)**.

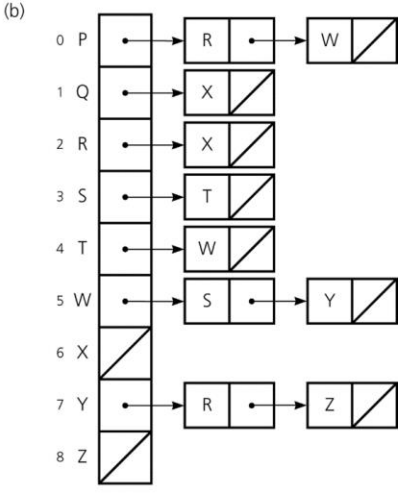
Right: A **directed graph (a)** and its **adjacency list (b)**.



(b)

		0	1	2	3	4	5	6	7	8
		P	Q	R	S	T	W	X	Y	Z
0	P	0	0	1	0	0	1	0	0	0
1	Q	0	0	0	0	0	0	1	0	0
2	R	0	0	0	0	0	0	1	0	0
3	S	0	0	0	0	1	0	0	0	0
4	T	0	0	0	0	0	1	0	0	0
5	W	0	0	0	1	0	0	0	1	0
6	X	0	0	0	0	0	0	0	0	0
7	Y	0	0	1	0	0	0	0	0	1
8	Z	0	0	0	0	0	0	0	0	0

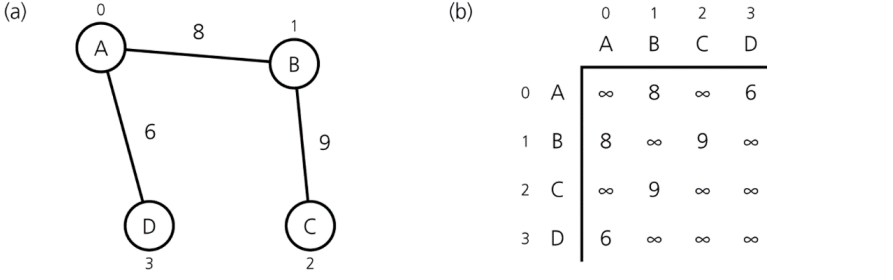


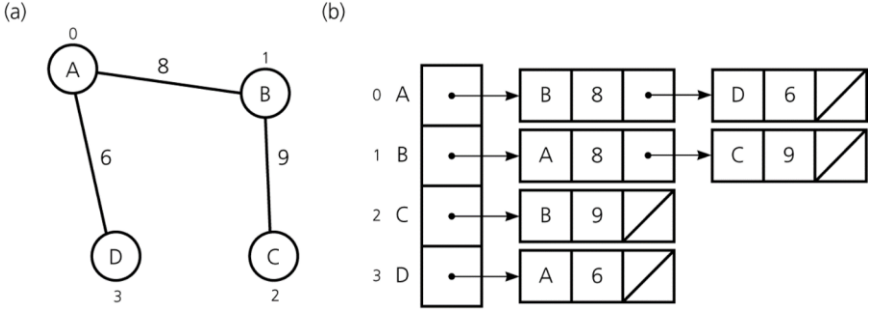


ADT GRAPH – U/W GRAPH REPRESENTATIONS

Left: A **weighted undirected graph (a)** and its **adjacency matrix (b)**.

Right: A **weighted undirected graph (a)** and its **adjacency list (b)**.





ADT GRAPH - REPRESENTATION COMPARISON

COMPARISON OF GRAPH REPRESENTATIONS

Adjacency matrix compared with adjacency list:

- **Operation 1:** determine if there is an edge from vertex **a** to vertex **b**.
- **Operation 2:** find all vertices adjacent to a given vertex **a**.

- **Adjacency Matrix:** supports operation 1 more efficiently.
- **Adjacency List:** supports operation 2 more efficiently.

- **Adjacency Matrix:** for dense graphs needs less memory than adjacency list.
- **Adjacency List:** for sparse graphs needs less memory than adjacency matrix.

GRAPH TRAVERSALS - INTRODUCTION

A **graph-traversal** is the visit of the vertices we can reach through paths, starting from a "source" vertex.

A graph-traversal **visits all graph vertices if and only if the graph is connected**.

A graph-traversal can **loop indefinitely if there is a cycle**. To avoid this, the traversal algorithm must: mark vertices when visited, and never visit a vertex more than once.

There are 2 important graph-traversals that process all vertices and edges of a graph:

- **Depth-First Search (DFS) traversal** using a last-visited-first-explored strategy.
- **Breadth-First Search (BFS) traversal** using a first-visited-first-explored strategy.

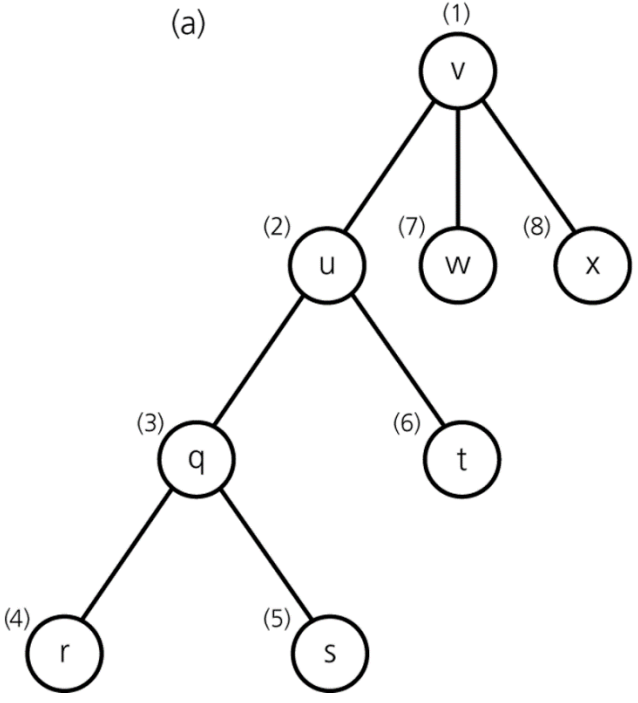
GRAPH TRAVERSALS - DFS ALGORITHM

1. **Start** traversal at an **arbitrary vertex** (**"source"**), marking it as visited.
2. **From current vertex**, move to an **adjacent unvisited vertex** (arbitrary).
3. **Loop (2)** until reaching a **dead end**: vertex without adjacent unvisited vertices.
4. **At a dead end**, the traversal **backs up 1 edge** to the vertex it came from.
5. **After backing up**, the traversal tries step (2) if possible, otherwise step (4).
6. **Traversal stops** after backing up to the **"source" vertex**, if it is a **dead end**.

When DFS stops: connected component of "source" has been completely visited.
If unvisited vertices remain restart DFS at any one of them.

GRAPH TRAVERSALS - DFS EXAMPLE

Problem: **DFS traversal** of an **undirected graph** (node visit order as superscript).
Note: Adjacent unvisited vertices chosen alphabetically.
Result: **v**, **u**, **q**, **r**, **s**, **t**, **w**, **x**.



GRAPH TRAVERSALS - DFS USING A STACK

It is convenient to use a **stack** to trace the operation of the DFS traversal:

- **push** a vertex onto the stack when an unvisited adjacent vertex is reached.
- **pop** a vertex off the stack when a dead end is reached.

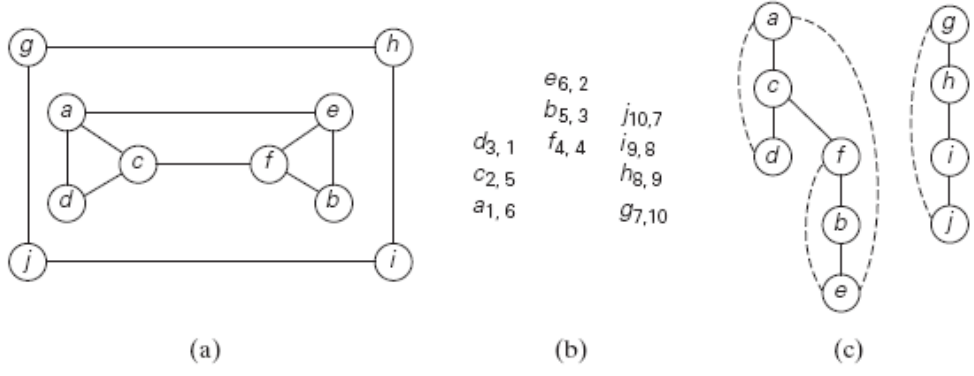


FIGURE 3.10 Example of a DFS traversal. (a) Graph. (b) Traversal's stack (the first subscript number indicates the order in which a vertex is visited, i.e., pushed onto the stack; the second one indicates the order in which it becomes a dead-end, i.e., popped off the stack). (c) DFS forest with the tree and back edges shown with solid and dashed lines, respectively.

GRAPH TRAVERSALS - DFS RECURSIVE CODE

The following is a **recursive Java implementation of the DFS traversal**:

```
void dfs( Graph g ) { // Main DFS algorithm.
   int c = 0; // Init counter to keep track of the traversal order.
   for( int v = 0; v < g.getVNum(); v++ ) { // Iterate among all graph vertices.
      // If vertex not visited, start a DFS at that vertex.
      if( g.getVisited(v) == 0 ) { c = dfsRec( g, v, c ); } }
}

int dfsRec( Graph g, int v, int c ) { // Recursive DFS on vertex v and counter c.
   c++; g.setVisited( v, c ); // Mark input vertex with current traversal order.
   for( int w = 0; w < g.getVNum(); w++ ) { // Iterate vertices adjacent to input vertex.
      if( g.isAdjacent( v, w ) ) { // If vertex not visited, start a DFS at that vertex.
         if( g.getVisited( w ) == 0 ) { c = dfsRec( g, w, c ); } } }
   return c;
}
```

GRAPH TRAVERSALS - DFS FOREST 1

It is useful to accompany a DFS traversal by constructing a **DFS forest**:

- **Start vertex** of DFS traversal is the root of a tree in DFS forest.
- **When an unvisited vertex** is reached, it is linked as a child of the vertex we came from, and that edge is a **tree edge**.
- **When an edge is found** that leads to a previously visited vertex other than its immediate predecessor, that edge is a **back edge**.

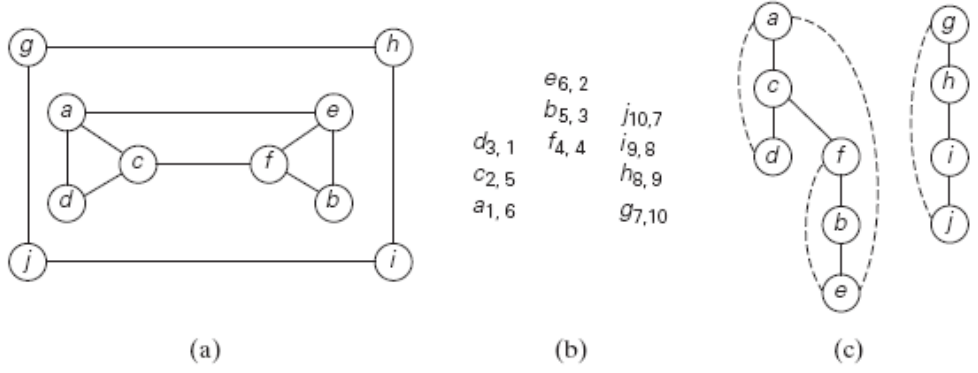


FIGURE 3.10 Example of a DFS traversal. (a) Graph. (b) Traversal's stack (the first subscript number indicates the order in which a vertex is visited, i.e., pushed onto the stack; the second one indicates the order in which it becomes a dead-end, i.e., popped off the stack). (c) DFS forest with the tree and back edges shown with solid and dashed lines, respectively.

GRAPH TRAVERSALS - DFS FOREST 2

A **DFS forest**, that is obtained as a product of a DFS traversal, is not actually a forest. In fact, a DFS forest is just the original graph with its edges classified in:

- **tree edges**, used by DFS traversal to reach unvisited vertices; and
- **back edges**, connecting vertices to previously visited vertices.

Note: A DFS traversal and its DFS forest have been extremely helpful to develop algorithms for checking important properties of graphs.

The DFS traversal generates 2 orderings of vertices:

- The order in which the vertices are reached for the first time (**push**).
- The order in which the vertices become dead ends (**pop**).

GRAPH TRAVERSALS - DFS TIME EFFICIENCY

The DFS traversal is quite efficient since it takes just the time proportional to the size of the data structure used for representing the graph in question.

- For the **adjacency matrix** representation, the **traversal time** is in **$\Theta(|V|^2)$**.
- For the **adjacency list** representation, the **traversal time** is in **$\Theta(|V|+|E|)$**.
- **Note: $|V|$** and **$|E|$** are the number of vertices and edges of the graph.

GRAPH TRAVERSALS - DFS APPLICATIONS

How to check the connectivity of a graph:

1. Start a DFS traversal at an arbitrary vertex.
2. After the DFS stops, check if all graph vertices have been visited:
    a. If they have, the **graph is connected**.
    b. Otherwise, if some vertices are still unvisited, the **graph is not connected**.

How to check the acyclicity of a graph:

1. Start a DFS traversal at an arbitrary vertex.
2. Check the DFS forest generated:
    a. If it does not have back edges, the **graph is acyclic**.
    b. If it has a back edge **a-b**, the **graph has a cycle** including path from **b** to **a** via a sequence of tree edges followed by the back edge **a-b**.

GRAPH TRAVERSALS - BFS ALGORITHM

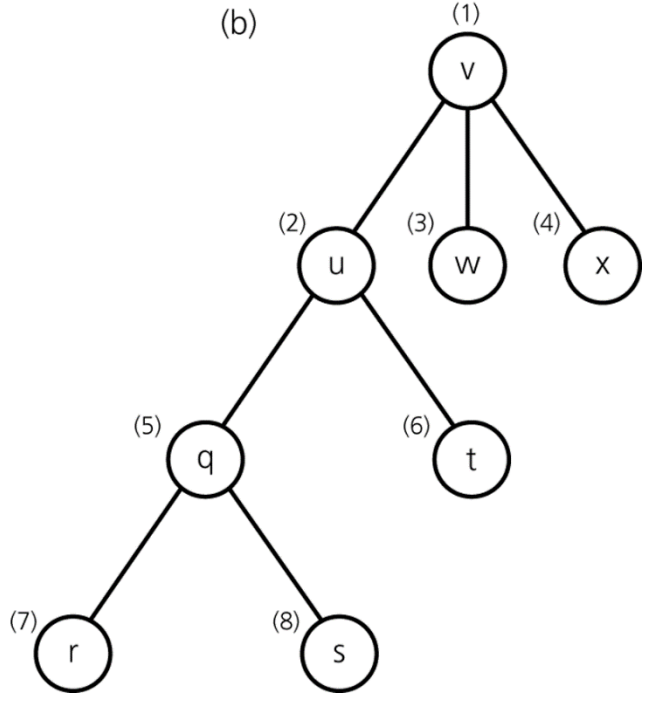
DFS traversal is for the "brave" (goes as far from start as it can),
BFS traversal is for the "cautious" (stays as close at start as it can then goes far).

1. **Visit (at once) all adjacent unvisited vertices** to start vertex (“source”).
2. **Select one of the recently visited vertices** (arbitrary), and move to it.
4. **Repeat (1)**, with the current vertex as the “source”.
5. **Traversal stops** when all visited vertices have been processed by step 1.

When BFS stops: connected component of “source” has been completely visited.
If unvisited vertices remain restart BFS at any one of them.

GRAPH TRAVERSALS - BFS EXAMPLE

Problem: **BFS traversal** of an **undirected graph** (node visit order as superscript).
Note: Adjacent unvisited vertices chosen alphabetically.
Result: **v**, **u**, **w**, **x**, **q**, **t**, **r**, **s**.



GRAPH TRAVERSALS - BFS USING A QUEUE

DFS traversal uses a stack,
BFS traversal uses a queue.

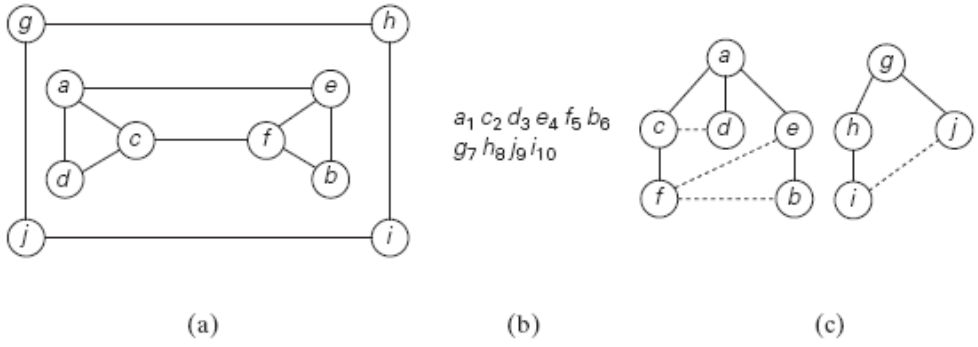


FIGURE 3.11 Example of a BFS traversal. (a) Graph. (b) Traversal queue, with the numbers indicating the order in which the vertices are visited, i.e., added to (and removed from) the queue. (c) BFS forest with the tree and cross edges shown with solid and dotted lines, respectively.

1. **Init queue** with start vertex, mark it visited.

On each iteration:

2. **Add queue** all unvisited adjacent vertices to front vertex, mark them visited.
3. **Dequeue front vertex**, and do (2).

4. **BFS stops** if queue is empty.

GRAPH TRAVERSALS - BFS ITERATIVE CODE

The following is an **iterative Java implementation of the BFS traversal**:

```
void bfs( Graph g ) {
   int c = 0;
   for( int v = 0; v < g.getVNum(); v++ ) {
      if( g.getVisited(v) == 0 ) {
         c++; g.setVisited( v, c );
         Queue<Integer> queue = new LinkedList<Integer>(); queue.add(v);
         int front = -1;
         while( !queue.isEmpty() ) {
            front = queue.peek();
            for( int w = 0; w < g.getVNum(); w++ ) {
               if( ( g.isAdjacent( front, w ) ) && ( g.getVisited(w) == 0 ) ) {
                  c++; g.setVisited( w, c ); queue.add(w); } }
            queue.poll(); } } }
}
```

GRAPH TRAVERSALS - BFS FOREST 1

It is useful to accompany a BFS traversal by constructing a **BFS forest**:

- **Start vertex** of BFS is the root of a tree in BFS forest.
- **When an unvisited vertex** is reached:
  link it as a child of the
  vertex we came from,
  this is a **tree edge**.
- **When an edge is found**
  that leads to a previously
  visited vertex other than
  its immediate predecessor,
  that edge is a **cross edge**.

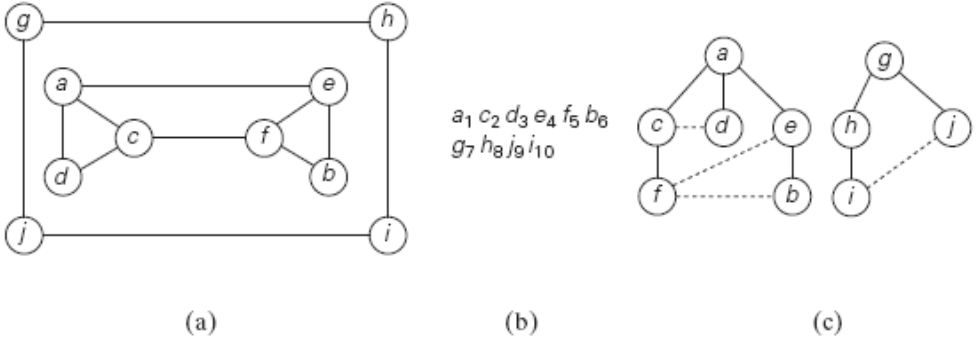


FIGURE 3.11 Example of a BFS traversal. (a) Graph. (b) Traversal queue, with the numbers indicating the order in which the vertices are visited, i.e., added to (and removed from) the queue. (c) BFS forest with the tree and cross edges shown with solid and dotted lines, respectively.

GRAPH TRAVERSALS - BFS FOREST 2

A **BFS forest**, that is obtained as a product of a BFS traversal, is not actually a forest. In fact, a BFS forest is just the original graph with its edges classified in:

- **tree edges**, used by BFS traversal to reach unvisited vertices; and
- **cross edges**, connecting vertices to previously visited vertices.

The BFS traversal generates 1 ordering of vertices:

- The order in which the vertices are reached for the first time (**enqueue**).

Note: Unlike DFS, BFS yields 1 ordering of vertices because the queue is a **FIFO** (first-in first-out) structure and hence the order in which the vertices are added (**enqueue**) to the queue is the same order in which they are removed (**dequeue**) from it.

GRAPH TRAVERSALS - BFS TIME EFFICIENCY

The BFS traversal has the same efficiency as the DFS traversal.

- For the **adjacency matrix** representation, the **traversal time** is in **$\Theta(|V|^2)$**.
- For the **adjacency list** representation, the **traversal time** is in **$\Theta(|V|+|E|)$**.
- **Note: $|V|$** and **$|E|$** are the number of vertices and edges of the graph.

GRAPH TRAVERSALS - BFS APPLICATIONS

How to find the min-edge path between two given vertices:

1. Start a BFS traversal at one of the two given vertices.
2. Stop the BFS traversal as soon as the other vertex is reached.

Example: From vertex **a** to vertex **g** of this graph.

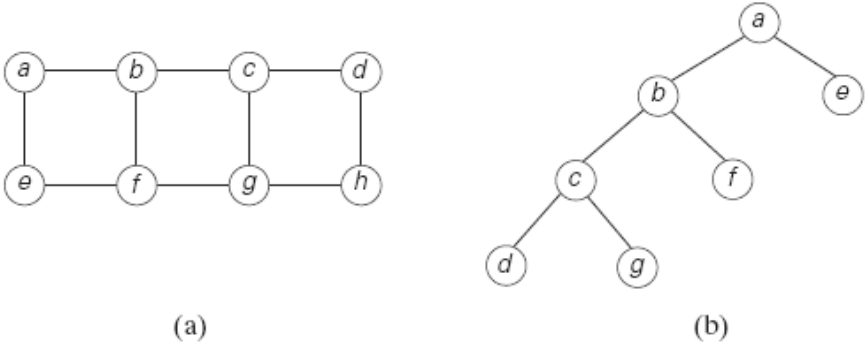


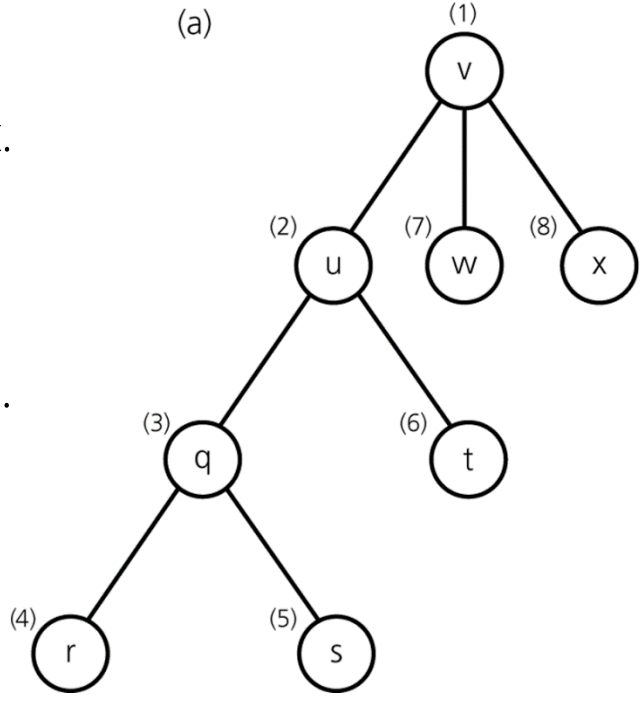
FIGURE 3.12 Illustration of the BFS-based algorithm for finding a minimum-edge path. (a) Graph. (b) Part of its BFS tree that identifies the minimum-edge path from a to g.

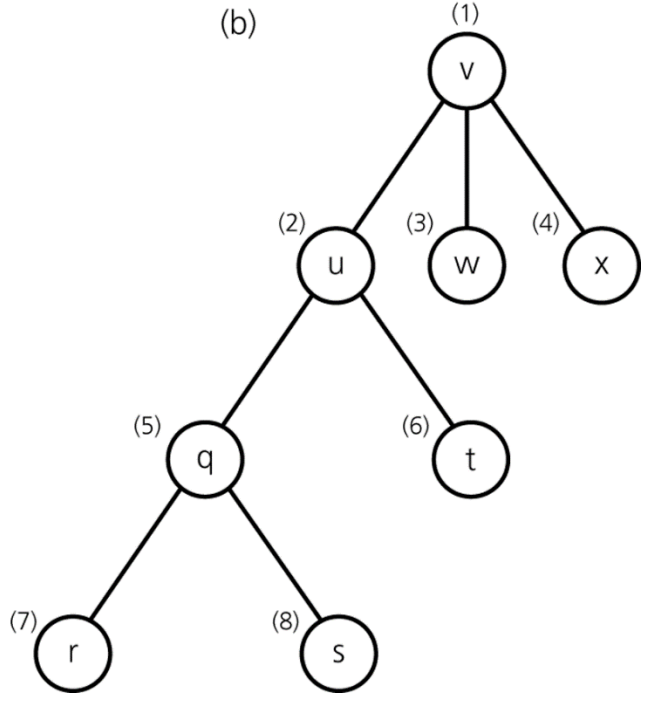
GRAPH TRAVERSALS - DFS-BFS COMPARISON 1

Comparison of **DFS and BFS traversals** of this **undirected graph**.

Left: DFS Traversal
v, **u**, **q**, **r**, **s**, **t**, **w**, **x**.

Right: BFS Traversal
v, **u**, **w**, **x**, **q**, **t**, **r**, **s**.





GRAPH TRAVERSALS - DFS-BFS COMPARISON 2

DFS-BFS COMPARISON - SUMMARY TABLE

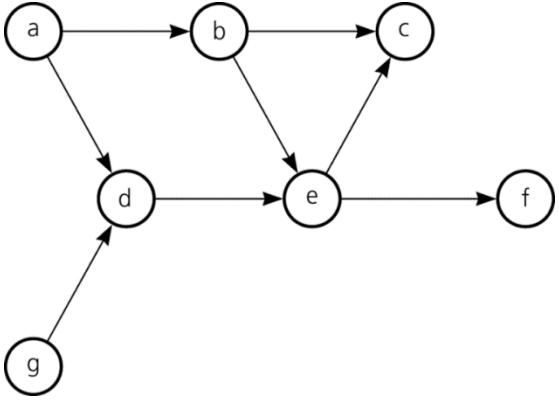
	DFS	BFS
Data structure	stack	queue
Vertex orderings	2	1
Edge types (undirected graphs)	tree edges, and back edges	tree edges, and cross edges
Applications	connectivity, acyclicity, articulation points	connectivity, acyclicity, min-edge path
Efficiency (adj matrix)	$\Theta(\|V\|^2)$	$\Theta(\|V\|^2)$
Efficiency (adj list)	$\Theta(\|V\|+\|E\|)$	$\Theta(\|V\|+\|E\|)$

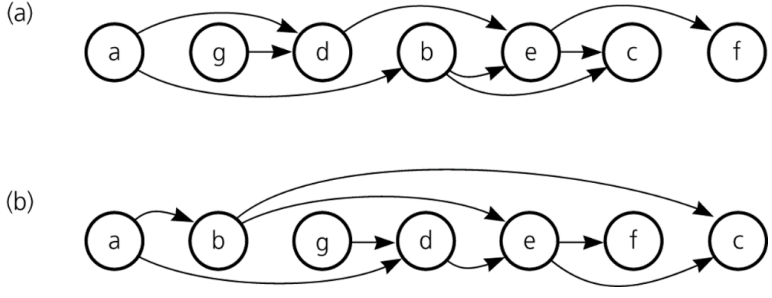
TOPOLOGICAL SORT - INTRODUCTION 1

A **topological order** of a **directed graph G** is a vertex ordering such that: vertex **a** precedes vertex **b** in the sorting if there is a directed edge **(a, b)** in **G**.

Note: There may be **several topological orders** for a digraph **G** (left): e.g. (a) and (b).

A **topological sorting** is an algorithm to sort a digraph vertices in topological order.





TOPOLOGICAL SORT - INTRODUCTION 2

Note: The **topological sorting** cannot be solved if a digraph has a **directed cycle**.

Note: The **topological sorting** can be solved if and only if the digraph is a **DAG**.

There are two different **algorithms to verify whether a digraph is a DAG**, and, if it is, produce an ordering of vertices that solves the topological sorting problem:

- **depth-first search**, and
- **source-removal**.

TOPOLOGICAL SORT - DFS-BASED 1

DFS-BFS can traverse digraphs as well, generating forests more complex than for undirected graphs. For example: this DFS forest shows all 4 possible types of edges.

- **tree edges:** (a,b) (b,c) (d,e)
  vertex → child
- **forward edges:** (a,c)
  vertex → non-child descendant
- **back edges:** (b,a)
  vertex → ancestor
- **cross edges:** (d,c)
  vertex → non ancestor/descendant

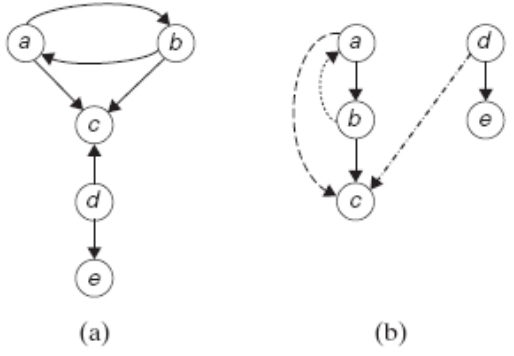


FIGURE 4.5 (a) Digraph. (b) DFS forest of the digraph for the DFS traversal started at a.

TOPOLOGICAL SORT - DFS-BASED 2

Note: A **back edge** in a DFS forest of a directed graph can connect a vertex to its parent, and its presence indicates that the digraph has a **directed cycle**.

A **directed cycle** is a sequence of 3 or more vertices that starts and ends with the same vertex, in which every vertex is connected to its immediate predecessor by an edge: **predecessor → successor**.

If a DFS forest of a digraph has no back edges, the digraph is a DAG (directed acyclic graph).

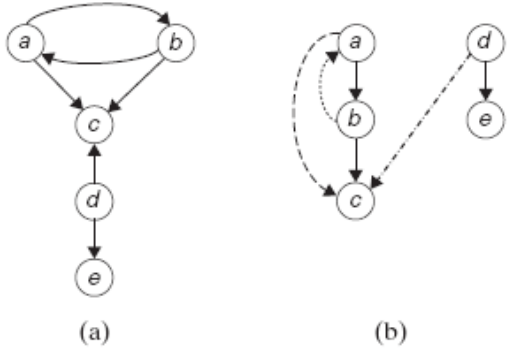


FIGURE 4.5 (a) Digraph. (b) DFS forest of the digraph for the DFS traversal started at a.

TOPOLOGICAL SORT - DFS-BASED 3

This algorithm verifies whether a digraph is a DAG by simply applying the DFS:

1. **Perform DFS traversal.**
2. **Store dead-end order** when vertices become dead-ends (pop off the stack).
3. **If no back edges is found: reverse dead-end order = topological sort.**

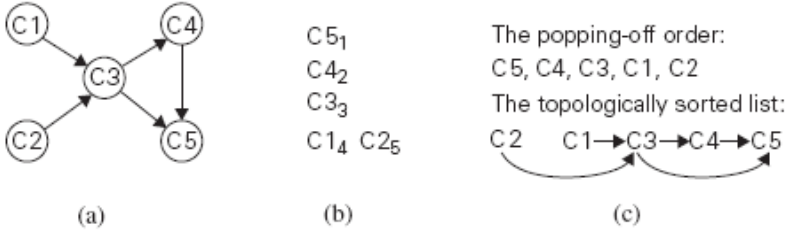


FIGURE 4.7 (a) Digraph for which the topological sorting problem needs to be solved. (b) DFS traversal stack with the subscript numbers indicating the popping-off order. (c) Solution to the problem.

TOPOLOGICAL SORT - DFS-BASED 4

Question: Why does the DFS solve the topological sorting problem?

Answer: When a vertex **v** is popped off a DFS stack, no vertex **u** with an edge **(u,v)** can be among the vertices popped off before **v**. Otherwise, **(u,v)** would have been a **back edge**. Hence, any such vertex **u** will be listed after **v** in the popped-off order list, and before **v** in the reversed list.

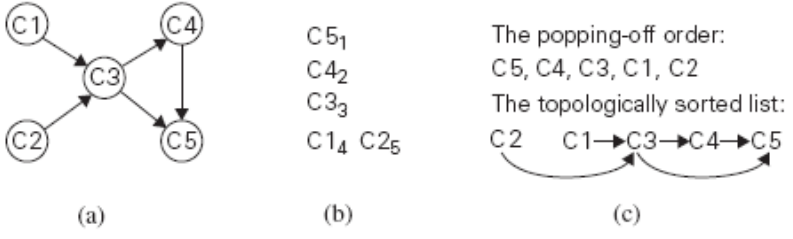


FIGURE 4.7 (a) Digraph for which the topological sorting problem needs to be solved. (b) DFS traversal stack with the subscript numbers indicating the popping-off order. (c) Solution to the problem.

TOPOLOGICAL SORT - SOURCE REMOVAL 1

Source-removal verifies if a digraph is a DAG by applying a **decrease-by-1** approach (at each step digraph is smaller by 1 vertex):

1. **Identify** in the current digraph a **source: a vertex with no incoming edges**.
2. **Delete the source** along with all its outgoing edges.
3. **Repeat step 1** (and then step 2).
4. **If there is no source**, stop because the problem cannot be solved,
5. **If digraph is empty: order of deletion = topological sort.**

Note: Topological sort by source-removal could be different from DFS-based.

TOPOLOGICAL SORT - SOURCE REMOVAL 2

Example: The source-removal algorithm processing a digraph, by deleting at each step a source vertex with all its outgoing edges, and generating at the end a topological order of the vertices of the digraph (a DAG in this case).

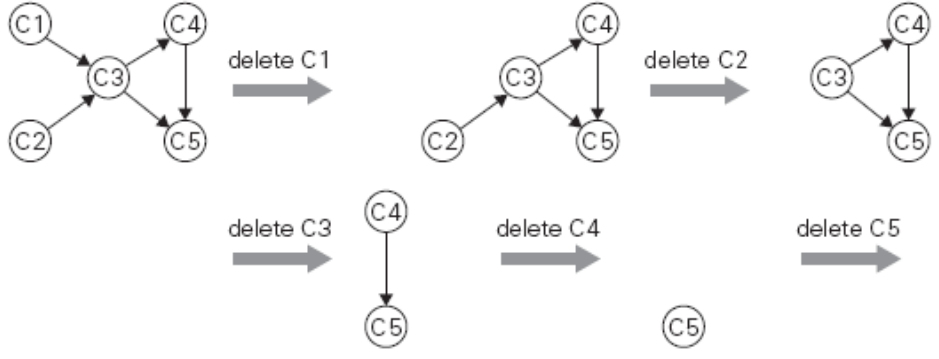


FIGURE 4.8 Illustration of the source-removal algorithm for the topological sorting problem. On each iteration, a vertex with no incoming edges is deleted from the digraph.