DATA STRUCTURE AND ALGORITHM

Dr. Khine Thin Zar
Professor
Computer Engineering and Information Technology Dept.
Yangon Technological University

Lecture 11

Graphs

Outlines of Class (Lecture 11)

- Introduction
- Applications of Graphs
- Terminology and Representations
- Directed vs. Undirected Graphs
- Representation for Graphs
- The Graph ADT
- Graph Traversals
- Shortest Paths
- Minimum-Cost Spanning Trees

Introduction

- Graph
  - ✓ a way of representing relationships that exist between pairs of objects.
  - ✓ can model both real-world systems and abstract problems
- Subgraph
  - ✓ consists of a subset of a graph's vertices and a subset of its edges

Applications of Graphs

- Modeling connectivity in computer and communications networks.
- Representing a map as a set of locations with distances between locations; used to compute shortest routes between locations.
- Modeling flow capacities in transportation networks.
- Finding a path from a starting condition to a goal condition; for example, in artificial intelligence problem solving.
- Modeling computer algorithms, showing transitions from one program state to another.
- Finding an acceptable order for finishing subtasks in a complex activity, such as constructing large buildings.
- Modeling relationships such as family trees, business or military organizations, and scientific taxonomies.

Terminology and Representations

- A graph G is a set of vertices and edges, which is defined as follows:

$$G = (V,E)$$

V(G): a finite, nonempty set of vertices

E(G): a set of edges

Terminology and Representations (Cont.)

- neighbors: Two vertices are adjacent if they are joined by an edge
- incident: An edge connecting Vertices U and V
- weight: Associated with each edge
- path: A sequence of vertices v1, v2, ..., vn
- length: the number of edges it contains
- cycle: path of length three or more that connects some vertex v1 to itself
- connected: An undirected graph there is at least one path from any vertex to any other
- acyclic: A graph without cycles
- directed acyclic graph: a directed graph without cycles
- free tree: a connected, undirected graph with no simple cycles

Terminology and Representations (Cont.)

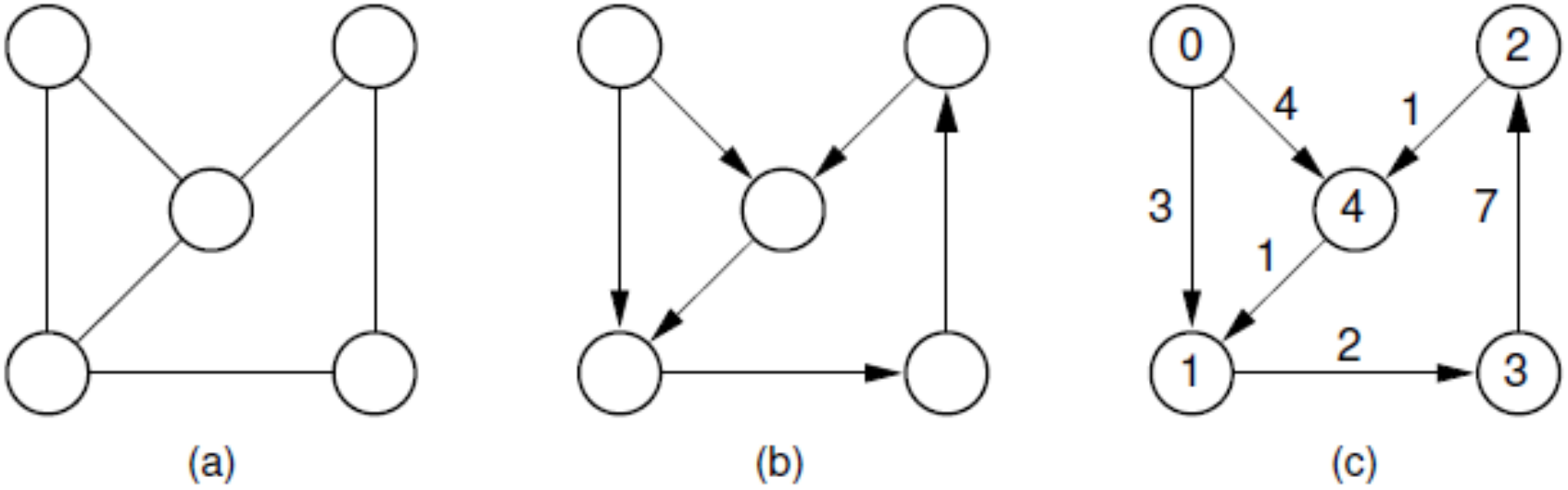


Figure: Examples of graphs and terminology. (a) A graph. (b) A directed graph (digraph). (c) A labeled (directed) graph with weights associated with the edges.

Terminology and Representations (Cont.)

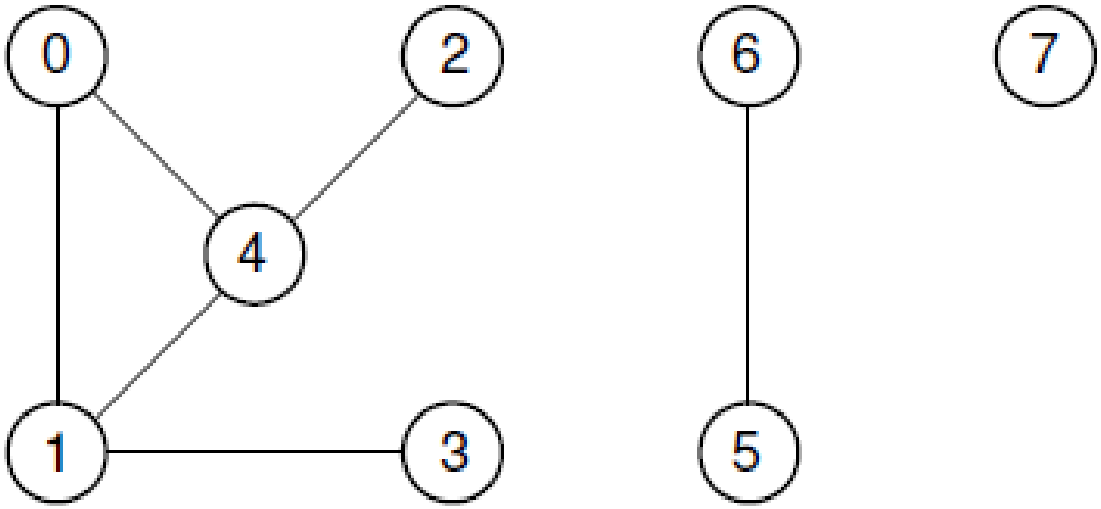


Figure 11.2 An undirected graph with three connected components. Vertices 0, 1, 2, 3, and 4 form one connected component. Vertices 5 and 6 form a second connected component. Vertex 7 by itself forms a third connected component.

Directed vs. Undirected Graphs

- An edge (u,v) is said to be directed from u to v if the pair (u,v) is ordered, with u preceding v.
- Also called a digraph
- If the graph is directed, the order of the vertices is important

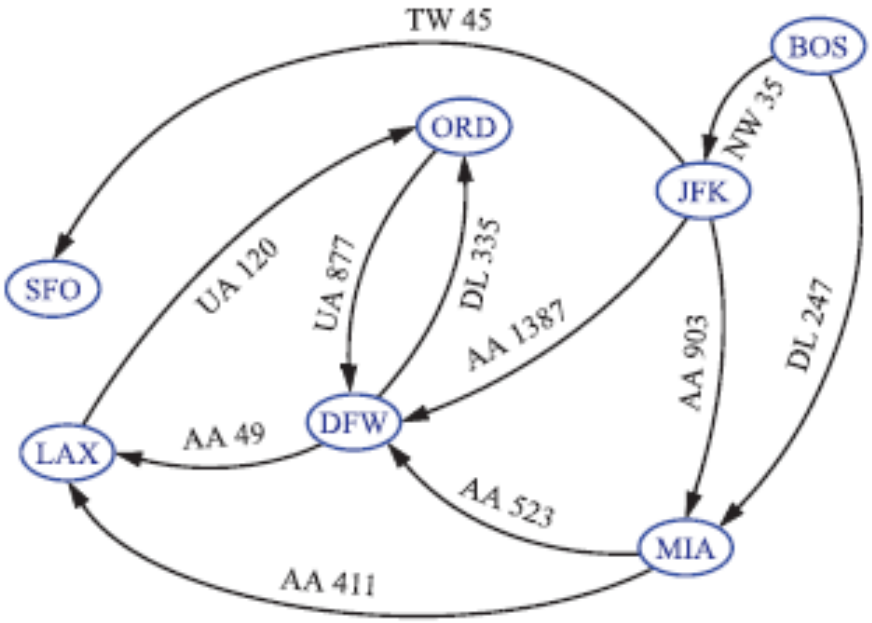


Figure: Example of a directed graph representing a flight network.

Directed vs. Undirected Graphs (Cont.)

- An edge (u,v) is said to be undirected if the pair (u,v) is not ordered (not directed).

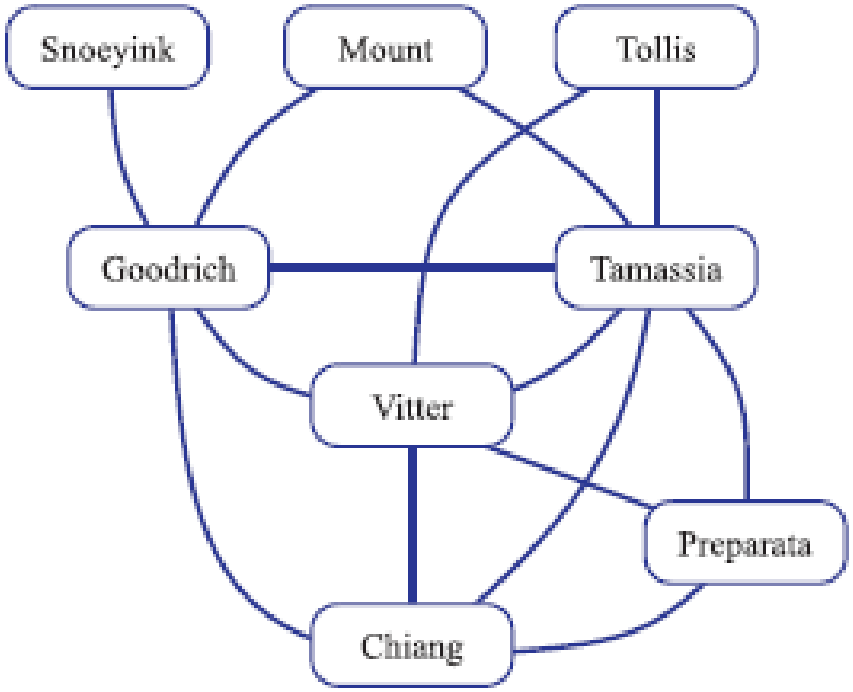


Figure: Graph of coauthorship among some authors.

Weighted Graph

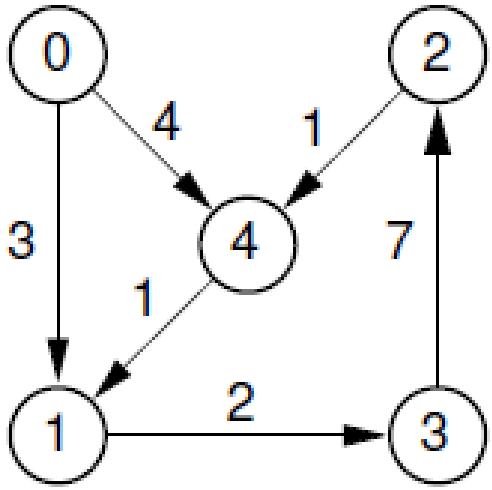
- A weighted graph is defined as follows:

  a triple (V, E, W)

  (V, E) : a graph (directed or undirected)

  W : a function from E into R,

- For an edge e, W(e) is called the weight of e.



Representation for Graphs

- Two commonly used methods for representing graphs
  - ✓ The adjacency matrix
  - ✓ The adjacency list

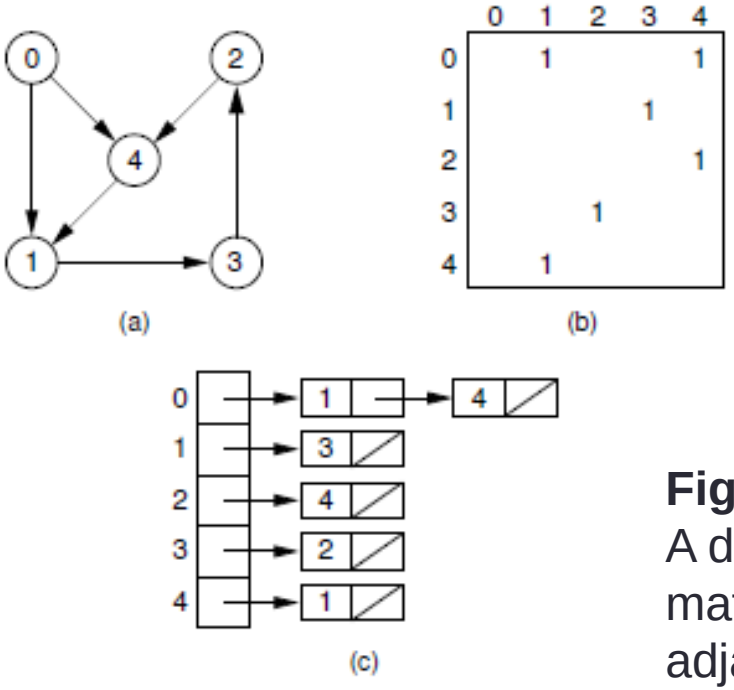


Figure: Two graph representations. (a) A directed graph. (b) The adjacency matrix for the graph of (a). (c) The adjacency list for the graph of (a).

Representation for Graphs (Cont.)

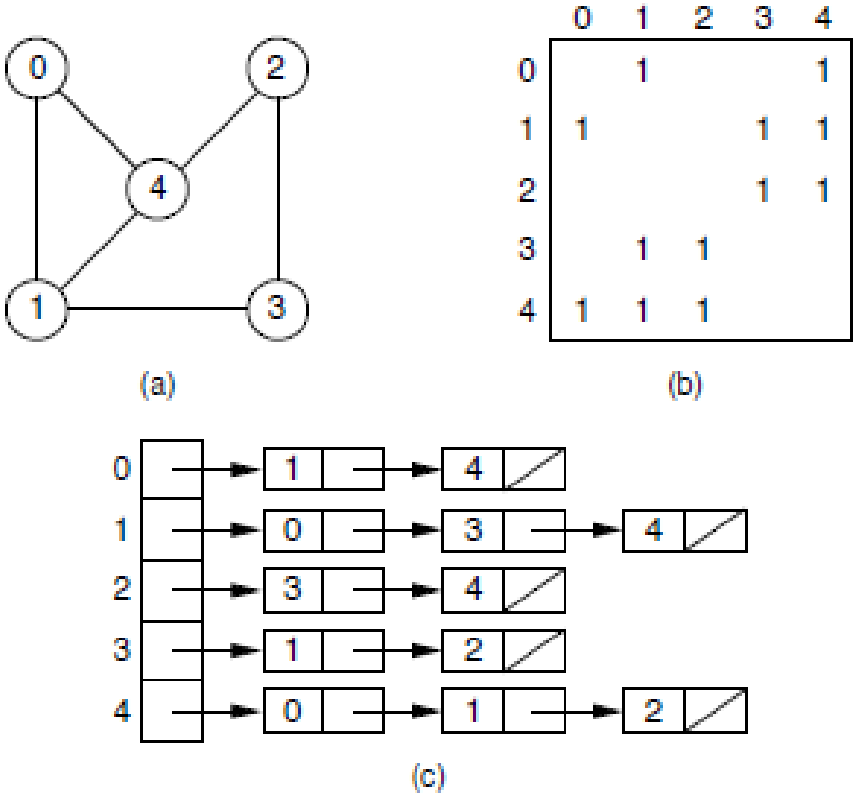


Figure: Using the graph representations for undirected graphs. (a) An undirected graph. (b) The adjacency matrix for the graph of (a). (c) The adjacency list for the graph of (a).

The Adjacency Matrix

- Adjacency matrix: $|V| \times |V|$ array
- Assume that $|V| = n$ and that the vertices are labeled from v_0 through v_{n-1}.
- The adjacency matrix requires one bit at each position.
- Space requirements: $O(|V| + |E|) = O(|V|^2)$
- Connectivity between two vertices can be tested quickly

The Adjacency List

- Adjacency list: an array of linked lists
- The array is |V| items long, with position i storing a pointer to the linked list of edges for Vertex v_i.
- Represents the edges by the vertices that are adjacent to Vertex v_i.
- A generalization of the "list of children" representation for trees
- Space requirements: O(|V| + |E|) = O(|V|)
- Vertices adjacent to another vertex can be found quickly

The Graph ADT

- Suitable for undirected graphs, that is, graphs whose edges are all undirected.
- As an abstract data type,
  - ✓ a graph is a collection of elements that are stored at the graph's positions—its vertices and edges.
  - ✓ can store elements in a graph at either its edges or its vertices (or both).
  - ✓ The graph ADT defines two types, Vertex and Edge.
  - ✓ It also provides two list types for storing lists of vertices and edges, called VertexList and EdgeList,

The Graph ADT (Cont.)

- operator*(): Return the element associated with u.
- incidentEdges(): Return an edge list of the edges incident on u.
- isAdjacentTo(v): Test whether vertices u and v are adjacent.
- operator*(): Return the element associated with e.
- endVertices(): Return a vertex list containing e's end vertices.
- opposite(v): Return the end vertex of edge e distinct from vertex v; an error occurs if e is not incident on v.
- isAdjacentTo(f): Test whether edges e and f are adjacent.
- isIncidentOn(v): Test whether e is incident on v.

The Graph ADT (Cont.)

- vertices(): Return a vertex list of all the vertices of the graph.
- edges(): Return an edge list of all the edges of the graph.
- insertVertex(x): Insert and return a new vertex storing element x.
- insertEdge(v,w,x): Insert and return a new undirected edge with end vertices v and w and storing element x.
- eraseVertex(v): Remove vertex v and all its incident edges.
- eraseEdge(e): Remove edge e.

Graph Traversals

- Visit the vertices of a graph in some specific order based on the graph's topology.
- Begin with a start vertex and attempt to visit the remaining vertices from there
- May not be possible to reach all vertices from the start vertex (occurs when the graph is not connected)
- May contain cycles (make sure that cycles do not cause the algorithm to go into an infinite loop)

Depth-First Search (DFS)

- Used to define a depth-first search tree
- DFS can be applied to directed or undirected graphs.
- Whenever a vertex V is visited during the search, DFS will recursively visit all of V's unvisited neighbors
  - Add all edges leading out of v to a stack.
  - The next vertex to be visited is determined by popping the stack and following that edge.

The DFS algorithm

```
Algorithm DFS(G,v):
    Input: A graph G and a vertex v of G
    Output: A labeling of the edges in the connected component of v as
                discovery edges and back edges
     label v as visited
     for all edges e in v.incidentEdges() do
          if edge e is unvisited then
            w←e.opposite(v)
            if vertex w is unexplored then
                        label e as a discovery edge
                        recursively call DFS(G,w)
            else
                        label e as a back edge
```

Depth-First Search (DFS) (Cont.)

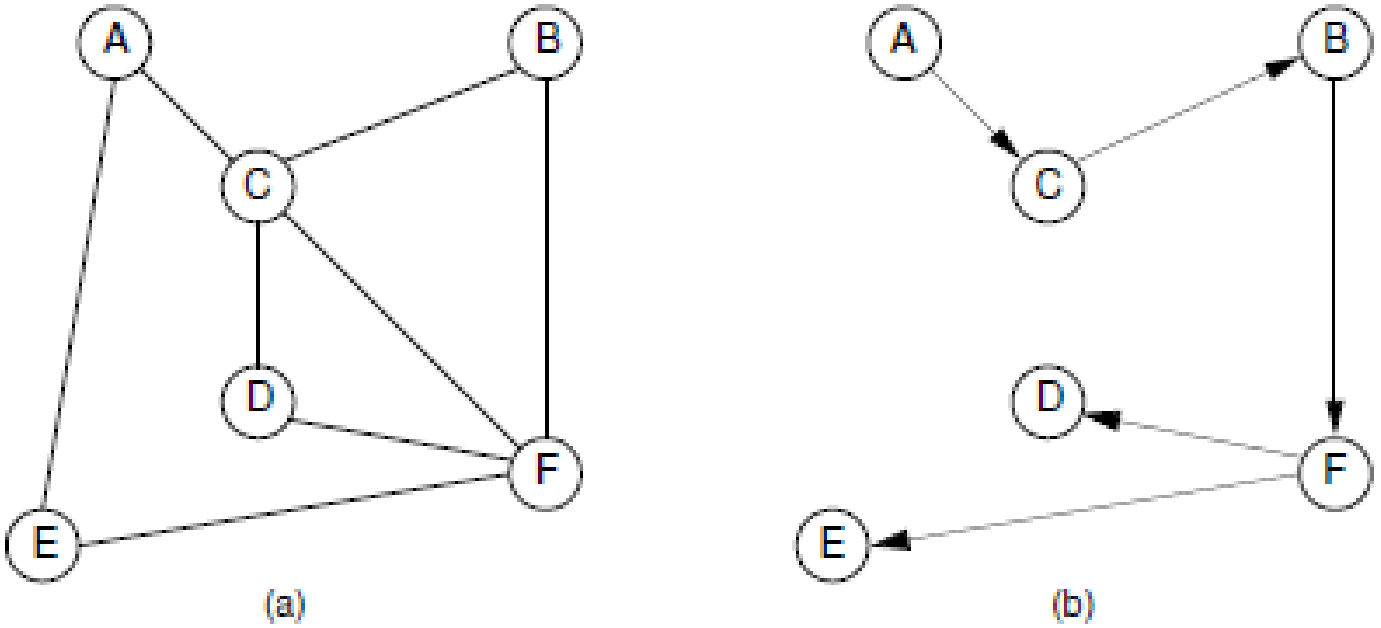


Figure: (a) A graph. (b) The depth-first search tree for the graph when starting at Vertex A.

Depth-First Search (DFS) (Cont.)

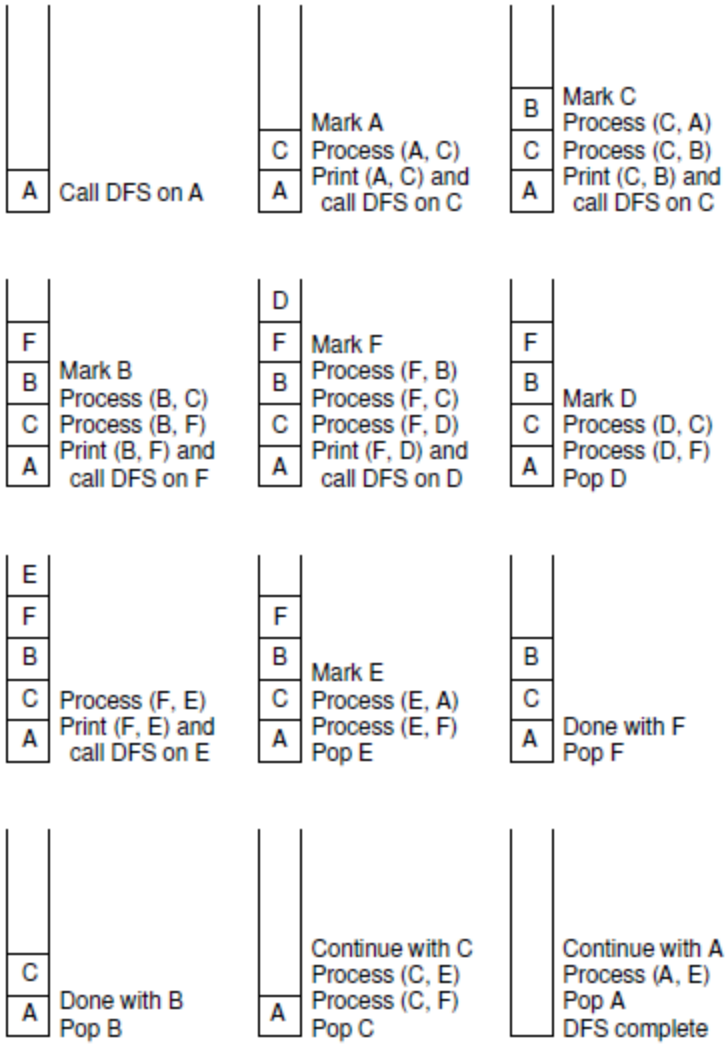


Figure: A detailed illustration of the DFS process for the graph

Breadth-First Search (BFS)

- BFS examines all vertices connected to the start vertex before visiting vertices further away
- Similarly to DFS, except that a queue replaces the recursion stack.
- If the graph is a tree and the start vertex is at the root, BFS is equivalent to visiting vertices level by level from top to bottom.

The BFS algorithm

```
Algorithm BFS(s):
    initialize collection L0 to contain vertex s
    i←0
    while Li is not empty do
        create collection Li+1 to initially be empty
        for all vertices v in Li do
            for all edges e in v.incidentEdges() do
                if edge e is unexplored then
                    w←e.opposite(v)
                    if vertex w is unexplored then
                        label e as a discovery edge
                        insert w into Li+1
                    else
                        label e as a cross edge
    i←i+1
```

Breadth-First Search (BFS) (Cont.)

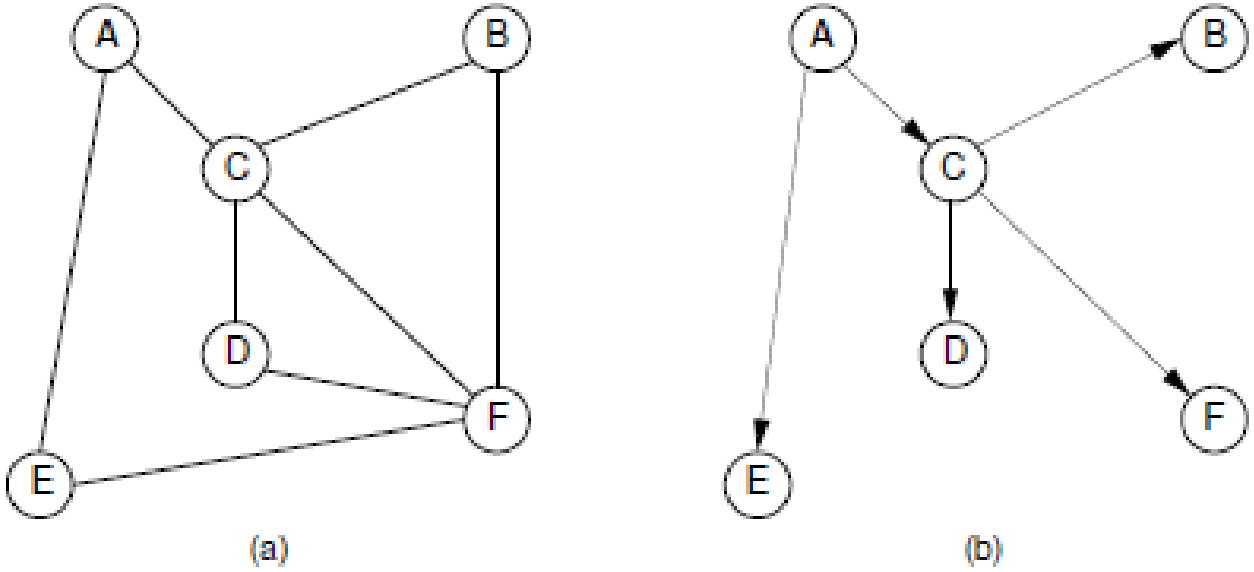
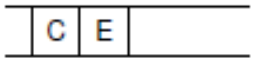


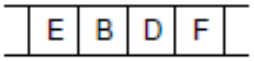
Figure: (a) A graph. (b) The breadth-first search tree for the graph when starting at Vertex A.

Breadth-First Search (BFS) (Cont.)

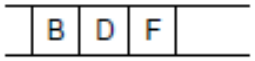
Initial call to BFS on A.
Mark A and put on the queue.



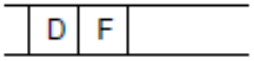
Dequeue A.
Process (A, C).
Mark and enqueue C. Print (A, C)
Process (A, E).
Mark and enqueue E. Print(A, E).



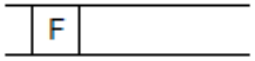
Dequeue C.
Process (C, A). Ignore.
Process (C, B).
Mark and enqueue B. Print (C, B).
Process (C, D).
Mark and enqueue D. Print (C, D).
Process (C, F).
Mark and enqueue F. Print (C, F).



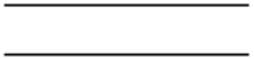
Dequeue E.
Process (E, A). Ignore.
Process (E, F). Ignore.



Dequeue B.
Process (B, C). Ignore.
Process (B, F). Ignore.



Dequeue D.
Process (D, C). Ignore.
Process (D, F). Ignore.



Dequeue F.
Process (F, B). Ignore.
Process (F, C). Ignore.
Process (F, D). Ignore.
BFS is complete.

Figure: A detailed illustration of the BFS process for the graph

Shortest Paths

- Can model a road network as a directed graph whose edges are labeled with real numbers.
- Numbers: the distance (or other cost metric, such as travel time) between two vertices.
- Labels : weights, costs, or distances, depending on the application.
- To find the total length of the shortest path between two specified vertices.

An implementation for Dijkstra's algorithm

```
// Compute shortest path distances from "s".
// Return these distances in "D".
void Dijkstra(Graph* G, int* D, int s) {
  int i, v, w;
  for (i=0; i<G->n(); i++) {       // Process the vertices
    v = minVertex(G, D);
    if (D[v] == INFINITY) return; // Unreachable vertices
    G->setMark(v, VISITED);
    for (w=G->first(v); w<G->n(); w = G->next(v,w))
      if (D[w] > (D[v] + G->weight(v, w)))
        D[w] = D[v] + G->weight(v, w);
  }
}
```

Shortest Paths (Cont.)

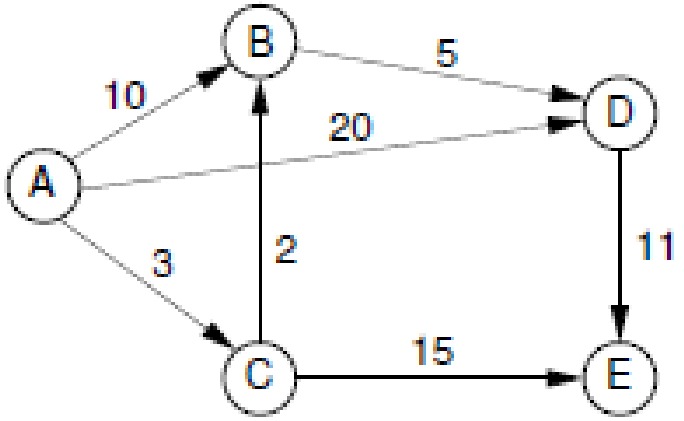


Figure: Example graph for shortest-path definitions.

- In Figure, there is no path from E to B
- Set d(E, B) = 1.
- Define w(A, D) = 20 to be the weight of edge (A, D), that is, the weight of the direct connection from A to D.

Minimum-Cost Spanning Trees

- Takes as input a connected, undirected graph G, where each edge has a distance or weight measure attached.
- The MST is the graph containing the vertices of G along with the subset of G's edges that
  - ✓ has minimum total cost as measured by summing the values for all of the edges in the subset, and
  - ✓ keeps the vertices connected. weight measure attached.
- Useful in soldering the shortest set of wires needed to connect a set of terminals on a circuit board, and connecting a set of cities by telephone lines in such a way as to require the least amount of cable.

Minimum-Cost Spanning Trees (Cont.)

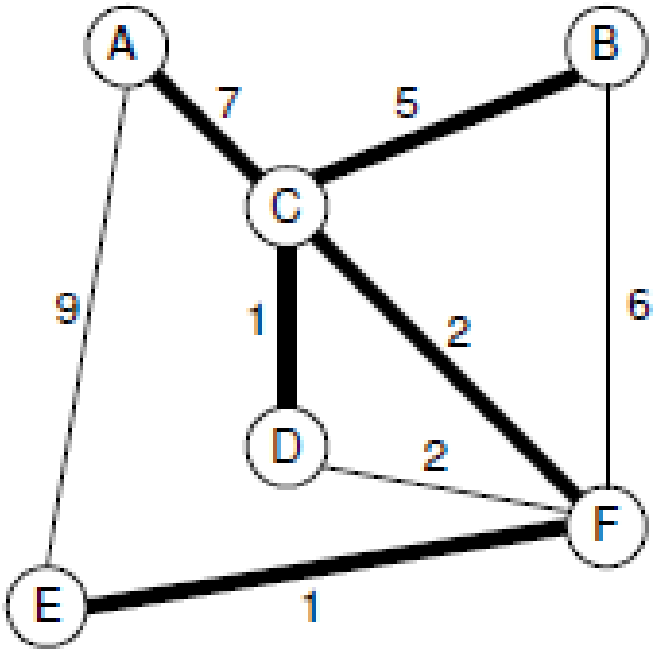


Figure: A graph and its MST. All edges appear in the original graph. Those edges drawn with heavy lines indicate the subset making up the MST. Note that edge (C, F) could be replaced with edge (D, F) to form a different MST with equal cost.

Next Week Lecture (Week 12)

Lecture 12: Advanced Tree Structures

Thank you!