Graphs
Lecture 15

Introduction

- Over the past 200 years, graph theory has been applied to a variety of problems, including:
  - Model electrical circuits, chemical compounds, highway maps, etc.
  - Analysis of electrical circuits, finding the shortest route, project planning, linguistics, genetics, social science, etc.

Graph Definitions

- Graph *G: G* = (*V*, *E*)
  - *V* is a finite nonempty set of vertices of *G*
  - $E \subseteq V \times V$
  - Elements in *E* are the pairs of elements of *V*
  - *E* is called set of edges

Graph Definitions and Notations (cont'd.)

- Directed graph or digraph: elements of $E(G)$ are ordered pairs
- Undirected graph: elements not ordered pairs
- If (u, v) is an edge in a directed graph
  - Origin: u
  - Destination: v
- Subgraph *H of G:* if $V(H) \subseteq V(G)$ and $E(H) \subseteq E(G)$
  - Every vertex and edge of V is in G

Graph Definitions and Notations (cont'd.)

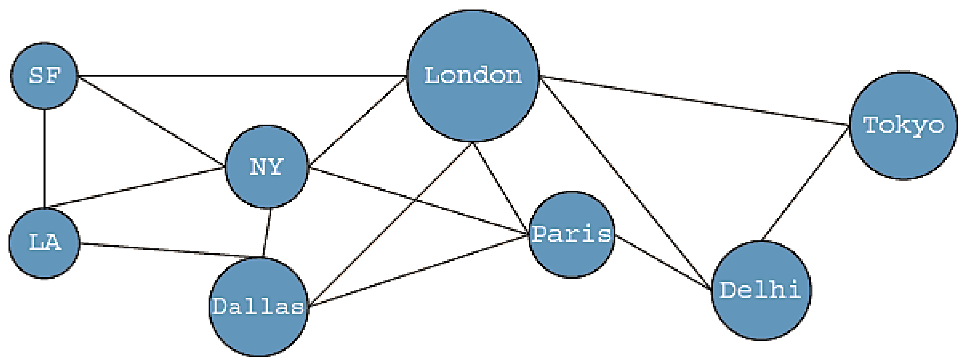


FIGURE 20-3 Various undirected graphs

Graph Definitions and Notations (cont'd.)

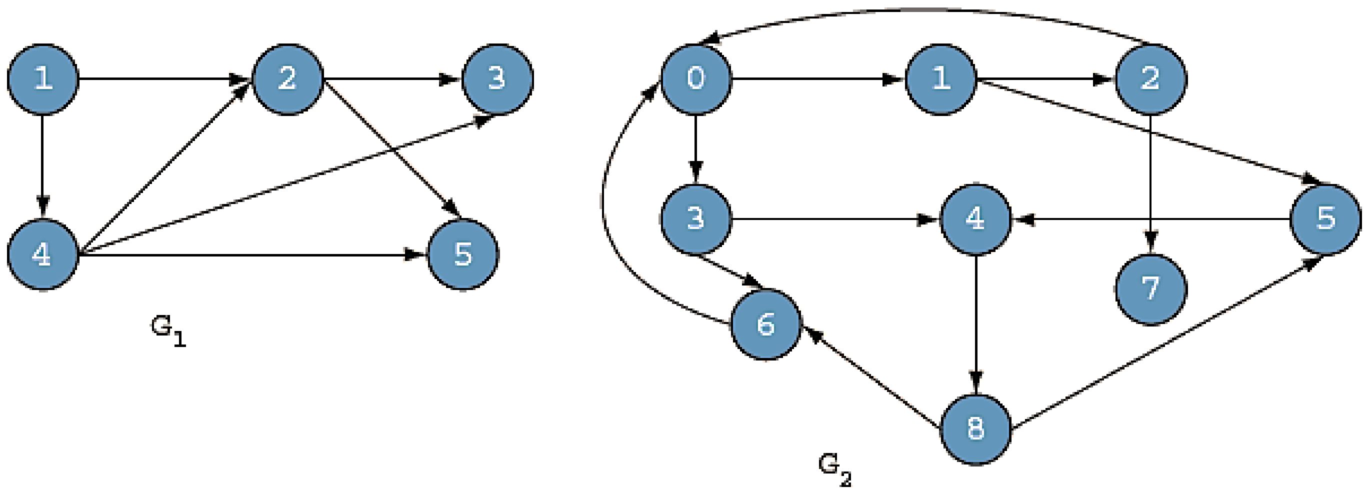


FIGURE 20-4 Various directed graphs

Graph Definitions and Notations (cont'd.)

- Adjacent: there is an edge from one vertex to the other; i.e., $(u, v) \in E(G)$
- Incident: if edge $e = (u, v)$ then e is incident on u and v
  - Loop: edge incident on a single vertex
- Parallel edges: associated with the same pair of vertices
- Simple graph: has no loops or parallel edges

Graph Definitions and Notations (cont'd.)

- Path: sequence of vertices u_1, u_2, ..., u_n such that $u = u_1$, $u_n = v$, and (u_i, u_{i+1}) is an edge for all $i = 1, 2, ..., n - 1$
- Connected vertices**:** there is a path from u to v
- Simple path: path in which all vertices, except possibly the first and last, are distinct
- Cycle: simple path in which the first and last vertices are the same

Graph Definitions and Notations (cont'd.)

- Connected: path exists from any vertex to any other vertex
  - Component: maximal subset of connected vertices
- In a connected graph G, if there is an edge from u to v, i.e., $(u, v) \in E(G)$, then u is adjacent to v and v is adjacent from u
- Strongly connected: any two vertices in G are connected

Graph Representation

- To write programs that process and manipulate graphs
  - Must store graphs in computer memory
- A graph can be represented in several ways:
  - Adjacency matrices
  - Adjacency lists

Adjacency Matrix

- *G:* graph with *n* vertices ($n > 0$)
  - $V(G) = \{v_1, v_2, \ldots, v_n\}$
- <u>Adjacency matrix</u> (A_G of *G*): two-dimensional $n \times n$ matrix such that:

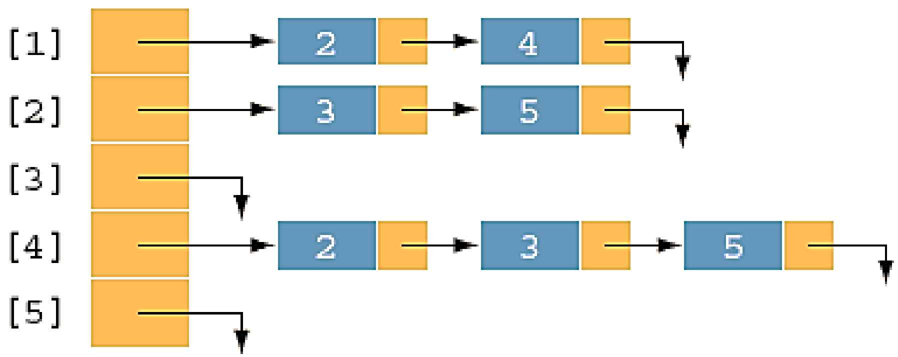
$$A_G(i,j) = \begin{cases} 1 & \text{if } (\nu_i, \nu_j) \in E(G) \\ 0 & \text{otherwise} \end{cases}$$

- A ymmetric

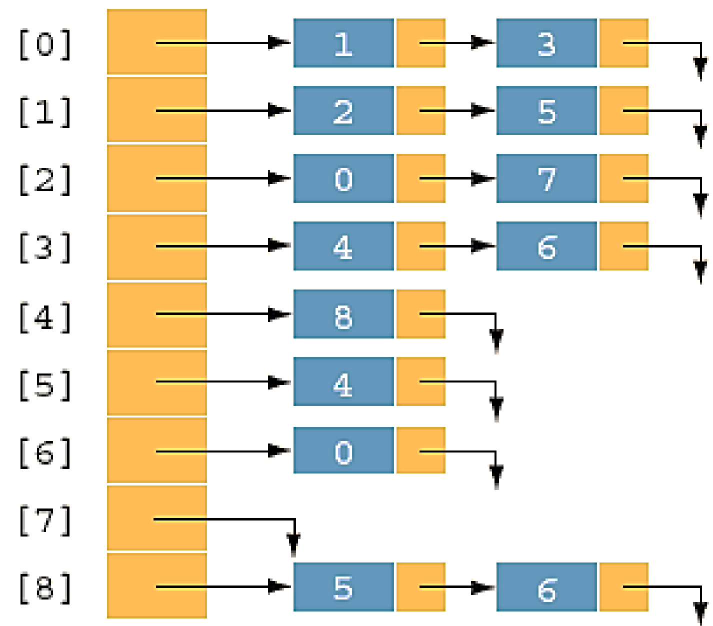
Adjacency Lists

- *G:* graph with *n* vertices ($n > 0$)
  - $V(G) = \{v_1, v_2, \ldots, v_n\}$
- Linked list corresponding to each vertex, *v*,
  - Each node of linked list contains the vertex, *u*, such that $(u,v) \in E(G)$
  - Each node has two components, such as `vertex` and `link`

Adjacency Lists (cont'd.)



Adjacency list of graph G_1 of Figure 20-4



Adjacency list of graph G_2 of Figure 20-4

FIGURE 20-5 Adjacency list of graphs of Figure 20-4

Operations on Graphs

- Operations commonly performed on a graph:
  - Create the graph
  - Clear the graph
    - Makes the graph empty
  - Determine whether the graph is empty
  - Traverse the graph
  - Print the graph

Graphs as ADTs

- We implement graphs as an abstract data type (ADT), including functions to:
  - Create/clear the graph
  - Print the graph
  - Traverse the graph
  - Determine the graph’s size

Graph Traversals

- Traversing a graph is similar to traversing a binary tree, except that:
  - A graph might have cycles
  - Might not be able to traverse the entire graph from a single vertex
- Most common graph traversal algorithms:
  - Depth first traversal
  - Breadth first traversal

Depth First Traversal

- Depth first traversal at a given node, *v*:
  - Mark node `v` as visited
  - Visit the node
  - `for` each vertex `u` adjacent to `v`

    `if` `u` is not visited

    start the depth first traversal at `u`
- This is a recursive algorithm

Depth First Traversal (cont'd.)

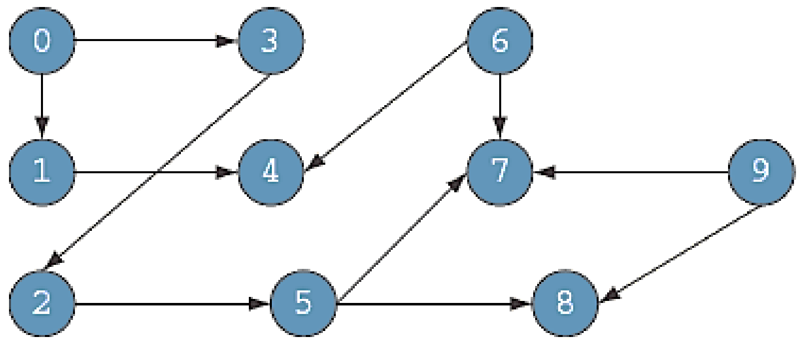


FIGURE 20-6 Directed graph *G*

- 0, 1, 4, 3, 2, 5, 7, 8, 6, 9

- Breadth-first ordering of vertices:
  - 0, 1, 3, 4, 2, 5, 7, 8, 6, 9

Breadth First Traversal

- Breadth first traversal of a graph
  - Similar to traversing a binary tree level by level
  - Nodes at each level are visited from left to right
- Starting at the first vertex, the graph is traversed as much as possible
  - Then go to next vertex not yet visited
- Use a queue to implement the breadth first search algorithm

Shortest Path Algorithm

- Weight of the edge: nonnegative real number assigned to the edges connecting two vertices
- Weighted graph: every edge has a nonnegative weight
- Weight of the path *P*
  - Sum of the weights of all edges on the path *P*
  - Also called the weight of *v* from *u* via *P*
- Source: starting vertex in the path

Shortest Path Algorithm (cont'd.)

- <u>Shortest path</u>: path with the smallest weight
- <u>Shortest path algorithm</u>
  - Called the <u>greedy algorithm</u>, developed by Dijkstra
  - *G*: graph with *n* vertices, where $n \geq 0$
  - $V(G) = \{v_1, v_2, ..., v_n\}$
  - *W*: two-dimensional $n \times n$ matrix

$$W(i,j) = \begin{cases} w_{ij} & \text{if } (v_i, v_j) \text{ is an edge in } G \text{ and } w_{ij} \text{ is the weight of the edge } (v_i, v_j) \\ \infty & \text{if there is no edge from } v_i \text{ to } v_j \end{cases}$$

Shortest Path Algorithm (cont'd.)

- Shortest path algorithm:

1. Initialize the array `smallestWeight` so that:

   `smallestWeight[u] = weights[vertex, u]`
2. Set `smallestWeight[vertex]` = `0`.
3. Find the vertex, `v`, that is closest to the `vertex` for which the shortest path has not been determined.
4. Mark `v` as the (next) vertex for which the smallest weight is found.
5. For each vertex `w` in `G`, such that the shortest path from `vertex` to `w` has not been determined and an edge `(v, w)` exists, if the weight of the path to `w` via `v` is smaller than its current weight, update the weight of `w` to the weight of `v` + the weight of the edge `(v, w)`.

Shortest Path Algorithm (cont'd.)

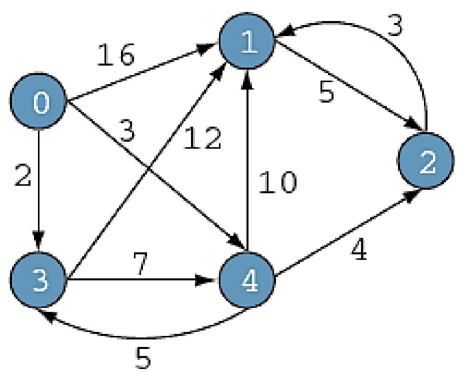
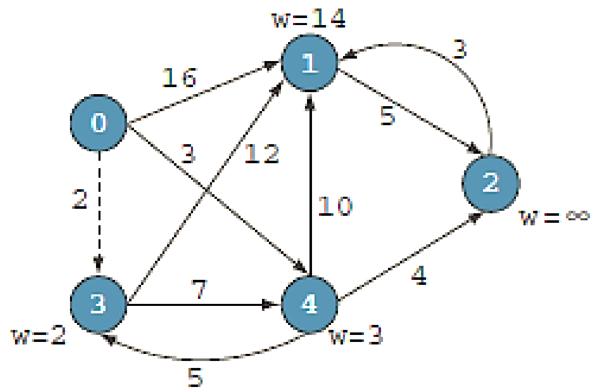


FIGURE 20-7 Weighted graph G

Shortest Path Algorithm (cont'd.)

- Graph after first iteration of Steps 3, 4, and 5



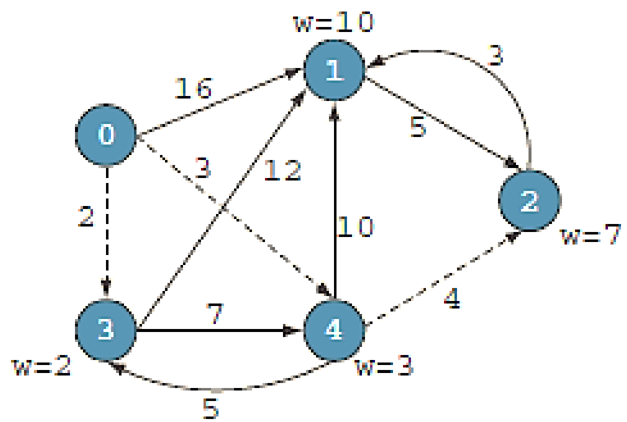
	[0]	[1]	[2]	[3]	[4]
smallestWeight	0	14	∞	2	3

	[0]	[1]	[2]	[3]	[4]
weightFound	T	F	F	T	F

FIGURE 20-9 Graph after the first iteration of Steps 3, 4, and 5

Shortest Path Algorithm (cont'd.)

- Graph after third iteration of Steps 3, 4, and 5



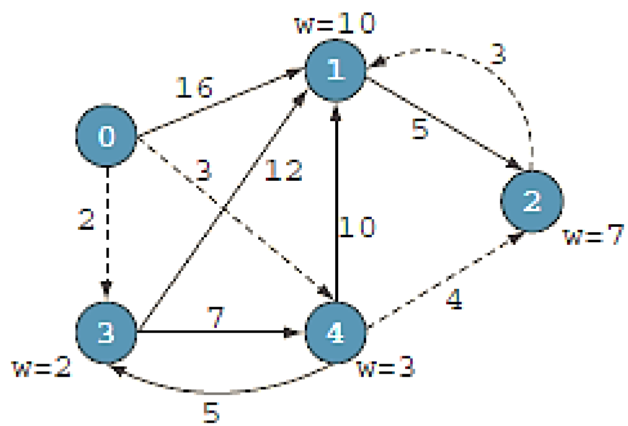
	[0]	[1]	[2]	[3]	[4]
smallestWeight	0	10	7	2	3

	[0]	[1]	[2]	[3]	[4]
weightFound	T	F	T	T	T

FIGURE 20-11 Graph after the third iteration of Steps 3, 4, and 5

Shortest Path Algorithm (cont'd.)

- Graph after fourth iteration of Steps 3, 4, and 5



	[0]	[1]	[2]	[3]	[4]
smallestWeight	0	10	7	2	3

	[0]	[1]	[2]	[3]	[4]
weightFound	T	T	T	T	T

FIGURE 20-12 Graph after the fourth iteration of Steps 3, 4, and 5

Minimal Spanning Tree

- Company needs to shut down a maximum number of connections and still be able to fly from one city to another

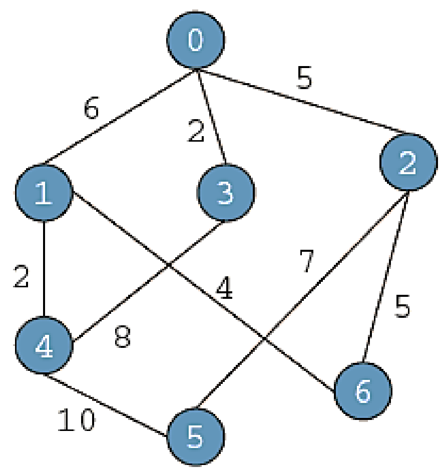


FIGURE 20-13 Airline connections between cities and the cost factor of maintaining the connections

Minimal Spanning Tree (cont'd.)

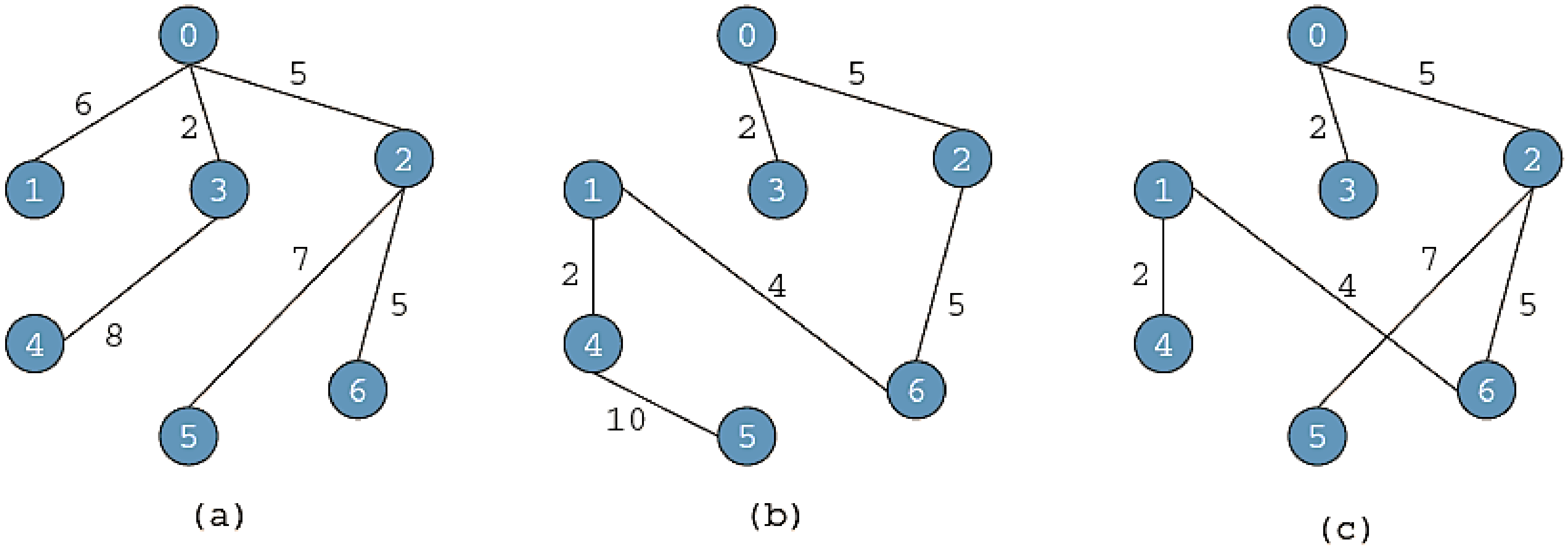


FIGURE 20-14 Possible solutions to the graph of Figure 20-13

Minimal Spanning Tree (cont'd.)

- Spanning tree of graph G: if T is a subgraph of G such that V(T) = V(G)
  - All the vertices of G are in T
  - Figure 20-14 shows three spanning trees of the graph shown in Figure 20-13
- Theorem: a graph G has a spanning tree if and only if G is connected
- Minimal spanning tree: spanning tree in a weighted graph with the minimum weight