Software Engineering 2 (C++)

CSY2006

Graphs

Objectives

- Learn about graphs
- Learn the basic terminology of graph theory
- Represent graphs in computer memory
- Explore graphs as ADTs

Objectives

- Examine and implement various graph traversal algorithms
- Implement the shortest path algorithm

Introduction

- Königsberg bridge problem:
  - The river Pregel flows around the island Kneiphof and then divides into two branches

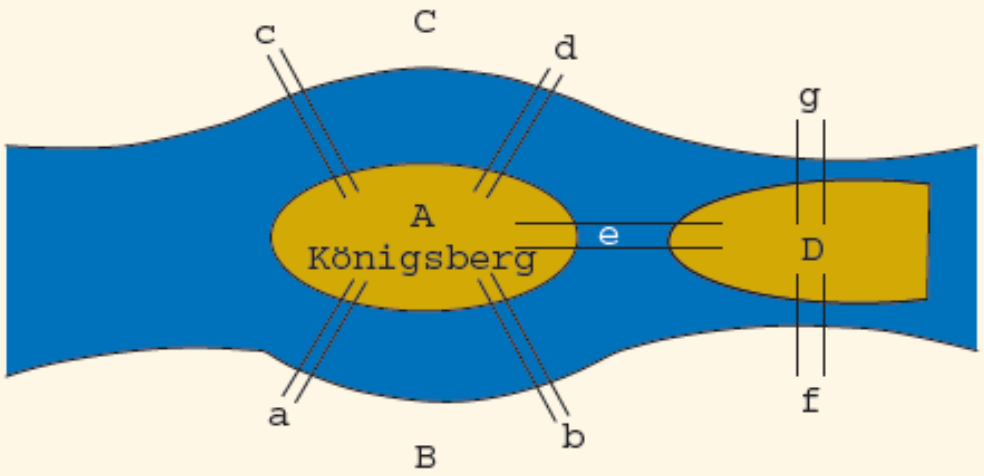


FIGURE 20-1 Königsberg bridge problem

Introduction

- Starting at one land area, can you cross all bridges exactly once and return to the start?
  - In 1736, Euler represented the problem as a graph and answered the question: No

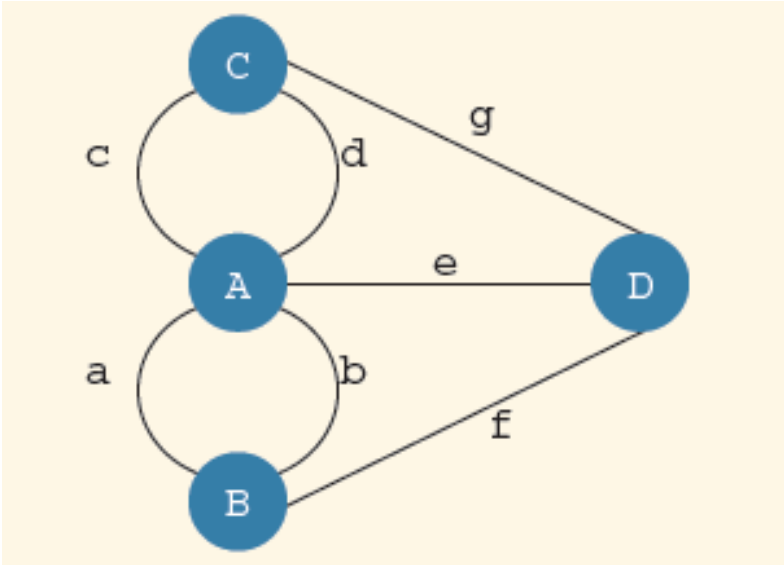
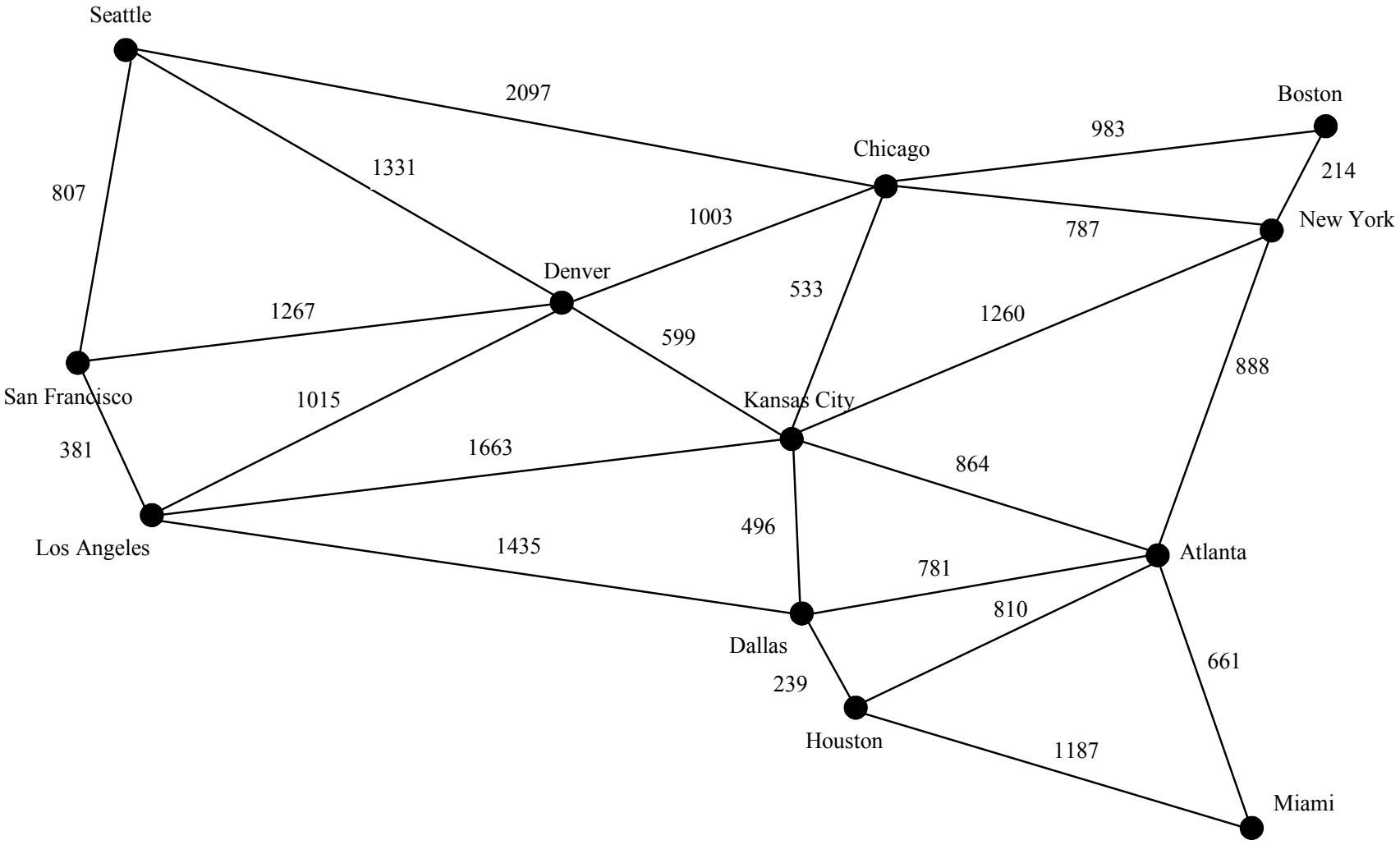


FIGURE 20-2 Graph representation of Königsberg bridge problem

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.

Modeling Using Graphs



Graph Definitions and Notations

- $a \in X$: a is an element of the set X
- Subset ($Y \subseteq X$): every element of Y is also an element of X
- Intersection ($A \cap B$): contains all the elements in A and B
  - $A \cap B = \{x \mid x \in A \text{ and } x \in B\}$
- Union ($A \cup B$): set of all the elements that are in A or in B
  - $A \cup B = \{x \mid x \in A \text{ or } x \in B\}$

Graph Definitions and Notations

- $A \times B$: set of all the ordered pairs of elements of A and B
  - $A \times B = \{(a, b) \mid a \in A, b \in B\}$
- 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

- 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 H is in G

Graph Definitions and Notations

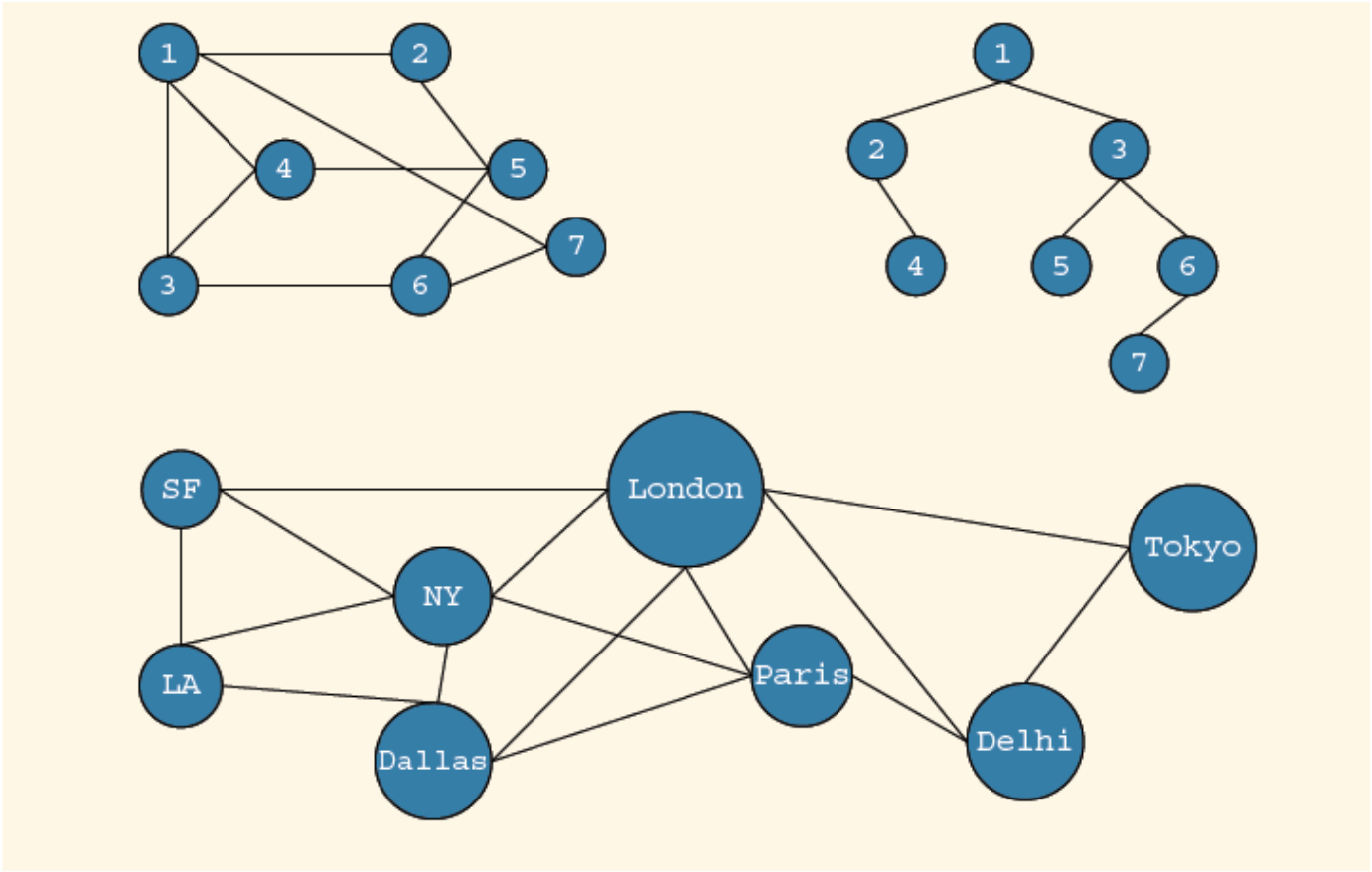


FIGURE 20-3 Various undirected graphs

Graph Definitions and Notations

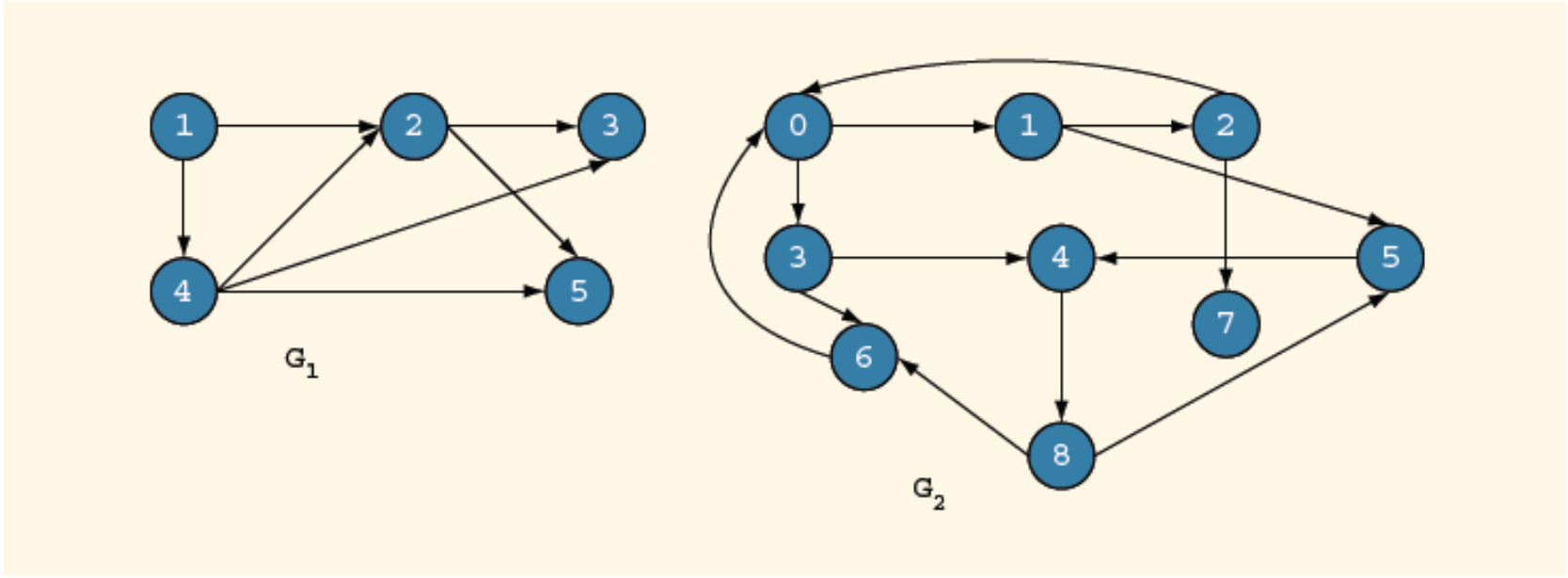


FIGURE 20-4 Various directed graphs

Graph Definitions and Notations

- 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

- Path: sequence of vertices $u_1, u_2, \ldots, u_n$ such that $u = u_1$, $u_n = v$, and (u_i, u_{i+1}) is an edge for all $i = 1, 2, \ldots, n - 1$
- 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

- 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 Representations

- 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

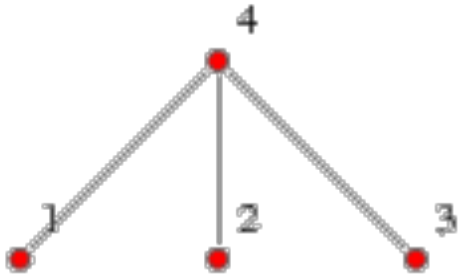
Adjacency Matrix

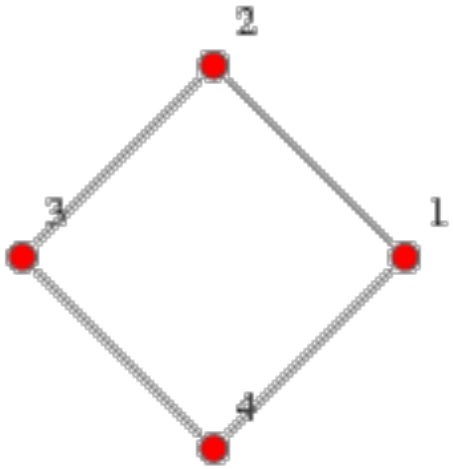
- *G:* graph with *n* vertices ($n > 0$)
  - $V(G) = \{v_1, v_2, \ldots, v_n\}$
- Adjacency matrix (A_G of *G)*: two-dimensional $n \times n$ matrix such that:

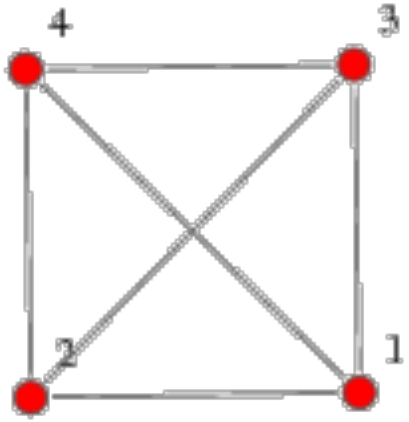
$$A_G(i,j) = \begin{cases} 1 & \text{if } (v_i, v_j) \in E(G) \\ 0 & \text{otherwise} \end{cases}$$

- Adjacency matrix of an undirected graph is symmetric

Adjacency Matrix







$$\begin{pmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{pmatrix}$$

$$\begin{pmatrix} 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 \end{pmatrix}$$

$$\begin{pmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{pmatrix}$$

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

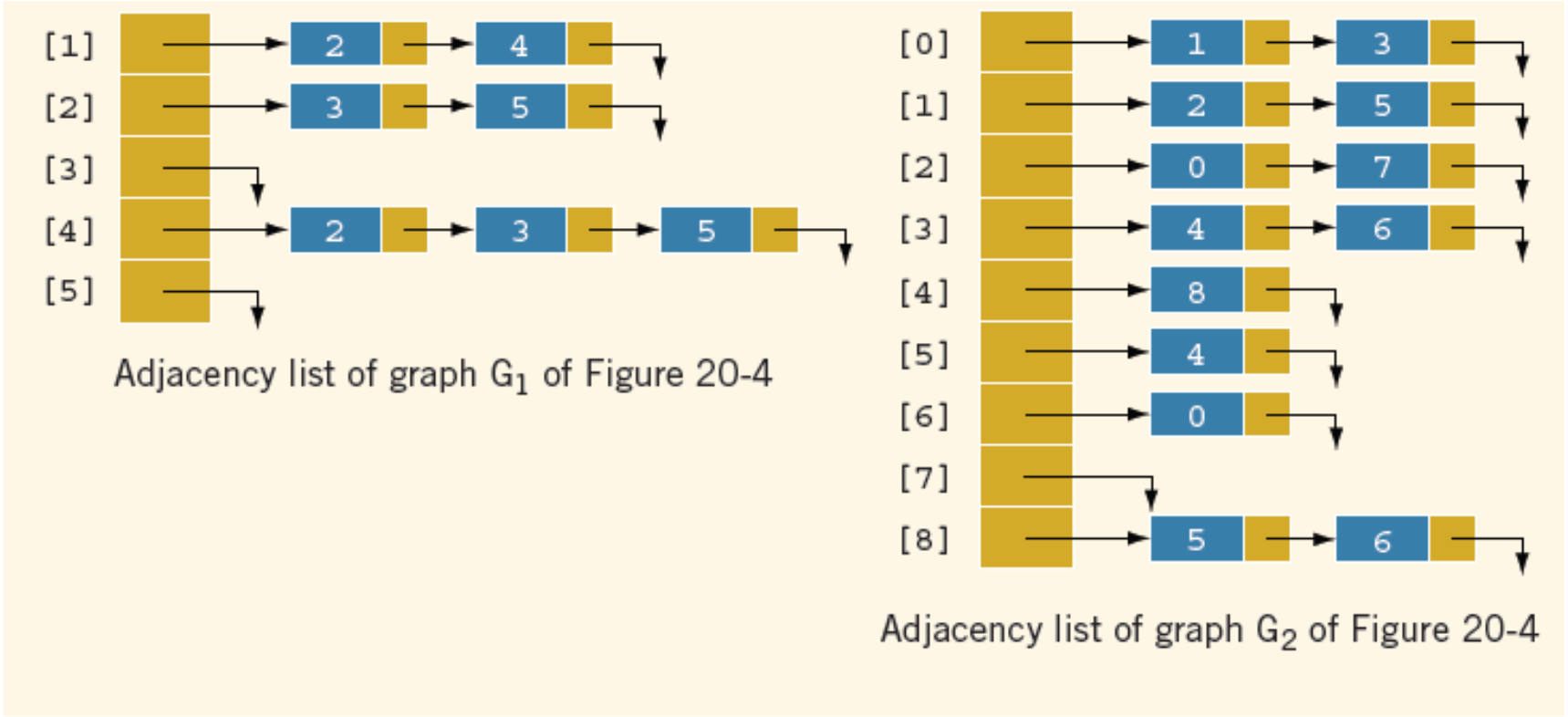


FIGURE 20-5 Adjacency list of graphs of Figure 20-4

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

Operations on Graphs

- The adjacency list (linked list) representation:
  - For each vertex, *v*, vertices adjacent to *v* are stored in linked list associated with *v*
    - In a directed graph, vertices adjacent to *v* are called immediate successors
  - To manage data in a linked list, use class `unorderedLinkedList`

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 Search

The depth-first search of a graph is like the depth-first search of a tree discussed. In the case of a tree, the search starts from the root. In a graph, the search can start from any vertex.

```
dfs(vertex v) {
  visit v;
  for each neighbor w of v
    if (w has not been visited)
{
      dfs(w);
    }
}
```

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

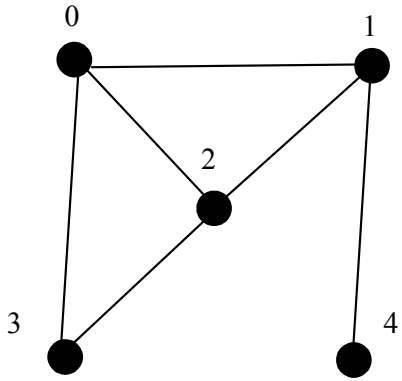
Breadth-First Search General Algorithm

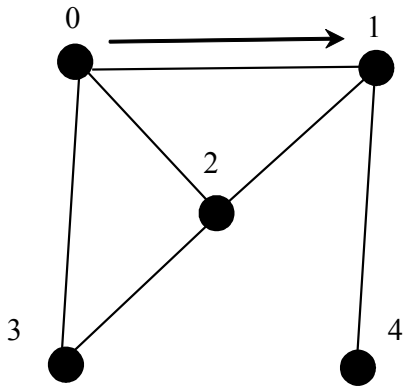
a. `for` each vertex `v` in the graph
   if `v` is not visited
      add `v` to the `queue`
b. Mark `v` as visited
c. `while` the `queue` is not empty
   c.1. Remove vertex `u` from the queue
   c.2. Retrieve the vertices adjacent to `u`
   c.3. `for` each vertex `w` that is adjacent to `u`
      if `w` is not visited
         c.3.1. Add `w` to the queue
         c.3.2. Mark `w` as visited

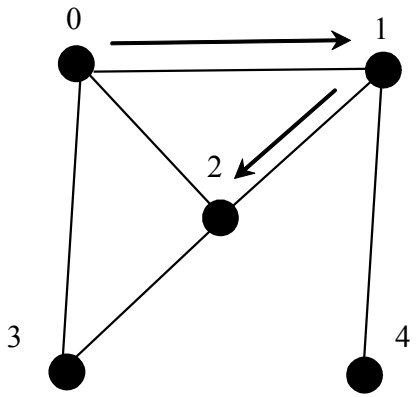
Breadth-First Search Algorithm

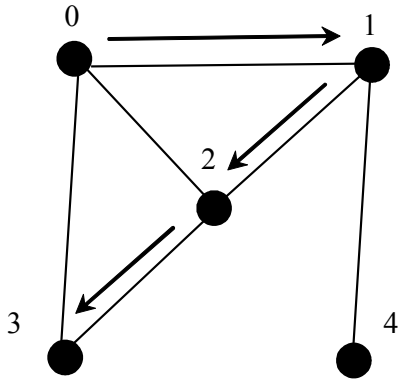
```
bfs(vertex v) {
  create an empty queue for storing vertices to be visited;
  add v into the queue;
  mark v visited;
  while the queue is not empty {
    dequeue a vertex, say u, from the queue
    visit u;
    for each neighbor w of u
      if w has not been visited {
        add w into the queue;
        mark w visited;
      }
  }
}
```

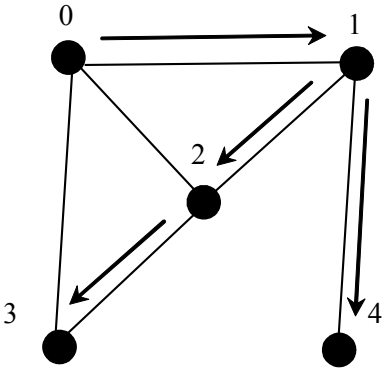
Depth-First Search Example



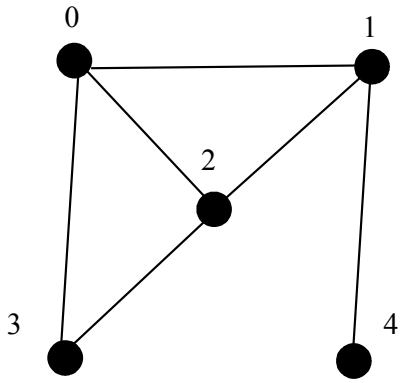


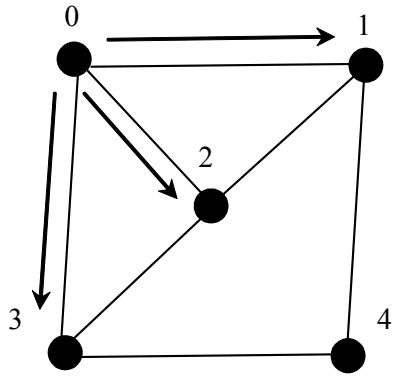


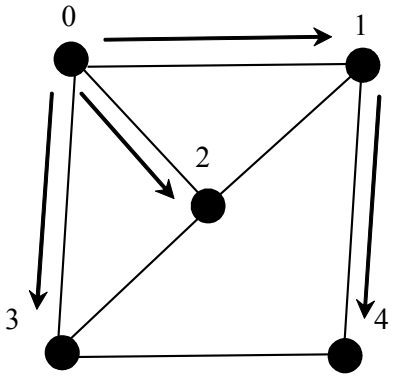




Breadth-First Search Example







Example

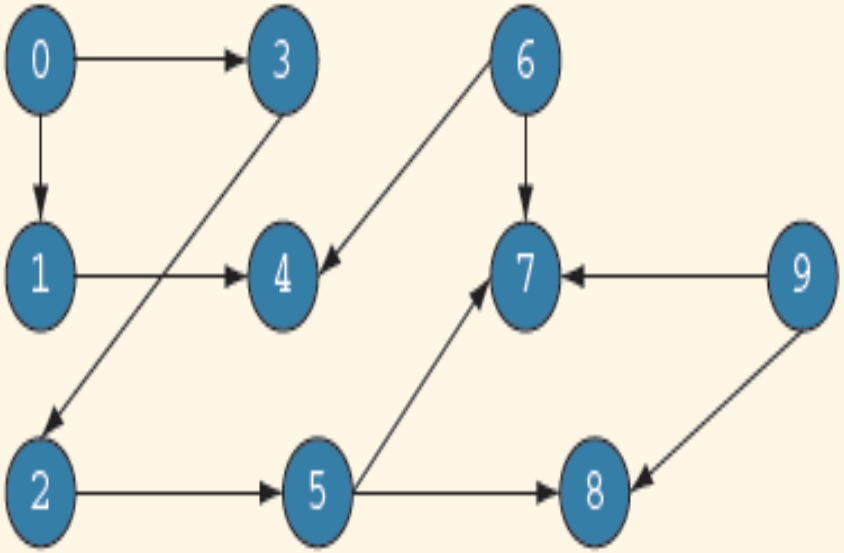


FIGURE 20-6 Directed graph *G*

- Depth-first ordering of vertices:
  - 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

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

- Shortest path: path with the smallest weight
- Shortest path algorithm
  - Called the greedy algorithm, developed by Dijkstra
  - *G*: graph with *n* vertices, where $n \geq 0$
  - $V(G) = \{v_1, v_2, \ldots, v_n\}$
  - *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

- 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

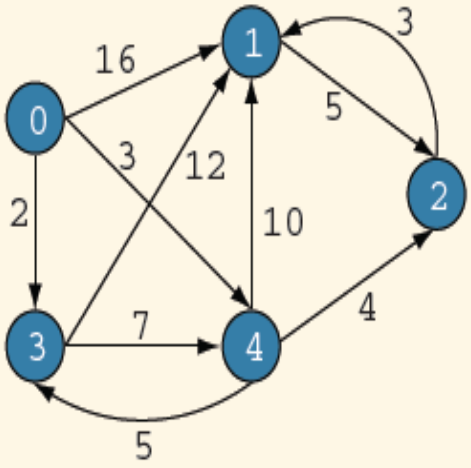
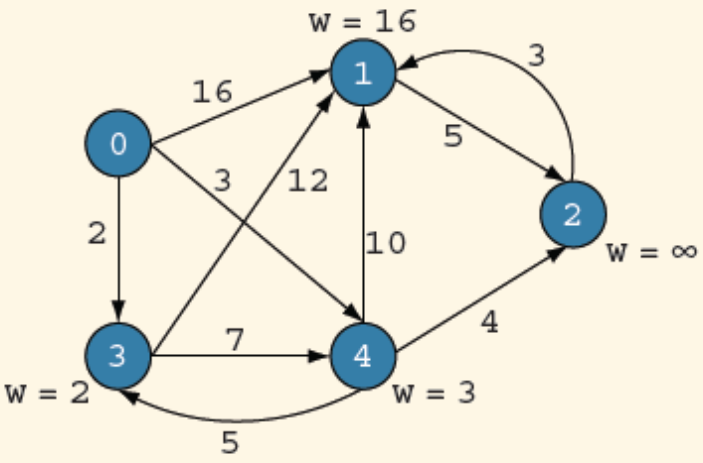


FIGURE 20-7 Weighted graph *G*

Shortest Path Algorithm



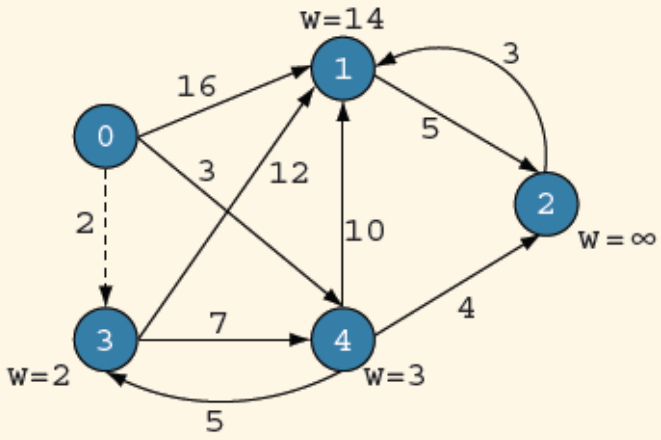
	[0]	[1]	[2]	[3]	[4]
smallestWeight	0	16	∞	2	3

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

FIGURE 20-8 Graph after Steps 1 and 2 execute

Shortest Path Algorithm

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

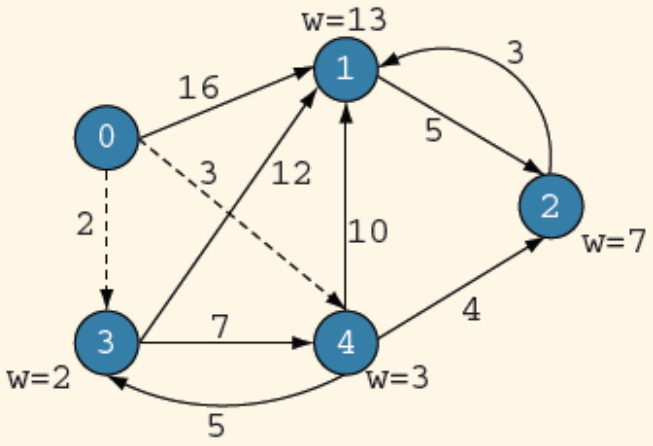


	[0]	[1]	[2]	[3]	[4]
smallestWeight	0	14	∞	2	3

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

Shortest Path Algorithm

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

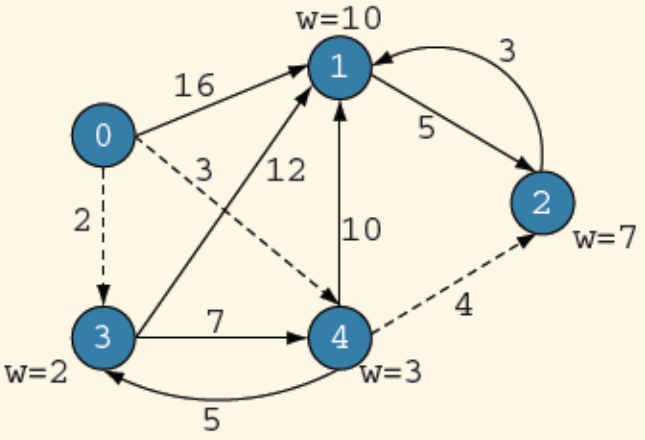


	[0]	[1]	[2]	[3]	[4]
smallestWeight	0	13	7	2	3

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

Shortest Path Algorithm

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

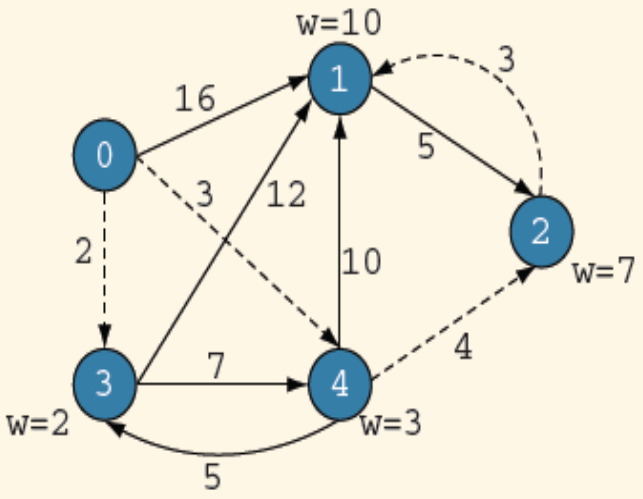


	[0]	[1]	[2]	[3]	[4]
smallestWeight	0	10	7	2	3

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

Shortest Path Algorithm

- 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

Animations

- https://www.cs.usfca.edu/~galles/visualization/BFS.html
- https://www.cs.usfca.edu/~galles/visualization/DFS.html
- https://www.cs.usfca.edu/~galles/visualization/Dijkstra.html
- http://www.ee.ryerson.ca/~courses/coe428/graphs/dijkstra.html

Miscellaneous-Extra Slides

- Alternative Graph representations/code (From Book by Daniel Liang – chapter of ebook available)

Representing Graphs

Representing Vertices

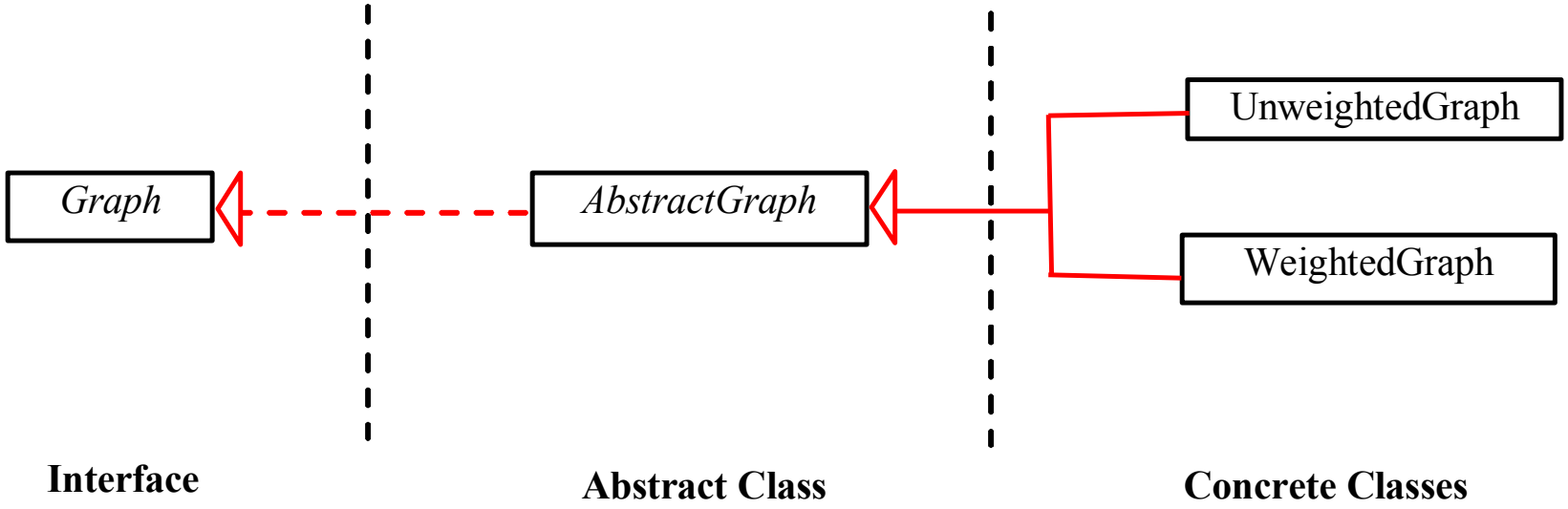
Representing Edges: Edge Array

Representing Edges: Edge Objects

Representing Edges: Adjacency Matrices

Representing Edges: Adjacency Lists

Modeling Graphs



Graph<T>	
#vertices: vector<T>	Vertices in the graph.
#neighbors: vector<vector<int>>	neighbors[i] stores all vertices adjacent to vertex with index i.
+Graph()	Constructs an empty graph.
+Graph(vertices: vector<T>, edges[][2]: int, numberOfEdges: int)	Constructs a graph with the specified vertices in a vector and edges in a 2-D array.
+Graph(numberOfVertices: int, edges[][2]: int, numberOfEdges: int)	Constructs a graph whose vertices are 0, 1, …, n-1and edges are specified in a 2-D array.
+Graph(vertices: vector<T>, edges: vector<Edge>)	Constructs a graph with the vertices in a vector and edges in a vector of Edge objects.
+Graph(numberOfVertices: int, edges: vector<Edge>)	Constructs a graph whose vertices are 0, 1, …, n-1 and edges in a vector of Edge objects.
+getSize(): int	Returns the number of vertices in the graph.
+getDegree(v: int): int	Returns the degree for a specified vertex index.
+getVertex(index: int): T	Returns the vertex for the specified vertex index.
+getIndex(v: T): int	Returns the index for the specified vertex.
+getVertices(): vector<T>	Returns the vertices in the graph in a vector.
+getNeighbors(v: int): vector<int>	Returns the neighbors of vertex with index v.
+printEdges(): void	Prints the edges of the graph to the console.
+printAdjacencyMatrix(): void	Prints the adjacency matrix of the graph to the console.
+dfs(v: int): Tree	Obtains a depth-first search tree.
+bfs(v: int): Tree	Obtains a breadth-first search tree.

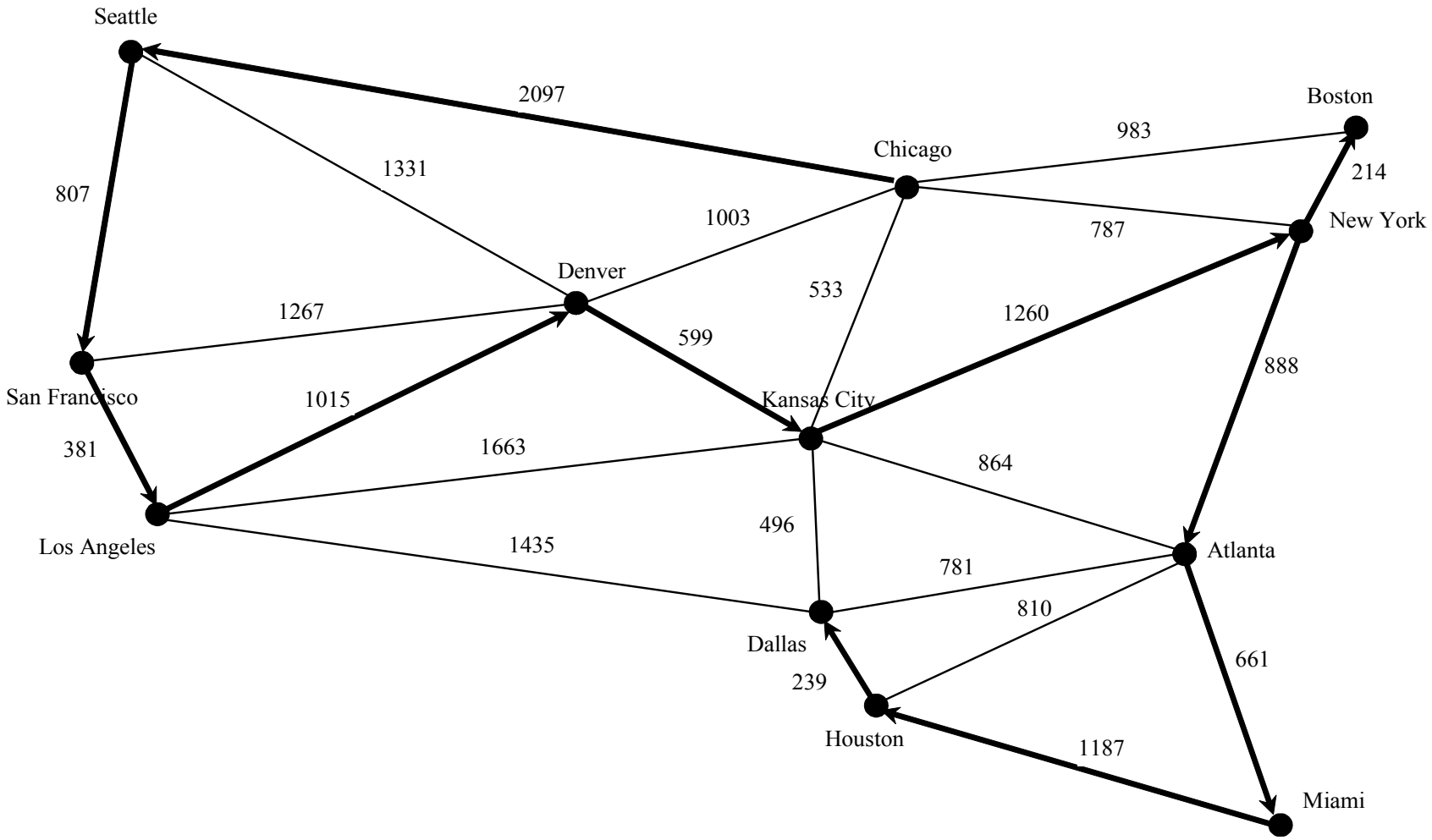
Graph Traversals

Depth-first search and breadth-first search

Both traversals result in a spanning tree, which can be modeled using a class.

Tree	
-root: int	The root of the tree.
-parent: vector<int>	The parents of the vertices.
-searchOrders: vector<int>	The orders for traversing the vertices.
+Tree()	Constructs an emtpy tree.
+Tree(root: int, parent: vector<int>, searchOrders: vector<int>)	Constructs a tree with the specified root, parent, and searchOrders.
+Tree(root: int, parent: vector<int>)	Constructs a tree with the specified root, parent.
+getRoot(): int	Returns the root of the tree.
+getSearchOrders(): vector<int>t	Returns the order of vertices searched.
+getParent(v: int): int	Returns the parent of vertex v.
+getNumberOfVerticesFound(): int	Returns the number of vertices searched.
+getPath(v: int): vector<int>	Returns a path of all vertices leading to the root from v. The return values are in a vector.
+printTree(): void	Displays tree with the root and all edges.

Depth-First Search Example



Applications of the DFS

Detecting whether a graph is connected. Search the graph starting from any vertex. If the number of vertices searched is the same as the number of vertices in the graph, the graph is connected. Otherwise, the graph is not connected

Detecting whether there is a path between two vertices.

Finding a path between two vertices.

Finding all connected components. A connected component is a maximal connected subgraph in which every pair of vertices are connected by a path.

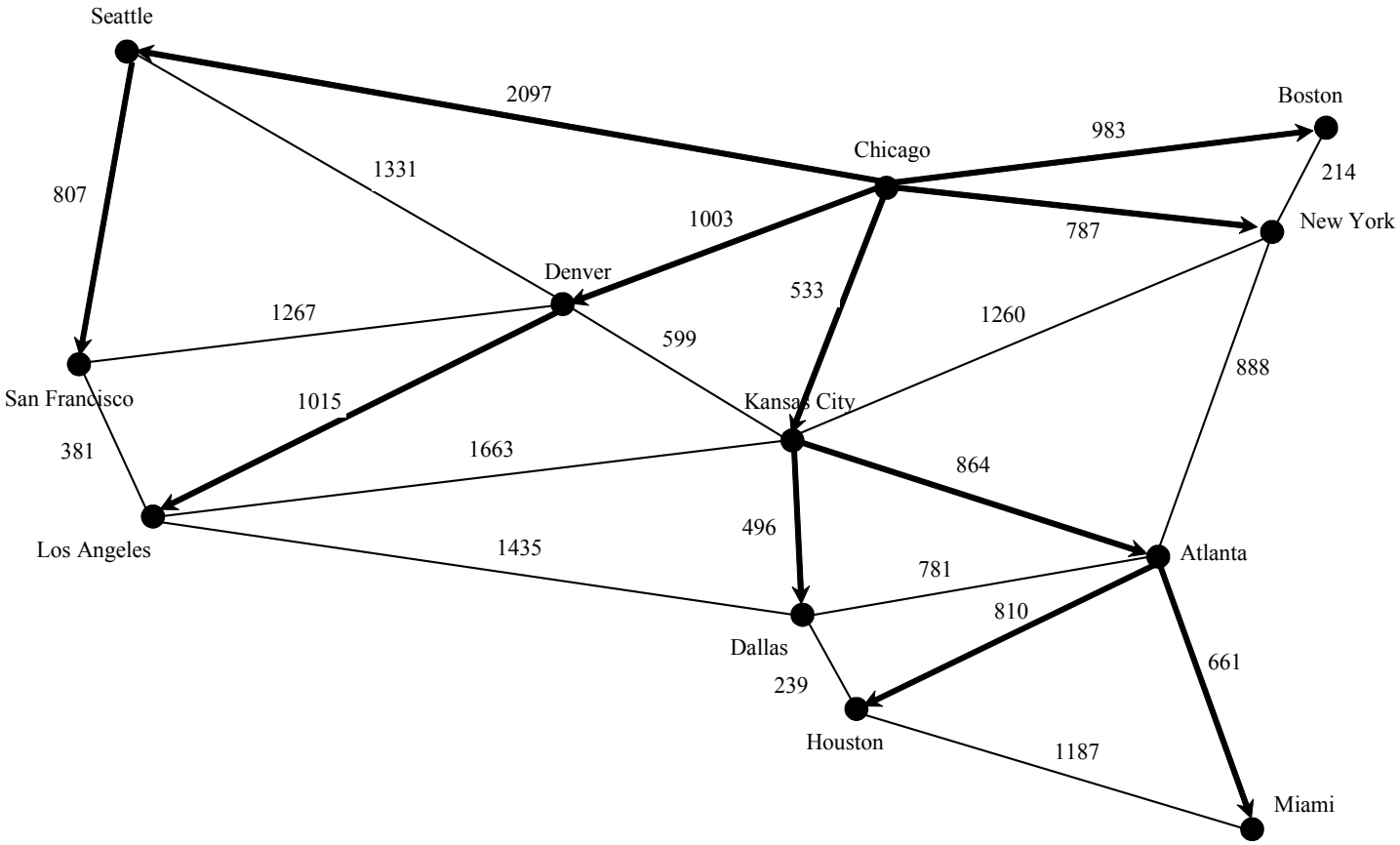
Detecting whether there is a cycle in the graph.

Finding a cycle in the graph.

Breadth-First Search

The breadth-first traversal of a graph is like the breadth-first traversal of a tree. With breadth-first traversal of a tree, the nodes are visited level by level. First the root is visited, then all the children of the root, then the grandchildren of the root from left to right, and so on.

Breadth-First Search Example



Applications of the BFS

Detecting whether a graph is connected. A graph is connected if there is a path between any two vertices in the graph.

Detecting whether there is a path between two vertices.

Finding a shortest path between two vertices. You can prove that the path between the root and any node in the BFS tree is the shortest path between the root and the node

Finding all connected components. A connected component is a maximal connected subgraph in which every pair of vertices are connected by a path.

Detecting whether there is a cycle in the graph.

Finding a cycle in the graph.

Testing whether a graph is bipartite. A graph is bipartite if the vertices of the graph can be divided into two disjoint sets such that no edges exist between vertices in the same set.

The Nine Tail Problem

The problem is stated as follows. Nine coins are placed in a three by three matrix with some face up and some face down. A legal move is to take any coin that is face up and reverse it, together with the coins adjacent to it (this does not include coins that are diagonally adjacent). Your task is to find the minimum number of the moves that lead to all coins face down.

H	H	H
T	T	T
H	H	H

H	H	H
T	H	T
T	T	T

T	T	T
T	T	T
T	T	T