Chapter 14

Graphs
Terminology

• \( G = \{V, E\} \)
• A graph \( G \) consists of two sets
  – A set \( V \) of vertices, or nodes
  – A set \( E \) of edges
• A subgraph
  – Consists of a subset of a graph’s vertices and a subset of its edges
• Adjacent vertices
  – Two vertices that are joined by an edge
Terminology

Figure 14-2
a) A campus map as a graph; b) a subgraph
Terminology

• A path between two vertices
  – A sequence of edges that begins at one vertex and ends at another vertex
  – May pass through the same vertex more than once

• A simple path
  – A path that passes through a vertex only once

• A cycle
  – A path that begins and ends at the same vertex

• A simple cycle
  – A cycle that does not pass through a vertex more than once
Terminology

• A connected graph
  – A graph that has a path between each pair of distinct vertices

• A disconnected graph
  – A graph that has at least one pair of vertices without a path between them

• A complete graph
  – A graph that has an edge between each pair of distinct vertices
Figure 14-3
Graphs that are a) connected; b) disconnected; and c) complete
Terminology

- **Multigraph**
  - Not a graph
  - Allows multiple edges between vertices

**Figure 14-4**

a) A multigraph is not a graph; b) a self edge is not allowed in a graph
Terminology

- **Weighted graph**
  - A graph whose edges have numeric labels

**Figure 14-5a**

a) A weighted graph
Terminology

- Undirected graph
  - Edges do not indicate a direction

- Directed graph, or diagraph
  - Each edge is a directed edge

Figure 14-5b
b) A directed graph
Terminology

- Directed graph
  - Can have two edges between a pair of vertices, one in each direction
  - Directed path
    - A sequence of directed edges between two vertices
    - Vertex y is adjacent to vertex x if
      - There is a directed edge from x to y
Graphs As ADTs

• Graphs can be used as abstract data types
• Two options for defining graphs
  – Vertices contain values
  – Vertices do not contain values
• Operations of the ADT graph
  – Create an empty graph
  – Determine whether a graph is empty
  – Determine the number of vertices in a graph
  – Determine the number of edges in a graph
Graphs As ADTs

• Operations of the ADT graph (Continued)
  – Determine whether an edge exists between two given vertices; for weighted graphs, return weight value
  – Insert a vertex in a graph whose vertices have distinct search keys that differ from the new vertex’s search key
  – Insert an edge between two given vertices in a graph
  – Delete a particular vertex from a graph and any edges between the vertex and other vertices
  – Delete the edge between two given vertices in a graph
  – Retrieve from a graph the vertex that contains a given search key
Implementing Graphs

- Most common implementations of a graph
  - Adjacency matrix
  - Adjacency list

- Adjacency matrix
  - Adjacency matrix for a graph with n vertices numbered 0, 1, …, n – 1
    - An n by n array matrix such that matrix[i][j] is
      - 1 (or true) if there is an edge from vertex i to vertex j
      - 0 (or false) if there is no edge from vertex i to vertex j
Implementing Graphs

Figure 14-6

a) A directed graph and b) its adjacency matrix

© 2011 Pearson Addison-Wesley. All rights reserved
Implementing Graphs

• Adjacency matrix for a weighted graph with \( n \) vertices numbered \( 0, 1, \ldots, n-1 \)
  – An \( n \) by \( n \) array matrix such that \( \text{matrix}[i][j] \) is
    • The weight that labels the edge from vertex \( i \) to vertex \( j \) if there is an edge from \( i \) to \( j \)
    • \( \infty \) if there is no edge from vertex \( i \) to vertex \( j \)

Figure 14-7
a) A weighted undirected graph and b) its adjacency matrix
Implementing Graphs

• Adjacency list
  - An adjacency list for a graph with \( n \) vertices numbered 0, 1, \( \ldots \), \( n - 1 \)
  - Consists of \( n \) linked lists
  - The \( i^{\text{th}} \) linked list has a node for vertex \( j \) if and only if the graph contains an edge from vertex \( i \) to vertex \( j \)
    - This node can contain either
      » Vertex \( j \)'s value, if any
      » An indication of vertex \( j \)'s identity
Implementing Graphs

Figure 14-8
a) A directed graph and
b) its adjacency list
Implementing Graphs

- Adjacency list for an undirected graph
  - Treats each edge as if it were two directed edges in opposite directions

Figure 14-9
a) A weighted undirected graph and b) its adjacency list
Implementing Graphs

- Adjacency matrix compared with adjacency list
  - Two common operations on graphs
    - Determine whether there is an edge from vertex i to vertex j
    - Find all vertices adjacent to a given vertex i
  - Adjacency matrix
    - Supports operation 1 more efficiently
  - Adjacency list
    - Supports operation 2 more efficiently
    - Often requires less space than an adjacency matrix
Implementing a Graph Class Using the JCF

• ADT graph not part of JCF
• Can implement a graph using an adjacency list consisting of a vector of maps
• Implementation presented uses TreeSet class
Graph Traversals

• A graph-traversal algorithm
  – Visits all the vertices that it can reach
  – Visits all vertices of the graph if and only if the graph is connected
    • A connected component
      – The subset of vertices visited during a traversal that begins at a given vertex
  – Can loop indefinitely if a graph contains a loop
    • To prevent this, the algorithm must
      – Mark each vertex during a visit, and
      – Never visit a vertex more than once
Graph Traversals

Figure 14-10
Visitation order for a) a depth-first search; b) a breadth-first search
Depth-First Search

• Depth-first search (DFS) traversal
  – Proceeds along a path from $v$ as deeply into the graph as possible before backing up
  – Does not completely specify the order in which it should visit the vertices adjacent to $v$
  – A last visited, first explored strategy
Breadth-First Search

• Breadth-first search (BFS) traversal
  – Visits every vertex adjacent to a vertex \( v \) that it can before visiting any other vertex
  – A first visited, first explored strategy
  – An iterative form uses a queue
  – A recursive form is possible, but not simple
Implementing a BFS Iterator Class Using the JCF

- **BFSIterator class uses the ListIterator class**
  - As a queue to keep track of the order the vertices should be processed

- **BFSIterator constructor**
  - Initiates methods used to determine BFS order of vertices for the graph

- **Graph is searched by processing vertices from each vertex’s adjacency list**
  - In the order that they were pushed onto the queue
Applications of Graphs: Topological Sorting

- **Topological order**
  - A list of vertices in a directed graph without cycles such that vertex x precedes vertex y if there is a directed edge from x to y in the graph
  - There may be several topological orders in a given graph

- **Topological sorting**
  - Arranging the vertices into a topological order
Topological Sorting

Figure 14-14
A directed graph without cycles

Figure 14-15
The graph in Figure 14-14 arranged according to the topological orders a) a, g, d, b, e, c, f and b) a, b, g, d, e, f, c
Topological Sorting

- Simple algorithms for finding a topological order
  - topSort1
    - Find a vertex that has no successor
    - Remove from the graph that vertex and all edges that lead to it, and add the vertex to the beginning of a list of vertices
    - Add each subsequent vertex that has no successor to the beginning of the list
    - When the graph is empty, the list of vertices will be in topological order
Topological Sorting

• Simple algorithms for finding a topological order (Continued)
  – topSort2
    • A modification of the iterative DFS algorithm
    • Strategy
      – Push all vertices that have no predecessor onto a stack
      – Each time you pop a vertex from the stack, add it to the beginning of a list of vertices
      – When the traversal ends, the list of vertices will be in topological order
Spanning Trees

• A tree
  – An undirected connected graph without cycles

• A spanning tree of a connected undirected graph G
  – A subgraph of G that contains all of G’s vertices and enough of its edges to form a tree

• To obtain a spanning tree from a connected undirected graph with cycles
  – Remove edges until there are no cycles
Spanning Trees

- You can determine whether a connected graph contains a cycle by counting its vertices and edges
  - A connected undirected graph that has $n$ vertices must have at least $n - 1$ edges
  - A connected undirected graph that has $n$ vertices and exactly $n - 1$ edges cannot contain a cycle
  - A connected undirected graph that has $n$ vertices and more than $n - 1$ edges must contain at least one cycle
Spanning Trees

Figure 14-19
Connected graphs that each have four vertices and three edges
The DFS Spanning Tree

- To create a depth-first search (DFS) spanning tree
  - Traverse the graph using a depth-first search and mark the edges that you follow
  - After the traversal is complete, the graph’s vertices and marked edges form the spanning tree
The BFS Spanning Tree

- To create a breath-first search (BFS) spanning tree
  - Traverse the graph using a bread-first search and mark the edges that you follow
  - When the traversal is complete, the graph’s vertices and marked edges form the spanning tree
Minimum Spanning Trees

• **Minimum spanning tree**
  – A spanning tree for which the sum of its edge weights is minimal

• **Prim’s algorithm**
  – Finds a minimal spanning tree that begins at any vertex
  – **Strategy**
    • Find the least-cost edge \((v, u)\) from a visited vertex \(v\) to some unvisited vertex \(u\)
    • Mark \(u\) as visited
    • Add the vertex \(u\) and the edge \((v, u)\) to the minimum spanning tree
    • Repeat the above steps until there are no more unvisited vertices
Shortest Paths

• Shortest path between two vertices in a weighted graph
  – The path that has the smallest sum of its edge weights

• Dijkstra’s shortest-path algorithm
  – Determines the shortest paths between a given origin and all other vertices
  – Uses
    • A set vertexSet of selected vertices
    • An array weight, where weight[v] is the weight of the shortest (cheapest) path from vertex 0 to vertex v that passes through vertices in vertexSet
Circuits

- A circuit
  - A special cycle that passes through every vertex (or edge) in a graph exactly once
- Euler circuit
  - A circuit that begins at a vertex $v$, passes through every edge exactly once, and terminates at $v$
  - Exists if and only if each vertex touches an even number of edges

Figure 14-27

a) Euler’s bridge problem
and b) its multigraph representation
Some Difficult Problems

• Three applications of graphs
  – The traveling salesperson problem
  – The three utilities problem
  – The four-color problem

• A Hamilton circuit
  – Begins at a vertex $v$, passes through every vertex exactly once, and terminates at $v$
Summary

• The two most common implementations of a graph are the adjacency matrix and the adjacency list

• Graph searching
  – Depth-first search goes as deep into the graph as it can before backtracking
  – Bread-first search visits all possible adjacent vertices before traversing further into the graph

• Topological sorting produces a linear order of the vertices in a directed graph without cycles
Summary

• Trees are connected undirected graphs without cycles
  – A spanning tree of a connected undirected graph is a subgraph that contains all the graph’s vertices and enough of its edges to form a tree
• A minimum spanning tree for a weighted undirected graph is a spanning tree whose edge-weight sum is minimal
• The shortest path between two vertices in a weighted directed graph is the path that has the smallest sum of its edge weights
Summary

- An Euler circuit in an undirected graph is a cycle that begins at vertex v, passes through every edge in the graph exactly once, and terminates at v.
- A Hamilton circuit in an undirected graph is a cycle that begins at vertex v, passes through every vertex in the graph exactly once, and terminates at v.