Module 6

This is a single, concatenated file, suitable for printing or saving as a PDF for offline viewing. Please note that some animations or images may not work.

Module 6 Study Guide and Deliverables

Topics:	• Graph Algorithms • Computational Complexity
Readings:	• Module 6 online content • Textbook: Chapter 14 (except 14.4)
Assignments:	• Assignment 6 due **Tuesday, April 26 at 6:00 AM ET** • Term Project due **Tuesday, April 26 at 6:00 AM ET**
Assessments:	• Quiz 6 due **Tuesday, April 26 at 6:00 AM ET**
Course Evaluation:	Course Evaluation opens on **Monday, April 18, at 10:00 AM ET** and closes on **Monday, April 25, at 11:59 PM ET**. Please complete the course evaluation. Your feedback is important to MET, as it helps us make improvements to the program and the course for future students.
Live Classrooms:	• **Tuesday, April 19 from 8:00-9:30 PM** • **Thursday, April 21 from 8:00-10:00 PM** • Another one-hour live office hour session led by your facilitator: TBD

Module 6 Learning Objectives

In this module, we discuss graph algorithms and computational complexity.

After successfully completing this module, you will be able to:

1. Explain the basic definitions and terminologies related to graphs.
2. Illustrate graph data structure using an edge list, adjacency list, adjacency map, or an adjacency matrix.
3. Performance analysis of graph data structures using an edge list, adjacency list, adjacency map, or an adjacency matrix.
4. Illustrate breadth-first search.
5. Implement breadth-first search.
6. Illustrate depth-first search.
7. Implement depth-first search.
8. Implement a topological sort algorithm.

9. Illustrate Dijkstra’s algorithm.
10. Implement Dijkstra’s algorithm.
11. Illustrate Prim-Jarnik algorithm.
12. Implement Prim-Jarnik algorithm.
13. Illustrate Kruskal’s algorithm.
14. Implement Kruskal’s algorithm.

Section 6.1 Graphs Algorithms

Section 6.1. Graphs Algorithms

Overview

A graph is an abstraction that is used to model relationships among objects. We can solve many real-world problems using graphs. In this section, we discuss various graph algorithms.

Section 6.1.1. Graphs

In this section, we discuss basic definitions and terminologies and describe a graph ADT.

Section 6.1.1.1 Basic Definitions and Terminologies

A *graph* is a set V of *vertices* and a collection V of *edges*. An edge connects two vertices and is represented as (u, v), where u and v are vertices. Sometimes, vertices are referred to as *nodes* and edges are referred to as *arcs*.

An edge (u, v) is said to be *directed* from u to v if the pair (u, v) is ordered, with u preceding v.
An edge (u, v) is said to be *undirected* if the pair (u, v) is not ordered.

A graph is a *directed graph*, or a *digraph*, if all edges in the graph are directed.
A graph is an *undirected graph* if all edges in the graph are undirected.
A graph that has both directed and undirected edges is called a *mixed graph*.

The following figure shows examples of an undirected graph and a directed graph.

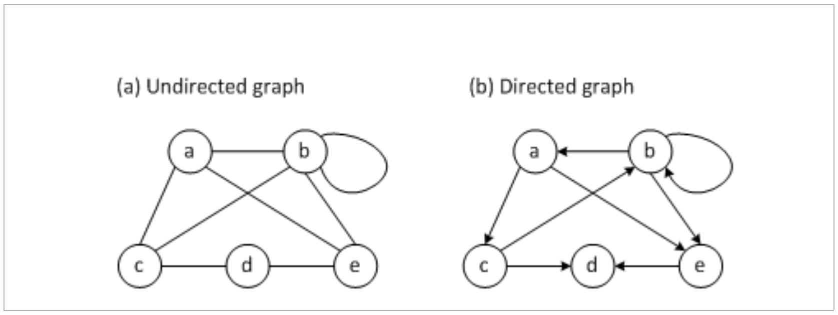


Figure 6.1. An undirected graph and a directed graph.

Two vertices connected by an edge are called *end vertices* (or *endpoints*).
If an edge (u, v) is directed, the first end-point u is called the *origin* of the edge, and the second end point v is called the *destination* of the edge.

Two vertices u and v are said to be *adjacent* if there is an edge (u, v).
An edge is said to be *incident* to a vertex if the vertex is one of the edge's endpoints.

In Figure 6.1(a), the vertices a and b are adjacent because there is an edge connecting them, and the edge a, b is incident to the vertex a.

Edges in a graph may have some information associated with them. For example, if a graph represents a road network in the U.S., each vertex representing a city and each edge representing a road connecting two cities, we can assign a distance to each edge, as shown below:

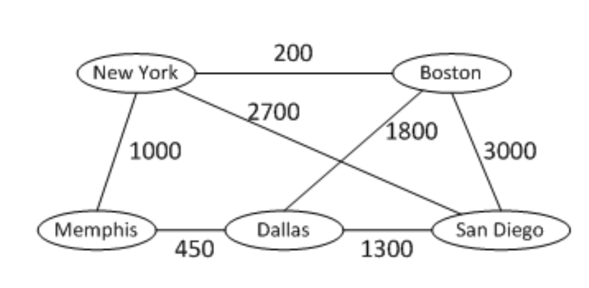


Figure 6.2. Each edge is associated with a distance.

The *outgoing edges* of a vertex are the directed edges of which the origin is that vertex. The *incoming edges* of a vertex are the directed edges of which the destination is that vertex.
The degree of a vertex v, denoted $deg(v)$, is the number of edges incident to v.
The *in-degree*, denoted $indeg(v)$, is the number of incoming edges of v.
The *out-degree*, denoted $outdeg(v)$, is the number of outgoing edges of v.

Consider the following directed graph:

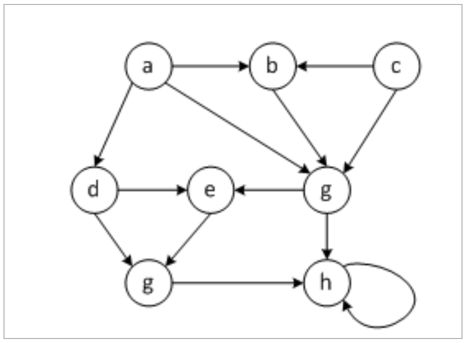


Figure 6.3. A directed graph.

The outgoing edges of vertex g are (g, e) and (g, h).
The incoming edges of vertex g are $(a, g), (b, g), (c, g)$.
The degree of vertex $g, deg(g) = 5$.
The in-degree of vertex $g, indeg(g) = 3$.
The out-degree of vertex $g, outdeg(g) = 2$.

By definition, a graph has a *collection* of edges. So, two edges with the same end vertices are allowed in an undirected graph, and two edges with the same origin vertex and the same destination edge are allowed in a directed graph. Such edges are called *parallel edges* or *multiple edges*. An edge is a *self-loop* if its two end vertices are identical.

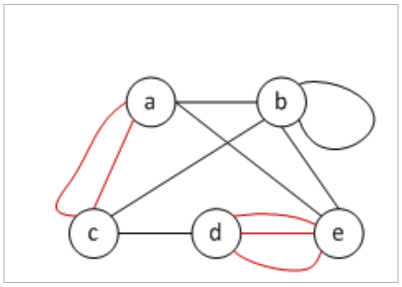


Figure 6.4. A graph with parallel edges and a self-loop.

In the figure above, there are parallel edges between vertices a and c, and between d and e. The vertex b has a self-loop (b, b).

A graph is a *simple* graph if it has neither parallel edges nor self-loops. In this chapter, we assume that graphs are simple graphs unless otherwise specified.

We use the notation $G = (V, E)$ to denote a graph G with a vertex set V and an edge set E.

A *path* is a sequence of alternating vertices and edges that starts at a vertex and ends at a vertex, such that each edge is incident to its predecessor and successor vertices. If a graph is a simple graph, a path is represented as a sequence of vertices only (because there is at most one edge between two vertices).

A *cycle* is a path that starts and ends at the same vertex.

A path is *simple* if each vertex in the path is distinct.
A cycle is *simple* if each vertex in the cycle is distinct, except for the first and last vertices.

A *directed path* is a path in which all edges are directed and are traversed along their direction.

A *directed cycle* is cycle in which all edges are directed and are traversed along their direction.

When we discuss directed paths and directed cycles in a directed graph, we simply call them paths and cycles without qualifying them with "directed."

Let's consider the following example graphs again:

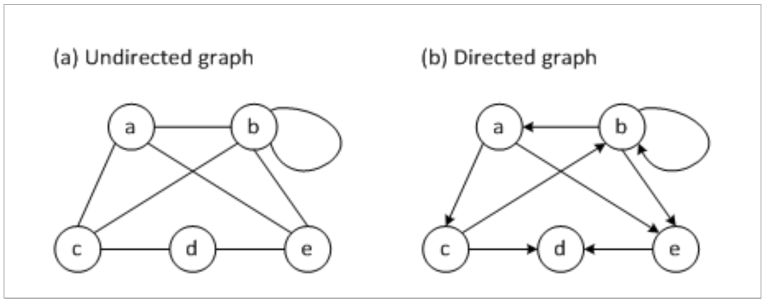


Figure 6.5. An undirected graph and a directed graph.

In graph (a), the sequence <a, c, d, e> is a simple path from vertex a to vertex e. The sequence $< a, b, c, a, e >$ is also a path from vertex a to vertex e, but it is not a simple path because vertex a appears twice. The sequence $< a, b, c, a >$ is a simple cycle. The sequence $< a, b, c, a, e, b, a >$ is cycle, but not a simple cycle.

In graph (b), the sequence $< a, c, b, e >$ is a directed path, and the sequence $< a, c, b, a >$ is a directed cycle.

If there is path from vertex u to vertex v, we say u reaches v, and v is *reachable* from u. The *reachability* is symmetric in an undirected graph, but not in a directed graph.

A graph is *connected* if, for any two vertices, there is a path between them. A directed graph is *strongly connected* if, for any two vertices u and v, u reaches v and v reaches u.

Figure 6.6 shows examples of a connected graph and a strongly connected graph:

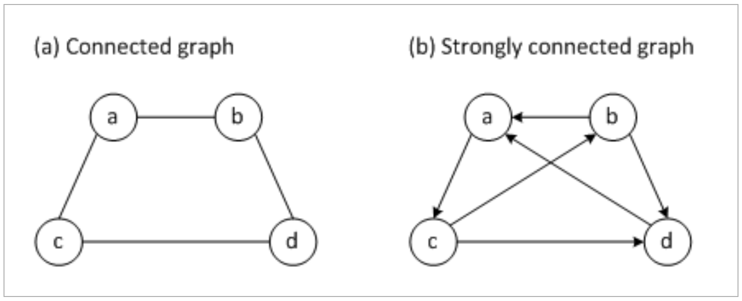


Figure 6.6. A connected graph and a strongly connected graph.

A *subgraph* of a graph G is a graph H, the vertices and edges of which are subsets of the vertices and edges of G, respectively. A *spanning subgraph* of G is a subgraph of G that contains all vertices of G.

The following figure shows two subgraph examples, indicated by thick red edges. The one on the left is a subgraph, but it is not a spanning subgraph (because vertex *d* is not included). The subgraph on the right is a spanning subgraph.

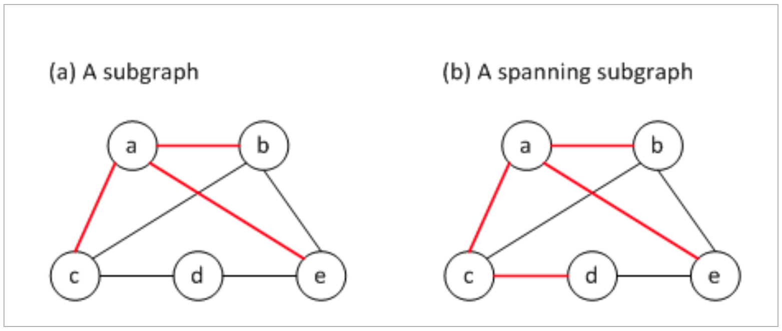


Figure 6.7. Subgraph examples.

A *forest* is a graph without cycles. A *tree* is a connected forest—that is, a connected graph without cycles. In the following figure, the graph on the left is a forest, and the one on the right is a tree:

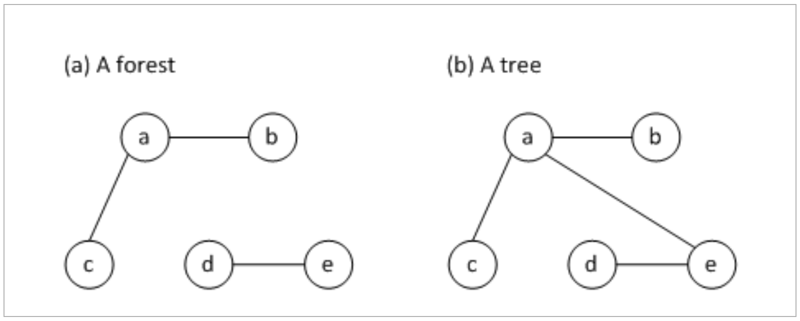


Figure 6.8. A forest and a tree.

A *spanning tree* of a graph is a spanning subtree that is a tree. In fact, the subgraph of Figure 6.7(b), indicated with thick red edges, is a spanning tree because it is also a tree.

We now show some important properties of graphs:

- If a graph $G = (V, E)$ has m edges, then the following is true:

  $$\sum_{v \ in \ V} \deg(v) = 2m$$

  Proof: When we count the degrees of all vertices in G, each edge (u, v) is counted twice—once by its endpoint u, and once by its endpoint v. Therefore, the total number of degrees of all vertices is twice the number of edges.

- If $G = (V, E)$ is a directed graph with m edges, then the following is true:

  $$\sum_{v \ in \ V} in \deg(v) = \sum_{v \ in \ V} out \deg(v) = m$$

  Proof: In a directed graph, when counting the in-degrees of all vertices, each edge (u, v) is counted once for vertex u. So, the total number of in-degrees is the same as the number of edges. When counting the total number of out-degrees of all vertices, each edge (u, v) is counted once for vertex u. So, the total number of out-degrees is the same as the number of edges.

- Let G be a simple graph with n vertices and m edges.

1. If G is undirected, then $m \leq \frac{n(n-1)}{2}$
2. If G is directed, then $m \leq n(n-1)$.

Proof:

1. In an undirected simple graph, no two edges can have the same endpoints. So, the maximum degree of a vertex is $n-1$, and the total number of degrees of all vertices is at most $n(n-1)$. So, we have $\sum_{v\ in\ V} \deg(v) = 2m \leq n(n-1)$. Therefore, $m \leq \frac{n(n-1)}{2}$.
2. In a directed simple graph, no two edges can have the same origin and destination. So, the maximum in-degree (or out-degree) of a vertex is $n-1$. So, we have $\sum_{v\ in\ V} in \deg(v) = \sum_{v\ in\ V} out \deg(v) = m \leq n(n-1)$ Therefore, $m \leq n(n-1)$.

- Let G be an undirected graph with n vertices and m edges:
    - If G is connected, then $m \geq n-1$.
    - If G is a tree, then $m = n-1$.
    - If G is a forest, then $m \leq n-1$.

Section 6.1.1.2. The Graph ADT

A graph abstraction involves three data types: *Vertex*, *Edge*, and *Graph*. A *Vertex* object represents a vertex in a graph and stores an arbitrary element provided by the user. We assume that the *getElement*( ) method retrieves the element from the object. An *Edge* object represents an edge in a graph and stores an element associated with the edge, such as a distance between two cities in a road network. Again, we assume that it has the *getElement*( ) method, which returns the element of an edge.

We now present the *Graph* ADT, which supports both undirected and directed graphs. The following operations are defined in the ADT:

- numVertices( )—Returns the number of vertices of the graph.
- vertices( )—Returns an iteration of all vertices of the graph.
- numEdges( )—Returns the number of edges of the graph.
- edges( )—Returns an iteration of all edges of the graph.
- getEdge(u, v)—Returns an edge from vertex u to vertex v, if one exists, and returns *null* otherwise. In an undirected graph, getEdge(u, v) and getEdge(v, u) are the same.
- endVertices(e)— Returns an array containing the two end vertices of edge e. If the graph is directed, the first vertex is the origin, and the second vertex is the destination.
- opposite(v, e)—For edge e incident to vertex v, returns the other vertex of the edge. An error occurs if e is not incident to v.
- outDegree(v)— Returns the number of outgoing edges from vertex v.
- indegree(v)— Returns the number of incoming edges to vertex v. For an undirected graph, outDegree(v) and inDegree(v) return the same value.
- outgoingEdges(v)—Returns an iteration of all outgoing edges from v.
- incomingEdges(v)—Returns an iteration of all incoming edges to v. For an undirected graph, outgoingEdges(v) and incomingEdges(v) return the same collection.
- insertVertex(x)— Creates and returns a new *Vertex* storing element x.
- insertEdge(u, v, x)—Creates and returns a new *Edge*, storing element x, from u to v. An error occurs if there already exists an edge from u to v.
- removeVertex(v)— Removes vertex v and all its incident edges from the graph.
- removeEdge(e)— Removes edge e from the graph.

Section 6.1.2. Data Structures for Graphs

The following four data structures can be used to represent a graph:

- *Edge list*
    - We maintain an unordered list of all edges.
    - This is not efficient in locating a particular edge or all edges incident to a vertex.
- *Adjacency list*
    - For each vertex, a list containing the edges that are incident to the vertex is maintained.
    - We can efficiently find all edges incident to a given vertex.
- *Adjacent map*
    - This is similar to an adjacency list.
    - For each vertex, a map containing the edges that are incident to the vertex is maintained.
    - For each edge in the map, the vertex that is the other end of the edge is used as the key to locate the edge.
    - Accessing a specific edge (u, v) can be done in $O(1)$ time.
- *Adjacency matrix*
    - We maintain an $n \times n$ matrix, where n is the number of vertices in the graph.
    - Each entry in the matrix stores a reference to the edge (u, v) for a particular pair of vertices, (\u\) and <(\v\). If there is no such edge in the graph, it stores *null*.
    - Accessing a specific edge takes $O(1)$ time in the worst case.

The space requirement for all data structures, except the adjacency matrix is $O(n+m)$. The adjacency matrix requires $O(n^2)$ space.

The following table summarizes the running time performance of the four data structures for some graph ADT operations. In the table, n denotes the number of vertices in the graph, and d_v denotes the degree of vertex v.

Figure 6.9. Performance of Four Data Structures

Method	Edge list	Adj. List	Adj. Map	Adj. Matrix
vertices( )	$O(n)$	$O(n)$	$O(n)$	$O(n)$
edges( )	$O(m)$	$O(m)$	$O(m)$	$O(m)$
getEdge(u, v)	$O(m)$	$O(\min(d_u, d_v)$	$O(1)\mathrm{exp.}$	$O(1)$

outDegree(v)
inDegree(v)
	$O(m)$	$O(1)$	$O(1)$	$O(n)$

outgoingEdges(v)
incomingEdges(v)
	$O(m)$	$O(d_v)$	$O(d_v)$	$O(n)$

insertVertex(x)	$O(1)$	$O(1)$	$O(1)$	$O(n^2)$
removeVertex(v)	$O(m)$	$O(d_v)$	$O(d_v)$	$O(n^2)$
insertEdge(u, v, x)	$O(1)$	$O(1)$	$O(1)\mathrm{exp.}$	$O(1)$
remove Edge(e)	$O(1)$	$O(1)$	$O(1)\mathrm{exp.}$	$O(1)$

Each of these four data structures will be described in more detail in the following sections.

Section 6.1.2.1. Edge-List Structure

An edge list maintains an unordered list E of all edges. It also keeps an unordered list V of all vertices. This data structure is illustrated below. Notice that each *Edge* object references two *Vertex* objects, to which it is incident.

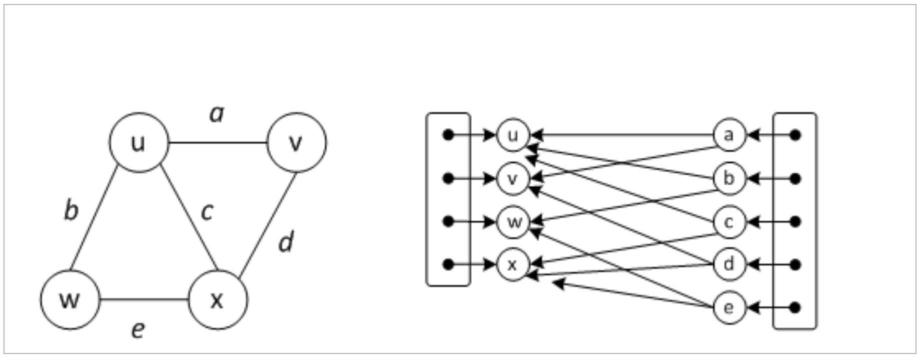


Figure 6.10. Edge list.

We assume that a *Vertex* object and an *Edge* object will have the following properties when they are implemented.

A *Vertex* object v storing element x has the following instance variables:

- A reference to element x, to support the *getElement*( ) method.
- A reference to vertex v in the list V, allowing v to be removed efficiently.

An *Edge* object e storing element x has the following instance variables:

- A reference to element x, to support the *getElement*( ) method.
- References to vertex objects that are endpoints of e, allowing constant time support for the *endVertices*(e) method and the *opposite*(v, e) method.
- A reference to edge e in the list E, allowing e to be removed efficiently.

The list V stores n vertices, and the list E stores m edges. So, the total space requirement is $O(n + m)$.

The following are running times of some operations:

- vertices( )—Iterating over the list V needs $O(n)$.
- edges( )—Iterating over the list E needs $O(m)$.
- getEdge(u, v), outDegree(v), inDegree(v)—Each of these operations requires the scan of the list E. So, the running time is $O(m)$.
- insertVertex(x)—A new vertex can be simply added to the (unordered) list V. So, it takes $O(1)$ time.
- insertEdge(u, v, x)—A new edge can be simply added to the (unordered) list E. So, it takes $O(1)$ time.
- removeEdge(x)—Each edge object keeps the reference to the edge in E. So, an edge can be removed using the reference in $O(1)$ time.
- remove Vertex(v)—When a vertex is remove, all edges that are incident to the vertex must be also removed. Removing those edges requires the scan of the E list. So, it takes $O(m)$ time.

Section 6.1.2.2. Adjacency-List Structure

We keep a list of vertices, and each vertex v has a reference to a separate collection containing those vertices that are adjacent to v. The collection is called an *incidence collection*. The adjacency-list data structure is illustrated below:

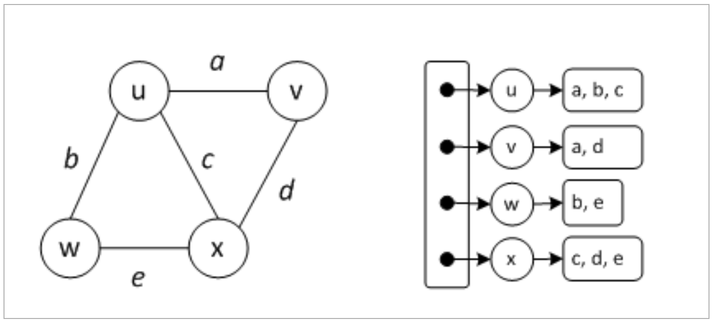


Figure 6.11. Adjacency list.

The vertex list V needs $O(n)$ space. The edge list E stores $2m$ edges, because each edge is incident to two endpoints, and it requires $O(2m) = O(m)$ space. So, the total space requirement is $O(n + m)$.

Let $I(v)$ be the collection storing edges that are incident to vertex v, and $deg(v)$ be the degree of vertex v. The running times of some operations are shown below:

- vertices( )—Iterating over the list V needs $O(n)$.
- edges( )—Iterating over all collections needs $O(m)$.
- getEdge(u, v)—To locate a specific edge (u, v), we need to scan $I(u)$ or $I(v)$. If we scan the shorter of the two, it takes $O(\min(deg(u), deg(v))$.
- outDegree(v)—If we assume that the $I(v)$ collection has a *size* method, the $outDegree(v)$ method runs in $O(1)$ time.
- inDegree(v)—It takes $O(1)$ (the same as $outDegree$).
- insertVertex(x)— A new vertex can be simply added to the (unordered) list V. So, it takes $O(1)$ time.
- insertEdge(u, v, x)—A new edge can be simply added to the collection of u and v. So, it takes $O(1)$ time.
- removeEdge(x)—If an edge object has references to the its position in $I(u)$ and $I(v)$, an edge can be removed using the reference in $O(1)$ time.
- remove Vertex(v)—When a vertex is removed, all edges that are incident to the vertex must be also removed. So, it takes $O(deg(v))$ time.

Section 6.1.2.3. Adjacency-Map Structure

This data structure is similar to the adjacency list, except that each $I(v)$ is implemented as a map. Suppose that an edge $e = (v, u)$ is stored in $I(v)$. Then, the (u, e) pair is stored in the map. In other words, the vertex u is used as a key to search the edge e. The data structure is shown below:

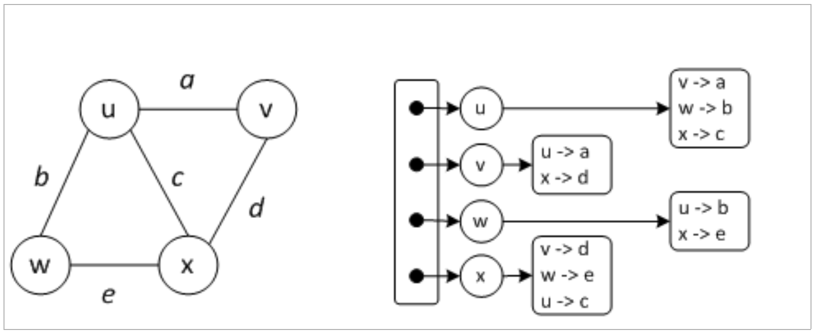


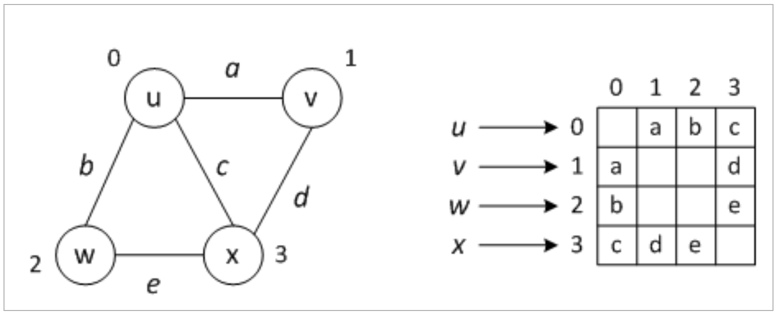
Figure 6.12. Adjacency map.

$I(v)$ uses $O(deg(v))$ space for each vertex v. So, the total space requirement is $O(n+m)$.

The running times of the map-ADT operations with the adjacency map are the same as those with the adjacency list, except for the *getEdge*, *insertEdge*, and *removeEdge* methods. Since an edge can be located in the map in expected $O(1)$ time, these three methods run in expected $O(1)$. The worst-case running time of these methods is $O(min(deg(u), deg(v))$.

Section 6.1.2.4. Adjacency-Matrix Structure

The adjacency-matrix data structure stores a graph in an $n \times n$ matrix. Both columns and rows represent vertices. Vertices are encoded to integers and these integers, which are used as indexes in the matrix. Each entry in a matrix, corresponding to a vertex u and a vertex v, stores the edge with u and v as its endpoints. A graph with its adjacency matrix is shown below:



	0	1	2	3
0		a	b	c
1	a			d
2	b			e
3	c	d	e	

Figure 6.13. Adjacency matrix.

It is obvious that this data structure requires $O(n^2)$ storage. If a given graph is *sparse*, there is a large amount of wasted space.

The primary advantage, in terms of running time, is that we can access any edge in worst-case $O(1)$ time. However, the insertion and removal of vertices may require the resizing of the matrix, so they are inefficient, resulting in $O(n^2)$ time. The $outDegree(v)$ or $inDegree(v)$ method requires $O(n)$ time because a row (or column) corresponding to the vertex v needs to be scanned. The running times of other operations can be found in the table in Section 6.2.1.

A Java implementation of the *Graph* ADT, which uses the adjacency map, can be found at the *AdjacencyMapGraph.java* file.

Section 6.1.3. Graph Traversals

A *graph traversal* is a systematic procedure for visiting (and processing) all vertices in the graph. We say a traversal is efficient if its running time is proportional to the number of vertices and edges in the graph.

Graph traversal algorithms are used to solve many graph-related problems. Interesting problems in an undirected graph G include the following:

- Find a path from vertex u to vertex v.
- Given a start vertex s , find a path with the minimum number of edges from s to every other vertex.
- Test whether G is connected.
- Find a spanning tree of G.
- Identify a cycle in G.

The following are examples of problems in a directed graph G:

- Find a direct path from vertex u to vertex v.
- Find all vertices of G that are reachable from a given vertex s.
- Determine whether G is acyclic.
- Determine whether G is strongly connected.

We will discuss two graph traversal algorithms, called *depth-first search* and *bread-first search*.

Section 6.1.3.1. Depth-First Search

A depth-first search (DFS) begins at a specific starting vertex s. The vertex s is marked as "visited" and becomes the "current" vertex; then the procedure is repeated. To make the description simple, we will use u to denote the current vertex. In general, when we are on the current vertex, we randomly choose one of the edges that are incident to u. Let this edge be (u, v). If v has not been visited, vertex v is marked as visited, edge (u, v) becomes the "discovery" edge for vertex v, vertex v becomes the current vertex, and the process is repeated. If v has already been visited, we consider another edge incident to u. If all edges incident to u have been visited, we roll back to the vertex that was the current vertex before u and repeat the process.

The pseudocode of a DFS algorithm is shown below:

Code Segment 6.1

```
Algorithm DFS (G, u)
Input: A graph G and a vertex u of G
Output: A collection of vertices reachable from u, with their discovery edges
Mark u as visited
for each of u’s outgoing edges, e = (u,v) do
  if v has not been visited then
    Record edge e as the discovery edge for vertex v
    Recursively call DFS(G, v)
```

Figure 6.14 illustrates a DFS on a directed graph. In the graph, when a vertex is visited, it is colored yellow, and a discovery edge is indicated with a thick red line. It is assumed that, at the current vertex u, an edge (u, v) is examined in alphabetical order of v (not randomly) from the edges incident to u.

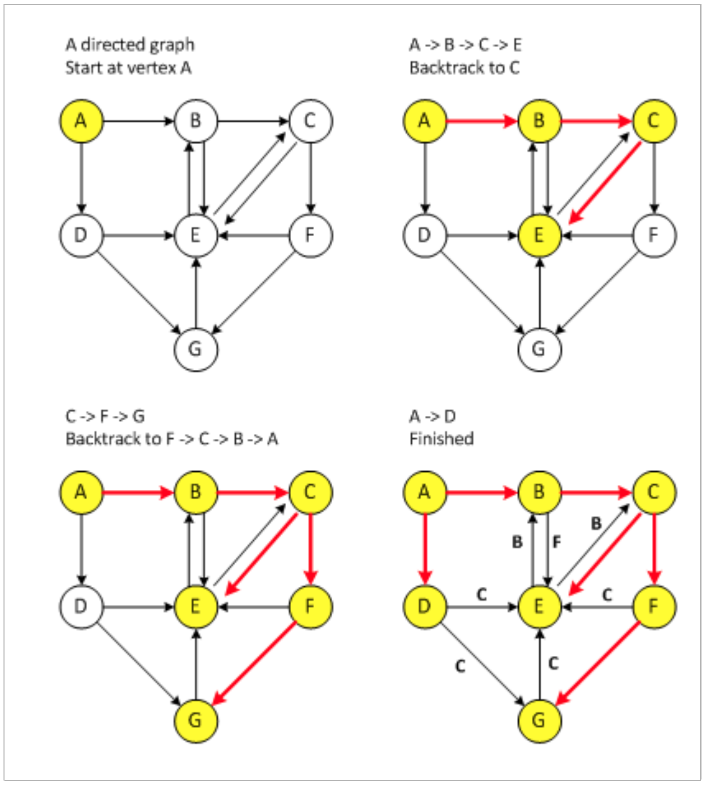


Figure 6.14. Illustration of a DFS on a directed graph.

The result of a DFS creates a tree, called a DFS tree, rooted at the starting vertex. The DFS tree is indicated with thick red lines, which are discovery edges. Discovery edges are also called *tree edges*. After a DFS is run on a directed graph, we can observe that there are other types of edges:

- *Back edges*—A back edge connects a vertex to its ancestor in the DFS tree. In the figure, back edges are labeled B in the last graph. For example, the edge (E, B) is a back edge because the vertex B is an ancestor of the vertex E.
- *Forward edges*—A forward edge connects a vertex to its descendant in the DFS tree. In the figure, forward edges are labeled F in the last graph. For example, the edge (B, E) is a forward edge because the vertex E is a descendant of the vertex B.
- *Cross edge*—A cross edge connects a vertex to a vertex that is neither its ancestor nor its descendant. Cross edges are labeled C in the figure. An example is the edge (D, C). The vertex C is neither D's ancestor nor D's descendant.

The following figure illustrates a DFS on an undirected graph:

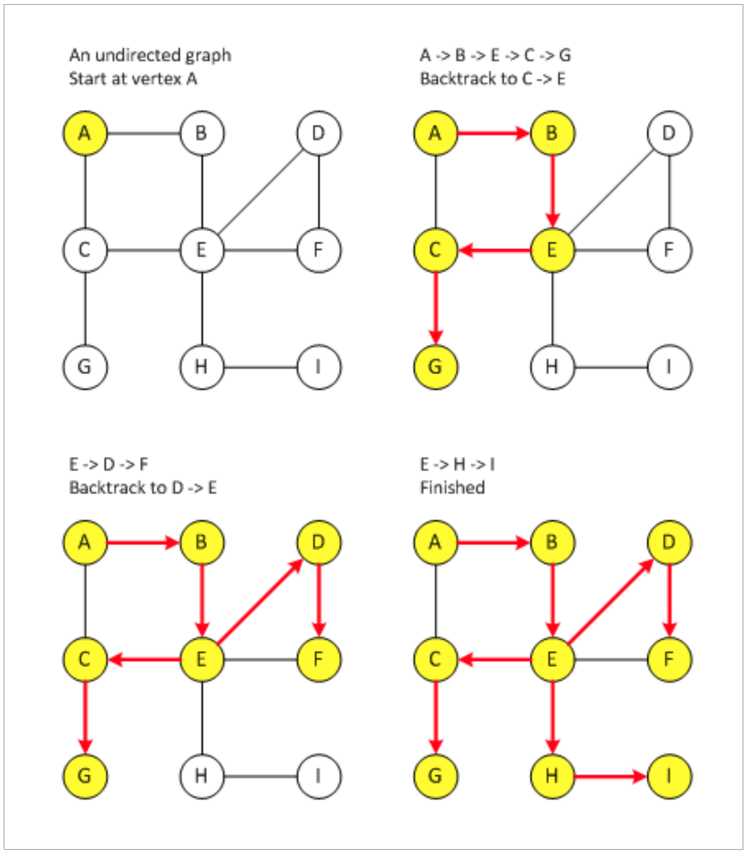


Figure 6.15. Illustration of a DFS on an undirected graph.

Properties of a DFS

A DFS has the following two useful properties:

- Let G be an undirected graph. A DFS on G starting at a vertex S visits all vertices in the connected component of S , and the discovery edges form a *spanning tree* of the connected component of S .
- Let G be a directed graph. A DFS on G starting at a vertex S visits all vertices reachable from S , and the DFS tree contains the directed paths from S to every vertex reachable from S .

Running Time of DFS

The DFS is called at most once on each vertex. In an undirected graph, each edge is examined at most twice, once for each endpoint. In a directed graph, each edge is examined at most once. Let $n_s \leq n$ be the number of vertices reachable from the starting vertex s, and $m_s \leq m$ be the number of edges that are incident to those vertices. Then the DFS runs in $O(n_s + m_s)$ time. This analysis assumes the following:

- Scanning outgoing edges of a vertex v takes $O(deg(v))$. Given an edge and one end vertex of the edge, finding the other end vertex takes $O(1)$. This can be achieved with appropriate data structures, such as adjacency lists or adjacency maps.
- There is a way to mark a vertex as visited in $O(1)$ time. This can be easily achieved by maintaining a set data structure, for example, which keeps vertices that were already visited. We will discuss this implementation issue in the next section.

We can solve the following interesting problems in $O(n + m)$ time using the DFS.

For an undirected graph G, we can solve the following:

- Find a path between given two vertices of G, if one exists.
- Test whether G is connected.
- Find a spanning tree of G, if G is connected.
- Find the connected components of G.
- Find a cycle in G, or report that G has no cycle.

For a directed graph G, we can solve the following:

- Find a directed path between given two vertices of G, if one exists.
- Find a set of vertices that are reachable form a given vertex s .
- Test whether G is strongly connected.
- Find a directed cycle in G, or report that G is acyclic.

Section 6.1.3.2. DFS Implementation and Extensions

We first describe how to implement the DFS in Java.

Two issues we need to resolve are (1) how to mark vertices as visited in $O(1)$ time, and (2) how to store DFS-tree edges.

For the first issue, we can use a hash-based, set data structure, which can answer whether a vertex has been already visited in $O(1)$ time. Alternatively, we can store the status of a vertex ((i.e., of whether a vertex has been visited) as an instance variable of a *Vertex* object. In the implementation we describe in this section, the first approach is used.

One way to resolve the second issue is to keep the tree edges in a hash table. In the implementation described here, a hash table with open addressing is used.

A Java code implementing the DFS is shown below:

Code Segment 6.2

```
public static <V,E> void DFS(Graph<V,E> g, Vertex<V> u,
                Set<Vertex<V>> known, Map<Vertex<V>,Edge<E>> forest) {
  known.add(u);                            // u has been discovered
  for (Edge<E> e : g.outgoingEdges(u)) {  // for every outgoing edge from u
    Vertex<V> v = g.opposite(u, e);
    if (!known.contains(v)) {
      forest.put(v, e);                // e is the tree edge that discovered v
      DFS(g, v, known, forest);        // recursively explore from v
    }
  }
}
```

This DFS code receives the following four arguments:

- g is a graph object on which the DFS is performed.
- u is the starting vertex.
- *known* is a *HashSet* object that stores vertices that have been already visited.
- *forest* is a *ProbeHashMap* object that keeps the DFS-tree edges. The *ProbeHashMap* implements a hash table using open

addressing with linear probing (discussed in Section 4.2.2.3).

In line 2, we mark the vertex u as visited (by inserting it into *known*). The *for* loop of lines 5 and 6 explore each vertex v that is adjacent to u. If v has not yet been visited (line 5), then the edge (u, v) is added to *forest* as a newly discovered tree edge in line 6. Then, we recursively call DFS on v in line 7.

In the *DFS* method, a hash table *forest* is used to store tree edges. We can use the hash table to reconstruct a path from a vertex u to a vertex v, if v is reachable from u. A Java code is shown below:

Code Segment 6.3

```
1   constructPath(Graph<V,E> g, Vertex<V> u, Vertex<V> v,
                Map<Vertex<V>,Edge<E>> forest) {
2     PositionalList<Edge<E>> path = new LinkedPositionalList<>();
3     if (forest.get(v) != null) {    // v was discovered during the search
4       Vertex<V> walk = v;           // we construct the path from back to front
5       while (walk != u) {
6         Edge<E> edge = forest.get(walk);
7         path.addFirst(edge);              // add edge to *front* of path
8         walk = g.opposite(walk, edge);    // repeat with opposite endpoint
9       }
10    }
11    return path;
12  }
```

Line 2 creates a *PositionalList* object *path*, which will store the path from u to v. We reconstruct the path backward, starting from the vertex v. The variable *walk*, which is a vertex, is used to walk backward (line 5). In line 6, the edge associated with *walk* is retrieved from the hash table *forest*. Initially, *walk* is v. Suppose that the last edge in the path is (w, v). Then line 6 retrieves that edge (w, v), and it is added to the path in line 7. Then line 8 retrieves the vertex at the other end and it is set to new *walk*, which is the vertex w. Then the same process is repeated in the *while* loop.

The above DFS method performs a DFS on a graph starting at a particular vertex, and it receives two auxiliary data structures to implement a recursion. It will visit all vertices reachable from the starting vertex.

When a graph is not connected, the DFS method stops once all vertices reachable from the given starting vertex are found. It does not explore vertices in other connected components in the graph. If we want to be able to continue the DFS on other connected components in the graph, we can write a separate method that invokes the *DFS* method. This method can also identify all connected components in an undirected graph. A Java code of this method is shown below:

Code Segment 6.4

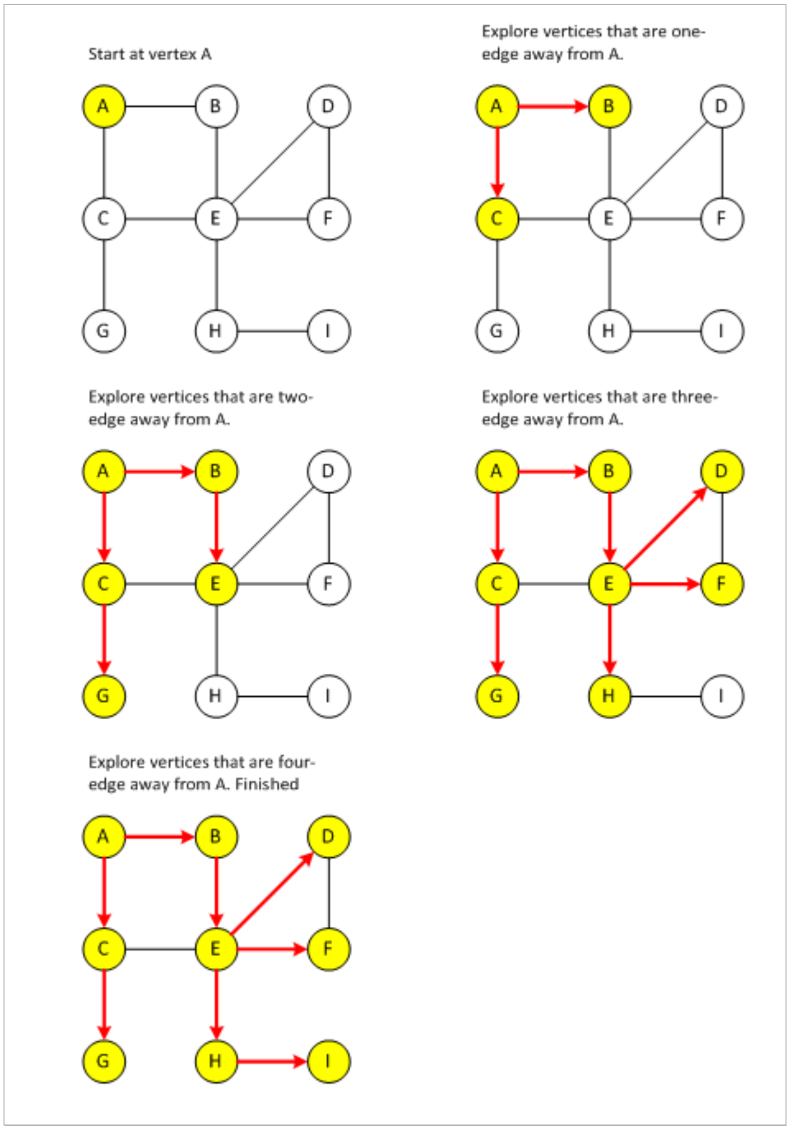
```
public static <V,E> Map<Vertex<V>,Edge<E>> DFSComplete(Graph<V,E> g) {
  Set<Vertex<V>> known = new HashSet<>();
  Map<Vertex<V>,Edge<E>> forest = new ProbeHashMap<>();
  for (Vertex<V> u : g.vertices())
    if (!known.contains(u))
      DFS(g, u, known, forest);           // (re)start the DFS process at u
7     return forest;
8   }
```

Lines 2 and 3 creates a *HashSet* object and a *ProbeHashMap* object, respectively, and these are passed to the DFS method. The *for* loop of line 4 explores all vertices in the given graph. So, all connected components are identified by this method.

Section 6.1.3.3. Breadth-First Search

In this section, we describe another graph traversal algorithm, called a *breadth-first search* (BFS). In a BFS, vertices are divided into *levels*. A traversal begins at a starting vertex S. At the beginning, S is placed at level zero and is marked as visited. In the first round, all vertices that are one edge away from S are explored. These vertices are placed at level one and are marked as visited. In the second round, all vertices that are two edges away from S (which are one edge away from level one vertices) are explored. For each vertex explored, if it has not been visited, it is placed at level two and marked as visited. The process is repeated until no newer vertices are found. Whenever a vertex is marked as visited, the corresponding edge becomes a discovery edge, as was done in the BFS. After a BFS is completed on a graph G, the discovery edges form a BFS tree rooted at S.

This search process is illustrated in the following figure. Those vertices that have been visited are colored yellow and discovery edges are drawn as thick red lines. Note that the last graph shows the BFS tree that is rooted at vertex A and formed by all discovery edges.


Start at vertex A

Explore vertices that are one-edge away from A.

Explore vertices that are two-edge away from A.

Explore vertices that are three-edge away from A.

Explore vertices that are four-edge away from A. Finished

Figure 6.16. Illustration of a BFS.

A Java code implementing the BFS is shown below:

Code Segment 6.5

```
public static <V,E> void BFS(Graph<V,E> g, Vertex<V> s,
                Set<Vertex<V>> known, Map<Vertex<V>,Edge<E>> forest) {
  PositionalList<Vertex<V>> level = new LinkedPositionalList<>();
  known.add(s);
  level.addLast(s);                         // first level includes only s
  while (!level.isEmpty()) {
    PositionalList<Vertex<V>> nextLevel = new LinkedPositionalList<>();
    for (Vertex<V> u : level)
      for (Edge<E> e : g.outgoingEdges(u)) {
        Vertex<V> v = g.opposite(u, e);
        if (!known.contains(v)) {
          known.add(v);
12            forest.put(v, e);        // e is the tree edge that discovered v
13            nextLevel.addLast(v);    // v will be further considered in next pass
14          }
15        }
16      level = nextLevel;             // relabel 'next' level to become the current
17    }
18  }
```

Like the DFS code, which was discussed in the previous section, this implementation also uses two auxiliary data structures, *known* and *forest*, to store already-visited vertices and discovery edges, respectively. It also uses two *PositionalList* objects to keep track of *levels*. As described above, the BFS explores vertices level by level. The *level* object keeps the vertices that are at the level currently being explored. The *nextLevel* stores vertices that will be explored at the next round (these vertices are one-edge away from those at the current level).

The *for* loop of lines 7 through 15 explores each vertex u at the current level. The *for* loop of lines 8 through 14 examines all outgoing edges of u. Suppose that (u, v) is an outgoing edge of u. If a vertex v has not been visited yet (line 10), then it is marked as visited (line 11), the edge (u, v) is added to *forest* as a discovery edge (line 12), and v is added to the next level. After all vertices at the current level are explored, the *nextLevel* becomes the new current level (in line 16), and the process is repeated (in the *while* loop).

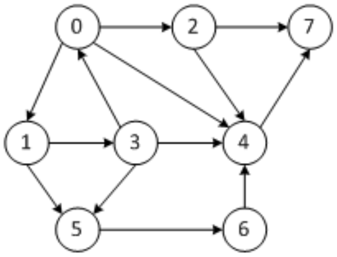
One interesting property of the BFS is that the path from the starting vertex s to any other vertex v is the shortest path from s to v in terms of the number of edges. After a BFS is performed on a graph G starting at vertex s, the following are some properties that hold:

- The traversal visits all vertices reachable from s .
- For each vertex v at level i, the path in the BFS tree from s to v has i edges, and any other path from s to v in G has at least i edges.
- If (u, v) is an edge that is not in the BFS tree, the level number of v is at most one greater than the level number of u.

The running time of a BFS is $O(n + m)$, which can be obtained via an analysis similar to the one of a DFS.

Test Yourself 6.1

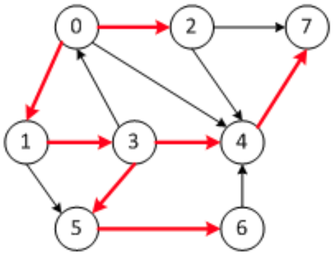
Consider the following directed graph:



Execute the depth-first search algorithm, which is described in Section 6.1.3.1 (this algorithm is also described in Section 14.3.1 of the textbook), on the above graph and highlight the resulting tree edges with thick lines. Start at the vertex 0. Assume that, at the current vertex *u*, an edge (*u*, *v*) is examined in the order of the numeric value associated with *v* (not randomly) from the edges incident to *u*.

Please think carefully, write your answer, and then click "Show Answer" to compare yours to the possible algorithm.

Suggested answer: Tree edges are highlighted with red thick lines.



Section 6.1.4. Topological Ordering

A directed graph without cycles is called a *directed acyclic graph* DAG. An interesting application on a DAG is to find the *topological ordering* of vertices in it.

Given a directed acyclic graph G with n vertices, a topological ordering of G is $v_1, v_2, \ldots, v_n$ of the vertices of G such that for every edge (v_i, v_j) in $G, i < j$. In other words, for every edge (u, v) in G, u always appears before v in the topological ordering. If there is a cycle in G—that is, if G is not a DAG—then there is no topological ordering of G's vertices.

We describe an algorithm that finds a topological ordering of a directed acyclic graph G. A sketch of the algorithm is shown below:

- Step 1—We first create a list L, which will eventually contain all vertices of G in the topological ordering.
- Step 2—Since G has no cycle, there must be at least one vertex that does not have an incoming edge. Let v be such a vertex. We remove v from G and add it to the end of L.
- Step 3—We also remove all outgoing edges of v.
- Step 4—The resulting graph is still acyclic. So, we go back to Step 2 and continue.

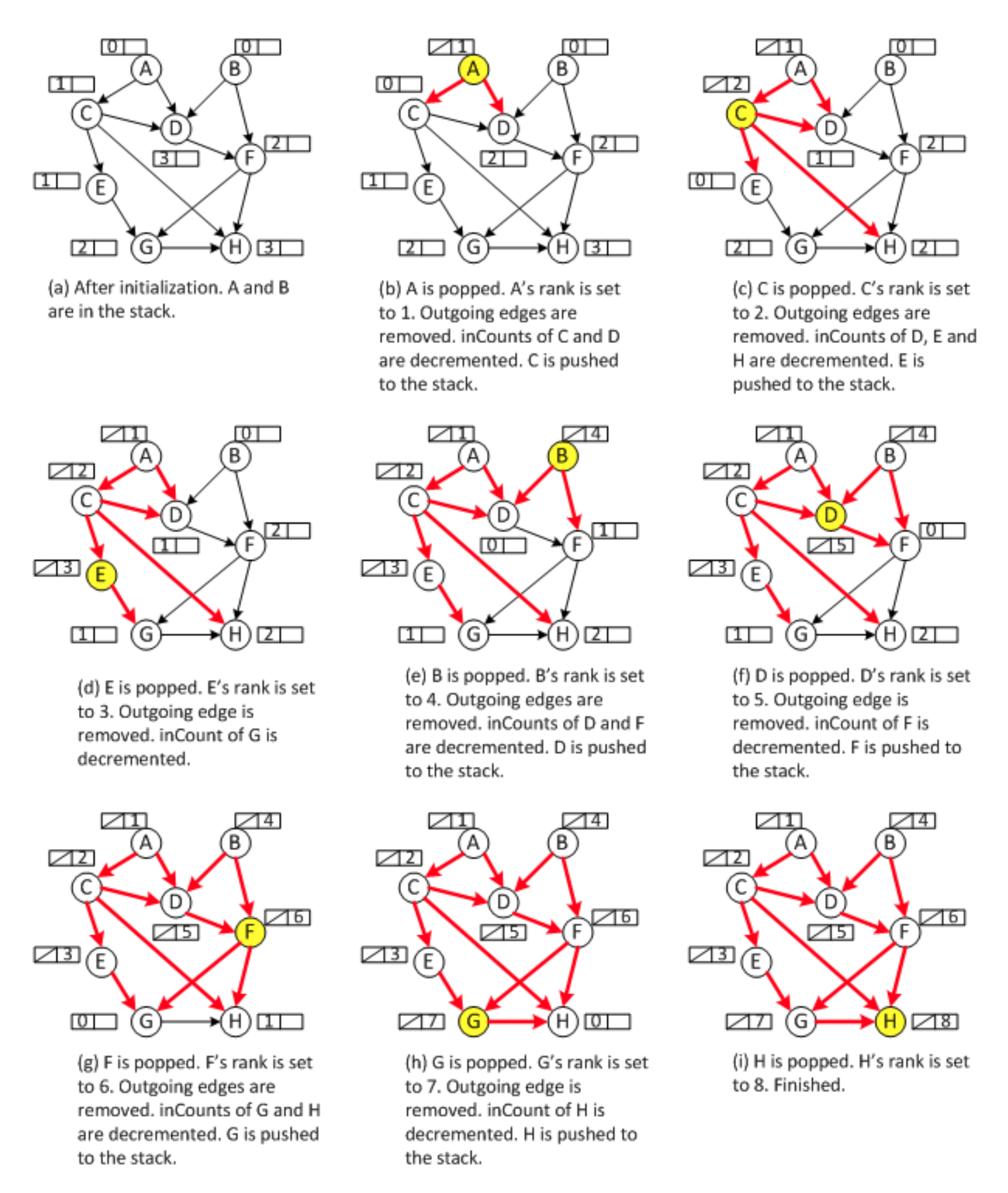
A Java code implementing the algorithm is shown below:

Code Segment 6.6

```
public static <V,E> PositionalList<Vertex<V>> topologicalSort(Graph<V,E> g) {
  // list of vertices placed in topological order
  PositionalList<Vertex<V>> topo = new LinkedPositionalList<>();
  // container of vertices that have no remaining constraints
  Stack<Vertex<V>> ready = new LinkedStack<>();
  // map keeping track of remaining in-degree for each vertex
  Map<Vertex<V>, Integer> inCount = new ProbeHashMap<>();
  for (Vertex<V> u : g.vertices()) {
    inCount.put(u, g.inDegree(u));       // initialize with actual in-degree
    if (inCount.get(u) == 0)             // if u has no incoming edges,
      ready.push(u);                     // it is free of constraints
  }
  while (!ready.isEmpty()) {
    Vertex<V> u = ready.pop();
    topo.addLast(u);
    for (Edge<E> e : g.outgoingEdges(u)) { // consider all outgoing neighbors
17        Vertex<V> v = g.opposite(u, e);
18        inCount.put(v, inCount.get(v) - 1);  // v has one less constraint without u
19        if (inCount.get(v) == 0)
20          ready.push(v);
21      }
22    }
23    return topo;
24  }
```

Line 3 creates a list, *topo*, which will contain all vertices in the topological ordering. The variable *ready* in line 5 is a stack, which will contain vertices that do not have incoming edges. The *for* loop in lines 8 through 12 initializes in-degrees of all vertices. If a vertex does not have any incoming edge, that vertex is pushed to the stack *ready* in line 11. In the *while* loop of lines 13 through 22, we repeatedly pop a vertex from the *ready* stack and perform the following. Let u be the vertex that is popped up from the stack (line15). For each outgoing edge e of u (line 16), we decrement the in-degree of the opposite vertex v (because all outgoing edges of u are removed) in line 18. If the new in-degree (after the decrement) of v is zero, then it is pushed to the stack. This is repeated in the *while* loop until the stack is empty (or all vertices are processed).

Figure 6.17 shows an example run of the *topologicalSort*. Each vertex has a small rectangular label with two components. A number on the left is the in-degree of the vertex. We call this the *inCount*. The inCount of a vertex decreases as its incoming edges are removed. The second number, on the right, is the position of the vertex in the final topological ordering. We call this the *rank*. The vertex that is popped is highlighted yellow. When an edge is removed, it is indicated by a thick red line.



(a) After initialization. A and B are in the stack.

(b) A is popped. A's rank is set to 1. Outgoing edges are removed. inCounts of C and D are decremented. C is pushed to the stack.

(c) C is popped. C's rank is set to 2. Outgoing edges are removed. inCounts of D, E and H are decremented. E is pushed to the stack.

(d) E is popped. E's rank is set to 3. Outgoing edge is removed. inCount of G is decremented.

(e) B is popped. B's rank is set to 4. Outgoing edges are removed. inCounts of D and F are decremented. D is pushed to the stack.

(f) D is popped. D's rank is set to 5. Outgoing edge is removed. inCount of F is decremented. F is pushed to the stack.

(g) F is popped. F's rank is set to 6. Outgoing edges are removed. inCounts of G and H are decremented. G is pushed to the stack.

(h) G is popped. G's rank is set to 7. Outgoing edge is removed. inCount of H is decremented. H is pushed to the stack.

(i) H is popped. H's rank is set to 8. Finished.

Figure 6.17. Illustration of a topological sort.

Section 6.1.5. Shortest Paths

We saw in Section 6.1.3.3 that the breadth-first search finds a shortest path, in terms of the number of edges, from the starting vertex to every other vertex. In some applications, edges are associated with numeric values, which are referred to as *weights*. In such applications, the shortest path between two vertices may not be defined in terms of the number of edges. For example, in a road network, a shortest path between two cities may be defined in terms of the miles a car has to drive from one city to the other. In this section, we describe an algorithm that finds the shortest path between two vertices in a graph, which has numeric values associated with edges.

Section 6.1.5.1. Weighted Graph

A *weighted graph* is a graph that has a numeric label $w(e)$ associated with each edge e. The numeric label is called the *weight*. The following is an example of a weighted graph, which represents a road network between cities in America. The weight of an edge is the distance, in miles, between two cities.

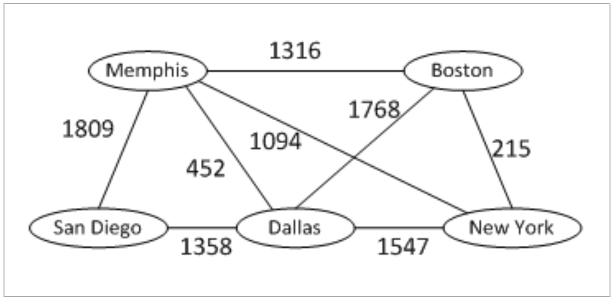


Figure 6.18. Example of a weighted graph.

Let G be a weighted graph.

- The *length* (or weight) of a path P is the sum of the weights of all edges on P. Let $P = \langle (v_0, v_1), (v_1, v_2), \ldots, (v_{k-1}, v_k) \rangle$. Then the length of P, denoted $w(P)$, is defined as follows:
$$w(P) = \sum_{i=0}^{k-1} w(v_i, v_{i+1})$$
- The *distance* from a vertex u to a vertex v in G, denoted $d(u, v)$, is the length of a minimum-length path from u to v, if such path exists. The minimum-length path is referred to as the *shortest path*.
- $d(u, v) = \infty$, if there is no path from u to v in G.

In general, the weights of edges of a graph can be negative numbers. If negative weights are allowed in a graph, then we can have a cycle with a negative length, called a *negative-weight cycle*. Consider the following directed graph:

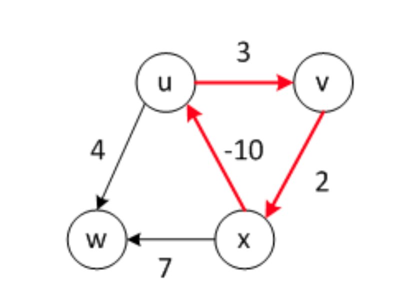


Figure 6.19. A graph with a negative cycle.

The cycle $C = \langle (u, v), (v, x), (x, u) \rangle$ is a negative-weight cycle because its length is –5 . Suppose that we want to find a shortest path from u to w. It may look like the shortest path is $\langle (u, w) \rangle$, the length of which is 4. However, many paths have shorter lengths. For example, consider a path $P = \langle (u, v), (v, x), (x, u), (u, v), (v, x), (x, u), (u, v), (v, x), (x, w) \rangle$. Note that this path includes two rounds of the cycle C. The length of this path is $= 3 + 2 + (-10) + 3 + 2 + (-10) + 3 + 2 + 7 = 2$. If we add one more round of the cycle C to P, then the length of P will be decreased by 5. So, if there is a negative-weight cycle in a path between two vertices, the shortest path between the two vertices is not defined. Therefore, we assume that there is no negative-weight cycle in a graph.

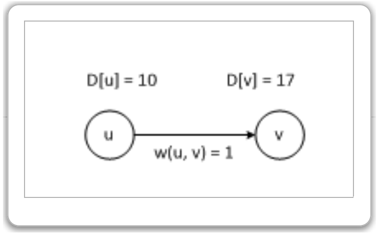
Section 6.1.5.2. Dijkstra's Algorithm

The Dijkstra's algorithm is a greedy algorithm that finds a shortest path from a given vertex s to every other vertex in a graph that does not have negative-weight edges. The vertex s is referred to as the *source* vertex.

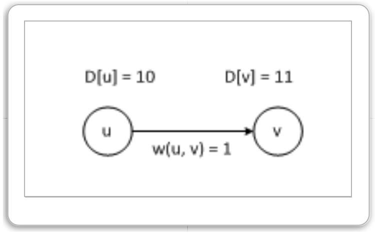
Edge Relaxation

Before we describe the Dijkstra's algorithm, we first describe an operation called *edge relaxation*. Each vertex v is associated with a label $D[v]$. $D[v]$ is an estimated distance from the source vertex s to v. It is the length of the best path from s to v that we have found so far. Initially $D[s] = 0$ and $D[v] = \infty$ for all other vertices. During the execution of the algorithm, $D[v]$ is iteratively updated. In each iteration, if we find a path from s to v that is shorter than the previously found path, then $D[v]$ is set to the length of the new, shorter path. This update is done by an edge relaxation, which works as follows.

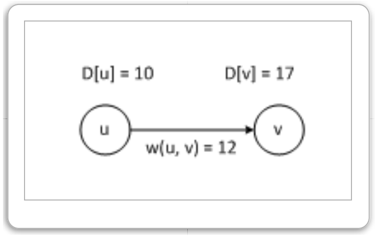
Consider the edge (u, v) in the following figure:



The current estimated distance from s to v is 17, via a certain path. If, however, we take the new path that goes to v via u, then the distance will be shorter—that is $D[v] = D[u] + w(u, v) = 11$. So, we update the $D[v]$ as shown below:



Consider the following configuration:



In this case, taking the path that goes via u does not reduce $D[v]$, so we do not update it.

The edge-relaxation operation can be codified as follows:

```
if $D[u] + w(u, v) < D[v]$ then
    $D[v] = D[u] + w(u, v)$
```

Description of Dijkstra’s Algorithm

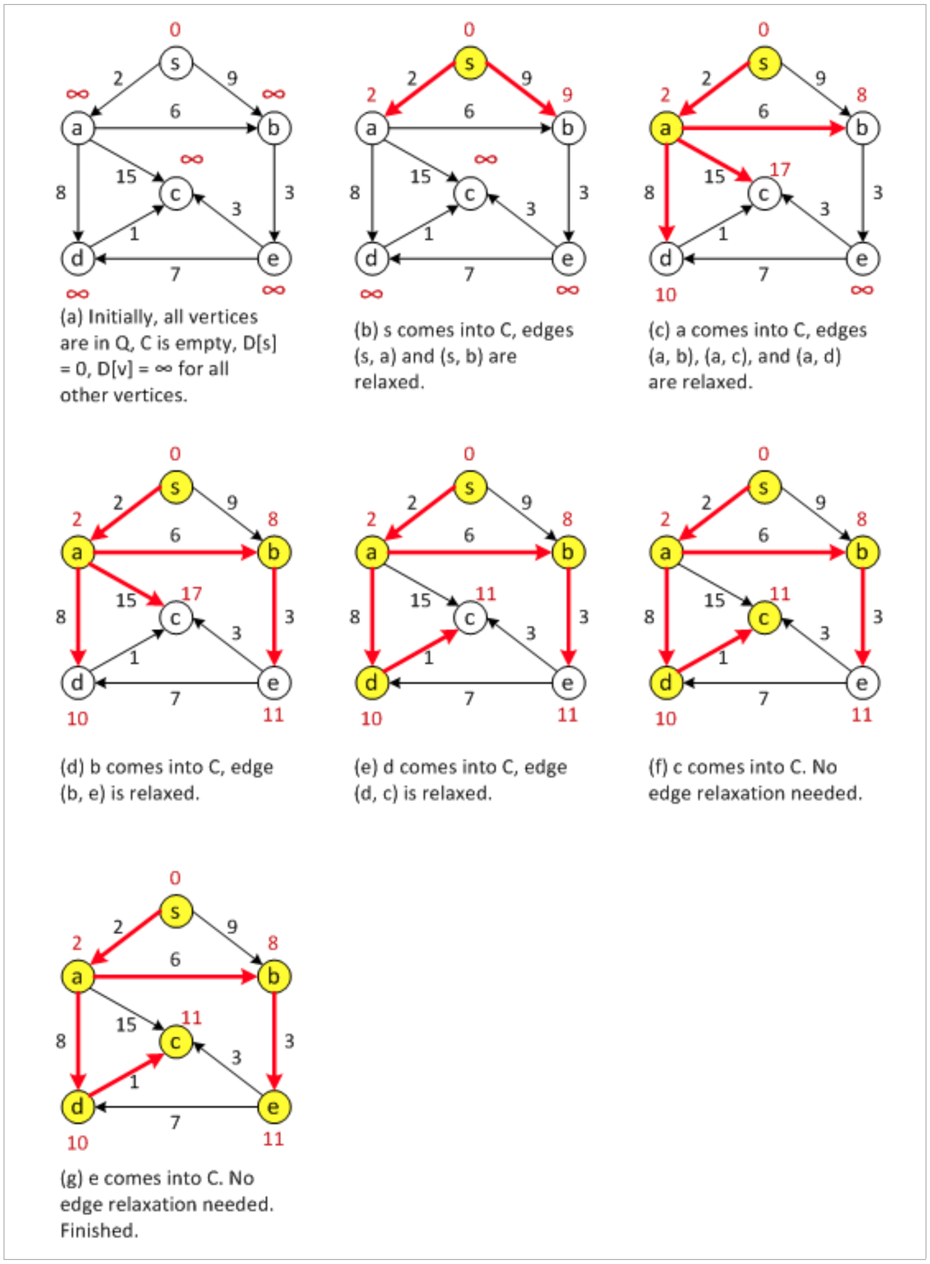
Now, we describe the algorithm. Let C be a set of vertices, called *cloud*. Initially, C is empty. In each iteration, we select a vertex u, not in C, with the smallest $D[u]$, and we pull it into C. In the first iteration, s is pulled into C. Once a new vertex u is pulled into C, we apply the edge-relaxation operation on each edge (u, v) for v that is not in Q. This is repeated until all vertices are pulled into C. The final value in $D[v]$ is the length of the shortest path from s to v for each v in G. This algorithm also creates a *shortest-path tree*. Whenever an edge (u, v) is relaxed, u becomes the parent of v in the shortest-path tree. We can reconstruct a shortest path from s to each v by following the path from s to v in the shortest-path tree.

The following is a pseudocode of the Dijkstra’s algorithm:

Code Segment 6.7

```
Algorithm ShortestPath(G, s):
Input: A directed or undirected graph G with nonnegative weights, and a
    distinguished vertex s of G
Output: The length of a shortest path from s to v for every vertex v of G
Initialize D[s] = 0 and D[v] = ∞ for each vertex v ≠ s
Let a priority queue Q contains all vertices of G using D labels as keys
while Q is not empty do
  u = Q.removeMin( ) // vertex with the smallest D[u] is pulled into “cloud”
  for each edge (u, v) such that v is in Q do
    // perform relaxation
    if D[u] + w(u, v) < D[v] then
      D[v] = D[u] + w(u, v)
      Change the key of vertex v in Q to D[v]
return the label D[v] of each vertex v
```

The following example illustrates the Dijkstra’s algorithm. The vertices that are in the cloud, C, are highlighted yellow. Thick red lines are the edges in the shortest-path tree.



(a) Initially, all vertices are in Q, C is empty, D[s] = 0, D[v] = ∞ for all other vertices.

(b) s comes into C, edges (s, a) and (s, b) are relaxed.

(c) a comes into C, edges (a, b), (a, c), and (a, d) are relaxed.

(d) b comes into C, edge (b, e) is relaxed.

(e) d comes into C, edge (d, c) is relaxed.

(f) c comes into C. No edge relaxation needed.

(g) e comes into C. No edge relaxation needed. Finished.

Figure 6.20. Illustration of the Dijkstra's algorithm.

Running Time of Dijkstra’s Algorithm

We analyze the running time of the Dijkstra’s algorithm in terms of queue operations.

- Initially, *n* vertices are inserted into Q.
- In the while loop, each of *n* vertices are removed.
- In the for loop, the keys of edges are changed, if needed. The total number of this operation—collectively, over all iterations of the while loop—is m, the number of edges.

If we implement Q as an adaptable priority queue (this is necessary because keys need to be updated) using a heap, each operation—insert, remove, and replace—takes $O(\log n)$ time. So, the total running time is the sum of the following three:

- n insert—$n \times O(\log n)$
- n remove—$n \times O(\log n)$

- m replace—$n \times O(\log n)$

That is, the running time of the Dijkstra's algorithm is $O((n+m)\log n)$ when the priority queue is implemented as an adaptable priority queue using a heap.

Java Implementation

The main body of a Java code implementing the Dijkstra's algorithm is shown below:

Code Segment 6.8

```
1   while (!pq.isEmpty()) {
2      Entry<Integer, Vertex<V>> entry = pq.removeMin();
3      int key = entry.getKey();
4      Vertex<V> u = entry.getValue();
5      cloud.put(u, key);                             // this is actual distance to u
6      pqTokens.remove(u);                            // u is no longer in pq
7      for (Edge<Integer> e : g.outgoingEdges(u)) {
8        Vertex<V> v = g.opposite(u,e);
9        if (cloud.get(v) == null) {
10          // perform relaxation step on edge (u,v)
11          int wgt = e.getElement();
12         if (d.get(u) + wgt < d.get(v)) {              // better path to v?
13            d.put(v, d.get(u) + wgt);                   // update the distance
14            pq.replaceKey(pqTokens.get(v), d.get(v));   // update the pq entry
15          }
16        }
17      }
18    }
```

The following are some data structures used in the code:

- d—A hash table storing (v, d) pairs. d is the distance from s to v. At the termination of the algorithm d is the shortest-path distance.
- pq—An adaptable priority queue implemented using heap. It keeps the vertices that are not processed yet. An entry in the queue stores a $(D[v], v)$ pair, where $D[v]$ is the key.
- *cloud*— A hash table storing vertices that are pulled into *cloud*.

An entry (with vertex u of line 4) is removed from the priority queue in ine 2 and added to the cloud in line 5. In the *for* loop of line 7, each vertex v that is adjacent to u (line 8) is examined.

If v is not in the cloud (line 9), relaxation is performed on (u, v). If necessary, the distance from s to v is updated in *d* in line 13, and it is also updated in the priority queue in line 14.

A complete code implementing the Dijkstra's algorithm can be found at the *GraphAlgorithms.java* file. In the file, it is implemented as the *shortestPathLengths* method. Note that this file includes other graph algorithms discussed in the textbook, in addition to the shortest-path algorithm.

Section 6.1.6. Minimum Spanning Trees

Given a tree T in an undirected, weighted graph G, the weight of T, $w(T)$,is defined as follows:

$$w(T) = \sum_{(u,v)\ in\ T} w(u,v)$$

Recall that a spanning tree of a graph G is a tree that contains all vertices of G.

A *minimum spanning tree* of an undirected, weighted graph G is a spanning tree with the minimum weight. The minimum-spanning-tree problem is to find such a tree in G.

We discuss two algorithms in this section: th ePrim-Jarnik algorithm and the Kruskal's algorithm.

We assume that a graph G is undirected, weighted, connected, and simple.

An Important Property Regarding Minimum Spanning Trees

Both algorithms are greedy algorithms and they rely on the following property.

Consider a partition of the vertices G into two disjointed nonempty sets, V_1 and V_2. An edge is called a *bridge* edge if one end is in V_1 and the other is in V_2. A bridge edge e is called a *minimum-weight edge* if the weight of e, $w(e)$, is the minimum among all bridge edges of the partition. In the following figure, there are three bridge edges that connect V_1 and V_2. Among them, the edge $e = (x, y)$ has the minimum weight. So, *e* is the minimum-weight edge.

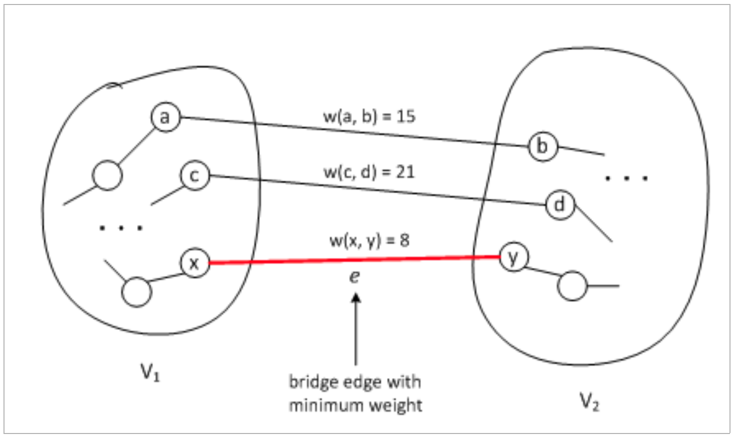


Figure 6.21. Minimum-weight edge.

The following property holds:

- Let G be a weighted connected graph and let V_1 and V_2 reflect a partition of the vertices of G into two nonempty sets. If e is a minimum-weight edge, then there is a minimum spanning tree T that has e as one of its edges.

In general, there can be multiple minimum spanning trees in a graph. However, if the weights in G are distinct, then the minimum spanning tree is unique.

Section 6.1.6.1. Prim-Jarnik Algorithm

The Prim-Jarnik algorithm begins at some "root" vertex s. This algorithm proceeds like the Dijkstra's algorithm does. It keeps a set of vertices C, called "cloud." Initially, C has only vertex s. In each iteration, we find a minimum-weight edge connecting a vertex u in the cloud of C and a vertex v that is outside the cloud. Then the vertex v is pulled into C, and this process is repeated until a spanning tree is formed. Since we always select a minimum-weight edge when bringing a vertex into C, according to the property stated in the previous section, the resulting tree is guaranteed to be a minimum spanning tree.

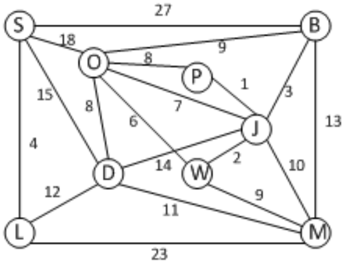
For efficient implementation, we maintain a priority queue Q to keep vertices that are not brought into C yet. We also maintain a label $D[v]$ for each v. $D[v]$ stores the weight of the minimum observed edge connecting v to the cloud C. These labels, $D[v]$, are used as keys in the priority queue. If we choose a vertex in the priority queue with the minimum $D[v]$, then it is a minimum-weight edge.

The following is a pseudocode of the Prim-Jarnik algorithm:

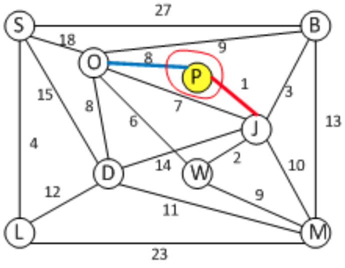
Code Segment 6.9

```
Algorithm PrimJarnik(G)
Input: An undirected, weighted, connected graph G with n vertices and m edges
Output: A minimum spanning T of G
Pick a vertex s of G          // start at this vertex
D[s] = 0
for each vertex v ≠ s do
  D[v] = ∞
Initialize T = ∅
Initialize a priority Q with entry (D[v], v) for each vertex v
For each vertex v, maintain connect(v) as the edge achieving D[v] (if any)
while Q is not empty do
  Let u be the value of the entry returned by Q.removeMin( )
  Connect vertex u to T using the edge connect(u)
  for each edge e’ = (u, v) such that v is in Q do
    if w(u, v) < D[v] then
      D[v] = w(u, v)
      connect(v) = e’
      Change the key of vertex v in Q to D[v]
return T                    // this is a minimum spanning tree
```

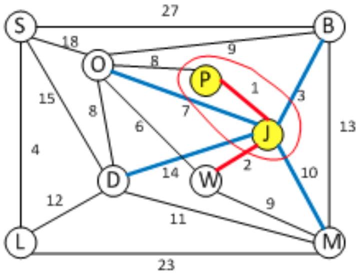
The following figure illustrates the Prim-Jarnik algorithm. The algorithm starts at the vertex P. The cloud is shown as a bag with a thin red outline. When a vertex is brought into the cloud, it is marked in yellow. The bridge edges are drawn with thick blue lines. Note that if a vertex outside the cloud has multiple edges connecting itself to the cloud, then only the edge with the smallest weight is marked in blue. A minimum-weight edge is shown with a thick red line, and all edges that are included in T are also drawn in thick red lines.



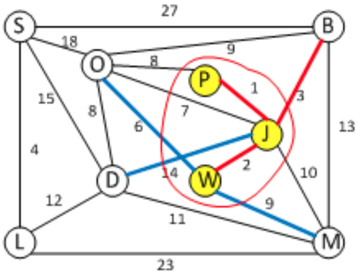
(a) Initial tree



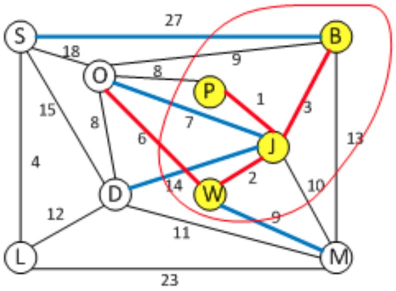
(b) (P,J) is minimum-weight edge.



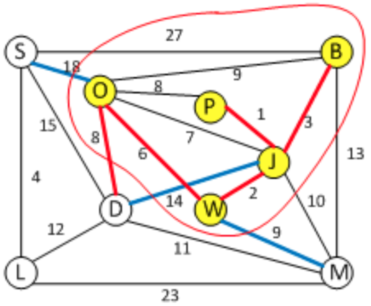
(c) (J,W) is minimum-weight edge.



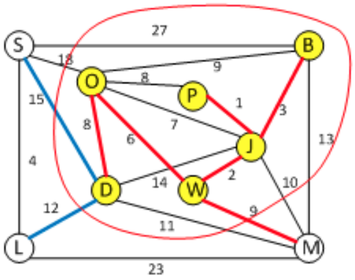
(d) (J,B) is minimum-weight edge.



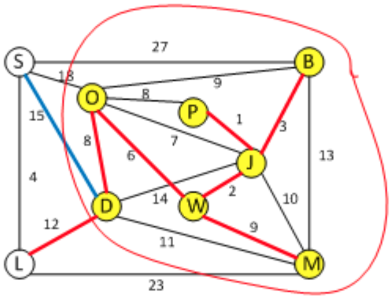
(e) (W,O) is minimum-weight edge.



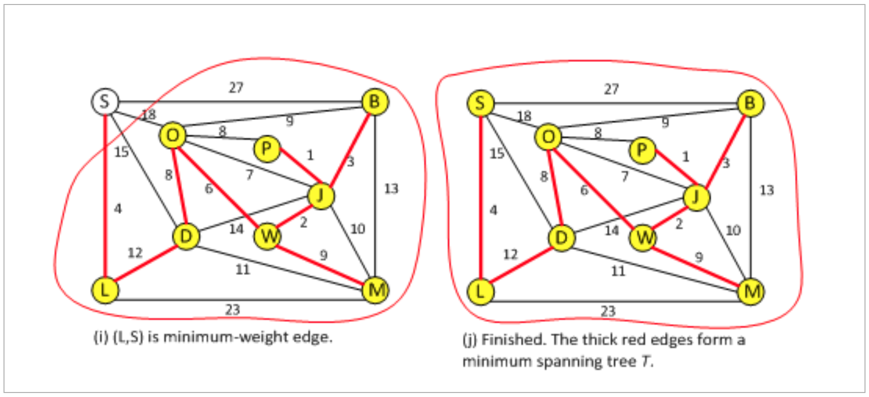
(f) (O,D) is minimum-weight edge.



(g) (W,M) is minimum-weight edge.



(h) (D,L) is minimum-weight edge.



(i) (L,S) is minimum-weight edge.

(j) Finished. The thick red edges form a minimum spanning tree *T*.

Figure 6.22. Illustration of the Prim-Jarnik algorithm.

Initially, n vertices are added to the priority queue; later, they are extracted one at a time. This needs $O(n)$ time. Keys in the priority queue are updated at most m times, collectively, during the entire execution of the algorithm. This takes $O(m)$ time. If we implement the priority queue using a heap, those operations take $O(\log n)$ time. So, the total running time of the Prim-Jarnik algorithm is $O((n+m)\log n)$.

Section 6.1.6.2. Kruskal's Algorithm

In the Prim-Jarnik algorithm, we begin with the root vertex and grow a tree until it contains all vertices. In the process of building a minimum spanning tree (MST), there is always a single tree. In the Kruskal's algorithm, there are multiple smaller trees at the beginning, forming a forest. Then we merge the smaller trees and eventually build a single tree, which becomes an MST.

Initially, a spanning tree T is empty, and each vertex is its own "cluster." Then we perform the following repeatedly:

- Step 1—Find an edge e with the smallest weight.
- Step 2—If two endpoints of e belong to different clusters, merge those two clusters.
- Step 3—Include e in T.
- Step 4—Stop if all vertices are included by T. Otherwise, return to Step 1 and repeat.

When we examine an edge at Step 1, we always choose an edge with the smallest weight among those that are not included in T.

The following shows a pseudocode of the Kruskal's algorithm.

Code Segment 6.10

```
Algorithm Kruskal(G)
  Input: A simple connected weighted graph G with n vertices and m edges
  Output: A minimum spanning tree T of G
  for each vertex v in G do
    Define an elementary cluster C(v) = {v}
  Initialize a priority Q with all edges in G, using the weights as keys
  T = ∅
  while T has fewer than n – 1 degree // not a spanning tree yet
    (u, v) = Q.removeMin();
    Let C(u) be the cluster containing u, let C(v) be the cluster containing v
    if C(u) ≠ C(v)  // u and v are in different clusters
      Add edge (u, v) to T
```

```
13          Merge C(u) and C(v) into one cluster
14      return T
```

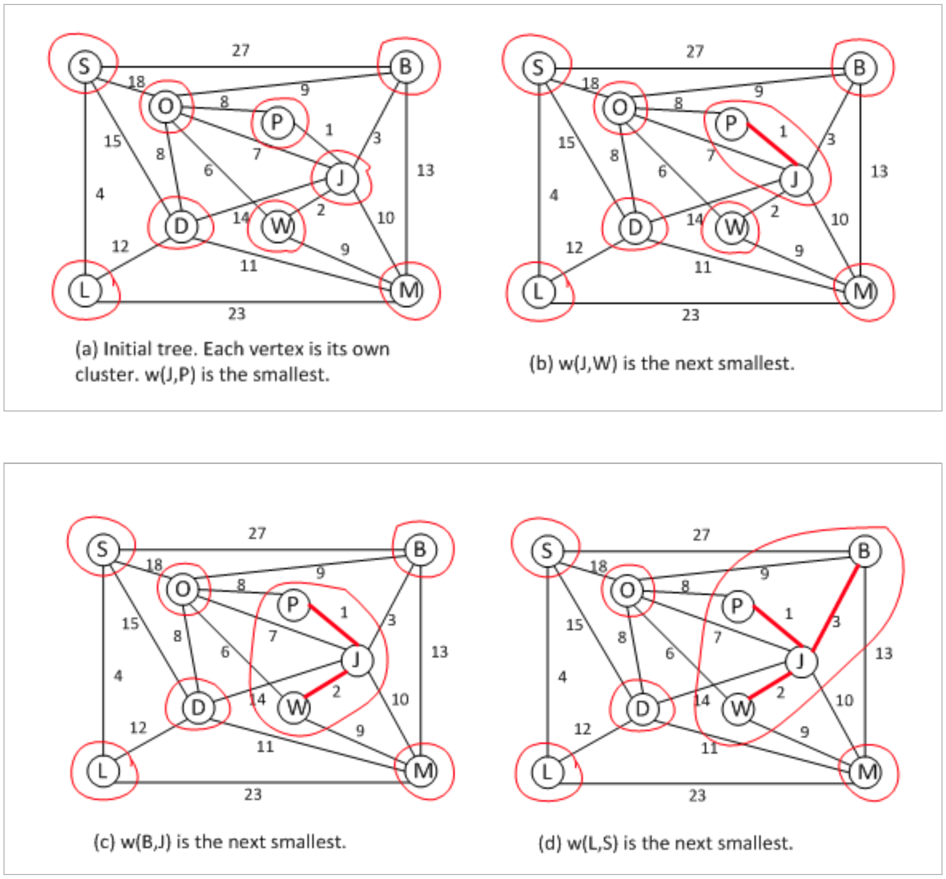
In lines 4 and 5, a cluster is created for each vertex. In line 6, all edges are added to a priority queue Q with their weights as keys. An edge with the smallest weight is at the front. The spanning tree is initially empty (line 7). In the *while* loop of line 8, edges are removed one at a time from Q, in non-decreasing order of weight, and examined. If two endpoints of an edge belong to two different clusters, then we add the edge to T and merge the two clusters into one cluster. If two endpoints of an edge belong to the same cluster, we ignore it because adding that edge to T will create a cycle in T.

The following figure illustrates an example run of the Kruskal's algorithm.

- Edges are examined in the following order (each entry in the following sequence shows an edge and its weight):

$$(J, P):1, (J, W):2, (B, J):3, (L, S):4, (O, W):6, (J, O):7, (D, O):8, (O, P):8, (B, O):9, (M, W):9, (J, M):10, (D, M):11, (D, L):12, (B, PM):13, (D, J):14, (D, S):15, (O, S):18, (L, M):23, (B, S):27$$

- A cluster is shown as a bag with a thin red boundary.
- When an edge is added to a spanning tree, the edge is drawn with a thick red line.

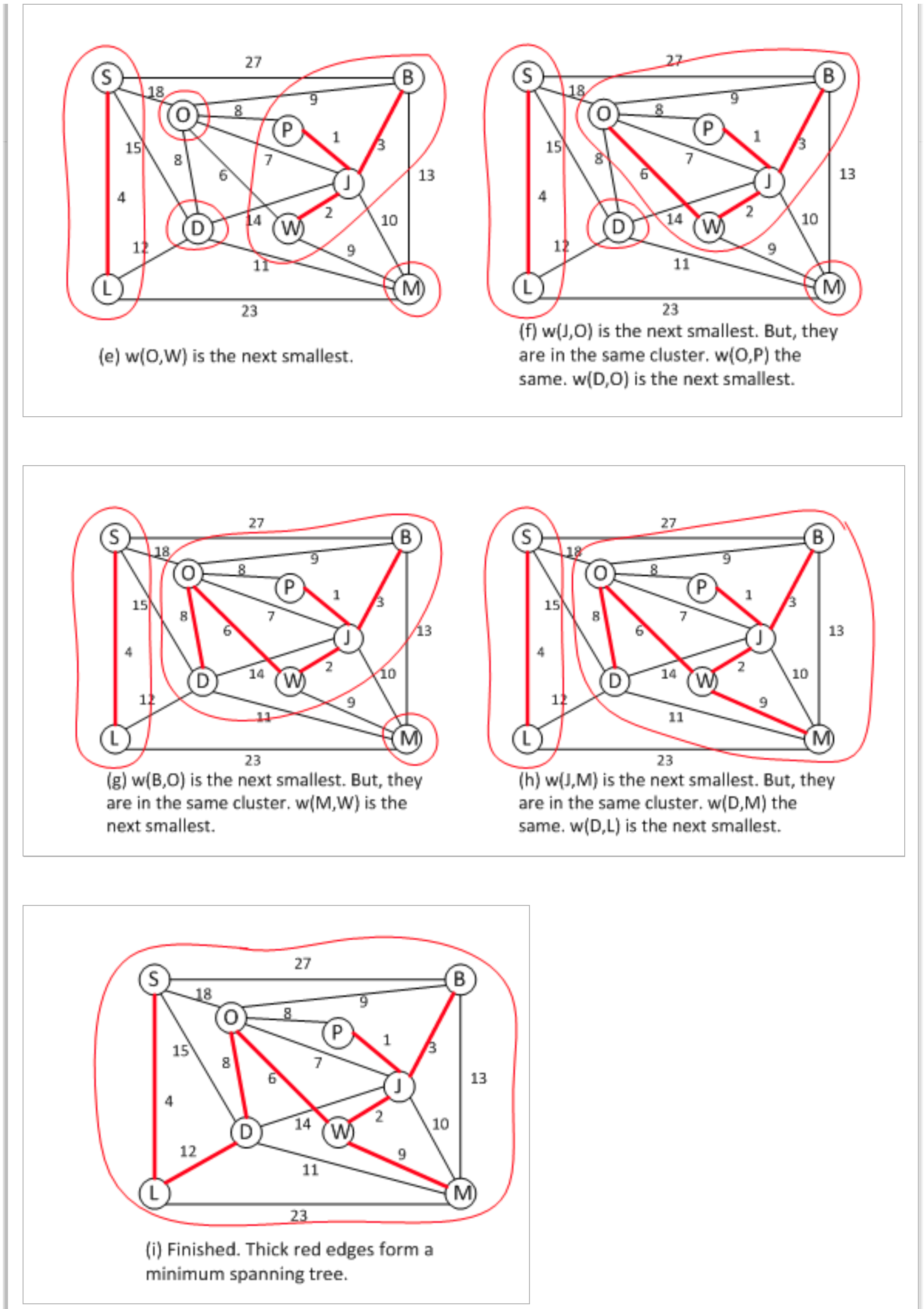


(a) Initial tree. Each vertex is its own cluster. w(J,P) is the smallest.

(b) w(J,W) is the next smallest.

(c) w(B,J) is the next smallest.

(d) w(L,S) is the next smallest.



(e) w(O,W) is the next smallest.

(f) w(J,O) is the next smallest. But, they are in the same cluster. w(O,P) the same. w(D,O) is the next smallest.

(g) w(B,O) is the next smallest. But, they are in the same cluster. w(M,W) is the next smallest.

(h) w(J,M) is the next smallest. But, they are in the same cluster. w(D,M) the same. w(D,L) is the next smallest.

(i) Finished. Thick red edges form a minimum spanning tree.

Figure 6.23. Illustration of the Kruskal's algorithm.

In the analysis of the running time of the Kruskal's algorithm, we need to consider two main components. The first component to manage the priority queue, which keeps all edges. Ordering edges takes $O(m \log m)$ time, and the *removeMin* method takes $O(\log m)$ time. In a simple graph, $O(\log m)$ is the same as $O(\log n)$ because m is $O(n^2)$. So, the first component takes $O(m \log n)$.

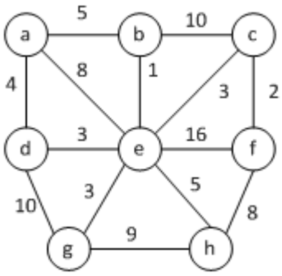
The second component is to manage clusters. This involves determining to which clusters an edge belongs and merging the two clusters. If we implement the management of clusters using the *disjoint partition* (or *disjoint set*) data structure, the second component, including all necessary operations for managing clusters, takes $O(m + n \log n)$ time.

Combining the two, we conclude that the Kruskal's algorithm takes $O(m \log n)$) time.

Note that we do not discuss the *disjoint partition* in this class. Interested students are referred to Section 14.73 of our textbook.

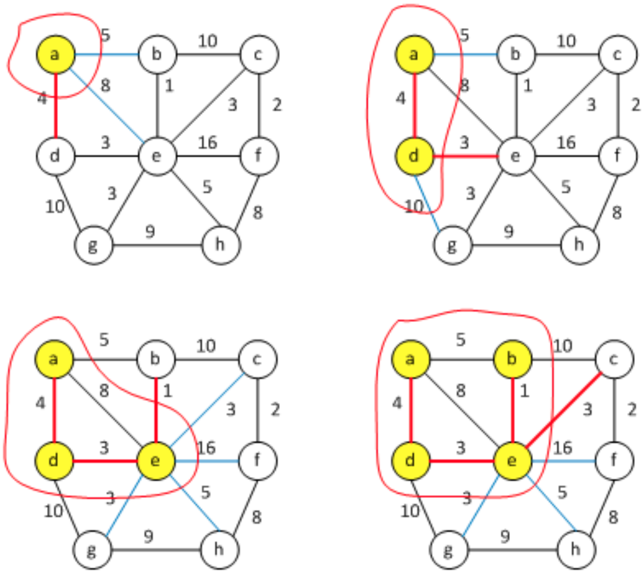
Test Yourself 6.2

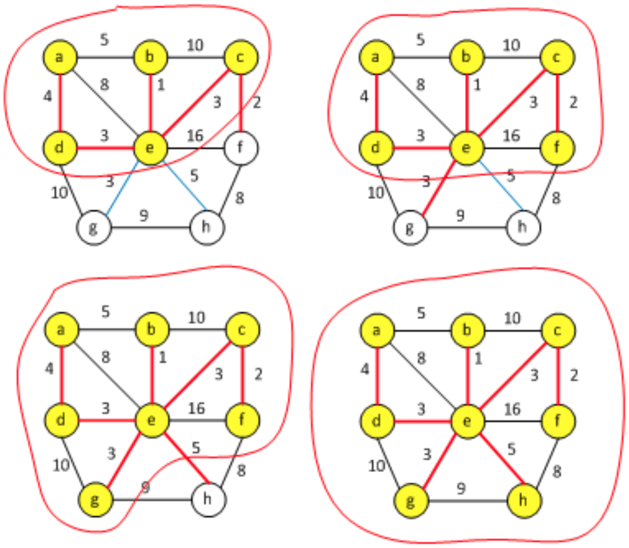
Illustrate the execution of the Prim-Jarnik algorithm on the following graph, using Figure 6.22 as a model. Start at the vertex *a*.



Please think carefully, figure out your answer, and then click "Show Answer" to compare yours to the possible algorithm.

Suggested answer:





Section 6.2 Computational Complexity

Section 6.2 Computational Complexity

Overview

In this section, we briefly discuss the basic concept of computational complexity.

Section 6.2.1. Decision Problem

A decision problem P is a set of questions, each of which has a *yes* or *no* answer. A simple example follows:

> A decision problem P_E—Determine whether an arbitrary, nonnegative number is an even number or not. This problem consists of the following questions:
>
> p_0—Is 0 even?
> p_1—Is 1 even?
> p_2—Is 2 even?
> $\ldots$
>
> Here, p_i is also called an instance of P.

A solution to a decision problem is an algorithm that determines the answer to every question $p_i \in P$.

An algorithm that solves a decision problem should meet the following criteria:

- *Complete*—It produces an answer, either positive or negative, to each question in the problem domain
- *Mechanistic*—It consists of a finite sequence of instructions, each of which can be carried out without requiring insight, ingenuity,

or guesswork.

- *Deterministic*—When presented with identical input, it always produces the same result.

A Turing machine is a formalism that can be used to model algorithms solving decision problems.

A decision problem is said to be solvable in polynomial time if there is at least one polynomially bounded algorithm that solves the problem. Such an algorithm is called an *efficient* algorithm, and a decision problem that is solved in polynomial time is also said to be *tractable*.

A decision problem is said to be *intractable* if there is no polynomially bounded algorithm (or no efficient algorithm) that solves the problem.

Solvable problems are equivalent to recursive languages so *decision problems* and *languages* are used interchangeably.

Section 6.2.2. Reducibility and P and NP

A decision problem P is Turing reducible to a problem P' if there is a Turing machine that takes any problem $p_i \in P$ as input and produces an associated problem $p'_i \in P$, whereby the answer to the original problem p_i can be obtained from the answer to p'_i.

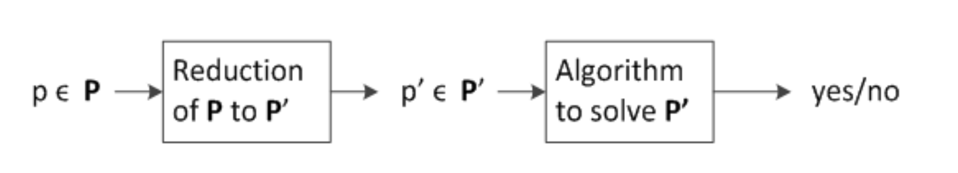


Figure 6.24. Reducing a decision problem.

A language L is decidable in polynomial time if there is a standard (or deterministic) Turing machine M that accepts L in polynomial time, or $O(n^r)$, where r is a natural number independent of n. The family of languages decidable in polynomial time is denoted P.

A deterministic machine solves a decision problem by generating a solution. A nondeterministic machine only needs to determine whether one of multiple possibilities is a solution.

We say a language L is accepted in nondeterministic polynomial time if there is a nondeterministic Turing machine that accepts L in polynomial time, or $O(n^r)$, where r is a natural number independent of n. The family of languages accepted in nondeterministic polynomial time is denoted NP.

NP can be alternatively defined as follows:

> A problem is in NP if it is "verifiable" in polynomial time. What "verifiable" means is that given a possible solution (which is also called a ***certificate***), we can verify whether it is a solution or not in polynomial time.

Whether $P = NP$ is an unsolved question. Since every deterministic machine is also nondeterministic, $P \subseteq NP$. But it was never proved that $NP \subseteq P$, nor that $NP = P$.

If a decision problem Q is reducible to another decision problem L in polynomial time, and $L \in P$, then $Q \in P$.

A problem L is called ***NP-hard*** if, for every $Q \in NP$, Q is reducible to L in polynomial time. An ***NP-hard*** problem that is also in NP is called ***NP-complete***.

If there is an ***NP-complete*** problem that is also in P, then $P = NP$.

Since an NP problem does not have an efficient algorithm to solve it, once we know a certain problem is NP, we try to develop an approximation algorithm(s). We describe a well-known ***NP-complete*** problem, known as the *traveling salesman problem* (*TSP*), and discuss approximation algorithms.

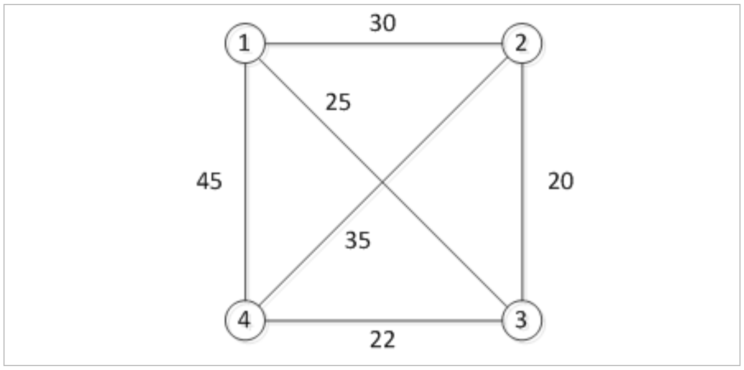
The following is the TSP:

> Given a complete, nonnegative weighted graph, find a *Hamiltonian cycle* of minimum weight.

Here, a *Hamiltonian cycle* of an undirected graph G is a simple cycle that contains all vertices of G.

The TSP is known to be ***NP-complete***. So, researchers developed approximation algorithms. We now discuss three approximation algorithms.

We use the following graph to illustrate the first two approximation algorithms:



Minimum-weight cycle $= 1 \to 2 \to 4 \to 3 \to 1$

Total weight $= 30 + 35 + 22 + 25 = 112$ (this is an optimal solution)

Figure 6.25. A sample graph to illustrate *TSP* approximation algorithms.

Approximation Algorithm 1: Nearest-Neighbor Strategy

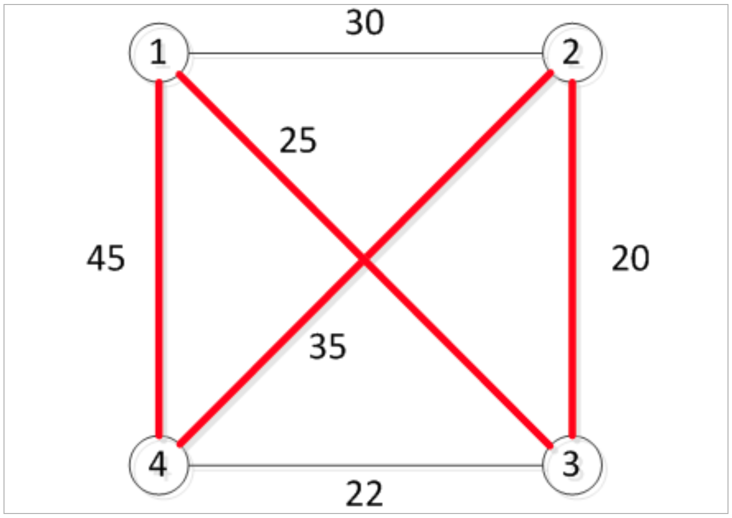
The following is a pseudocode of the nearest-neighbor-strategy algorithm:

Code Segment 6.11

```
Algorithm NEAREST-TSP(G, f)   /* f is a cost function, or a weight function */
select an arbitrary vertex s;
v = s; Q = {v}; S = V – Q; // V is all vertices of G
C = ∅;
while S != ∅
select an edge (v, w) of minimum weight, where w ∈ S;
C = C ∪ {(v, w)};
Q = Q ∪{w};
```

```
S = S – {w};
v = w;
C = C ∪ {(v, s)};
return C;
```

The running time of this algorithm is $O(n^2)$, where n is the number of vertices of the graph. Figure 6.26 illustrates the nearest-neighbor-approximation algorithm:



Starting at vertext $1 : (1, 3), (3, 2) (2, 4), (4, 1)$

Total weight $= 25 + 20 + 35 + 45 = 125$

Figure 6.26. Nearest-neighbor strategy.

Approximation Algorithm 2: Shortest-Link Strategy

The following is a pseudocode of the shortest-link-strategy algorithm:

Code Segment 6.12

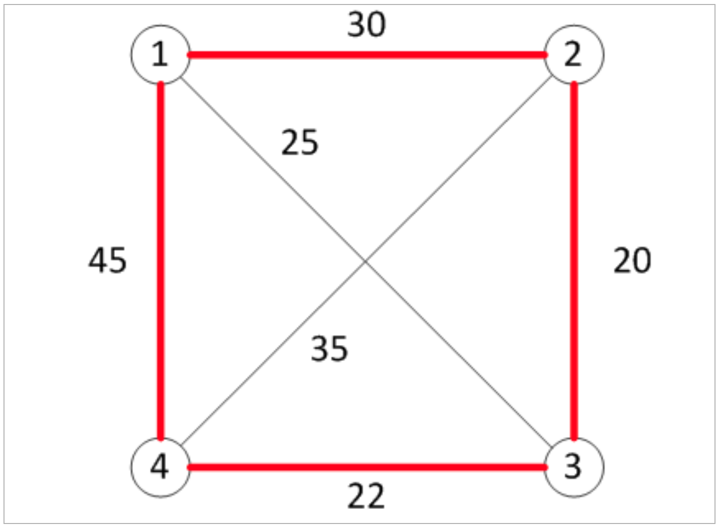
```
Algorithm SHORTEST-LINK-TSP(G, f)

 R = E; // E is all edges of G
 C = ∅
 while R != ∅
   choose the shortest edge (v, w) from R;
   if (v, w) does not make a cycle with edges in C and (v, w) would not be

                 the third edge in C incident on v or w
   then
         C = C + {(v, w)};
         R = R – {(v, w)};
```

```
    add the edge connecting the end points of the path in C;
    return C;
```

The running time of the shortest-link-approximation algorithm is $O(m \log n)$. The following figure illustrates the shortest-link-approximation algorithm:



Edges added: $(2,3), (3,4), (2,1), (1,4)$

Total weight $= 20 + 22 + 30 + 45 = 117$

Figure 6.27. Shortest-link strategy.

In general, we cannot establish a bound on how much the weight of an approximation algorithm differs from the weight of a minimum tour. If we assume the triangle inequality holds on distances among vertices, we can develop an approximation algorithm that has an upper bound on the weight. The following is the triangle inequality:

$f(u,v) \leq f(u,w) + f(w,v)$, for all $u, v, w \in V$ where V is all vertices of G

The Euclidean distance has the triangle-inequality property.

Approximation Algorithm 3

The following approximation algorithm has an upper bound on the weight: the total weight of a cycle is no more than the twice that of a minimum spanning tree's weight of the given graph. The following is a pseudocode of the algorithm:

Code Segment 6.13

```
Algorithm APPROX-TSP-TOUR(G, f)

select a vertex r ∈ V to be the root; // V is all vertices of G
compute MST T from r using Prim-Jarnik algorithm;
let H be a list of vertices, ordered according to when they are first visited
   in a preorder traversal of T;
```

```
    return H;
```

The following example illustrates this approximation algorithm:

a. Consider the following complete graph, in which there are edges from each vertex to all other vertices. For simplicity, edges are not shown in the graph. The weight of an edge is the Euclidean distance between the two endpoints of the edge.

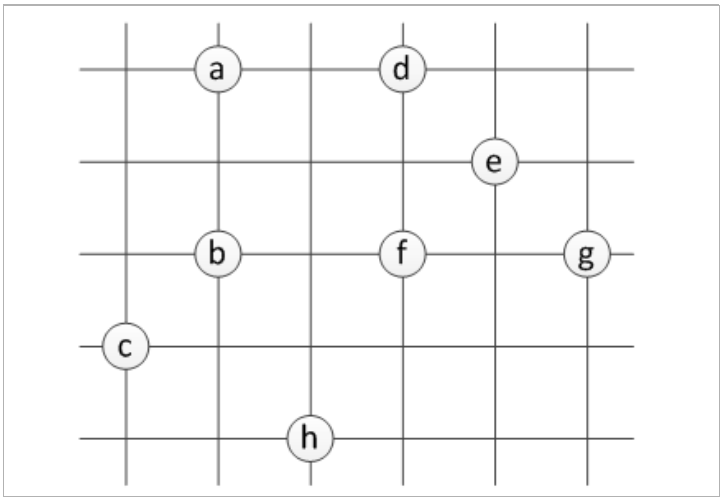


Figure 6.28. A complete graph (edges are not shown).

b. A minimum spanning tree T (a is the root)

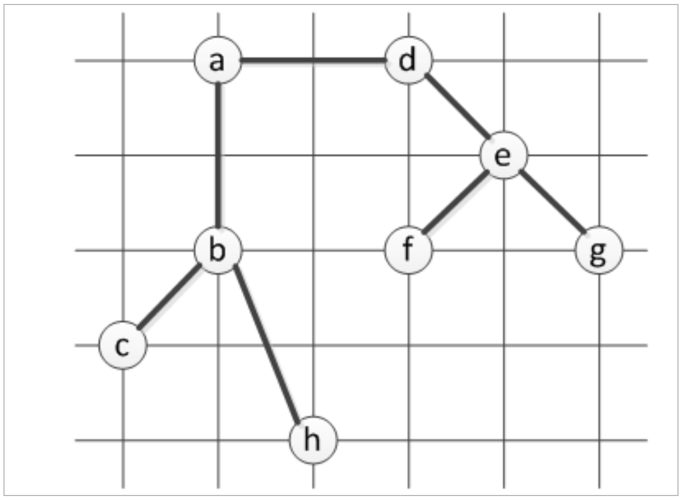


Figure 6.29. A minimum spanning tree T rooted at vertex a.

c. Preorder tree traversal (*a* is the root)

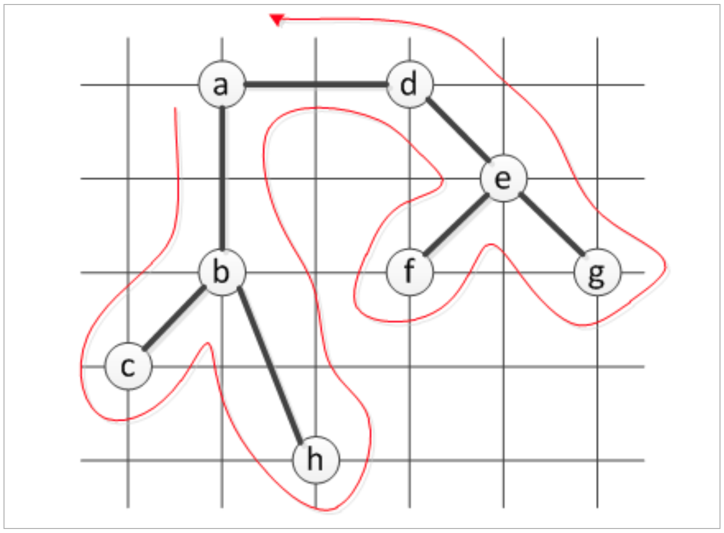
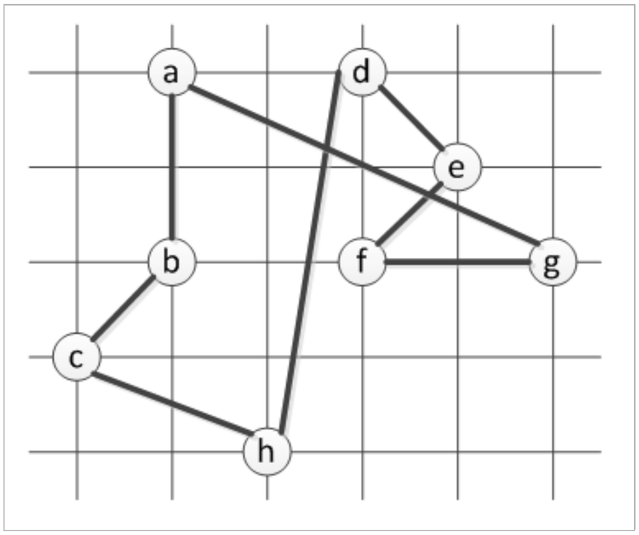


Figure 6.30. Preorder traversal of T.

The order of vertices visited by the preorder traversal is as follows:

$$a \to b \to c \to h \to d \to e \to f \to g \to a$$

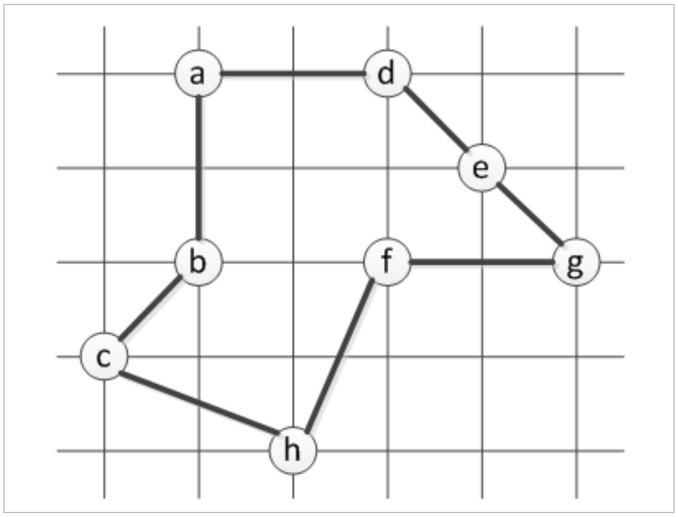
d. H returned by APPROX-TSP-TOUR



Total weight = approx. 19.074

Figure 6.31. Connected vertices following the preorder traversal of T.

e. An optimal tour (or a Hamiltonian cycle with a minimum weight)



Total weight = approx. 14.715

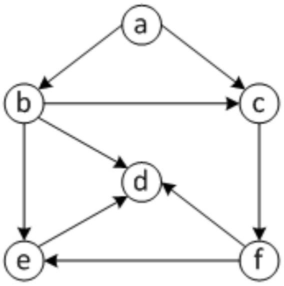
Figure 6.32. An optimal solution.

Module 6 Practice Questions

The following are some review questions for you to practice. Please read each question, think carefully, figure out your own answer or write your own program first, and then click "Show Answer" to compare yours to the suggested answer or the possible solution.

Test Yourself 6.3

Consider the following directed graph:



Suggested answer:

Test Yourself 6.4

A graph is said to be simple if it does not have parallel edges or self-loops. True or False?

True.

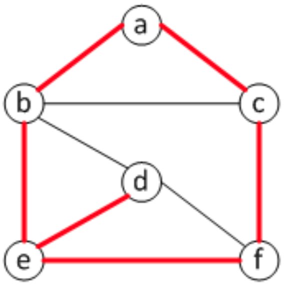
Your option is right.

False.

Your option is wrong.

Test Yourself 6.5

In the following graph, the red thick lines form a spanning tree. True or False?



True.

Your option is wrong.

False.

Your option is right. Since it contains all vertices, it is a spanning graph. However, it is not a spanning tree because it has a cycle.

Test Yourself 6.6

A *tree* is a connected graph without cycle. Prove that there is a unique path between any two vertices in a tree.

Suggested answer: We prove by contradiction.

Since a tree is a connected graph, there is at least one path between any two vertices. Assume that there are two distinct paths $p1$ and $p2$ between a vertex u and a vertex v. We can represent $p1 = \langle u, u1, u2, ...v\rangle$ and $p2 = \langle v, v1, v2, ...u\rangle$. Then, by combining the two paths, we have another path $p3 = \langle u, u1, u2, ...v, v1, v2, ...u\rangle$. This path $p3$ is a cycle (which starts at *u* and ends at u). This contradicts the definition of a tree. Therefore, there is a unique path between any two vertices in a tree.

Test Yourself 6.7

Suppose we represent a graph G having *n* vertices and m edges with the edge list structure. Why, in this case, does the *insertVertex* method take $O(1)$ time while the *removeVertex* method runs in $O(m)$ time?

Suggested answer: Inserting a vertex runs in $O(1)$ time since it is simply inserting an element into a doubly linked list. To remove a vertex, on the other hand, we must inspect all edges. Therefore, this will take $O(m)$ time.

Test Yourself 6.8

For each of the following cases, which data structure is more appropriate, adjacency matrix or adjacency list?

1. The graph has 10,000 vertices and 20,000 edges, and it is important to use as little space as possible.

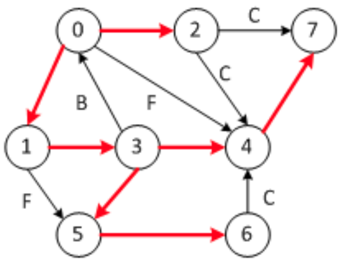
2. The graph has 10,000 vertices and 20,000,000 edges, and it is important to use as little space as possible.
3. You need to answer the query *getEdge*(*u*, *v*) as fast as possible, no matter how much space you use.

Suggested answers:

Test Yourself 6.9

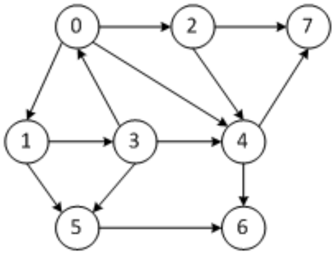
This question is a continuation of Question 6.1. Classify each nontree edge as *back edge*, *forward edge* or *cross edge*.

Suggested answers: In the graph below, back edges are labeled *B*, forward edges are labeled *F*, and cross edges are labeled *C*.



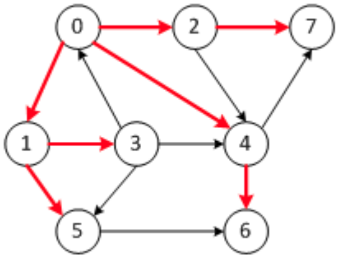
Test Yourself 6.10

Consider the following directed graph (note that this graph is not exactly the same as the one used in Questions 9):



Execute the bread-first search algorithm, which is described in Section 6.1.3.3 (this algorithm is also described in Section 14.3.3 of the textbook), on the above graph and highlight the resulting tree edges with thick lines. Also show vertices at each level. Start at the vertex 0 (so the vertex 0 is at level 0).

Suggested answer:



Level 0: vertex 0; Level 1: vertices 1, 2, 4; Level 2: vertices 3, 5, 6, 7

Test Yourself 6.11

For BFS on a directed graph, all nontree edges are cross edges. True or False?

True.

Your option is wrong.

False.

Your option is right. All nontree edges are either back edges or cross edges.

Test Yourself 6.12

After the execution of *BFS* on an undirected graph, the tree edges form a tree and it is called a *BFS tree*. Let *T* be a *BFS* tree of graph *G*. There is a unique path in *T* from the start node *s* to every other node *v* (because *T* is a tree). Let this path denoted *p*. If there are other paths from *s* to *v* in *G* (these paths are not in *T*), then *p* has the least number of edges among all these paths. True or False?

True.

Your option is right.

False.

Your option is wrong.

Test Yourself 6.13

A simple undirected graph is complete if there is an edge between every pair of distinct vertices in the graph. (a). What does a DFS tree of a complete graph look like? (b). What does a BFS tree of a complete graph look like?

Suggested answer:

Test Yourself 6.14

The following table shows prerequisite relationship among some college courses:

Course	Prerequisite
CS200	none
CS205	CS200
CS220	none
CS300	CS200
CS350	CS205,CS300
CS375	CS220, CS350
CS400	CS205

CS410	CS205, CS220
CS450	CS350

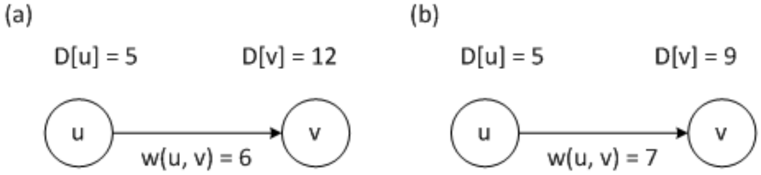
Show one possible sequence of courses a student can take satisfying all prerequisite requirements shown above.

Suggested answer: We can find such a sequence of courses as follows:

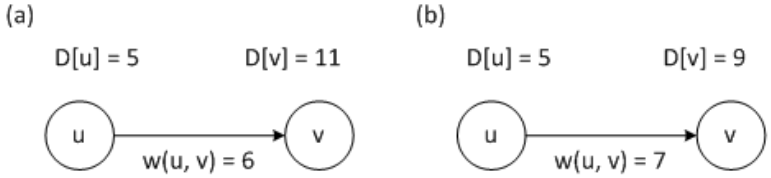
In general, it is possible there are multiple sequences that satisfy the requirements. For the above prerequisite relationships, these are two possible sequences:

Test Yourself 6.15

This question is about the edge relaxation operation described in Section 6.1.5.2 (which is also described in Section 14.6.2 of the textbook). Show the resulting D[v] after applying the edge relaxation operation on each of the following two edges:

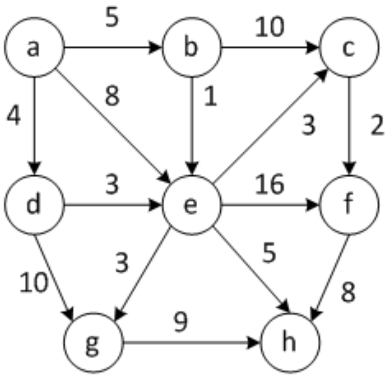


Suggested answer:



Test Yourself 6.16

This question is about Code Segment 6.7 of Dijkstra's shortest path algorithm described in Section 6.1.5.2 (which is also described in Section 14.6.2 of the textbook). Show the first four iterations of the execution of the while loop on the following graph. For each iteration, show the vertex that was dequeued and the updated *D* values of all vertices.



Suggested answer: The *D* values are shown in a table.

Initial *D* values:

After the first iteration: Vertex *a* was dequeued. Updated table is:

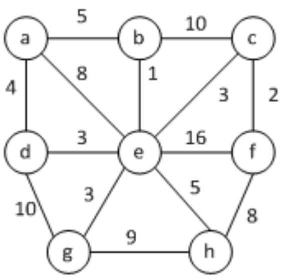
After the second iteration: Vertex *d* was dequeued. Updated table is:

After the third iteration: Vertex *b* was dequeued. Updated table is:

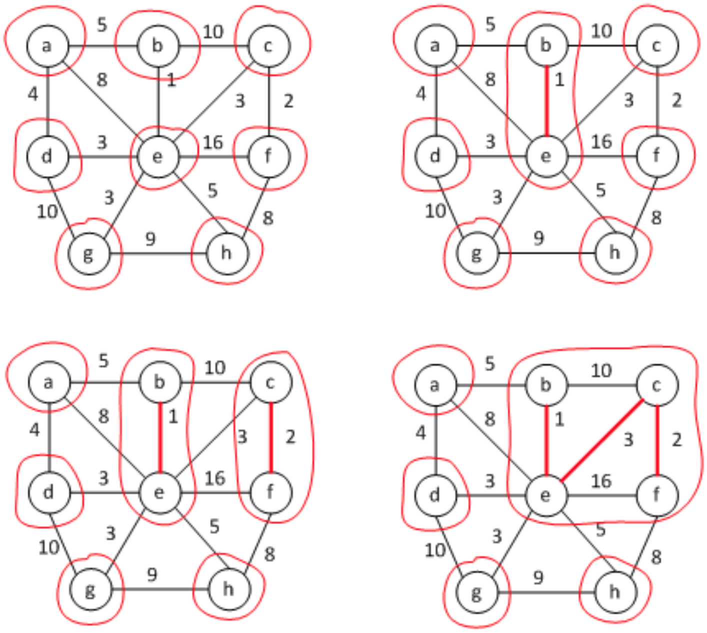
After the fourth iteration: Vertex *e* was dequeued. Updated table is:

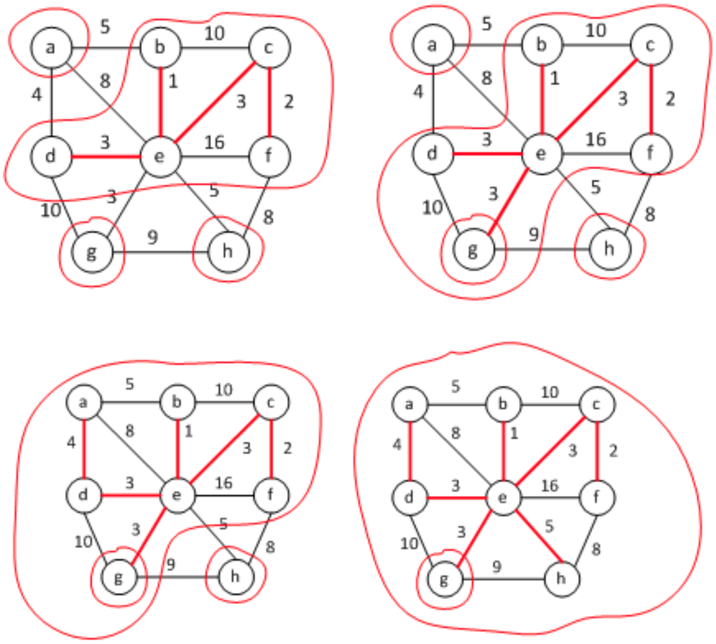
Test Yourself 6.17

Illustrate the execution of the Kruskal's algorithm on the following graph, using Figure 6.23 as a model.



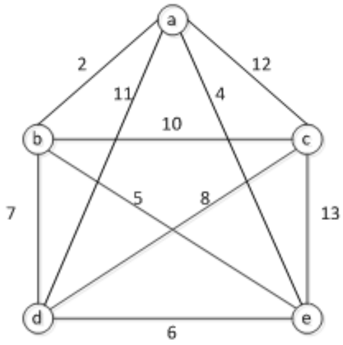
Suggested answer: Edges are processed in the following order:





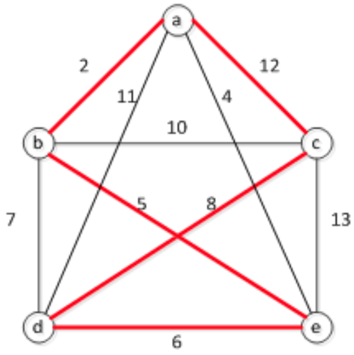
Test Yourself 6.18

18. This question is about TSP approximation algorithms, which are discussed in Section 6.2.2. Consider the following undirected weighted graph.



Execute the nearest-neighbor strategy algorithm on the graph, starting at the vertex *a*. What is the total weight?

Suggested answer:

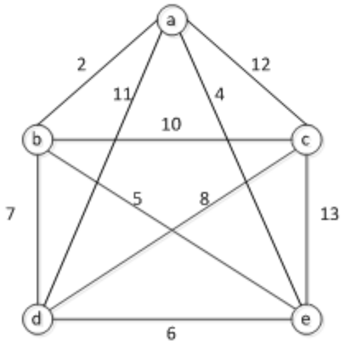


Edges are added in the following order:

The total weight = 2 + 5 + 6 + 8 + 12 = 33.

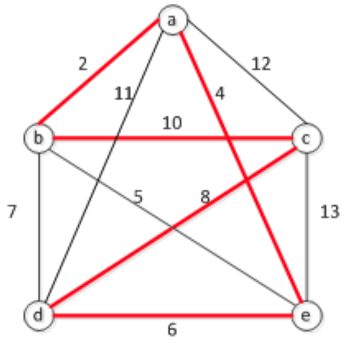
Test Yourself 6.19

This question is about the approximately TSP algorithms, which is discussed in Section 6.2.2. Consider the following undirected weighted graph.



Execute the shortest-link strategy algorithm on the graph. What is the total weight?

Suggested answer:



Edges are added in the following order:

The total weight = 2 + 4 + 6 + 8 + 10 = 30.

Test Yourself 6.20

Answer with True or False for the following questions.

▶ **(1) An algorithm that solves a decision problem should be deterministic. True or False?**

Suggested answer: True

▶ **(2) An algorithm is called efficient or tractable if it solves a decision problem in polynomial time. True or False?**

Suggested answer: True

▶ **(3) The family of languages decidable in polynomial time is denoted P. True or False?**

Suggested answer: True

▶ **(4) The family of languages accepted in nondeterministic polynomial time is denoted NP. True or False?**

Suggested answer: True

▶ **(5) A problem L is called *NP-hard* if, for every $Q \in NP$, Q is reducible to L in polynomial time. True or False?**

Suggested answer: True

▶ **(6) An *NP-hard* problem that is also in NP is called *NP-complete*. True or False?**

Suggested answer: True

▶ **(7) $NP \subseteq P$. True or False?**

Suggested answer: False

References

- Goodrich, M. T., Tamassia, R., & Goldwasser, M. H. (2014). *Data structures and algorithms in Java* (6th ed.). Wiley.
- Gosling, J., Joy, B., Steele, G., Bracha, G., & Buckley, A. (2015). *The Java® language specification, Java SE 8 edition.* Oracle America Inc.
- Oracle. *The Java Tutorials*. Retrived from https://docs.oracle.com/javase/tutorial/index.html.
- Sudkamp, T. A. (1988). *Languages and machines.* Addison-Wesley.
- Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to algorithms* (3rd ed.). MIT Press.

Boston University Metropolitan College