Chapter 16

Graphs

Concepts:
- ▷ Graphs
- ▷ Adjacency Lists
- ▷ Adjacency Matrices
- ▷ Graph Algorithms

$\ldots 314159 \ldots$
—π (digits 176452–176457)

RELATIONS ARE OFTEN AS USEFUL AS DATA. The process of building and accessing a data structure can be thought of as a means of effectively focusing the computation. Linear structures record the history of their accesses, ordered structures perform incremental sorting, and binary trees encode decisions about the partitioning of collections of data.

The most general mechanism for encoding relations between data is the *graph*. Simple structures, like arrays, provide implicit connections, such as adjacency, between stored values. Graphs are more demanding to construct but, as a result, they can encode more detailed information. Indeed, the versatility of graphs allows them to represent many of the most difficult theoretical problems of computer science.

This chapter investigates two traditional implementations of graphs, as well as several standard algorithms for analyzing their structure. We first agree on some basic terminology.

16.1 Terminology

A *graph* G consists of a collection of *vertices* $v \in V_G$ and relations or *edges* $(u, v) \in E_G$ between them (see Figure 16.1). An edge is *incident to* (or *mentions*) each of its two component vertices. A graph is *undirected* if each of its edges is considered a set of two unordered vertices, and *directed* if the mentioned vertices are ordered (e.g., referred to as the *source* and *destination*). A graph S is a *subgraph* of G if and only if $V_S \subseteq V_G$ and $E_S \subseteq E_G$. Simple examples of graphs include the *list* and the *tree*.

In an undirected graph, the number of edges (u, v) incident to a vertex u is its *degree*. In a directed graph, the outgoing edges determine its *out-degree* (or just *degree*) and incoming edges its *in-degree*. A *source* is a vertex with no incoming edges, while a *sink* is a vertex with no outgoing edges.

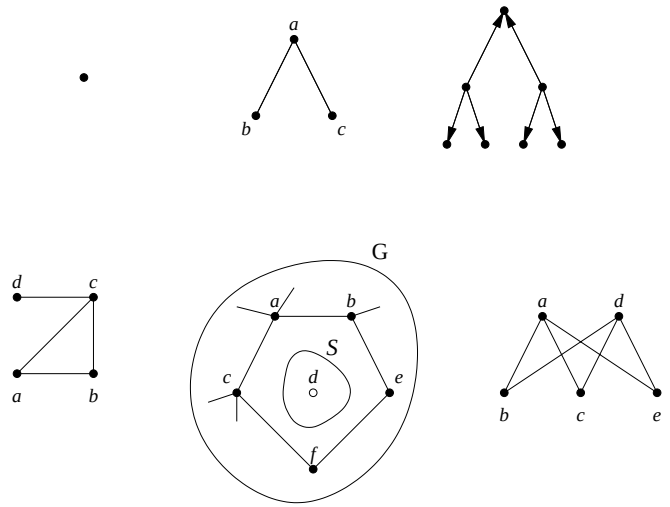


Figure 16.1 Some graphs. Each node a is adjacent to node b, but never to d. Graph G has two components, one of which is S. The directed tree-shaped graph is a directed, acyclic graph. Only the top left graph is complete.

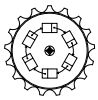
Two edges (u, v) and (v, w) are said to be *adjacent*. A *path* is a sequence of n distinct, adjacent edges $(v_0, v_1), (v_1, v_2), \ldots, (v_{n-1}, v_n)$. In a *simple path* the vertices are distinct, except for, perhaps, the *end points* v_0 and v_n. When $v_0 = v_n$, the simple path is a *cycle*.

Components are always connected.

Two vertices u and v are *connected* (written $u \rightsquigarrow v$) if and only if a simple path of the graph mentions u and v as its end points. A subgraph S is a *connected component* (or, often, just a *component*) if and only if S is a largest subgraph of G such that for every pair of vertices $u, v \in V_S$ either $u \rightsquigarrow v$ or $v \rightsquigarrow u$. A connected component of a directed graph G is *strongly connected* if $u \rightsquigarrow v$ and $v \rightsquigarrow u$ for all pairs of vertices $u, v \in V_S$.

A graph containing no cycles is *acyclic*. A directed, acyclic graph (*DAG*) plays an important role in solving many problems. A *complete graph* G contains an edge (u, v) for all vertices $u, v \in V_G$.

16.2 The Graph Interface



`Graph`

Vertices of a graph are usually labeled with application-specific information. As a result, our implementations of a graph structure depend on the user specifying unique labels for vertices. In addition, edges may be labeled, but not necessarily uniquely. It is common, for example, to specify weights or lengths for edges. All the graph implementations allow addition and removal of vertices and edges:

```
public interface Graph<V,E> extends Structure<V>
{
    public void add(V label);
    // pre: label is a non-null label for vertex
    // post: a vertex with label is added to graph
    //       if vertex with label is already in graph, no action

    public void addEdge(V vtx1, V vtx2, E label);
    // pre: vtx1 and vtx2 are labels of existing vertices
    // post: an edge (possibly directed) is inserted between
    //       vtx1 and vtx2.

    public V remove(V label);
    // pre: label is non-null vertex label
    // post: vertex with "equals" label is removed, if found

    public E removeEdge(V vLabel1, V vLabel2);
    // pre: vLabel1 and vLabel2 are labels of existing vertices
    // post: edge is removed, its label is returned

    public V get(V label);
    // post: returns actual label of indicated vertex

    public Edge<V,E> getEdge(V label1, V label2);
    // post: returns actual edge between vertices

    public boolean contains(V label);
    // post: returns true iff vertex with "equals" label exists

    public boolean containsEdge(V vLabel1, V vLabel2);
    // post: returns true iff edge with "equals" label exists

    public boolean visit(V label);
    // post: sets visited flag on vertex, returns previous value

    public boolean visitEdge(Edge<V,E> e);
    // pre: sets visited flag on edge; returns previous value

    public boolean isVisited(V label);
    // post: returns visited flag on labeled vertex

    public boolean isVisitedEdge(Edge<V,E> e);
    // post: returns visited flag on edge between vertices

    public void reset();
    // post: resets visited flags to false

    public int size();
    // post: returns the number of vertices in graph
```

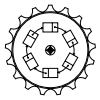
```
    public int degree(V label);
    // pre: label labels an existing vertex
    // post: returns the number of vertices adjacent to vertex

    public int edgeCount();
    // post: returns the number of edges in graph

    public Iterator<V> iterator();
    // post: returns iterator across all vertices of graph

    public Iterator<V> neighbors(V label);
    // pre: label is label of vertex in graph
    // post: returns iterator over vertices adj. to vertex
    //       each edge beginning at label visited exactly once

    public Iterator<Edge<V,E>> edges();
    // post: returns iterator across edges of graph
    //       iterator returns edges; each edge visited once

    public void clear();
    // post: removes all vertices from graph

    public boolean isEmpty();
    // post: returns true if graph contains no vertices

    public boolean isDirected();
    // post: returns true if edges of graph are directed
}
```

Because edges can be fully identified by their constituent vertices, edge operations sometimes require pairs of vertex labels. Since it is useful to implement both directed and undirected graphs, we can determine the type of a specific graph using the `isDirected` method. In undirected graphs, the addition of an edge effectively adds a directed edge in both directions. Many algorithms keep track of their progress by visiting vertices and edges. This is so common that it seems useful to provide direct support for adding (`visit`), checking (`isVisited`), and removing (`reset`) marks on vertices and edges.

Two iterators—generated by `iterator` and `edges`—traverse the vertices and edges of a graph, respectively. A special iterator—generated by `neighbors`—traverses the vertices adjacent to a given vertex. From this information, outbound edges can be determined.

Before we discuss particular implementations of graphs, we consider the abstraction of vertices and edges. From the user's point of view a vertex is a label. Abstractly, an edge is an association of two vertices and an edge label. In addition, we must keep track of objects that have been visited. These features of vertices and edges are independent of the implementation of graphs; thus we commit to an interface for these objects early. Let's consider the `Vertex` class.

```
class Vertex<E>
{
    public Vertex(E label)
    // post: constructs unvisited vertex with label

    public E label()
    // post: returns user label associated w/vertex

    public boolean visit()
    // post: returns, then marks vertex as being visited

    public boolean isVisited()
    // post: returns true iff vertex has been visited

    public void reset()
    // post: marks vertex unvisited

    public boolean equals(Object o)
    // post: returns true iff vertex labels are equal
}
```

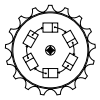


Vertex

This class is similar to an `Association`: the label portion of the `Vertex` cannot be modified, but the visited flag can be freely set and reset. Two `Vertex` objects are considered equal if their labels are equal. It is a bare-bones interface. It should also be noted that the `Vertex` is a nonpublic class. Since a `Vertex` is not visible through the `Graph` interface, there is no reason for the user to have access to the `Vertex` class.

Because the `Edge` class is visible "through" the `Graph` interface (you might ask why—see Problem 16.8), the `Edge` class is declared `public`:

```
public class Edge<V,E>
{
    public Edge(V vtx1, V vtx2, E label,
                boolean directed)
    // post: edge associates vtx1 and vtx2; labeled with label
    //       directed if "directed" set true

    public V here()
    // post: returns first node in edge

    public V there()
    // post: returns second node in edge

    public void setLabel(E label)
    // post: sets label of this edge to label

    public E label()
    // post: returns label associated with this edge

    public boolean visit()
```



Edge

```
        // post: visits edge, returns whether previously visited

        public boolean isVisited()
        // post: returns true iff edge has been visited

        public boolean isDirected()
        // post: returns true iff edge is directed

        public void reset()
        // post: resets edge's visited flag to initial state

        public boolean equals(Object o)
        // post: returns true iff edges connect same vertices
    }
```

As with the `Vertex` class, the `Edge` can be constructed, visited, and reset. Unlike its `Vertex` counterparts, an `Edge`'s label may be changed. The methods `here` and `there` provide access to labels of the vertices mentioned by the edge. These method names are sufficiently ambiguous to be easily used with undirected edges and convey a slight impression of direction for directed edges. Naming of these methods is important because they are used by those who wish to get vertex information while traversing a (potentially directed) graph.

16.3 Implementations

As "traditional" as this science gets, anyway!

Now that we have a good feeling for the graph interface, we consider traditional implementations. Nearly every implementation of a graph has characteristics of one of these two approaches. Our approach to specifying these implementations, however, will be dramatically impacted by the availability of object-oriented features. We first discuss the concept of a partially specified *abstract class* in Java.

16.3.1 Abstract Classes Reemphasized

Normally, when a class is declared, code for each of the methods must be provided. Then, when an instance of the class is constructed, each of the methods can be applied to the resulting object. As is common with our design approach, however, it is useful to partially implement a class and later finish the implementation by *extending* the class in a particular direction. The partial base class is *abstract*; it cannot be constructed because some of the methods are not completely defined. The extension to the class *inherits* the methods that have been defined and specifies any incomplete code to make the class *concrete*.

`GraphMatrix`

Again, we use abstract classes in our design of various graph implementations. Each implementation will be declared abstract, with the `abstract` keyword:

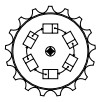
```
abstract public class GraphMatrix<V,E>
    extends AbstractStructure<V> implements Graph<V,E>
```

Our approach will be to provide all the code that can be written without considering whether the graph is undirected or directed. When we must write code that is dependent on the "directedness" of the graph, we delay it by writing just an abstract header for the particular method. For example, we will need to add edges to our graph, but the implementation depends on whether or not the graph is directed. Looking ahead, here is what the declaration for `addEdge` looks like in the abstract class `GraphMatrix`:

```
abstract public void addEdge(V v1, V v2, E label);
// pre: vtx1 and vtx2 are labels of existing vertices
// post: an edge (possibly directed) is inserted between
//       vtx1 and vtx2.
```

That's it! It is simply a *promise* that code will eventually be written.

Once the abstract class is described as fully as possible, we extend it, committing the graph to being undirected or directed. The directed version of the `Graph` implementation, called `GraphMatrixDirected`, specifies the `addEdge` method as follows:



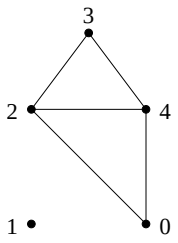
GraphMatrix-Directed

```
public class GraphMatrixDirected<V,E> extends GraphMatrix<V,E>
{
    public void addEdge(V vLabel1, V vLabel2, E label)
    // pre: vLabel1 and vLabel2 are labels of existing vertices
    // post: an edge is inserted between vLabel1 and vLabel2;
    //       if edge is new, it is labeled with label (can be null)
    {
        GraphMatrixVertex<V> vtx1,vtx2;
    }
}
```

Because we declare the class `GraphMatrixDirected` to be an extension of the `GraphMatrix` class, all the code written for the `GraphMatrix` class is inherited; it is as though it had been written for the `GraphMatrixDirected` class. By providing the missing pieces of code (tailored for directed graphs), the extension class becomes concrete. We can actually construct instances of the `GraphMatrixDirected` class.

A related concept, *subtyping*, allows us to use any extension of a class wherever the extended class could be used. We call the class that was extended the *base type* or *superclass*, and the extension the *subtype* or *subclass*. Use of subtyping allows us to write code like

```
GraphMatrix<String,String> g = new GraphMatrixDirected<String,String>();

g.add("Alice");
g.add("Bob");
g.addEdge("Alice","Bob","helps"); // "Alice helps Bob!"
```



(a)

	0	1	2	3	4
0	F	F	T	F	T
1	F	F	F	F	F
2	T	F	F	T	T
3	F	F	T	F	T
4	T	F	T	T	F

(b)

Figure 16.2 (a) An undirected graph and (b) its adjacency matrix representation. Each nontrivial edge is represented twice across the diagonal—once in the gray and once in the white—making the matrix symmetric.

Because `GraphMatrixDirected` is an extension of `GraphMatrix`, it *is* a `GraphMatrix`. Even though we cannot construct a `GraphMatrix`, we can correctly manipulate concrete subtypes using the methods described in the abstract class. In particular, a call to the method `addEdge` calls the method of `GraphMatrixDirected`.

We now return to our normally scheduled implementations!

16.3.2 Adjacency Matrices

Beware: Edges on the diagonal appear exactly once.

An $n \times n$ matrix of booleans is sufficient to represent an arbitrary graph of relations among n vertices. We simply store `true` in the boolean at matrix location $[u][v]$ to represent the fact that there is an edge between u and v (see Figure 16.2), and `false` otherwise. Since entries $[u][v]$ and $[v][u]$ are independent, the representation is sufficient to describe directed graphs as well. Our convention is that the first index (the row) specifies the source and the second index (the column) indicates the destination. To represent undirected graphs, we simply duplicate the entry $[u][v]$ at entry $[v][u]$. This is called an *adjacency matrix* representation of a graph. The abstract graphs of Figures 16.2a and 16.3a are represented, respectively, by the matrices of Figures 16.2b and 16.3b.

One difficult feature of our implementation is the arbitrary labeling of vertices and edges. To facilitate this, we maintain a `Dictionary` that translates a vertex label to a `Vertex` object. To help each vertex keep track of its associated index we extend the `Vertex` class to include methods that manipulate an `index` field. Each index is a small integer that identifies the dedicated row and column that maintain adjacency information about each vertex. To help allocate the indices, we keep a free list (see Section 9.2) of available indices.

One feature of our implementation has the potential to catch the unwary programmer by surprise. Because we keep a `Map` of vertex labels, it is important that the vertex label class implement the `hashCode` function in such a way as to

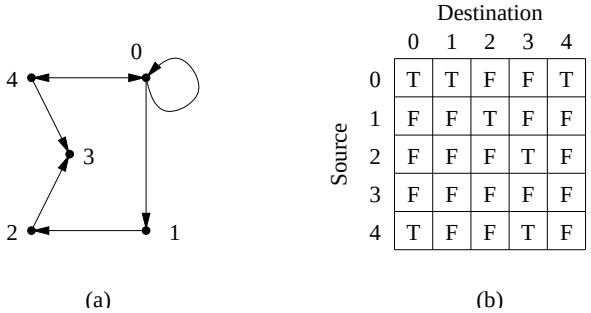
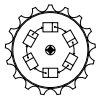


Figure 16.3 (a) A directed graph and (b) its adjacency matrix representation. Each edge appears exactly once in the matrix.

guarantee that if two labels are equal (using the `equals` method), they have the same `hashCode`.

We can now consider the protected data and constructors for the `GraphMatrix` class:


GraphMatrix

```
protected int size;          // allocation size for graph
protected Object data[][];     // matrix - array of arrays
protected Map<V,GraphMatrixVertex<V>> dict;   // labels -> vertices
protected List<Integer> freeList;    // available indices in matrix
protected boolean directed;  // graph is directed

protected GraphMatrix(int size, boolean dir)
{
    this.size = size; // set maximum size
    directed = dir;   // fix direction of edges
    // the following constructs a size x size matrix
    data = new Object[size][size];
    // label to index translation table
    dict = new Hashtable<V,GraphMatrixVertex<V>>(size);
    // put all indices in the free list
    freeList = new SinglyLinkedList<Integer>();
    for (int row = size-1; row >= 0; row--)
        freeList.add(new Integer(row));
}
```

To construct the graph, the user specifies an upper bound on the number of vertices. We allocate `size` arrays of length `size`—a two-dimensional array. By default, the array elements are `null`, so initially there are no edges. We then put each of the indices into the list of available vertex indices.

This constructor is declared protected. It takes a second parameter, `directed`, that identifies whether or not the graph constructed is to act like a

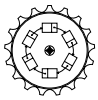
GraphMatrix-Directed

directed graph. When we extend the graph to implement either directed or undirected graphs, we write a public constructor to call the abstract protected class's constructor with an appropriate boolean value:

```
public GraphMatrixDirected(int size)
// pre: size > 0
// post: constructs an empty graph that may be expanded to
//       at most size vertices.  Graph is directed if dir true
//       and undirected otherwise
{
    super(size,true);
}
```

As we discussed before, this technique allows the implementor to selectively inherit the code that is common between directed and undirected graphs. Since we hide the implementation, we are free to reimplement either type of graph without telling our users, perhaps allowing us to optimize our code.

Returning to the `GraphMatrix` class, the `add` method adds a labeled vertex. If the vertex already exists, the operation does nothing. If it is new to the graph, an index is allocated from the free list, a new `Vertex` object is constructed, and the label-vertex association is recorded in the `Map`. The newly added vertex mentions no edges, initially.



GraphMatrix

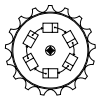
```
public void add(V label)
// pre: label is a non-null label for vertex
// post: a vertex with label is added to graph;
//       if vertex with label is already in graph, no action
{
    // if there already, do nothing
    if (dict.containsKey(label)) return;

    Assert.pre(!freeList.isEmpty(), "Matrix not full");
    // allocate a free row and column
    int row = freeList.removeFirst().intValue();
    // add vertex to dictionary
    dict.put(label, new GraphMatrixVertex<V>(label, row));
}
```

Removing a vertex reverses the `add` process. We must, however, be sure to set each element of the vertex's matrix row and column to `null`, removing any mentioned edges (we may wish to add a new, isolated vertex with this index in the future). When we remove the vertex from the `Map`, we "recycle" its index by adding it to the list of free indices. As with all of our `remove` methods, we return the previous value of the label. (Even though the labels match using `equals`, they may not be precisely the same; once returned the user can extract any unknown information from the previous label before the value is collected as garbage.)

```
public V remove(V label)
```

```
// pre: label is non-null vertex label
// post: vertex with "equals" label is removed, if found
{
    // find and extract vertex
    GraphMatrixVertex<V> vert;
    vert = dict.remove(label);
    if (vert == null) return null;
    // remove vertex from matrix
    int index = vert.index();
    // clear row and column entries
    for (int row=0; row<size; row++) {
        data[row][index] = null;
        data[index][row] = null;
    }
    // add node index to free list
    freeList.add(new Integer(index));
    return vert.label();
}
```

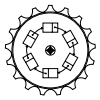
Within the graph we store references to `Edge` objects. Each `Edge` records all of the information necessary to position it within the graph, including whether it is directed or not. This allows the `equals` method to work on undirected edges, even if the vertices were provided in the opposite order (see Problem 16.12). To add an edge to the graph, we require two vertex labels and an edge label. The vertex labels uniquely identify the vertices within the graph, and the edge label is used to form the value inserted within the matrix at the appropriate row and column. To add the edge, we construct a new `Edge` with the appropriate information. This object is written to appropriate matrix entries: undirected graphs update one or two locations; directed graphs update just one. Here is the `addEdge` method for undirected graphs:



GraphMatrix-Undirected

```
public void addEdge(V vLabel1, V vLabel2, E label)
// pre: vLabel1 and vLabel2 are labels of existing vertices, v1 & v2
// post: an edge (undirected) is inserted between v1 and v2;
//       if edge is new, it is labeled with label (can be null)
{
    GraphMatrixVertex<V> vtx1,vtx2;
    // get vertices
    vtx1 = dict.get(vLabel1);
    vtx2 = dict.get(vLabel2);
    // update matrix with new edge
    Edge<V,E> e = new Edge<V,E>(vtx1.label(),vtx2.label(),label,false);
    data[vtx1.index()][vtx2.index()] = e;
    data[vtx2.index()][vtx1.index()] = e;
}
```

Here is a similar method for directed graphs:



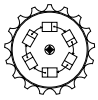
GraphMatrix-Directed

```
public void addEdge(V vLabel1, V vLabel2, E label)
```

```
// pre: vLabel1 and vLabel2 are labels of existing vertices
// post: an edge is inserted between vLabel1 and vLabel2;
//       if edge is new, it is labeled with label (can be null)
{
    GraphMatrixVertex<V> vtx1,vtx2;
    // get vertices
    vtx1 = dict.get(vLabel1);
    vtx2 = dict.get(vLabel2);
    // update matrix with new edge
    Edge<V,E> e = new Edge<V,E>(vtx1.label(),vtx2.label(),label,true);
    data[vtx1.index()][vtx2.index()] = e;
}
```

The differences are quite minor, but the two different subtypes allow us to write specialized code without performing explicit run-time tests.[1]

The `removeEdge` method removes and returns the label associated with the `Edge` found between two vertices. Here is the undirected version (the directed version is similar):

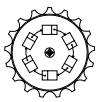


GraphMatrix-Undirected

```
public E removeEdge(V vLabel1, V vLabel2)
// pre: vLabel1 and vLabel2 are labels of existing vertices
// post: edge is removed, its label is returned
{
    // get indices
    int row = dict.get(vLabel1).index();
    int col = dict.get(vLabel2).index();
    // cache old value
    Edge<V,E> e = (Edge<V,E>)data[row][col];
    // update matrix
    data[row][col] = null;
    data[col][row] = null;
    if (e == null) return null;
    else return e.label();
}
```

The `get`, `getEdge`, `contains`, and `containsEdge` methods return information about the graph in an obvious way. Modifying the objects returned by these methods can be dangerous: they have the potential of invalidating the state of the underlying graph implementation.

Each of the `visit`-type methods passes on requests to the underlying object. For example, the `visit` method simply refers the request to the associated `Vertex`:



GraphMatrix

```
public boolean visit(V label)
// post: sets visited flag on vertex, returns previous value
```

[1] This is somewhat misleading, as the obvious run-time tests are replaced by less obvious decreases in performance due to subtyping. Still, the logical complexity of the code can be dramatically reduced using these techniques.

```
{
    Vertex<V> vert = dict.get(label);
    return vert.visit();
}
```

The process of resetting the visitation marks on a graph traverses each of the vertices and edges, resetting them along the way.

But I reiterate myself.

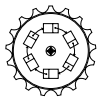
We now consider the implementation of each of the three iterators. The first, generated by `iterator`, traverses the vertices. The values returned by the `Iterator` are vertex labels. This `Iterator` is easily constructed by returning the value of the `Map`'s `keys` function!

```
public Iterator<V> iterator()
// post: returns traversal across all vertices of graph
{
    return dict.keySet().iterator();
}
```

The `neighbors` iterator, which traverses the edges adjacent to a single vertex, considers only the outgoing edges. We simply look up the index associated with the vertex label and scan across the row, building up a list of vertex labels that are adjacent using each of the edges. By putting these values in a list, we can return a `ListIterator` that will give us iterative access to each of the adjacent vertex labels. With this information we may retrieve the respective edges with `getEdge` if necessary.

```
public Iterator<V> neighbors(V label)
// pre: label is label of vertex in graph
// post: returns traversal over vertices adj. to vertex
//       each edge beginning at label visited exactly once
{
    GraphMatrixVertex<V> vert;
    vert = dict.get(label);
    List<V> list = new SinglyLinkedList<V>();
    for (int row=size-1; row>=0; row--)
    {
        Edge<V,E> e = (Edge<V,E>)data[vert.index()][row];
        if (e != null) {
            if (e.here().equals(vert.label()))
                 list.add(e.there());
            else list.add(e.here());
        }
    }
    return list.iterator();
}
```

All that remains is to construct an iterator over the edges of the graph. Again, we construct a list of the edges and return the result of the `iterator` method invoked on the list. For directed edges, we include every edge; for undirected



GraphMatrix-Undirected

edges we include only the edges found in, say, the lower half of the array (including the diagonal). Here is the version for the undirected graph:

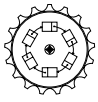
```
public Iterator<Edge<V,E>> edges()
// post: returns traversal across all edges of graph (returns Edges)
{
    List<Edge<V,E>> list = new SinglyLinkedList<Edge<V,E>>();
    for (int row=size-1; row>=0; row--)
        for (int col=size-1; col >= row; col--) {
            Edge<V,E> e = (Edge<V,E>)data[row][col];
            if (e != null) list.add(e);
        }
    return list.iterator();
}
```

The great advantage of the adjacency matrix representation is its simplicity. The access to a particular edge in a graph of size n can be accomplished in constant time. Other operations, like `remove`, appear to be more complex, taking $O(n)$ time. The disadvantage is that the implementation may vastly overestimate the storage required for edges. While we have room for storing $O(n^2)$ directed edges, some graphs may only need to make use of $O(n)$ edges. Graphs with superlinear numbers of edges are called *dense;* all other graphs are *sparse*. When graphs are sparse, most of the elements of the adjacency matrix are not used, leading to a significant waste of space. Our next implementation is particularly suited for representing sparse graphs.

16.3.3 Adjacency Lists

Recalling the many positive features of a linked list over a fixed-size array, we now consider the use of an *adjacency list*. As with the adjacency matrix representation, we maintain a `Map` for identifying the relationship between a vertex label and the associated `Vertex` object. Within the vertex, however, we store a collection (usually a linked list) of edges that mention this vertex. Figures 16.4 and 16.5 demonstrate the adjacency list representations of undirected and directed graphs. The great advantage of using a collection is that it stores only edges that appear as part of the graph.

As with the adjacency matrix implementation, we construct a privately used extension to the `Vertex` class. In this extension we reference a collection of edges that are incident to this vertex. In directed graphs, we collect edges that mention the associated vertex as the source. In undirected graphs any edge incident to the vertex is collected. Because the edges are stored within the vertices, most of the actual implementation of graphs appears within the implementation of the extended vertex class. We see most of the implementation here:



GraphListVertex

```
class GraphListVertex<V,E> extends Vertex<V>
{
    protected Structure<Edge<V,E>> adjacencies; // adjacent edges
    public GraphListVertex(V key)
```

```
// post: constructs a new vertex, not incident to any edge
{
    super(key); // init Vertex fields
    adjacencies = new SinglyLinkedList<Edge<V,E>>(); // new list
}

public void addEdge(Edge<V,E> e)
// pre: e is an edge that mentions this vertex
// post: adds edge to this vertex's adjacency list
{
    if (!containsEdge(e)) adjacencies.add(e);
}

public boolean containsEdge(Edge<V,E> e)
// post: returns true if e appears on adjacency list
{
    return adjacencies.contains(e);
}

public Edge<V,E> removeEdge(Edge<V,E> e)
// post: removes and returns adjacent edge "equal" to e
{
    return adjacencies.remove(e);
}

public Edge<V,E> getEdge(Edge<V,E> e)
// post: returns the edge that "equals" e, or null
{
    Iterator<Edge<V,E>> edges = adjacencies.iterator();
    while (edges.hasNext())
    {
        Edge<V,E> adjE = edges.next();
        if (e.equals(adjE)) return adjE;
    }
    return null;
}

public int degree()
// post: returns the degree of this node
{
    return adjacencies.size();
}

public Iterator<V> adjacentVertices()
// post: returns iterator over adj. vertices
{
    return new GraphListAIterator<V,E>(adjacentEdges(), label());
}

public Iterator<Edge<V,E>> adjacentEdges()
```

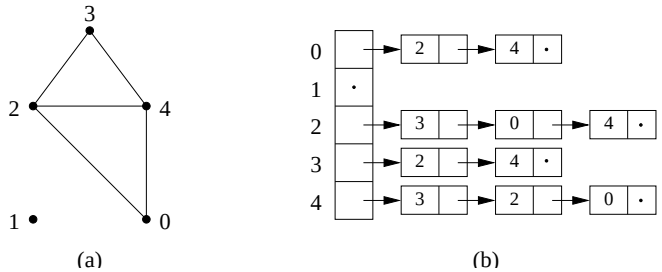


Figure 16.4 (a) An undirected graph and (b) its adjacency list representation. Each edge is represented twice in the structure. (Compare with Figure 16.2.)

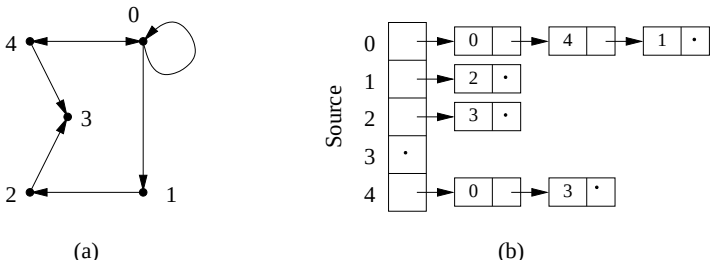


Figure 16.5 (a) A directed graph and (b) its adjacency list representation. Each edge appears once in the source list. (Compare with Figure 16.3.)

```
    // post: returns iterator over adj. edges
    {
        return adjacencies.iterator();
    }
}
```

The constructor initializes its `Vertex` fields, and then constructs an empty adjacency list. Elements of this list will be `Edge` objects. Most of the other methods have obvious behavior.

The only difficult method is `getEdge`. This method returns an edge from the adjacency list that logically equals (i.e., is determined to be equal through a call to `Edge`'s `equals` method) the edge provided. In an undirected graph the order of the vertex labels may not correspond to the order found in edges in the edge list. As a result, `getEdge` returns a *canonical edge* that represents the edge specified as the parameter. This ensures that there are not multiple instances of edges that keep track of shared information.

We are now ready to implement most of the methods required by the `Graph`

interface. First, we consider the protected `GraphList` constructor:

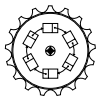
```
protected Map<V,GraphListVertex<V,E>> dict; // label -> vertex
protected boolean directed; // is graph directed?

protected GraphList(boolean dir)
{
    dict = new Hashtable<V,GraphListVertex<V,E>>();
    directed = dir;
}
```

Our approach to extending the abstract `GraphList` type to support directed and undirected graphs is similar to that described in the adjacency matrix implementation. With the list-based implementation, though, we need not provide an upper bound on the number of vertices that will appear in the graph. This is because the underlying structures automatically extend themselves, if necessary.

The process of adding and removing a vertex involves simple manipulations of the `Map`. Here, for example, is the code for adding a new vertex to the graph:

```
public void add(V label)
// pre: label is a non-null label for vertex
// post: a vertex with label is added to graph;
//       if vertex with label is already in graph, no action
{
    if (dict.containsKey(label)) return; // vertex exists
    GraphListVertex<V,E> v = new GraphListVertex<V,E>(label);
    dict.put(label,v);
}
```

To add an edge to the graph we insert a reference to the `Edge` object in the appropriate adjacency lists. For a directed graph, we insert the edge in the list associated with the source vertex. For an undirected graph, a reference to the edge must be inserted into both lists. It is important, of course, that a *reference* to a single edge be inserted in both lists so that changes to the edge are maintained consistently. Here, we show the undirected version:



`GraphList-Undirected`

```
public void addEdge(V vLabel1, V vLabel2, E label)
// pre: vLabel1 and vLabel2 are labels of existing vertices, v1 & v2
// post: an edge (undirected) is inserted between v1 and v2;
//       if edge is new, it is labeled with label (can be null)
{
    GraphListVertex<V,E> v1 = dict.get(vLabel1);
    GraphListVertex<V,E> v2 = dict.get(vLabel2);
    Edge<V,E> e = new Edge<V,E>(v1.label(), v2.label(), label, false);
    v1.addEdge(e);
    v2.addEdge(e);
}
```

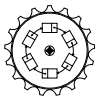
Removing an edge simply reverses this process:

```
public E removeEdge(V vLabel1, V vLabel2)
// pre: vLabel1 and vLabel2 are labels of existing vertices
// post: edge is removed, its label is returned
{
    GraphListVertex<V,E> v1 = dict.get(vLabel1);
    GraphListVertex<V,E> v2 = dict.get(vLabel2);
    Edge<V,E> e = new Edge<V,E>(v1.label(), v2.label(), null, false);
    v2.removeEdge(e);
    e = v1.removeEdge(e);
    if (e == null) return null;
    else return e.label();
}
```

Steve doesn't like this I agree.

Notice that to remove an edge a "pattern" edge must be constructed to identify (through `equals`) the target of the remove.

Now that we can remove edges, we can remove a vertex. Since the removal of a vertex should remove incident edges, it is important that each of the adjacency lists be checked. Our approach is to iterate across each of the vertices and remove any edge that mentions that vertex. This requires some care. Here is the directed version:



`GraphList-Directed`

```
public V remove(V label)
// pre: label is non-null vertex label
// post: vertex with "equals" label is removed, if found
{
    GraphListVertex<V,E> v = dict.get(label);

    Iterator<V> vi = iterator();
    while (vi.hasNext())
    {
        V v2 = vi.next();
        if (!label.equals(v2)) removeEdge(v2,label);
    }
    dict.remove(label);
    return v.label();
}
```

The complexity of this method counterbalances the simplicity of adding a vertex to the graph.

Many of the remaining edge and vertex methods have been greatly simplified by our having extended the `Vertex` class. Here, for example, is the `degree` method:

```
public int degree(V label)
// pre: label labels an existing vertex
// post: returns the number of vertices adjacent to vertex
{
    Assert.condition(dict.containsKey(label), "Vertex exists.");
    return dict.get(label).degree();
}
```

This code calls the GraphListVertex degree method. That, in turn, calls the size method of the underlying collection, a SinglyLinkedList. Most of the remaining methods are simply implemented.

At this point, it is useful to discuss the implementation of iterators for the adjacency list representation. Like the adjacency matrix implementation, the iterator method simply returns the result of the keys iterator on the underlying Map. Each of the values returned by the iterator is a vertex label, which is exactly what we desire.

The neighbors iterator should return an iterator over the neighbors of the provided vertex. Since each vertex maintains a Collection of edges, the iterator method of the collection returns Edge values. Our approach is similar to the approach we used in constructing the iterators for Maps: we construct a private, special-purpose iterator that drives the Collection iterator as a slave. The process of extracting the "other" vertex from each edge encountered is made complex by the fact that "this" vertex can appear as either the source or destination vertex when the graph is undirected.

The Edge's iterator has similar complexities. The easiest approach is to construct a list of edges by traversing each of the edge lists found in each of the vertices. The result is an iterator over the resulting list. Here is the code for the constructor of our private GraphListEIterator class:

```
protected AbstractIterator<Edge<V,E>> edges;

public GraphListEIterator(Map<V,GraphListVertex<V,E>> dict)
// post: constructs a new iterator across edges of
//       vertices within dictionary
{
    List<Edge<V,E>> l = new DoublyLinkedList<Edge<V,E>>();
    Iterator<GraphListVertex<V,E>> dictIterator = dict.values().iterator();
    while (dictIterator.hasNext())
    {
        GraphListVertex<V,E> vtx =
            (GraphListVertex<V,E>)dictIterator.next();
        Iterator<Edge<V,E>> vtxIterator = vtx.adjacentEdges();
        while (vtxIterator.hasNext())
        {
            Edge<V,E> e = vtxIterator.next();
            if (vtx.label().equals(e.here())) l.addLast(e);
        }
    }
    edges = (AbstractIterator<Edge<V,E>>)l.iterator();
}
```

Each of the edges is traversed in the construction of the iterator, so there is considerable overhead just during initialization. Once constructed, however, the traversal is quick. An alternative implementation would distribute the cost over each step of the traversal. Construction of the iterator would be less expensive, but each step of the traversal would be slightly slower. In the end, both methods

consume similar amounts of time. If, however, partial traversals of the edge lists are expected, the alternative implementation has its merits.

With two implementations of graphs in mind, we now focus on a number of examples of their use.

16.4 Examples: Common Graph Algorithms

Because the graph structure is so flexible there are many good examples of graph applications. In this section, we investigate a number of beautiful algorithms involving graphs. These algorithms provide a cursory overview of the problems that may be cast as graph problems, as well as techniques that are commonly used to solve them.

16.4.1 Reachability

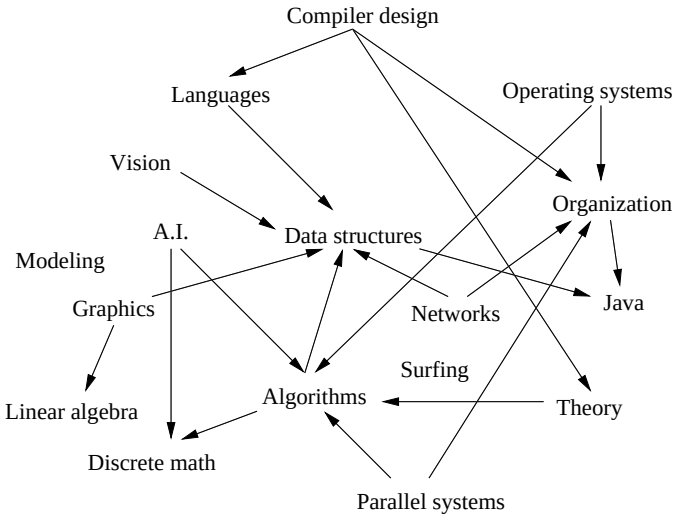
Once data are stored within a graph, it is often desirable to identify vertices that are reachable from a common source (see Figure 16.6). One approach is to treat the graph as you would a maze and, using search techniques, find the reachable vertices. For example, we may use *depth-first search*: each time we visit an unvisited vertex we seek to further deepen the traversal.

The following code demonstrates how we might use recursion to search for unvisited vertices:

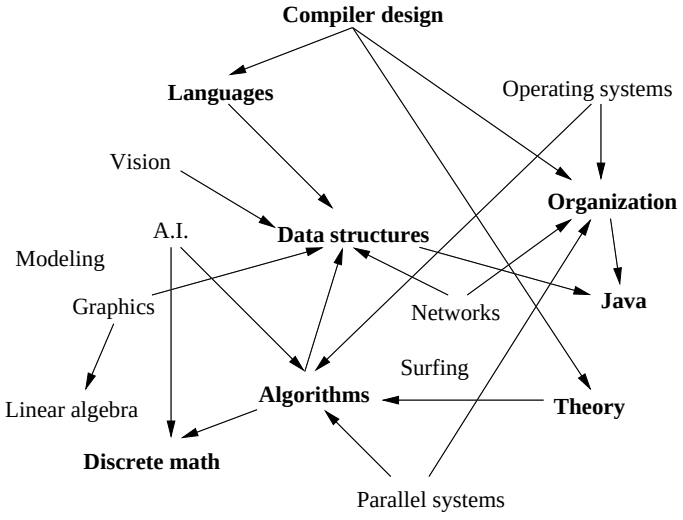
Reachability

```
static void reachableFrom(Graph<V,E> g, V vertexLabel)
// pre: g is a non-null graph, vertexLabel labels a vertex of g
// post: unvisited vertices reachable from vertex are visited
{
    g.visit(vertexLabel);   // visit this vertex

    // recursively visit unvisited neighbor vertices
    Iterator<V> ni = g.neighbors(vertexLabel);
    while (ni.hasNext())
    {
        V neighbor = ni.next(); // adjacent node label
        if (!g.isVisited(neighbor))
        {
            reachableFrom(g,neighbor); // depth-first search
        }
    }
}
```

We clear each `Vertex`'s visited flag with a call to `reset`, and then call `reachableFrom` with the graph and the source vertex for the reachability test. Before the call to `reachableFrom`, the vertex labeled with the `vertexLabel` has not been visited. After the call, every vertex reachable from the vertex has been visited. Some vertices may be left unvisited and are not reachable from the



(a)



(b)

Figure 16.6 Courses you might be expected to have taken if you're in a compiler design class. (a) A typical prerequisite graph (classes point to prerequisites). Note the central nature of data structures! (b) Bold courses can be reached as requisite courses for compiler design.

source. So, to determine whether you may reach one vertex from another, the following code can be used:

```
g.reset();
reachableFrom(g,sourceLabel);
canGetThere = g.isVisited(destinationLabel);
```

In Section 10.3 we discussed the use of a `Linear` structure to maintain the state of a search of a maze. The use of a `Stack` led to a depth-first search. Here, however, no `Stack` appears! The reason is that the act of calling the procedure recursively maintains an implied stack of local variables.

How long does it take to execute the procedure? Suppose that, ultimately, we visit the reachable vertices V_r. Let E_r be the edges of the graph found among the vertices of V_r. Clearly, each vertex of V_r is visited, so there is one call to `reachableFrom` from each vertex $v \in V_r$. For each call, we ask each destination vertex if it has been visited or not. There is one such test for every edge within E_r. Thus, the total time is $O(|V_r| + |E_r|)$. Since $|E_r| \geq |V_r - 1|$ (every new vertex is visited by traversing a new edge), the algorithm is dominated by the number of edges actually investigated. Of course, if the graph is dense, this is bounded above by the square of the number of vertices.

In an undirected graph the reachable vertices form a component of the graph. To count the components of a graph (the undirected version of the graph of Figure 16.6 has three components), we iterate across the vertices of the graph, calling the `reachableFrom` procedure on any vertex that has not yet been visited. Since each unvisited vertex is not reachable from those that have been encountered before, the number of searches determines the number of components.

16.4.2 Topological Sorting

Occasionally it is useful to list the vertices of a graph in such a way as to make the edges point in one direction, for example, toward the front of the list. Such graphs have to be directed and acyclic (see Problem 16.13). A listing of vertices with this property is called a *topological sort*.

One technique for developing a topological sort involves keeping track of a counter or virtual timer. The timer is incremented every time it is read. We now visit each of the nodes using a depth-first search, labeling each node with two *time stamps*. These time stamps determine the span of time that the algorithm spends processing the descendants of a node. When a node is first encountered during the search, we record the *start time*. When the recursive depth-first search returns from processing a node, the timer is again read and the *finish time* is recorded. Figure 16.7 depicts the intervals associated with each vertex of the graph of Figure 16.6. (As arbitrary convention, we assume that a vertex iterator would encounter nodes in the diagram in "reading" order.)

One need only observe that the finish time of a node is greater than the finish time of any node it can reach. (This depth-first search may have to be

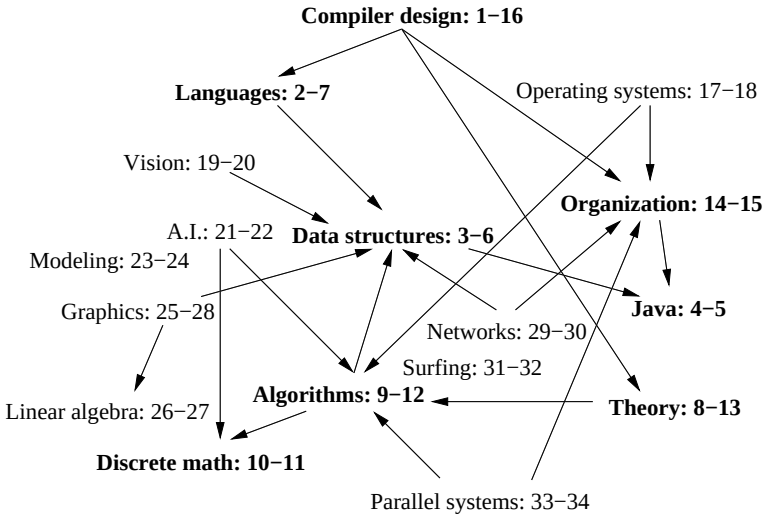


Figure 16.7 The progress of a topological sort of the course graph. The time interval following a node label indicates the time interval spent processing that node or its descendants. Dark nodes reachable from compiler design are all processed during the interval [1–16]—the interval associated with compiler design.

started at several nodes if there are several independent components, or if the graph is not strongly connected.) The algorithm, then, simply lists the vertices in the order in which they are finished. For our course graph we generate one of many course schedules that allow students to take courses without violating course requirements:

Vertices Ordered by Finish Time					
5.	Java	15.	Organization	27.	Linear algebra
6.	Data structures	16.	Compiler design	28.	Graphics
7.	Languages	18.	Operating systems	30.	Networks
11.	Discrete math	20.	Vision	32.	Surfing
12.	Algorithms	22.	A.I.	34.	Parallel systems
13.	Theory	24.	Modeling		

Actually, the time stamps are useful only for purposes of illustration. In fact, we can simply append vertices to the end of a list at the time that they would normally be finished. Here is a sample code:

TopoSort

```
public static List<V> topoSort(Graph<V,E> g)
// pre: g is non-null
// post: returns list of all vertices of g, topologically ordered
{
    // construct result list
    List<V> l = new DoublyLinkedList<V>();
    Iterator<V> vi = g.elements();
    while (vi.hasNext())
    {
        V v = vi.next();
        // perform depth-first search on unvisited vertices
        if (!g.isVisited(v))
        {
            DFS(g,v,l);
        }
    }
    // result is queue of vertex labels
    return l;
}

static protected void DFS(Graph<V,E> g, V n, List<V> l)
// post: performs depth-first search enqueuing
//       unvisited descendants of node n into l
{
    g.visit(n); // mark node visited
    Iterator<V> ei = g.neighbors(n); // get neighbors
    while (ei.hasNext())
    {
        V neighbor = ei.next();
        // potentially deepen search if neighbor not visited
        if (!g.isVisited(neighbor)) {
```

```
            DFS(g,neighbor,l);
        }
    }
    l.addLast(n); // add this value once decendants added
}
```

These functions are declared as static procedures of a program that might make use of a topological sort. Alternatively, they could be written as methods of a graph, reducing the complexity of method calls.

16.4.3 Transitive Closure

Previously we discussed a reachability algorithm that determines if it is possible to reach any particular vertex from a particular source. It is also useful to compute the *transitive closure* of a graph: for *each pair* of vertices $u, v \in V$, *is* v *reachable from* u*?* These questions can be answered by $O(|V|)$ calls to the depth-first search algorithm (leading to an algorithm that is $O(|V|(|V| + |E|))$), or we can look for a more direct algorithm that has similar behavior.

One algorithm, *Warshall's algorithm*, computes reachability for each pair of vertices by modifying the graph. When the algorithm is applied to a graph, edges are added until there is an edge for every pair of connected vertices (u, v). The concept behind Warshall's algorithm is relatively simple. Two connected vertices u and v are either directly connected, or the path from u to v passes through an intermediate node w. The algorithm simply considers each node and connects all pairs of nodes u and v that can be shown to use w as an intermediate node. Here is a Java implementation:

Warshall

```
static void warshall(Graph<V,E> g)
// pre: g is non-null
// post: g contains edge (a,b) if there is a path from a to b
{
    Iterator<V> witer = g.iterator();

    while (witer.hasNext())
    {
        Iterator<V> uiter = g.iterator();
        V w = witer.next();
        while (uiter.hasNext())
        {
            Iterator<V> viter = g.iterator();
            V u = uiter.next();
            while (viter.hasNext())
            {
                V v = viter.next();
                // check for edge from u to v via w
                if (g.containsEdge(u, w) &&
                    g.containsEdge(w, v))
                {
```

```
                        g.addEdge(u, v, null);
                    }
                }
            }
        }
    }
```

This algorithm is clearly $O(|V|^3)$: each iterator visits $|V|$ vertices and (for adjacency matrices) the check for existence of an edge can be performed in constant time.

To see how the algorithm works, we number the vertices in the order they are encountered by the vertex iterator. After k iterations of the outer loop, all "reachability edges" of the subgraph containing just the first k vertices are completely determined. The next iteration extends this result to a subgraph of $k+1$ vertices. An inductive approach to proving this algorithm correct (which we avoid) certainly has merit.

16.4.4 All Pairs Minimum Distance

A slight modification of Warshall's algorithm gives us a method for computing the minimum distance between all pairs of points. The method is due to Floyd. Again, we use three loops to compute the new edges representing reachability, but these edges are now labeled, or *weighted*, with integer distances that indicate the current minimum distance between each pair of nodes. As we consider intermediate nodes, we merge minimum distance approximations by computing and updating the distance if the sum of path lengths through an intermediate node w is less than our previous approximation. Object orientation makes this code somewhat cumbersome:

Floyd

```
static void floyd(Graph<V,E> g)
// post: g contains edge (a,b) if there is a path from a to b
{
    Iterator<V> witer = g.iterator();

    while (witer.hasNext())
    {
        Iterator<V> uiter = g.iterator();
        V w = witer.next();
        while (uiter.hasNext())
        {
            Iterator<V> viter = g.iterator();
            V u = uiter.next();
            while (viter.hasNext())
            {
                V v = viter.next();
                if (g.containsEdge(u,w) && g.containsEdge(w,v))
                {
                    Edge<V,E> leg1 = g.getEdge(u,w);
```

```
                    Edge<V,E> leg2 = g.getEdge(w,v);
                    int leg1Dist = leg1.label();
                    int leg2Dist = leg2.label();
                    int newDist = leg1Dist+leg2Dist;

                    if (g.containsEdge(u,v))
                    {
                        Edge<V,E> across = g.getEdge(u,v);
                        int acrossDist = across.label();
                        if (newDist < acrossDist)
                            across.setLabel(newDist);
                    } else {
                        g.addEdge(u,v,newDist);
                    }
                }
            }
        }
    }
}
```

Clearly, edge labels could contain more information than just the path length. For example, the path itself could be constructed, stored, and produced on request, if necessary. Again, the complexity of the algorithm is $O(|V|^3)$. This is satisfactory for dense graphs, especially if they're stored in adjacency matrices, but for sparse graphs the checking of all possible edges seems excessive. Indeed, other approaches can improve these bounds. We leave some of these for your next course in algorithms!

16.4.5 Greedy Algorithms

We now consider two examples of *greedy* algorithms—algorithms that compute optimal solutions to problems by acting in the optimal or "most greedy" manner at each stage in the algorithm. Because both algorithms seek to find the best choice for the next step in the solution process, both make use of a priority queue.

Minimum Spanning Tree

The solution to many network problems involves identifying a *minimum spanning tree* of a graph. A minimum spanning tree of an edge-weighted graph is a tree that connects every vertex of a component whose edges have minimum total edge weight. Such a tree might represent the most inexpensive way to connect several cities with telephone trunk lines. For this reason, we will interpret the weights as edge lengths. For the purposes of our discussion, we will assume that the graph under consideration is composed of a single component. (If the graph contains multiple components, we can compute a minimum spanning forest with multiple applications of the minimum spanning tree algorithm.)

MCST

It is useful to note that if the vertices of a connected graph are partitioned into any two sets, any minimum spanning tree contains a shortest edge that connects nodes between the two sets. We will make use of this fact by segregating visited nodes from unvisited nodes. The tree (which spans the visited nodes and does so minimally) is grown by iteratively incorporating a shortest edge that incorporates an unvisited node to the tree. The process stops when $|V| - 1$ edges have been added to the tree.

```
static public void mcst(Graph<String,Integer> g)
// pre: g is a graph
// post: edges of minimum spanning tree of a component are visited
{
    // keep edges ranked by length
    PriorityQueue<ComparableEdge<String,Integer>> q =
        new SkewHeap<ComparableEdge<String,Integer>>();
    String v = null;        // current vertex
    Edge<String,Integer> e; // current edge
    boolean searching;      // looking for a nearby vertex
    g.reset();              // clear visited flags
    // select a node from the graph, if any
    Iterator<String> vi = g.iterator();
    if (!vi.hasNext()) return;
    v = vi.next();
    do
    {
        // visit the vertex and add all outgoing edges
        g.visit(v);
        Iterator<String> ai = g.neighbors(v);
        while (ai.hasNext()) {
            // turn it into outgoing edge
            e = g.getEdge(v,ai.next());
            // add the edge to the queue
            q.add(new ComparableEdge<String,Integer>(e));
        }
        searching = true;
        while (searching && !q.isEmpty())
        {
            // grab next shortest edge on tree fringe
            e = q.remove();
            // does this edge take us somewhere new?
            v = e.there();
            if (g.isVisited(v)) v = e.here();
            if (!g.isVisited(v)) {
                searching = false;
                g.visitEdge(g.getEdge(e.here(),e.there()));
            }
        }
    } while (!searching);
}
```

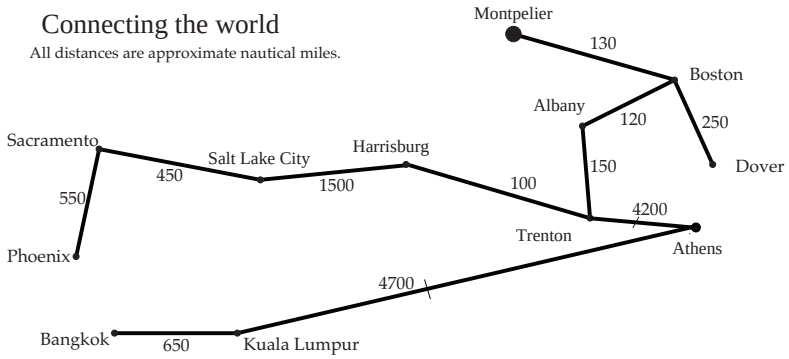


Figure 16.8 The progress of a minimum spanning tree computation. Bold vertices and edges are part of the tree. Harrisburg and Trenton, made adjacent by the graph's shortest edge, are visited first. At each stage, a shortest external edge adjacent to the tree is incorporated.

First, we use a priority queue to rank edges based on length. As we remove the edges from the queue, the smallest edges are considered first (see Figure 16.8). When an edge is considered that includes an unvisited vertex, we visit it, logically adding it to the minimum spanning tree. We then add any edges that are outward-bound from the newly visited node. At any time, the priority queue contains only edges that mention at least one node of the tree. If, of course, an edge is considered that mentions two previously visited nodes, the edge is superfluous, as the nodes are already connected by a path in the tree (albeit a potentially long one). When the priority queue "runs dry," the tree is fully computed. The result of the algorithm will be visited marks on all nodes and edges that participate in the tree. (If the graph has multiple components, some vertices will not have been visited.)

Our implementation begins by finding a source vertex (`v`) to "prime" the greedy algorithm. The main loop of the algorithm then runs until no new vertices can be added to the tree. Each new vertex is marked as visited and its outbound edges[2] are then added to the priority queue (`q`) of those to be considered. Short edges are removed from the queue until an unvisited vertex is mentioned by a new tree edge (`e`), or the queue is emptied.

Over the course of the algorithm, consideration of each edge and vertex results in a priority queue operation. The running time, then is $O((|V| +$

[2] We use `ComparableEdges` here, an extension to an edge that assumes that the labels implement `Comparable`.

$|E|) \log (|V|))$.

Notice that the first edge added may not be the graph's shortest.

Single-Source Shortest Paths

The minimum spanning tree algorithm is related to a fast, single-source, shortest-path algorithm attributed to Dijkstra. In this algorithm, we desire the minimum-length paths from a single source to all other nodes. We expect the algorithm, of course, to run considerably faster than the all-pairs version. This algorithm also runs in time proportional to $O((|V| + |E|) \log (|V|))$ due to the fact that it uses much the same control as the minimum spanning tree. Here is the code:

Dijkstra

```
public static
    Map<String,ComparableAssociation<Integer,Edge<String,Integer>>>
    dijkstra(Graph<String,Integer> g, String start)
// pre: g is a graph; start is source vertex
// post: returns a dictionary of vertex-based results
//       value is association (total-distance,prior-edge)
{
    // keep a priority queue of distances from source
    PriorityQueue<ComparableAssociation<Integer,Edge<String,Integer>>>
        q = new SkewHeap<ComparableAssociation<Integer,
                                               Edge<String,Integer>>>();
    // results, sorted by vertex
    Map<String,ComparableAssociation<Integer,Edge<String,Integer>>>
        result = new Table<String,
                           ComparableAssociation<Integer,
                                                 Edge<String,Integer>>>();
    String v = start;       // last vertex added
    // result is a (total-distance,previous-edge) pair
    ComparableAssociation<Integer,Edge<String,Integer>> possible =
        new ComparableAssociation<Integer,Edge<String,Integer>>(0,null);
    // as long as we add a new vertex...
    while (v != null)
    {
        if (!result.containsKey(v))
        {
            // visit node v - record incoming edge
            result.put(v,possible);
            // vDist is shortest distance to v
            int vDist = possible.getKey();

            // compute and consider distance to each neighbor
            Iterator<String> ai = g.neighbors(v);
            while (ai.hasNext())
            {
                // get edge to neighbor
                Edge<String,Integer> e = g.getEdge(v,ai.next());
                // construct (distance,edge) pair for possible result
```

```
                possible = new ComparableAssociation<Integer,
                            Edge<String,Integer>>(vDist+e.label(), e);
                q.add(possible);    // add to priority queue
            }
        }
        // now, get closest (possibly unvisited) vertex
        if (!q.isEmpty())
        {
            possible = q.remove();
            // get destination vertex (take care w/undirected graphs)
            v = possible.getValue().there();
            if (result.containsKey(v))
                v = possible.getValue().here();
        } else {
            // no new vertex (algorithm stops)
            v = null;
        }
    }
    return result;
}
```

Unlike the minimum cost spanning tree algorithm, we return a `Table` of results. Each entry in the `Table` has a vertex label as a key. The value is an association between the total distance from the source to the vertex, and (in the nontrivial case) a reference to the last edge that supports the minimum-length path.

We initially record trivial results for the source vertex (setting its distance to zero) and place every outgoing edge in the priority queue (see Figure 16.9). Unlike the minimum spanning tree algorithm, we rank the edges based on *total* distance from the source. These edges describe how to extend, in a nearest-first, greedy manner, the paths that pass from the source through visited nodes. If, of course, an edge is dequeued that takes us to a vertex with previously recorded results, it may be ignored: some other path from the source to the vertex is shorter. If the vertex has not been visited, it is placed in the `Table` with the distance from the source (as associated with the removed edge). New outbound edges are then enqueued.

The tricky part is to rank the edges by the distance of the destination vertex from the source. We can think of the algorithm as considering edges that fall within a neighborhood of increasing radius from the source vertex. When the boundary of the neighborhood includes a new vertex, its minimum distance from the source has been determined.

Since every vertex is considered once and each edge possibly twice, the worst-case performance is $O(|V| + |E|)$, an improvement over the $O(|V|^3)$ performance for sparse graphs.

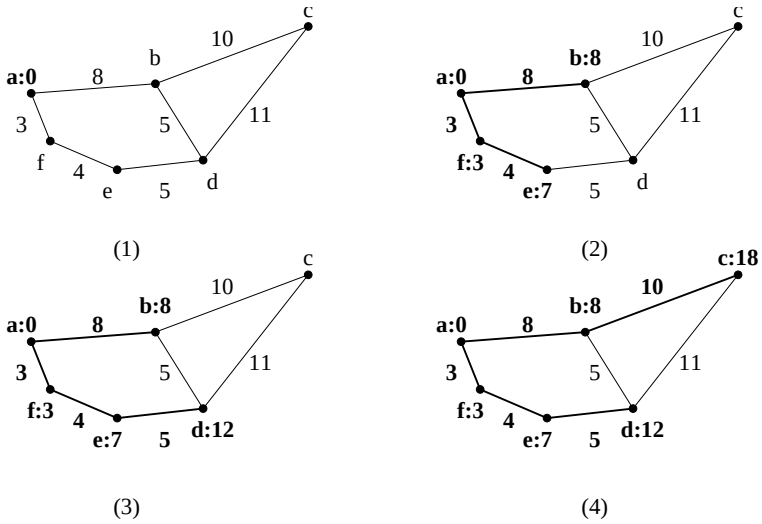


Figure 16.9 The progress of a single-source, shortest-path computation from source **a**. As nodes are incorporated, a minimum distance is associated with the vertex. Compare with Figure 16.8.

16.5 Conclusions

In this chapter we have investigated two traditional implementations of graphs. The adjacency matrix stores information about each edge in a square matrix while the adjacency list implementation keeps track of edges that leave each vertex. The matrix implementation is ideal for dense graphs, where the number of actual edges is high, while the list implementation is best for representing sparse graphs.

Our approach to implementing graph structures is to use partial implementations, called abstract classes, and extend them until they are concrete, or complete. Other methods are commonly used, but this has the merit that common code can be shared among similar classes. Indeed, this inheritance is one of the features commonly found in object-oriented languages.

This last section is, in effect, a stepping stone to an investigation of algorithms. There are many approaches to answering graph-related questions, and because of the dramatic differences in complexities in different implementations, the solutions are often affected by the underlying graph structure.

Finally, we note that many of the seemingly simple graph-related problems cannot be efficiently solved with *any* reasonable representation of graphs. Those problems are, themselves, a suitable topic for many future courses of study.

Self Check Problems

Solutions to these problems begin on page 450.

16.1 What is the difference between a graph and a tree?

16.2 What is the difference between an undirected graph and a directed graph?

16.3 Under what conditions would you use an adjacency matrix over an adjacency list implementation of a graph?

16.4 What do we know if the adjacency matrix is symmetric?

16.5 What is the time potentially required to add an edge to a graph represented as an adjacency list? What if the graph is represented using an adjacency matrix?

16.6 What is a spanning tree of a graph?

16.7 What is a minimum spanning tree of a weighted graph?

16.8 What is the transitive closure of a graph?

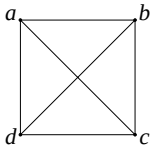
16.9 What is the topological ordering of vertices of a graph?

16.10 Under what conditions is a topological sort of the vertices of a graph possible?

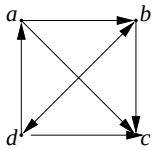
Problems

Solutions to the odd-numbered problems begin on page 486.

16.1 Draw the adjacency matrix and list representations of the following (undirected and complete) graph:



16.2 Draw the adjacency matrix and list representations of the following (directed) graph:



16.3 Draw the adjacency matrix and list representations of a complete tree with seven nodes and undirected edges.

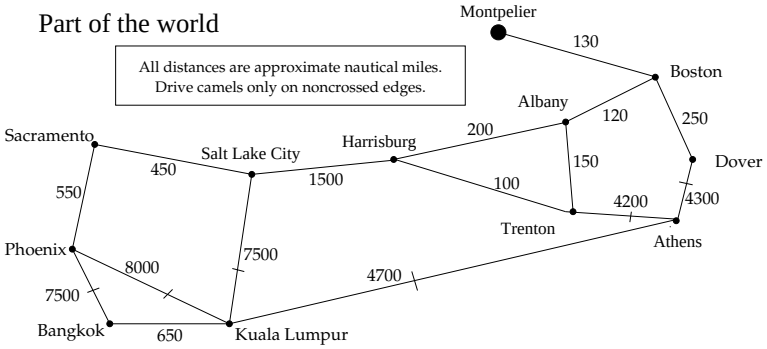
16.4 What are the transitive closures of each of the following graphs?

(a) (b)

16.5 Suppose that we use an $n \times n$ boolean matrix to represent the edges of a directed graph. Assume, as well, that the diagonal elements are all `true`. How should we interpret the nth power of this adjacency matrix?

16.6 What topological characteristics distinguish a general graph from a general tree?

16.7 Consider the following (simplified) map of the world:



a. Compute the shortest air distance to each of the cities from scenic Montpelier, Vermont. Redraw the map with distances to cities and include only the air routes that support the most direct travel to Vermont.

b. Suppose you're interested in setting up a network among the capitals. Redraw the map to depict the minimum spanning network.

c. Suppose you're interested in setting up a Camel Express system. Redraw the map to depict the minimum spanning road systems that don't cross bodies of water (indicated by crossed edges).

16.8 Explain why it is necessary that the `Edge` class "show through" the `Graph` interface. (Hint: Consider implementations of the `Iterator` constructed by the `edges` method.)

16.9 Compare and contrast the performance of the adjacency list and adjacency matrix implementations of graphs.

16.10 For both implementations of graphs, write a method, `isSink`, that returns `true` if and only if the vertex indicated is a sink. (A sink has out-degree 0.)

16.11 For both implementations of graphs, write a method, `isSource`, that returns `true` if and only if the vertex indicated is a source. (A source has in-degree 0.)

16.12 In an undirected graph, it is possible for a single edge to be represented by `Edge` objects whose vertices appear in opposite orders. Describe how a general `equals` method for `Edges` might be written.

16.13 Explain why graphs with topologically sortable vertices must be (1) directed and (2) acyclic.

16.14 Suppose we have a cycle-free graph that describes the dependencies between Java modules. How would you compute the order of compilations that had to occur?

Topological sorting solves this, given no cycles.

16.15 It is a fairly common practice to traverse the vertices and edges of a graph. Consider a new implementation of graphs that keeps a `Map` of vertices as well as an unordered `List` of edges. This makes traversal of edges simple. What is the complexity of each of the other `Graph` operations?

16.16 Extend the all-pairs minimum distance algorithm to keep track of the shortest *path* between the nodes.

16.17 Explain why it is sometimes more efficient to compute the distance from a single source to all other nodes, even though a particular query may be answered with a partial solution.

16.18 Under what conditions can a graph component have nonunique minimum spanning trees?

16.19 Prove that a minimum spanning tree of a graph component must include a shortest edge.

16.20 It is possible, in Dijkstra's algorithm, that an edge removed from the priority queue is not useful: it takes us to a previously visited node. Some of these extraneous edges can be avoided by not placing an edge in the priority queue if the destination has already been visited. Is it still possible to encounter an edge to a previously visited node?

16.6 Laboratory: Converting Between Units

Objective. To perform the transitive closure of a graph.

Discussion. An interesting utility available on UNIX systems is the `units` program. With this program you can convert between one unit and another. For example, if you were converting between feet and yards, you might have the following interaction with the `units` program:

```
You have: yard
You want: inch
Multiply by 36.0
```

The program performs its calculations based on a database of values which has entries that appear as follows:

```
1 yard 3 foot
1 foot 12 inch
1 meter 39.37 inch
```

Notice that there is no direct conversion between yards and inches.

In this lab you are to write a program that computes the relations between units. When the program starts, it reads in a database of values that describe the ratios between units. Each unit becomes a node of a graph, and the conversions are directed edges between related units. Note that the edges of the graph must be directed because the factor that converts inches to yards is the reciprocal of the factor that converts yards to inches.

In order to deduce conversions between distantly related units, it will be necessary for you to construct the closure of the graph. The labels associated with adjacent edges are multiplied together to label the direct edge.

Once your program reads the unit conversion database, the program should prompt for the source units and the destination units. It should then print out the conversion factor used to multiply the source units by to get the destination units. If the units do not appear to be related, you should print out a message that indicates that fact. Prompting continues until the user enters a blank line for the source units.

Thought Questions. Consider the following questions as you complete the lab:

1. There are two approaches to this problem: (1) construct the closure of the graph, or (2) perform a search from the source unit node to the destination unit node. What are the trade-offs to the two approaches?

2. What does it mean if the graph is composed of several disconnected components?

Notes: