23 Graphs

23.1 Introduction

One of the most important modeling tools in computing is the *graph*, which is informally understood as a collection of points connected by lines. Graphs are used to model networks, processes, relationships between entities, and so on—almost every picture we draw in computer science is a graph of one sort or another. In this chapter we present the graph abstract data type and consider two data structures for representing graphs. In the next chapter we study a few graph algorithms.

23.2 Directed and Undirected Graphs

We defined a graph in the course of discussing trees in Chapter 16.

> **Graph**: A collection of *vertices* (or *nodes*) and *edges* connecting the nodes. An edge may be thought of as a pair of vertices. Formally, a graph is an ordered pair <*V*,*E*> where *V* is a set of vertices and *E* is a set of pairs of elements of *V*.
>
> **Undirected graph**: A graph in which the edges are sets of two vertices. In this case the edges have no direction and are represented by line segments in pictures.
>
> **Directed graph** or **digraph**: A graph in which the edges are ordered pairs of vertices. In this case the edges have direction and are represented by arrows.

To illustrate these definitions, consider the images of graphs in Figure 1.

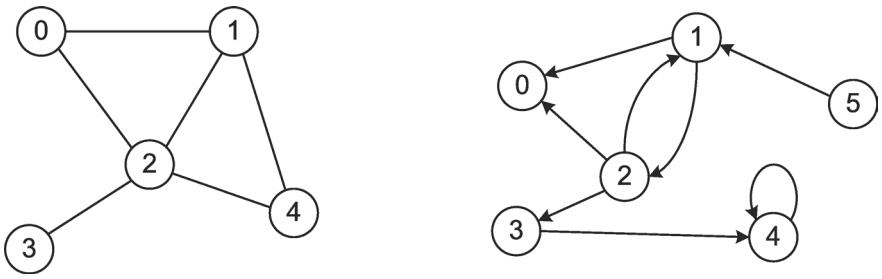

Figure 1: Two Graphs

In these images the vertices are identified by circled numbers. In general we may use any symbol to identify vertices, but as a rule we will use an initial set of natural numbers (that is, any set {0, 1, 2, … *n*}, where $n \geq 0$). The graph on the left is an undirected graph so its edges have no arrows. In its set representation, this graph is

<{0, 1, 2, 3, 4}, {{0,1}, {0,2}, {1,2}, {1,4}, {2,4}, {2,3}}>.

The graph on the right is a directed graph, so it has arrows on its edges. Note that this allows edges from a node to itself (such as the edge from 4 to itself), and two distinct edges between a pair of vertices (such as the two edges connecting vertices 1 and 2); neither of these can occur in an undirected graph. The set representation of the right-hand graph is

$$<\{0, 1, 2, 3, 4\}, \{<1,0>, <1,2>, <2,0>, <2,1>, <2,3>, <3,4>, <4,4>, <5,1>\}>.$$

Note that the edge set in the left-hand graph is a set of sets while the edge set in the right-hand graph is a set of ordered pairs.

Although a graph is really an ordered pair of sets, representing graphs this way is awkward and hard to read, as the examples above illustrate. Consequently we will almost always represent graphs as pictures.

Both undirected and directed graphs are very important and widely applicable in computer science, but we will focus for the remainder of our discussion on undirected graphs. From now on we will refer to undirected graphs as simply *graphs*.

23.3 Basic Terminology

There are several additional terms that must be learned to talk about graphs.

Adjacency: Vertices v_1 and v_2 in a graph $G=<V, E>$ such that $\{v_1, v_2\} \in E$.

In Figure 2 below, vertices 0 and 1 and vertices 7 and 4 are adjacent, but vertices 0 and 4 and vertices 1 and 3 are not adjacent.

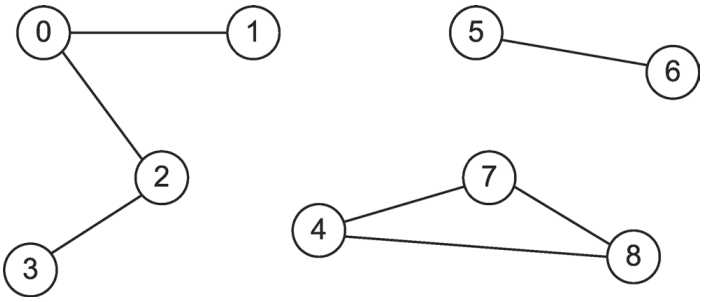

Figure 2: A Graph

Path: A sequence of vertices $p = <v_1, v_2, \ldots, v_n>$ in a graph where $n \geq 2$ and every pair of vertices v_i and v_{i+1} in p are adjacent.

Path length: The number of edges in a path.

In Figure 2, the paths <0, 2, 3> and <4, 7, 8> both have length two.

Cycle: A path $<v_1, v_2, \ldots, v_n>$ in which $v_1 = v_n$.

Simple cycle: A cycle with no repeated edges or vertices (except the first and the last).

In Figure 2, the path <4, 7, 8, 4> is a simple cycle, while the path <4, 7, 8, 7, 4> is a cycle, but not a simple cycle.

Sub-graph: A graph $H=<W, F>$ is a **sub-graph** of graph $G=<V, E>$ if $W \subseteq V$ and $F \subseteq E$.

Connected vertices: Two vertices with a path between them.

Connected graph: A graph with a path between every pair of vertices. A graph that is **not connected** consists of a set of **connected components** that are sub-graphs of the graph.

Figure 2 shows a single graph with three connected components. Each of these components is a sub-graph of the whole graph.

Acyclic graph: A graph with no cycles.

The graph in Figure 2 is not acyclic, but the sub-graph consisting of the vertices 0, 1, 2, and 3 and the edges that connect them, and the sub-graph consisting of vertices 5 and 6 and the edge that connects them, are acyclic graphs.

Tree: An acyclic connected graph.

Forest: A set of trees with no vertices in common.

Spanning tree: Any sub-graph of a connected graph *G* that is a tree and contains every vertex of *G*.

Figure 3 shows a graph with a spanning tree outlined in grey.

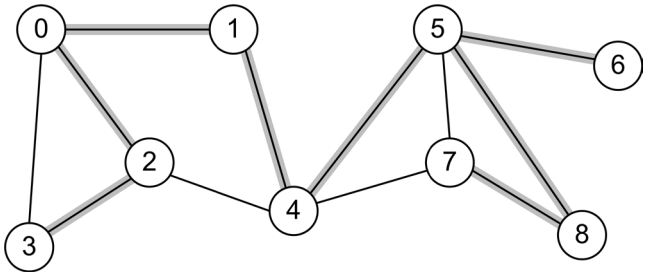

Figure 3: A Graph and A Spanning Tree

23.4 The Graph ADT

A graph is a mathematical entity consisting of an ordered pair of sets. Vertices can be anything; for example, they could be numbers. Hence we can specify the carrier set of the graph ADT as the set of all sets that meet the definition of a graph stated above, with initial sets of natural numbers acting as vertices. The method set of the graph ADT consists of a few basic operations for constructing and querying graphs. We could include operations for adding and deleting vertices and deleting edges, as well as several other query operations, but they are not necessary for the applications we will consider.

newGraph(*n*)—Return a graph with *n* vertices and no edges. The precondition of this operation is that $n \geq 0$.

edges(*g*)—Return the number of edges in graph *g*.

vertices(*g*)—Return the number of vertices in graph *g*.

addEdge(*g*,*v*,*w*)—Return a graph just like *g* except it has an edge connecting *v* and *w*. The precondition is that *v* and *w* are distinct vertices in *g*.

edge?(*g*,*v*,*w*)—Return true if and only if there is an edge between vertices *v* and *w* in *g*.

23.5 The Graph Class

The Graph class is an interface for the graph ADT. It also implements counters for the number of vertices and edges and query functions on these attributes because these are common to all implementations of graphs. A graph is not a collection so the Graph class is not a sub-class of any other. For the same reason, it does include the Enumerable interface. However, it is very convenient to be able to iterate over the vertices adjacent to a given vertex so the Graph class does include a special edge iterator. This class appears in Figure 4.

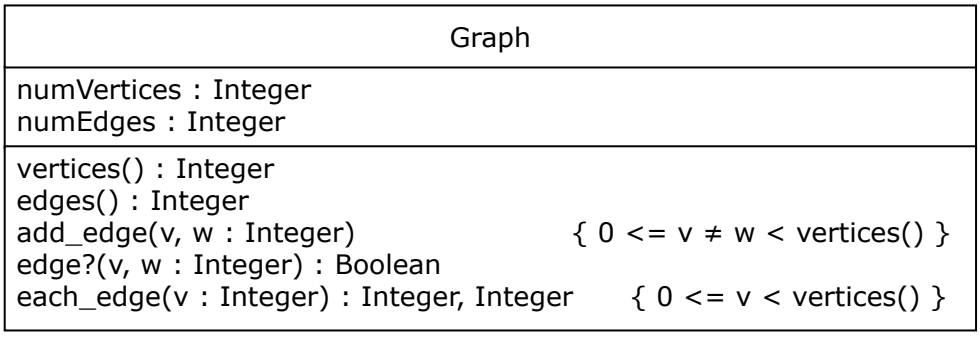

Figure 4: The Graph Interface

The operation each_edge() is an internal iterator that yields each of the edges connected to v as the pair of vertices v, w, where w is an adjacent vertex. There is no need for an iterator over the vertices in the graph because we know that they are (represented by) the integers between 0 and vertices()-1.

23.6 Contiguous Implementation of the Graph ADT

A contiguous implementation of the graph ADT represents a graph using an array. An initial set of natural numbers already represents vertices, so the only thing left to represent is the set of edges. An **adjacency matrix** m is an $n \times n$ Boolean matrix that represents a graph with n vertices by storing true at location $m[v,w]$ if and only if there is an edge between v and w. Notice that this means that every edge is represented twice in the matrix. This approach is realized in an ArrayGraph class.

This scheme is very simple and very fast: adding an edge to a graph or detecting whether an edge exists between two vertices are both $O(1)$ operations, and iterating over the vertices adjacent to a vertex v takes time proportional to the number of vertices adjacent to v. Unfortunately, this speed comes at great cost because the matrix requires n^2 storage locations. Most graphs are *sparse*, meaning they have far fewer than n^2 edges, so often most of this storage space is wasted. Even if a list is *dense* (the opposite of sparse), then space can be saved by storing the edges that are not in the graph, so in either case, the adjacency matrix representation does make efficient use of space.

23.7 Linked Implementation of the Graph ADT

A linked implementation of the graph ADT represents graphs by using space only for the edges in the graph. An **adjacency list** is a linked list of vertices adjacent to a given vertex. An array of adjacency lists holds all the edges in a graph. The diagram in Figure 5 shows the adjacency lists representation of the graph in Figure 2. Note that the array holds list headers and the adjacency lists are singly-linked.

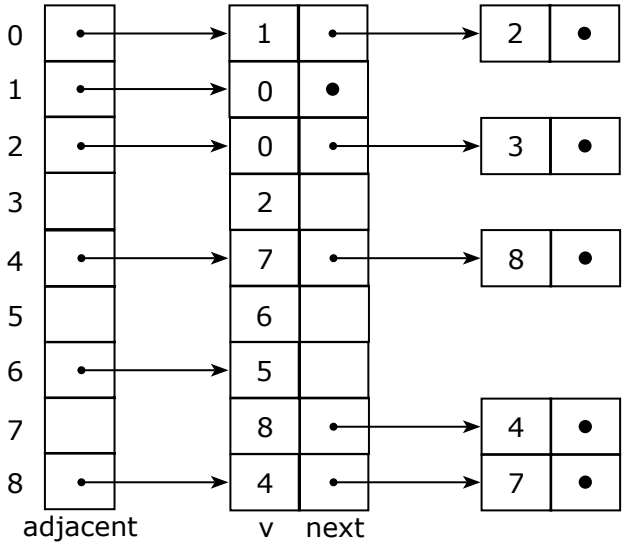

Figure 5: An Adjacency Lists Representation of a Graph

Notice that the adjacency lists representation, like the adjacency matrix representation, records every edge $\{v, w\}$ twice: once on the list for the edges of v and once on the list for the edges of w. This approach is implemented in a LinkedGraph class.

The adjacency lists data structure uses space proportional to the sum of the number of vertices and edges (the array has space for every vertex, and there are twice as many nodes in the linked lists as there are edges in the graph). This is typically much less than the space required for the adjacency matrix representation. Also, adding an edge takes $O(1)$ time and determining whether there is an edge between two vertices takes time proportional to the number of edges emanating from (one of) the vertices; this is $O(e)$ (where e is the number of edges in the graph) in the worst case, but typically it is much less. Thus the adjacency lists representation uses relatively little space but is still quite efficient.

It is convenient to use a the linked list from our Container hierarchy to realize each adjacency list. The adjacent array then holds linked lists implementing the List interface rather than pointers to nodes.

23.8 Summary and Conclusion

Graphs are an important modeling tool in computing. The graph ADT provides a few operations for building and querying a graph and this is carried over into the Graph class. The adjacency matrix technique is a contiguous implementation of the graph ADT. This representation makes graph operations efficient but uses a great deal of space. The adjacency lists approach is a linked implementation of the graph ADT. It makes graph operations nearly as fast as the adjacency list approach but uses much less space. As a rule, unless a graph is dense or it has a small number of nodes (say, less than a few hundred), the adjacency lists representation is preferable.

23.9 Review Questions

1. List several elements of the graph ADT carrier set.
2. What is the result of applying the graph ADT operation *addEdge*() twice with the same vertices? In other words, if g is a graph and v and w are vertices, what is the result of *addEdge*(*addEdge*(g,v,w),v,w)?
3. How could you iterate over every vertex in a graph?
4. Why is every edge in a graph represented twice in both the adjacency matrix and adjacency lists representations?

23.10 Exercises

1. Can a vertex be adjacent to itself?
2. If a graph has no edges, can it have any paths? If a graph has edges, does it have a longest path?
3. How many vertices must there be in the smallest cycle in an undirected graph? How many in the smallest cycle in a directed graph?
4. In Chapter 16 a tree was defined as a graph with a distinguished vertex *r*, called the *root*, such that there is exactly one simple path between each vertex in the tree and *r*. Show that this definition is equivalent to the definition stated in this chapter.
5. Can a graph have more than one spanning tree? Explain.
6. Use the operations of the graph ADT to construct the graph in Figure 2.
7. Represent the graph in Figure 2 using an adjacency matrix.
8. Represent the graph in Figure 2 using adjacency lists. Draw a picture like the one in Figure 5.
9. In Ruby a sparse graph with n vertices represented using an adjacency matrix frequently uses less than n^2 array locations because a matrix in Ruby is an array of arrays. Explain why this may cause a sparse graph to be represented using less space.
10. Write the `Graph` class in Ruby.
11. Write an `ArrayGraph` class in Ruby that represents graphs using an adjacency matrix. Its `initialize()` method should accept an argument specifying the number of vertices in the graph.
12. Write a `LinkedGraph` class in Ruby that represents graphs using adjacency lists. Its `initialize()` method should accept an argument specifying the number of vertices in the graph. Use the `containers/LinkedList` class for lists of vertices.

23.11 Review Question Answers

1. The graph ADT carrier set includes the empty graph, which as no vertices and no edges: <∅, ∅>. The next largest graph has a single vertex and no edges: <{0}, ∅>. The next largest graphs have two vertices and either no edges or one edge: <{0,1}, ∅>, and <{0,1}, {{0,1}}>. There are several graphs with three vertices: <{0, 1, 2}, ∅>, <{0, 1, 2}, {{0, 1}}>, <{0, 1, 2}, {{0, 2}}>, <{0, 1, 2}, {{1, 2}}>, <{0, 1, 2}, {{0, 1}, {0, 2}}>, <{0, 1, 2}, {{0, 1}, {1, 2}}>, <{0, 1, 2}, {{0, 2}, {1, 2}}>, <{0, 1, 2}, {{0, 1}, {0, 2}, {1, 2}}>.
2. If g is a graph and v and w are vertices, the result of *addEdge*(g,v,w) is a graph just like g except that the edge {v, w} is added to its edge set—call this result h. The result of *addEdge*(h,v,w) will be a graph just like h except that the edge {v, w} is added to the edge set of h. But this edge was already in the edge set of h, so the result is just h. Hence applying *addEdge*() to a graph with the same vertices more than once simply returns the same graph again every time after the first.
3. Iterating over every vertex in a graph g simply requires looping over every integer from 0 to *vertices*(g)−1.

4. Every edge in a graph is represented twice in both the adjacency matrix and adjacency lists representations because in each case the representation "indexes" edges by their vertices. Because each edge has two vertices, each appears twice. Note that we could easily come up with representations in which an edge appears only once. For example, we could only put the smaller of the two vertices of an edge into the adjacency array in both the adjacency matrix and adjacency lists representations. This would save about half the space, but it would make iterating over the vertices adjacent to a given vertex (which turns out to be a very important operation) very slow: we would have to search the entire representation to find all the vertices adjacent to a given vertex. So in this case space is traded for time, and we use more space to get much faster performance in an essential operation.