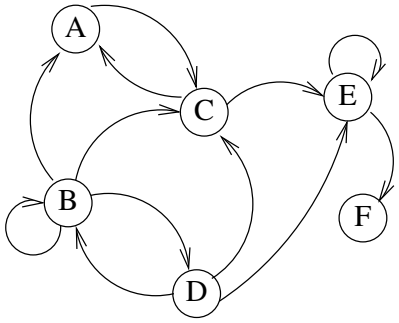
Chapter 8

Graphs

The linked structures that we have studied so far, such as linked lists and binary trees, consist of nodes that are connected to one another in a strictly specified pattern. But more general structures, consisting of nodes that can be linked in arbitrary ways, are often useful. These structures are called **graphs**. A graph consists of a set of nodes and a set of links that connect them. In the context of graphs, though, the nodes are called **vertices** and the links are called **edges**. An edge connects a pair of vertices.

Graphs come in two main varieties: *directed* and *undirected*. In a directed graph, the connections between vertices are one-way and can be visualized as arrows linking pairs of vertices:



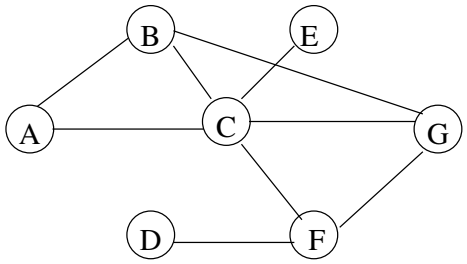
This directed graph has six vertices, labeled A through F. We will use the notation $\overrightarrow{AC}$ to represent an edge from vertex A to vertex C. More formally, we can consider an edge in a directed graph to be an ordered pair of vertices,

Definition 8.1. A **directed graph**, G, consists of a set, V, of vertices and a set, E, of edges, where each edge is an ordered pair of vertices. We write $G = (V, E)$ to indicate that V is the set of vertices in G and E is the set of edges in G. If $\overrightarrow{v_1 v_2}$ is an edge in a directed graph, then the vertex v_1 is called the **tail** and v_2 is called the **head** of the edge.

The directed graph shown above can be written as $G = (V, E)$ where the set of vertices is $V = \{A, B, C, D, E, F\}$, and the set of edges is $E = \{\overrightarrow{AC}, \overrightarrow{BA}, \overrightarrow{BB}, \overrightarrow{BC}, \overrightarrow{BD}, \overrightarrow{CA}, \overrightarrow{CE}, \overrightarrow{DC}, \overrightarrow{DE}, \overrightarrow{EE}, \overrightarrow{EF}\}$.

Note that it is possible for an edge in a directed graph to lead from a vertex back to the same vertex. That is, the head and the tail of the edge can be the same vertex. The edges $\overrightarrow{BB}$ and $\overrightarrow{EE}$ are examples of this in the above graph. Also, for a given pair of vertices, it is possible for the graph to contain two edges that connect the two vertices. The edges $\overrightarrow{AC}$ and $\overrightarrow{CA}$ are an example of this in the above graph.

In an undirected graph, an edge is a two-way connection linking a pair of vertices. An edge does not have a direction and therefore does not have a head or tail. Edges are visualized as lines rather than arrows:



Formally, an edge in an undirected graph is a *set* of two vertices, rather than an ordered pair. We will denote the edge connecting vertices A and B as $\overline{AB}$, but you should be careful to note that $\overline{AB}$ and $\overline{BA}$ represent the same edge.

Definition 8.2. An **undirected graph**, G, consists of a set, V, of vertices and a set, E, of edges, where each edge is a set of two vertices. If $\overline{v_1 v_2}$ is an edge, then the vertices v_1 and v_2 are called the **endpoints** of the edge.

The undirected graph shown above has vertex set $V = \{A, B, C, D, E, F, G\}$ and edge set $E = \{\overline{AB}, \overline{AC}, \overline{BC}, \overline{BG}, \overline{CE}, \overline{CF}, \overline{CG}, \overline{DF}, \overline{FG}\}$.

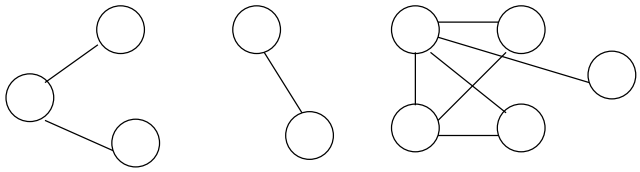
Note that an edge in an undirected graph must have two distinct endpoints. That is, we do not allow an edge in an undirected graph to connect a vertex to itself.

It should be clear that an undirected graph can be considered to be equivalent to a directed graph that has the same set of vertices and in which each edge $\overline{uv}$ in the undirected graph is replaced by two edges, $\overrightarrow{uv}$ and $\overrightarrow{vu}$, in the directed graph.

There are many basic definitions related to graphs. A **path** in a graph (directed or undirected) is a sequence of vertices $v_0 v_1 \ldots v_n$ such that for each $i = 1, 2, \ldots, n$, there is an edge in the graph from v_{i-1} to v_i. The **length** of this path is n. That is, the length of a path is the number of *edges* in the path. A path is said to be **closed** if the starting and ending vertex of the path are the same. The path $v_0 v_1 \ldots v_n$ is said to be **simple** if there are no repeated vertices in the path, except possibly that the starting vertex v_0 can be equal to the ending vertex v_n. A simple closed path is called a **cycle**. A graph that contains no cycles is said to be **acyclic**. A **d**irected **a**cyclic **g**raph, that is, a directed graph that contains no cycles, is referred to as a **dag**.

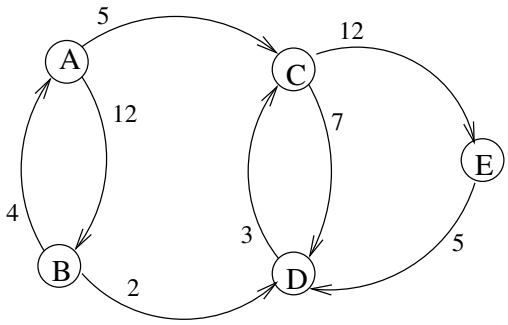
If u and v are vertices in a graph, then v is said to be **reachable** from u if either $u = v$ or there is a path in the graph from u to v. An undirected graph is said to be **connected** if any vertex in

the graph is reachable from any other vertex. The undirected graph shown above is connected, but the following graph is not:



However, this graph is clearly made up of three connected "pieces." These pieces are called the **connected components** of the graph. In an undirected graph, each vertex, v, is contained in some connected component, which is the largest connected subgraph that contains v. This connected component can also be described as the set of vertices that can be reached from v, together with all edges whose endpoints lie in that set. Connectivity is a little more complicated in directed graphs, because the fact that vertex v is reachable from vertex u does not imply that u is reachable from v. We say that a directed graph is **strongly connected** if for any pair of vertices u and v, there is both a path from u to v and a path from v to u in the graph. The directed graph at the beginning of this chapter is *not* strongly connected.

Finally, we consider one more variation on graphs. A **weighted graph** is a graph in which each edge is labeled with a number. The number associated with an edge is called the **weight** of that edge. In a weighted graph, the length of a path refers to the sum of the weights of all the edges in that path. This terminology is a little inconsistent with the previous definition of the length of a path in an unweighted graph, but it makes sense if we think of an unweighted graph as equivalent to a weighted graph in which every edge has weight one. Here is an example of a weighted directed graph:



8.1 Representations of Graphs

When graphs are used in computer programs, they must be represented by data structures. The simplest representation for a graph is an **adjacency matrix**. A matrix is simply a two-dimensional array. An adjacency matrix for a graph is a two-dimensional array of boolean values in which the number of rows and the number of columns are both equal to the number of vertices in the graph.

Note that the vertices of a graph are not listed in any particular order as far as the graph itself goes. However, in order to represent the graph as an adjacency matrix, we must assign some order to the vertices. Suppose that the graph is $G = (V, E)$. Let's say that there are n vertices and that the set of vertices is $V = \{v_0, v_1, \ldots, v_{n-1}\}$. Then we define the adjacency matrix representation of G to be an n-by-n array, A, of boolean values such that $A[i][j]$ is true if and only if there is an edge in E from v_i to v_j. We will write the boolean values in adjacency matrices as 0 and 1 rather than as *false* and *true*.

Note that we can use adjacency matrices to represent both directed and undirected graphs. The adjacency matrix for an undirected graph is symmetric; that is, $A[i][j] = A[j][i]$ for all i and j. Furthermore, the adjacency matrix for an undirected graph satisfies the condition that $A[i][i] = 0$ for all i, since an undirected graph cannot contain an edge from a vertex to itself.

We can also use a variation of adjacency matrices to represent weighted graphs. The adjacency matrix for a weighted graph is a two-dimensional array of numbers. Suppose A is such a matrix for a weighted graph and that there is an edge from vertex v_i to vertex v_j in that graph. Then $A[i][j]$ is the weight of the edge from v_i to v_j. In the case where there is no edge from v_i to v_j, we take the value of $A[i][j]$ to be ∞, even though we might not be able to represent an infinite value in a program.

As an example, consider the adjacency matrix for the directed graph at the beginning of this chapter. If we assume that the vertices are ordered in the obvious way (A, B, C, D, E, F), then the adjacency matrix is the following 6-by-6 array of boolean values:

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

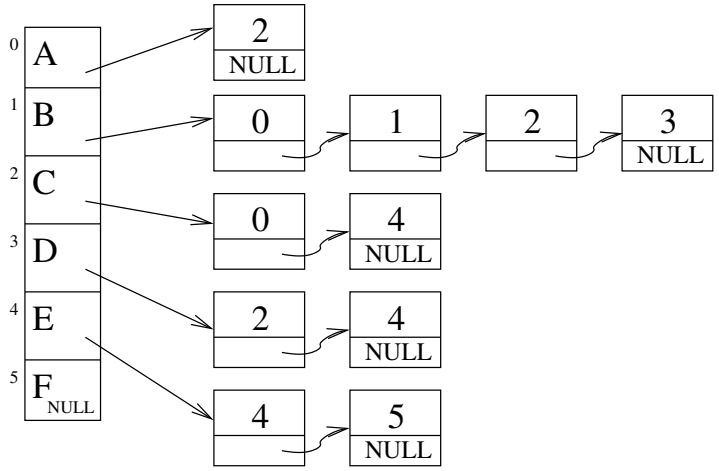
And an adjacency matrix for the weighted graph on the previous page is:

$$\begin{pmatrix} \infty & 12 & 5 & \infty & \infty \\ 4 & \infty & \infty & 2 & \infty \\ \infty & \infty & \infty & 7 & 12 \\ \infty & \infty & 3 & \infty & \infty \\ \infty & \infty & \infty & 5 & \infty \end{pmatrix}$$

The adjacency matrix representation is easy to use, but it is not appropriate for sparse graphs. A graph is said to be **sparse** if the number of edges in the graph is a small fraction of the maximum possible number of edges. In practice, sparse graphs are very common, and it is worthwhile to have a representation designed especially for them.

An alternative representation for graphs is the **edge list** representation. There are several variations on this representation. In the simplest variation, a single array of linked lists is used to store the graph's edges. The array contains one linked list for each vertex. The linked list for vertex

number k contains a list of edges that leave vertex number k. The data for each edge includes the identity of the other endpoint of the edge. For a weighted graph, it also includes the weight of that edge. The edge list representation of the directed graph from the beginning of this section can be visualized like this:



Here, the labels (A through F) of the vertices are shown in the array. The edge list for vertex A contains a single node, since there is a single edge with tail A in the graph. The 2 in this node indicates that the head of this edge is vertex number 2, that is, C. Thus, this node represents the edge $\overrightarrow{AC}$. Similarly, the vertex list for vertex B contains four nodes representing the four edges that have B as their tail: $\overrightarrow{BA}$, $\overrightarrow{BB}$, $\overrightarrow{BC}$, and $\overrightarrow{BD}$. The edge list for vertex F is empty because there are no edges that exit vertex F.

The adjacency matrix representation of a sparse graph will consist almost entirely of zeros. The edge list representation will contain only a few list nodes for each vertex. Thus, the edge list representation of a sparse graph will use less memory than an adjacency matrix. In many cases, it can also be processed more efficiently.

For some applications, it's useful to have more information in the edge list representation. For example, the node that represents an edge can contain both the head and tail vertices of that edge. Similarly, more information about the vertices, such as the vertex labels in the above example, can be stored in the array along with the pointers to the edge lists. Sometimes, in addition to having a list of edges that leave each vertex, it is convenient to have a separate list of edges that enter the vertex.

For the edge list representation of an undirected graph, it is common to represent an edge $\overline{uv}$ with two nodes, one in the edge list for vertex u and one in the edge list for vertex v. This is consistent with viewing each edge in an undirected graph as being equivalent two two edges in a directed graph.

8.2 Recursive Depth-First Search

Most of the things that we want to do with graphs involve "visiting" the various vertices or edges in the graph, presumably for the purpose of processing them in some way. This is called **traversing** the graph. In this section, we will look at several methods for traversing the graph by starting at one vertex and following edges in the graph to reach other vertices. We will discuss these graph traversal algorithms in terms of directed graphs, but they also apply to undirected graphs, keeping in mind that an edge in an undirected graph can be traversed in both directions.

The basic problem is, giving a starting vertex, to visit all the vertices that can be reached from that vertex. That is, we want to explore all the paths that begin at the starting vertex. However, we can't just keep following paths blindly, or we will end up going around in circles! A graph can contain cycles, and we have to be careful not to travel around the same cycle over and over. The solution is to "mark" each vertex as we visit it, so that if we encounter the same vertex again during our exploration of the graph, we can tell that we have already been there. For this purpose, we use an array of boolean values, *visited*[i], where the length of the array is equal to the number of vertices. Before starting a graph traversal, we set *visited*[i] = *false* for each i. When we encounter vertex number k during the traversal, we set *visited*[k] = *true* to indicate that vertex k has been visited.

The first traversal algorithm that we consider is **recursive depth-first search**. The term "depth-first" means that we explore a path as far (that is, as deeply) as we can. Only when we reach a point where we cannot continue do we double back and explore alternative paths. The recursive version of this algorithm can be expressed in pseudocode as:

```
Algorithm RecursiveDFS( Vertex v ):
   if v has already been visited:
      return
   else:
      mark v as visited
      process( v )
      for each edge e that leaves v:
         let u be the other endpoint of e
         call RecursiveDFS( u )
```

This algorithm will visit and process every vertex that can be reached from v and that has not already been visited. It does this by following each edge that leaves v and applying the same procedure recursively to the other endpoint of the edge. On the first level of the recursion, it visits vertices that can be reached from v in one step; on the second level, it visits vertices that can be reached from v in two steps; and so on. To see that the algorithm implements a depth-first search, consider what happens when there are several edges $\overrightarrow{vu_1}$, $\overrightarrow{vu_2}$, ..., leaving vertex v. The algorithm will call *RecursiveDFS*(u_1), then *RecursiveDFS*(u_2), and so on. The call to *RecursiveDFS*(u_1) visits u_1 and all vertices that can be reached from u_1. It only when this exploration is complete that the algorithm doubles back to call *RecursiveDFS*(u_2).

In order to use *RecursiveDFS*, all vertices should first be marked as unvisited. Then the algorithm should be called with some starting vertex as its parameter. The algorithm will visit all vertices that can be reached from that starting vertex.

The coding of the algorithm into C++ depends on the representation that is used for the graphs. Let's look at the two possibilities, the adjacency matrix and the edge-list representation. For either representation, we will also need the *visited* array. To represent a graph with N vertices as an adjacency matrix, we can use the declarations:

```
bool edgeExists[N][N];
bool visited[N];
```

Let's assume that these are global variables and that *edgeExists* has already been filled with values to represent the edges of the graph. Assuming that there is a function named *process* that we want to call for each edge, we can code *RecursiveDFS* as follows:

```
void RecursiveDFS( int v ) {
   if ( visited[v] ) { // Vertex has already been processed.
      return;
   }
   visited[v] = true;  // Mark this vertex as visited.
   process( v );       // Process this vertex.
   for (int u = 0; u < N; u++) {
      if ( edgeExists[v][u] ) {
            // There is an edge from vertex v to vertex u,
            // so call this function for vertex u.
         RecursiveDFS( u );
      }
   }
}
```

To apply this function to, say, vertex number 0 as the starting vertex, we would say:

```
for (int i = 0; i < N; i++)
   visited[i] = false;   // Mark all nodes as unvisited.
RecursiveDFS( 0 );
```

It's interesting to analyze the run time of this algorithm. The body of the function includes a recursive call to the same function. This means that we can't simply add up the run times for each line in *RecursiveDFS*—to do that, we would already have to know the run time! In general, it can be difficult to analyze recursive algorithms, but in this case, we can note that in *all* the recursive calls to *RecursiveDFS*, the for loop is encountered at most once for each vertex in the graph. Since there are N steps in the for loop and N vertices in the graph, we can see that the run time of the algorithm is $\mathcal{O}(N^2)$, where N is the number of vertices.

Now, let's consider the edge-list representation. For this representation, we need an array of linked lists, where each node in a list represents an edge. We can use the following C++ data structures (although in practice, we might want to store more data in the nodes):

```
struct EdgeNode {
   int head;  // Vertex number of the far endpoint of this edge.
   EdgeNode *next;   // Pointer to next edge in the list.
};

EdgeNode *edgeList[N];  // Array of pointers to edge lists,
                        // edgeList[i] contains edges that
                        // leave vertex number i.
bool visited[N];
```

Again, assuming that these are global variables and that the edges lists have already been constructed to represent a graph, we can code *RecursiveDFS* as:

```
void RecursiveDFS( int v ) {
   if ( visited[v] )
      return;
   visited[v] = true;
   process( v );
   EdgeNode *runner;     // For traversing the edge list.
   runner = edgeList[v];
   while ( runner ) {
      RecursiveDFS( runner->head );
      runner = runner->next;
   }
}
```

For this version of the algorithm, the function *RecursiveDFS* is called at most once for each edge in the graph. It follows that the run time is $\mathcal{O}(M)$, where M is the number of edges. The maximum number of edges in an N-vertex graph is N^2, so we know that $M < N^2$ and $\mathcal{O}(M)$ is no worse than $\mathcal{O}(N^2)$. However, for a sparse graph, M is much smaller than N^2 and the edge-list version of *RecursiveDFS* is more efficient than the adjacency matrix version.

8.3 Breadth-First Search

An alternative to depth-first search is **breadth-first search**. In depth-first search, vertices that are many steps away from the starting vertex can be processed before some of the vertices that are very close. In breadth-first search, vertices that are one step away from the starting vertex are all processed before any vertices that are two steps away; vertices that are two steps from the starting vertex are processed before those that are three steps away; and so on. That is, vertices are

processed in order of their distances from the starting vertex, where the distance from one vertex to another means the length of the shortest path that leads from one vertex to the other.

Breadth-first search can be implemented using a queue. You should think of it as a queue of vertices, but since the vertices are numbered, we can in fact use a queue of integers and put vertex numbers on the queue. Using the adjacency matrix representation, and assuming that the number of vertices is N, breadth-first search can be written like this:

```
void BFS( int startingVetex ) {
    IntQueue queue;  // Start with an empty queue of integers.
    for (int i = 0; i < N; i++) {
       visited[i] = false;  // Mark all vertices unvisited.
    }
    visited[ startingVertex ] = true;
    queue.enqueue( startingVertex );
    while ( ! queue.isEmpty() ) {
       int v = queue.dequeue();   // Get next vertex from queue.
       process( v );              // Process the vertex.
       for (int u = 0; u < N; u++) {
          if ( edgeExists[v][u] && visited[u] == false ) {
                // We have an edge from v to an unvisited vertex;
                // mark it as visited and add it to the queue.
             visited[u] = true;
             queue.enqueue( u );
          }
       }
    }
}
```

In this function, each vertex that is "encountered" is marked as visited and placed on the queue. It is processed when it comes off the queue. If it is encountered a second time (through an alternate path from the starting vertex), it has already been marked as visited and so is not placed on the queue again. Since no vertex is placed on the queue more than once, the while loop is executed at most N times, so it is easy to see that the run time for this algorithm is $\mathcal{O}(N^2)$, where N is the number of vertices. If we were to rewrite the algorithm in terms of the edge-list representation, the run time would be $\mathcal{O}(M)$, where M is the number of edges in the graph.

To see that this algorithm does, in fact, implement breadth-first search, consider the order in which the vertices are processed. The first vertex on the queue is the starting vertex. When the starting vertex is removed from the queue, all vertices that are connected to the starting vertex by a single edge are located and are placed on the queue. These are the vertices at distance one from the starting vertex. As these vertices are removed from the queue, the vertices to which they are directly connected are added to the queue. These are the vertices at distance two from the starting vertex, and they follow all the distance-one vertices on the queue. As the distance-two vertices are processed, the distance three vertices are encountered and are added to queue, following all the

distance-two vertices. And so on. Vertices are added to the queue in order of their distance from the starting vertex. That is, they are added in breadth-first order.

It is not possible to write breadth-first search as a recursive function. However, it *is* possible to write a depth-first search function without using recursion. All that is necessary is to replace the queue used in breadth-first search with a stack:

```
void DFS( int startingVetex ) {
    IntStack stack;  // Start with an empty stack of integers.
    for (int i = 0; i < N; i++) {
       visited[i] = false;  // Mark all vertices unvisited.
    }
    visited[ startingVertex ] = true;
    stack.push( startingVertex );
    while ( ! stack.isEmpty() ) {
       int v = stack.pop();    // Get next vertex from stack.
       process( v );           // Process the vertex.
       for (int u = 0; u < N; u++) {
          if ( edgeExists[v][u] && visited[u] == false ) {
                // We have an edge from v to an unvisited vertex;
                // mark it as visited and add it to the stack.
             visited[u] = true;
             stack.push( u );
          }
       }
    }
}
```

This algorithm does not visit the vertices in exactly the same order as *RecursiveDFS*. Can you see what the difference is and how to modify *DFS* to make the order exactly the same?