most $\sum_u n_v = 2m$. This proves the desired $O(m+n)$ bound on the running time of DFS.

(3.13) *The above implementation of the DFS algorithm runs in time $O(m+n)$ (i.e., linear in the input size), if the graph is given by the adjacency list representation.*

Finding the Set of All Connected Components

In the previous section we talked about how one can use BFS (or DFS) to find all connected components of a graph. We start with an arbitrary node s, and we use BFS (or DFS) to generate its connected component. We then find a node v (if any) that was not visited by the search from s and iterate, using BFS (or DFS) starting from v to generate its connected component—which, by (3.8), will be disjoint from the component of s. We continue in this way until all nodes have been visited.

Although we earlier expressed the running time of BFS and DFS as $O(m+n)$, where m and n are the total number of edges and nodes in the graph, both BFS and DFS in fact spend work only on edges and nodes in the connected component containing the starting node. (They never see any of the other nodes or edges.) Thus the above algorithm, although it may run BFS or DFS a number of times, only spends a constant amount of work on a given edge or node in the iteration when the connected component it belongs to is under consideration. Hence the overall running time of this algorithm is still $O(m+n)$.

3.4 Testing Bipartiteness: An Application of Breadth-First Search

Recall the definition of a bipartite graph: it is one where the node set V can be partitioned into sets X and Y in such a way that every edge has one end in X and the other end in Y. To make the discussion a little smoother, we can imagine that the nodes in the set X are colored red, and the nodes in the set Y are colored blue. With this imagery, we can say a graph is bipartite if it is possible to color its nodes red and blue so that every edge has one red end and one blue end.

The Problem

In the earlier chapters, we saw examples of bipartite graphs. Here we start by asking: What are some natural examples of a nonbipartite graph, one where no such partition of V is possible?

Clearly a triangle is not bipartite, since we can color one node red, another one blue, and then we can't do anything with the third node. More generally, consider a cycle C of odd length, with nodes numbered $1, 2, 3, \ldots, 2k, 2k+1$. If we color node 1 red, then we must color node 2 blue, and then we must color node 3 red, and so on—coloring odd-numbered nodes red and even-numbered nodes blue. But then we must color node $2k+1$ red, and it has an edge to node 1, which is also red. This demonstrates that there's no way to partition C into red and blue nodes as required. More generally, if a graph G simply *contains* an odd cycle, then we can apply the same argument; thus we have established the following.

(3.14) *If a graph G is bipartite, then it cannot contain an odd cycle.*

It is easy to recognize that a graph is bipartite when appropriate sets X and Y (i.e., red and blue nodes) have actually been identified for us; and in many settings where bipartite graphs arise, this is natural. But suppose we encounter a graph G with no annotation provided for us, and we'd like to determine for ourselves whether it is bipartite—that is, whether there exists a partition into red and blue nodes, as required. How difficult is this? We see from (3.14) that an odd cycle is one simple "obstacle" to a graph's being bipartite. Are there other, more complex obstacles to bipartitness?

Designing the Algorithm

In fact, there is a very simple procedure to test for bipartiteness, and its analysis can be used to show that odd cycles are the *only* obstacle. First we assume the graph G is connected, since otherwise we can first compute its connected components and analyze each of them separately. Next we pick any node $s \in V$ and color it red; there is no loss in doing this, since s must receive some color. It follows that all the neighbors of s must be colored blue, so we do this. It then follows that all the neighbors of *these* nodes must be colored red, their neighbors must be colored blue, and so on, until the whole graph is colored. At this point, either we have a valid red/blue coloring of G, in which every edge has ends of opposite colors, or there is some edge with ends of the same color. In this latter case, it seems clear that there's nothing we could have done: G simply is not bipartite. We now want to argue this point precisely and also work out an efficient way to perform the coloring.

The first thing to notice is that the coloring procedure we have just described is essentially identical to the description of BFS: we move outward from s, coloring nodes as soon as we first encounter them. Indeed, another way to describe the coloring algorithm is as follows: we perform BFS, coloring

s red, all of layer L_1 blue, all of layer L_2 red, and so on, coloring odd-numbered layers blue and even-numbered layers red.

We can implement this on top of BFS, by simply taking the implementation of BFS and adding an extra array `Color` over the nodes. Whenever we get to a step in BFS where we are adding a node v to a list $L[i+1]$, we assign `Color[`v`] = red` if $i+1$ is an even number, and `Color[`v`] = blue` if $i+1$ is an odd number. At the end of this procedure, we simply scan all the edges and determine whether there is any edge for which both ends received the same color. Thus, the total running time for the coloring algorithm is $O(m+n)$, just as it is for BFS.

Analyzing the Algorithm

We now prove a claim that shows this algorithm correctly determines whether G is bipartite, and it also shows that we can find an odd cycle in G whenever it is not bipartite.

(3.15) *Let G be a connected graph, and let $L_1, L_2, \ldots$ be the layers produced by BFS starting at node s. Then exactly one of the following two things must hold.*

(i) There is no edge of G joining two nodes of the same layer. In this case G is a bipartite graph in which the nodes in even-numbered layers can be colored red, and the nodes in odd-numbered layers can be colored blue.

(ii) There is an edge of G joining two nodes of the same layer. In this case, G contains an odd-length cycle, and so it cannot be bipartite.

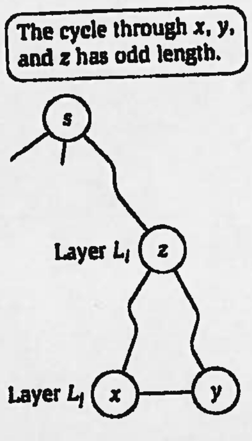

Figure 3.6 If two nodes x and y in the same layer are joined by an edge, then the cycle through x, y, and their lowest common ancestor z has odd length, demonstrating that the graph cannot be bipartite.

Proof. First consider case (i), where we suppose that there is no edge joining two nodes of the same layer. By (3.4), we know that every edge of G joins nodes either in the same layer or in adjacent layers. Our assumption for case (i) is precisely that the first of these two alternatives never happens, so this means that *every* edge joins two nodes in adjacent layers. But our coloring procedure gives nodes in adjacent layers the opposite colors, and so every edge has ends with opposite colors. Thus this coloring establishes that G is bipartite.

Now suppose we are in case (ii); why must G contain an odd cycle? We are told that G contains an edge joining two nodes of the same layer. Suppose this is the edge $e=(x,y)$, with $x, y \in L_j$. Also, for notational reasons, recall that L_0 ("layer 0") is the set consisting of just s. Now consider the BFS tree T produced by our algorithm, and let z be the node whose layer number is as large as possible, subject to the condition that z is an ancestor of both x and y in T; for obvious reasons, we can call z the *lowest common ancestor* of x and y. Suppose $z \in L_i$, where $i < j$. We now have the situation pictured in Figure 3.6. We consider the cycle C defined by following the z-x path in T, then the edge e, and then the y-z path in T. The length of this cycle is $(j - i) + 1 + (j - i)$, adding the length of its three parts separately; this is equal to $2(j - i) + 1$, which is an odd number. ∎

3.5 Connectivity in Directed Graphs

Thus far, we have been looking at problems on undirected graphs; we now consider the extent to which these ideas carry over to the case of directed graphs.

Recall that in a directed graph, the edge (u, v) has a direction: it goes from u to v. In this way, the relationship between u and v is asymmetric, and this has qualitative effects on the structure of the resulting graph. In Section 3.1, for example, we discussed the World Wide Web as an instance of a large, complex directed graph whose nodes are pages and whose edges are hyperlinks. The act of browsing the Web is based on following a sequence of edges in this directed graph; and the directionality is crucial, since it's not generally possible to browse "backwards" by following hyperlinks in the reverse direction.

At the same time, a number of basic definitions and algorithms have natural analogues in the directed case. This includes the adjacency list representation and graph search algorithms such as BFS and DFS. We now discuss these in turn.

Representing Directed Graphs

In order to represent a directed graph for purposes of designing algorithms, we use a version of the adjacency list representation that we employed for undirected graphs. Now, instead of each node having a single list of neighbors, each node has two lists associated with it: one list consists of nodes *to which* it has edges, and a second list consists of nodes *from which* it has edges. Thus an algorithm that is currently looking at a node u can read off the nodes reachable by going one step forward on a directed edge, as well as the nodes that would be reachable if one went one step in the reverse direction on an edge from u.

The Graph Search Algorithms

Breadth-first search and depth-first search are almost the same in directed graphs as they are in undirected graphs. We will focus here on BFS. We start at a node s, define a first layer of nodes to consist of all those to which s has an edge, define a second layer to consist of all additional nodes to which these first-layer nodes have an edge, and so forth. In this way, we discover nodes layer by layer as they are reached in this outward search from s, and the nodes in layer j are precisely those for which the shortest path *from* s has exactly j edges. As in the undirected case, this algorithm performs at most constant work for each node and edge, resulting in a running time of $O(m + n)$.