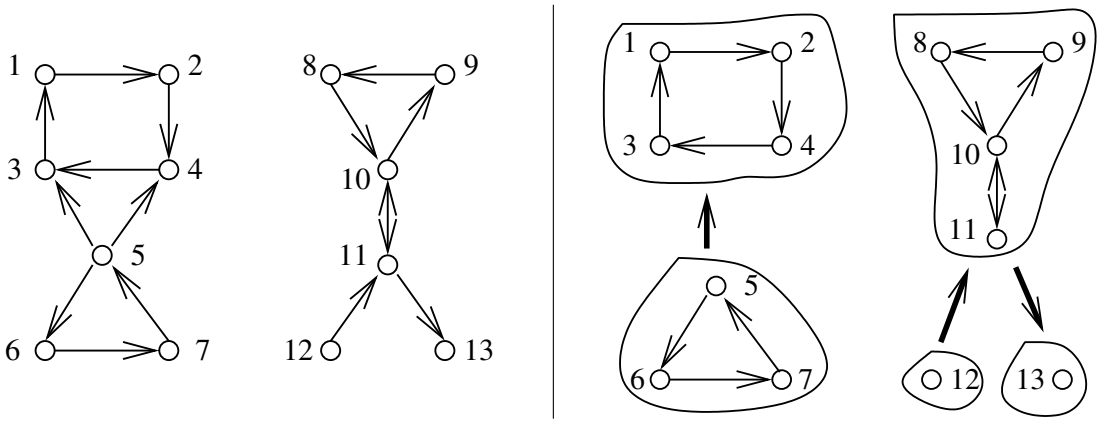

INPUT OUTPUT

15.1 Connected Components

Input description: A directed or undirected graph G.

Problem description: Identify the different pieces or components of G, where vertices x and y are members of different components if no path exists from x to y in G.

Discussion: The connected components of a graph represent, in grossest terms, the pieces of the graph. Two vertices are in the same component of G if and only if there exists some path between them.

Finding connected components is at the heart of many graph applications. For example, consider the problem of identifying natural clusters in a set of items. We represent each item by a vertex and add an edge between each pair of items deemed "similar." The connected components of this graph correspond to different classes of items.

Testing whether a graph is connected is an essential preprocessing step for every graph algorithm. Subtle, hard-to-detect bugs often result when an algorithm is run only on one component of a disconnected graph. Connectivity tests are so quick and easy that you should always verify the integrity of your input graph, even when you know for certain that it *has* to be connected.

Testing the connectivity of any undirected graph is a job for either depth-first or breadth-first search, as discussed in Section 5 (page 145). Which one you choose doesn't really matter. Both traversals initialize the *component-number* field for each vertex to 0, and then start the search for component 1 from vertex v_1. As each vertex is discovered, the value of this field is set to the current component

number. When the initial traversal ends, the component number is incremented, and the search begins again from the first vertex whose *component-number* remains 0. Properly implemented using adjacency lists (as is done in Section 5.7.1 (page 166)) this runs in $O(n+m)$ time.

Other notions of connectivity also arise in practice:

- *What if my graph is directed?* – There are two distinct notions of connected components for directed graphs. A directed graph is *weakly connected* if it would be connected by ignoring the direction of edges. Thus, a weakly connected graph consists of a single piece. A directed graph is *strongly connected* if there is a directed path between every pair of vertices. This distinction is best made clear by considering the network of one- and two-way streets in any city. The network is strongly connected if it is possible to drive legally between every two positions. The network is weakly connected when it is possible to drive legally or *illegally* between every two positions. The network is disconnected if there is no possible way to drive from a to b.

  Weakly and strongly connected components define unique partitions of the vertices. The output figure at the beginning of this section illustrates a directed graph consisting of two weakly or five strongly-connected components (also called *blocks* of G).

  Testing whether a directed graph is weakly connected can be done easily in linear time. Simply turn all edges of G into undirected edges and use the DFS-based connected components algorithm described previously. Tests for strong connectivity are somewhat more complicated. The simplest linear-time algorithm performs a search from some vertex v to demonstrate that the entire graph is reachable from v. We then construct a graph G' where we reverse all the edges of G. A traversal of G' from v suffices to decide whether all vertices of G can reach v. Graph G is strongly connected iff all vertices can reach, and are reachable, from v.

  All the strongly connected components of G can be extracted in linear time using more sophisticated DFS-based algorithms. A generalization of the above "two-DFS" idea is deceptively easy to program but somewhat subtle to understand exactly why it works:

  1. Perform a DFS, starting from an arbitrary vertex in G, and labeling each vertex in order of its completion (not discovery).
  2. Reverse the direction of each edge in G, yielding G'.
  3. Perform a DFS of G', starting from the highest numbered vertex in G. If this search does not completely traverse G', continue with the highest numbered unvisited vertex.
  4. Each DFS tree created in Step 3 is a strongly connected component.

My implementation of a single-pass algorithm appears in Section 5.10.2 (page 181). In either case, it is probably easier to start from an existing implementation than a textbook description.

- *What is the weakest point in my graph/network?* – A chain is only as strong as its weakest link. Losing one or more internal links causes a chain to become disconnected. The *connectivity* of graphs measures their strength—how many edges or vertices must be removed to disconnect it. Connectivity is an essential invariant for network design and other structural problems.

  Algorithmic connectivity problems are discussed in Section 15.8 (page 505). In particular, *biconnected components* are pieces of the graph that result from cutting the edges incident on a single vertex. All biconnected components can be found in linear time using DFS. See Section 5.9.2 (page 173) for an implementation of this algorithm. Vertices whose deletion disconnects the graph belong to more than one biconnected component, whose edges are uniquely partitioned by components.

- *Is the graph a tree? How can I find a cycle if one exists?* – The problem of cycle identification often arises, particularly with respect to directed graphs. For example, testing if a sequence of conditions can deadlock often reduces to cycle detection. If I am waiting for Fred, and Fred is waiting for Mary, and Mary is waiting for me, there is a cycle and we are all deadlocked.

  For undirected graphs, the analogous problem is tree identification. A tree is, by definition, an undirected, connected graph without any cycles. As described above, a depth-first search can be used to test whether it is connected. If the graph is connected and has $n - 1$ edges for n vertices, it is a tree.

  Depth-first search can be used to find cycles in both directed and undirected graphs. Whenever we encounter a back edge in our DFS—i.e. , an edge to an ancestor vertex in the DFS tree—the back edge and the tree together define a directed cycle. No other such cycle can exist in a directed graph. Directed graphs without cycles are called DAGs (directed acyclic graphs). Topological sorting (see Section 15.2 (page 481)) is the fundamental operation on DAGs.

Implementations: The graph data structure implementations of Section 12.4 (page 381) all include implementations of BFS/DFS, and hence connectivity testing to at least some degree. The C++ Boost Graph Library [SLL02] (*http://www.boost.org/libs/graph/doc*) provides implementations of connected components and strongly connected components. LEDA (see Section 19.1.1 (page 658)) provides these plus biconnected and triconnected components, breadth-first and depth-first search, connected components and strongly connected components, all in C++.

With respect to Java, *JUNG* (*http://jung.sourceforge.net/*) also provides biconnected component algorithms, while JGraphT (*http://jgrapht.sourceforge.net/*) does strongly connected components.

My (biased) preference for C language implementations of all basic graph connectivity algorithms. including strongly connected components and biconnected components, is the library associated with this book. See Section 19.1.10 (page 661) for details.

Notes: Depth-first search was first used to find paths out of mazes, and dates back to the nineteenth century [Luc91, Tar95]. Breadth-first search was first reported to find the shortest path by Moore in 1957 [Moo59].

Hopcroft and Tarjan [HT73b, Tar72] established depth-first search as a fundamental technique for efficient graph algorithms. Expositions on depth-first and breadth-first search appear in every book discussing graph algorithms, with [CLRS01] perhaps the most thorough description available.

The first linear-time algorithm for strongly connected components is due to Tarjan [Tar72], with expositions including [BvG99, Eve79a, Man89]. Another algorithm—simpler to program and slicker—for finding strongly connected components is due to Sharir and Kosaraju. Good expositions of this algorithm appear in [AHU83, CLRS01]. Cheriyan and Mehlhorn [CM96] propose improved algorithms for certain problems on dense graphs, including strongly connected components.

Related Problems: Edge-vertex connectivity (see page 505), shortest path (see page 489).