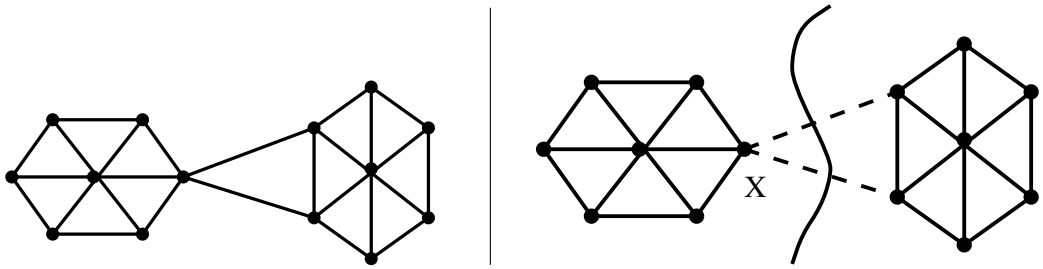

INPUT OUTPUT

15.8 Edge and Vertex Connectivity

Input description: A graph G. Optionally, a pair of vertices s and t.

Problem description: What is the smallest subset of vertices (or edges) whose deletion will disconnect G? Or which will separate s from t?

Discussion: Graph connectivity often arises in problems related to network reliability. In the context of telephone networks, the vertex connectivity is the smallest number of switching stations that a terrorist must bomb in order to separate the network—i.e. , prevent two unbombed stations from talking to each other. The edge connectivity is the smallest number of wires that need to be cut to accomplish the same objective. One well-placed bomb or snipping the right pair of cables suffices to disconnect the above network.

The edge (vertex) connectivity of a graph G is the smallest number of edge (vertex) deletions sufficient to disconnect G. There is a close relationship between the two quantities. The vertex connectivity is always less than or equal to the edge connectivity, since deleting one vertex from each edge in a cut set succeeds in disconnecting the graph. But smaller vertex subsets may be possible. The minimum vertex degree is an upper bound for both edge and vertex connectivity, since deleting all its neighbors (or cutting the edges to all its neighbors) disconnects the graph into one big and one single-vertex component.

Several connectivity problems prove to be of interest:

- *Is the graph already disconnected?* – The simplest connectivity problem is testing whether the graph is in fact connected. A simple depth-first or breadth-first search suffices to identify all connected components in linear time, as discussed in Section 15.1 (page 477). For directed graphs, the issue is whether the graph is *strongly connected*, meaning there is a directed path between any pair of vertices. In a *weakly connected* graph, there may exist paths to nodes from which there is no way to return.

- *Is there one weak link in my graph?* – We say that G is *biconnected* if no single vertex deletion is sufficient to disconnect G. Any vertex that is such a weak point is called an *articulation vertex*. A *bridge* is the analogous concept for edges, meaning a single edge whose deletion disconnects the graph.

  The simplest algorithms for identifying articulation vertices (or bridges) try deleting vertices (or edges) one by one, and then use DFS or BFS to test whether the resulting graph is still connected. More sophisticated linear-time algorithms exist for both problems, based on depth-first search. Indeed, a full implementation is given in Section 5.9.2 (page 173).

- *What if I want to split the graph into equal-sized pieces?* – What is often sought is a small cut set that breaks the graph into roughly equal-sized pieces. For example, suppose we want to split a big computer program into two maintainable units. We can construct a graph whose the vertices represent subroutines. Edges can be added between any two subroutines that interact, namely where one calls the other. We now seek to partition the subroutines into roughly equal-sized sets so that few pairs of interacting routines span the divide.

  This is the *graph partition* problem, further discussed in Section 16.6 (page 541). Although the problem is NP-complete, reasonable heuristics exist to solve it.

- *Are arbitrary cuts OK, or must I separate a given pair of vertices?* – There are two flavors of the general connectivity problem. One asks for the smallest cut-set for the entire graph, the other for the smallest set to separate s from t. Any algorithm for $(s-t)$ connectivity can be used with each of the $n(n-1)/2$ possible pairs of vertices to give an algorithm for general connectivity. Less obviously, $n-1$ runs suffice for testing edge connectivity, since we know that vertex v_1 must end up in a different component from at least one of the other $n-1$ vertices after deleting any cut set.

Edge and vertex connectivity can both be found using network-flow techniques. Network flow, discussed in Section 15.9 (page 509), interprets a weighted graph as a network of pipes where each edge has a maximum capacity and we seek to maximize the flow between two given vertices of the graph. The maximum flow between v_i, v_j in G is exactly the weight of the smallest set of edges to disconnect v_i from v_j. Thus the edge connectivity can be found by minimizing the flow between v_i and each of the $n-1$ other vertices in an unweighted graph G. Why? After deleting the minimum-edge cut set, v_i will be separated from some other vertex.

Vertex connectivity is characterized by *Menger's theorem*, which states that a graph is k-connected if and only if every pair of vertices is joined by at least k vertex-disjoint paths. Network flow can again be used to perform this calculation, since a flow of k between a pair of vertices implies k *edge*-disjoint paths. To exploit Menger's theorem, we construct a graph G' such that any set of edge-disjoint paths

in G' corresponds to vertex-disjoint paths in G. This is done by replacing each vertex v_i of G with two vertices $v_{i,1}$ and $v_{i,2}$, such that edge $(v_{i,1}, v_{i,2}) \in G'$ for all $v_i \in G$, and by replacing every edge $(v_i, x) \in G$ by edges $(v_{i,j}, x_k)$, $j \neq k \in \{0, 1\}$ in G'. Thus two edge-disjoint paths in G' correspond to each vertex-disjoint path in G.

Implementations: `MINCUTLIB` is a collection of high-performance codes for several different cut algorithms, including both flow and contraction-based methods. They were implemented by Chekuri, et al. as part of a substantial experimental study [CGK+97]. The codes are available for noncommercial use at *http://www.avglab.com/andrew/soft.html*. Also included is the full version of [CGK+97]—an excellent presentation of these algorithms and the heuristics needed to make them run fast.

Most of the graph data structure libraries of Section 15.1 (page 477) include routines for connectivity and biconnectivity testing. The C++ Boost Graph Library [SLL02] (*http://www.boost.org/libs/graph/doc*) is distinguished by also including an implementation of edge connectivity testing.

GOBLIN (*http://www.math.uni-augsburg.de/~fremuth/goblin.html*) is an extensive C++ library dealing with all of the standard graph optimization problems, including both edge and vertex connectivity.

LEDA (see Section 19.1.1 (page 658)) contains extensive support for both low-level connectivity testing (both biconnected and triconnected components) and edge connectivity/minimum-cut in C++.

Combinatorica [PS03] provides Mathematica implementations of edge and vertex connectivity, as well as connected, biconnected, and strongly connected components with bridges and articulation vertices. See Section 19.1.9 (page 661).

Notes: Good expositions on the network-flow approach to edge and vertex connectivity include [Eve79a, PS03]. The correctness of these algorithms is based on Menger's theorem [Men27] that connectivity is determined by the number of edge and vertex disjoint paths separating a pair of vertices. The maximum-flow, minimum-cut theorem is due to Ford and Fulkerson [FF62].

The theoretically fastest algorithms for minimum-cut/edge connectivity are based on graph contraction, not network flows. Contracting an edge (x, y) in a graph G merges the two incident vertices into one, removing self-loops but leaving multiedges. Any sequence of such contractions can raise (but not lower) the minimum cut in G, and leaves the cut unchanged if no edge of the cut is contracted. Karger gave a beautiful randomized algorithm for minimum cut, observing that the minimum cut is left unchanged with nontrivial probability over the course of any random series of deletions. The fastest version of Karger's algorithm runs in $(m \lg^3 n)$ expected time [Kar00]. See Motwani and Raghavan [MR95] for an excellent treatment of randomized algorithms, including a presentation of Karger's algorithm.

Nagamouchi and Ibaraki [NI92] give a deterministic contraction-based algorithm to find the minimum cut in $O(n(m + n \log n))$. In each round, this algorithm identifies and contracts an edge that is provably not in the minimum cut. See [CGK+97, NOI94] for experimental comparisons of algorithms for finding minimum cuts.

Minimum-cut methods have found many applications in computer vision, including image segmentation. Boykov and Kolmogorov [BK04] report on an experimental evaluation of minimum-cut algorithms in this context.

A nonflow-based algorithm for edge k-connectivity in $O(kn^2)$ is due to Matula [Mat87]. Faster k-connectivity algorithms are known for certain small values of k. All three-connected components of a graph can be generated in linear time [HT73a], while $O(n^2)$ suffices to test 4-connectivity [KR91].

Related Problems: Connected components (see page 477), network flow (see page 509), graph partition (see page 541).