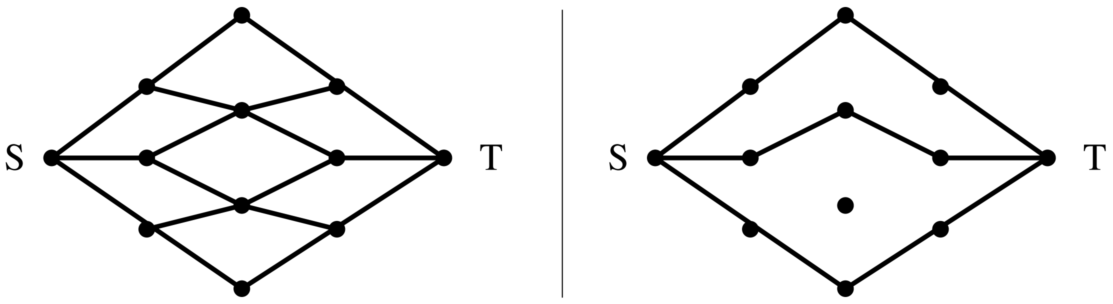

INPUT OUTPUT

15.9 Network Flow

Input description: A directed graph G, where each edge $e = (i, j)$ has a capacity c_e. A source node s and sink node t.

Problem description: What is the maximum flow you can route from s to t while respecting the capacity constraint of each edge?

Discussion: Applications of network flow go far beyond plumbing. Finding the most cost-effective way to ship goods between a set of factories and a set of stores defines a network-flow problem, as do many resource-allocation problems in communications networks.

The real power of network flow is (1) that a surprising variety of linear programming problems arising in practice can be modeled as network-flow problems, and (2) that network-flow algorithms can solve these problems much faster than general-purpose linear programming methods. Several graph problems we have discussed in this book can be solved using network flow, including bipartite matching, shortest path, and edge/vertex connectivity.

The key to exploiting this power is recognizing that your problem can be modeled as network flow. This requires experience and study. My recommendation is that you first construct a linear programming model for your problem and then compare it with linear programs for the two primary classes of network flow problems: *maximum flow* and *minimum-cost flow*:

- *Maximum Flow* – Here we seek the heaviest possible flow from s to t, given the edge capacity constraints of G. Let x_{ij} be a variable accounting for the flow from vertex i through directed edge (i, j). The flow through this edge is constrained by its capacity c_{ij}, so

  $$0 \leq x_{ij} \leq c_{ij} \text{ for } 1 \leq i, j \leq n$$

Furthermore, an equal flow comes in as goes out at each nonsource or sink vertex, so

$$\sum_{j=1}^{n} x_{ij} - \sum_{j=1}^{n} x_{ji} = 0 \text{ for } 1 \leq i \leq n$$

We seek the assignment that maximizes the flow into sink t, namely $\sum_{i=1}^{n} x_{it}$

- *Minimum Cost Flow* – Here we have an extra parameter for each edge (i, j), namely the cost (d_{ij}) of sending one unit of flow from i to j. We also have a targeted amount of flow f we want to send from s to t at minimum total cost. Hence, we seek the assignment that minimizes

  $$\sum_{j=1}^{n} d_{ij} \cdot x_{ij}$$

  subject to the edge and vertex capacity constraints of maximum flow, plus the additional restriction that $\sum_{i=1}^{n} x_{it} = f$.

Special considerations include:

- *What if I have multiple sources and/or sinks?* – No problem. We can handle this by modifying the network to create a vertex to serve as a super-source that feeds all the sources, and a super-sink that drains all the sinks.

- *What if all arc capacities are identical, either 0 or 1?* – Faster algorithms exist for 0-1 network flows. See the Notes section for details.

- *What if all my edge costs are identical?* – Use the simpler and faster algorithms for solving maximum flow as opposed to minimum-cost flow. Max-flow without edge costs arises in many applications, including edge/vertex connectivity and bipartite matching.

- *What if I have multiple types of material moving through the network?* – In a telecommunications network, every message has a given source and destination. Each destination needs to receive *exactly* those calls sent to it, not an equal amount of communication from arbitrary places. This can be modeled as a *multicommodity flow* problem, where each call defines a different commodity and we seek to satisfy all demands without exceeding the total capacity of any edge.

  Linear programming will suffice for multicommodity flow if fractional flows are permitted. Unfortunately, integral multicommodity flow is NP-complete, even for only two commodities.

Network flow algorithms can be complicated, and significant engineering is required to optimize performance. Thus, we strongly recommend that you use an existing code rather than implement your own. Excellent codes are available and described below. The two primary classes of algorithms are:

- *Augmenting path methods* – These algorithms repeatedly find a path of positive capacity from source to sink and add it to the flow. It can be shown that the flow through a network is optimal if and only if it contains no augmenting path. Since each augmentation adds something to the flow, we eventually find the maximum. The difference between network-flow algorithms is in *how* they select the augmenting path. If we are not careful, each augmenting path will add but a little to the total flow, and so the algorithm might take a long time to converge.

- *Preflow-push methods* – These algorithms push flows from one vertex to another, initially ignoring the constraint that in-flow must equal out-flow at each vertex. Preflow-push methods prove faster than augmenting-path methods, essentially because multiple paths can be augmented simultaneously. These algorithms are the method of choice and are implemented in the best codes described in the next section.

Implementations: High-performance codes for both maximum flow and minimum cost flow were developed by Andrew Goldberg and his collaborators. The codes `HIPR` and `PRF` [CG94] are provided for maximum flow, with the proviso that `HIPR` is recommended in most cases. For minimum-cost flow, the code of choice is `CS` [Gol97]. Both are written in C and available for noncommercial use from *http://www.avglab.com/andrew/soft.html.*

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 several different algorithms for both maximum flow and minimum-cost flow. The same holds true for LEDA, if a commercial-strength solution is needed. See Section 19.1.1 (page 658).

The First DIMACS Implementation Challenge on Network Flows and Matching [JM93] collected several implementations and generators for network flow, which can be obtained by anonymous FTP from *dimacs.rutgers.edu* in the directory *pub/netflow/maxflow*. These include: (1) a preflow-push network flow implementation in C by Edward Rothberg, and (2) an implementation of eleven network flow variants in C, including the older Dinic and Karzanov algorithms by Richard Anderson and Joao Setubal.

Notes: The primary book on network flows and its applications is [AMO93]. Good expositions on network flow algorithms old and new include [CCPS98, CLRS01, PS98]. Expositions on the hardness of multicommodity flow [Ita78] include [Eve79a].

There is a very close connection to maximum flow and edge connectivity in graphs. The fundamental maximum-flow, minimum-cut theorem is due to Ford and Fulkerson

[FF62]. See Section 15.8 (page 505) for simpler and more efficient algorithms to compute the minimum cut in a graph.

Conventional wisdom holds that network flow should be computable in $O(nm)$ time, and there has been steady progress in lowering the time complexity. See [AMO93] for a history of algorithms for the problem. The fastest known general network flow algorithm runs in $O(nm \lg(n^2/m))$ time [GT88]. Empirical studies of minimum-cost flow algorithms include [GKK74, Gol97].

Information flows through a network can be modeled as multicommodity flows through a network, with the observation that replicating and manipulating information at internal nodes can eliminate the need for distinct sources to sink paths when multiple sinks are interested in the same information. The field of *network coding* [YLCZ05] uses such ideas to achieve information flows through networks at the theoretical limits of the max-flow, min-cut theorem.

Related Problems: Linear programming (see page 411), matching (see page 498), connectivity (see page 505).