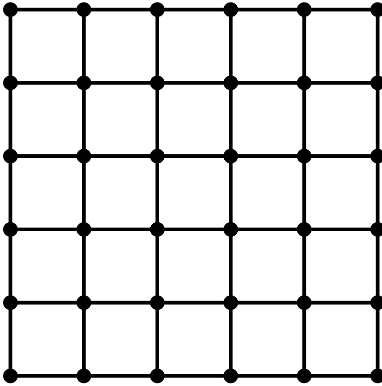

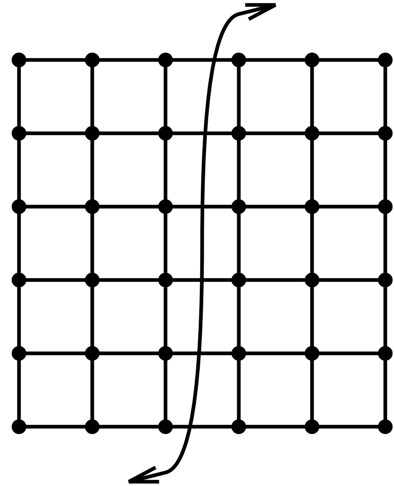

INPUT OUTPUT

16.6 Graph Partition

Input description: A (weighted) graph $G = (V, E)$ and integers k and m.

Problem description: Partition the vertices into m roughly equal-sized subsets such that the total edge cost spanning the subsets is at most k.

Discussion: Graph partitioning arises in many divide-and-conquer algorithms, which gain their efficiency by breaking problems into equal-sized pieces such that the respective solutions can easily be reassembled. Minimizing the number of edges cut in the partition usually simplifies the task of merging.

Graph partition also arises when we need to cluster the vertices into logical components. If edges link "similar" pairs of objects, the clusters remaining after partition should reflect coherent groupings. Large graphs are often partitioned into reasonable-sized pieces to improve data locality or make less cluttered drawings.

Finally, graph partition is a critical step in many parallel algorithms. Consider the finite element method, which is used to compute the physical properties (such as stress and heat transfer) of geometric models. Parallelizing such calculations requires partitioning the models into equal-sized pieces whose interface is small. This is a graph-partitioning problem, since the topology of a geometric model is usually represented using a graph.

Several different flavors of graph partitioning arise depending on the desired objective function:

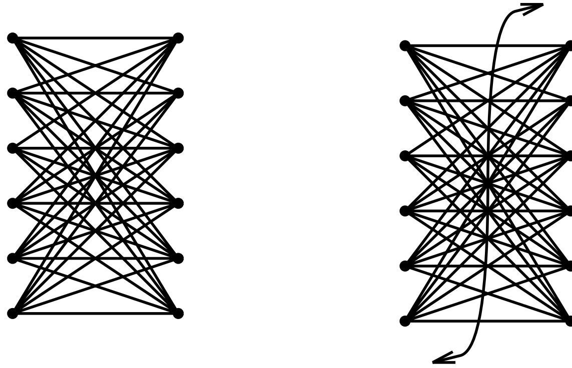

Figure 16.1: The maximum cut of a graph

- *Minimum cut set* – The *smallest* set of edges to cut that will disconnect a graph can be efficiently found using network flow or randomized algorithms. See Section 15.8 (page 505) for more on connectivity algorithms. The smallest cutset might split off only a single vertex, so the resulting partition could be very unbalanced.

- *Graph partition* – A better partition criterion seeks a small cut that partitions the vertices into roughly equal-sized pieces. Unfortunately, this problem is NP-complete. Fortunately, the heuristics discussed below work well in practice.

  Certain special graphs always have small *separators*, that partition the vertices into balanced pieces. For any tree, there always exists a single vertex whose deletion partitions the tree so that no component contains more than $n/2$ of the original n vertices. These components need not always be connected; consider the separating vertex of a star-shaped tree. This separating vertex can be found in linear time using depth first-search. Every planar graph has a set of $O(\sqrt{n})$ vertices whose deletion leaves no component with more than $2n/3$ vertices. These separators provide a useful way to decompose geometric models, which are often defined by planar graphs.

- *Maximum cut* – Given an electronic circuit specified by a graph, the *maximum cut* defines the largest amount of data communication that can simultaneously occur in the circuit. The highest-speed communications channel should thus span the vertex partition defined by the maximum edge cut. Finding the maximum cut in a graph is NP-complete [Kar72], however heuristics similar to those of graph partitioning work well.

The basic approach for dealing with graph partitioning or max-cut problems is to construct an initial partition of the vertices (either randomly or according

to some problem-specific strategy) and then sweep through the vertices, deciding whether the size of the cut would improve if we moved this vertex over to the other side. The decision to move vertex v can be made in time proportional to its degree, by identifying which side of the partition contains more of v's neighbors. Of course, the desirable side for v may change after its neighbors jump, so several iterations are likely to be needed before the process converges on a local optimum. Even so, such a local optimum can be arbitrarily far away from the global max-cut.

There are many variations of this basic procedure, by changing the order we test the vertices in or moving clusters of vertices simultaneously. Using some form of randomization, particularly simulated annealing, is almost certain to be a good idea. When more than two components are desired, the partitioning heuristic should be applied recursively.

Spectral partitioning methods use sophisticated linear algebra techniques to obtain a good partitioning. The spectral bisection method uses the second-lowest eigenvector of the *Laplacian matrix* of the graph to partition it into two pieces. Spectral methods tend to do a good job of identifying the general area to partition, but the results can be improved by cleaning up with a local optimization method.

Implementations: Chaco is a widely-used graph partitioning code designed to partition graphs for parallel computing applications. It employs several different partitioning algorithms, including both Kernighan-Lin and spectral methods. Chaco is available at *http://www.cs.sandia.gov/∼bahendr/chaco.html*.

METIS (*http://glaros.dtc.umn.edu/gkhome/views/metis*) is another well-regarded code for graph partitioning. It has successfully partitioned graphs with over 1,000,000 vertices. Available versions include one variant designed to run on parallel machines and another suitable for partitioning hypergraphs. Other respected codes include Scotch (*http://www.labri.fr/perso/pelegrin/scotch/*) and JOSTLE (*http://staffweb.cms.gre.ac.uk/∼wc06/jostle/*).

Notes: The fundamental local improvement heuristics for graph partitioning are due to Kernighan-Lin [KL70] and Fiduccia-Mattheyses [FM82]. Spectral methods for graph partition are discussed in [Chu97, PSL90]. Empirical results on graph partitioning heuristics include [BG95, LR93].

The planar separator theorem and an efficient algorithm for finding such a separator are due to Lipton and Tarjan [LT79, LT80]. For experiences in implementing planar separator algorithms, see [ADGM04, HPS$^+$05].

Any random vertex partition will expect to cut half of the edges in the graph, since the probability that the two vertices defining an edge end up on different sides of the partition is 1/2. Goemans and Williamson [GW95] gave an 0.878-factor approximation algorithm for maximum-cut, based on semi-definite programming techniques. Tighter analysis of this algorithm was followed by Karloff [Kar96].

Related Problems: Edge/vertex connectivity (see page 505), network flow (see page 509).