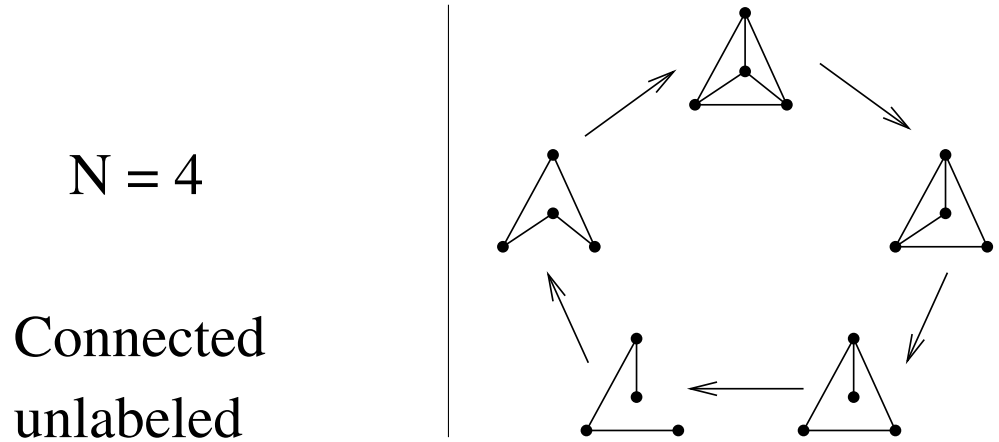

INPUT OUTPUT

14.7 Generating Graphs

Input description: Parameters describing the desired graph, including the number of vertices n, and the number of edges m or edge probability p.

Problem description: Generate (1) all, or (2) a random, or (3) the next graph satisfying the parameters.

Discussion: Graph generation typically arises in constructing test data for programs. Perhaps you have two different programs that solve the same problem, and you want to see which one is faster or make sure that they always give the same answer. Another application is experimental graph theory, verifying whether a particular property is true for all graphs or how often it is true. It is much easier to believe the four-color theorem after you have demonstrated four colorings for all planar graphs on 15 vertices.

A different application of graph generation arises in network design. Suppose you need to design a network linking ten machines using as few cables as possible, such that this network can survive up to two vertex failures. One approach is to test all the networks with a given number of edges until you find one that will work. For larger graphs, heuristic approaches like simulated annealing will likely be necessary.

Many factors complicate the problem of generating graphs. First, make sure you know exactly what types of graphs you want to generate. Figure 5.2 on page 147 illustrates several important properties of graphs. For purposes of generation, the most important questions are:

- *Do I want labeled or unlabeled graphs?* – The issue here is whether the names of the vertices matter in deciding whether two graphs are the same. In generating *labeled graphs*, we seek to construct all possible labelings of all possible graph topologies. In generating *unlabeled graphs*, we seek only one representative for each topology and ignore labelings. For example, there are only two connected unlabeled graphs on three vertices—a triangle and a simple path. However, there are four connected labeled graphs on three vertices—one triangle and three 3-vertex paths, each distinguished by the name of their central vertex. In general, labeled graphs are much easier to generate. However, there are so many more of them that you quickly get swamped with isomorphic copies of the same few graphs.

- *Do I want directed or undirected graphs?* – Most natural generation algorithms generate undirected graphs. These can be turned into directed graphs by flipping coins to orient the edges. Any graph can be oriented to be directed and acyclic (i.e. , a DAG) by randomly permuting the vertices on a line and aiming each edge from left to right. With all such ideas, careful thought must be given to decide whether you are generating all graphs uniformly at random, and how much this matters to you.

You also must define what you mean by random. There are three primary models of random graphs, all of which generate graphs according to different probability distributions:

- *Random edge generation* – The first model is parameterized by a given edge probability p. Typically, $p = 0.5$, although smaller values can be used to construct sparser random graphs. In this model, a coin is flipped for each pair of vertices x and y to decide whether to add an edge (x, y). All *labeled* graphs will be generated with equal probability when $p = 1/2$.

- *Random edge selection* – The second model is parameterized by the desired number of edges m. It selects m distinct edges uniformly at random. One way to do this is by drawing random (x, y)-pairs and creating an edge if that pair is not already in the graph. An alternative approach to computing the same things constructs the set of $\binom{n}{2}$ possible edges and selects a random m-subset of them, as discussed in Section 14.5 (page 452).

- *Preferential attachment* – Under a rich-get-richer model, newly created edges are likely to point to high-degree vertices than low-degree ones. Consider new links (edges) being added to the graph of webpages. Under any realistic web generation model, it is much more likely the next link will be to Google than *http://www.cs.sunysb.edu/∼algorith.*[1] Selecting the next neighbor with

[1]Please link to us from your homepage to correct for this travesty.

probability proportional to its degree yields graphs with *power law* properties encountered in many real networks.

Which of these options best models your application? Probably none of them. Random graphs have very little structure by definition. But graphs are used to model relationships, which are often highly structured. Experiments conducted on random graphs, although interesting and easy to perform, often fail to capture what you are looking for.

An alternative to random graphs is "organic" graphs—graphs that reflect the relationships among real-world objects. The Stanford GraphBase, discussed below, is an outstanding source of organic graphs. Many raw sources of relationships are available on the web that can be turned into interesting organic graphs with a little programming and imagination. Consider the graph defined by a set of web-pages, with any hyperlink between two pages defining an edge. Or, what about the graph implicit in railroad, subway, or airline networks, with vertices being stations and edges between two stations connected by direct service? As a final example, every large computer program defines a call graph, where the vertices represent subroutines, and there is an edge (x, y) if x calls y.

Two classes of graphs have particularly interesting generation algorithms:

- *Trees* – Prüfer codes provide a simple way to rank and unrank *labeled* trees and thus solve all standard generation problems (see Section 14.4 (page 448)). There are exactly n^{n-2} labeled trees on n vertices, and exactly that many strings of length $n-2$ on the alphabet $\{1, 2, \ldots, n\}$.

  The key to Prüfer's bijection is the observation that every tree has at least two vertices of degree 1. Thus, in any labeled tree the vertex v incident on the leaf with lowest label is well defined. We take v to be S_1, the first character in the code. We then delete the associated leaf and repeat the procedure until only two vertices are left. This defines a unique code S for any given labeled tree that can be used to rank the tree. To go from code to tree, observe that the degree of vertex v in the tree is one more than the number of times v occurs in S. The lowest-labeled leaf will be the smallest integer missing from S, which when paired with S_1 determines the first edge of the tree. The entire tree follows by induction.

  Algorithms for efficiently generating unlabeled rooted trees are discussed in the Implementation section.

- *Fixed degree sequence graphs* – The *degree sequence* of a graph G is an integer partition $p = (p_1, \ldots, p_n)$, where p_i is the degree of the ith highest-degree vertex of G. Since each edge contributes to the degree of two vertices, p is an integer partition of $2m$, where m is the number of edges in G.

  Not all partitions correspond to degree sequences of graphs. However, there is a recursive construction that constructs a graph with a given degree sequence if one exists. If a partition is realizable, the highest-degree vertex v_1 can be

connected to the next p_1 highest-degree vertices in G, or the vertices corresponding to parts $p_2, \ldots, p_{p_1+1}$. Deleting p_1 and decrementing $p_2, \ldots, p_{p_1+1}$ yields a smaller partition, which we recur on. If we terminate without ever creating negative numbers, the partition was realizable. Since we always connect the highest-degree vertex to other high-degree vertices, it is important to reorder the parts of the partition by size after each iteration.

Although this construction is deterministic, a semi-random collection of graphs realizing this degree sequence can be generated from G using *edge-flipping* operations. Suppose edges (x, y) and (w, z) are in G, but (x, w) and (y, z) are not. Exchanging these pairs of edges creates a different (not necessarily connected) graph without changing the degrees of any vertex.

Implementations: The Stanford GraphBase [Knu94] is perhaps most useful as an instance generator for constructing graphs to serve as test data for other programs. It incorporates graphs derived from interactions of characters in famous novels, Roget's Thesaurus, the Mona Lisa, expander graphs, and the economy of the United States. It also contains routines for generating binary trees, graph products, line graphs, and other operations on basic graphs. Finally, because of its machine-independent, random-number generators, it provides a way to construct random graphs such that they can be reconstructed elsewhere, thus making them perfect for experimental comparisons of algorithms. See Section 19.1.8 (page 660) for additional information.

Combinatorica [PS03] provides Mathematica generators for such graphs as stars, wheels, complete graphs, random graphs and trees, and graphs with a given degree sequence. Further, it includes operations to construct more interesting graphs from these, including join, product, and line graph.

The *Combinatorial Object Server* (*http://theory.cs.uvic.ca/*) developed by Frank Ruskey of the University of Victoria provides routines for generating both free and rooted trees.

Viger [VL05] has made available a C++ implementation of his algorithm to generate simple connected graphs with a prescribed degree sequence. See *http://www.liafa.jussieu.fr/~fabien/generation/*.

The graph isomorphism testing program Nauty (see Section 16.9 (page 550)) includes a suite of programs for generating nonisomorphic graphs, plus special generators for bipartite graphs, digraphs, and multigraphs. They are available at *http://cs.anu.edu.au/~bdm/nauty/*.

The mathematicians Brendan McKay (*http://cs.anu.edu.au/~bdm/data/*) and Gordon Royle (*http://people.csse.uwa.edu.au/gordon/data.html*) provide exhaustive catalogs of several families of graphs and trees up to the largest reasonable number of vertices.

Nijenhuis and Wilf [NW78] provide efficient Fortran routines to enumerate all labeled trees via Prüfer codes and to construct random unlabeled rooted trees. See Section 19.1.10 (page 661). Kreher and Stinson [KS99] generate labeled trees in C,

with implementations available at *http://www.math.mtu.edu/~kreher/cages/Src.html*.

Notes: Extensive literature exists on generating graphs uniformly at random. Surveys include [Gol93, Tin90]. Closely related to the problem of generating classes of graphs is counting them. Harary and Palmer [HP73] survey results in graphical enumeration.

Knuth [Knu06] is the best recent reference on generating trees. The bijection between $n-2$ strings and labeled trees is due to Prüfer [Prü18].

Random graph theory is concerned with the properties of random graphs. Threshold laws in random graph theory define the edge density at which properties such as connectedness become highly likely to occur. Expositions on random graph theory include [Bol01, JLR00].

The preferential attachment model of graphical evolution has emerged relatively recently in the study of networks. See [Bar03, Wat04] for introductions to this exciting field.

An integer partition is *graphic* if there exists a simple graph with that degree sequence. Erdős and Gallai [EG60] proved that a degree sequence is graphic if and only if the sequence observes the following condition for each integer $r < n$:

$$\sum_{i=1}^{r} d_i \leq r(r-1) + \sum_{i=r+1}^{n} \min(r, d_i)$$

Related Problems: Generating permutations (see page 448), graph isomorphism (see page 550).