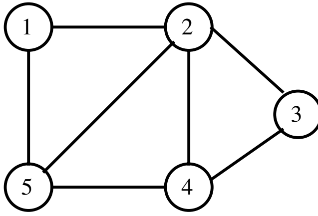

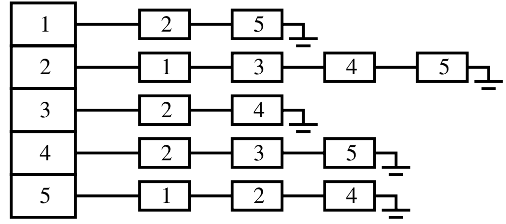

INPUT OUTPUT

12.4 Graph Data Structures

Input description: A graph G.

Problem description: Represent the graph G using a flexible, efficient data structure.

Discussion: The two basic data structures for representing graphs are *adjacency matrices* and *adjacency lists*. Full descriptions of these data structures appear in Section 5.2 (page 151), along with an implementation of adjacency lists in C. In general, for most things, adjacency lists are the way to go.

The issues in deciding which data structure to use include:

- *How big will your graph be?* – How many vertices will it have, both typically and in the worse case? Ditto for the number of edges? Graphs with 1,000 vertices imply adjacency matrices with 1,000,000 entries. This seems too be the boundary of reality. Adjacency matrices make sense only for small or very dense graphs.

- *How dense will your graph be?* – If your graph is very dense, meaning that a large fraction of the vertex pairs define edges, there is probably no compelling reason to use adjacency lists. You will be doomed to using $\Theta(n^2)$ space anyway. Indeed, for complete graphs, matrices will be more concise due to the elimination of pointers.

- *Which algorithms will you be implementing?* – Certain algorithms are more natural on adjacency matrices (such as all-pairs shortest path) and others favor adjacency lists (such as most DFS-based algorithms). Adjacency matrices win for algorithms that repeatedly ask, "Is (i, j) in G?" However, most graph algorithms can be designed to eliminate such queries.

- *Will you be modifying the graph over the course of your application?* – Efficient *static graph* implementations can be used when no edge insertion/deletion operations will done following initial construction. Indeed, more

common than modifying the topology of the graph is modifying the *attributes* of a vertex or edge of the graph, such as size, weight, label, or color. Attributes are best handled as extra fields in the vertex or edge records of adjacency lists.

Building a good general purpose graph type is a substantial project. For this reason, we suggest that you check out existing implementations (particularly LEDA) before hacking up your own. Note that it costs only time linear in the size of the larger data structure to convert between adjacency matrices and adjacency lists. This conversion is unlikely to be the bottleneck in any application, so you may decide to use both data structures if you have the space to store them. This usually isn't necessary, but might prove simplest if you are confused about the alternatives.

Planar graphs are those that can be drawn in the plane so no two edges cross. Graphs arising in many applications are planar by definition, such as maps of countries. Others are planar by happenstance, like trees. Planar graphs are always sparse, since any n-vertex planar graph can have at most $3n - 6$ edges. Thus they should be represented using adjacency lists. If the planar drawing (or *embedding*) of the graph is fundamental to what is being computed, planar graphs are best represented geometrically. See Section 15.12 (page 520) for algorithms for constructing planar embeddings from graphs. Section 17.15 (page 614) discusses algorithms for maintaining the graphs implicit in the arrangements of geometric objects like lines and polygons.

Hypergraphs are generalized graphs where each edge may link subsets of more than two vertices. Suppose we want to represent who is on which Congressional committee. The vertices of our hypergraph would be the individual congressmen, while each hyperedge would represent one committee. Such arbitrary collections of subsets of a set are naturally thought of as hypergraphs.

Two basic data structures for hypergraphs are:

- *Incidence matrices*, which are analogous to adjacency matrices. They require $n \times m$ space, where m is the number of hyperedges. Each row corresponds to a vertex, and each column to an edge, with a nonzero entry in $M[i, j]$ iff vertex i is incident to edge j. On standard graphs there are two nonzero entries in each column. The degree of each vertex governs the number of nonzero entries in each row.

- *Bipartite incidence structures*, which are analogous to adjacency lists, and hence suited for sparse hypergraphs. There is a vertex of the incidence structure associated with each edge and vertex of the hypergraphs, and an edge (i, j) in the incidence structure if vertex i of the hypergraph appears in edge j of the hypergraph. Adjacency lists are typically used to represent this incidence structure. Drawing the associated bipartite graph provides a natural way to visualize the hypergraph.

Special efforts must be taken to represent very large graphs efficiently. However, interesting problems have been solved on graphs with millions of edges and

vertices. The first step is to make your data structure as lean as possible, by packing your adjacency matrix in a bit vector (see Section 12.5 (page 385)) or removing unnecessary pointers from your adjacency list representation. For example, in a static graph (which does not support edge insertions or deletions) each edge list can be replaced by a packed array of vertex identifiers, thus eliminating pointers and potentially saving half the space.

If your graph is extremely large, it may become necessary to switch to a hierarchical representation, where the vertices are clustered into subgraphs that are compressed into single vertices. Two approaches exist to construct such a hierarchical decomposition. The first breaks the graph into components in a natural or application-specific way. For example, a graph of roads and cities suggests a natural decomposition—partition the map into districts, towns, counties, and states. The other approach runs a graph partition algorithm discussed as in Section 16.6 (page 541). If you have one, a natural decomposition will likely do a better job than some naive heuristic for an NP-complete problem. If your graph is really unmanageably large, you cannot afford to do a very good job of algorithmically partitioning it. First verify that standard data structures fail on your problem before attempting such heroic measures.

Implementations: LEDA (see Section 19.1.1 (page 658)) provides the best graph data type currently implemented in C++. It is now a commercial product. You should at least study the methods it provides for graph manipulation, so as to see how the right level of abstract graph type makes implementing algorithms very clean and easy.

The C++ Boost Graph Library [SLL02] (*http://www.boost.org/libs/graph/doc*) is more readily available. Implementations of adjacency lists, matrices, and edge lists are included, along with a reasonable library of basic graph algorithms. Its interface and components are generic in the same sense as the C++ standard template library (STL).

JUNG (*http://jung.sourceforge.net/*) is a Java graph library particularly popular in the social networks community. The *Data Structures Library in Java* (JDSL) provides a comprehensive implementation with a decent algorithm library, and is available for noncommercial use at *http://www.jdsl.org/*. See [GTV05, GT05] for more detailed guides to JDSL. JGraphT (*http://jgrapht.sourceforge.net/*) is a more recent development with similar functionality.

The Stanford Graphbase (see Section 19.1.8 (page 660)) provides a simple but flexible graph data structure in CWEB, a literate version of the C language. It is instructive to see what Knuth does and does not place in his basic data structure, although we recommend other implementations as a better basis for further development.

My (biased) preference in C language graph types is the library from my book *Programming Challenges* [SR03]. See Section 19.1.10 (page 661) for details. Simple graph data structures in Mathematica are provided by *Combinatorica* [PS03], with a library of algorithms and display routines. See Section 19.1.9 (page 661).

Notes: The advantages of adjacency list data structures for graphs became apparent with the linear-time algorithms of Hopcroft and Tarjan [HT73b, Tar72]. The basic adjacency list and matrix data structures are presented in essentially all books on algorithms or data structures, including [CLRS01, AHU83, Tar83]. Hypergraphs are presented in Berge [Ber89]

The improved efficiency of static graph types was revealed by Naher and Zlotowski [NZ02], who sped up certain LEDA graph algorithms by a factor of four by simply switching to a more compact graph structure.

An interesting question concerns minimizing the number of bits needed to represent arbitrary graphs on n vertices, particularly if certain operations must be supported efficiently. Such issues are surveyed in [vL90b].

Dynamic graph algorithms are data structures that maintain quick access to an invariant (such as minimum spanning tree or connectivity) under edge insertion and deletion. *Sparsification* [EGIN92] is a general approach to constructing dynamic graph algorithms. See [ACI92, Zar02] for experimental studies on the practicality of dynamic graph algorithms.

Hierarchically-defined graphs arise often in VLSI design problems, because designers make extensive use of cell libraries [Len90]. Algorithms specifically for hierarchically-defined graphs include planarity testing [Len89], connectivity [LW88], and minimum spanning trees [Len87a].

Related Problems: Set data structures (see page 385), graph partition (see page 541).