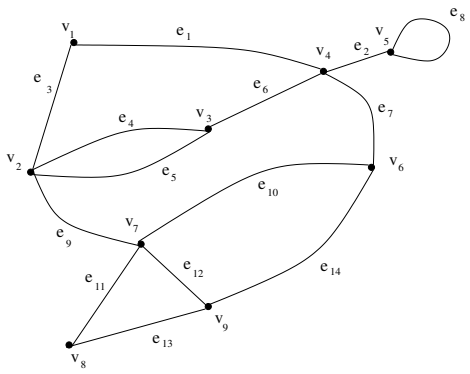
8.6 Representing graphs

Graphs are indispensable tools for modeling information. Accordingly, many algorithms on graphs—computing properties of graphs, finding routes in graphs, iterating over the vertices in a graph—are important problems in computer science.

Our first step is representing a graph in a way that can be stored in computer memory. The natural way to "represent" a graph to a human is visually, with a diagram—in fact, that is what we think of a "graph" as being. Clearly that will not do for a computer, however.

To store the information about a graph formally, we would need some way of representing all the vertices on a graph. Since graphs have a lot in common with relations (in fact, a directed graph with no parallel edges is identical to a relation), we might look at our ways of representing relations as a place to start. Consider this graph from Section 8.1:



The straightforward way to interpret the definition of *graph* using set notation would be something like (V, E) where

$$V = \{v_1, v_2, v_3, v_4, v_5, v_6, v_7, v_8, v_9\}$$

$$E = \begin{array}{ll} \{ & e_1(v_1, v_4), e_2(v_4, v_5), e_3(v_1, v_2), e_4(v_2, v_3), e_5(v_2, v_3), \\ & e_6(v_3, v_4), e_7(v_4, v_6), e_8(v_5, v_5), e_9(v_2, v_7), e_{10}(v_6, v_7), \\ & e_{11}(v_7, v_8), e_{12}(v_7, v_9), e_{13}(v_8, v_9), e_{14}(v_6, v_9) \quad \} \end{array}$$

Thus we list our elements of E both by label and by description of what vertices they connect. If our graph had no parallel edges, we could list them only as a pair of edges. Suppose we unify e_4 and e_5.

$$E = \begin{array}{ll} \{ & (v_1, v_4), (v_4, v_5), (v_1, v_2), (v_2, v_3), \\ & (v_3, v_4), (v_4, v_6), (v_5, v_5), (v_2, v_7), (v_6, v_7), \\ & (v_7, v_8), (v_7, v_9), (v_8, v_9), (v_6, v_9) \quad \} \end{array}$$

This, then, is the basis for our first way to represent graphs on a computer: an explicit listing of both the vertices and the edges. Assume a vertex will be labelled

by an int, and a vertex may also have information of some type stored at it.

```
- datatype 'a vertex = V of int * 'a;
```

The type parameter 'a stands for the type of the information, if any, stored at this vertex (the vertex's *payload*). An edge, then, will be represented as two ints indicating its end points and its own payload of type 'b. (The most common payload to store with an edge is a *weight*; intuitively this can be thought of as something like the distance between the two vertices along that edge.)

```
- datatype 'b edge = E of int * int * 'b;
```

Note that we are distinguishing between the vertex objects themselves and the numerical labels for vertices—edges contain a pair of vertex labels, not a pair of vertices. Furthermore, to support parallel edges, we would need also to have labels for edges. We finish this off by defining the graph type itself.

```
- datatype ('a, 'b) graph = G of 'a vertex list * 'b edge list;
```

There are three factors for evaluating representations of graphs. First, how efficiently does the representation use space in memory—how compact is it? Second, how efficiently can information from the graph be read? Third, how easy is it to store information at the vertices and edges?

If we were to use the types defined above, the size of the representation of a particular graph would be proportional to the number of vertices plus the number of edges, and it is difficult to imagine doing better than that. We also have built into it a way to store extra information at both the vertices and edges. As far as the efficiency reading information—say, finding a particular edge—we will have to compare it to other representations.

A common representation is the *adjacency matrix*. In this case, each vertex is represented by a row and column in a matrix, and the entries in the matrix indicate whether there exists an edge between the two matrices—1 for the existence of an edge, 0 for its absence. (For a *directed* graph, we could decide that the row stands for the first vertex in the pair, the column for the second.) For the graph above (with the parallel edge e_5 removed),

adjacency matrix

$$\begin{pmatrix} 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 \end{pmatrix}$$

Looking up information about a graph in adjacency matrix form is very fast. If we want to know if two vertices are adjacent, we use the vertex labels as coordinates in the matrix; if we want to know all the vertices that a certain vertex is adjacent to, we can iterate through that vertex's row (or column).

This representation is less space-efficient, however, since the matrix must have an entry not only for existent vertices but also for absent ones. (Notice that for undirected graphs, the matrix will always be symmetric about the upper-left to lower-right diagonal.) Unless the number of edges is proportional to the square of the number of vertices, this is a serious loss of efficiency, especially for large graphs with sparse edges.

We could store extra information about edges, such as a weight, in the matrix itself instead of the value 1 (the no-edge entries could be any special value indicating the lack of an edge, not necessarily 0). Any extra information for vertices would need to be stored in a separate structure. Note also that in the adjacency matrix representation we have almost no hope of representing parallel edges.

The convenience of using a matrix and the efficiency of the actual lookup depends on the programming language. In programming languages that provide direct support for multi-dimensional arrays in computer memory such as C and Java, this is easy. ML's support for arrays is poor, and for this reason alone we will not be using adjacency matrices for representing graphs.

adjacency llist

An alternative to the adjacency matrix is the the *adjacency llist* representation. In this case, we do not store edges separately, but instead our representation of each vertex contains a list of the vertices (or their labels) to which that vertex is adjacent.

```
- datatype 'a vertex = V of int * 'a * int list;
```

In this datatype, the first int is the vertex's label, the 'a is the payload, and the int list is the labels of the adjacent vertices. (If we want a 'b payload for the edges, we could make the last component to be a (int * 'a) list. If we want parallel edges, we can further augment each entry in the adjacency list with a label for the edge.)

In this representation, edge lookup is less efficient because we must first find a vertex's adjacency list and then iterate through the list rather than jumping to an entry in an adjacency matrix. This is mitigated, however, by ML's being optimized for list processing Moreover, the adjacency list representation is more space efficient because it uses space proportional to the number of edges. In fact, it is similar to our first attempt of storing a list of edges, with the main difference of not requiring a separate structure for the edges. In fact, we can now represent a graph as just a list of vertices:

```
- datatype 'a graph = G of 'a vertex list;
```

Finding the list of adjacent vertices for a given vertex is straightforward:

```
- exception NoSuchVertex;
```

```
- fun getAdjacent(G([]), v) = raise NoSuchVertex
=   | getAdjacent(G(V(u, x, adj)::rest), v) =
=       if u = v then adj else
=            getAdjacent(G(rest), v);
```