1 Graphs

A **graph** is a data structure that expresses relationships between objects. The objects are called "nodes" and the relationships are called "edges".

For example, social networks can be represented as graphs. In the Twitter follower graph, the nodes would be Twitter users, and there would be an edge pointing from me to Justin Bieber if I was following Justin Bieber. (We could write this edge as a 2-tuple: (Jessica, Justin).)

Task dependencies can also be represented as graphs. Here the nodes might be tasks like "check your email," "put on your shoes," or "go to work," and you would draw an edge from "put on your shoes" to "go to work" because you need to put on your shoes before going to work.

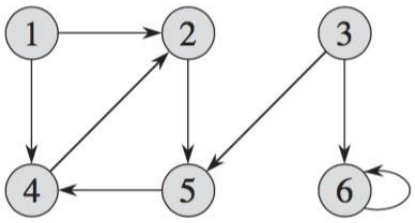

Figure 1: A directed graph.

1.0.1 Directed and undirected graphs

In a **directed graph**, the connection between two nodes is one-directional. The Twitter graph is a directed graph, because I might follow Justin Bieber, but Justin Bieber doesn't follow me.

In an **undirected graph**, all connections are bi-directional. Facebook friendships form an undirected graph, because if I was friends with Justin Bieber, he would also be friends with me. Undirected graphs can be represented as directed graphs, because if (u,v) is an edge in an undirected graph, it would be the same as having a directed graph with the edges (u,v) and (v,u). For this reason, we will mostly focus on directed graphs.

The degree of a node $deg(v)$ is the number of nodes connected to that node. In a directed graph, we have both outdegree (the number of edges pointing away from the node) and indegree (the number of edges pointing towards the node).

A cycle is a path in a graph that goes from a node to itself.

1.0.2 Weighted and unweighted graphs

The nodes and edges in a graph might also carry extra data with them. Often we will want to assign a node a certain color, like "white," "grey," or "black." Edges and nodes may carry numeric values, known as weights.

For example, in the Google Maps graph (where nodes are locations, and edges are the roads connecting the locations), the weight on each edge might be the length of the road.

1.1 Representations of graphs

From here on out, we will often denote graphs as G, and they will have m edges and n nodes. The set of vertices (or nodes) will be denoted V, and the set of edges will be E. This is standard notation but we may use different notation depending on the circumstances.

There are two common ways to represent a graph on a computer: as an adjacency matrix, and as an adjacency list.

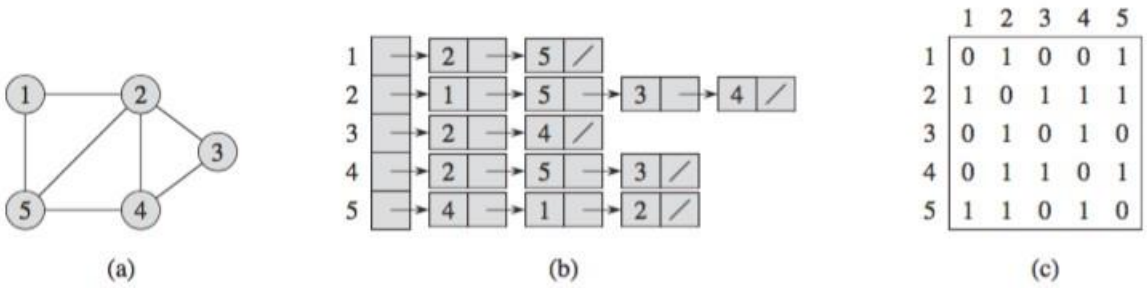

Figure 22.1 Two representations of an undirected graph. **(a)** An undirected graph G with 5 vertices and 7 edges. **(b)** An adjacency-list representation of G. **(c)** The adjacency-matrix representation of G.

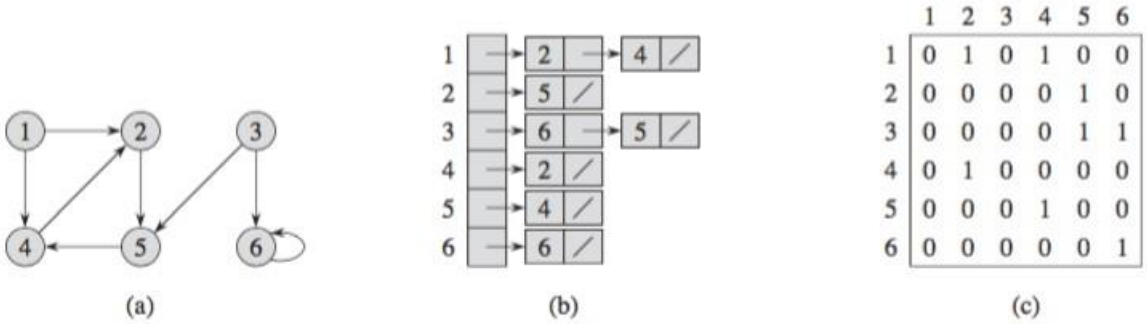

Figure 22.2 Two representations of a directed graph. **(a)** A directed graph G with 6 vertices and 8 edges. **(b)** An adjacency-list representation of G. **(c)** The adjacency-matrix representation of G.

1.1.1 Adjacency matrices

The adjacency matrix A is an n-by-n matrix, where the row and column numbers correspond to the nodes. Entry $A[i,j]$ is 1 if there is an edge from node i to node j, and 0 otherwise.

For weighted graphs, we may replace 1 with the weight on the edge, and replace 0 with a sensible value (like NIL, 0, or ∞, depending on the application).

Note that the adjacency matrix of an undirected graph is symmetric.

Adjacency matrices are easy to implement, but they suffer from a severe drawback. When $m << n^2$, most of the entries are 0, and this wastes a lot of space. For example, in the Twitter follower graph, there are hundreds of millions of users, so if we created a 100M-by100M adjacency matrix, we would have 10^{16} entries. This costs ten thousand terabytes of space, and since most people follow less than a hundred people, most of this space is wasted.

You'd also use a lot of computation time. I might want to get the followers of a certain user, and to do that, I would have to iterate over all 100 million rows just to find 50 followers.

1.1.2 Adjacency lists

The adjacency list Adj is an array of linked lists. Each list contains the neighbors of a vertex. That is, v is in $Adj[u]$ if there is an edge from u to v. (For weighted graphs, we may store the weight $w(u,v)$ along with v in the linked list.)

Observe that the lengths of the linked lists sum to m, because each element in each list corresponds to an edge in the graph. Adj as a whole takes $\Theta(m + n)$ space (because there are n slots in the array, and the lengths of the linked lists sum to m). In many cases, this is much better than the $\Theta(n^2)$ space taken by the adjacency matrix.

On the other hand, testing whether two nodes are neighbors takes potentially $O(m)$ time, because you have to iterate through all the neighbors of a node. In the adjacency matrix, this would take constant time. So which representation to use really depends on your application. But most of the algorithms in the textbook assume that we have an adjacency list, because it is generally better.