12. Graphs

In this chapter, we study two representations of graphs and basic algorithms that use these representations.

Mathematically, a (directed) graph is a pair $G=(V,E)$ where V is a set of vertices and E is a set of ordered pairs of vertices called edges. An edge $(\mathtt{i},\mathtt{j})$ is directed from $\mathtt{i}$ to $\mathtt{j}$; $\mathtt{i}$ is called the source of the edge and $\mathtt{j}$ is called the target. A path in G is a sequence of vertices $v_0,\ldots,v_k$ such that, for every $i\in\{1,\ldots,k\}$, the edge (v_{i-1},v_i) is in E. A path $v_0,\ldots,v_k$ is a cycle if, additionally, the edge (v_k,v_0) is in E. A path (or cycle) is simple if all of its vertices are unique. If there is a path from some vertex v_i to some vertex v_j then we say that v_j is reachable from v_i. An example of a graph is shown in Figure 12.1.

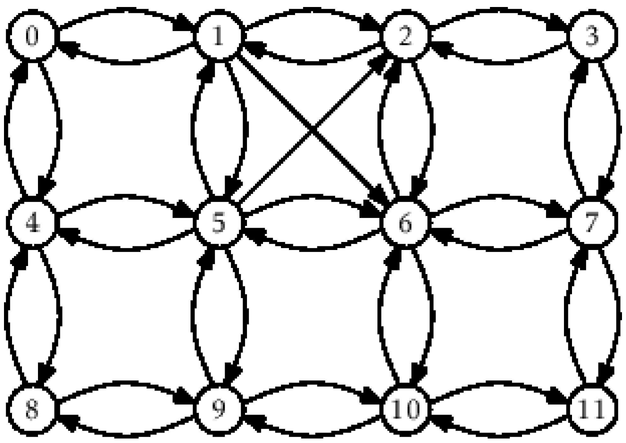

Figure 12.1: A graph with twelve vertices. Vertices are drawn as numbered circles and edges are drawn as pointed curves pointing from source to target.

Due to their ability to model so many phenomena, graphs have an enormous number of applications. There are many obvious examples. Computer networks can be modelled as graphs, with vertices corresponding to computers and edges corresponding to (directed) communication links between those computers. City streets can be modelled as graphs, with vertices representing intersections and edges representing streets joining consecutive intersections.

Less obvious examples occur as soon as we realize that graphs can model any pairwise relationships within a set. For example, in a university setting we might have a timetable conflict graph whose vertices represent courses offered in the university and in which the edge $(\mathtt{i},\mathtt{j})$ is present if and only if there is at least one student that is taking both class $\mathtt{i}$ and class $\mathtt{j}$. Thus, an edge indicates that the exam for class $\mathtt{i}$ should not be scheduled at the same time as the exam for class $\mathtt{j}$.

Throughout this section, we will use $\mathtt{n}$ to denote the number of vertices of G and $\mathtt{m}$ to denote the number of edges of G. That is, $\mathtt{n}=|V|$ and $\mathtt{m}=|E|$. Furthermore, we will assume that $V=\{0,\ldots,\mathtt{n}-1\}$. Any other data that we would like to associate with the elements of V can be stored in an array of length $\mathtt{n}$.

Some typical operations performed on graphs are:

- $\mathtt{addEdge}(\mathtt{i},\mathtt{j})$: Add the edge $(\mathtt{i},\mathtt{j})$ to E.
- $\mathtt{removeEdge}(\mathtt{i},\mathtt{j})$: Remove the edge $(\mathtt{i},\mathtt{j})$ from E.
- $\mathtt{hasEdge}(\mathtt{i},\mathtt{j})$: Check if the edge $(\mathtt{i},\mathtt{j})\in E$
- $\mathtt{outEdges}(\mathtt{i})$: Return a `List` of all integers $\mathtt{j}$ such that $(\mathtt{i},\mathtt{j})\in E$
- $\mathtt{inEdges}(\mathtt{i})$: Return a `List` of all integers $\mathtt{j}$ such that $(\mathtt{j},\mathtt{i})\in E$

Note that these operations are not terribly difficult to implement efficiently. For example, the first three operations can be implemented directly by using a `USet`, so they can be implemented in constant expected time using the hash tables discussed in Chapter 5. The last two operations can be implemented in constant time by storing, for each vertex, a list of its adjacent vertices.

However, different applications of graphs have different performance requirements for these operations and, ideally, we can use the simplest implementation that satisfies all the application's requirements. For this reason, we discuss two broad categories of graph representations.

Subsections