this aspect, there are many ways to do this. Obviously, some graphical representations supply us with more information than other ones. This can be easily seen in Figure 17.2 which contains two different representations of the same graph. Even though the two ways are topologically equivalent, the right one immediately allows us to understand we are dealing with a portion of a two-dimensional square lattice. It would have been even more difficult to realize this, if we had chosen a representation in which the edges intersected. Unfortunately, the latter is often inevitable when trying to "project" a very connected graph on a piece of paper. Finally, we observe that, in the worst possible case, recognizing whether two graphs are the same apart from a permutation of their vertex names (graph isomorphism) is an NP hard problem (see box on page 439). So, in general, it is not easy to immediately know whether there exists a simple graphical representation (for example, one in which only a small number of edges intersect) for a given graph.

Exercise 2 - Drawing graphs

Which complete graphs can you draw on a piece of paper such that none of its edges intersect with each other? If we could use a hologram to represent graphs (i.e., if we could use a three-dimensional space) which complete graphs could we possibly represent without any intersecting edges?

17.2 From graphs to data structures

To represent graph related information in a computer program, it would be enough to write the set V of vertices and the set E of edges in the computer memory. Without loss of generality, we can assume that for a graph of N vertices, the set V is represented by the integer numbers between 0 and $N-1$. Nevertheless, we advise against representing the set E with the mathematical notation used in equation (17.1). Indeed, simply writing the vertex couples corresponding to the graph edges in an array complicates the operations we typically want to perform on graphs. For example, to locate the neighbors of a given vertex we would have to scroll through the entire list.

There basically exist two efficient data structures to represent the set E of edges of a graph. The first one uses an *adjacency matrix* and the second

is based on *adjacency lists*. In the following sections, we study the merits and flaws of these two data structures to represent a graph in a computer. We also discuss which one is more convenient depending on the problem under study.

17.2.1 *Adjacency matrix*

Given a graph with N vertices, the adjacency matrix A is an $N \times N$ matrix such that:

$$A_{ij} = \begin{cases} 1 \text{ if the edge } (i,j) \in E, \\ 0 \text{ otherwise.} \end{cases}$$

On the left side of Figure 17.3 an example of an undirected graph and its corresponding adjacency matrix is given. The adjacency matrix of an undirected graph is always symmetric. It is natural to store the elements of A in a two-dimensional array `A[][]`.

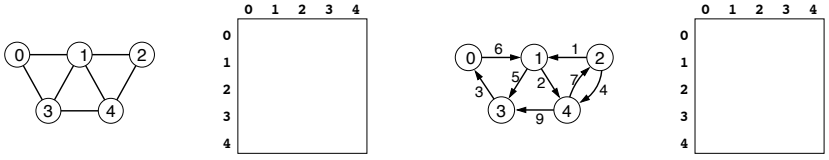


Fig. 17.3 An undirected and unweighted graph (on the left) and a directed and weighted graph (on the right), each with their corresponding adjacency matrix.

The main merit of the adjacency matrix is that it is very easy to write a program using it. For instance, to find out whether two vertices i and j are connected, we just have to read the content of the element `A[i][j]`. Instead, its main flaw is the memory needed to store such a matrix, which is always N^2, independently of the average degree of the graph. Moreover, to determine the neighbors of site i we necessarily need to examin all elements on the ith row. This is why we strongly advise against using it for sparse graphs. Indeed, the matrix would be full of zeros, thus uselessly occupying memory and making it unhandy to work. For these kinds of graphs it is preferable to use adjacency lists (Section 17.2.2).

The adjacency matrix is convenient when working on dense graphs. In this case, the nonzero matrix elements are a reasonable fraction of the total and the size of the matrix, N^2, has the same order of magnitude as the set of edges. So, we are not truly "wasting" memory. Moreover, in case the dense graph is also weighted, the adjacency matrix easily allows to include

this information on the weights: the matrix element A_{ij} is set equal to the weight of the link (i, j) or zero[1] if such a link does not exist. In case of directed graphs, the adjacency matrix allows to easily distinguish between the edges *leaving* the vertex i (those on the ith row) from those *arriving* at vertex i (those on the ith column). On the right side of Figure 17.3 we included an example of a weighted, directed graph, together with its adjacency matrix representation.

17.2.2 *Adjacency lists*

In order to obtain the adjacency list representing a graph, we build for each of its vertices a list with its neighbors. Each list has a length equal to the corresponding vertex degree. In general, the order in which the neighbors of a given vertex are included in the list is unimportant. So, for each of the N lists we can use the easiest data structure, for example, the bucket described in Section 16.1. Figure 17.4 contains an example of a graph and its representation in terms of adjacency lists: `neigh[]` is an array of pointers (Section 6.5.1) with each element pointing to a list of neighbors. It is convenient to also keep an array `degree[]` with the degrees of the vertices, such that the element `degree[i]` indicates how many neighbors we can read starting from the memory location pointed by `neigh[i]`.

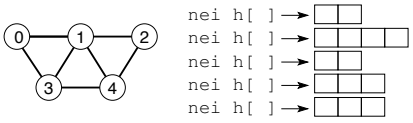


Fig. 17.4 The same undirected and unweighted graph of Figure 17.3 and its representation in terms of adjacency lists.

In total, this graph representation takes $2(N + M)$ memory locations. More precisely, a vector of length N for the degrees, another of the same length for the pointers to the lists and N vectors for the adjacency lists, occupying in total $2M$ memory locations, as for each edge we include two vertices in the adjacency lists. As we already mentioned, the adjacency matrix is typically less efficient to use compared to the adjacency lists. This is why we generally advise to use the latter. In case of a weighted graph, the adiacency lists should also keep hold of the edge weights. To

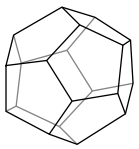
[1]The conventional value of the weight we choose to indicate the absence of a link can be substituted by another one in case 0 is considered a valid weight.

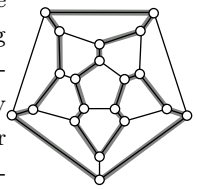
this purpose, we can simply transform each element of the adjacency lists in a `struct` containing both the vertex nearest neighbor as the corresponding edge weight.

Hands on 1 - Adjacency lists

Write a program reading a graph from a file and creating the corresponding adjacency lists. The formatted file contains, on the first line, the number of vertices, and on the following lines, a couple of integer numbers per line corresponding to the vertices joined by an edge. Print the histogram of the vertices degrees.

Hamiltonian cycles





In 1858, the Irish mathematician and physicist Sir William Rowan Hamilton (1805-1865) presented the following problem: given a dodecahedron, is it possible, starting from a given vertex and moving along its edges, to visit all vertices exactly once before returning to the starting vertex? In terms of the equivalent graph, the question is whether there exists a simple cycle passing through all vertices of the graph. In case of the dodecahedron the answer can be found by means of simply trying (a possible solution is included in the figure). For a generic graph it is much harder to answer this question. Only in the firs half of the seventies, it was discovered that finding Hamiltonian cycles is an NP-complete problem, and thus, very difficult to solve in a worst case scenario.

17.3 Graph algorithms

Many interesting problems can be transformed into problems on graphs. For example, we already considered the traveling salesman problem, which