3
Graphs

3.1 Extending the Relationships between Records

In a generalization of the linear lists of the previous chapter, we may allow the connection between any two elements, without insisting on some specific order. The resulting structure is known as a *graph*. This models quite naturally several real life situations, like networks of communicating computers, or road systems connecting the various locations of some neighborhood. We shall, however, see that a graph may be a useful tool for many other applications, some of which seeming a priori completely unconnected to a graph structure.

Mathematically, a graph G is defined as a pair of sets $G = (V, E)$. There is no restriction on the set V, called the *vertices* of the graph, and its elements are usually denoted $V = \{v_1, v_2, \ldots, v_n\}$, or simply $V = \{1, 2, \ldots, n\}$. The set of *edges* E satisfies $E \subseteq V \times V$ and actually describes whether some binary relation exists between certain pairs of vertices. We may have $E = \emptyset$ or $E = V \times V$, in which cases the graph is called *empty* or *full*, respectively. Note that this refers only to the number of edges, so a graph may be empty and yet have many vertices. The complexities of algorithms involving graphs are often given as a function of the sizes of V and E, for example $O(|V| + |E|)$, but it has become common practice to simplify this notation to $O(V + E)$ when no confusion can arise.

It is customary, and often very helpful as a visual aid, to represent a graph by a drawing in which the vertices appear as dots or circles, and an edge (a, b) as an arrow from the circle corresponding to a to that of b. Figure 3.1 is such a drawing for a graph with $V = \{1, 2, \ldots, 13, 14\}$ and $E = \{(1, 2), (1, 3), (1, 6), (2, 2), \ldots, (10, 11), (12, 13)\}$, and will serve as a running example in this chapter. It should be emphasized that this drawing is *not* the graph G itself, but rather one of its infinitely many possible *representations*. Indeed, there are almost no restrictions on the layout of the drawing: the circles

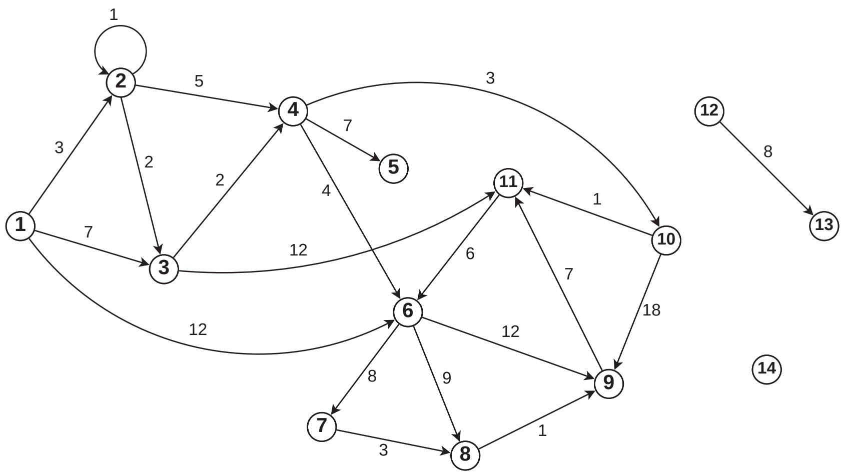

Figure 3.1. An example graph $G = (V, E)$ with $V = \{1, 2, \dots, 13, 14\}$ and $E = \{(1, 2), (1, 3), (1, 6), (2, 2), (2, 3), (2, 4), (3, 4), (3, 11), (4, 5), (4, 6), (4, 10), (6, 7), (6, 8), (6, 9), (7, 8), (8, 9), (9, 11), (10, 9), (10, 11), (11, 6), (12, 13)\}$.

representing the vertices are generally scattered in any visually appealing way, the edges may be straight lines or arcs or have any other form, they may cross each other, etc. The formalism of not identifying a graph with its representation is thus necessary, because one can easily get completely different drawings, yet representing the same graph.

A graph in which we distinguish between edges (a, b) and (b, a) is called *directed*, and its edges are usually drawn as arrows, like in Figure 3.1. In other applications, only the fact whether there is a connection between the elements a and b is relevant, but the direction of the edge (a, b) or (b, a) is not given any importance; such graphs are called *undirected* and their edges are shown as lines rather than arrows, as in Figure 3.2. Even though in a strictly mathematical sense, we should then denote edges as sets $\{a, b\}$ instead of ordered pairs (a, b), it is common practice to use the pair notation (a, b) even for the undirected case.

An edge of the form (a, a), like the one emanating from and pointing to vertex 2, is called a *loop*. A sequence of edges

$$(v_a, v_b), (v_b, v_c), (v_c, v_d), \dots, (v_w, v_x),$$

in which the starting point of one edge is the endpoint of the preceding edge, is called a *path* from v_a to v_x. For example, $(2, 4), (4, 6), (6, 9), (9, 11)$ is a path from 2 to 11. If $v_a = v_x$, that is, if the last edge of a path points to the beginning of the first edge, the path is called a *cycle*. For example, $(8, 9), (9, 11), (11, 6), (6, 8)$ is a cycle. If there is a path from every vertex to

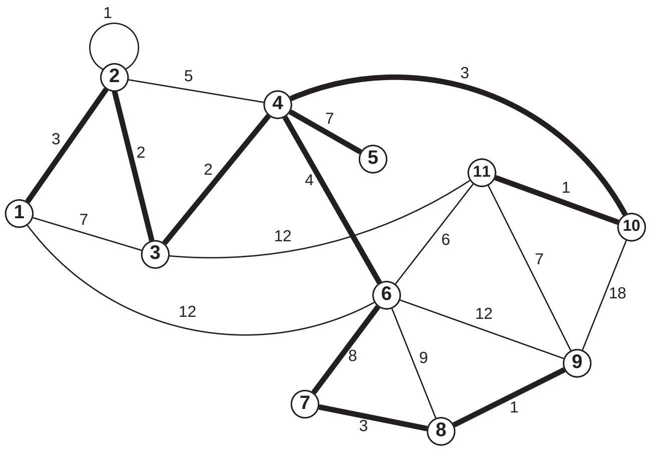

Figure 3.2. A minimum spanning tree.

any other, the graph is *connected*. Our example graph is not connected, because, e.g., vertex 13 cannot be reached from vertex 4.

The number of outgoing edges from a given vertex is called its *out-degree* and the number of incoming edges is called its *in-degree*. In undirected graphs, there is no such distinction and one simply defines the *degree* of a vertex v as the number of edges touching v. Vertex 9 has in-degree 3 and out-degree 1, and vertex 4 in Figure 3.2 has degree 5. A vertex with in-degree 0, like vertex 1, is a *source* and a vertex with out-degree 0, like vertex 5, is a *sink*, and if there is no edge touching the vertex at all, like for vertex 14, it is *isolated*.

An important special case is a *tree*, defined as a graph that is connected but has no cycles. Our graph is not a tree, because it is not connected, but even if we restrict the intention to the subgraph G' induced by the vertices $V' = \{1, 2, \ldots, 11\}$, that is, ignoring the vertices in $V - V'$ and the edges touching them, as in Figure 3.2, the graph would still not be a tree, because it contains cycles. On the other hand, the bold edges in Figure 3.2 form a tree.

Regretfully, there is another, quite different, notion in Computer Science that is also called a *tree*, as we shall see already in the following chapter. There are certain similarities between a tree as a data structure and a tree as a special kind of a graph, but there are many differences and calling both by the same name is a source of confusion. More details will be given in Chapter 4.

In many applications, it is convenient to assign a *weight* $w(a, b)$ to each edge (a, b) of a graph, as we did in Figure 3.1. The weight of a path will be defined as the sum of the weights of the edges forming it. There are many possible interpretations for these weights, and a few examples will be given in what follows.

There are entire courses devoted to the study of graph related algorithms. They are beyond the scope of this book, and we mention here only a few of the well-known problems, without presenting possible solutions. What is relevant to our topic of data structures is the way to transform a problem we might be dealing with into a related one that can be solved by means of a known graph algorithm. Such a transformation is known as a *reduction*, which is an essential tool in the study and development of algorithms.

3.1.1 The Minimum Spanning Tree Problem

Suppose the vertices of the undirected graph $G = (V, E)$ represent locations and that we wish to construct an electricity network reaching each vertex $v \in V$. The weight $w(a, b)$ may stand for the cost of building a high voltage transmission line from location a to location b. For technical or topographical reasons, there may be vertices that are not connected by an edge, so the graph is not necessarily full. The problem is to find the cheapest layout of the network of transmission lines. Alternatively, one could think of the vertices as computers in a network, of the edges as wired communication lines and of the weights as the costs to build the lines.

There must be a possibility to transfer electricity, or allow communication, from every point to each other, therefore the graph G has to be connected. On the other hand, we assume that there is no reason to build a network with more than one possibility to get from a vertex a to a vertex b. This implies that the sought network should not include any cycle, so what we are looking for is in fact a tree. A tree touching all the vertices in V is called a *spanning tree*. The challenge is that there might be many such trees.

The problem can thus be reformulated mathematically as: given an undirected connected graph $G = (V, E)$ with weights $w(a, b)$ for each $(a, b) \in E$, find a subset $T \subseteq E$ of the edges such that T is a spanning tree, and such that $\sum_{(a,b)\in T} w(a, b)$ is minimized over all possible choices of such trees T. Figure 3.2 is an undirected version of the graph of Figure 3.1, restricted to the vertices $V' = \{1, 2, \ldots, 11\}$, in which the edges of one of the possible *Minimum Spanning Trees*, with weight 34, have been boldfaced.

3.1.2 The Shortest Path Problem

Returning to the directed version of the graph in Figure 3.1, imagine that the vertices are locations and the edges represent roads connecting certain pairs of them. The weight of an edge (a, b) could be the distance between its endpoints, or the cost of getting from a to b, or the time it takes to do it, etc. Accordingly,

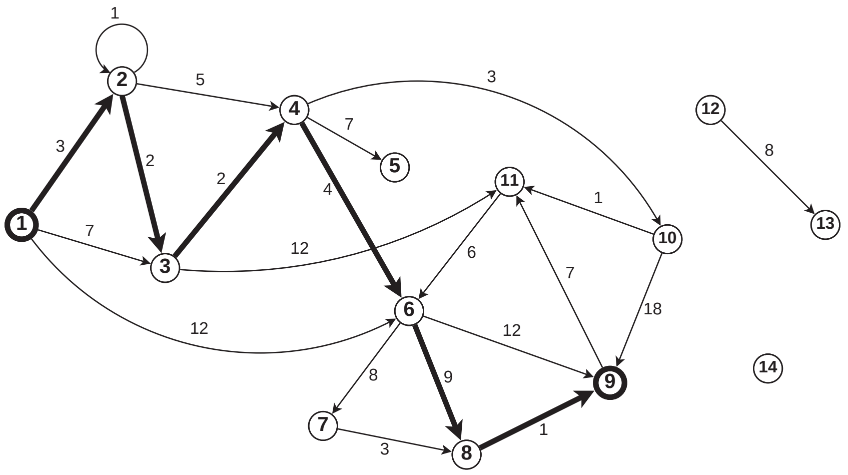

Figure 3.3. A shortest path from 1 to 9.

the problem to deal with is to find the shortest, or cheapest, or fastest path from a given point to another.

These seem at first sight as if they were different problems, but they are all equivalent to the following formulation: given a directed graph $G = (V, E)$ with weights $w(a, b)$ for each $(a, b) \in E$, and given a source vertex s and a target vertex t, find a path from s to t such that its weight is minimized over all the (possibly many) paths from s to t.

Figure 3.3 shows, in boldface, one of the possible *Shortest paths*, with weight 21, from source vertex 1 to target vertex 9.

3.1.3 The Maximum Flow Problem

In this application, the edges are considered as highways and their meeting points, the vertices, are interchanges. We assume that there is some measure to describe numerically the traffic load on a given highway at a certain moment. The weight $w(a, b)$ is the maximal possible traffic capacity on the highway from a to b. The goal is to measure the maximal possible traffic flow from a source interchange s to a target point t.

Alternatively, the edges could be pipelines and the weight $w(a, b)$ would represent the maximal quantity of some liquid that can traverse the edge (a, b) in some given time interval. The problem is then to evaluate how much of the liquid can be shipped from s to t.

Figure 3.4 shows one of the possible *Maximal Flows*, with total weight 16, from source vertex 1 to target vertex 9. Each edge is annotated with a pair a/c,

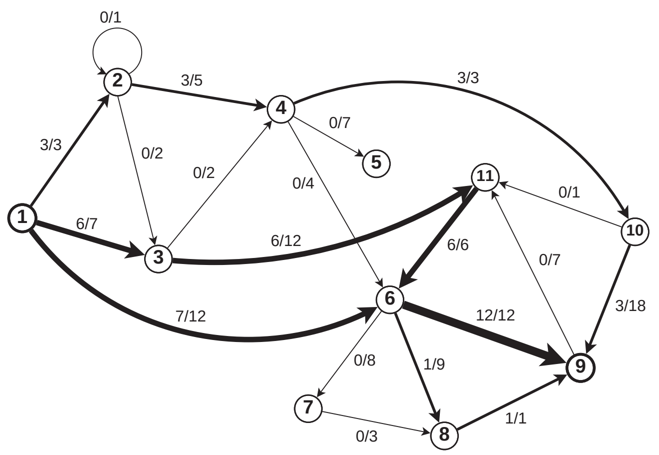

Figure 3.4. A maximal flow network.

with $a \leq c$, where c is the capacity of the edge, and a is the actual value of the flow assigned to it. Note that for certain edges $a = c$, meaning that their capacity is fully exploited, like on edge (11,6); for others, $a = 0$, that is, the edges are not used at all, like edge (3,4); finally, for certain edges, their capacity is only partially used, like edge (10,9), using only 3 of the possible 18. Note also that at each interchange, the incoming and outgoing flows must be the same, but that the whole quantity may be reshuffled. For example, the three incoming edges on vertex 6 contribute $7 + 0 + 6$, which are redistributed into the three outgoing edges as $0 + 1 + 12$.

3.2 Graph Representations

One way to represent a graph $G = (V, E)$ in a computer program is to define a two dimensional matrix $A = (a_{ij})$, with $1 \leq i, j \leq |V|$, called an *adjacency matrix*, and setting

$$a_{ij} = \begin{cases} 1 & \text{if } (i, j) \in E \\ 0 & \text{otherwise,} \end{cases}$$

as can be seen in the upper part of Figure 3.5, corresponding to this chapter's running example. Outgoing edges from vertex i can then easily be checked by inspecting the ith row, and incoming edges on vertex j are found in the jth column. Instead of using a Boolean matrix that only shows whether a certain edge exists, one may define a general matrix for a weighted graph by setting $a_{ij} = w(i, j)$ if $(i, j) \in E$, but care has to be taken to reserve a special

	1	2	3	4	5	6	7	8	9	10	11	12	13	14
1	0	**1**	**1**	0	0	**1**	0	0	0	0	0	0	0	0
2	0	**1**	**1**	0	**1**	0	0	0	0	0	0	0	0	0
3	0	0	0	**1**	0	0	0	0	0	0	**1**	0	0	0
4	0	0	0	0	**1**	**1**	0	0	0	**1**	0	0	0	0
5	0	0	0	0	0	0	0	0	0	0	0	0	0	0
6	0	0	0	0	0	0	**1**	**1**	**1**	0	0	0	0	0
7	0	0	0	0	0	0	0	**1**	0	0	0	0	0	0
8	0	0	0	0	0	0	0	0	**1**	0	0	0	0	0
9	0	0	0	0	0	0	0	0	0	0	**1**	0	0	0
10	0	0	0	0	0	0	0	0	**1**	0	**1**	0	0	0
11	0	0	0	0	0	**1**	0	0	0	0	0	0	0	0
12	0	0	0	0	0	0	0	0	0	0	0	0	**1**	0
13	0	0	0	0	0	0	0	0	0	0	0	0	0	0
14	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Matrix

vertex	neighbors	weights
1	2,3,6	3,7,12
2	2,3,4	1,2,5
3	4,11	2,12
4	5,6,10	7,4,3
5		
6	7,8,9	8,9,12
7	8	3
8	9	1
9	11	7
10	9,11	18,1
11	6	6
12	13	8
13		
14		

List of neighbors

Figure 3.5. Graph representations.

value, which cannot be confused with a weight, to indicate the absence of an edge.

The matrix representation may be wasteful, because algorithms requiring a full scan of the graph will often imply the processing of all the V^2 elements of the matrix, even if the number of edges is significantly lower. This leads to the idea of representing the graph by an array of linked lists, as shown in the lower part of Figure 3.5. The array is indexed by the vertices, and the ith element of the array points to the list of the neighbors of i, which are in fact the endpoints of the outgoing edges from i. Another linked list may be added giving the weights on the edges in the same order, if necessary. For a full scan of the

```
BFS($G, v$)                                  DFS($G, v$)
    mark $v$                                     mark $v$
    $Q \Longleftarrow v$                         $S \Longleftarrow v$
    while $Q \neq \emptyset$ do                  while $S \neq \emptyset$ do
        $w \Longleftarrow Q$                         $w \Longleftarrow S$
        visit $w$                                    visit $w$
        for all $x$ such that $(w, x) \in E$         for all $x$ such that $(w, x) \in E$
            if $x$ is not marked then                    if $x$ is not marked then
                mark $x$                                     mark $x$
                $Q \Longleftarrow x$                         $S \Longleftarrow x$
```

Figure 3.6. BFS and DFS algorithms for a graph G, starting at vertex v.

graph, all these linked lists will be processed, but even for an undirected graph, no edge appears more than twice in the union of the lists, so the algorithms may be bounded by $O(E)$, which may be lower than $O(V^2)$.

3.3 Graph Exploration

One of the motivations for passing from linear lists to the more general graphs was to overcome the dependency on some order among the vertices, which is often imposed artificially and cannot be justified. Nevertheless, even in applications for which a graph representation is most natural, the need often arises to scan the vertices in some systematic way, processing each of them exactly once.

The most popular scan orders are known as *Breadth first search* (BFS) and *Depth first search* (DFS). This is most easily understood for a tree: in BFS, starting at some chosen vertex v, the vertices are explored by layers, first the direct neighbors of v, then the neighbors of the neighbors, etc.; in DFS, the order is by branches, fully exploring one branch before starting another. However, both algorithms can be applied to general graphs, not only to trees.

The formal algorithms for BFS and DFS are given in Figure 3.6. The parameters are a graph G and a chosen vertex v from which the search emanates. A function mark is used to avoid the processing of a vertex more than once. The algorithms are in a generic form, concentrating on the order in which the vertices are scanned, rather than specifying what exactly to do with them. The actions to be performed with any given vertex v may differ according to the application at hand, and they are globally referred to here as "visiting" v.

The algorithms appear side by side to show their similarity. Indeed, the only difference is the data structure in which newly encountered vertices are temporarily stored: BFS uses a queue Q, whose FIFO paradigm assures that the vertices are processed by layers, whereas DFS stores the elements in a stack S. Vertices are marked before being inserted into Q or S, and are visited immediately after being extracted. The sequences produced by visiting our example restricted graph G' of Figure 3.1, starting at vertex 1, are

$$\text{BFS}(G', 1)\text{: } 1, 2, 3, 6, 4, 11, 7, 8, 9, 5, 10$$
$$\text{DFS}(G', 1)\text{: } 1, 2, 4, 5, 10, 9, 11, 3, 6, 7, 8.$$

It should be noted that the order of visiting the vertices is not necessarily unique, neither for BFS nor for DFS, and depends on the order in which the neighbors of a given vertex are inserted into Q or S. In the preceding example, this insertion order has been chosen by increasing indices for BFS, and by decreasing indices for BFS, for example, the neighbors of vertex 4 have been inserted as 5,6,10 for BFS and as 10,6,5 for DFS.

The complexity of both algorithms is $O(V + E)$, because every edge is processed once. There are applications for which BFS is preferable, like finding a shortest path as in Section 3.1.2 when all weights are equal. For other applications, like dealing with a maze, DFS might be better. In many cases, however, the exact order does not matter and all we want to assure is that no vertex is skipped or processed twice, so BFS or BFS can be applied indifferently.

3.4 The Usefulness of Graphs

This section presents several examples of problems, chosen from different domains of Computer Science, that can be solved by means of a transformation into related problems involving graphs.

3.4.1 Example: Compressing a Set of Correlated Bitmaps

A *bitmap* or bit-vector is a simple data structure often used to represent a set, as we shall see later in Chapter 8. For example,

$$A = \mathsf{000100110000000010000000}$$

is a bitmap of length $\ell = 24$ bits, of which $m = 4$ are ones and 20 are zeros. A possible application could be to an *Information Retrieval* system, in which such bitmaps may replace the index by acting as occurrence maps for the terms in the different documents. The length ℓ of the maps will be the number of documents in the system, and there will be a bitmap B_x for each different term x. The ith bit of B_x will be set to 1 if and only if the term x appears in the document indexed i.

In the example bitmap A, the corresponding term appears in documents 4, 7, 8, and 17.

Most of the terms occur only in very few documents, thus in this application, as well as in many others, most of the bitmaps used are sparse, with an overwhelming majority of zeros. The maps can therefore be compressed efficiently, and we shall assume that the size of a compressed form of a bitmap A is an increasing function of $N(A)$, which denotes the number of 1-bits in A. A simple way to achieve this is by encoding the indices of the 1-bits, using $\lceil \log_2 \ell \rceil$ bits for each. If indeed the bitmap is sparse, its compressed size $m\lceil \log_2 \ell \rceil$ will be smaller than the original length ℓ.

Suppose now that we are given a set of n bitmaps $\mathcal{B} = \{B_1, \ldots, B_n\}$, all of the same length ℓ. One may compress this set by processing each bitmap individually. But certain bitmaps may be correlated and include similar bit-patterns, simply because the corresponding terms appear mostly together. To continue the example, consider a second bitmap

$$B = \mathsf{0100001100000000010000000}.$$

Instead of compressing A and B, let us exploit their similarity to produce a new map C by recording only the differences between A and B. This can be done by defining $C = A \text{ XOR } B$:

$$C = \mathsf{0101000000000000000000000}.$$

The bitmap C will be sparser than both A or B, and thereby more compressible. We may thus keep A and C instead of A and B. No information is lost, because the bitmap B can be recovered by simply XORing again: $B = A \text{ XOR } C$.

Generalizing this idea to n bitmaps seems to be a difficult task. How can one decide which bitmaps should be XORed? There might be chains, like $A \leftarrow B \leftarrow C \leftarrow \cdots$, meaning that one should keep A on its own, and B XOR A, and C XOR B, etc. These chains should not contain any cycles otherwise the decoding would not be possible. The problem is thus to find a partition of the set of bitmaps $\mathcal{B}$ into two disjoint subsets: $\mathcal{B}_i$, the bitmaps that will be compressed individually, and $\mathcal{B}_x$, those that will be XORed before being compressed. In addition, we seek a function f from $\mathcal{B}_x$ to $\mathcal{B}$, assigning to each bitmap $B \in \mathcal{B}_x$ a bitmap $B' \in \mathcal{B}_i \cup \mathcal{B}_x$ with which it should be XORed. The additional constraints are:

(i) for all bitmaps $A \in \mathcal{B}_x$, the sequence $A, f(A), f(f(A)), \ldots$ should not contain any cycle, as long as it is defined;

(ii) the total number of 1-bits in the set of compressed bitmaps,

$$\sum_{B \in \mathcal{B}_i} N(B) + \sum_{B \in \mathcal{B}_x} N(B \text{ XOR } f(B)),$$

should be minimized over all possible partitions and choices of f.

An exhaustive search over all possible partitions and functions f would have a complexity that is exponential in n, which is clearly not feasible for larger sets of bitmaps. Here is a faster solution.

Recall that given two equal length bitmaps A and B, their *Hamming distance*, denoted by $HD(A, B)$, is the number of indices at which the bitmaps differ, so that

$$HD(A, B) = N(A \text{ XOR } B).$$

In particular, the number of 1-bits in a bitmap A can be expressed in terms of the Hamming distance as

$$N(A) = HD(A, Z),$$

where Z is defined as a bitmap consisting only of zeros.

Define an undirected weighted graph $G = (V, E)$, with

$$V = \mathcal{B} \cup Z, \quad E = V \times V - \{(a, a) \mid a \in V\}, \quad w(a, b) = HD(a, b).$$

In other words, the vertices are the given bitmaps, to which the 0-bitmap Z has been adjoined. The edges are those of a full graph without loops, and the weight on edge (a, b) is the number of 1-bits in a XOR b. We are looking for a subset $T \subseteq E$ of the edges that does not contain any cycles, touches all vertices while minimizing the sum of weights on the edges in T. This is a *minimum spanning tree* of G.

The direct neighbors of Z in T are those bitmaps that are best compressed individually, and for any other bitmap A, its best choice $f(A)$ will be the successor of A in the unique directed path from A to Z in T. For example, refer to the minimum spanning tree of Figure 3.2 as if it were the result of this procedure, and to vertex 6 as the one representing the bitmap Z. This induces a partition into two trees, as shown in Figure 3.7. We get $\mathcal{B}_i = \{4, 7\}$ and the directed edges are $(A, f(A))$, i.e., each other bitmap should be compressed by XORing it first with the bitmap it points to in Figure 3.7.

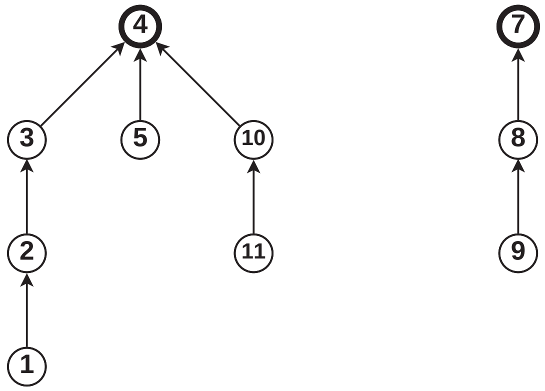

Figure 3.7. Optimal layout for compressing bitmaps.

3.4.2 Example: Optimal Parsing of a Text by means of a Dictionary

Suppose we are given a text $T = x_1x_2 \cdots x_n$ of length n characters, which all belong to some alphabet Σ. For example, T could be the text aaabccbaaaa of length 11 and $\Sigma = \{$a, b, c$\}$. To encode the text, we assume the existence of a *dictionary* D. Unlike standard collections like the Oxford English Dictionary, the elements of D are not restricted to be words in some language, and can also be word fragments, of phrases, or in fact any string of characters of Σ. We shall use as an example

$$D = \{\text{a, b, c, aa, aaaa, ab, baa, bccb, bccba}\}.$$

The goal is to encode the text T by means of the dictionary D, replacing substrings of T by (shorter) pointers to elements of D, if indeed the replaced substring is one of the elements of D. We do not deal here with the (difficult) problem of building such a dictionary according to some optimization rules, and assume that D is given as a part of the description of the problem. Formally, we wish to represent the text as a sequence $T = w_1w_2 \cdots w_k$, with $w_i \in D$, and such that the number k of elements in this sequence is minimized. The decomposition of a text into a sequence of dictionary elements is called a *parsing*.

The problem is that because the elements of D might be overlapping, the number of different possible parsings can be too large to allow an exhaustive search over all possibilities. Simple solutions do not yield an optimal parsing. For example, a greedy approach, trying to match the longest possible element of D in a left to right scan would yield

greedy: aa – ab – c – c – baa – aa,

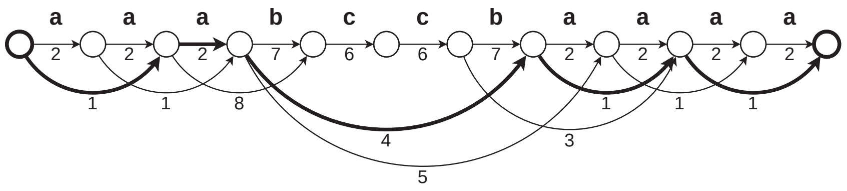

Figure 3.8. Optimal parsing of a text by means of a dictionary.

a partition into 6 elements. A better solution would be to try to locate the longest possible fragment w, and continue this heuristic recursively on the remaining text preceding w and on that following w. This gives

longest fragment: aa – a – bccba – aa – a,

a partition of only 5 elements, which is still not optimal, since one can get a sequence of only 4 elements as in

minimal number of elements: aa – a – bccb – aaaa.

However, the number of elements is only relevant if one assumes that the pointers to the dictionary will all be of the same length, which could be $\lceil \log_2 |D| \rceil$, or 4 bits in our example. If D is encoded by some *variable length code* (see Chapter 11), even if it is given and fixed in advance, the problem is even more involved.

Suppose then that in addition to T and D, there is also a given encoding function $\lambda(w)$ defined for all $w \in D$, producing binary strings of variable length. The length (in bits) of the string encoding w is denoted by $|\lambda(w)|$. The lengths of the elements in our example could be, in order, 2,7,6,1,8,8,3,4,5. The generalized problem is then to find a parsing $T = w_1 w_2 \cdots w_k$ as earlier, minimizing the total length $\sum_{i=1}^{k} |\lambda(w_i)|$ of the encoded text. For the given lengths, the preceding three solution would yield encodings of length 25, 11, and 15 bits, respectively, none of which is optimal.

To build the optimal solution, define a weighted directed graph $G = (V, E)$. The vertices $V = \{1, 2, \ldots, n, n+1\}$ correspond, in sequence, to the characters of the string, plus an additional vertex numbered $n+1$. The edges correspond to substrings of the text that are elements of the dictionary D, setting, for all $1 \le i < j \le n+1$:

$$(i, j) \in E \qquad \text{if and only if} \qquad x_i x_{i+1} \cdots x_{j-1} \in D.$$

The weight on edge (i, j) is defined as $|\lambda(x_i x_{i+1} \cdots x_{j-1})|$.

Figure 3.8 displays the graph corresponding to our running example. The sought solution is the weight of the *shortest path* from vertex 1 to vertex $n+1$. The edges of such a path have been emphasized in the figure, and the weight of this path, which is the length in bits of the optimal parsing of the text, is 9.

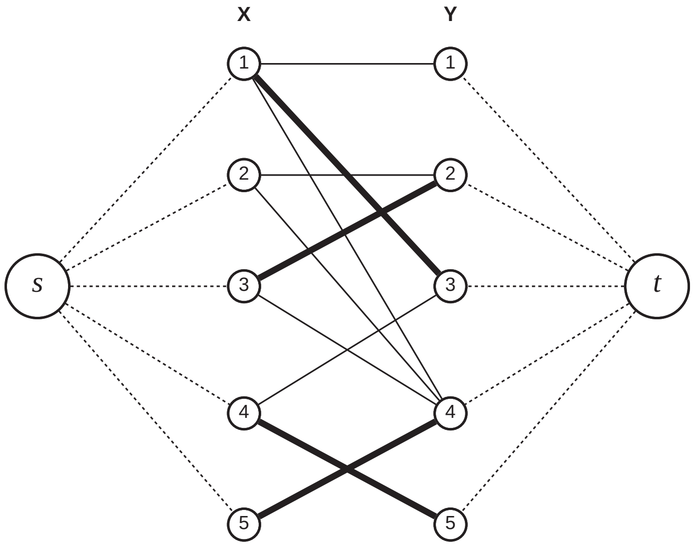

Figure 3.9. Optimal matching.

3.4.3 Example: Generating an Optimal Set of Matches

The solution of the following problem is essential for the survival of the human race. Given are two sets of the same size n, one of girls, $X = \{x_1, x_2, \ldots, x_n\}$, and one of boys, $Y = \{y_1, y_2, \ldots, y_n\}$. We further assume the existence of a collection $\mathcal{C}$ of potential matches (x_i, y_j), that is, $\mathcal{C} \subseteq X \times Y$, with $(x_i, y_j) \in \mathcal{C}$ expressing the fact that x_i would be willing to marry y_j and vice versa, with complete symmetry between the sexes. So unlike the more tragic situation that gave rise to much of the romantic literature, we assume here an idyllic world without unrequited love, in which x wants to marry y if and only if y wants to marry x. Moreover, the set of potential mates of x, $\{y_j \mid (x, y_j) \in \mathcal{C}\}$, as well as the set of potential mates of y, $\{x_i \mid (x_i, y) \in \mathcal{C}\}$, are not ordered, so there are no preferences, and each of the possibly many choices are equally acceptable for all involved parties.

For example, set $n = 5$ and $\mathcal{C} = \{(x_1, y_1), (x_1, y_4), (x_2, y_2), (x_2, y_4), (x_3, y_2), (x_3, y_4), (x_4, y_3), (x_4, y_5), (x_5, y_4)\}$. The goal is to produce as many marriages as possible. This is a problem, because there is still an additional constraint of monogamy, allowing any girl or boy to have at most one spouse.

The solution is the construction of a weighted graph $G = (V, E)$, with

$$V = X \cup Y \cup \{s, t\} \qquad E = \mathcal{C} \cup \{(s, x) \mid x \in X\} \cup \{(y, t) \mid y \in Y\},$$

and the weights on all the edges being set to 1, as shown in Figure 3.9.

Consider the edges as oriented from left to right in the figure, and interpret the graph as a flow network with equal capacity on each edge. The maximal matching is then the set of edges having nonzero flow in a *maximum flow* from s to t. For the given example, only four couples can be formed, one of the several possible solutions corresponds to the boldfaced edges, while x_2 and y_1 will stay singles.

Exercises

3.1 One of the algorithms solving the shortest path problem of Section 3.1.2 is due to *Edsger W. Dijkstra*. Its correctness, however, depends on the additional assumption that all the weights $w(a, b)$ are nonnegative. This might sound reasonable in many applications, but there are others in which negative weights should also be supported. A natural suggestion is the following reduction.

Given a graph $G = (V, E)$ with weights $w_G(a, b)$, some of which are negative, let $m = \min\{w_G(a, b) \mid (a, b) \in E\}$ be the smallest weight in the graph. By assumption, m is negative. Define a graph $G' = (V, E)$ with the same vertices and edges as G, only with different weights, by setting, for all $(a, b) \in E$,

$$w_{G'}(a, b) = w_G(a, b) - m.$$

All the weights in G' are thus nonnegative, so we can apply Dijkstra's algorithm and deduce the shortest paths also for G.

Show that this reduction does not give a correct shortest path, that is, give an example of a graph G and a shortest path in G which does not correspond to a shortest path in G'. Note that there is no need to know the details of Dijkstra's algorithm to answer this and any of the subsequent questions.

3.2 Suppose you are given a complete list of all the airports, and all the flights and their exact schedules. For simplicity, assume that the schedule is periodic, repeating itself every day. You may use the algorithm A which finds the shortest path in a weighted graph, but you cannot change A. What can be done to find:

(a) a cheapest path from a source s to a target t, under the constraint that airport T should be avoided;
(b) a cheapest path from s to t, but preferring, whenever possible, the use of airline X, because you are a frequent flyer there;

(c) a path from s to t minimizing the number of intermediate stops;
(d) a path from s to t minimizing the number of changes from one airline to another;
(e) a path from s to t minimizing the overall travel time from departure from s to landing at t.

3.3 In a communication network connecting n computers, a message may be sent from a source computer s to a target computer t via several intermediate stations. The probability of a message to get from a to b without corruption is $p(a, b)$. How can one choose a path from s to t in the network, so that the probability of receiving the correct message is maximized? Like in exercise 3.2, you may use algorithm A, but you may not change it.

3.4 Cartographers have been struggling for long with the problem of how many colors they need to be able to print any map according to the rules that

(a) there is no restriction on the number, shape or size of the countries in a given map;
(b) countries having a common border (not just a single point, like Utah and New Mexico) should be given different colors.

It has been shown in 1977 that *four colors suffice*: this is the famous 4-color problem, which has been open for many years. Show how to translate the problem of coloring a map using k colors into a graph-theoretic one, and give an algorithm for $k = 2$.
Remark: No algorithm is needed for $k = 4$, since by the mentioned theorem, every map is 4-colorable. Interestingly, for $k = 3$, the problem seems to be very difficult, and so far, no reasonable algorithm is known.

3.5 A *De Bruijn sequence* of order k is a cyclic binary string of length 2^k that includes each of the 2^k possible binary strings of length k exactly once as a substring. For example, 11010001 is a De Bruijn sequence of order 3, as each of the possible 8 strings of length 3, 000, 001, 010, 011, 100, 101, 110, 111, appears exactly once, namely, at positions 5, 6, 3, 7, 4, 2, 1, and 8, respectively. Construct a De Bruijn sequence of order 4.

The following reduction may be used to solve the problem. Consider a graph $G = (V, E)$ defined as follows. The vertices correspond to the 2^{k-1} binary strings representing the integers

$$0, 1, \ldots, 2^{k-1} - 1$$

in a standard binary encoding. There is an edge labeled b, for $b \in \{0, 1\}$, emanating from a vertex corresponding to $x_1x_2 \cdots x_{k-1}$ to the vertex

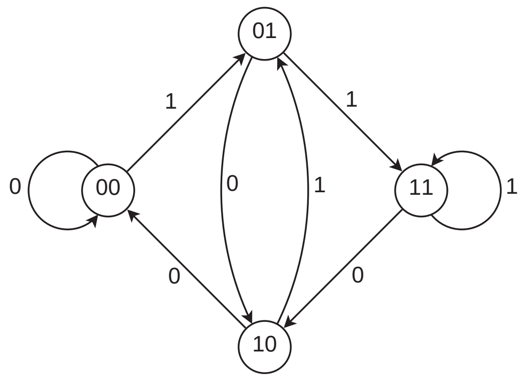

Figure 3.10. Graph for the construction of a De Bruijn sequence.

corresponding to $x_2x_3\cdots x_{k-1}b$, that is, if the suffix of length $k-2$ of the former is the prefix of the latter. Figure 3.10 displays this graph for $k=3$. Every vertex has exactly two outgoing and two incoming edges. It can be shown that in this case, there exists a so-called *Euler circuit*, which is a closed path traversing each of the edges exactly once. One possible such path in our example is 01-11-11-10-01-10-00-00-01. Concatenating the labels on the edges of such a path yields the De Bruijn sequence.