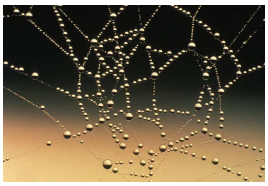
8

Graph Representation



Scientific results are mostly available in the form of articles in journals and conference proceedings, and on various web[1] *resources. These articles are not self-contained, but cite previous articles with related content. When you read an interesting article from 1975 today, you may ask yourself what the current state of the art is. In particular, you may want to know which newer articles cite the old article. Projects such as Google Scholar provide this functionality by analyzing the reference sections of articles and building a database of articles that efficiently supports looking up articles that cite a given article.*

We can easily model this situation by a directed graph. The graph has a node for each article and an edge for each citation. An edge (u,v) from article u to article v means that u cites v. When an article is processed, the outgoing edges can be constructed directly from the list of references. In this way, every node (= article) stores all its outgoing edges (= the articles cited by it) but not the incoming edges (the articles citing it). If every node were also to store the incoming edges, it would be easy to find the citing articles. One of the main tasks of Google Scholar is to construct the reversed edges. This example shows that the cost of even a very basic elementary operation on a graph, namely finding all edges entering a particular node, depends heavily on the representation of the graph. If the incoming edges are stored explicitly, the operation is easy; if the incoming edges are not stored, the operation is nontrivial.

In this chapter, we shall give an introduction to the various possibilities for representing graphs in a computer. We focus mostly on directed graphs and assume that an undirected graph $G = (V,E)$ is represented as the corresponding (bi)directed graph $G' = (V, \bigcup_{\{u,v\}\in E} \{(u,v),(v,u)\})$. The top row of Fig. 8.1 shows an undirected graph and the corresponding bidirected graph. Most of the data structures presented also allow us to represent multiple parallel edges and self-loops. We start with a survey of the operations that we may want to support:

- *Accessing associated information.* Given a node or an edge, we frequently want to access information associated with it, for example the weight of an edge or the distance to a node. In many representations, nodes and edges are objects, and we can store this information directly as a member of these objects. If not

[1] The picture above shows a spider web (USFWS; see `commons.wikimedia.org/wiki/Image:Water_drops_on_spider_web.jpg`).

otherwise mentioned, we assume that $V = 1..n$ so that information associated with nodes can be stored in arrays. When all else fails, we can always store node or edge information in a hash table. Hence, accesses can be implemented to run in constant time. In the remainder of this book, we abstract from the various options for realizing access by using the data types *NodeArray* and *EdgeArray* to indicate array-like data structures that can be indexed by nodes and by edges, respectively.

- *Navigation.* Given a node, we may want to access its outgoing edges. This operation is at the heart of most graph algorithms. As we have seen in the example above, we sometimes also want to know the incoming edges.
- *Edge queries.* Given a pair of nodes (u,v), we may want to know whether this edge is in the graph. This can always be implemented using a hash table, but we may want to have something even faster. A more specialized but important query is to find the *reverse edge* (v,u) of a directed edge $(u,v) \in E$ if it exists. This operation can be implemented by storing additional pointers connecting edges with their reversals.
- *Construction, conversion and output.* The representation most suitable for the algorithmic problem at hand is not always the representation given initially. This is not a big problem, since most graph representations can be translated into each other in linear time.
- *Update.* Sometimes we want to add or remove nodes or edges. For example, the description of some algorithms is simplified if a node is added from which all other nodes can be reached (see, Fig. 10.10).

In Sect. 8.1 we begin with a very simple representation by a list of edges and continue in Sects. 8.2 and 8.3 with more structured representations that allow direct access from nodes to their incident edges. An interesting link to linear algebra comes from the matrix representation discussed in Sect. 8.4. For graphs with special structure, the *implicit* representations introduced in Sect. 8.5 can give extra performance. Parallel aspects are treated in Sect. 8.6.

8.1 Unordered Edge Sequences

Perhaps the simplest representation of a graph is as an unordered sequence of edges. Each edge contains a pair of node indices and, possibly, associated information such as an edge weight. Whether these node pairs represent directed or undirected edges is merely a matter of interpretation. Sequence representation is often used for input and output. It is easy to add edges or nodes in constant time. However, many other operations, in particular navigation, take time $\Theta(m)$, which is prohibitively slow. Only a few graph algorithms work well with the edge sequence representation; most algorithms require easy access to the edges incident to any given node. In this case the ordered representations discussed in the following sections are appropriate. In Chap. 11, we shall see two minimum-spanning-tree algorithms: One works well with an edge sequence representation and the other needs a more sophisticated data structure.

8.2 Adjacency Arrays – Static Graphs

To support easy access to the edges leaving any particular node, we can store the edges leaving any node in an array. If no additional information is stored with the edges, this array will just contain the indices of the target nodes. If the graph is *static*, i.e., does not change over time, we can concatenate all these little arrays into a single edge array E. An additional array V stores the starting positions of the subarrays, i.e., for any node v, $V[v]$ is the index in E of the first edge out of v. It is convenient to add a dummy entry $V[n+1]$ with $V[n+1] = m+1$. The edges out of any node v are then easily accessible as $E[V[v]], \ldots, E[V[v+1]-1]$; the dummy entry ensures that this also holds true for node n. If a node v has no outgoing edge, $V[v] = V[v+1]$. Figure 8.1 (middle row, left side) shows an example.

The memory consumption for storing a directed graph using adjacency arrays is $n+m+\Theta(1)$ words. This is even more compact than the $2m$ words needed for an edge sequence representation.

Adjacency array representations can be generalized to store additional information: We may store information associated with edges in separate arrays or within the edge array. If we also need incoming edges, we may use additional arrays V' and E' to store the reversed graph.

Exercise 8.1. Design a linear-time algorithm for converting an edge sequence representation of a directed graph into an adjacency array representation. You should use only $O(1)$ auxiliary space. Hint: View the problem as the task of sorting edges by their source node and adapt the integer-sorting algorithm shown in Fig. 5.34.

8.3 Adjacency Lists – Dynamic Graphs

Edge arrays are a compact and efficient graph representation. Their main disadvantage is that it is expensive to add or remove edges. For example, assume that we want to insert a new edge (u,v). Even if there is room in the edge array E to accommodate it, we still have to move the edges associated with nodes $u+1$ to n one position to the right, which takes time $O(m)$.

In Chap. 3, we learned how to implement dynamic sequences. We can use any of the solutions presented there to produce a dynamic graph data structure. For each node v, we represent the sequence E_v of outgoing (or incoming, or both outgoing and incoming) edges by an unbounded array or by a (singly or doubly) linked list. We inherit the advantages and disadvantages of the respective sequence representations. Unbounded arrays are more cache-efficient. Linked lists allow constant-time insertion and deletion of edges at arbitrary positions. Most graphs arising in practice are sparse in the sense that every node has only a few incident edges. Adjacency lists for sparse graphs should be implemented without the dummy item introduced in Sect. 3.2, because an additional item per node would waste $\Theta(n)$ space. In the example in Fig. 8.1 (middle row, right side), we show circularly linked lists.

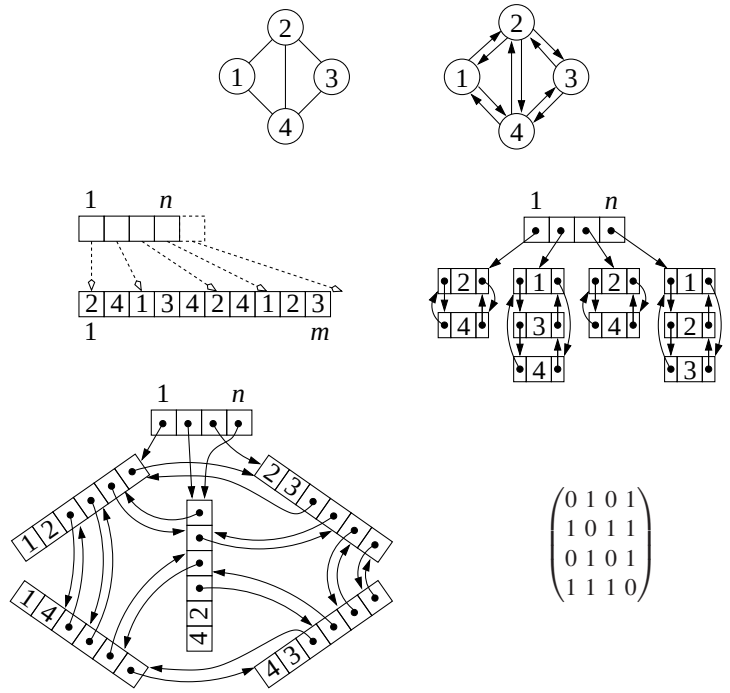


Fig. 8.1. The *first row* shows an undirected graph and the corresponding bidirected graph. The *second row* shows the adjacency array and adjacency list representations of this bidirected graph. The *third row* shows the linked-edge-objects representation and the adjacency matrix. In the former, there are five edge objects. Each object stores the names of its two endpoints and four pointers. The first two pointers point to the predecessor and successor edges of the first endpoint and the second two pointers do the same for the second endpoint. Thus, the third and fourth pointers of the edge $(1,2)$ point to the edges $(4,2)$ and $(2,3)$ respectively.

Exercise 8.2. Suppose the edges adjacent to a node u are stored in an unbounded array E_u, and an edge $e = (u,v)$ is specified by giving its position in E_u. Explain how to remove $e = (u,v)$ in constant amortized time. Hint: You do *not* have to maintain the relative order of the other edges.

Exercise 8.3. Explain how to implement the algorithm for testing whether a graph is acyclic discussed in Sect. 2.12 so that it runs in linear time, i.e., design an appropriate graph representation and an algorithm using it efficiently. Hint: Maintain a queue of nodes with outdegree 0.

Bidirected graphs arise frequently. Undirected graphs are naturally presented as bidirected graphs, and some algorithms that operate on directed graphs need access not only to outgoing edges but also to incoming edges. In these situations, we frequently want to store the information associated with an undirected edge or a directed

edge and its reversal only once. Also, we may want to have easy access from an edge to its reversal.

We shall describe two solutions. The first solution simply associates two additional pointers with every directed edge. One points to the reversal, and the other points to the information associated with the edge.

The second solution has only one item for each undirected edge (or pair of directed edges) and makes this item a member of two adjacency lists. So, the item for an undirected edge $\{u,v\}$ would store the node names u and v (in no particular order) and be a member of lists E_u and E_v. If we want doubly linked adjacency information, the edge object for any edge $\{u,v\}$ stores four pointers: two are used for the doubly linked list representing E_u, and two are used for the doubly linked list representing E_v. We may use the convention that the first two pointers are for the node listed first in the edge item and the last two pointers for the node listed second. Any node stores a pointer to some edge incident to it. Starting from it, all edges incident to the node can be traversed. The bottom part of Fig. 8.1 gives an example. A small complication lies in the fact that finding the other end of an edge now requires some work. Note that the edge object for an edge $\{u,v\}$ stores the endpoints in no particular order. Hence, when we explore the edges out of a node u, we must inspect both endpoints and then choose the one which is different from u. An elegant alternative is to store $u \oplus v$ in the edge object [235]. An exclusive OR with either endpoint then yields the other endpoint. This representation saves space because only one node name has to be stored for each edge instead of two. However, we now need a different convention for how to interpret the four pointers in an edge object. We could, for example, say that the first two pointers are for the node with the smaller name and the last two pointers for the node with larger name.

If, in the case of a directed graph, one wants access to the incoming and the outgoing edges, again both solutions apply. In either solution, a node must store a pointer to one of its outgoing edges and to one of its incoming edges.

8.4 The Adjacency Matrix Representation

An n-node graph can be represented by an $n \times n$ *adjacency matrix* A. A_{ij} is 1 if $(i,j) \in E$ and 0 otherwise. Edge insertion or removal and edge queries work in constant time. It takes time $O(n)$ to obtain the edges entering or leaving a node. This is only efficient for very dense graphs with $m = \Omega(n^2)$. The storage requirement is n^2 bits. For very dense graphs, this may be better than the $n+m+O(1)$ words required for adjacency arrays. However, even for dense graphs, the advantage is small if additional edge information is needed.

Exercise 8.4. Explain how to represent an undirected graph with n nodes and without self-loops using $n(n-1)/2$ bits.

Perhaps more important than actually storing the adjacency matrix is the conceptual link between graphs and linear algebra introduced by the adjacency matrix. On the

one hand, graph-theoretic problems can be solved using methods from linear algebra. For example, if $C = A^k$, then C_{ij} counts the number of paths from i to j with exactly k edges.

Exercise 8.5. Explain how to store an $n \times n$ matrix A with m nonzero entries using storage $O(m+n)$ such that a matrix–vector multiplication Ax can be performed in time $O(m+n)$. Describe the multiplication algorithm. Expand your representation so that products of the form $x^T A$ can also be computed in time $O(m+n)$.

On the other hand, graph-theoretic concepts can be useful for solving problems from linear algebra. For example, suppose we want to solve the matrix equation $Bx = c$, where B is a symmetric matrix. Now consider the corresponding adjacency matrix A, where $A_{ij} = 1$ if and only if $B_{ij} \neq 0$. If an algorithm for computing connected components finds that the undirected graph represented by A contains two distinct connected components, this information can be used to reorder the rows and columns of B such that we obtain an equivalent equation of the form

$$\begin{pmatrix} B_1 & 0 \\ 0 & B_2 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} c_1 \\ c_2 \end{pmatrix}.$$

This equation can now be solved by solving $B_1 x_1 = c_1$ and $B_2 x_2 = c_2$ separately. In practice, the situation is more complicated, since we rarely have matrices whose corresponding graphs are not connected. Still, more sophisticated graph-theoretic concepts such as cuts can help to discover structure in the matrix which can then be exploited to solve problems in linear algebra.

8.5 Implicit Representations

Many applications work with graphs of special structure. Frequently, this structure can be exploited to obtain simpler and more efficient representations. We shall give two examples.

The *grid graph* $G_{k\ell}$ with node set $V = [0..k-1] \times [0..\ell-1]$ and edge set

$$E = \big\{((i,j),(i,j')) \in V^2 : |j-j'| = 1\big\} \cup \big\{((i,j),(i',j)) \in V^2 : |i-i'| = 1\big\}$$

is completely defined by the two parameters k and ℓ. Figure 8.2 shows $G_{3,4}$. Edge weights could be stored in two two-dimensional arrays, one for the vertical edges and one for the horizontal edges.

An *interval graph* is defined by a set of intervals. For each interval, we have a node in the graph, and two nodes are adjacent if the corresponding intervals overlap. We may use open or closed intervals.

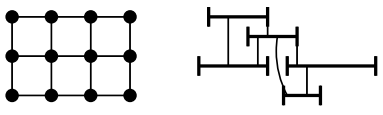


Fig. 8.2. The grid graph G_{34} (*left*) and an interval graph with five nodes and six edges (*right*).

Exercise 8.6 (representation of interval graphs).

(a) Show that, for any set of n intervals, there is a set of intervals whose endpoints are integers in $[1..2n]$, and that defines the same graph.
(b) Devise an algorithm that decides whether the graph defined by a set of n intervals is connected. Hint: Sort the endpoints of the intervals and then scan over the endpoints in sorted order. Keep track of the number of intervals that have started but not ended.
(c*) Devise a representation for interval graphs that needs $O(n)$ space and supports efficient navigation. Given an interval I, you need to find all intervals I' intersecting it; I' intersects I if I contains an endpoint of I' or $I \subseteq I'$. How can you find the former and the latter kinds of interval?

8.6 Parallel Graph Representation

Here, we discuss graph representations suitable for parallel processing. We do so first for shared memory and then for distributed memory.

8.6.1 Shared Memory

In a shared-memory machine, one may use the same graph representation as in the sequential case. As with other shared-memory data structures, read-only parallel access to the graph is efficient. However, we have to discuss how to do parallel construction and (bulk) updates. As already mentioned in Exercise 8.1, construction of an adjacency array can be viewed as the problem of sorting the edges by their endpoints. Hence, we can use the parallel algorithm presented in Sect. 5.11. At least in theory, it is relevant that we have very small keys – fewer than the elements. In this case, a CRCW-PRAM can perform the conversion using expected linear work and logarithmic time [260] (see also Sect. 5.17).

8.6.2 Distributed Memory

In a distributed-memory machine, one often *partitions* the graph by assigning each node to one PE. Another instance of the owner computes principle. For a partitioned graph, *cut* edges whose endpoints reside on different PEs require special attention. For cut edges, the graph representation needs to be able to identify the ID of the PEs responsible for the endpoints. During computations on a distributed graph, passing information along cut edges implies communication. Figure 8.3 gives an example.

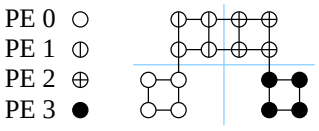


Fig. 8.3. A graph with 16 nodes partitioned between four PEs such that four edges are cut.

In order to minimize the communication required, it makes sense to choose the partition carefully. Hence, there has been intensive research on graph partitioning [59]. The best general-purpose graph-partitioning methods are quite sophisticated multilevel algorithms beyond the scope of this book. Since these methods are also relatively expensive, one also uses application-specific information to get a good partition fast.

For example, suppose we want to partition a web graph whose nodes represent web pages. We can obtain a reasonable partition by sorting the nodes by their URL. More precisely, the key used for a URL of the form d/f could be d^R/f, where d^R is the mirror image of d. For example, `algo2.iti.kit.edu/sanders.php` could be mapped to `ude.tik.iti.2ogla/sanders.php`. Then we cut the resulting ordering into balanced pieces. This partition has reasonable quality since many links are between nearby pages in this ordering.

Similarly, in many applications the nodes have a position in a geometrical space, for example geographical positions in road networks or three-dimensional coordinates in a graph describing a mathematical simulation. Interestingly, we can use the sorting idea from the URL example for multidimensional positions also. We map multidimensional coordinates to a single dimension. This is done using *space-filling curves* [27]. Figure 8.4 gives examples. A particularly simple such mapping is called *Z-order* or *Morton ordering*. It maps a d-tuple of k-bit integers $(x_1,\ldots,x_d)$ to dk bits by interleaving the bits. First come the first bits of all x_i's, then their second bits, and so on. Formally, bit $i \in 0..k-1$ of x_j is mapped to bit $id+j$ of the output. Figure 8.5 gives an example for $d=3$ and $k=4$.

Handling High Degree Vertices. The node based graph partitioning described above does not work for graphs which have nodes with very high degree. For nodes v with degree $\Omega(m/p)$, the work involved in handling just v can exceed the average work

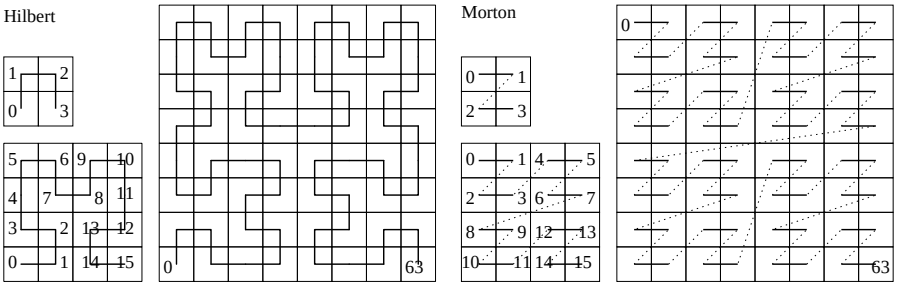


Fig. 8.4. Mapping two-dimensional points to a single dimension using space-filling curves.

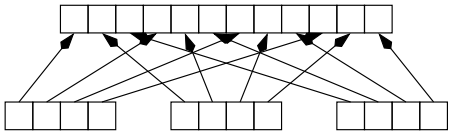


Fig. 8.5. Mapping of bits for 3D Morton ordering (3×4 bits $\rightarrow$ 12 bits).

per PE – leading to severe load imbalance. These nodes can be assigned to several PEs by partitioning their adjacency list between them. These PEs will then have to coordinate their work.

8.7 Implementation Notes

We have seen several representations of graphs in this chapter. They are suitable for different sets of operations on graphs, and can be tuned further for maximum performance in any particular application. The edge sequence representation is good only in specialized situations. Adjacency matrices are good for rather dense graphs. Adjacency lists are good if the graph changes frequently. Very often, some variant of adjacency arrays is fastest. This may be true even if the graph changes, because often there are only a few changes, all changes happen in an initialization phase of a graph algorithm, changes can be agglomerated into occasional rebuildings of the graph, or changes can be simulated by building several related graphs.

There are many variants of the adjacency array representation. Information associated with nodes and edges may be stored together with these objects or in separate arrays. A rule of thumb is that information that is frequently accessed should be stored with the nodes and edges. Rarely used data should be kept in separate arrays, because otherwise it would often be moved to the cache without being used. However, there can be other, more complicated reasons why separate arrays may be faster. For example, if both adjacency information and edge weights are read but only the weights are changed, then separate arrays may be faster because the amount of data written back to the main memory is reduced.

Unfortunately, no graph representation is best for all purposes. How can one cope with the zoo of graph representations? First, libraries such as LEDA and the Boost graph library offer several different graph data types, and one of them may suit your purposes. Second, if your application is not particularly time- or space-critical, several representations might do and there is no need to devise a custom-built representation for the particular application. Third, we recommend that graph algorithms should be written in the style of generic programming [119]. The algorithms should access the graph data structure only through a small interface – a set of operations such as iterating over the edges out of a node, accessing information associated with an edge, and proceeding to the target node of an edge. A graph algorithm that only accesses the graph using this interface can be run on any representation that realizes the interface. In this way, one can experiment with different representations. Fourth, if you have to build a custom representation for your application, make it available to others.

8.7.1 C++

LEDA [194, 217, 235] offers a powerful graph data type that supports a large variety of operations in constant time and is convenient to use, but is also space-consuming.

Therefore LEDA also implements several more space-efficient adjacency array representations.

The Boost graph library [50, 195] emphasizes a strict separation of representation and interface. In particular, Boost graph algorithms run on any representation that realizes the Boost interface. Boost also offers its own graph representation class *adjacency_list*. A large number of parameters allow one to choose between variants of graphs (directed and undirected graphs and multigraphs[2]), types of navigation available (in-edges, out-edges, ...), and representations of node and edge sequences (arrays, linked lists, sorted sequences, ...).

LEMON (Library for Efficient Modeling and Optimization in Networks) [200] also emphasizes the strict separation of representation and interface. LEMON offers a variety of general graph concepts, e.g., undirected graphs, directed graphs, and bipartite graphs, as well as special graph classes, e.g., grids and hypercubes. LEMON offers a richer class of graph algorithms than does the Boost library.

8.7.2 Java

JGraphT [166] offers rich support for graphs. It has a clear separation between interfaces, algorithms, and representations. It offers a rich class of algorithms.

8.8 Historical Notes and Further Findings

Special classes of graphs may result in additional requirements on their representation. An important example is *planar graphs* – graphs that can be drawn in the plane without edges crossing. Here, the ordering of the edges adjacent to a node should be in counterclockwise order with respect to a planar drawing of the graph. In addition, the graph data structure should efficiently support iterating over the edges along a *face* of the graph, a cycle that does not enclose any other node. LEDA offers representations for planar graphs.

Recall that *bipartite graphs* are special graphs where the node set $V = L \cup R$ can be decomposed into two disjoint subsets L and R such that the edges are only between nodes in L and nodes in R. All representations discussed here also apply to bipartite graphs. In addition, one may want to store the two sides L and R of the graph. A bipartite graph can be represented by an $|L| \times |R|$ matrix.

Hypergraphs $H = (V,E)$ are generalizations of graphs, where edges can connect more than two nodes. Hypergraphs are conveniently represented as the corresponding bipartite graph $B_H = (E \cup V, \{(e,v) : e \in E, v \in V, v \in e\})$.

Cayley graphs are an interesting example of implicitly defined graphs. Recall that a set V is a *group* if it has an associative multiplication operation $*$, a neutral element, and a multiplicative inverse operation. The *Cayley graph* (V,E) with respect to a set $S \subseteq V$ has the edge set $\{(u, u*s) : u \in V, s \in S\}$. Cayley graphs are useful because graph-theoretic concepts can be useful in group theory. On the other hand,

[2] Multigraphs allow multiple parallel edges.

group theory yields concise definitions of many graphs with interesting properties. For example, Cayley graphs have been proposed as interconnection networks for parallel computers [17].

In this book, we have concentrated on convenient data structures for *processing* graphs. There has also been a lot of work on *storing* graphs in a flexible, portable, space-efficient way. Significant compression is possible if we have a priori information about the graph. For example, the edges of a triangulation of n points in the plane can be represented with about $6n$ bits [74, 281].