ALAN GIBBONS

Algorithmic graph theory

ALAN GIBBONS

Department of Computer Sciences, University of Warwick

Algorithmic graph theory

The right of the University of Cambridge to print and sell all manner of books was granted by Henry VIII in 1534. The University has printed and published continuously since 1584.

CAMBRIDGE UNIVERSITY PRESS

Cambridge

London New York New Rochelle

Melbourne Sydney

Published by the Press Syndicate of the University of Cambridge
The Pitt Building, Trumpington Street, Cambridge CB2 1RP
32 East 57th Street, New York, NY 10022, USA
10 Stamford Road, Oakleigh, Melbourne 3166, Australia

First published 1985

Printed in Great Britain by the University Press, Cambridge

Library of Congress catalogue card number: 84–23835

British Library cataloguing in publication data

Gibbons, Alan
Algorithmic graph theory.
1. Graph theory 2. Algorithms
I. Title
511′.5 QA166

ISBN 0 521 24659 8 hard covers
ISBN 0 521 28881 9 paperback

UP

Preface

In the last decade or so work in graph theory has centred on algorithmic interests rather than upon existence or characterisation theorems. This book reflects that change of emphasis and is intended to be an introductory text for undergraduates or for new postgraduate students.

The book is aimed primarily at computer scientists. For them graph theory provides a useful analytical tool and algorithmic interests are bound to be uppermost. The text does, however, contain an element of traditional material and it is quite likely that the needs of a wider audience, including perhaps mathematicians and engineers, will be met. Hopefully, enough of this material has been included to suggest the mathematical richness of the field.

Prerequisites for an understanding of the text have been kept to a minimum. It is essential however to have had some exposure to a high-level, procedural and preferably recursive programming language, to be familiar with elementary set notation and to be at ease with (for example, inductive) theorem proving. Where more advanced concepts are required the text is largely self-contained. This is true, for example, in the use of linear programming and in the proofs of *NP*-completeness.

There is rather more material than would be required for a one-semester course. It is possible to use the text for courses of more or of less difficulty, or to select material as it appeals. For example an elementary course might not include, amongst other material, that on branchings (in chapter 2), minimum-cost flows (in chapter 4), maximum-weight matchings (in chapter 5), postman problems (in chapter 6) and proofs of *NP*-completeness (all of chapter 8). Whatever the choice of material, any course will inevitably reflect the main preoccupation of the text. This is to identify those important problems in graph theory which have an efficient algorithmic solution (that is, those whose time-complexity is polynomial in the problem size) and those which, it is thought, do not. In this endeavour the *most* efficient of the known polynomial time algorithms have not necessarily been described. These algorithms can require explanations that are *too* lengthy and may have difficult proofs of correctness. One such example is graph planarity testing in *linear*-time. It has been thought preferable to go for breadth of material and, where required, to provide references to more difficult and stronger results. Nevertheless, a body of material and quite a few results, which are not easily available elsewhere, have been presented in elementary fashion.

The exercises which appear at the ends of chapters often extend or motivate the material of the text. For this reason outlines of solutions are invariably included. Some benefit can certainly be obtained by reading these sections even if detailed solutions are not sought.

Thanks are due to Valerie Gladman for her cheerful typing of the manuscript. Primary and secondary sources of material are referenced at the ends of chapters. I gratefully acknowledge my debt to the authors of these works. However, I claim sole responsibility for any obscurities and errors that the text may contain.

A. M. Gibbons *Warwick*, January 1984

Contents

	Preface	(xi)
1	**Introducing graphs and algorithmic complexity**	1
1.1	Introducing graphs	1
1.2	Introducing algorithmic complexity	8
1.3	Introducing data structures and depth-first searching	16
	1.3.1. Adjacency matrices and adjacency lists	17
	1.3.2. Depth-first searching	20
	1.3.3. Two linear-time algorithms	24
1.4	Summary and references	32
	Exercises	33
2	**Spanning-trees, branchings and connectivity**	39
2.1	Spanning-trees and branchings	39
	2.1.1. Optimum weight spanning-trees	40
	2.1.2. Optimum branchings	42
	2.1.3. Enumeration of spanning-trees	49
2.2	Circuits, cut-sets and connectivity	54
	2.2.1. Fundamental circuits of a graph	54
	2.2.2. Fundamental cut-sets of a graph	57
	2.2.3. Connectivity	60
2.3	Summary and references	62
	Exercises	63
3	**Planar graphs**	67
3.1	Basic properties of planar graphs	67
3.2	Genus, crossing-number and thickness	71
3.3	Characterisations of planarity	75
	3.3.1. Dual graphs	81
3.4	A planarity testing algorithm	85
3.5	Summary and references	92
	Exercises	93

4 Networks and flows 96
4.1 Networks and flows 96
4.2 Maximising the flow in a network 98
4.3 Menger's theorems and connectivity 106
4.4 A minimum-cost flow algorithm 111
4.5 Summary and references 118
Exercises 120

5 Matchings 125
5.1 Definitions 125
5.2 Maximum-cardinality matchings 126
5.2.1. Perfect matchings 134
5.3 Maximum-weight matchings 136
5.4 Summary and references 147
Exercises 148

6 Eulerian and Hamiltonian tours 153
6.1 Eulerian paths and circuits 153
6.1.1. Eulerian graphs 155
6.1.2. Finding Eulerian circuits 156
6.2 Postman problems 161
6.2.1. Counting Eulerian circuits 162
6.2.2. The Chinese postman problem for undirected graphs 163
6.2.3. The Chinese postman problem for digraphs 165
6.3 Hamiltonian tours 169
6.3.1. Some elementary existence theorems 169
6.3.2. Finding all Hamiltonian tours by matricial products 173
6.3.3. The travelling salesman problem 175
6.3.4. 2-factors of a graph 182
6.4 Summary and references 184
Exercises 185

7 Colouring graphs 189
7.1 Dominating sets, independence and cliques 189
7.2 Colouring graphs 195
7.2.1. Edge-colouring 195
7.2.2. Vertex-colouring 198
7.2.3. Chromatic polynomials 201
7.3 Face-colourings of embedded graphs 204
7.3.1. The five-colour theorem 204
7.3.2. The four-colour theorem 207
7.4 Summary and references 210
Exercises 212

8 Graph problems and intractability 217
8.1 Introduction to *NP*-completeness 217

8.1.1. The classes *P* and *NP* 217
8.1.2. *NP*-completeness and Cook's theorem 222
8.2 *NP*-complete graph problems 227
8.2.1. Problems of vertex cover, independent set and clique 227
8.2.2. Problems of Hamiltonian paths and circuits and the travelling salesman problem 229
8.2.3. Problems concerning the colouring of graphs 235
8.3 Concluding comments 241
8.4 Summary and references 244
Exercises 245

Appendix: On linear programming 249

Author index 254

Subject index 256

1

Introducing graphs and algorithmic complexity

In this chapter we introduce the basic language of graph theory and of algorithmic complexity. These mainstreams of interest are brought together in several examples of graph algorithms.

Most problems on graphs require a systematic traversal or search of the graph. The actual method of traversal used can have advantageous structural characteristics which make an efficient solution possible. We illustrate both this and the use of an efficient representation of a graph for computational purposes.

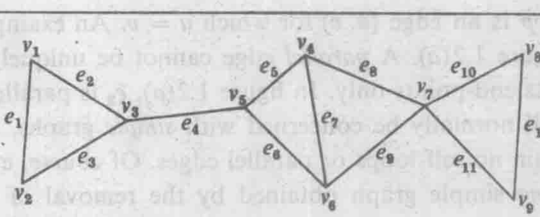
The definitions and concepts outlined here will serve as a foundation for the material of later chapters.

1.1 Introducing graphs

This section introduces the basic vocabulary of graph theory. The subject contains an excess of non-standardised terminology. In the following paragraphs we introduce a relatively small number of widely used definitions which will nevertheless meet our needs with very few later additions.

Geometrically we define a *graph* to be a set of points (*vertices*) in space which are interconnected by a set of lines (*edges*). For a graph *G* we denote

Fig. 1.1



the *vertex-set* by V and the *edge-set* by E and write $G = (V, E)$. Figure 1.1 shows a graph, $G = (\{v_1, v_2, \ldots, v_9\}, \{e_1, e_2, \ldots, e_{12}\})$.

We shall denote the number of vertices in a graph by $n = |V|$ and the number of edges by $|E|$. If both n and $|E|$ are finite, as we shall normally presume to be the case, then the graph is said to be *finite*.

We can specify an edge by the two vertices (called its *end-points*) that it connects. If the end-points of e are v_i and v_j then we write $e = (v_i, v_j)$ or $e = (v_j, v_i)$. Thus an equivalent definition of the graph in figure 1.1 is:

$$G = (V, E), V = \{v_1, v_2, \ldots, v_9\}$$

$$E = \{(v_1, v_2), (v_1, v_3), (v_2, v_3), (v_3, v_5), (v_4, v_5), (v_4, v_6), (v_4, v_7), (v_5, v_6), (v_6, v_7), (v_7, v_8), (v_7, v_9), (v_8, v_9)\}$$

If an edge e has v as an end-point, then we say that e is *incident with* v. Also if $(u, v) \in E$ then u is said to be *adjacent to* v. For example, in figure 1.1 the edges e_4, e_5 and e_6 are incident with v_5 which is adjacent to v_3, v_4 and v_6. We also say that two edges are adjacent if they have a common end-point. In figure 1.1, for example, any pair of e_8, e_9, e_{10} and e_{11} are adjacent.

The *degree* of a vertex v, written $d(v)$, is the number of edges incident with v. In figure 1.1 we have $d(v_1) = d(v_2) = d(v_8) = d(v_9) = 2$, $d(v_3) = d(v_4) = d(v_5) = d(v_6) = 3$ and $d(v_7) = 4$. A vertex v for which $d(v) = 0$ is called an *isolated* vertex. Our first theorem is a well-known one concerning the vertex degrees of a graph.

Theorem 1.1. The number of vertices of odd-degree in a finite graph is even.

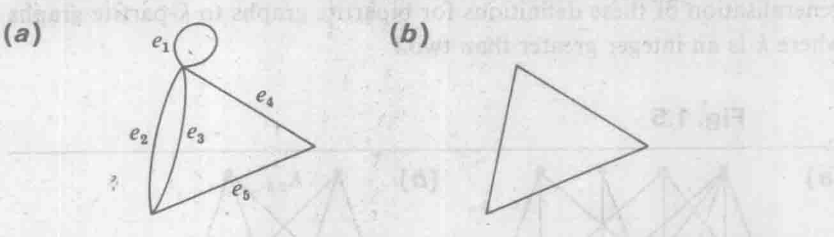
Proof. If we add up the degrees of all the vertices of a graph then the result must be twice the number of edges. This is because each edge contributes once to the sum for each of its ends. Hence:

$$\sum_i d(v_i) = 2 \cdot |E|$$

The right-hand side of this equation is an even number as is the contribution to the left-hand side from vertices of even-degree. Therefore the sum of the degrees of those vertices of odd-degree is even and the theorem follows. ■

A *self-loop* is an edge (u, v) for which $u = v$. An example is e_1 in the graph of figure 1.2(a). A *parallel* edge cannot be uniquely identified by specifying its end-points only. In figure 1.2(a), e_2 is parallel to e_3. In this text we shall normally be concerned with *simple* graphs, that is, graphs which contain no self-loops or parallel edges. Of course, every graph has an *underlying* simple graph obtained by the removal of self-loops and parallel edges. Thus figure 1.2(*b*) shows the simple graph underlying figure 1.2(*a*). By the term *multi-graph* we mean a graph with parallel edges but with no self-loops. From now on we shall employ the term *graph* to mean a simple graph unless we explicitly say otherwise.

Fig. 1.2



A graph for which every pair of distinct vertices defines an edge is called a *complete* graph. The complete graph with n vertices is denoted by K_n. Figure 1.3 shows K_3 and K_5. In a *regular* graph every vertex has the same degree, if this is k then the graph is called k*-regular*. Notice that K_n is $(n-1)$-regular. Figure 1.4 shows two examples of 3-regular graphs (also called *cubic* graphs) which, as a class, are important in colouring planar maps as we shall see in a later chapter.

Fig. 1.3

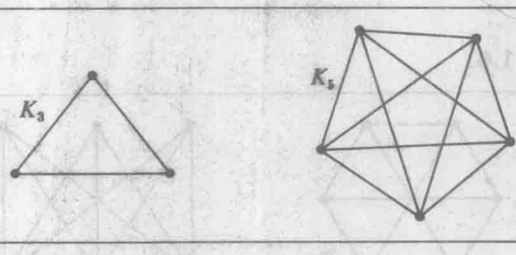
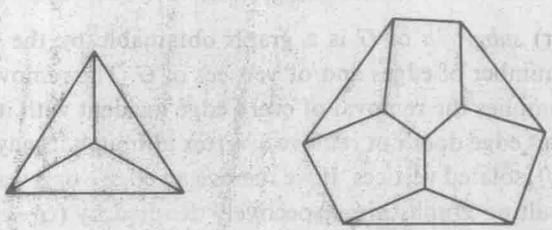
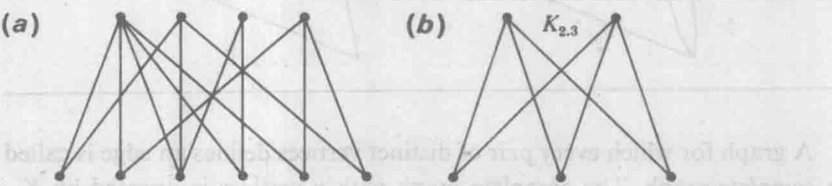


Fig. 1.4



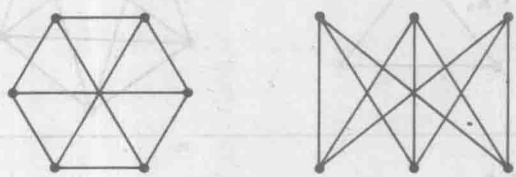
If it is possible to partition the vertices of a graph G into two subsets, V_1 and V_2, such that every edge of G connects a vertex in V_1 to a vertex in V_2 then G is said to be *bipartite*. Figure 1.5(a) and (b) shows two bipartite graphs. If every vertex of V_1 is connected to every vertex of V_2 then G is said to be a *complete bipartite* graph. In this case we denote the graph by $K_{i,j}$ where $|V_1| = i$ and $|V_2| = j$. Figure 1.5(b) shows $K_{2,3}$. There is an obvious generalisation of these definitions for bipartite graphs to k-partite graphs where k is an integer greater than two.

Fig. 1.5



Two graphs G_1 and G_2 are *isomorphic* if there is a one-to-one correspondence between the vertices of G_1 and the vertices of G_2 such that the number of edges joining any two vertices in G_1 is equal to the number of edges joining the corresponding two vertices in G_2. For example, figure 1.6 shows two graphs which are isomorphic, each being a representation of $K_{3,3}$.

Fig. 1.6



A (proper) *subgraph* of G is a graph obtainable by the removal of a (non-zero) number of edges and/or vertices of G. The removal of a vertex necessarily implies the removal of every edge incident with it, whereas the removal of an edge does not remove a vertex although it may result in one (or even two) isolated vertices. If we remove an edge e or a vertex v from G, then the resulting graphs are respectively denoted by $(G-e)$ and $(G-v)$. If H is a subgraph of G then G is called a *supergraph* of H and we write

$H \subseteq G$. A subgraph of G *induced* by a subset of its vertices, $V' \subset V$, is the graph consisting of V' and those edges of G with both end-points in V'.

A *path* from v_1 to v_i is a sequence $P = v_1, e_1, v_2, e_2, \ldots, e_{i-1}, v_i$ of alternating vertices and edges such that for $1 \leqslant j < i$, e_j is incident with v_j and v_{j+1}. If $v_1 = v_i$ then P is said to be a *cycle* or a *circuit*. In a simple graph a path or a cycle $v_1, e_1, v_2, e_2, \ldots, e_{i-1}, v_i$ can be more simply specified by the sequence of vertices $v_1, v_2, \ldots, v_i$. If in a path each vertex only appears once, then the sequence is called a *simple* path. If each vertex appears once except that $v_1 = v_i$ then P is a *simple* circuit. The *length* of a path or a cycle is the number of edges it contains. Two paths are *edge-disjoint* if they do not have an edge in common.

Two vertices v_i and v_j are *connected* if there is a path from v_i to v_j. By convention, every vertex is connected to itself. Connection is an equivalence relation (see problem 1.9) on the vertex set of a graph which partitions it into subsets $V_1, V_2, \ldots, V_k$. A pair of vertices are connected if and only if they belong to the same subset of the partition. The subgraphs induced in turn by the subsets $V_1, V_2, \ldots, V_k$, are called the *components* of the graph. A *connected* graph has only one component, otherwise it is *disconnected*. Thus the graph of figure 1.1 is connected whilst that of figure 1.9 has two components.

A *spanning* subgraph of a connected graph G is a subgraph of G obtained by removing edges only and such that any pair of vertices remain connected.

Let H be a connected graph or a component. If the removal of a vertex v disconnects H, then v is said to be an *articulation point*. For example, in figure 1.1 v_3, v_5 and v_7 are all articulation points. If H contains no articulation point then H is a *block*, sometimes called a 2-connected graph or component. If H contains an edge e, such that its removal will disconnect H, then e is said to be a *cut-edge*. Thus in figure 1.1 e_4 is a cut-edge. The end-points of a cut-edge are usually articulation points.

A graph with one or more articulation points is also called a *separable* graph. This refers to the fact that the blocks of a separable graph can be identified by disconnecting the graph at each articulation point in turn in such a way that each separated part of the graph retains a copy of the articulation point. For example, figure 1.7 shows the separated parts (or blocks) of the graph depicted in figure 1.1. Clearly, any graph is the union of its blocks.

In some applications it is natural to assign a *direction* to each edge of a graph. Thus in a diagram of the graph each edge is represented by an arrow. A graph augmented in this way is called a *directed* graph or a *digraph*. An example is shown in figure 1.8. If $e = (v_i, v_j)$ is an edge of a digraph then the order of v_i and v_j becomes significant. The edge e is under-

Fig. 1.7

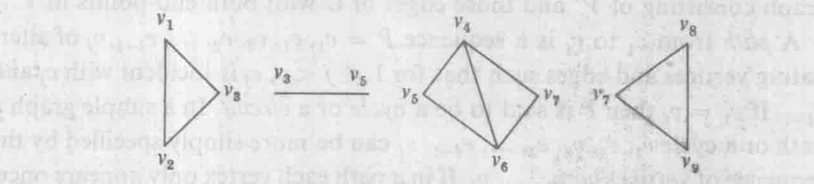
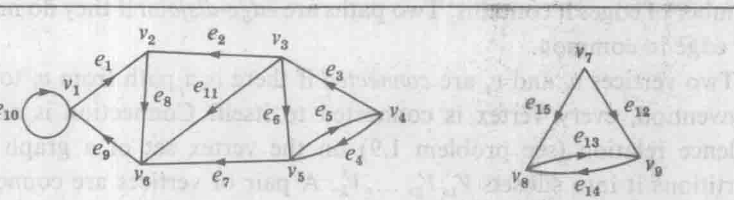


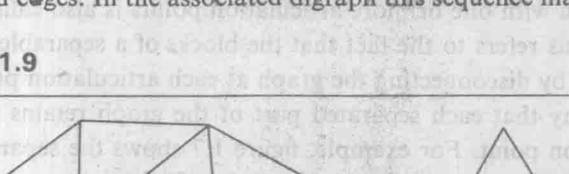
Fig. 1.8



stood to be directed from the first vertex v_i to the second vertex v_j. Thus if a digraph contains the edge (v_i, v_j) then it may or it may not contain the edge (v_j, v_i). The directed edge (v_i, v_j) is said to be *incident from* v_i and *incident to* v_j. For the vertex v, the *out-degree* $d^+(v)$ and the *in-degree* $d^-(v)$ are, respectively, the number of edges incident from v and the number of edges incident to v. A *symmetric* digraph is a digraph in which for every edge (v_i, v_j) there is an edge (v_j, v_i). A digraph is *balanced* if for every vertex v, $d^+(v) = d^-(v)$.

Of course, every digraph has an *underlying* (*undirected simple*) *graph* obtained by deleting the edge directions. Thus figure 1.9 shows this graph for the digraph of figure 1.8. As defined earlier, a path (or circuit) in a corresponding undirected graph is a sequence $S = v_1, e_1, v_2, e_2, \ldots, v_{i-1}, e_i$, of vertices and edges. In the associated digraph this sequence may be such

Fig. 1.9



that for all j, $1 \leqslant j < i$, e_j is incident from v_j and incident to v_{j+1}. In this case S is said to be a *directed* path (or circuit). Otherwise it is an *undirected* path (or circuit). Thus in figure 1.8 $(v_2, e_2, v_3, e_6, v_5, e_4, v_4, e_3, v_3, e_{11}, v_6)$ is an undirected non-simple path, while $(v_1, e_1, v_2, e_8, v_6, e_9, v_1)$ is a simple directed circuit. Because in a digraph we can define two different types of paths we can also define two different types of connectedness. Two vertices, v_1 and v_2, are said to be *strongly connected* if there is a directed path from v_1 to v_2 *and* a directed path from v_2 to v_1. If v_1 and v_2 are not strongly connected but are connected in the corresponding undirected graph, then v_1 and v_2 are said to be *weakly connected*.

Both strong connection and weak connection are equivalence relations (see problem 1.9) on the vertex set of a digraph. Of course weak connection partitions the vertices in precisely the same way that connection would partition the vertices of the corresponding undirected graph. Thus for the graph in figure 1.8, weak connection partitions the vertices into the two subsets $\{v_1, v_2, v_3, v_4, v_5, v_6\}$ and $\{v_7, v_8, v_9\}$. The subgraphs induced by these subsets are called the *weakly connected components* of the digraph. On the other hand strong connection partitions the vertices of this graph into the subsets $\{v_1, v_2, v_6\}$, $\{v_3, v_4, v_5\}$, $\{v_7\}$ and $\{v_8, v_9\}$. Each of these subsets induces a *strongly connected component* of the digraph. Notice that each edge of a digraph belongs to some weakly connected component but that it does not necessarily belong to a strongly connected component.

We now briefly introduce an important class of graphs called trees. A *tree* is a connected graph containing no circuits. A *forest* is a graph whose components (one or more in number) are trees. An *out-tree* is a directed tree in which precisely one vertex has zero in-degree. Similarly, an *in-tree* is a directed tree in which precisely one vertex has zero out-degree. A tree in which one vertex, the *root*, is distinguished, is called a *rooted-tree*. In a rooted-tree any vertex of degree one, unless it is the root, is called a *leaf*. As we shall see in theorem 1.2 there is precisely one path between any two vertices of a tree. The *depth* or *level* of a vertex in a rooted-tree is the number of edges in the path from the root to that vertex. If (u, v) is an edge of a rooted-tree such that u lies on the path from the root to v, then u is said to be the *father* of v and v is the *son* of u. An *ancestor* of u is any vertex of the path from u to the root of the tree. A *proper* ancestor of u is any ancestor of u excluding u. Similarly, if u is an ancestor of v, then v is a *descendant* of u. A *proper* descendant of u excludes u. Finally, a *binary* tree is a rooted-tree in which every vertex, unless it is a leaf, has two sons.

Theorem 1.2. If T is a tree with n vertices, then

(*a*) Any two vertices of T are connected by precisely one path.

(*b*) For any edge e, not in T, but connecting two vertices of T, the graph $(T+e)$ contains exactly one circuit.

(*c*) T has $(n-1)$ edges.

Proof. (*a*) T is connected and so there exists at least one path between any two vertices u and v. Suppose that two distinct paths, P_1 and P_2 exist between u and v. Following these paths from u to v, let them first diverge at u' and first converge at v'. That section of P_1 from u' to v' followed by that section of P_2 from v' to u' must form a circuit. By definition, T contains no circuit and so we have a contradiction.

(*b*) Let $e = (u, v)$. According to (*a*) there is precisely one path P from u to v within T. The addition of e therefore creates exactly one circuit $(P+e)$.

(*c*) Proof is by induction on the number of vertices n in T. If $n = 1$ or 2 then, trivially, the number of edges in T is $(n-1)$. We assume that the statement is true for all trees with less than n vertices. Let T have n vertices. There must be a vertex of degree one contained in T, otherwise we could trace a circuit by following any path from vertex to vertex entering each vertex by one edge and leaving by another. If we remove a vertex of degree one, v, from T we neither disconnect T or create a circuit. Hence $(T-v)$ is a tree with $(n-1)$ vertices. By the induction hypothesis $(T-v)$ has $(n-2)$ edges. Hence replacing v provides T with $(n-1)$ edges. ■

We complete our catalogue of definitions by introducing *weighted* graphs. In some applications it is natural to assign a number to each edge of a graph. For any edge e, this number is written $w(e)$ and is called its *weight*. Naturally the graph in question is called a *weighted graph*. The *weight* of a (*sub*)*graph* is equal to the sum of the weights of its edges. Often of interest here is a path (or cycle) in which case it may be appropriate to refer to the *length* rather than the weight of the path (or cycle). This should not be confused with the length of a path (or cycle) in an unweighted graph which we defined earlier.

In the following section we introduce the other central interest of this text, namely, that of algorithmic complexity.

1.2 Introducing algorithmic complexity

Although fairly brief, this introduction to algorithmic efficiency will provide a sufficient basis for all but the final chapter of this text. That chapter provides further insight into what is introduced here, and, in particular, it explores an important class of intractable problems.

Our interest in efficiency is particularly concerned with what is called the *time-complexity* of algorithms. Since the analogous concept of *space-complexity* will be of little interest to us, we can use the term *complexity* in an unambiguous way. The *complexity* of an algorithm is simply the number of computational steps that it takes to transform the input data to the result of a computation. Generally this is a function of the quantity of the input data, commonly called the *problem size*. For graph algorithms the problem size is determined by one or perhaps both of the variables n and $|E|$.

For a problem size s, we denote the complexity of a graph algorithm A by $C_A(s)$, dropping the subscript A when no ambiguity will arise. $C_A(s)$ may vary significantly if algorithm A is applied to structurally different graphs but which are nevertheless of the same size. We therefore need to be more specific in our definition. In this text we take $C_A(s)$ to mean the *worst-case* complexity. Namely, to be the maximum number, over all input sizes s, of computational steps required for the execution of algorithm A. Other definitions can be used. For example, the *expected time*-complexity is the *average*, over all input sizes s, of the number of computational steps required.

The complexities of two algorithms for the same problem will in general differ. Let A_1 and A_2 be two such algorithms and suppose that $C_{A_1}(n) = \frac{1}{2}n^2$ and that $C_{A_2}(n) = 5n$. Then A_2 is faster than A_1 for all problem sizes $n > 10$. In fact whatever had been the (finite and positive) coefficients of n^2 and of n in these expressions, A_2 would be faster than A_1 for all n greater than some value, n_0 say. The reason, of course, is that the asymptotic growth, as the problem size tends to infinity, of n^2 is greater than that of n. The complexity of A_2 is said to be of lower *order* than that of A_1. The idea of the *order* of a function is important in complexity theory and we now need to define and to further illustrate it.

Given two functions F and G whose domain is the natural numbers, we say that the order of F is lower than or equal to the order of G provided that:

$$F(n) \leqslant K \cdot G(n)$$

for all $n > n_0$, where K and n_0 are two positive constants. If the order of F is lower than or is equal to the order of G then we write $F = O(G)$ or we say that F is $O(G)$. F and G are of the *same* order provided that $F = O(G)$ *and* that $G = O(F)$. It is occasionally convenient to write $\theta(G)$ to specify the set of all functions which are of the same order as G. Although $\theta(G)$ is defined to be a set, we conventionally write $F = \theta(G)$ to mean $F \in \theta(G)$. Illustrating these definitions, we see that $5n$ is $O(\frac{1}{2}n^2)$ but that $5n \neq \theta(\frac{1}{2}n^2)$ because $\frac{1}{2}n^2$ is not $O(5n)$. Note also that low order terms of a function can be ignored in determining the overall order. Thus the polynomial $(3n^3+6n^2+n+6)$ is $O(3n^3)$. It is obviously convenient when specifying the order of a function to describe it in terms of the simplest representative function. Thus $(3n^3+6n^2)$ is $O(n^3)$ and $\frac{1}{2}n^2$ is $O(n^2)$.

When comparing two functions in terms of order, it is often convenient