TREES AND NETWORKS
MT1151
course notes

NIGEL RAY
Department of Mathematics, University of Manchester
Manchester M13 9PL, England
nige@ma.man.ac.uk   http://www.ma.man.ac.uk/~nige/

December 18, 2003

Chapter 1

Introduction

1.1 Background

These notes serve as an introduction to a subject which bridges the traditional divide between pure and applied mathematics, and may be categorised under the general heading of **graph theory**.

At various stages of its development, graph theory has often been identified with recreational mathematics, and it is still much used in creating and solving puzzles. However, for the last century or so it has become a serious branch of pure mathematics in its own right, and it is currently the focus of intensive research activity. One reason for this popularity is the wide variety of applications, in areas so diverse as biology, economics and geography; it is also becoming a useful and important tool within pure mathematics itself.

The advent of the modern high speed computer has been an explosive catalyst, and graph theory now plays a fundamental rôle in both the research and development of hardware and software. It also underpins exciting new fields such as the study of neural networks, where attempts are being made to design computer models which function in a similar fashion to the human brain.

The computer has also revolutionised the *study* of graph theory, insofar as massive experimental calculations may now be carried out to assist with the search for theoretical order and proof. A case in point is the celebrated Four Colour Theorem, which is one of the most famous results in all of mathematics; although it was first established in the 1970s, its proof still relies crucially on extensive machine computation.

At a more practical level, many of the processes of everyday life rely on the computer analysis and control of graph theoretic models. For example, the optimisation of the flow of information through a communications network, a search through a library cataloging system, and the distribution schedule for restocking a chain of supermarkets, are all processes of this nature.

This course is intended to be a pure mathematical investigation of applicable graph theory, presented in the modern algorithmic style so influenced by the requirements of computer implementation. It is *not*, however, a course in computer programming, and can be enjoyed equally well by those who have decided that the lure of the machine is definitely not for them! One of the beauties of the subject is that it has few prerequisites other than an enquiring mind, and it may therefore be undertaken from a common starting point by students with a wide variety of backgrounds.

1.2 Source texts

It is currently highly fashionable to set up courses in this area of mathematics, and new textbooks appear in the shops every few week. In spite of this, I have found no single text which offers satisfactory coverage of all the topics discussed here, so I have chosen simply to list those that have been helpful both to myself and to students over recent years. Most of them place graph theory firmly within the wider context of Discrete Mathematics (or Combinatorics), and therefore contain a great deal of additional information. Several of them are in paperback, and they should all be available in major bookshops and libraries; those marked * may be most helpful.

- Michael Albertson & Joan Hutchinson, *Discrete Mathematics with Algorithms*, John Wiley (1988)
- Norman Biggs, *Discrete Mathematics*, Oxford University Press (1989)*
- Judith Gersting, *Mathematical Structures for Computer Science*, W H Freeman (1993)
- Alan Gibbons, *Algorithmic Graph Theory*, Cambridge University Press (1985)*
- Ronald Gould, *Graph Theory*, Benjamin/Cummings (1988)
- Bradley Jackson & Dmitri Thoro, *Applied Combinatorics with Problem Solving*, Addison-Wesley (1990)
- James McHugh, *Algorithmic Graph Theory*, Prentice Hall (1990)
- H F Mattson, Jr. *Discrete Mathematics with applications*, John Wiley (1993)
- Alan Tucker, *Applied Combinatorics*, John Wiley (1984)*

Even with the assistance of these authors, and because of the constraints of space and time, I have developed certain sections of the material in my own fashion. Errors have undoubtedly tiptoed in as a result, and I am entirely responsible for their presence; please inform me of any that you find (or merely suspect), and due acknowledgement will be given.

1.3 To the student

These notes (together with the attendant problems) are meant to be entirely self-contained, and to include all material on which you might be examined. I have taken the opportunity to make them as rigorous as possible, and to introduce notation, concepts and methodology without interrupting the flow. In consequence, they may make daunting reading at first.

Thus enter your lectures! They are intended to be an indispensable part of the learning process; unencumbered by the needs of transferring all the details from these pages to yours, they can be devoted to informal and enlightening explanations, and to providing a series of signposts through the mathematical landscape. Without them, you may be lost.

So far as my expectations of you are concerned, I have three wishes. The second of these may seem demanding, but it will be considerably mitigated by attention to the first and third.

Firstly, I would like you to absorb some of the *philosophy* of graph theory, and to appreciate that it constitutes a stimulating and useful branch of mathematics, whose concepts can be

established with proper rigour. Secondly, I hope that you will commit to memory some workable version (which may well be drastically different in style and detail from mine) of each of the 14 algorithms, and be able to apply it in straightforward cases which you have not previously seen. Thirdly, I expect you to come to terms with the mathematician's insistence that you never fully understand *how* an algorithm works, until you understand *why* it works.

One extra wish, without which no course of study is memorable, is that you should enjoy the challenge of working through the mathematics presented here. Good luck!

1.4 Guarantee

This course has been tried and tested on ten previous cohorts of first and second year Manchester undergraduates, and their comments and reactions have helped hone it to its present form. Some ex-students have even asked for updated copies of the notes more than five years after graduation! You too will have the opportunity to make your contribution later in the Semester, via the course questionnaires.

1.5 Contents

- Chapter 1: Introduction
- Chapter 2: The Language of Graphs, Trees and Networks
- Chapter 3: Connectedness
- Chapter 4: The Enumeration of Trees
- Chapter 5: Eulerian Tours
- Chapter 6: Weighted Graphs
- Chapter 7: Networks and Flows
- Chapter 8: Hamiltonian Tours and Complexity

1.6 Pictures

Chapter 2

The Language of Graphs, Trees and Networks

2.1 Questions

In order to pursue the study of graph theory from a mathematical viewpoint, it is important to ensure that we make rigorous and usable definitions of the underlying ideas. This is more easily said than done!

The commonest interpretation of a graph is as a *diagram*, consisting of finitely many points in the plane (or in space), and finitely many lines connecting certain pairs of them together. We shall make no definitions in this section, but simply indicate informally how such configurations arise in a host of practical situations, and how the solutions to related problems may be expressed by investigating the configurations in a variety of systematic ways.

Many of these problems involve vast amounts of data, and are exceedingly difficult to represent as a diagram. This is our motivation for the development of alternative descriptions, and suggests that we should tailor our methods to the possibility of computerised implementation. Such aims lie at the heart of the algorithmic approach.

As our first example, we offer the accompanying Picture 1, which is a photograph of part of an exceedingly complicated printed circuit.

Its underlying structure is indeed that of a graph, since the connectors meet at specific junctions which have been carefully chosen so that the circuit performs the required task. We can now ask certain questions about the circuit, which have important practical implications.

Question 2.1 (Connectivity) *Is every junction in the circuit connected to every other junction by at least one pathway?*

If the circuit is small and simple, we will have no trouble answering this question; but in a world which is increasingly controlled by electronic processes, we must expect to treat large and complicated examples as the norm.

If we develop a method for answering 2.1, we may then make a more subtle enquiry.

Question 2.2 (Components) *Into how many disjoint constituent parts is the circuit divided?*

If the answer to 2.1 is affirmative, then of course the answer to 2.2 is 1; but if the answer to 2.1 is negative, then the answer to 2.2 is some integer k greater than or equal to 2.

Once more, if the circuit is small and simple the components may easily be counted; and it is possible that a complicated circuit may be visibly divided into k parts. But even on a single circuit board the components may interlace in a highly intricate way, and if we further take into account that pathways may cross without junctions being made (for example by using both sides of the board), and that several boards may be connected together in space, we begin to appreciate the true nature of the question.

Let us now change scale, and display the street plan for part of a large city in the format of Picture 2. If the streets are interpreted as joining pairs of junctions then the plan has similar characteristics to those of a printed circuit, and related questions become important in the planning of services such as refuse disposal, during which expensive vehicles have to be deployed on their rounds as efficiently as possible.

Question 2.3 (Traversibility) *Can we traverse the entire plan without duplicating any street?*

An affirmative answer to 2.3 means that the equipment can be in constant use during its progression through the city neighbourhood. If the plan incorporates several nonadjacent neighbourhoods then the answer to 2.2 is obviously greater than 1, and the immediate answer to 2.3 is negative; in this case we implicitly assume that the question applies to each constituent part.

Unfortunately, cities did not evolve with such considerations in mind (at least in Britain), and we would usually expect the answer to 2.3 to be negative, even for a single neighbourhood. In this case, it would still be helpful to *minimize* the number of streets which the route would have to duplicate, thereby helping to minimize the overall cost of deployment.

Question 2.4 (Repetition) *What is the minimal number of streets which have to be repeated when traversing the entire plan?*

In answering 2.4 it will often be important to take into account the *length* of each street, since it may be more efficient to duplicate several short streets rather than a single long one. This suggests that we should associate to each street its length, and then pose a more sophisticated question.

Question 2.5 (Weighted Traverse) *What is the minimal distance involved in traversing the entire plan?*

This question was originally asked (and an algorithm given for its solution) by Kwan in 1962; everafter, it has been known as the Chinese Postperson Problem.

A street map is an example of a *communication network*, consisting of a set of nodes and a set of connections, each of which is labelled by a specific parameter. A similar example considers certain cities as nodes, with transportation links (such as air routes, railways, or roads) as connections, labelled with distances or costs. We may raise two further classic questions.

Question 2.6 (Visitation) *Is it possible to traverse the network in such a way that every node is visited exactly once?*

If the answer to 2.6 is affirmative, we then ask

Question 2.7 (Weighted Visitation) *What is the minimal cost involved in visiting every node of the network exactly once?*

In fact 2.6 is most naturally considered for a network where *every* pair of nodes is connected, and is then known as the Travelling Salesperson Problem, for obvious reasons. We give an example in Picture 3. In spite of an enormous literature having grown up around this problem during the last thirty years (and fabulous riches hanging on its solution), no truly efficient algorithm applying to 2.6 or 2.7 has ever been found.

Another example is provided by a Local Area Network, or LAN. This has computer sites as its nodes, for example in different buildings of a university, and electronic cables for its connections. Let us consider the problems associated with constructing such a LAN.

Question 2.8 (Spanning Trees) *In how many ways can the sites be connected such that there is a unique route between every pair?*

The answer is clearly at least 1, since the sites may be connected in a zigzag fashion along a single cable; but such a solution is unlikely to be the most economical. We therefore assign a construction cost as the parameter for every potential connection (so recreating the underlying data of 2.7), and ask

Question 2.9 (Weighted Spanning Trees) *What is the minimal cost of connecting the sites such that there is a unique route between every pair?*

Practically, of course, we may sometimes wish to duplicate connections in order to ensure that the system can still function when a section of the cable either fails or requires maintenance. But that is another story!

A further example consists of a system of oil pumping stations (say on the ocean floor) as nodes, and pipes, labelled with their maximum capacities, as connections. One node is assumed to be the wellhead, and another to be the terminal, and many duplicate routes may exist within the network. Assuming that the pumping stations leak no oil in operation, we need to know the total capacity of the system.

Question 2.10 (Maximum Flow) *What is the maximum rate at which oil can be pumped from wellhead to terminal, under the constraints imposed by the capacities of the pipes?*

Alternatively, we may take a set of telephone exchanges as nodes, with optical fibre connections labelled with the maximum number of calls that each can handle simultaneously. Then 2.10 is equally important in this context, addressing the maximum number of calls which the network can simultaneously transmit between two chosen exchanges.

As we shall see below, several of the above examples are mathematically indistinguishable. This motivates one of the main aims of our study, which is to develop a language in which a single algorithm may be applied to an entire class of situations which have similar mathematical structure. Indeed, such unifying aims are central to all of mathematics!

2.2 Terminology

In this section we introduce some of the basic notions and nomenclature of graph theory.

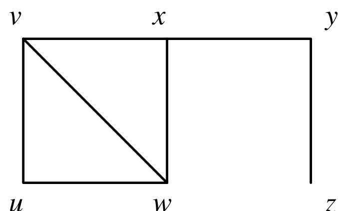
The most concise definition of a graph G is as a pair of finite sets $(V(G), E(G))$, known as the set of vertices and the set of edges respectively. Each edge consists of a pair of distinct vertices, whose order of presentation is immaterial. We often write

$$V(G) = \{v_1, \ldots, v_n\} \qquad \text{and} \qquad E(G) = \{e_1, \ldots, e_p\}, \tag{2.11}$$

and insist that n is nonzero (although the graph may have no edges, in which case we refer to it as the null graph N_n). We may abbreviate $V(G)$ to V and $E(G)$ to E if the context is unambiguous. When an edge e consists of vertices u and v, it is convenient to write $e = uv = vu$.

A diagram $D(G)$ for G consists of a set of points in the plane (or space) which correspond to the vertices, joined by lines which correspond to the edges.

Example 2.12 *Let G have vertices $\{u, v, w, x, y, z\}$ and edges $\{uv, uw, vw, vx, yz, xw, xy\}$; then*



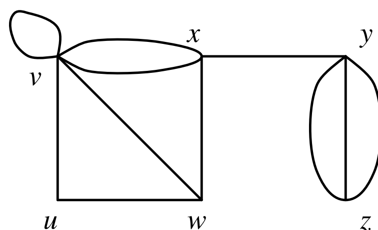
is a diagram for G.

This diagram may also be interpreted as the graph whose vertices are the appropriate points on the page, and whose edges are the connected pairs.

We say that two vertices are adjacent if they define an edge, and that an edge is incident on its endpoints, which are the defining vertices. The degree $d(v)$ of a vertex v is the number of edges incident on v; for example $d(w)$ is 3 in 2.12. A vertex is isolated if it has degree 0.

Sometimes it is important to allow a graph to have loops and multiple edges. A loop is an edge of the form vv for some vertex v, and a multiple edge is a repetition such as $\{\ldots, uv, uv, uv, \ldots\}$ amongst the edges; we may distinguish between these repetitions by writing them as $\{\ldots, e_{i-1}, e_i, e_{i+1}, \ldots\}$, in the notation of 2.11. If we wish to emphasise that loops and multiple edges are allowed in G, we refer to G as a multigraph. If, on the other hand, we wish to exclude them, we call G a simple graph. We decree that each loop contributes 2 to the degree of the vertex on which it is incident.

Example 2.13 *If we add a loop vv and edges vx, yz and yz to G in 2.12 we obtain a multigraph for which*

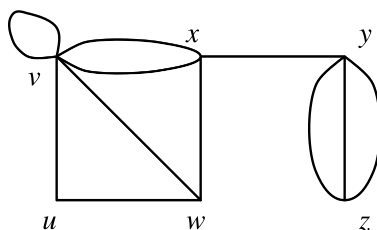


is a diagram.

In general, the term graph will encompass both multigraphs and simple graphs.

Often, for example in problems of one-way flow, it is necessary to insist that all the edges of a graph have a preferred direction. This is achieved by defining each edge e to be an *ordered* pair of vertices (u, v), and may conveniently be indicated in a diagram for G by inserting an arrow pointing from u to v along the edge. Replacing (u, v) by (v, u) will reverse the direction of the arrow. A graph whose edges are directed in this fashion will be called a digraph. Note that several different digraphs may be constructed from the same graph; thus

Example 2.14



is a diagram for one of the many digraphs which has 2.13 as its underlying graph. Reversing the arrow on the directed loop (v, v) does *not* alter this diagram!

Given a digraph, every vertex v now has an indegree $d_{\text{in}}(v)$ and an outdegree $d_{\text{out}}(v)$ defined in the obvious fashion; thus $d_{\text{in}}(w)$ is 3 and $d_{\text{out}}(w)$ is 0 in 2.14. Of course, $d(v)$ in the underlying graph coincides with $d_{\text{in}}(v) + d_{\text{out}}(v)$.

In order to store a graph on machine, the adjacency relations are usually of paramount importance. An appropriate way to display this information is in the form of an adjacency list $A(G)$. There is one list $A(v)$ for each vertex v, and $A(v)$ contains those vertices u for which uv is an edge. If G is a multigraph, the loops and multiple edges appear as repetitions in the lists. Thus the adjacency lists for the graphs of 2.12 and 2.13 are

$A(u)$	v, w
$A(v)$	u, w, x
$A(w)$	u, v, x
$A(x)$	v, w, y
$A(y)$	x, z
$A(z)$	y

and

$A(u)$	v, w
$A(v)$	u, v, w, x, x
$A(w)$	u, v, x
$A(x)$	v, v, w, y
$A(y)$	x, z, z, z
$A(z)$	y, y, y

respectively. If G is a digraph, the list $A(v)$ contains u only when (v, u) is an edge, giving

$A(u)$	w
$A(v)$	u, v, w, x
$A(w)$	
$A(x)$	v, w
$A(y)$	x, z
$A(z)$	y, y

for the digraph of 2.14. A diagram $D(G)$ can be constructed as easily from $A(G)$ as from the pair $(V(G), E(G))$, and in stating our algorithms we shall usually assume that it is $A(G)$ which is given. Strictly speaking, $A(G)$ is a list of lists!

Adjacency lists are especially suited to languages such as LISP, *Mathematica*, or C, which handle lists with ease. In FORTRAN, however, the adjacency matrix is more appropriate, displaying the same information as the lists, but in a square array. If $V(G)$ is $\{v_1, \ldots, v_n\}$, the array has n rows and n columns, and the entry in row i and column j (and therefore also in row j and column i) is the number of edges $v_i v_j$ in E. Thus if we label the vertices of G

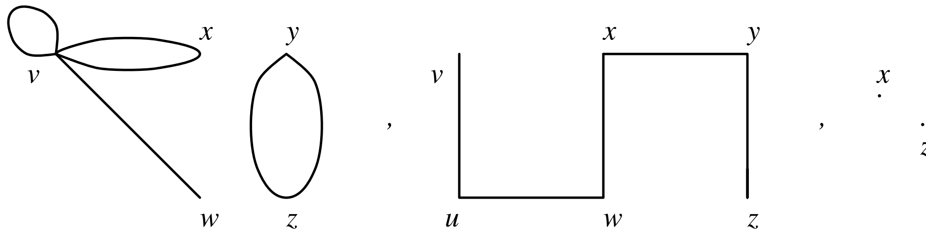
in alphabetical order, we obtain matrices

$$\begin{pmatrix} 0 & 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 \end{pmatrix} \quad \text{and} \quad \begin{pmatrix} 0 & 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 2 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 & 0 \\ 0 & 2 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 3 \\ 0 & 0 & 0 & 0 & 3 & 0 \end{pmatrix}$$

for the graphs of 2.12 and 2.13 respectively. For the digraph of 2.14, the adjacency matrix is modified to take account of the edge directions:

$$\begin{pmatrix} 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 2 & 0 \end{pmatrix}.$$

In studying graphs, we shall repeatedly find ourselves considering some portion F of a given graph G as a graph in its own right. We therefore define a subgraph F of G to be any graph F for which $V(F) \subseteq V(G)$ and $E(F) \subseteq E(G)$. Thus, for example, all three of



are subgraphs of 2.13. If $V(F)$ is all of $V(G)$, as in the second case but neither of the others, we say that F is a spanning subgraph of G.

Suppose that $S \subseteq V(G)$ is a subset of the vertices of G; then the subgraph $G\langle S\rangle$ of G induced by S has vertex set S, and edge set consisting of those edges of G whose endpoints both lie in S. In other words,

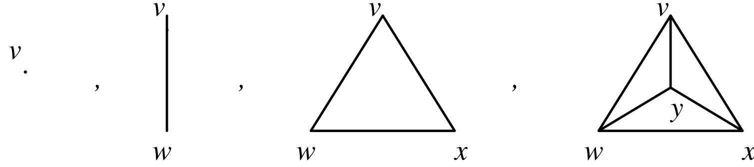
$$V(G\langle S\rangle) = S \qquad \text{and} \qquad E(G\langle S\rangle) = \{st \in E(G) : s, t \in S\}.$$

Example 2.15 *Let S_1 and S_2 be the subsets $\{v, x, z\}$ and $\{u, w, x, y\}$ of the vertices of 2.13; then the respective induced subgraphs are*

$G\langle S_1\rangle \;=$ v x z *and* $G\langle S_2\rangle \;=$ u w x y .

The complete graphs K_n are very important examples. There is one K_n for each integer $n \geq 1$, being a simple graph with n vertices and one edge for every pair of distinct vertices. The first four examples are

Example 2.16



No further edges can be added to these graphs without turning them into multigraphs; in this sense, they are the opposite extreme to null graphs.

2.3 Methodology

In this section we introduce some simple methods for establishing mathematical properties of graphs, and outline the principles behind the algorithmic approach.

Our first result is extremely straightforward yet highly useful. It applies to every multigraph G with p edges.

Proposition 2.17 *The degrees of the vertices of G satisfy*

$$\sum_{v \in V(G)} d(v) = 2p.$$

Proof Every edge uv of G contributes 1 to $d(u)$ and 1 to $d(v)$, and therefore contributes 2 to the sum of all the degrees. □

This reasoning may be extended to digraphs by taking account of indegrees and outdegrees, and yields the following generalisation.

Proposition 2.18 *The degrees of the vertices of a digraph G satisfy*

$$\sum_{v \in V(G)} d_{\text{in}}(v) + \sum_{v \in V(G)} d_{\text{out}}(v) = 2p.$$

An important consequence of 2.17, which we shall use many times in future, requires more careful reasoning. It concerns the vertices of odd degree, so it is convenient henceforth to refer to a vertex as either odd or even, according to the parity of its degree.

Corollary 2.19 *In any multigraph G, the number of odd vertices is even.*

Proof Let G have vertices $\{v_1, \ldots, v_n\}$, ordered so that the odd ones appear first in the list, say as $\{v_1, \ldots, v_m\}$, for some $1 \leq m \leq n$. Applying 2.17, we have that

$$\sum_{i=1}^{m} d(v_i) + \sum_{i=m+1}^{n} d(v_i) = 2p.$$

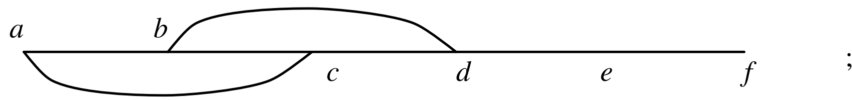
Now the second sum on the left hand side is even since each constituent degree is even by assumption; the right hand side is also even, whence $\sum_{i=1}^{m} d(v_i)$ is even as well. But each constituent degree is *odd*, so m itself must be even, as required. □

By way of illustration, if we refer back to G of 2.12 we observe that it has four odd vertices, namely v, w, x, and z; the multigraph of 2.13, on the other hand, has only the two odd vertices w and z.

At some point, we have to grasp the nettle of formalising what we mean by two graphs being *the same*. For example, we do not wish to distinguish between 2.12 and any of its (possibly very different looking) diagrams; such distinction would negate the whole point of introducing $D(G)$ in the first place! However, there *is* a serious problem here, since it may be very unclear when two large adjacency lists, or two intricate diagrams, actually do represent graphs with identical mathematical structure. For precision, we are therefore forced to make a definition such as the following. Two graphs G and H are called isomorphic if their vertices may respectively be labelled as $\{v_1, \ldots, v_n\}$ and $\{w_1, \ldots, w_n\}$ so that v_i and v_j are adjacent in G if and only if w_i and w_j are adjacent in H for all pairs $1 \leq i,\ j \leq n$. In such circumstances, the mathematical properties of G and H are indistinguishable.

Example 2.20 *Let H have vertices $\{a, b, c, d, e, f\}$ and edges $\{ac, ed, ba, bc, bd, ef, cd\}$; then* 2.12 *and H are isomorphic by listing the respective vertex sets in alphabetical order.*

Here is a diagram for H



;

it may not be clear to the casual observer that this diagram and the diagram of 2.12 represent isomorphic graphs.

A deeper study of isomorphisms will not concern us here, although an entire course could be constructed around related issues. Suffice it to say that we now have a consistent logical justification for referring to any diagram representing a graph, or any other format for the graph, as the graph itself. We shall appeal to this principal so often whilst setting up and running our algorithms that it will become second nature and fade into the background. In fact the concept of isomorphism is one of the universal themes of modern mathematics; whenever a general notion is defined, it is extremely important to specify when two examples of it are to be treated as identical.

We shall investigate many properties of graphs by establishing a sequence of instructions which need to be repeated, possibly many times, before terminating and outputting the relevant information. The foundations underlying a serious study of such sequences lie in the realms of mathematical logic, and we shall content ourselves here with an informal description which seems adequate for our purposes. We therefore assert that an algorithm consists of a finite sequence of steps, or unambiguous instructions, with the following two properties

- included in each step is an instruction for which step to perform next
- the instruction to stop is always reached after a finite number of steps;

in addition, the algorithm has an input consisting of a description of the type of graph to which it applies, and an output which is information concerning the input.

Here is an example of an algorithm (from beyond the realms of graph theory).

Algorithm 2.21 (Breakfast Eggs: BE)
INPUT: *hob and pan of water, large supply of eggs, one eggcup for each diner*
OUTPUT: *one boiled egg for each diner*

(1) *turn on hob, and bring pan of water to boil*

(2) *if no eggcup is empty, go to* (5)

(3) *add one egg to water, and boil for 3 minutes*

(4) *remove egg, place in empty eggcup, and go to* (2)

(5) *turn off hob, and stop*

This algorithm has at least one defect, and readers might like to try and improve its performance!

An algorithm is intended to be a form of instructions which can easily be turned into a computer program *if so desired*. We shall not pursue this option here, although it may readily be explored by readers interested in software. Rather, we shall follow the philosophy that to give an algorithm for the solution of a mathematical problem is to understand the problem in the clearest and most constructive fashion. This belief has a long and distinguished pedigree, and can be seen in action in many other areas of mathematics, as exemplified by the Euclidean algorithm in elementary number theory.

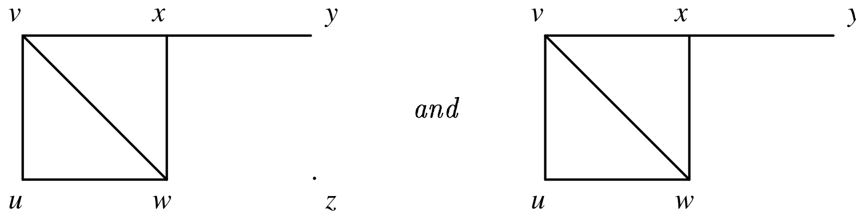
We emphasise that it is not usually sufficient simply to give an algorithm and claim that it performs a certain function. Unless this is clearly an immediate logical consequence of the stated steps, and because we are mathematicians, we have to *prove* that the algorithm performs as claimed.

When running an algorithm, we may wish to assign values to various entities which may (or may not!) change as the algorithm proceeds. There are several possibilities. For example, an integer might be assigned to some counter, so as to increase its value by 1 each time a certain step is performed; or a real number might be assigned to a label for some vertex or vertices; or a set of edges might be assigned to some variable which builds up a subgraph of the input. In every case we indicate such assignments with the symbol $:=$. In consequence, an apparently contradictory statement such as $i := i+1$ (which appears in our first algorithm BFS in Section 3.1 below) simply means that the current value i of the counter is to be increased by 1. It will often be helpful, as in our example 3.10 below, to use the abbreviation lab( ) for a numerical label attached to a vertex or edge of the input graph.

With these few provisos, we have tried to write our algorithms in the simplest and most accessible language that is logically correct; expert programmers may therefore find them irritatingly naïve.

Typical sequences of instructions may involve the systematic modification of an input graph, for example by removing edges one at a time. Such procedures require a modicum of care, both in their description and their execution, so we describe the details here. Given a graph G and an edge e, we shall write $G \setminus e$ to denote the graph whose vertex set remains unaltered, but whose edge set has e deleted. Thus $G := G \setminus e$ expresses this procedure as an algorithmic instruction. One or two of the vertices of G (but no more) may become isolated as a consequence of the deletion, and in certain circumstances we may wish to remove these vertices as well; we write $G \setminus\setminus e$ to denote the result. If the deletion of e does *not* create isolated vertices, then the two graphs $G \setminus e$ and $G \setminus\setminus e$ are identical. By way of illustration, we refer back to the graph G of 2.12.

Example 2.22 *The instructions* $G := G \setminus yz$ *and* $G := G \setminus\!\setminus yz$ *modify* G *as*



respectively.

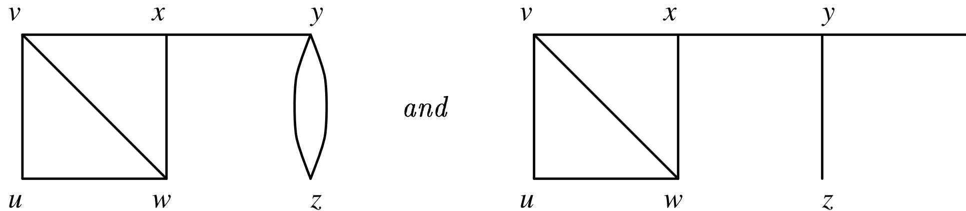
An algorithm incorporating such instructions is provided by PC1 in Section 4.2 below.

The same notation may be extended to cover a *set* of edges, so that, for example, $G \setminus E(G)$ is the null graph on vertices $V(G)$, whereas $G \setminus\!\setminus E(G)$ is empty whenever G contains no isolated vertices.

Likewise, our algorithmic steps will often *add* edges to G systematically. Sometimes this will involve only vertices which are already members of $V(G)$, but on other occasions we may need to introduce up to two new vertices as well. In either case we shall write $G \cup uv$ to describe the graph whose vertex set is suitably enlarged (if necessary), and whose edge set contains the single extra edge uv.

We refer again to 2.12.

Example 2.23 *The instructions* $G := G \cup yz$ *and* $G := G \cup ty$ *modify* G *as*



respectively.

An algorithm incorporating such instructions is provided by PC2 in Section 4.3 below.

In the extreme case, we may start with the empty set $\emptyset$, and build up G to be arbitrarily large by a sequence of edge additions. This notation may again be extended to cover a set of added edges.

Occasionally we will have to delete a subset W of the *vertices* of G. The resulting graph will be denoted by $G \setminus W$, on the understanding that its edge set is also modified by the removal of all edges incident on any of the deleted vertices.

Chapter 3

Connectedness

3.1 Distance in Graphs

In this section we address the problem of deciding when a graph consists of several disconnected parts.

A path π of length n in a graph G consists of an alternating sequence of *distinct* vertices and edges

$$v_0, e_1, v_1, e_2, \ldots, e_n, v_n, \tag{3.1}$$

where each e_i is $v_{i-1}v_i$; we often write the length of π as $\ell(\pi)$. We call v_0 the beginning of π and v_n the end of π, and say that π is a path from v_0 to v_n. If we insist that π has the same beginning and end, we refer to it as a cycle.

So long as G contains no multiple edges, we may reduce (3.1) to the list of vertices

$$v_0, v_1, \ldots, v_n \tag{3.2}$$

and still determine π.

We shall need certain simple properties of paths in G.

Lemma 3.3 *If there is a path from v to u in G, there is also a path from u to v; if there are paths from v to u and u to w, then there is also a path from v to w.*

Proof For the first statement, we simply take the given path π as in (3.1), and reverse the order of all the symbols. For the second statement, we are given paths

$$\pi_1 = v, e_1, v_1, e_2, \ldots, e_m, u \qquad \text{and} \qquad \pi_2 = u, f_1, u_1, f_2, \ldots, f_n, w$$

from v to u and u to w respectively. Let v_i be the *first* vertex of π_1 which also lies in π_2; it may therefore also be written as u_j for some j (such a vertex must exist, since u is an example). Then

$$\pi = v, e_1, v_1, e_2, \ldots, e_i, u_j, f_{j+1}, \ldots, f_n, w$$

is a path from v from to w, as required. $\square$

Corollary 3.4 *If there are paths from u to v and u to w in G, then there is also a path from v to w.*

Proof Use the first statement of 3.3 obtain a path from v to u, and then the second statement to obtain the path from v to w. $\square$

We say that G is connected if there is a path from v to w for every pair of vertices v, w in G. If G fails to be connected, we say that it is disconnected.

Given two vertices u and v in G, there may be several different paths between them. Motivated by the standard planar concept of the straight line, we therefore define the distance between u and v in G to be

$$d(u,v) = \min_{\pi} \ell(\pi), \tag{3.5}$$

where π ranges over all paths between u and v. There may be more than one such path whose length is $d(u,v)$. If there is *no* path between u and v, we insist that $d(u,v)$ is ∞.

We may now state that G is connected if and only if $d(v,w)$ is finite (written $d(v,w) < \infty$) for all pairs of vertices v, w in G. We may also deduce from 3.3 and 3.4 that

$$d(u,v) = d(v,u), \qquad \left.\begin{array}{r} d(v,u) < \infty \\ d(u,w) < \infty \end{array}\right\} \Rightarrow d(v,w) < \infty \tag{3.6}$$

and

$$\left.\begin{array}{r} d(u,v) < \infty \\ d(u,w) < \infty \end{array}\right\} \Rightarrow d(v,w) < \infty \tag{3.7}$$

respectively.

Proposition 3.8 *For any fixed vertex u in G, we have that G is connected if and only if $d(u,v) < \infty$ for all v in G.*

Proof Suppose first that G is connected. By definition, $d(u,v) < \infty$ for all v in G.

Conversely, suppose we are given $d(u,v) < \infty$ for all v in G, and select any pair of vertices v, w in G. Then $d(u,v) < \infty$ and $d(u,w) < \infty$, so (3.7) applies to tell us that $d(v,w) < \infty$, whence G is connected. $\square$

Note that everything we have described so far is valid in the presence of loops and multiple edges; for example a loop at any vertex v is a cycle of length 1.

Our strategy to determine whether or not a graph is connected is now clear. We fix a beginning vertex (or root) u in G, and measure the distance between u and all other vertices; if every vertex has finite distance from u, then G must be connected by 3.8,.

We can perform this task efficiently with our first algorithm, which answers Question 2.1.

Algorithm 3.9 (Breadth First Search: BFS)
INPUT: *a graph G with root vertex u*
OUTPUT: *d(u,v) for all v in G*

(1) $r := u$

(2) $i := 0$ *and label r with i*

(3) *find all unlabelled vertices v adjacent to a vertex with label i; if none, go to* (6)

(4) *label all vertices v of* (3) *with $i + 1$*

(5) $i := i + 1$ *and go to* (3)

(6) *label all unlabelled vertices with* ∞

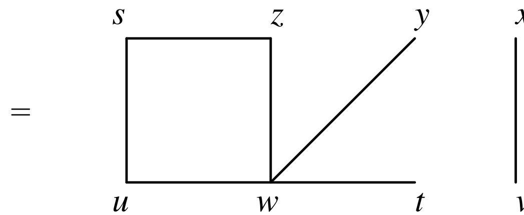
(7) *output the labelled vertices, and stop*

We shall prove in Theorem 3.11 below that when BFS stops, every vertex of G is labelled with its distance from u; thus any occurrence of the label ∞ in the output indicates that G is disconnected. Note that BFS performs just as well for multigraphs as for simple graphs.

First we give an example of BFS in action, making a few self-explanatory abbreviations as we proceed.

Example 3.10 *Run* BFS *on the following graph* G*:*

$A(s)$	u, z
$A(t)$	w
$A(u)$	s, w
$A(v)$	x
$A(w)$	t, u, y, z
$A(x)$	v
$A(y)$	w
$A(z)$	s, w



Solution

(1) $r := u$ 1

(2) $i := 0$ and $\text{lab}(u) := 0$ 2

(3) adjacent unlabelled vertices are s, w 3

(4) $\text{lab}(s) := 1$ and $\text{lab}(w) := 1$ 4

(5) $i := 1$ and go to (3) 5

(3) adjacent unlabelled vertices are z, t, y 6

(4) $\text{lab}(z) := 2$, $\text{lab}(t) := 2$ and $\text{lab}(y) := 2$ 7

(5) $i := 2$ and go to (3) 8

(3) no adjacent unlabelled vertices; go to (6) 9

(6) $\text{lab}(v) := \infty$ and $\text{lab}(x) := \infty$ 10

(7) OUTPUT: 11

s	t	u	v	w	x	y	z
1	2	0	∞	1	∞	2	2

We deduce that our graph is disconnected.

In fact it is possible to give a summary of the above information in terms of *search trees*. We shall describe this approach in Section 3.2.

We now verify that BFS does indeed label the vertices of G as claimed. In giving the proof, it is notationally simpler to take advantage of (3.2), and assume that G has no loops or multiple edges; it is straightforward to provide the extra details for a general multigraph.

Theorem 3.11 *When* BFS *stops, every vertex v of G is labelled with the distance $d(u,v)$.*

Proof Run BFS on G, with root u. Notice first that every vertex acquires *some* label, by step (6). Fix v, and assume that $\text{lab}(v) = n$, which may be ∞.

Assume further that $d(u,v) = m$, where $m < \infty$. There is therefore a shortest path from u to v of the form

$$u, u_1, \ldots, u_k, \ldots, u_{m-1}, v.$$

At the first application of step (4), we have that $\text{lab}(u_1) = 1$. At the second application of step (4) we have that $\text{lab}(u_2) = 2$, unless it had already been labelled with 1 at the first application; thus $\text{lab}(u_2) \leq 2$. Proceeding in this way, we see that at the kth application of step (4), $\text{lab}(u_k) \leq k$ for all $1 \leq k \leq m$. In particular, at the mth application we obtain $n \leq m$. If $n = \infty$ this inequality is impossible, so there is *no* shortest path from u to v, and v is indeed in a different component, as required.

On the other hand, and assuming now that $n < \infty$, at the nth application of step (3) we utilise a vertex v_{n-1} which is adjacent to v and has $\text{lab}(v_{n-1}) = n-1$. At the $(n-1)$th application of step (3) we utilise a vertex v_{n-2} adjacent to v_{n-1} and with $\text{lab}(v_{n-2}) = n-2$, and in general, at the kth application, a vertex v_{k-1} which is adjacent to v_k and has $\text{lab}(v_{k-1}) = k-1$ for all $n \geq k \geq 1$. In particular, at the first application we utilise a vertex v_0 which is adjacent to v_1 and has label $\text{lab}(v_0) = 0$; by step (1), v_0 is therefore u. In this way, we have constructed a path

$$u, v_1, \ldots, v_k, \ldots, v_{n-1}, v$$

of length n from u to v. Thus by (3.5), $m \leq n$.

Hence $m = n$, as desired. □

Corollary 3.12 *The graph G is connected if and only if* BFS *does not assign the label ∞ to any vertex.*

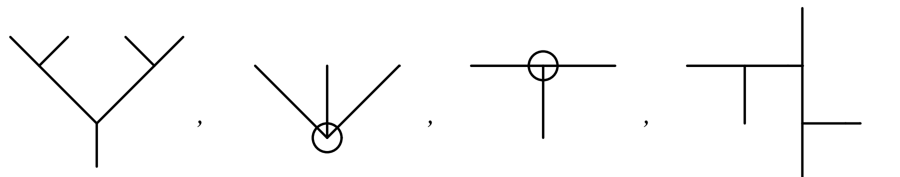
Proof Combine 3.11 with Proposition 3.8. □

3.2 Spanning Trees

In this section we investigate some fundamental ideas concerning trees, motivated by the search procedures of Section 3.1.

Formally, a tree is a connected graph which contains no cycles. Following our conventions of Section 3.1, we refer to any distinguished vertex of a tree as a root. Here are some examples of trees with and without roots: readers should ask themselves which (if any!) are isomorphic.

Example 3.13



As these diagrams suggest, a tree T will always have a certain number of vertices of degree 1; these are called the leaves of T, and we shall explain below how their existence is assured. It may not be obvious from the adjacency list of T that it is a tree.

We begin with a useful alternative characterisation.

Proposition 3.14 *A graph with n vertices and p edges is a tree if and only if it is connected and $p = n - 1$.*

Proof Firstly, assume that we are given a tree T with n vertices and p edges. It is certainly connected, and we shall prove that $p = n - 1$ by induction on n. The base case is $n = 1$, which is clearly true by appeal to 3.13! To make the inductive step, we take as our hypothesis that whenever a tree T has n vertices for any $n \leq k$, then it has $n - 1$ edges. So assume now that T has $k + 1$ vertices, and remove one of its edges e. We are left with two trees, having n and $k + 1 - n$ vertices respectively, where both these numbers lie between 1 and k; we may therefore deduce from our hypothesis that they have $n - 1$ and $k - n$ edges respectively. Thus the original tree had

$$(n - 1) + (k - n) + 1 = k$$

edges, remembering to add back e. This is $(k + 1) - 1$, as required. Hence $p = n - 1$ for all n, by induction.

Secondly, assume that a connected graph G with n vertices and p edges satisfies $p = n - 1$. We shall prove that it is a tree by induction on n. The base case $n = 1$ is again clear from 3.13. To make the inductive step, we take as our hypothesis that whenever G has k vertices and $k - 1$ edges, then it is a tree. So assume now that G has $k + 1$ vertices $\{v_1, \ldots, v_{k+1}\}$ and k edges, whence

$$\sum_{i=1}^{k+1} d(v_i) = 2k \tag{3.15}$$

by appeal to Proposition 2.17. This is only possible if there are at least *two* leaves, otherwise the sum would exceed $2k$. So remove one of these leaves, and its single incident edge e, from G. The result must still be connected, and has k vertices and $k - 1$ edges; we therefore deduce from our hypothesis that it is a tree. Restoring e cannot make it disconnected, and creates no cycles, so G is also a tree, as required. Hence our statement is true for all n, by induction. □

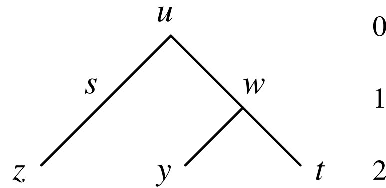
We may prove from (3.15) that a tree with n vertices not only has at least two leaves, but also at most $n - 1$.

If G is a connected graph, a spanning tree for G is a subgraph T which contains all the vertices of G and is itself a tree. We now explain how to construct a spanning tree $T(G)$ for any such graph, by running BFS; indeed, this construction is one of the most important applications of the algorithm.

The idea is to begin with the root u of G, and use it for the root of $T(G)$. We adjoin to u each edge uv which is found by step (3) of BFS. We then repeat the procedure using the vertices v, and iterate. The process can easily be incorporated into BFS itself, so that $T(G)$ is part of the output. For obvious reasons, $T(G)$ is known as a search tree in G. The simplest way to understand how it is constructed is to recall Example 3.10, and to display the corresponding spanning tree for the relevant subgraph of G. In fact this tree contains all the information involved in solving 3.10, and is an ideal way to summarise the execution of the

algorithm. Note that we have recorded the output distances from u in terms of the depth of the vertices in the tree.

Example 3.16



Several different spanning trees for G may be created by this method, since each implementation of BFS may involve choice in the order of labelling vertices.

The question now arises as to how to proceed when G is disconnected, since BFS will always label at least one vertex with ∞ in this case. To provide an answer, we need first to understand how an arbitrary multigraph decomposes in a unique way as a disjoint union of its components.

Given a particular vertex u, we define the component of u in G to be the induced subgraph

$$G_u = G\langle v : d(u,v) < \infty \rangle \tag{3.17}$$

Alternatively, G_u consists of all vertices v for which there exists a path from u to v in G, and all edges which belong to such paths. In the light of (3.17), we may immediately apply Proposition 3.8 to G_u, and deduce that each component is itself a connected subgraph of G.

We need one fundamental property of components in order to proceed.

Lemma 3.18 *Two components G_u and G_v are either equal, or else completely disjoint.*

Proof First suppose that $d(u,v) < \infty$ (in other words, that $u \in G_v$ and $v \in G_u$). For every $w \in G_u$ we have that $d(u,w) < \infty$, so $d(v,w) < \infty$ by applying (3.7). Thus $w \in G_v$, and we have established that $G_u \subseteq G_v$. By initially choosing $w \in G_v$, we may similarly establish that $G_v \subseteq G_u$. Hence $G_u = G_v$.

Now suppose that $d(u,v) = \infty$, and select w satisfying $d(u,w) < \infty$ as before. This time we must have $d(v,w) = \infty$; for if $d(v,w) < \infty$, then by applying (3.7) again (but with the rôle of the vertices interchanged) we would obtain $d(u,v) < \infty$. Thus no vertex of G_u can also be a vertex of G_v, and the components are disjoint. □

Corollary 3.19 (The Component Principle) *Any multigraph may be uniquely decomposed into a finite number of components.*

As an example of this principal, we remark that any graph which contains no cycles must in fact be a disjoint union of trees; we refer to such a graph as a forest. The following useful observation follows immediately from 3.14.

Lemma 3.20 *A forest on n vertices contains more than one tree if and only if it has less than $n-1$ edges.*

Proof Such a forest contains more than one tree if and only if it can be turned into a single tree by adding extra edges. □

We should emphasize that the properties we have verified above are transparently true for the components of a small graph which is represented by its diagram. The purpose of our proofs is to give the statements logical foundation for graphs which are of arbitrary size (such as in Picture 1 of Section 2.2) and which may be presented in the form of adjacency lists.

Given a multigraph, we may now seek to describe its constituent components; this is a more sophisticated goal than simply determining whether or not it is connected. We shall define an improved version of BFS which performs the task algorithmically. Once it has run on the component of the original root, it selects a root in another component and runs again; it repeats this procedure until all the components are exhausted, thereby assigning to every vertex a label less than ∞. As we have seen, 3.19 ensures that the choice of roots is immaterial to the construction of the components.

Algorithm 3.21 (Extended Breadth First Search: EBFS)
INPUT: *a graph G with root vertex u*
OUTPUT: *a root r for each component of G, and $d(r,v)$ for all v in the same component as r*

(1) $r := u$

(2) $i := 0$ *and label r with i*

(3) *find all unlabelled v adjacent to a vertex with label i; if none, go to* (6)

(4) *label all vertices v from* (3) *with $i+1$*

(5) $i := i+1$ *and go to* (3)

(6) *find an unlabelled vertex w, then $r := w$ and go to* (2)*; if none, go to* (7)

(7) *output the labelled vertices, and stop*

We illustrate the performance of EBFS.

Example 3.22 *Run* EBFS *on the graph G of* 3.10.

Solution The first 8 steps are the same as in 3.10. We take up the action at step 9.

(3) no adjacent unlabelled vertices; go to (6) 9

(6) v is unlabelled, so $r := v$ and go to (2) 10

(2) $i := 0$ and $\text{lab}(v) := 0$ 11

(3) adjacent unlabelled vertex is x 12

(4) $\text{lab}(x) := 1$ 13

(5) $i := 1$ and go to (3) 14

(3) no adjacent unlabelled vertices; go to (6) 15

(6) no unlabelled vertices; go to (7) 16

(7) OUTPUT:

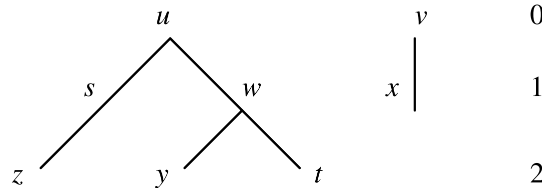
s	t	u	v	w	x	y	z
1	2	0	0	1	1	2	2

17

The proof that EBFS attaches the required labels is a simple extension of Theorem 3.11, applying it to each component of G in turn. Note that we can read off the number of components by counting the number of roots; that is, the number of vertices whose labels are 0. We have now answered Question 2.2.

In parallel with the connected case, a concise way to summarise the output of EBFS is to give a spanning tree for each component of G in turn. The result is known as a spanning forest for G, and its existence is assured simply by running the algorithm.

By way of illustration, we may display the results of 3.22 as the following spanning forest:



As in the connected case, the construction of the search forest may itself be incorporated into EBFS.

3.3 Depth First Search

In this section we examine an alternative method for searching through the vertices of a multigraph; the procedure is also of great importance in its own right.

In fact the resulting labels provide an ordering of the vertices, and we need to begin by saying a few words about precisely what this means. Suppose that there are n elements in the vertex set $V(G)$ of our graph G. In order to store G in a machine, we have to use either adjacency lists or the adjacency matrix, as defined in Section 2.2. Both these methods assume implicitly that we have already chosen an order for the elements of $V(G)$; either the order in which the vertices are listed, or else the order in which they label the rows of the matrix, respectively.

Often this ordering will be chosen in purely arbitrary fashion by the operator who feeds the information into the computer. Given the same graph and the same computer, a different operator may select a different ordering. Indeed, there are $n!$ possible such orderings in total. For many purposes it is therefore important to consider *reordering* the vertices in a way which is directly related to some structural properties of the graph. We shall describe here an algorithm which does this explicitly, as a byproduct of searching through $V(G)$; the nature of the search is usually very different from that of BFS.

We shall give a *recursive* version of the algorithm, involving a subsidiary procedure which is repeatedly recalled whilst it is running. This simplifies the statement, but makes the effect of the algorithm more subtle to understand.

Algorithm 3.23 (Depth First Search: DFS)
INPUT: *a graph G with n vertices, and root vertex u*
OUTPUT: *a total ordering of the vertices in G_u, with u as the first element*

(1) $i := 1$ *and* $r := u$

(2) *do* DFS(r)

(3) *output the labelled vertices, and stop*

where DFS(r) *is*

(1) *label* r *with* i*; then* $i := i + 1$

(2) *for all unlabelled* v *adjacent to* r*, do* DFS(v)

As we shall see below in Theorem 3.25, it is simple to prove that when DFS stops, every vertex of G_u is assigned a unique label between 1 and n. In particular, DFS is an alternative method of testing for the connectivity of G, since the existence of an unlabelled vertex implies that $G_u \neq G$.

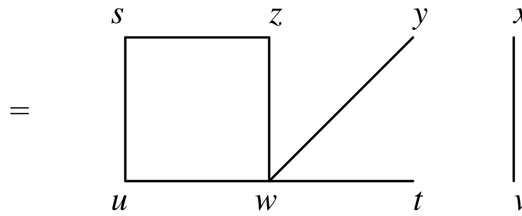
We should emphasise that the vertex ordering produced by DFS is not unique, since certain choices have still to be made during implementation. In general, however, the number of possibilities is significantly less than $n!$.

First we give an example of DFS in action, again making a few self-explanatory abbreviations as we proceed.

Example 3.24 *Run* DFS *on the graph* G

$A(s)$	u, z
$A(t)$	w
$A(u)$	s, w
$A(v)$	x
$A(w)$	t, u, y, z
$A(x)$	v
$A(y)$	w
$A(z)$	s, w

=



of 3.10.

Solution

(1) $i := 1$, $r := u$ — 1

(2) do DFS(u) — 2

  (1) lab(u) := 1; then $i := 2$ — 3

  (2) $w \in A(u)$ is unlabelled, so do DFS(w) — 4

    (1) lab(w) := 2; then $i := 3$ — 5

    (2) $y \in A(w)$ is unlabelled, so do DFS(y) — 6

      (1) lab(y) := 3; then $i := 4$ — 7

      (2) all elements of $A(y)$ are labelled — 8

    (2) $t \in A(w)$ is unlabelled, so do DFS(t) — 9

      (1) lab(t) := 4; then $i := 5$ — 10

      (2) all elements of $A(t)$ are labelled — 11

(2) $z \in A(w)$ is unlabelled, so do DFS(z) 12

(1) lab(z) := 5; then i := 6 13

(2) $s \in A(z)$ is unlabelled, so do DFS(s) 14

(1) lab(s) := 6; then i := 7 15

(2) all elements of $A(s)$ are labelled 16

(2) all elements of $A(z)$ are labelled 17

(2) all elements of $A(w)$ are labelled 18

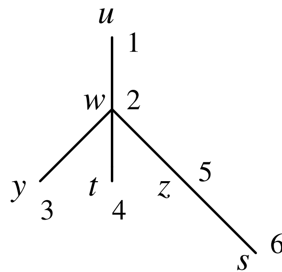
(2) all elements of $A(u)$ are labelled 19

(3) OUTPUT: 20

s	t	u	v	w	x	y	z
6	4	1		2		3	5

We deduce that our graph is disconnected.

We may give a more concise display of the above information in terms of a DFS search tree, as follows.



We now verify that DFS does indeed label the vertices of G as claimed. As with BFS the proof is notationally simpler if we assume that G has no loops or multiple edges; it is straightforward to provide the extra details for a general multigraph.

Theorem 3.25 *When* DFS *stops, the vertices of* G_u *are ordered by their labels.*

Proof Run DFS on G, with root u. By DFS(r) step (2), every vertex which acquires a label is in the same component as u. So let v be an arbitrary vertex in G_u, from which we deduce the existence of some path

$$u, u_1, \dots, u_k, \dots, u_{n-1}, v.$$

Now DFS(u_1) runs within DFS(u), and so labels u_1 at step (1). Similarly, DFS(u_2) runs within DFS(u_1) *unless* it already ran within DFS(u); either way, u_2 acquires a label at step (1). In general, DFS(u_k) runs within one of DFS(u), DFS(u_1),..., DFS(u_{k-1}) for each $1 \leq k \leq n$, so that u_k acquires a label at step (1). In particular, v acquires a label before DFS stops. □

Corollary 3.26 *The graph* G *is connected if and only if* DFS *orders every vertex of* G.

Proof Combine 3.25 with Proposition 3.8. □

We conclude with a modified version of DFS, which selects a new root for each component of a disconnected graph G and then continues ordering the vertices within the component of that root. As a result, all the vertices in G are eventually ordered.

Algorithm 3.27 (Extended Depth First Search: EDFS)
INPUT: *a graph G with root vertex u*
OUTPUT: *a total ordering of the vertices in G, with u as the first element*

(1) $i := 1$ *and* $r := u$

(2) *do* DFS(r)

(3) *if some v is unlabelled, $r := v$ and go to* (2)*; otherwise, go to* (4)

(4) *output the labelled vertices, and stop*

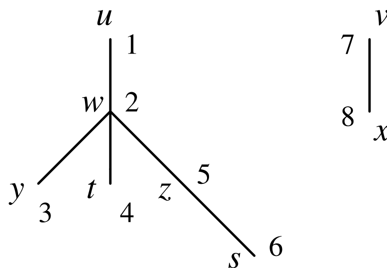
Example 3.28 *Run* EDFS *on the graph G of* 3.10.

Solution The first 18 steps are the same as in 3.24. We take up the action at step 19.

Step	
(2) all elements of $A(u)$ are labelled	19
(3) v is unlabelled, so $r := v$ and go to (2);	20
(2) do DFS(v)	21
(1) lab(v) := 7; then $i := 8$	22
(2) $x \in A(v)$ is unlabelled, so do DFS(x)	23
(1) lab(x) := 8; then $i := 9$	24
(2) all elements of $A(x)$ are labelled	25
(2) all elements of $A(v)$ are labelled	26
(3) all vertices of G are labelled, so go to (4)	27
(4) OUTPUT: (see table below)	28

s	t	u	v	w	x	y	z
6	4	1	7	2	8	3	5

We may display the results of 3.28 in terms of the following spanning forest:



As with BFS and EBFS, the construction of the depth first search trees and forests may be incorporated into the algorithms themselves.

Chapter 4

The Enumeration of Trees

4.1 Labelled Trees

Suppose we are given n vertices $\{v_1, \ldots, v_n\}$, and wish to count the number of possible ways in which they can be joined to form a tree; we have already introduced this problem as the Spanning Trees Question 2.8. The development of counting procedures to answer such questions is known in combinatorial mathematics as enumeration theory.

In fact the problem was solved over a century ago (before the advent of computer networks!), for reasons that remain extremely interesting. During the 1850s and 1860s, chemists were beginning to develop the notion of the molecule as an aggregation of atoms, and the Scotsman Alexander Crum Brown proposed a 'graphical notation' to indicate the bonds involved. The following diagram is taken from his original paper of 1864, and is a startling indication of how the radical ideas of one generation can become basic tenets of another:

Diagram 4.1

But the diagram is of importance for another reason; it illustrates the chemical phenomena of *isomerism.* The two substances illustrated have the composition C_3H_7OH, yet according to Crum Brown the first molecule is of propylic alcohol and the second of Friedel's alcohol. The atoms are the same, but the bonds (and some of the properties of the alcohols) are different.

This example clearly indicates the need to enumerate spanning trees, and the chemists quickly turned to mathematicians for assistance. Amongst those who responded were the two English mathematicians (and friends) Arthur Cayley and J J Sylvester. The former answered

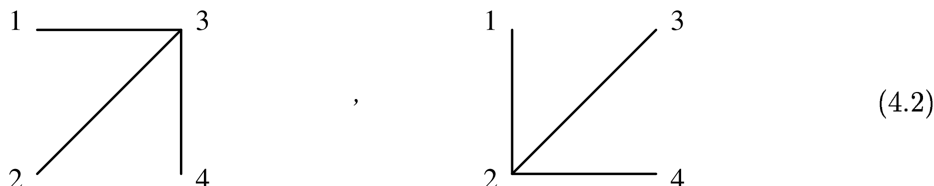
Question 2.8 in 1889, whilst the latter, writing on a related mathematical topic in 1877, had already derived the term 'graph' from Crum Brown's notation.

We shall explain Cayley's Theorem by appealing to a method due to the German mathematician Heinz Prüfer. Although his work was not published until 1918, it appears that he was unaware of the existence of Cayley's proof. We prefer Prüfer's approach since it may be expressed in terms of two beautiful algorithms, involving the construction of a *code* for each spanning tree. It is amusing to note that Prüfer justified his investigations by quoting the result (which we have blanked out, to maintain the suspense) in the following terms:

> *Consider a country with n towns. These towns must be connected by a railway network of* $n-1$ *lines (the smallest possible number) in such a way that one can get from each town to every other town. There are* ♣♣ *different railway networks of this kind.*

Implicit within this statement is the understanding that whenever n vertices are connected by $n-1$ edges, the corresponding graph is a spanning tree; we have already established this point in 3.14.

We must emphasise that the problem under consideration concerns a given set of vertices. They are therefore labelled, for example by the name of the building which will house the computer, or the town which will receive the rail link, and these labels cannot be changed during the course of the enumeration. Thus if n is 4, the two spanning trees



(4.2)

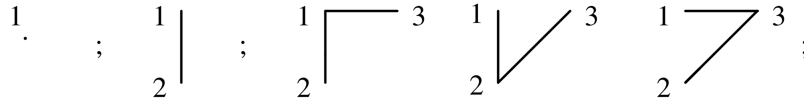
represent different LANs, or different rail networks, even though the underlying trees are actually isomorphic to each other by virtue of the mapping

$$1 \mapsto 1, \quad 2 \mapsto 3, \quad 3 \mapsto 2 \quad \text{and} \quad 4 \mapsto 4,$$

which may be interpreted as a change of labels. Enumerating trees *without* distinguishing between isomorphic possibilities (in other words by disregarding the labelling of the vertices) is an altogether more difficult proposition. It clearly gives rise to a smaller number of trees, and has a separate physical motivation; we shall mention it again only in passing.

An important alternative perspective on 2.8 is provided by joining *all* pairs of vertices $\{v_1, \ldots, v_n\}$ before we start, so obtaining a complete graph K_n. Asking to enumerate its spanning trees is then equivalent to 2.8, and we may restate our goal as being the construction of an enumerating function $t(n)$, whose value for each positive integer n is the number of spanning trees of K_n. In this form, the problem may immediately be generalised by seeking to enumerate the spanning trees of an arbitrary graph G, which need no longer be complete. This too is a classic problem with a beautiful solution, but we avoid it here since it requires some awkward algebra concerning the evaluation of determinants; disappointed readers may be consoled by the fact that the discovery of the formula for $t(n)$ was a defining moment in 19th century graph theory.

We begin our analysis with some examples, which act as a signpost to the final form of Cayley's Theorem. By observation, we list all the trees on 1, 2, 3, and 4 vertices respectively, as follows:



We have thus obtained some preliminary experimental data concerning $t(n)$, which we display as

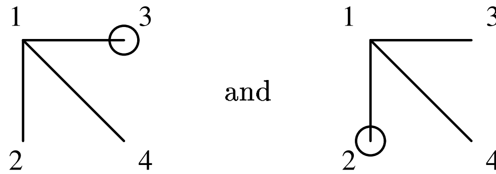
Table 4.3

n	1	2	3	4
$t(n)$	1	1	3	16

This does not offer many hints for deducing the nature of $t(n)$, and it is an instructive exercise for readers to try and evaluate $t(5)$ and $t(6)$.

Notice that, if we do not distinguish between isomorphic trees, the relevant values reduce to 1, 1, 1, and 2.

When studying search trees in Chapter 3, we saw that a tree may naturally be equipped with a root, for example the vertex at which the search begins. In the context of communication networks it is equally plausible that we might wish to distinguish a node; for example the housing for the fileserver in a LAN, or the site of the signalling controls in a railway network. We may therefore extend our enumeration problem to the consideration of rooted trees, on the understanding that two identical trees with different roots are now to be considered as distinct. Thus



are distinct as rooted trees, and altogether there are 4 possibilities based on the same underlying tree.

In fact each tree with n vertices gives rise to n possible rooted trees, since any one of the vertices may be chosen as the root. Therefore, if we modify 2.8 so as to ask for the number of *rooted* trees which span K_n, we may deduce that the answer is $nt(n)$ for every $n \geq 2$, without yet knowing how to evaluate $t(n)$ itself.

4.2 From Trees to Prüfer Codes

We now explain how to encode each of our trees as a sequence. For notational convenience, we assume that the given vertex set consists of the integers $\{1, \ldots, n\}$.

We describe a sequence s of integers $(s_1, s_2, \ldots, s_{n-2})$ as a *Prüfer code on* n (where $n > 2$) whenever it satisfies $1 \le s_i \le n$ for every $1 \le i \le n-2$; thus $(3,1,1,4,4,4)$ and $(1,4,3,4,1,4)$ are two distinct Prüfer codes on 8. This terminology will help us to describe our algorithms succinctly.

Algorithm 4.4 (Prüfer Code 1: PC1)
INPUT: *a tree* T *with vertex set* $\{1, \ldots, n\}$, *where* $n > 2$
OUTPUT: *a Prüfer code* $\mathrm{PC1}(T)$ *on* n

(1) $i := 1$

(2) $j :=$ *smallest leaf of* T, *and suppose the incident edge is* $e = jk$

(3) $s_i := k$ *and* $T := T \backslash\backslash e$

(4) *if* $i = n-2$, *output* $s = \mathrm{PC1}(T)$ *and stop; otherwise,* $i := i+1$ *and go to* (2)

Some explanations are required. Firstly, in step (2) we are employing (3.15) to remind us that every tree has two or more leaves, and selecting the one whose vertex has the smallest integer label. Secondly, in step (3), we are recalling the notation of Section 2.3 to indicate that both the edge jk *and* the vertex j are deleted from T, thereby creating a new and smaller tree in which the vertex k may or may not be a leaf. Thirdly, the output is indeed a Prüfer code, since all the entries lie between 1 and n by construction, and it is unique, since no choice is involved at any stage of the implementation.

Notice further that the leaves of T are precisely the integers which do *not* appear in $\mathrm{PC1}(T)$, and that the algorithm stops when T has been reduced to two vertices and a connecting edge.

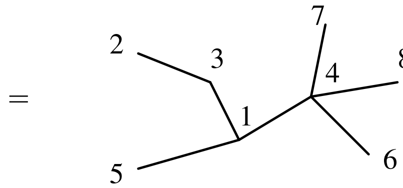
If $n = 2$, we take the Prüfer code of the unique tree to be empty; this is equivalent to running step (4) only.

The basic simplicity of PC1 is best illustrated by an example. As usual, we introduce some self-explanatory abbreviations.

Example 4.5 *Run* PC1 *on the following tree* T*:*

$A(1)$	$3,4,5$
$A(2)$	3
$A(3)$	$1,2$
$A(4)$	$1,6,7,8$
$A(5)$	1
$A(6)$	4
$A(7)$	4
$A(8)$	4

=



Solution

(1) $i := 1$ 1

(2) $j := 2$, and $e = 23$ 2

(3) $s_1 := 3$ and $T := T \backslash\backslash 23$ 3

(4) $i := 2$ and go to (2) 4

(2) $j := 3$, and $e = 31$ 5

(3) $s_2 := 1$ and $T := T \backslash\backslash\{23, 31\}$ 6

(4) $i := 3$ and go to (2) 7

(2) $j := 5$, and $e = 51$ 8

(3) $s_3 := 1$ and $T := T \backslash\backslash\{23, 31, 51\}$ 9

(4) $i := 4$ and go to (2) 10

(2) $j := 1$, and $e = 14$ 11

(3) $s_4 := 4$ and $T := T \backslash\backslash\{23, 31, 51, 14\}$ 12

(4) $i := 5$ and go to (2) 13

(2) $j := 6$, and $e = 64$ 14

(3) $s_5 := 4$ and $T := T \backslash\backslash\{23, 31, 51, 14, 64\}$ 15

(4) $i := 6$ and go to (2) 16

(2) $j := 7$, and $e = 74$ 17

(3) $s_6 := 4$ and $T := T \backslash\backslash\{23, 31, 51, 14, 64, 74\}$ 18

4. OUTPUT: $\text{PC1}(T) = (3, 1, 1, 4, 4, 4)$ 19

Notice that the leaves of the original tree are 2, 5, 6, 7, and 8, and that these are the integers which fail to appear in the output, whilst the nonleaves v are 1, 3, and 4, and each occurs $d(v) - 1$ times. When the algorithm stops, T is reduced to the single edge 84, which is incident upon the largest leaf of the original tree.

Of course, we are implementing each step (2) by inspection of the appropriate diagram. So far as running the algorithm on a machine is concerned, the adjacency list $A(T)$ will have to be scanned in some suitable fashion, and then updated after each implementation of step (3). Moreover, these procedures will themselves have to be defined algorithmically for a fully automated implementation! We content ourselves with noting that the leaves may easily be extracted from $A(T)$ as those vertices whose list has a single entry.

We can now explain our strategy for computing $t(n)$, the number of trees which can be constructed on vertices $\{1, \ldots, n\}$.

By running PC1 we may assign a Prüfer code on n to each such tree. If we can demonstrate that *every* Prüfer code s is $\text{PC1}(T)$ for some tree T, and that no s arises from more than one T, then we have established that there are exactly the same number of trees as there are Prüfer codes; in other words, that there is a 1-1 *correspondence* between them. This will be the purpose of the algorithm PC2 in the next section.

By way of anticipation, we define the enumerating function $s(n)$ to be the number of Prüfer codes on n for every integer $n \geq 2$, and consider the evaluation of $s(n)$ instead. Following the analogy with Table 4.3, we collect some experimental evidence for small values of n.

When n is 2, we have the single code consisting of the empty sequence ( ), and when n is 3 we have the 3 codes (1), (2), and (3) of length 1. When n is 4, we have the following 16 codes

$$\begin{array}{c} (1,1),\ (1,2),\ (1,3),\ (1,4),\ (2,1),\ (2,2),\ (2,3),\ (2,4) \\ (3,1),\ (3,2),\ (3,3),\ (3,4),\ (4,1),\ (4,2),\ (4,3),\ (4,4) \end{array} \tag{4.6}$$

of length 2. We may display this data as

Table 4.7

n	1	2	3	4
$s(n)$		1	3	16

,

which compares well with 4.3. It also suggests how to compute $s(n)$.

Lemma 4.8 *For each pair of positive integers n and m, there are n^m sequences of the form*

$$(r_1, r_2, \ldots, r_m),$$

where $1 \leq r_i \leq n$ for each $1 \leq i \leq m$.

Proof We use induction on m, taking $m = 1$ as our base case; the n 1-term sequences are (1), (2), ..., (n). To make the inductive step, we take as our hypothesis that there are n^k sequences with k entries between 1 and n, and then add a new final term in order to increase the number of entries by one. Since there are n possible values for the new entry, we obtain

$$n^k \cdot n = n^{k+1}$$

sequences in total, as required. Hence our statement is true for all m, by induction. □

Corollary 4.9 *For every integer $n \geq 2$, we have that $s(n) = n^{n-2}$.*

Proof Apply 4.8 with $m = n - 2$. □

Table 4.7 is now explained.

Before we turn our attention to PC2 we need to record one more aspect of Prüfer codes, as suggested by our remarks following Example 4.5. Given any sequence s of positive integers $(s_1, s_2, \ldots, s_m)$, where $1 \leq s_i \leq n$ for every $1 \leq i \leq m$, we let $o(j)$ denote the number of *occurrences* of the integer j in s, for each $1 \leq j \leq n$; in other words, $o(j)$ is the number of elements in the subsequence $\{s_i : s_i = j\}$. We then define the initial values of the frequency function $fr(\ )$ by

$$fr(j) = 1 + o(j)$$

for each $1 \leq j \leq n$. Thus the initial value of $fr(j)$ is 1 if and only if j does not occur in s at all. For example, given the output $(3, 1, 1, 4, 4, 4)$ of 4.5 we may tabulate $o(\ )$ and $fr(\ )$ as

j	1	2	3	4	5	6	7	8
$o(j)$	2	0	1	3	0	0	0	0

and (4.10)

j	1	2	3	4	5	6	7	8
$fr(j)$	3	1	2	4	1	1	1	1

respectively.

We note one simple property of $fr(\ )$.

Lemma 4.11 *The frequency function satisfies*

$$\sum_{j=1}^{n} fr(j) = 2n - 2.$$

for every Prüfer code s on n, where $n \geq 2$.

Proof Because s has $n-2$ terms, it follows that $\sum_{j=1}^{n} o(j) = n-2$. Since the summation involves n terms we may convert this into a formula for the frequency function by adding n to both sides, and so obtain the required result. □

For the example $(3,1,1,4,4,4)$, we see from (4.10) that the sum of the frequencies is 14, which is indeed $2 \cdot 8 - 2$.

4.3 From Prüfer Codes to Trees

We now introduce the algorithm PC2. This performs the reverse task from PC1, and utilises the frequency function introduced in Section 4.2.

Algorithm 4.12 (Prüfer Code 2: PC2)
INPUT: *a Prüfer code $(s_1, s_2, \ldots, s_{n-2})$ on n, and its frequency function $fr(\ )$*
OUTPUT: *a tree T with vertex set $\{1, \ldots, n\}$*

(1) $i := 1$ *and* $T := \emptyset$

(2) $j :=$ *the smallest integer with* $fr(j) = 1$, *and* $T := T \cup js_i$

(3) $fr(j) := 0$ *and* $fr(s_i) := fr(s_i) - 1$

(4) *if* $i = n-2$, *go to* (5); *otherwise*, $i := i+1$ *and go to* (2)

(5) $T := T \cup kl$, *where* $fr(k) = fr(l) = 1$ *and* $fr(j) = 0$ *for all other* j

(6) *output* $\mathrm{PC2}(s) = T$ *and stop*

If $n = 2$, we decree PC2( ) to be the unique tree on two vertices; this is equivalent to running steps (5) and (6) only.

We must confirm that the output of PC2 *is* a tree.

Lemma 4.13 *The output of* PC2 *contains no cycles.*

Proof Suppose that $\mathrm{PC2}(s)$ contains a cycle σ, and that js_i is the first edge of σ inserted whilst running PC2. Each time step (2) is called, the vertex j is never used again, since $fr(j)$ becomes 0 in step (3). Therefore no subsequently inserted edge can complete the cycle at j, which is a contradiction. □

Since the algorithm creates $n-1$ edges in all, by steps (4) and (5), so $\mathrm{PC2}(s)$ is a tree by Lemma 3.20.

To justify our description of step (5), we note that it is implemented only after step (3) has been executed $n-2$ times. Each such execution creates one new value of j for which

$fr(j) = 0$, so at step (5) there are exactly 2 values for which $fr(k)$ and $fr(l)$ are nonzero. Moreover, each execution reduces $\sum_{j=1}^{n} fr(j)$ by 2, leaving

$$\begin{aligned} fr(k) + fr(l) &= (2n-2) - 2(n-2) = 2 \qquad \text{by Lemma 4.11} \\ &= 2 \end{aligned}$$

at step (5). Therefore $fr(k) = fr(l) = 1$, as required.

Here is an example of PC2, acting on a familiar code.

Example 4.14 *Run* PC2 *on the Prüfer code* $s = (3, 1, 1, 4, 4, 4)$.

Solution First we recall $fr(\,)$ from 4.10:

j	1	2	3	4	5	6	7	8
$fr(j)$	3	1	2	4	1	1	1	1

(1) $i := 1$ and $T := \emptyset$ 1

(2) $j := 2$ and $T := \{23\}$ 2

(3) $fr(j) :=$ 3

j	1	2	3	4	5	6	7	8
$fr(j)$	3	0	1	4	1	1	1	1

(4) $i := 2$ and go to (2) 4

(2) $j := 3$ and $T := \{23, 31\}$ 5

(3) $fr(j) :=$ 6

j	1	2	3	4	5	6	7	8
$fr(j)$	2	0	0	4	1	1	1	1

(4) $i := 3$ and go to (2) 7

(2) $j := 5$ and $T := \{23, 31, 51\}$ 8

(3) $fr(j) :=$ 9

j	1	2	3	4	5	6	7	8
$fr(j)$	1	0	0	4	0	1	1	1

(4) $i := 4$ and go to (2) 10

(2) $j := 1$ and $T := \{23, 31, 51, 14\}$ 11

(3) $fr(j) :=$ 12

j	1	2	3	4	5	6	7	8
$fr(j)$	0	0	0	3	0	1	1	1

(4) $i := 5$ and go to (2) 13

(2) $j := 6$ and $T := \{23, 31, 51, 14, 64\}$ 14

(3) $fr(j) :=$ 15

j	1	2	3	4	5	6	7	8
$fr(j)$	0	0	0	2	0	0	1	1

(4) $i := 6$ and go to (2) 16

(2) $j := 7$ and $T := \{23, 31, 51, 14, 64, 74\}$ 17

(3) $fr(j) :=$ 18

j	1	2	3	4	5	6	7	8
$fr(j)$	0	0	0	1	0	0	0	1

(4) $i = 6$, so go to (5) 19

(5) $T := \{23, 31, 51, 14, 64, 74, 84\}$ 20

(6) OUTPUT: 21

$A(1)$	3, 4, 5
$A(2)$	3
$A(3)$	1, 2
$A(4)$	1, 6, 7, 8
$A(5)$	1
$A(6)$	4
$A(7)$	4
$A(8)$	4

=

5 7

2 3 1 4 6

8

We see that running PC2 in 4.14 has reversed the effect of running PC1 in 4.5, precisely as we hoped. We must establish this principle in full generality, by proving that PC1 and PC2 are reciprocal procedures in the following sense.

Proposition 4.15 *The Prüfer code s is obtained by running* PC1 *on the tree T if and only if the tree T is obtained by running* PC2 *on the Prüfer code s.*

Proof We use induction on n, the number of vertices. The base case may be taken as $n = 2$, for which the empty sequence is the unique Prüfer code and the tree on two vertices the unique T. Nervous readers may prefer to begin with $n = 3$! We take as our inductive hypothesis that PC1 and PC2 are reciprocal procedures for all trees on k vertices.

Now suppose that T is a tree on $k+1$ vertices; we must demonstrate that

$$\mathrm{PC1}(T) = s \iff \mathrm{PC2}(s) = T. \tag{4.16}$$

Well, the first edge removed by running PC1 on T is $e = js_1$, where j is the smallest leaf, and the resulting tree $T \backslash\backslash e$ has only k vertices, namely $\{1, \ldots, k+1\}$ with j removed. The steps involved in continuing to run PC1 on T, and in running PC1 on $T \backslash\backslash e$, are therefore identical, so

$$\mathrm{PC1}(T) = (s_1, s_2, \ldots, s_{k-1}) \iff \mathrm{PC1}(T \backslash\backslash e) = (s_2, \ldots, s_{k-1}). \tag{4.17}$$

Also, applying the inductive hypothesis yields

$$\mathrm{PC1}(T \backslash\backslash e) = (s_2, \ldots, s_{k-1}) \iff \mathrm{PC2}(s_2, \ldots, s_{k-1}) = T \backslash\backslash e. \tag{4.18}$$

Now the first edge created by running PC2 on s is the same $e = js_1$, since j is the smallest integer not appearing in s, and is therefore the smallest integer with $fr(j) = 1$. During the creation of this edge, s_1 is eliminated from s since $fr(s_1)$ is also reduced by 1 in step (3). The steps involved in continuing to run PC2 on s, and in running PC2 on $(s_2, \ldots, s_{k-1})$, are therefore identical, so

$$\mathrm{PC2}(s) = T \iff \mathrm{PC2}(s_2, \ldots, s_{k-1}) = T \backslash\backslash e. \tag{4.19}$$

Combining (4.19) with (4.17) and (4.18) yields (4.16), and our induction is complete. □

As a corollary we may deduce the celebrated

Theorem 4.20 (Cayley's Theorem) *Given any integer $n \geq 2$ there are n^{n-2} spanning trees for K_n.*

Proof By 4.15, every Prüfer code s is indeed the code of some tree T, namely the tree obtained by running PC2 on s; therefore T is the *only* tree which can have s as its code, since no choice is involved at any step of PC2. Hence codes and trees are in $1-1$ correspondence, in the sense of Section 4.2. Thus $s(n) = t(n)$ for all $n \geq 2$, and the result follows from Corollary 4.9. □

Note that the case $n = 1$ is also covered by the theorem, if only by accident!

Chapter 5

Eulerian Tours

5.1 Euler's Theorem

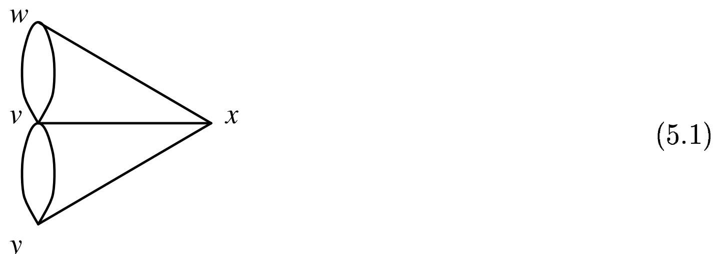
We now turn to the issues raised by the Traversibility Question 2.3. In fact this problem is one of the most famous in mathematics, and its consideration by Euler in 1736 not only launched the subject of graph theory, but also laid the foundations for the related study of Topology, another branch of pure mathematics which has been extremely active during the last few decades.

Euler's involvement stemmed from a debate amongst the citizens of Königsberg, which was then in Eastern Prussia. The city straddles the River Pregel, in which there is an island and a confluence, connected by a system of bridges as depicted (somewhat obscurely!) in the accompanying illustration from a 17th century book.

For years, the inhabitants had discussed the possibility of taking a walking tour which

crossed each of the seven bridges exactly once and then returned to the starting point. This, of course, is a more genteel version of the Traversibility Question 2.3.

Euler's genius was to understand how to abstract the problem into a format which displays all the information required for a solution, and no more. This is the multigraph



(5.1)

and although we may now think such a display is a trivial reformulation, at the time it represented a major advance in applying mathematical reasoning to the real world.

We have already explained in detail in Section 3.1 what we mean by paths and cycles in a graph. Recall that repetition of edges and vertices are not permitted, except that the initial and final vertices of a cycle must coincide. We now allow more general possibilities, and define a walk ω in a graph to be an alternating sequence of vertices and edges

$$v_0, e_1, v_1, e_2, \ldots, e_n, v_n, \tag{5.2}$$

where each e_i is $v_{i-1}v_i$; there is *no* requirement that the edges or vertices of ω be distinct. We say that ω enters v_i by e_i and exits by e_{i+1}. Predictably, the length $\ell(\omega)$ of ω is taken to be n. A subwalk of ω is any subsequence of consecutive terms which begins and ends with a vertex. A circuit is a walk with the same beginning and end. Observe that a path is a special case of a walk, and that a cycle is a special case of a circuit.

So long as G is a simple graph, we may reduce 5.2 to the list of vertices

$$v_0, v_1, \ldots, v_n$$

and still determine ω.

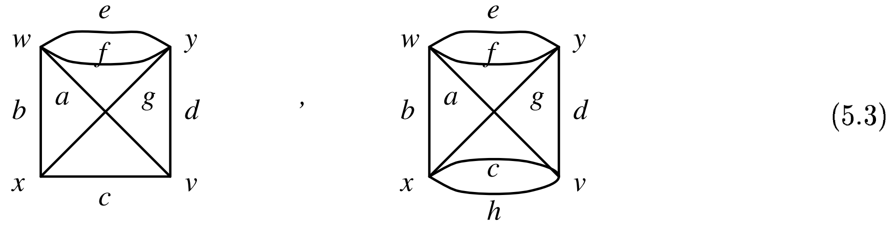
Because of its importance in this chapter, we shall reserve the symbol p exclusively for the number of edges of G.

In honour of Euler's solution, we define an Eulerian walk in a graph G to be a walk which includes every edge of G exactly once (although, of course, the vertices may be repeated several times); if the walk is a circuit, we call it an Eulerian circuit. Sometimes we apply the description Eulerian tour to cover both possibilities, and any graph which has such a tour is called Eulerian. Thus both 2.3 and the conundrum facing the Königsbergers involve deciding whether or not a certain graph is Eulerian.

For future reference, it is convenient to describe a circuit of G as full if it contains all the edges of G at least once. Any Eulerian circuit γ for G is therefore full, although there are many other full circuits of such a G which fail to be Eulerian; for example those obtained by repeating γ (or any subcircuit of γ) several times.

Notice that a diagram for an Eulerian graph may, at least in principle, be drawn without lifting pen from paper. Clearly a graph which is Eulerian must be connected. Here are two

examples of Eulerian graphs:



Thus $v, a, w, b, x, c, v, d, y, e, w, f, y, g, x$ is an Eulerian walk from v to x in the first example, and $v, a, w, b, x, c, v, d, y, e, w, f, y, g, x, h, v$ is an Eulerian circuit from v to v in the second.

Before describing Euler's original result, it is instructive to recall from Corollary 2.19 that every multigraph possesses an even number of odd vertices.

Proposition 5.4 (Euler) *If a connected graph G admits an Eulerian walk (which is not a circuit), then it contains precisely two odd vertices, which must be the beginning and the end of the walk; if G admits an Eulerian circuit, then all its vertices are even.*

Proof Suppose that ω is an Eulerian tour, and consider any vertex v in G other than the beginning or the end. Clearly ω must enter and exit v the same number of times, and since it is Eulerian, v must be even. If the tour is a walk (but not a circuit) this reasoning also applies to the beginning and end, except that the former is exited one more time than it is entered, and the latter is entered one more time than it is exited; hence both are odd. If the tour is a circuit the beginning and end coincide, and this vertex must then have even degree as well. □

Readers should now check that the examples in (5.3) are consistent with 5.4. Note that the degree of each vertex can be read off immediately from the adjacency list of any G, and that a machine may therefore instantly determine when a graph cannot be Eulerian.

We emphasise 5.4 only gives *necessary* conditions for the existence of an Eulerian tour; the best we can deduce is that if G fails to satisfy the stated conditions (as is clearly the case with (5.1)), then no such tour exists. Remarkably, these conditions are also sufficient for G to be Eulerian, a fact which Euler stated but did not prove. It was nearly 140 years before a rigorous justification finally appeared in the literature! This proof was due to Carl Hierholzer, a German mathematician who died tragically in 1871 whilst still a student, and it will form the basis of our algorithm for constructing Eulerian Tours in the next section.

Of course 5.4 naturally suggests that we should consider the properties of multigraphs with $2r$ odd vertices for $r > 1$. This issue is central to the application of Hierholzer's Algorithm to the Repetition Question 2.4, and motivates the following procedure. Suppose we partition the vertices of G as $V = V_{\text{od}} \cup V_{\text{ev}}$ according to their parity, and write V_{od} as $\{v_1, \ldots, v_{2r}\}$ for some $r \geq 1$. Then we define a matching set of walks in G to be a set

$$M = \{\omega_1, \ldots, \omega_r\}$$

of walks, where each ω_i begins and ends at some odd vertex and every odd vertex is used exactly once. Since we naturally assume G to be connected, the existence of at least one such M is assured. We refer to the sum

$$\ell(M) = \sum_{i=1}^{r} \ell(\omega_i)$$

as the $\boxed{\text{length}}$ of M.

Thus if G has an Eulerian circuit, then r is zero and any matching set of walks is empty; if G has an Eulerian walk, then r is 1 and a matching set of walks contains a single walk (or path) between the two odd vertices.

Here is a multigraph G with four odd vertices v, w, x and y (and therfore no Eulerian tour).

(5.5)

Any matching set in G contains two walks, and in general there will be several alternatives; for example, we may choose either

$$\{v, e, w \quad , \quad x, f, z, g, y\} \qquad \text{or} \qquad \{y, g, z, f, x, h, v \quad , \quad w, e, v, h, x\}$$

as our set, with lengths 3 and 5 respectively.

5.2 Hierholzer's Algorithm

We now turn to the problem of constructing Eulerian tours on a given multigraph G.

The beauty of Hierholzer's Theorem, given below, is that it provides sufficient conditions for such tours to exist whilst simultaneously offering an algorithmic procedure for their construction.

Theorem 5.6 (Hierholzer's Theorem) *If a connected graph G has precisely two odd vertices, they are the beginning and end of an Eulerian walk; if every vertex of G is even, then G admits an Eulerian circuit.*

Proof We assume first that G has two odd vertices u and v, and begin a walk ω at u, exiting each vertex by an unused edge until a vertex z is reached for which every incident edge has been used. Observe that z cannot be u, otherwise u would have *even* degree; therefore z must have odd degree, and so must be v. If ω uses every edge of G then it is an Eulerian tour and we have finished. Otherwise we write $E(\omega)$ for the set of its edges, and define the new graph H to be $G \backslash\backslash E(\omega)$; every vertex of H is even, because we have removed from G an odd number of edges incident on u and v, and an even number of edges incident on every other vertex.

Now H must contain some vertex x of ω which is incident on an unused edge of G, otherwise G could not have been connected. So we begin a second walk at x, and repeat our earlier procedure of extending it edge by edge. This time the walk is forced to terminate at x itself because there is no odd vertex to act as an end point, and we obtain a circuit σ. We may respectively describe ω and σ as

$$u, e_1, \ldots, e_{s-1}, x, e_s, \ldots, e_n, v \qquad \text{and} \qquad x, f_1, \ldots, f_{t-1}, x_{t-1}, f_t, \ldots, f_m, x,$$

and form the new walk

$$u, e_1, \ldots, e_{s-1}, x, f_1, \ldots, f_{t-1}, x_{t-1}, f_t, \ldots, f_m, x, e_s, \ldots, e_n, v$$

by interposing σ in ω at x.

If this new walk uses every edge of G then it is an Eulerian tour and we have finished. Otherwise, we repeat the procedure again with $H \setminus\setminus E(\sigma)$. Eventually the process must terminate, because G has only finitely many edges, and at least one of them is removed at every stage. The final tour we obtain is therefore Eulerian.

If G has no odd vertices, we simply assume that the initial walk ω is empty and let H be G; we may then move directly to the second stage, and begin the construction of ω at any vertex x. □

Corollary 5.7 *Euler's conditions (as in Proposition* 5.4*) are both necessary and sufficient for a graph to be Eulerian.*

It is worth remarking that the simplest case of 5.6 is when G has a single edge. If there are two vertices of degree 1 then G is the complete graph K_2, and the Eulerian tour passes from one vertex to the other; if there is one vertex of degree 2 then the edge is a loop, and the Eulerian circuit traverses it in either direction.

Also, we may reduce the case when G has two odd vertices u and v to the case when G has *no* odd vertices by the following simple device. Let G^+ be obtained from G by adding a new edge uv; that is, let G^+ be $G \cup uv$. Then the vertices of G^+ are all even, and each Eulerian circuit for G^+ corresponds to an Eulerian walk in G simply by omitting the edge uv from the circuit. A generalisation of this strategy will be especially important in the next section.

Let us now use the proof of 5.6 to construct our algorithm. As with DFS, the most convenient description is in terms of a certain subroutine.

Algorithm 5.8 (Hierholzer's Algorithm: HA)
INPUT: *a connected graph G with either two odd vertices u and v, or all vertices even*
OUTPUT: *an Eulerian tour of G, beginning at u*

(1) $r := u$, *and do* Walk(r) *to obtain a walk* ω

(2) *choose a vertex x in ω such that some edge of G not in ω is incident on x, and* $G := G \setminus\setminus E(\omega)$ *and* $r := x$*; if none, go to* (5)

(3) *do* Walk(r) *to obtain a circuit* σ

(4) $\omega := \omega$ *with σ interposed at x, and go to* (2)

(5) *output ω and stop*

where Walk(r) *is*

(1) $\omega := r$

(2) *choose an edge* $e = ru$*; then* $G := G \setminus\setminus e$ *and* $\omega := \omega, e, u$

(3) $r := u$*, and if $A(r)$ is nonempty, go to* (2)*; otherwise, go to* (4)

(4) *output ω and stop*

A suitable edge ru always exists at the first implementation of step (2) in Walk(r), since G is connected.

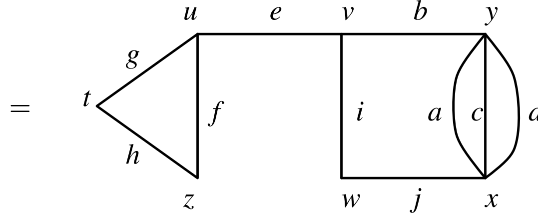
Note that, for an arbitrary G, the only property of the output of Walk(r) about which we can be certain is that it is a walk beginning at r! Only after appeal to the proof of 5.6 can we be sure that it constructs either a walk from u to v (or a circuit at r) during step (1) of HA, and a circuit at x during step (3).

The algorithm is best understood by considering an example. We give all the details; in fact they can easily be condensed into tabular form once the procedure is mastered.

Example 5.9 *Run* HA *on the following multigraph:*

$A(t)$	u, z
$A(u)$	t, v, z
$A(v)$	$u, w, y,$
$A(w)$	v, x
$A(x)$	w, y, y, y
$A(y)$	v, x, x, x
$A(z)$	t, u

=



Solution

(1) $r := u$, and do Walk(u) 1

(1) $\omega := u$ 2

(2) choose $e = g$; then $G := G \backslash\backslash g$ and $\omega := u, g, t$ 3

(3) $r := t$ and go to (2) 4

(2) choose $e = h$; then $G := G \backslash\backslash \{g, h\}$ and $\omega := u, g, t, h, z$ 5

(3) $r := z$ and go to (2) 6

(2) choose $e = f$; then $G := G \backslash\backslash \{g, h, f\}$ and $\omega := u, g, t, h, z, f, u$ 7

(3) $r := u$ and go to (2) 8

(2) choose $e = e$; then $G := G \backslash\backslash \{g, h, f, e\}$ and $\omega := u, g, t, h, z, f, u, e, v$ 9

(3) $r := v$ and go to (2) 10

(2) choose $e = b$; then $G := G \backslash\backslash \{g, h, f, e, b\}$ and $\omega := u, g, t, h, z, f, u, e, v, b, y$ 11

(3) $r := y$ and go to (2) 12

(2) choose $e = c$; then $G := G \backslash\backslash \{g, h, f, e, b, c\}$ and $\omega := u, g, t, h, z, f, u, e, v, b, y, c, x$ 13

(3) $r := x$ and go to (2) 14

(2) choose $e = j$; then $G := G \backslash\backslash \{g, h, f, e, b, c, j\}$
and $\omega := u, g, t, h, z, f, u, e, v, b, y, c, x, j, w$ 15

(3) $r := w$ and go to (2) 16

(2) choose $e = i$; then $G := G \backslash\backslash \{g, h, f, e, b, c, j, i\}$
and $\omega := u, g, t, h, z, f, u, e, v, b, y, c, x, j, w, i, v$ 17

To continue, we note at this point that G has become

(5.10)

(3) $r := v$; since $A(v)$ is empty, so go to (4) 18

(4) output $\omega = u, g, t, h, z, f, u, e, v, b, y, c, x, j, w, i, v$ 19

(2) choose vertex y; then $G := G \backslash\backslash E(\omega)$ (as in (5.10)) and $r := y$ 20

(3) do Walk(y) 21

(1) $\omega := y$ 22

(2) choose $e = a$; then $G := G \backslash\backslash a$ and $\omega := y, a, x$ 23

(3) $r := x$ and go to (2) 24

(2) choose $e = d$; then $G := G \backslash\backslash \{a, d\}$ and $\omega := y, a, x, d, y$ 25

(3) G is now empty, so go to (4) 26

(4) output $\sigma = y, a, x, d, y$ 27

(4) $\omega := u, g, t, h, z, f, u, e, v, b, y, a, x, d, y, c, x, j, w, i, v$ and go to (2) 28

(2) ω contains all edges of the original G, so go to (5) 29

(5) OUTPUT: $u, g, t, h, z, f, u, e, v, b, y, a, x, d, y, c, x, j, w, i, v$ 30

This is visibly an Eulerian walk for the multigraph in question.

5.3 Shortest Full Circuits

We now investigate the Repetition Question 2.4. Given a multigraph G, we may rephrase our aim as the determination of a shortest full circuit in G; as we shall see, such a circuit need not be unique.

Proposition 5.11 *If a circuit is full in G, then it necessarily repeats the edges of some matching set of walks M.*

Proof We proceed by induction on r, where $2r$ denotes the number of odd vertices in G. The base case is $r = 0$, for which the result is trivial because the only matching set of walks is empty. To make the inductive step, we take as our hypothesis that the result is true for every graph with $2r \leq 2k$ odd vertices, and let G be a graph with $2k + 2$ odd vertices. Now take any circuit γ which is full in G, and let v_1 be an odd vertex. At least one edge incident on v_1, say e_1, must occur twice or more in γ, since it enters and exits v_1 an even number of times in total. Let v_2 be the other end of e_1. If v_2 is odd, stop; otherwise, since v_2 is even

and e_1 is repeated in γ, some other edge e_2 incident on v_2 must also be repeated. Continuing in this way, we must eventually reach a second odd vertex v_m. We may therefore build up a walk

$$\tau = v_1, e_1, v_2, e_2, \ldots, v_m,$$

where v_1 and v_m are odd (and distinct), and the intermediate vertices are even.

Now we duplicate the edges of τ in G to obtain a new graph G^+. Since the degrees of v_1 and v_m are increased by 1 and the degrees of the intermediate vertices are increased by 2, we deduce that G^+ has only $2k$ odd vertices. Moreover, γ extends to a circuit γ^+ which is full in G^+, since it contains all the edges of τ at least twice. We may therefore apply the inductive hypothesis to γ^+, and deduce that it repeats the edges of some matching set of walks M^+ in G^+. But adjoining τ to M^+ yields a matching set of walks M for G, every edge of which is repeated by γ. Our inductive step is complete, and our original claim is therefore verified for all values of r. □

We emphasise that other edges than those in M may also be repeated in γ. Thus

$$p + \ell(M) \leq \ell(\gamma). \tag{5.12}$$

Conversely, we have

Proposition 5.13 *If M is a matching set of walks in G, then there is a full circuit in G which repeats only the edges of M.*

Proof Let M be $\{\omega_1, \ldots, \omega_r\}$, and suppose that the endpoints of each ω_i are v_{2i-1} and v_{2i}, for $1 \leq i \leq r$. Thus v_1, …,v_{2r} are the odd vertices of G. Now form G^+ by adding new edges $m_i = v_{2i-1}v_{2i}$ to G, so that every vertex of G^+ is even, and Theorem 5.6 applies to give an Eulerian circuit γ^+. Substituting ω_i for each triple v_{2i-1}, m_i, v_{2i} in γ^+ then yields the required circuit for G. □

We may combine 5.11 and 5.13 to answer the Repetition Question 2.4 for a multigraph G with p edges.

Theorem 5.14 *Given a shortest matching set of walks M_{sh} in G, there is a shortest full circuit γ_{sh} such that*

$$\ell(\gamma_{\text{sh}}) = p + \ell(M_{\text{sh}}),$$

and γ_{sh} repeats only the edges of M_{sh}.

Proof Apply 5.13 to M_{sh} and let γ be the resulting full circuit. Then $\ell(\gamma) = p + \ell(M_{\text{sh}})$, and γ repeats only the edges of M_{sh}; it therefore remains to show that γ is shortest.

For this we apply 5.11 to *any* shortest full circuit γ_{sh}, and let M be the resulting matching set of walks. Then $p + \ell(M) \leq \ell(\gamma_{\text{sh}})$ by 5.12, so

$$\ell(\gamma) = p + \ell(M_{\text{sh}}) \leq p + \ell(M) \leq \ell(\gamma_{\text{sh}}).$$

Thus $\ell(\gamma)$ must be the same as $\ell(\gamma_{\text{sh}})$, whence γ is also shortest. □

We have now answered the Repetition Question 2.4, and provided all the information we need for an algorithmic construction of a shortest full circuit.

Algorithm 5.15 (Shortest Full Circuit: SFC)
INPUT: *a connected graph G with more than two odd vertices*
OUTPUT: *a shortest full circuit in G*

(1) *identify the odd vertices* $\{v_1, \ldots, v_{2r}\}$

(2) *compute all distances* $d(v_i, v_j)$ *(by running* BFS *repeatedly)*

(3) *obtain a shortest matching set of walks M from* (2)

(4) $G^+ := G \cup_i \{m_i\}$, *where the edges m_i join the endpoints of each walk ω_i in M*

(5) *obtain an Eulerian circuit γ^+ for G^+ (by running* HA*)*

(6) *convert γ^+ to a circuit γ in G by substituting ω_i for each m_i*

(7) *output γ and stop*

Of course, the length of γ may be deduced immediately after step (3) is completed, by appealing to 5.14.

We have already given the necessary theoretical justification for SFC behaving as claimed; however the algorithm has several features which still require comment.

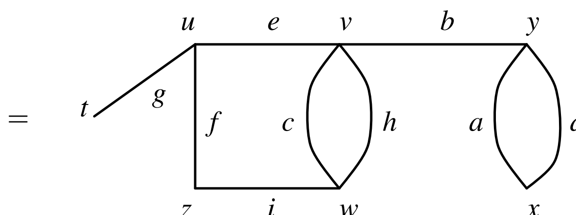
Firstly, during execution it calls on both BFS and HA. We could formulate this dependence more precisely (and *must* do so if writing a computer program), but we have preferred to adopt an informal approach in the interests of simplicity. Secondly, in step (3), we have specified no algorithmic method for finding an M of minimal length. This is actually a subtle problem, and there are several *Matching Algorithms* which we could invoke at this point; however, they all require considerable effort to describe. In the problems we consider the number of odd vertices is small, and tabulating all the possibilities will suffice. If we were not concerned with the minimality of γ, this complication would not arise, and we could adapt step (3) by extracting any matching set of walks from the appropriate BFS search trees.

We illustrate SFC with an example.

Example 5.16 *Run* SFC *on the following multigraph:*

$A(t)$	u
$A(u)$	t, v, z
$A(v)$	$u, w, w, y,$
$A(w)$	v, v, z
$A(x)$	y, y
$A(y)$	v, x, x
$A(z)$	u, w

=



Solution

(1) the odd vertices are $\{t, u, w, y\}$ (*by scanning the adjacency list*) 1

(2) the six distances $d(v_i, v_j)$ are given by the table 2

	t	u	w	y
t	0	1	3	3
u	1	0	2	2
w	3	2	0	2
y	3	2	2	0

(*by running* BFS *repeatedly; or in this case, by inspection!*)

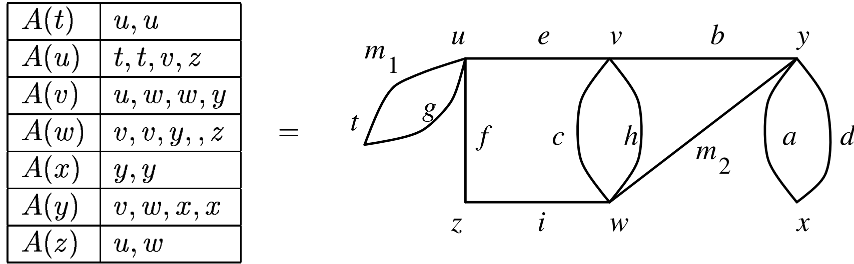
(3) the possible matching sets of walks, with their lengths, are 3

M			$\ell(M)$
t, g, u	and	w, c, v, b, y	$1+2=3$
t, g, u, e, v, b, y	and	u, f, z, i, w	$3+2=5$
t, g, u, f, z, i, w	and	u, e, v, b, y	$3+2=5$

;

so choose t, g, u and w, c, v, b, y (*by inspection*)

(4) construct G^+ to be 4



$A(t)$	u, u
$A(u)$	t, t, v, z
$A(v)$	u, w, w, y
$A(w)$	$v, v, y, , z$
$A(x)$	y, y
$A(y)$	v, w, x, x
$A(z)$	u, w

=

(5) obtain the Eulerian circuit 5

$$\gamma^+ = u, m_1, t, g, u, f, z, i, w, h, v, c, w, m_2, y, a, x, d, y, b, v, e, u$$

for G^+ (*by running* HA*; or in this case by inspection*)

(6) convert γ^+ to 6

$$\gamma = u, g, t, g, u, f, z, i, w, h, v, c, w, c, v, b, y, a, x, d, y, b, v, e, u$$

(7) OUTPUT: $u, g, t, g, u, f, z, i, w, h, v, c, w, c, v, b, y, a, x, d, y, b, v, e, u$ 7

Observe that the shortest full circuit has length 12, and that precisely three edges are necessarily repeated in answer to the Repetition Question 2.4. Although the repeated edges are unique in this case, the entire circuit is not, since there are several possible choices for γ^+ at step (5).

Chapter 6

Weighted Graphs

6.1 Floyd's Algorithm

Several of our introductory questions in Section 2.1 (for example the Weighted Spanning Trees Question 2.9) concerned graphs to whose edges had been assigned certain quantities such as cost, length, or capacity. We now formalise these ideas, and establish algorithms for solving the corresponding problems.

A **weight function** on a multigraph G is a function $w\colon E(G) \to \mathbf{R}^+$ which assigns a positive real number to every edge of the graph; we refer to the value $w(e)$ as the **weight** of the edge e, and to the pair (G, w) as a **weighted** multigraph. Almost always, we shall assume that the weight of an edge is a positive integer. This is not unrealistic, since in most problems we can multiply the weights by a common integer N which is large enough to cancel any denominators, then perform our computation, and finally divide out again by N to obtain our answer. We may store a weighted graph on machine by means of its **weighted** adjacency list $A^w(G)$, obtained from $A(G)$ by replacing each entry v in $A(u)$ by the pair $(v, w(uv))$.

An *un*weighted graph may naturally be weighted by taking $w(e)$ to be 1 for every edge e.

If π is a path in G from u to v, we let the expression

$$w(\pi) = \sum_{e \in E(\pi)} w(e)$$

denote the weight of π. We may then define the **weighted** distance from u to v as

$$d^w(u, v) = \min_{\pi} w(\pi), \tag{6.1}$$

where π ranges over all paths from u to v. There may be several such paths whose weight is $d^w(u, v)$; we refer to any one of them as a **lightest** path from u to v. If there is *no* path between u and v, we insist that $d^w(u, v)$ is ∞.

Whenever the weight of every edge of G is 1, then $d^w(u, v)$ reduces to the standard distance $d(u, v)$, and may be found by running BFS as in Section 2. In general, however, the determination of $d^w(u, v)$ is more subtle, and we shall solve it by introducing Floyd's Algorithm 6.4.

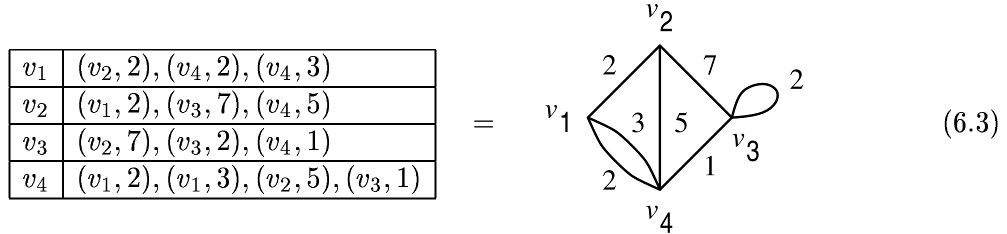
This algorithm is best described in terms of matrices, and we begin by constructing a matrix which will serve as input. We suppose that G is a connected, weighted multigraph

with vertices $\{v_1, \ldots, v_n\}$, and define the weight matrix a_G^w of G by

$$a_G^w(j,k) = \begin{cases} \min w(v_j v_k) & \text{if } v_j v_k \in E(G) \\ 0 & \text{if } j = k \\ \infty & \text{otherwise} \end{cases} \tag{6.2}$$

where $a_G^w(j,k)$ denotes the entry in row j and column k. The minimum $\min w(v_j v_k)$ is taken over all multiple edges from v_i to v_j, so if there is only one such edge, the weight of that edge is chosen. Thus the weight matrix ignores any multiple edges that are not of minimum weight, together with all loops, and so displays the weighted distances between *adjacent* vertices of G. Clearly a_G^w is $n \times n$. If every edge of G has weight 1, then a_G^w reduces to the adjacency matrix of G.

For example, the weighted multigraph G given by



v_1	$(v_2,2),(v_4,2),(v_4,3)$
v_2	$(v_1,2),(v_3,7),(v_4,5)$
v_3	$(v_2,7),(v_3,2),(v_4,1)$
v_4	$(v_1,2),(v_1,3),(v_2,5),(v_3,1)$

has weight matrix

$$a_G^w = \begin{pmatrix} 0 & 2 & \infty & 2 \\ 2 & 0 & 7 & 5 \\ \infty & 7 & 0 & 1 \\ 2 & 5 & 1 & 0 \end{pmatrix}$$

Algorithm 6.4 (Floyd's Algorithm: FA)
INPUT: *a connected, weighted multigraph G with vertices $\{v_1, \ldots, v_n\}$*
OUTPUT: $d^w(v_j, v_k)$ *for every pair of vertices v_j and v_k*

(1) $i := 1$ *and* $w_0(j,k) := a_G^w(j,k)$

(2) $w_i(j,k) := min\{w_{i-1}(j,k),\ w_{i-1}(j,i) + w_{i-1}(i,k)\}$ *for all* $1 \leq j,\ k \leq n$

(3) *if* $i = n$ *go to* (4)*; otherwise* $i := i+1$ *and go to* (2)

(4) *output* $w_n(j,k)$ *for all* $1 \leq j,\ k \leq n$ *and stop*

We shall prove that the output entry $w_n(j,k)$ is indeed the weighted distance $d^w(j,k)$ for all $1 \leq j,\ k \leq n$; but first we give an example of FA in action.

Example 6.5 *Run* FA *on the weighted multigraph G of (*6.3*).*

Solution At each implementation of step (2), it suffices to display the matrix w_i, whose entry in row j and column k is $w_i(j,k)$.

$$\text{(1)}\ \ i := 1 \text{ and } w_0 := \begin{pmatrix} 0 & 2 & \infty & 2 \\ 2 & 0 & 7 & 5 \\ \infty & 7 & 0 & 1 \\ 2 & 5 & 1 & 0 \end{pmatrix} \qquad 1$$

(2) $w_1 := \begin{pmatrix} 0 & 2 & \infty & 2 \\ 2 & 0 & 7 & 4 \\ \infty & 7 & 0 & 1 \\ 2 & 4 & 1 & 0 \end{pmatrix}$ 2

(3) $i := 2$ and go to (2) 3

(2) $w_2 := \begin{pmatrix} 0 & 2 & 9 & 2 \\ 2 & 0 & 7 & 4 \\ 9 & 7 & 0 & 1 \\ 2 & 4 & 1 & 0 \end{pmatrix}$ 4

(3) $i := 3$ and go to (2) 5

(2) $w_3 := \begin{pmatrix} 0 & 2 & 9 & 2 \\ 2 & 0 & 7 & 4 \\ 9 & 7 & 0 & 1 \\ 2 & 4 & 1 & 0 \end{pmatrix}$ 6

(3) $i := 4$ and go to (2) 7

(2) $w_4 := \begin{pmatrix} 0 & 2 & 3 & 2 \\ 2 & 0 & 5 & 4 \\ 3 & 5 & 0 & 1 \\ 2 & 4 & 1 & 0 \end{pmatrix}$ 8

(3) $i = 4$, so go to (4) 9

(4) OUTPUT: $\begin{pmatrix} 0 & 2 & 3 & 2 \\ 2 & 0 & 5 & 4 \\ 3 & 5 & 0 & 1 \\ 2 & 4 & 1 & 0 \end{pmatrix}$ 10

Theorem 6.6 *After running* FA *on the multigraph G, we have that $w_n(j,k) = d^w(v_j, v_k)$ for each $1 \leq j,\ k \leq n$.*

Proof We prove by induction on i the statement that $w_i(j,k)$ is the weight of a lightest path from v_j to v_k which uses at most the vertices $v_1, \dots, v_i$ (in addition, of course, to the endpoints themselves). The base case is $i = 0$, in which *no* additional vertices are allowed, and the statement is true by the definition (6.2) of w_0. To make the inductive step, we take as our hypothesis the given statement, and consider $w_{i+1}(j,k)$. By step (2), this is the smaller of the two quantities

$$w_i(j,k) \qquad \text{and} \qquad w_i(j, i+1) + w_i(i+1, k). \tag{6.7}$$

The former of these is, by our hypothesis, the weight of a lightest path from v_j to v_k using at most the vertices $v_1, \dots, v_i$. We claim that the latter is the weight of a lightest path from v_j to v_k which also uses v_{i+1}; such a path can be split into two parts at v_{i+1}, both of which are

lightest possible using at most $v_1, \ldots, v_i$, so our claim follows from two further applications of the hypothesis.

Thus $w_{i+1}(j,k)$ is indeed the weight of a lightest path from v_j to v_k using at most the vertices $v_1, \ldots, v_{i+1}$; it is given by the first expression in (6.7) if such a path does *not* require the use of v_{i+1}, and by the second expression if it does. Our induction is complete, and our theorem is proved by setting $i = n$, in which case $w_n(j,k)$ is the weight of a lightest path from v_j to v_k using *any* vertex of G. This is just $d^w(v_j, v_k)$, as sought. □

6.2 The Chinese Postperson Problem

We now apply Floyd's Algorithm to solve the Weighted Traverse Question 2.5. Given any walk or circuit ω in G, we define its weight $w(\omega)$ to be $\sum_{e \in E(\omega)} w(e)$, so that we may rephrase 2.5 as the determination of a *lightest* full circuit in G.

We shall again need to study matching sets of walks in G, and we refer to

$$w(M) = \sum_{\omega \in M} w(\omega)$$

as the weight of such an M, and call M lightest if it has minimum possible weight in G. If the weight of every edge of G is 1, then lightest is equivalent to shortest. Similarly, if H is a subgraph of G we let

$$w(H) = \sum_{e \in E(H)} w(e)$$

denote the weight of H; in particular, this applies to the case when H is the whole of G.

It is useful to recognise certain fundamental properties satisfied by lightest matching sets of walks.

Lemma 6.8 *Given a graph G with $2r$ odd vertices, then a matching set of walks is lightest only if each walk is a path and no two walks share a common edge.*

Proof Let M_{li} be a lightest matching set of walks. Suppose one of the walks ω fails to be a path, which means that some vertex x is repeated and ω therefore contains a subwalk of the form $x, e, \ldots, f, x$. We may then lighten ω by replacing this subwalk with x alone, a procedure which does not alter the end points, and yields a matching set of walks with fewer edges. This contradicts the assumed minimality of M_{li}, so ω must have been a path.

Similarly, suppose that two walks have a common edge g, and may therefore be displayed as

$$u_1, \ldots, e_{s-1}, u_{s-1}, g, u_s, e_{s+1}, \ldots, u_2 \quad \text{and} \quad v_1, \ldots, f_{t-1}, v_{t-1}, g, v_t, f_{t+1}, \ldots, v_2$$

respectively. Note that $u_{s-1} = v_{t-1}$ and $u_s = v_t$ (by reversing the order of one of the walks, if necessary). We may then remove g from both, and obtain two new walks

$$u_1, \ldots, e_{s-1}, v_{t-1}, f_{t-1}, \ldots, v_1 \quad \text{and} \quad u_2, \ldots, e_{s+1}, v_t, f_{t+1}, \ldots, v_2.$$

These may be combined with the remaining walks of M_{li} to reduce its weight by $2w(g)$, and again contradict minimality. So a common edge g cannot exist. □

The converse of 6.8 is false, since there are many examples of matching sets of walks which satisfy the stated properties yet fail to be lightest.

Remark that the first part of the proof shows that the lightest walk between *any* two vertices u and v of G must always be a path; this is why, in the definition (6.1) of weighted distance, we need only consider paths. If g and h are multiple edges incident on the same vertices, the second part of the proof applies without change to show that no two walks in M_{li} can contain g and h respectively.

We may deduce from 6.8 that the weight of a lightest matching set of walks always satisfies $w(M_{\text{li}}) \leq w(G)$.

We are now in a position to adapt Theorem 5.14 so as to obtain a method for constructing a lightest full circuit in G.

Theorem 6.9 *Given a lightest matching set of walks M_{li} in G, there is a lightest full circuit γ_{li} such that*

$$w(\gamma_{\text{li}}) = w(G) + w(M_{\text{li}}),$$

and γ_{li} repeats only the edges of M_{li}.

Proof Apply 5.14 to M_{li} and let γ be the resulting full circuit. Then $w(\gamma) = w(G) + w(M_{\text{li}})$, and γ repeats only the edges of M_{li}; it therefore remains to show that γ is lightest.

Now apply Proposition 5.11 to *any* lightest full circuit γ_{li}, and let M be the resulting matching set of walks. Then $w(G) + w(M) \leq w(\gamma_{\text{li}})$. But $w(M_{\text{li}}) \leq w(M)$ by definition, so

$$w(\gamma) = w(G) + w(M_{\text{li}}) \leq w(\gamma_{\text{li}}).$$

Thus $w(\gamma)$ must be the same as $w(\gamma_{\text{li}})$, so that γ is also lightest. □

We can now answer the Chinese Postperson Problem by incorporating 6.9 into Algorithm 5.15 (SFC) with appropriate amendments.

Algorithm 6.10 (Chinese Postperson Algorithm: CPA)
INPUT: *a connected, weighted multigraph G with more than two odd vertices*
OUTPUT: *a lightest full circuit in G*

(1) *identify the odd vertices $\{v_1, \ldots, v_{2r}\}$*

(2) *run* FA *to compute all weighted distances $d^w(v_i, v_j)$*

(3) *obtain a lightest matching set of walks M from* (2)

(4) *$G^+ := G \cup_i \{m_i\}$ and $w(m_i) := w(\omega_i)$, where the edges m_i join the endpoints of each walk ω_i in M*

(5) *obtain an Eulerian circuit γ^+ for G^+ (by running* HA*)*

(6) *convert γ^+ to a circuit γ in G by substituting ω_i for each m_i*

(7) *output γ and stop*

Observe that we can also output the weight of γ according to the formula $w(\gamma) = w(G) + w(M)$.

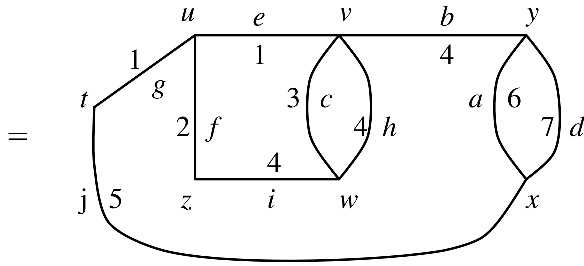
Just as with SFC, we shall always find a lightest matching set of walks in G by inspection; algorithms do exist for this purpose, but they are too sophisticated to describe here.

We now provide an example of 6.10 in action, by adapting 5.16.

Example 6.11 *The weighted multigraph*

$A(t)$	$(u,1),(x,5)$
$A(u)$	$(t,1),(v,1),(z,2)$
$A(v)$	$(u,1),(w,3),(w,4),(y,4)$
$A(w)$	$(v,3),(v,4),(z,4)$
$A(x)$	$(x,5),(y,6),(y,7)$
$A(y)$	$(v,4),(x,6),(x,7)$
$A(z)$	$(u,2),(w,4)$

=



represents a small railway system for which inspection data has to be collected; the weights give the travel times in hours for each branch line. Run CPA *to determine the shortest time in which the data can be collected for the entire system.*

Solution

(1) the odd vertices are $\{u,w,x,y\}$ (*by scanning the adjacency list*) 1

(2) the six weighted distances $d^w(v_i,v_j)$ are given by the table 2

	u	w	x	y
u	0	4	6	5
w	4	0	10	7
x	6	10	0	6
y	5	7	6	0

(*by running* FA*; or in this case, by inspection!*)

(3) the possible matching sets of walks, with their weights, are 3

M			$w(M)$
u,e,v,c,w	and	x,a,y	$4+6=10$
u,g,t,j,x	and	w,c,v,b,y	$6+7=13$
u,e,v,b,y	and	w,c,v,e,u,g,t,j,x	$5+10=15$

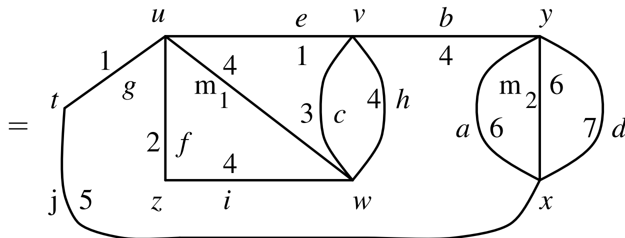
;

so choose u,e,v,c,w and x,a,y (*by inspection*)

(4) construct G^+ to be 4

$A(t)$	$(u,1),(x,5)$
$A(u)$	$(t,1),(v,1),(z,2),(w,4)$
$A(v)$	$(u,1),(w,3),(w,4),(y,4)$
$A(w)$	$(v,3),(v,4),(z,4),(u,4)$
$A(x)$	$(x,5),(y,6),(y,7),(y,6)$
$A(y)$	$(v,4),(x,6),(x,7),(x,6)$
$A(z)$	$(u,2),(w,4)$

=



(5) obtain the Eulerian circuit 5

$$\gamma^+ = u,f,z,i,w,h,v,c,w,m_1,u,g,t,j,x,a,y,d,x,m_2,y,b,v,e,u$$

for G^+ (*by running* HA*; or in this case by inspection*)

(6) convert γ^+ to 6

$$\gamma = u, f, z, i, w, h, v, c, w, c, v, e, u, g, t, j, x, a, y, d, x, a, y, b, v, e, u$$

(7) OUTPUT: $u, f, z, i, w, h, v, c, w, c, v, e, u, g, t, j, x, a, y, d, x, a, y, b, v, e, u$ 7

Since $w(G)$ is 37 and $w(M)$ is 10, we deduce that the minimum inspection time is 47 hours.

Since there are several possible choices for γ^+ at step (5), the lightest full circuit γ is not unique.

6.3 Weighted Spanning Trees

We now address the problem of finding a lightest spanning forest for a weighted multigraph G. If our initial graph is complete, this reduces to to the Weighted Spanning Trees Question 2.9.

It is instructive first to consider an arbitrary spanning subgraph H of G, on the understanding that each component of G is spanned by a connected component of H (for otherwise the vertices of G would always form a null spanning graph of zero weight).

Lemma 6.12 *Suppose that H is a lightest spanning subgraph of G; then H must be a forest.*

Proof Suppose that H contains a cycle. Then we may remove an edge from this cycle without destroying either the spanning property or the connectedness of H, and thereby reducing its weight. This contradicts the assumption that H is lightest, so no such cycle can exist; H is therefore a forest. □

This justifies our concentration on spanning forests. Of course, each component of a lightest spanning forest of G is a lightest spanning tree for the corresponding component of G.

In the following algorithm, (as for example in Algorithm 4.12 (PC2)) we build up a spanning forest F by adding an edge at a time, as described at the end of Section 2.3.

Algorithm 6.13 (Kruskal's Algorithm KA)
INPUT: *a weighted multigraph G*
OUTPUT: *a lightest spanning forest for G*

(1) $i := 1$ *and* $F := \emptyset$

(2) *choose $e_i \in E(G)$ to be a lightest edge such that $e_i \notin E(F)$ and $F \cup e_i$ contains no cycles; if no such edge exists, go to* (4)

(3) $F := F \cup e_i$ *and* $i := i + 1$*; then go to* (2)

(4) *output F, and stop*

Kruskal's Algorithm is extremely simple to operate on small diagrams by inspection, as we shall see in Example 6.14 below. However, implementing it on a machine, and proving that it does indeed output a lightest spanning tree are both considerably more subtle.

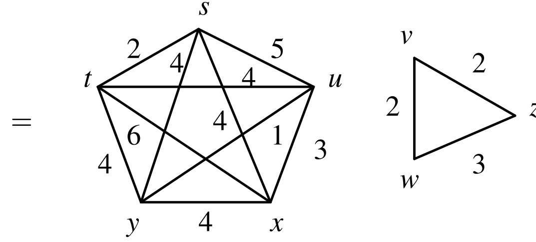
So far as implementation is concerned, we content ourselves with observing that step (2) is crucial; we have to call upon a subroutine which works on the adjacency list of each potential

$F \cup e_i$ in turn and checks whether or not there are any cycles present. Such a subroutine can easily be constructed and woven into the algorithm, and the result is still extremely efficient, but in order to concentrate on the central issues we shall not pursue this matter further. On the other hand, we *shall* give a proof that KA performs as claimed, since the proof is instructive, the outcome is by no means obvious, and we wish to be true to our stated mathematical principles.

Example 6.14 *Run* KA *on the weighted multigraph*

s	$(t,2),(u,5),(x,4),(y,4)$
t	$(s,2),(u,4),(x,6),(y,4)$
u	$(s,5),(t,4),(x,3),(y,1)$
v	$(w,2),(z,2)$
w	$(v,2),(z,3)$
x	$(s,4),(t,6),(u,3),(y,4)$
y	$(s,4),(t,4),(u,1),(x,4)$
z	$(v,2),(w,3)$

=



Solution

(1) $i := 1$ and $F := \emptyset$ 1

(2) choose $e_1 = uy$ 2

(3) $F := \{uy\}$ and $i := 2$; then go to (2) 3

(2) choose $e_2 = st$ 4

(3) $F := \{uy, st\}$ and $i := 3$; then go to (2) 5

(2) choose $e_3 = vw$ 6

(3) $F := \{uy, st, vw\}$ and $i := 4$; then go to (2) 7

(2) choose $e_4 = vz$ 8

(3) $F := \{uy, st, vw, vz\}$ and $i := 5$; then go to (2) 9

(2) choose $e_5 = ux$ 10

(3) $F := \{uy, st, vw, vz, ux\}$ and $i := 6$; then go to (2) 11

(2) choose $e_6 = tu$ 12

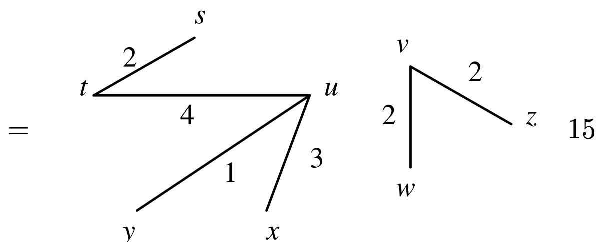
(3) $F := \{uy, st, vw, vz, ux, tu\}$ and $i := 7$; then go to (2) 13

(2) every edge creates a cycle; so go to (4) 14

4. OUTPUT:

s	$(t, 2)$
t	$(s, 2), (u, 4)$
u	$(t, 4), (x, 3), (y, 1)$
v	$(w, 2), (z, 2)$
w	$(v, 2)$
x	$(u, 3)$
y	$(u, 1)$
z	$(v, 2)$

=



15

Remark that the output is a forest of weight 14, and that it is not the only spanning forest with such weight.

It is important to learn from this example that KA may build up the forest in disconnected parts, even within the same component of G. Readers should also check (by exhausting all possibilities!) that 14 is indeed the minimum possible weight for a spanning forest. We now consider the theoretical justification for this fact.

Theorem 6.15 *The output F of* KA *is a forest, and is a lightest spanning forest for G.*

Proof We assume that we are given an arbitrary spanning forest D for G, and prove that $w(D) \geq w(F)$. We achieve this aim by constructing a sequence of spanning forests $D, D_1, \ldots, D_t$ for which $D_t = F$ and

$$w(D) \geq w(D_1) \geq \cdots \geq w(D_t) = w(F).$$

Consider the sequence of edges $\{e_1, \ldots, e_i, \ldots\}$ selected by successive implementations of step (2), so that $w(e_i) \leq w(e_j)$ whenever $i < j$. Let e_q be the first of these edges *not* in $E(D)$, and consider the graph $D \cup e_q$. This must now contain a cycle (since D spans G), and at least one edge of this cycle, say e_q^*, cannot lie in $E(F)$. If we form D_1 by deleting the edge e_q^* from $D \cup e_q$, we obtain a graph which

- is again a spanning forest for G
- satisfies $w(D_1) = w(D) + w(e_q) - w(e_q^*)$.

Informally, we have obtained D_1 from D by exchanging e_q^* for e_q.

According to the requirements of step (2), e_q is a lightest edge such that the graph with edge set $\{e_1, \ldots, e_{q-1}, e_q\}$ has no cycles; yet the graph with edge set $\{e_1, \ldots, e_{q-1}, e_q^*\}$ also has no cycles, being a subgraph of the spanning forest D. Hence $w(e_q) \leq w(e_q^*)$, and we deduce from above that $w(D) \geq w(D_1)$. By construction, D_1 and F now have e_q in common as well, and we may repeat the whole process by locating the first edge e_r in which they differ and using it to define D_2. We then iterate. The iteration must terminate, since r is necessarily greater than q and the sequence of edges is finite; therefore $D_t = F$ for some suitably large t. □

Kruskal's Algorithm is a typical example of a greedy algorithm, in which each step maximizes the short term gain without regard to any future penalty. Such an approach works beautifully in this case (and is often applied to real-world problems outside the realms of graph theory!), but can never be guaranteed in advance. Greedy algorithms can be exceedingly

simple to implement, even in situations where they are known not to provide optimal solutions, and for this reason there is an increasing belief that they deserve wider application, so long as it is possible to quantify (perhaps probabilistically) the extent of their potential failure. Such matters are the subject of intensive current research.

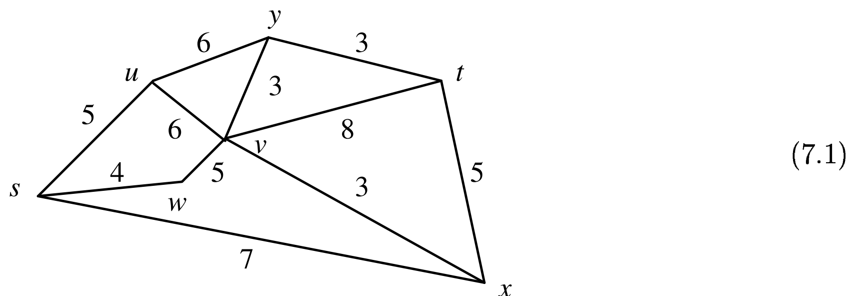
Chapter 7

Networks and Flows

7.1 Definitions

We now turn to problems concerning flow through networks, as embodied in our introductory Maximum Flow Question 2.10. We remind readers that practical applications of the relevant algorithms to the modern world are plentiful and diverse, covering, for example, the operation and management of fibre optics communication systems, transportation and distribution networks, and pipeline systems. Much of the mathematics we describe here has only been formalised since the 1950s.

We begin by establishing our notation and terminology, which is a straightforward task given the familiarity we have established in previous sections with the language of graphs and trees. To provide a point of reference, we centre the discussion around an example which exhibits the main features we seek to model:



(7.1)

This diagram represents a communications network, where s and t are respectively the source of, and target for, the transmitted information, and the other vertices are exchanges or relay stations whose function is simply to redistribute the incoming signals along the outgoing pathways. The connections have arrows to indicate the direction in which transmission can occur; two-way connections are therefore represented by a repeated edge, with one arrow in each direction. The maximum amount of information which each connection can handle (measured in suitable units such as calls per hour, or megabytes per second) is indicated by the label attached to each edge. Thus the flow of information along each connection cannot exceed this maximum, and the total flow in to each exchange is identical to the total flow out.

Let us consider 7.1 from a more formal viewpoint. It clearly consists of a weighted digraph (recalling terminology from Section 2.2) with two distinguished vertices. This suggests that we

define a network N to be a triple $(G, c, (s,t))$, where G is a connected digraph $(V(G), E(G))$, and $c\colon E(G) \to \mathbf{Z}^+$ is a capacity function assigning positive integral weights to the edges of G; in addition, (s,t) is an ordered pair of vertices in $V(G)$ known as the source and target respectively. We shall refer to the underlying graph of the digraph G as the underlying graph of the network. It is obtained by stripping the directions and weights from the edges of N, and forgetting that two of the vertices are distinguished.

A flow f in a network N is a real valued function $f\colon E(G) \to \mathbf{R}$ which satisfies the conditions of feasibility

$$0 \le f(e) \le c(e) \qquad \text{for all edges } e \tag{7.2}$$

and conservation

$$f_{\text{in}}(v) = f_{\text{out}}(v) \qquad \text{for all vertices } v \text{ other than } s \text{ and } t, \tag{7.3}$$

where

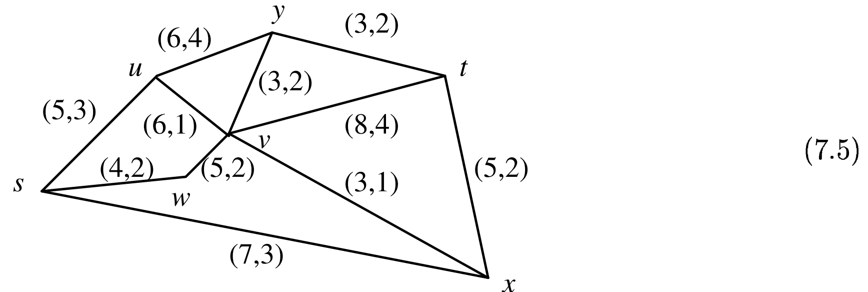
$$f_{\text{in}}(v) = \sum_{(u,v)\in E} f(u,v) \qquad \text{and} \qquad f_{\text{out}}(v) = \sum_{(v,w)\in E} f(v,w) \tag{7.4}$$

are the inflow and outflow at v respectively.

In expressions such as (7.4), we shall often abbreviate the summations to $\sum f(u,v)$ and $\sum f(v,w)$, on the understanding that f is only defined on those pairs of vertices which constitute a directed edge.

To store a flow on a machine, we have to extend the adjacency lists of the digraph G so as to incorporate an ordered *triple* for each directed edge. Thus each list $A(u)$ consists of triples (v, c, f), where v denotes the existence of a directed edge (u,v), c is its capacity, and f is the value of the flow along it. Since all our illustrative examples are small, we shall find it more convenient in this section to display networks and flows in terms of their diagrams, weighting each edge e by the pair $(c(e), f(e))$.

As a simple example of a flow, we have



(7.5)

in the network (7.1). Note that $f_{\text{out}}(s) = f_{\text{in}}(t) = 8$, and $f_{\text{in}}(s) = f_{\text{out}}(t) = 0$. Thus

$$f_{\text{out}}(s) - f_{\text{in}}(s) = f_{\text{in}}(t) - f_{\text{out}}(t) = 8;$$

in fact these two quantities agree for any flow in any network, as we shall demonstrate in Corollary 7.10 below.

Certain subdivisions of the vertices V of the network will play a prominent rôle in the theory. We define a cut in the network N to be a pair of subsets (S, T) of V with the properties that

- $S \cup T = V$ and $S \cap T = \emptyset$

- $s \in S$ and $t \in T$;

in other words, a cut is a division of the vertices into two nonoverlapping subsets, one containing the source and the other containing the target. The capacity of the cut is the integer

$$c(S,T) = \sum_{u \in S,\ v \in T} c(u,v),$$

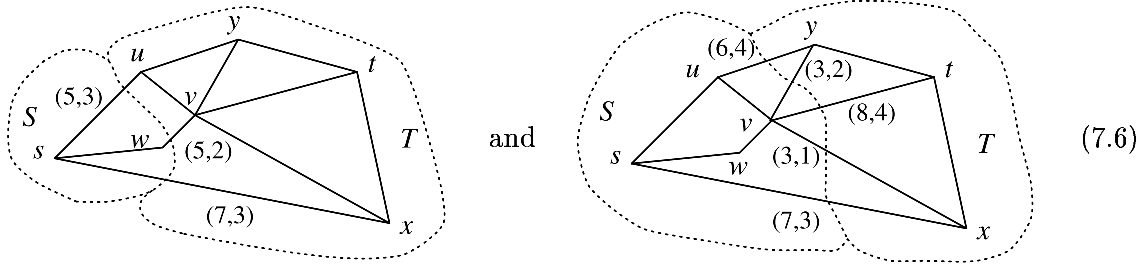
and the flow in the cut (given a flow f in N), is the real number

$$f(S,T) = \sum_{u \in S,\ v \in T} f(u,v).$$

This notation implicitly ensures that the only edges appearing in the sums are those which are directed from S to T in $E(N)$. An immediate consequence of feasibility is that $0 \leq f(S,T) \leq c(S,T)$ for any cut (S,T).

Of course, there may well be edges directed from T to S in N, and $f(T,S)$ is then defined in the corresponding manner.

Two examples of cuts in the network (7.1) are provided by



in which the flow f of (7.5) has also been assumed. In the first case $c(S,T) = 17$, whilst $f(S,T) = 8$ and $f(T,S) = 0$; in the second case $c(S,T) = 21$, whilst $f(S,T) = 11$ and $f(T,S) = 3$.

Let us now consider a cut (S,T) for which T contains at least one vertex x other than t, and suppose that a new cut (S',T') is obtained by transferring x to S from T. We may write the conservation of flow (7.3) at x in the form

$$\sum_{x \neq u \in V} f(u,x) = \sum_{x \neq w \in V} f(x,w), \tag{7.7}$$

and therefore as

$$\sum_{u \in S} f(u,x) + \sum_{w \in T'} f(w,x) = \sum_{w \in T'} f(x,w) + \sum_{u \in S} f(x,u), \tag{7.8}$$

an equation which we can utilise as follows.

Lemma 7.9 (The Transfer Lemma) *For any flow f and cut (S,T) in a network N, we have that*

$$f(S,T) - f(T,S) = f(S',T') - f(T',S').$$

Proof The left hand side of the equation may be rewritten as

$$\begin{aligned}\sum_{u\in S,\ w\in T}(f(u,w)-f(w,u)) &= \sum_{u\in S,\ w\in T'}(f(u,w)-f(w,u))+\sum_{u\in S}(f(u,x)-f(x,u))\\ &= \sum_{u\in S,\ w\in T'}(f(u,w)-f(w,u))+\sum_{w\in T'}(f(x,w)-f(w,x))\\ &\qquad\qquad\text{(from (7.8))}\\ &= \sum_{u\in S',\ w\in T'}(f(u,w)-f(w,u)),\end{aligned}$$

which is the right hand side of the equation, as required. □

The philosophy behind The Transfer Lemma is that the quantity

$$f(S,T)-f(T,S)$$

is unaltered by the transfer of a vertex to S from T.

Corollary 7.10 *For any flow f in N, we have that $f_{\text{out}}(s)-f_{\text{in}}(s)=f_{\text{in}}(t)-f_{\text{out}}(t)$.*

Proof Apply 7.9 repeatedly to the cut $S=\{s\}$ and $T=V\setminus\{s\}$, until all vertices except t have been transferred to S. Then $S'=V\setminus\{t\}$ and $T'=\{t\}$, and the result follows. □

The common value for $f_{\text{out}}(s)-f_{\text{in}}(s)$ and $f_{\text{in}}(t)-f_{\text{out}}(t)$ provided by 7.10 is known as the value of the flow, and written $\operatorname{val} f$. Thus the flow 7.5 has value 8.

Corollary 7.11 *For any flow f and cut (S,T) in N, we have that*

$$f(S,T)-f(T,S)=\operatorname{val} f \qquad \text{and} \qquad c(S,T)\geq \operatorname{val} f.$$

Proof Apply 7.9 repeatedly to the cut (S,T), until all the vertices except t have been transferred to S; then

$$f(S,T)-f(T,S)=\operatorname{val} f,$$

as sought. Now $f(T,S)\geq 0$, giving $f(S,T)\geq \operatorname{val} f$. The required inequality then follows from feasibility. □

Observe how the formulae of 7.11 are satisfied by the examples of (7.6). A major feature of the inequality is that it demonstrates how the value of f is constrained by the capacity of *every* cut in the network.

7.2 Maximum Flows and Minimum Cuts

Having learnt the appropriate language, we may now turn our attention to a fundamental problem in the study of networks, namely maximising the value of a flow. We know that such a maximum must exist and be finite, because the inequality

$$c(S,T)\geq \operatorname{val} f$$

of Corollary 7.11 confirms that the capacity of any cut in the network provides an upper bound. As is usual with such optimisation problems, there may be several different flows which all realise the maximum value. We call any one of them a maximum flow.

Since the number of cuts is finite, we may examine their capacities and determine a minimum value amongst them. Once more, several different cuts may realise this minimum, and we call any one of them a minimum cut. Thus if $f_{\max}$ is a maximum flow and $(S,T)_{\min}$ is a minimum cut, then

$$\operatorname{val} f_{\max} \leq c(S,T)_{\min} \tag{7.12}$$

follows from 7.11. We shall describe an elegant result (with attendant algorithm), due to Ford and Fulkerson in the 1950s, that these two quantities are actually equal. The purpose of this section is to introduce some preliminary ideas concerning paths in N.

Since the edges of N are directed, it is usual to insist that a path in N should respect these directions. We must now consider a more general concept, involving paths whose edges may oppose the directions given by N. To this end, let

$$\pi = x_0, e_1, x_1, e_2, \ldots, e_n, x_n$$

be a path in the underlying graph of N, and refer to it as a chain in N. Thus for any edge $e_i = x_{i-1}x_i$ in π, either (x_{i-1}, x_i) or (x_i, x_{i-1}) is an edge of N. In the former case we say that e_i is a forward edge of π, and in the latter case that it is a backward edge of π. This terminology simply reflects whether the arrow on the edge is consistent with, or opposed to, the direction of π.

We say that e_i is saturated by f if either it is a forward edge with $f(x_{i-1}, x_i) = c(x_{i-1}, x_i)$, or it is a backward edge with $f(x_i, x_{i-1}) = 0$; if e_i is not saturated we call it unsaturated. Every unsaturated edge has slack given by

$$\delta(x_{i-1}, x_i) = \begin{cases} c(x_{i-1}, x_i) - f(x_{i-1}, x_i) & \text{if } e_i \text{ is forward} \\ f(x_i, x_{i-1}) & \text{if } e_i \text{ is backward}. \end{cases}$$

By definition, the slack of any edge is always strictly positive.

We also refer to the chain π as being saturated if at least one edge is saturated, and unsaturated if every edge is unsaturated. In the latter case, we define the slack of π by

$$\delta(\pi) = \min_{e \in \pi} \delta(e).$$

Again, $\delta(\pi)$ is strictly positive for any unsaturated chain π. If π is an unsaturated chain from s to t (as opposed to a cycle, or a chain between any other pair of vertices), then we call it an f-augmenting chain; this terminology stems from the following observation.

Lemma 7.13 *If f is a flow in the network N and π is an f-augmenting chain, then we may define a new flow in N whose value is* $\operatorname{val} f + \delta(\pi)$.

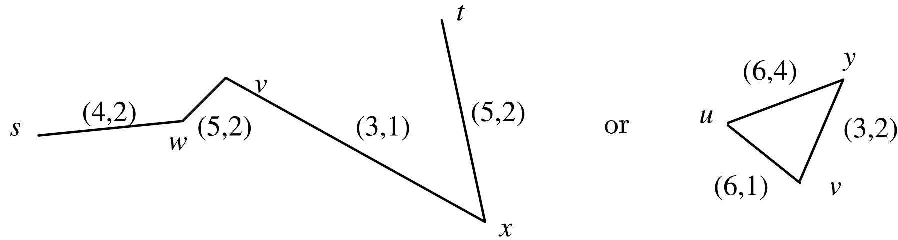
Proof Directly from the definition of slack, we may augment f by adding flow $\delta(\pi)$ along the forward edges of π and subtracting flow $\delta(\pi)$ along the backward edges of π without violating feasibility. This certainly increases the value of the flow as required, and it therefore remains to verify that conservation is satisfied.

The only vertices which require checking are those which lie in π (other than s or t), for which the flow is changed along exactly two incident edges. If both these edges are forward,

the flow in and the flow out of the vertex are each increased by $\delta(\pi)$, whereas if both are backward, the flow in and the flow out are decreased by $\delta(\pi)$. If one edge is forward and one is backward, the flow in and the flow out of the vertex are both unchanged. In all cases conservation is preserved, and we have indeed augmented the value of the flow by an amount $\delta(\pi)$. □

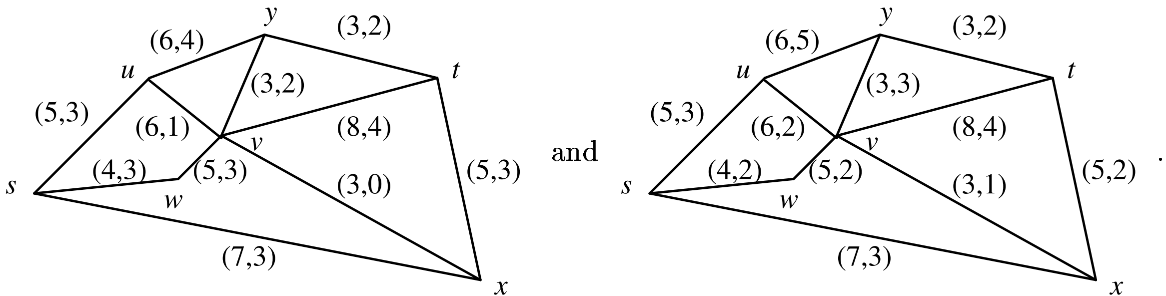
We label the flow provided by 7.13 as f_π, and say that it is obtained by augmenting f along π. Note that an unsaturated *cycle* can also be used to augment f in similar fashion, but that the value of the resulting flow will remain unchanged.

For example, if we again consider the flow f of (7.5) no edge is saturated, and any chain from s to t is therefore f-augmenting. Thus if we choose π to be either



both these chains have slack 1. The first chain is f-augmenting, and has three forward edges and one backward, whilst the second chain is *not* f-augmenting, and consists entirely of forward edges.

Following Lemma 7.13 we may use these chains to obtain the respective flows



The first of these has value $\operatorname{val} f + 1 = 9$, whereas the second has the same value as f, namely 8.

We can now prove the famous theorem of Ford and Fulkerson.

Theorem 7.14 (The Ford-Fulkerson Theorem) *Given a flow f in a network N which admits no f-augmenting chains, we have that*

$$c(S, T) = \operatorname{val} f$$

for some cut (S, T) in N.

Proof Define a subset $S \subset V(N)$ of the vertices of N to consist of all v which admit an unsaturated chain from from s to v, and let T be the complement $V(N) \setminus S$. We claim that (S, T) is then a cut. This follows since t lies in T; for if t belonged to S, there would be an f-augmenting chain, contradicting our assumptions.

Consider all vertices $x \in S$ and $y \in T$ such that either $(x, y) \in E(V)$ or $(y, x) \in E(V)$. If $(x, y) \in E(V)$, then $f(x, y) = c(x, y)$, for otherwise the unsaturated chain from from s to x could be extended to an unsaturated chain from s to y by appending the edge (x, y), and y could not then belong to T. On the other hand if $(y, x) \in E(V)$ then $f(x, y) = 0$, for otherwise we could in the same way construct an unsaturated chain from s to y.

We deduce that

$$\begin{aligned} f(S,T) &= \sum_{x \in S,\ y \in T} f(x,y) \\ &= \sum_{x \in S,\ y \in T} c(x,y) \\ &= c(S,T), \end{aligned}$$

and that

$$f(T,S) = \sum_{x \in T,\ y \in S} f(x,y) = 0.$$

Substituting in Corollary 7.11, we obtain

$$c(S,T) = f(S,T) - f(T,S) = \operatorname{val} f,$$

as sought. □

Corollary 7.15 *Any flow admitting no f-augmenting chains is maximum, and the corresponding cut defined in* 7.14 *is minimum.*

Proof This also follows from 7.11, which tells us that $c(S, T) \geq \operatorname{val} f$ for *any* flow f and cut (S, T). Thus if equality holds, the flow must be maximum and the cut must be minimum. □

Our aim, of course, is to use the Ford-Fulkerson Theorem for establishing an algorithm which constructs a maximum flow for any network N; this is the subject of the next section. Before we proceed, however, it is worth commenting that the Ford-Fulkerson Theorem has remarkable applications to other areas of graph theory, and even to combinatorial problems that do not appear (at first sight) to involve any graphs at all.

7.3 The Ford-Fulkerson Algorithm

This algorithm requires us to use the search procedures of Chapter 3. In almost all our examples it makes little difference whether we choose BFS or DFS, although one may be more efficient than the other in special circumstances (see Problem 33, Chapter 9). For consistency, we concentrate on BFS in this section.

Algorithm 7.16 (The Ford-Fulkerson Algorithm FFA)
INPUT: *a flow (usually the zero flow) in a network N*
OUTPUT: *a maximum flow and a minimum cut in N*

(1) *run* BFS *(with root s) to construct a search tree U of chains unsaturated by f which begin at s; if no such chains exist, $U := \{s\}$ and go to* (3)

(2) *if* U *reaches* t *along a chain* π, *then* $f := f_\pi$ *and go to* (1); *otherwise, go to* (3)

(3) *output maximum flow* f, *minimum cut* $(V(U), V(N) \setminus V(U))$, *and stop*

The fact that FFA does indeed output a maximum flow and a minimum cut follows immediately from Corollary 7.15, noting that the cut $(V(U), V(N) \setminus V(U))$ of the algorithm is identical to the cut (S, T) constructed in the proof of the Ford-Fulkerson Theorem 7.14.

The mechanics of the algorithm allows us to deduce one further interesting fact about maximum flows.

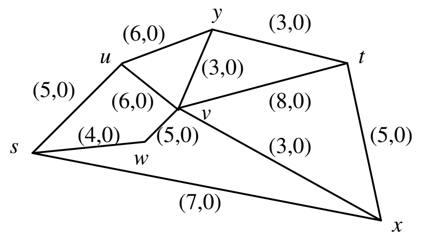
Proposition 7.17 *Every network* N *admits a maximum flow* f *such that* $f(e)$ *is an integer for all edges* e.

Proof Recalling our blanket assumption that all capacities $c(e)$ are integers, we simply run 7.16 by starting with the zero flow. The result is a flow with the required property. □

We give two examples of FFA in action, since it is of major importance and contains hidden subtleties which are worth illustrating.

First, let us take the network (7.1) and assume given the zero flow.

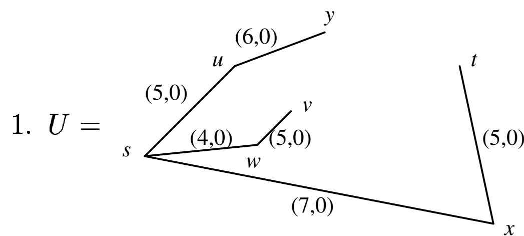
Example 7.18 *Run* FFA *on the flow* f *given by*



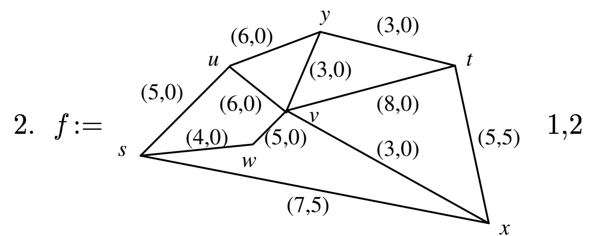
.

Solution

1. U =

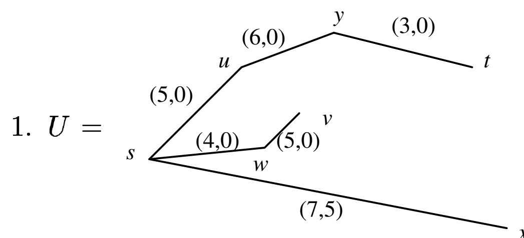


2. f :=

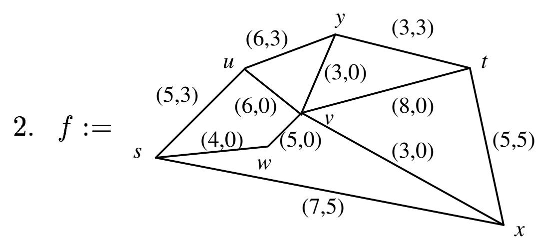


1,2

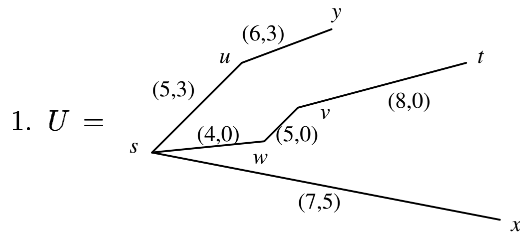
1. U =

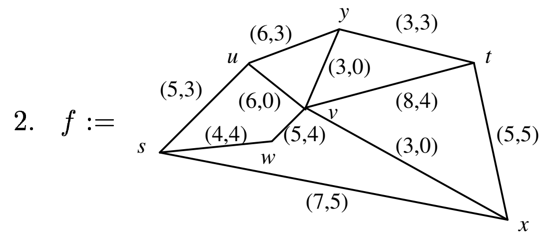


2. f :=

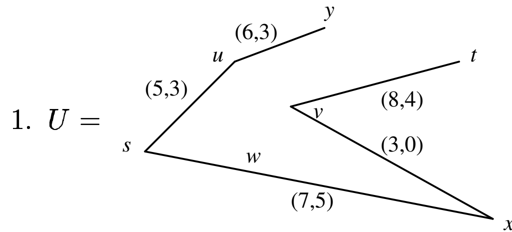


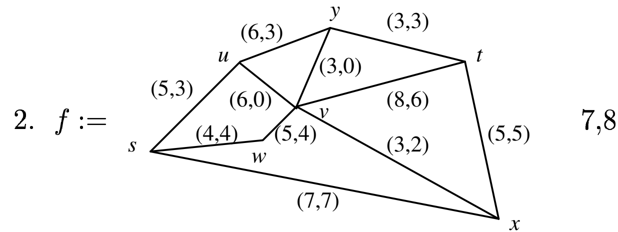
3,4

1. $U =$ 

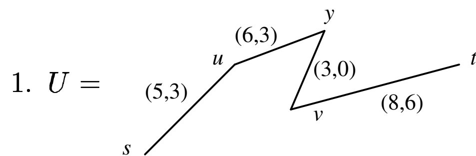
2. $f :=$ 

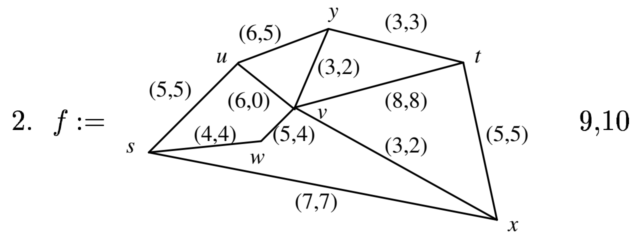
5,6

1. $U =$ 

2. $f :=$ 

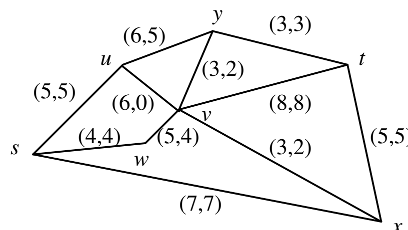
7,8

1. $U =$ 

2. $f :=$ 

9,10

1. $U = \{s\}$, so go to 3 11

3. OUTPUT: maximum flow $f =$ 

12

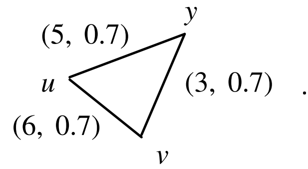
and minimum cut $(\{s\}, V \setminus \{s\})$

The value of the maximum flow and the capacity of the minimum cut are both 16; there is another obvious minimum cut, namely $(V \setminus \{t\}, \{t\})$.

It is tempting to believe that step 2 will always augment f along a chain consisting entirely of forward edges, as has happened in this example. In other words, we might expect FFA to build f up by repeatedly adding *elementary flows*, which are nonzero only along a chain of forward edges. This does occur in many simple cases, but is misleading and atypical; so our second example illustrates why the use of backward edges may be imperative.

Notice that the maximum flow we have obtained in 7.18 does have integral values on each edge, as predicted by Proposition 7.17. However, we can easily modify f so that this is no

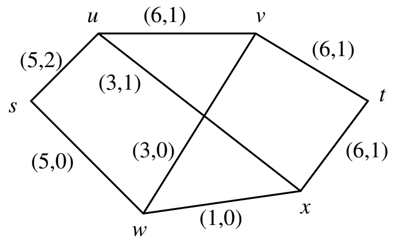
longer the case, by adding the elementary flow



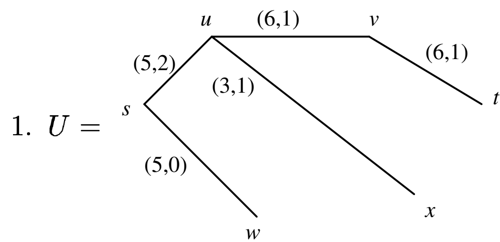
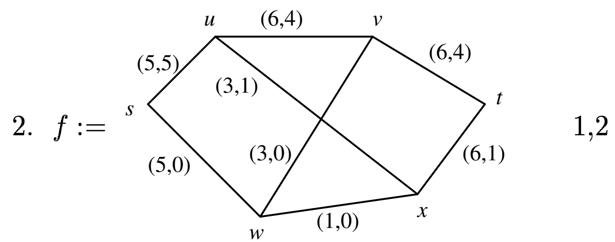
The resulting flow is still maximum, since the value 16 is unaltered.

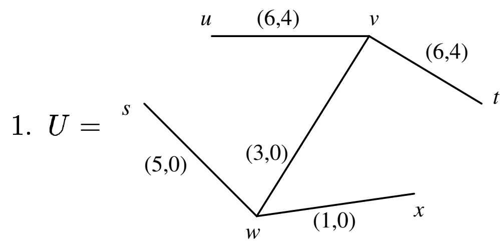
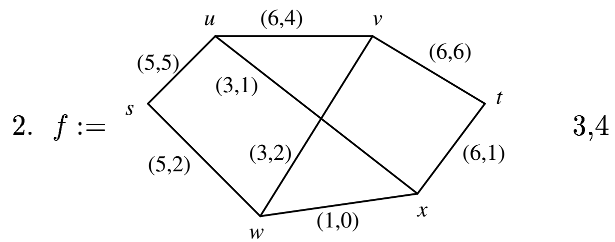
Now we consider our second example.

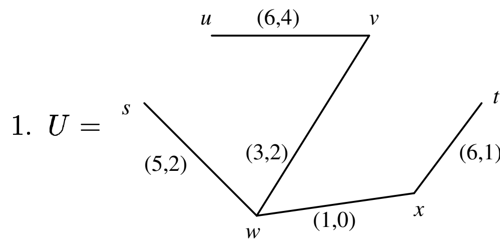
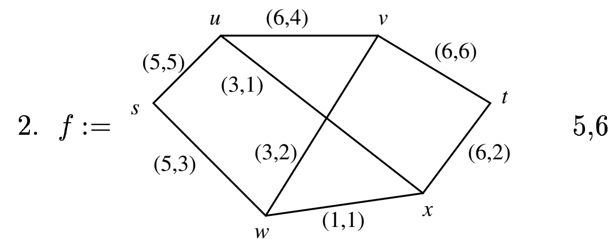
Example 7.19 *Run* FFA *on the flow f given by*

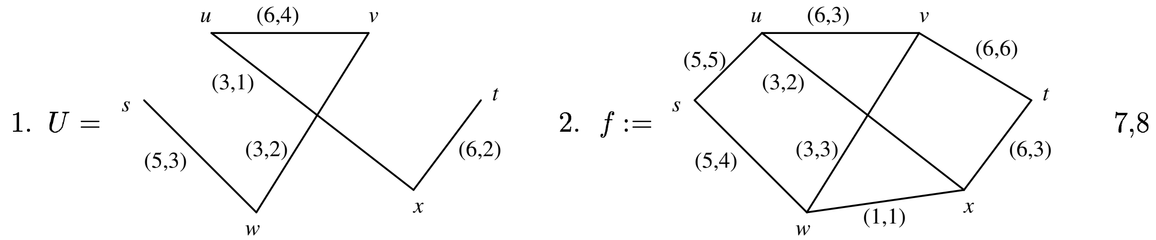


Solution

1. $U =$  2. $f :=$  1,2

1. $U =$  2. $f :=$  3,4

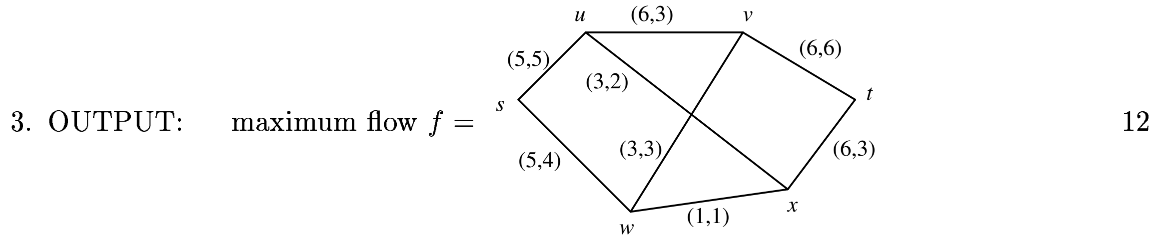
1. $U =$  2. $f :=$  5,6

1. $U =$ 

2. $f :=$ 7,8

1. $U =$

, so go to 3 9

3. OUTPUT: maximum flow $f =$ 

12

and minimum cut $(\{s, w\}, \{u, v, x, t\})$

The value of the maximum flow and the capacity of the minimum cut are both 9.

The crux of this example occurs at step 7, where the BFS search tree U and the corresponding f-augmenting chain from s to t both unavoidably contain the backward edge (v, u). This edge has slack 4, but the slack of the path is controlled by that of the forward edge (w, v), which is only 1.

In both our examples it is important to observe that there is choice in selecting the BFS search trees at step 2 of FFA; different choices may well lead to different maximum flows.

Chapter 8

Hamiltonian Tours and Complexity

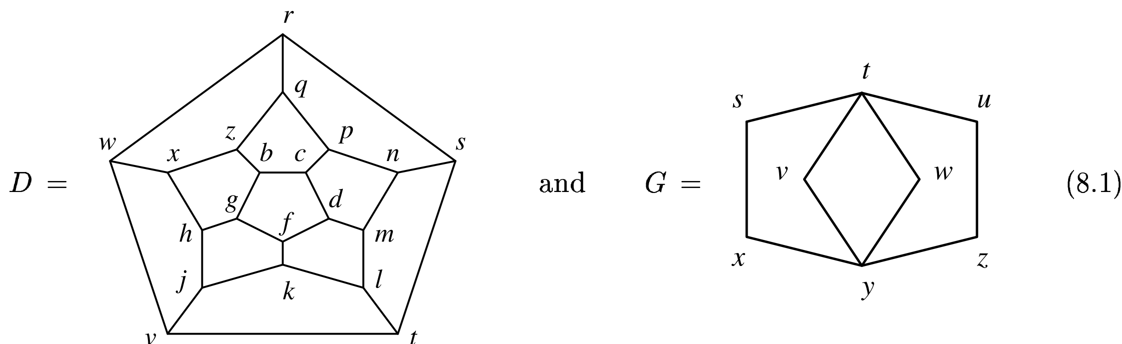
8.1 The Travelling Salesperson

We now turn to the Visitation Questions 2.6 and 2.7. Although superficially similar to problems concerning Eulerian tours, they actually raise fundamental problems of a wider nature which are currently unanswerable! These are exemplified by the search for an efficient algorithm with which to attack the Travelling Salesperson Problem.

In its simplest form, 2.6 is concerned with finding a tour which passes through every *vertex* of a given graph G exactly once. We refer to such a tour as **Hamiltonian**, in honour of Sir William Rowan Hamilton, the Irish mathematician who developed the concept in terms of a game, to be played on the vertices and edges of a regular dodecahedron, which was marketed in 1859. As with so much mathematics, however, the first published account of the problem was by a different author, in this case the Lancashire vicar Thomas P Kirkman, who introduced it in 1856.

Corresponding to the Eulerian case, we refer to a Hamiltonian tour as either a Hamiltonian path or a Hamiltonian cycle (noting that if no vertex is repeated in the tour, then nor is any edge). If a graph G admits a Hamiltonian tour, we may also call G Hamiltonian. We remark that multiple edges and loops can never be used in a Hamiltonian tour, and it therefore suffices to consider simple graphs throughout this section.

For example, of the two graphs



the first, or **dodecahedral** graph D, admits a Hamiltonian circuit by following the vertices alphabetically, whereas the second graph G allows no Hamiltonian tour at all. Notice that D has 20 odd vertices and no even ones! Observe also that every complete graph K_n admits a Hamiltonian circuit, simply by listing the vertices $v_1, \ldots, v_n$ in any order.

Ever since the original publicity, the study of Hamiltonian tours has been an active area of research and several conditions have been discovered which are *either* necessary *or* sufficient for a graph to be Hamiltonian; remarkably, there is no property which is yet known to be both, in stark contrast to the Eulerian case.

We shall discuss the Weighted Visitation Question 2.7 in the following important (and famous) form.

Problem 8.2 (The Travelling Salesperson Problem TSP) *Given a weighted complete graph (K_n, w), construct a lightest Hamiltonian cycle.*

The data for such a problem is usually given in the form of Picture 3 of section 2.1, which is really just the weighted adjacency list $A^w(K_n)$ in disguise. The weighting function may measure, amongst other possibilities, distance, cost, or time.

Since K_n does admit a Hamiltonian cycle, the essence of the Travelling Salesperson problem is to find an algorithm which constructs a lightest one. We may, of course, adopt the naive approach, and list *all* Hamiltonian cycles, compute their weights, and search for a minimum. However, there are $(n-1)!$ such cycles altogether, and if we suppose that $n = 50$ and that our machine is capable of carrying out 10^6 weight computations each second, it will take over 10^{49} years to compute all the weights. This is not acceptable!

In practice, no method yet exists for solving TSP in reasonable time, a situation which is bound up with the corresponding difficulty of finding necessary and sufficient conditions for a graph to be Hamiltonian. Amongst all the problems we have considered so far, these are the first for which no efficient algorithms are yet known. We shall discuss this issue more systematically in Section 8.2 below; in particular, we shall make a more formal definition of what we mean by efficiency.

For now, we introduce the concept of an approximate algorithm. Suppose that the weight of a solution to a given TSP is w_0, and that we have constructed an efficient algorithm whose output is *some* Hamiltonian cycle, with weight w in the given case. Thus $w/w_0 \geq 1$. Imagine further that we can always prove, no matter what the data, that

$$\alpha \geq w/w_0 \geq 1$$

for some universal constant α. Under these conditions, we say that our algorithm is an approximate algorithm for the TSP, with tolerance α. Once we have convinced ourselves that such algorithms exist, our aim then becomes to reduce the tolerance until it is as close to 1 as possible.

We shall present below a simple algorithm for which α is 2; it is based on Kruskal's Algorithm 6.13 and Depth First Search 3.23, and applies to a restricted class of TSPs satisfying a certain triangle inequality. This is not so feeble as it might seem, since the best algorithms currently available under the same restrictions have $\alpha \approx 3/2$. It is conjectured that these are the best values of α possible, although no proof has yet been found; such topics are the subject of much current debate amongst research workers.

We say that a TSP satisfies the Triangle Inequality if, for every pair of vertices u and v,

$$w(uv) \leq w(ux) + w(xv) \tag{8.3}$$

for all vertices x such that $u \neq x \neq v$. If the weights in question are distances, then this restriction will invariably hold. It will usually also be true for costs (special air or train fare deals sometimes excepted), and often for times. If we know by such external considerations

that (8.3) is satisfied, there is no need to check the data. If there is doubt, however, we will have to perform some preliminary computation to confirm (or confound) our suspicions.

Algorithm 8.4 (Twice Around the Lightest Spanning Tree TALST)
INPUT: *a* TSP (K_n, w) *satisfying the Triangle Inequality*
OUTPUT: *a Hamiltonian cycle of weight no more than twice the minimum possible*

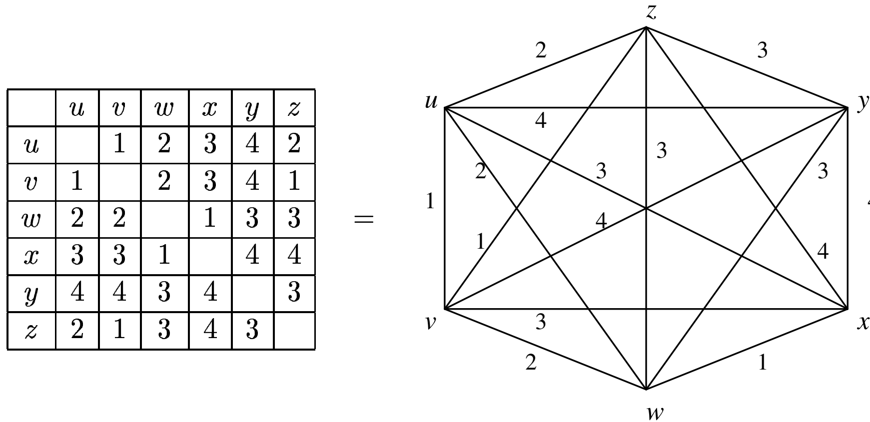
(1) *run* KA *to find a lightest spanning tree* T

(2) *run* DFS *on* T*, and let the resulting ordering of the vertices be* $v_1, v_2, \ldots, v_n$

(3) *output the Hamiltonian cycle* $v_1, v_2, \ldots, v_n, v_1$ *of weight* $\sum_i w(v_i v_{i+1})$*, and stop*

The reasons for the name given to this Algorithm will be explained by the proof of its properties below. First, we provide an example.

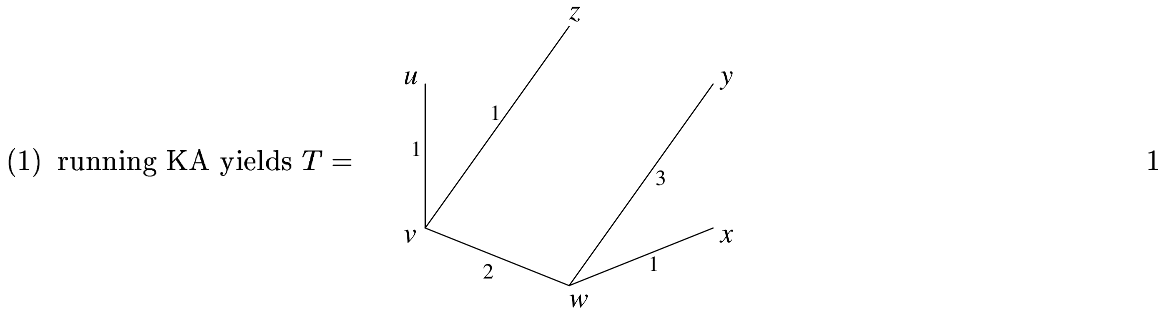
Example 8.5 *Run* TALST *on the* TSP



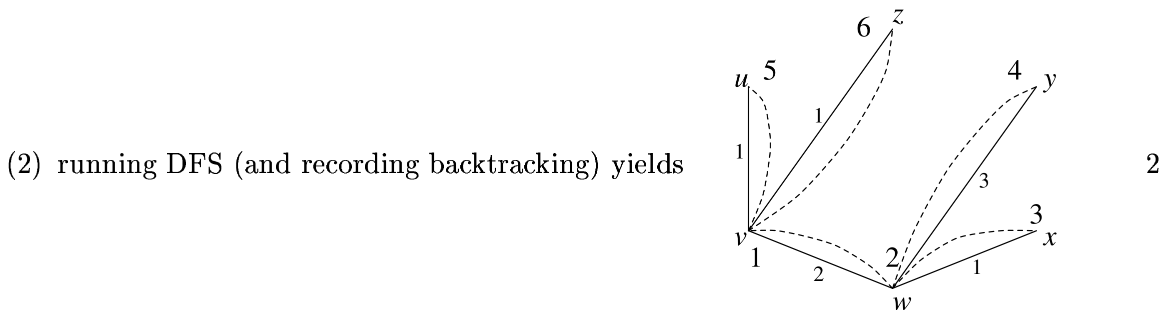
	u	v	w	x	y	z
u		1	2	3	4	2
v	1		2	3	4	1
w	2	2		1	3	3
x	3	3	1		4	4
y	4	4	3	4		3
z	2	1	3	4	3	

Solution We must confirm that the given data does indeed satisfy the Triangle Inequality. We assume this done by inspection, although 60 comparisons are necessary!

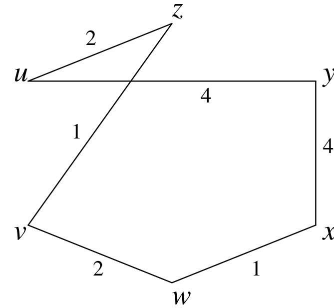
(1) running KA yields $T =$



(2) running DFS (and recording backtracking) yields



(3) OUTPUT: the Hamiltonian cycle $v, w, x, y, u, z, v =$



of weight 14.

Note that alternative choices are possible at each step of the algorithm, and that they may alter the weight of the output cycle. Moreover, the cycle given by 8.5 is not lightest; for example, u, v, w, x, y, z, u has weight 13.

Theorem 8.6 *For any* TSP *satisfying the Triangle Inequality, the output of* TALST *is a Hamiltonian cycle whose weight is at most twice the minimum possible.*

Proof Suppose that a lightest Hamiltonian cycle has weight w_0. Deleting an edge from this cycle must yield a spanning tree U such that $w(U) \leq w_0$. Hence the T provided by step 1 must satisfy

$$w(T) \leq w(U) \leq w_0.$$

After running step (2), DFS (together with backtracking) defines a circuit σ in T, and hence in K_n, which traverses every edge of T twice. This circuit therefore has weight

$$w(\sigma) = 2w(T) \leq 2w_0. \tag{8.7}$$

At step (3) the output is the cycle obtained by visiting the vertices of T in the order specified by DFS. This is a valid cycle since we are working with a complete graph, whence every possible edge is present. Moreover, it will be not be heavier than σ because each edge $v_i v_{i+1}$ in the cycle is either an edge of σ, or else replaces a subpath π in σ from v_i to v_{i+1}; in the latter case, $w(v_i v_{i+1}) \leq w(\pi)$ by a finite sequence of applications of the Triangle Inequality.

Hence by (8.7), the output cycle has weight at most $2w_0$, as required. □

Thus TALST is an approximate algorithm of tolerance 2.

8.2 Computational Complexity

We conclude our study of Hamiltonian cycles with an introduction to the theory of computational complexity; this is currently highly fashionable.

Whenever we run an algorithm to solve a problem on a machine, there are certain *space* requirements which have to be satisfied. The initial data, such as a weighted adjacency list, must be stored, and after each step of the algorithm additional information may have to be retained. As problems grow larger, the overall quantity of this information may exceed the available capacity of the memory.

Such considerations are obviously important and must be considered seriously for any proposed computer implementation. They are not, however, directly related to the efficiency

of the algorithm itself, which is primarily a matter of *time* requirements. As we saw when discussing the TSP, an algorithm which has to perform 49! operations cannot be expected to operate on any sensible timescale. We therefore establish a theoretical procedure for measuring the maximum number of computational steps which an algorithm involves, as a function of the input data. This is the province of computational complexity. We should observe that even when the complexity of an algorithm is known, the actual running time may still vary from machine to machine according to the speed with which each operation can be effected.

Suppose that an algorithm applies to a graph G with n vertices and p edges, and often with extra structure (such as directed edges or a weight function) as well. The complexity of the algorithm will normally depend on one or both of n and p. It is therefore important to note that when G is a *simple* graph, then p is at most $\frac{1}{2}(n^2 - n)$, the maximal case being provided by the complete graph K_n. If we know in addition that G is a tree then we have even better control over the value of p, which must be precisely $n-1$. At the other extreme, if G is a multigraph then p may be arbitrarily large and not expressible as any natural function of n at all.

We now make these considerations more precise by analysing the complexity of Floyd's Algorithm 6.4. We recall that it proceeds as follows:

INPUT: *a connected, weighted multigraph G with vertices $\{v_1, \ldots, v_n\}$*
OUTPUT: *$d^w(v_j, v_k)$ for every pair of vertices v_j and v_k*

(1) *$i := 1$ and $w_0(j,k) := w_0^G(j,k)$*

(2) *$w_i(j,k) := min\{w_{i-1}(j,k),\ w_{i-1}(j,i) + w_{i-1}(i,k)\}$ for all $1 \leq j,\ k \leq n$*

(3) *if $i = n$ go to (4); otherwise $i := i + 1$ and go to (2)*

(4) *output $w_n(j,k)$ for all $1 \leq j,\ k \leq n$ and stop*

Let us determine the number of operations entailed in running Floyd's Algorithm on a weighted graph of n vertices, supposing that the initial weight matrix has already been identified as part of the data. For each value of the counter i, step 2 is executed n^2 times as j and k range between 1 and n. Each such execution involves one addition and one comparison, and may be considered as a single operation requiring a fixed (and very small) amount of time; we therefore deem that step 2 consists of n^2 operations. Since step 3 increments the counter between 1 and n, we conclude that running the entire algorithm requires n^3 operations in all. We say that Floyd's Algorithm has order n^3, written $O(n^3)$; alternatively, we may declare that it has complexity $O(n^3)$. We remark in passing that the space requirements of the Algorithm may be enhanced by storing w_i only for so long as w_{i+1} is being computed.

For many algorithms, the number of operations turns out to be a more complicated polynomial such as

$$c(n) = An^3 + Bn^2 + Cn + D, \tag{8.8}$$

for certain constants A, B, C and D. In this situation, the dominating term is still n^3, in the sense that

$$c(n) \leq Kn^3 \tag{8.9}$$

for some suitably large constant K, such as $4\max(A, B, C, D)$, and all natural numbers n. We therefore say again that such an algorithm has complexity $O(n^3)$; indeed, (8.9) is the

definition of the statement that the polynomial $c(n)$ has order n^3. In any such analysis, constant terms such as D in (8.8) may clearly be ignored.

By way of illustration, suppose we incorporate the computation of the initial weight matrix w_0 into Floyd's Algorithm. This involves scanning every edge of G once, and (in the case of multiple edges) deciding whether or not to display its weight in w_0. This effectively requires p operations. Thus if G is simple, the number of additional operations involved is at most $\frac{1}{2}(n^2 - n)$, and we may write

$$c(n) \leq n^3 + \frac{1}{2}(n^2 - n);$$

in other words, the algorithm still has complexity $O(n^3)$. If G is a multigraph however, then p may swamp n^3 in an uncontrollable fashion. It thus becomes more informative to say that Floyd's Algorithm has complexity $O(\max(p, n^3))$.

As our second example, consider the Breadth First Search Algorithm 3.9:

INPUT: *a graph G with root vertex u*
OUTPUT: *$d(u,v)$ for all v in G*

(1) $r := u$

(2) $i := 0$ *and label r with i*

(3) *find all unlabelled v adjacent to a vertex with label i; if none, go to* (6)

(4) *label all vertices v from* (3) *with* $i + 1$

(5) $i := i + 1$ *and go to* (3)

(6) *label all unlabelled vertices with* ∞

(7) *output the labelled vertices, and stop*

It is not so straightforward to go through this algorithm step by step and calculate the number of operations required. On the other hand, it is easy to make such an estimate by considering the algorithm globally. Every operation in steps (3) and (4) may be considered as searching the edges incident on a specific vertex, and either labelling or not labelling the other ends. In this way, every edge is considered exactly twice, once from either end, giving $2p$ operations in all. Steps (5) and (6) both involve at most n operations each, and steps (1) and (2) require a constant number of operations, namely 3. If G is a simple graph, we conclude that

$$c(n) \leq 2p + 2n + 3 \leq n^2 + n + 3,$$

and that BFS has complexity $O(n^2)$. As with our second analysis of Floyd's algorithm, if G is a multigraph we must be more precise and give the complexity as $O(\max(p, n))$.

We decree that an algorithm is efficient if its complexity is $O(m^k)$ for some k, where m is a parameter (such as n or p) which quantifies the input data. Effectively this means that running the algorithm requires a polynomial number of operations, and our terminology is based on the assumption that a modern computer can handle such a situation in an acceptable amount of time. To justify this assertion, we offer the following table. It lists the total time of computation for the given complexities and input sizes, assuming one million operations per second; the units are seconds s, minutes m, hours h, years y, and centuries c. The lower

entries in the table record the alarming speed with which nonpolynomial algorithms can become umanageable.

$c(n)$ \ n	10	30	50	100
n	10^{-5} s	3×10^{-5} s	5×10^{-5} s	10^{-4} s
n^2	10^{-4} s	9×10^{-4} s	25×10^{-4} s	10^{-2} s
n^5	10^{-1} s	24.3 s	5.2 m	2.7 h
2^n	10^{-3} s	17.9 m	35.7 y	2.8×10^{14} c
3^n	5.9×10^{-2} s	6.5 y	2×10^{8} c	2.5×10^{33} c
$n!$	3.6 s	8.4×10^{16} c	9.6×10^{48} c	3.0×10^{142} c

All the algorithms that we have considered are efficient, although verification of this property may be awkward in certain cases. We should emphasise this point with reference to examples such as the Ford-Fulkerson Algorithm 7.16, which calls upon Breadth First Search as one of its subroutines. It is in fact not difficult to show that an algorithm which appeals a polynomial number of times to an efficient subroutine is itself efficient (so long as no nonpolynomial sequence of operations is involved in other steps). Thus for networks with underlying simple graphs, BFS has complexity $O(n^2)$, whilst FFA, carefully implemented, may be shown to have complexity $O(n^5)$.

Suppose that we have a problem in graph theory, such as any one of those described in Section 2.1, which can be solved algorithmically. If there is an efficient algorithm which applies to *any* instance of the problem, we say that the problem is polynomial; otherwise, we call it intractable. It has become traditional (since this subject was first formalised in the early 1970s) to let P denote the class of all polynomial problems.

There are many problems for which no efficient algorithm has yet been found; all the questions we have discussed concerning Hamiltonian circuits fall into this category. But that does *not* imply that they are intractable, since we may simply have been insufficiently ingenious in our search for an algorithm. To show that the Travelling Salesperson Problem, for example, is not polynomial, we have to develop a proof which shows that no efficient algorithm can possibly exist. In spite of intense effort over the last twenty years, no-one has yet managed to demonstrate this. From time to time there is a stir in the world of complexity theory, when some research workers boldly announce that they have found an efficient algorithm (which could probably earn them millions of dollars) for solving the TSP; but none of these has yet survived public scrutiny. There are several other famous problems whose status is similarly unclear, and the best that we can currently say is that any one of them lies in P if and only if they all lie in P. The general belief is that they are all intractable.

We can now put our approximate algorithms, such as Twice Around the Lightest Spanning Tree 8.4, into perspective. In fact TALST is efficient, since it operates by calling upon Kruskal's Algorithm and Depth First Search once each, and these have complexity $O(n^3)$ and $O(n^2)$ respectively. There are algorithms for solving the TSP precisely, using methods such as *backtracking* and *branch and bound*, but they have essentially exponential complexity.

Even when we know that a problem lies in P, there is still the interesting question of determining the most efficient algorithm for solving it. For example, we have indicated above that the Ford-Fulkerson Algorithm has complexity $O(n^5)$; but algorithms with complexity $O(n^3)$ can also be constructed for finding a maximum flow and minimum cut in a network,

although they are considerably more difficult to describe. It is a notable fact that the efficiency of several other of our algorithms may likewise be improved, but only at the expense of what we perceive to be simplicity. There may be some fundamental and philosophical discord between those notions of complexity which are best suited to machine, and to human, experience!

Chapter 9

Problems

These problems are intended to provide relatively straightforward practice in applying (and occasionally extending) the most important aspects of the theory; those marked * may provide a little more excitement than the rest. They follow the order of the subject matter as it is developed in the text, and often contain a numerical pointer to the relevant algorithm or result.

Problem* 1

Draw a graph G corresponding to the Hampton Court Maze (Chapter 1, Picture 4; smaller maze) as follows: let $V(G) = \{A, \ldots, M\}$ consist of the junctions and dead-ends, and let $E(G)$ consist of those pairs which are adjacent when traversing the maze. Then solve the maze by studying G. Can you do the same for the other maze (which is at Chevening, in Kent)?

Problem 2

Give four distinct digraphs with $G\langle S_1\rangle$ as underlying graph, and eight with $G\langle S_2\rangle$ (see Example 2.15). Does this exhaust all possibilities? Deduce that there are 1024 distinct digraphs with Example 2.13 as underlying graph (of which Example 2.14 is one), and then give a formula for the general case.

Problem 3

Construct a diagram for the graph whose adjacency list is

$A(p)$	t, u, x
$A(q)$	y
$A(r)$	s, u, w
$A(s)$	r, w
$A(t)$	p, v
$A(u)$	p, r, w
$A(v)$	t, x, z
$A(w)$	u, r, s
$A(x)$	p, v
$A(y)$	q
$A(z)$	v

Problem 4

Draw up a table for the number of edges in the complete graph K_n, for $1 \leq n \leq 6$ (see Example 2.16). Check that your answers are of the form $n(n-1)/2$, and prove this formula for general n (one possibility is by induction; but you may discover another method as well).

Problem* 5

Suppose that H is a graph which has p edges and n vertices (none of which are isolated), where $0 < p < n$.

- Give examples of three such graphs for which $p = 4$ and $n = 6$ both hold. Are any two of these isomorphic?
- Show from Proposition 2.17 that $p \geq n/2$, and that H has at least $2(n-p)$ vertices of degree 1
- Show that H has at most $n-1$ vertices of degree 1, unless $2p = n$; what happens then?

Problem 6

Improve Algorithm 2.21 (BE) so as to save fuel if the diners decide to stay in bed instead of having breakfast.

Problem 7

Show that the distance function $d(u, v)$ (as defined by (3.5)) satisfies

- $d(u, v) = d(v, u)$
- $d(u, v) = 0 \Longrightarrow u = v$

- $d(u,w) \leq d(u,v) + d(v,w)$

for any vertices u, v, and w in $V(G)$. (These properties mean that d is a *metric*).

Problem 8

Run Algorithm 3.9 (BFS) on the graph G of Problem 3, with r as root. Describe two paths from r to z of shortest length, and then construct a longest path from r to z. Is your longest path unique? Find a path from r to z which is neither shortest nor longest.

Problem 9

Run Algorithm 3.21 (EBFS) on the graph

$G =$

$A(p)$	t, x
$A(q)$	y
$A(r)$	s, u, w
$A(s)$	r, w
$A(t)$	p, v
$A(u)$	r, w
$A(v)$	t, x, z
$A(w)$	u, r, s
$A(x)$	p, v
$A(y)$	q
$A(z)$	v

Summarise your execution in terms of a search forest. How many components does G have ?

Problem* 10

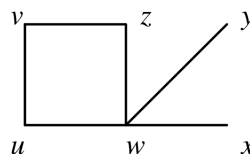
Improve Algorithm 3.9 (BFS) so that it outputs a search tree, as well as distances from the root vertex; apply the same technique to Algorithm 3.21 (EBFS).

Problem 11

Run Algorithm 3.23 (DFS) on the graph of Problem 3 with r as root, and summarise your execution in terms of a spanning tree.

Problem* 12

Run Algorithm 3.23 (DFS) on the graph



with u as root, and explain carefully why your output must be one of exactly 8 orderings of the vertices. Deduce that these outputs make up less than 7% of *all possible* orderings of the vertices which start with u.

Problem 13

Describe a rooted graph G on four vertices for which BFS and DFS necessarily search the vertices in identical order; generalise your answer to n vertices, giving reasons.

Problem* 14

Using Proposition 2.17, prove that any tree with n vertices has at most $n-1$ leaves. Then

- give examples of such a tree with t leaves, for every $2 \leq t \leq n-1$
- show that this tree is unique (up to isomorphism) for $t = 2$ and $n-1$
- show for the other values of t that there are at least $t/2$ distinct such trees (again up to isomorphism).

Discuss how your analysis depends on the value of n being sufficiently large.

Problem 15

Run Algorithm 4.4 (PC1) on the tree

$A(1)$	2
$A(2)$	$1,5$
$A(3)$	6
$A(4)$	8
$A(5)$	$2,6,8$
$A(6)$	$3,5$
$A(7)$	8
$A(8)$	$4,5,7$

.

Problem* 16

Run Algorithm 4.12 (PC2) on the Prüfer code (3,2,5,2,2,4): then run Algorithm 4.4 (PC1) on your result, and confirm that you recover the original code.

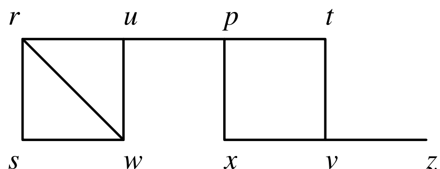
Problem 17

- Explain in two different ways (first by removal, second by addition) how to modify the system of bridges over the Pregel so that the Königsbergers may enjoy an Eulerian circuit of their city.
- Show that there is only one tree on n vertices which admits an Eulerian tour.

Problem 18

Explain why each walk in a *shortest* matching set must always be a path, and why two distinct such walks can never share a common edge.

Construct a matching set of walks M for the graph



such that each walk is a path and no two walks share a common edge, yet M is not shortest. What is a *longest* such M with these properties?

Problem* 19

Is it possible to draw the design below without lifting pen from paper? If so, do it; if not, why not?

Problem 20

Condense the running of Hierholzer's Algorithm HA on Example 5.9 into a tabular display. You may find it helpful to use column headings such as **step**, r, ω, σ, and G; you should consider abbreviating Walk( ) to a statement of output, and keeping track of G diagramatically, so long as your table unambiguously includes all the information involved in running the algorithm.

Problem 21

By running Algorithm SFC (5.15), find a shortest full circuit in the multigraph

$A(t)$	u, v, x
$A(u)$	v, z, t
$A(v)$	u, w, w, t
$A(w)$	v, v, x, x, z
$A(x)$	w, w, y, t
$A(y)$	x, z, z
$A(z)$	u, w, y, y

.

Describe a longer full circuit.

Problem22

With reference to Problem 7, investigate which of the properties

- $d^w(u, v) = d^w(v, u)$
- $d^w(u, v) = 0 \Longrightarrow u = v$
- $d^w(u, w) \leq d^w(u, v) + d^w(v, w)$

are necessarily satisfied by every weighted distance function $d^w(u, v)$ (as defined by (6.1)). Is your answer affected if the weight function takes the value 0? Is d^w a metric?

Problem* 23

Suppose that we allow the weight function w on the graph G to take *negative* values. Explain why there is no lightest walk between any pair of vertices if and only if G includes a cycle of negative weight.

Problem 24

Run Floyd's Algorithm FA (6.4) on the weighted multigraph

$G =$

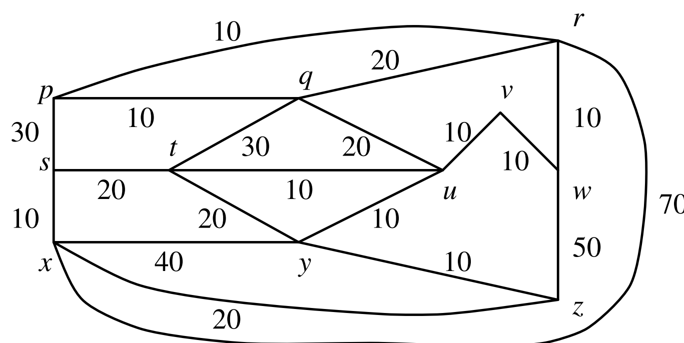
v_1	$(v_1, 3), (v_3, 2), (v_4, 1)$
v_2	$(v_3, 3), (v_4, 2), (v_4, 3)$
v_3	$(v_1, 2), (v_2, 3), (v_4, 1)$
v_4	$(v_1, 1), (v_2, 2), (v_2, 3), (v_3, 1)$

.

Give two vertices of G which are the greatest possible weighted distance apart.

Problem* 25

A postwoman delivers letters from the sorting office s in a neighbourhood whose street plan (together with traversal times) is given by the weighted multigraph



By running Algorithm CPA (6.10), determine the quickest possible route for the postwoman to deliver her letters, starting and finishing at s. Find a slower route which duplicates less streets.

Problem 26

Suppose the following table lists the nodes of a proposed LAN, together with data (u, c) indicating possible connections and their construction costs (in K£);

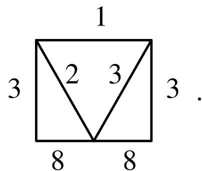
t	$(u, 6), (w, 4)$
u	$(t, 6), (v, 4), (x, 1), (y, 2)$
v	$(u, 4), (w, 6), (x, 3), (y, 4), (z, 1)$
w	$(t, 4), (v, 6), (y, 6), (z, 6)$
x	$(u, 1), (v, 3), (y, 1)$
y	$(u, 2), (v, 4), (w, 6), (x, 1), (z, 4)$
z	$(v, 1), (w, 6), (y, 4)$

.

By running Kruskal's Algorithm KA (6.13), find the cheapest LAN which can be built to connect these nodes.

Problem* 27

Adapt Kruskal's Algorithm KA so that it uses the greedy method of constructing a spanning *path* for a connected, weighted graph G. Show that your algorithm fails to find the *lightest* spanning path for the graph



.

Problem 28

Let N be a network whose underlying graph G has p edges, and adapt the proof of Proposition 2.18 to show that $\sum_{v \in V(G)} d_{\text{in}}(v) - \sum_{v \in V(G)} d_{\text{out}}(v) = 0$. If N admits a flow i which assigns the value 1 to every edge of G, show in addition that $d_{\text{in}}(v) - d_{\text{out}}(v) = 0$ for every vertex of G except s and t. In these circumstances, how are the in and outdegrees of s and t related to each other, and to val i?

Problem* 29

A flow in a network which has neither a source nor a target is known as a *circulation*:

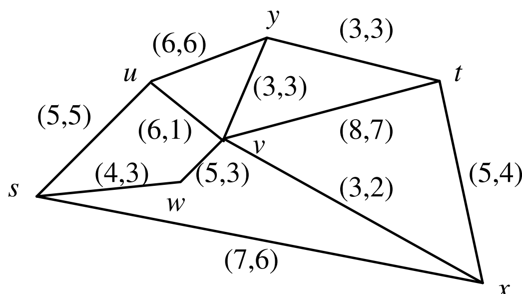
- describe how a circulation may always be constructed from a flow f in any network N by the addition of a single edge to N (take care to explain the rôle of val f)
- apply your construction to the flow of (7.5)
- discuss the reverse procedure for constructing a flow out of a given circulation.

Problem 30

Given the flow f of (7.5), find a cut (S, T) satisfying $f(S, T) = 11$ and $f(T, S) = 3$. Compute the capacity of your cut, and explain how these three statistics illustrate Corollary 7.11.

Problem 31

Run the Ford-Fulkerson Algorithm (7.16) on the flow



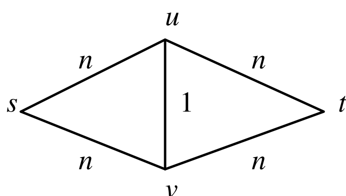
to find a maximum flow and minimum cut in the network. Compare your results with Example 7.18, and explain the difference.

Problem 32

If the capacity $c(e)$ of every edge in a network N satisfies $c(e) = k$ for some integer k, explain why the maximum flow in N must have value nk for some integer $n \geq 1$. Describe networks with 19 edges (none of them multiple) for which $n = 1$, $n = 2$, and $n = 10$ respectively.

Problem* 33

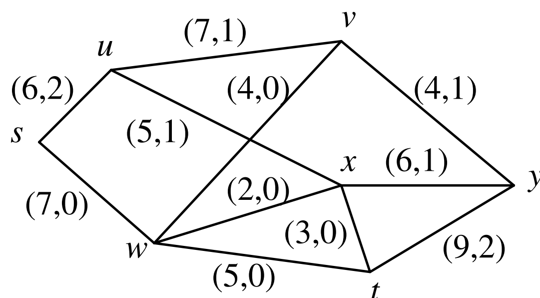
Consider running the Ford-Fulkerson Algorithm (7.16) on the network



using DFS instead of BFS at step (1). Show that $2n$ repetitions of steps (1) and (2) *may* be required before a maximum flow is identified; contrast this with the situation when BFS is used.

Problem 34

Run the Ford-Fulkerson Algorithm (7.16) on the flow



to find a maximum flow and minimum cut in the network.

Problem 35

Show how to remove $n(n-3)/2$ edges from a complete graph K_n so that the resulting graph

- has a Hamiltonian cycle
- does not have a Hamiltonian cycle

respectively.

Problem* 36

Suppose that a graph C consists of a single cycle on n vertices, and let $W \subseteq V(C)$ be a subset of vertices containing $|W| \leq n$ elements. Show (by induction on $|W|$, or otherwise) that the number of components $\kappa(C \setminus W)$ of the graph $C \setminus W$ does not exceed $|W|$.

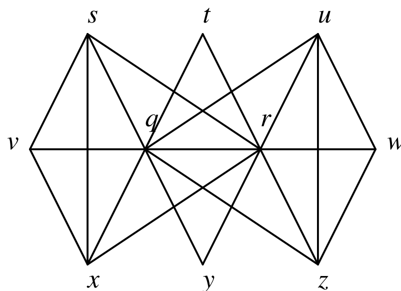
Hence show that any graph G which admits a Hamiltonian cycle must satisfy

$$\kappa(G \setminus W) \leq |W|$$

for *all* subsets $W \subseteq V(G)$. Apply this result to confirm that the graph G of (8.1) has no Hamiltonian cycle.

Problem 37

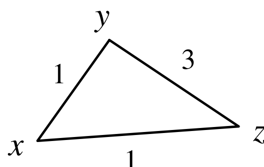
Strengthen the result of Problem 36 so as to it give a necessary condition for the existence of a Hamiltonian path. Deduce that the graph



has neither a Hamiltonian path nor a Hamiltonian cycle.

Problem 38

Explain why the TSP



fails to satisfy the triangle inequality TI, and construct a *circuit* which visits every vertex at least once, yet is lighter than the lightest Hamiltonian cycle. Prove that no TSP which does satisfy TI can admit such a circuit.

Problem 39

Run the Algorithm TALST (8.11) on the Travelling Salesperson Problem

	u	v	w	x	y	z
u		2	3	4	1	1
v	2		1	3	2	3
w	3	1		4	3	4
x	4	3	4		4	3
y	1	2	3	4		2
z	1	3	4	3	2	

,

checking first that the input conditions are satisfied. Can you find a lighter cycle than that produced by the Algorithm?

Problem 40

Estimate the complexity of the Algorithm PC1 (4.4).