# Discrete Mathematics 251

Semester 2021/2022

King's College London

Edited by Lassina Dembélé and Aled Walker
lassina.dembele@kcl.ac.uk and aled.walker@kcl.ac.uk

Version: April 27, 2022

# Contents

# Preface

These are the 2021/22 lecture notes for the module 5ccm251a. This document is an edited and moderately extended version of the notes written by Simon Salamon, the previous lecturer of this course, to whom we are greatly indebted.

It is likely that these notes will be updated week by week as the module is taught in the Winter/Spring 2022. For this reason, it is not recommended that they be printed.

We take this opportunity to also thank Naz Mihesi for his ongoing support in running the course. The module was successfully taught for many years by Simon Fairthorne. The current material is based on the module as he designed it. These lecture notes are dedicated to his memory.

Lassina Dembélé, Aled Walker
13 January 2022

# 1 Arithmetic

*Notation.* We will use the sets

$$\mathbb{N} = \{0, 1, 2, 3, \ldots\},$$
$$\mathbb{Z} = \{0, 1, -1, 2, -2, \ldots\}.$$

There are competing conventions here (most number-theory sources start the natural numbers $\mathbb{N}$ at 1 rather than at 0). The above will be our conventions in this course.

## 1.1 Induction

**First principle.** Let $\mathscr{P}(n)$ be some statement that makes sense for all $n \geqslant n_0$. (Typically, $n_0 = 0,\ 1$ or $2$.) Suppose that

(1) $\mathscr{P}(n_0)$ is true, and

(2) for all $n \geqslant n_0$, $\mathscr{P}(n)$ is true $\Rightarrow$ $\mathscr{P}(n+1)$ is true.

Then $\mathscr{P}(n)$ is true for all $n$.

**Example 1.1.** $\mathscr{P}(n)$ is the assertion that

$$1 + 3 + 5 + \cdots + (2n-1) = n^2.$$

Take $n_0 = 1$.

(1) $\mathscr{P}(1)$ is true because both sides equal 1.

(2) Now suppose that $\mathscr{P}(n)$ is true, and add $2n+1$ to both sides above to give

$$1 + 3 + 5 + \cdots + (2n-1) + (2n+1) = n^2 + (2n+1).$$

The right-hand side simplifies to $(n+1)^2$, so this is assertion $\mathscr{P}(n+1)$. Therefore $\mathscr{P}(n)$ must be true for all $n \geqslant 1$.

*Note.* Curly $\mathscr{P}$ emphasizes that $\mathscr{P}$ is a statement, not an arithmetical function.

**Second principle.** Same start as above. Suppose that

(1) $\mathscr{P}(n_0)$ is true, and

(2') for all $n \geqslant n_0$, $\mathscr{P}(k)$ is true for all $n_0 \leqslant k < n$ $\Rightarrow$ $\mathscr{P}(n)$ is true.

Then $\mathscr{P}(n)$ is true for all $n$.

**Example 1.2.** Take $n_0 = 2$. $\mathscr{P}(n)$ is the assertion "$n$ can be written as a product of (one or more) prime numbers".

(1) 2 is a prime number, so obviously $\mathscr{P}(2)$ is true.

(2') (i) If $n$ is prime, then $\mathscr{P}(n)$ is already true. (ii) If not, then $n$ has a divisor other than 1 and $n$, so we can write $n = ab$ with $1 < a < n$ and $1 < b < n$. If $\mathscr{P}(k)$ is true for all $k < n$ then $\mathscr{P}(a)$ and $\mathscr{P}(b)$ are both true, which means that $a$ is a prime or a product of primes, and $b$ similarly. The same must be true of $ab$, and $\mathscr{P}(n)$ is true.

Therefore any integer $n \geqslant 2$ is a product of primes.

**Summary.** Use the first principle when $\mathscr{P}(n+1)$ appears to depend only on $\mathscr{P}(n)$. The second is needed when $\mathscr{P}(n)$ or $\mathscr{P}(n+1)$ depends on more than one predecessor. But sometimes it becomes necessary to check more than one initial value.

**Example 1.3.** Prove that $a_n = 2^n + (-3)^n$ is a solution of

$$\begin{cases} a_n = 6a_{n-2} - a_{n-1}, & n \geqslant 2 \\ a_0 = 2,\ a_1 = -1. \end{cases}$$

We use the second principle with $\mathscr{P}(n)$ the assertion "$a_n = 2^n + (-3)^n$" for $n \geqslant 0$. Then $\mathscr{P}(0)$ is true, since $2^0 + (-3)^0 = 2$. Now assume that $\mathscr{P}(k)$ is true for all $k \leqslant n$. Then

$$\begin{aligned} a_n &= 6a_{n-2} = a_{n-1} \\ &= 6[2^{n-2} + (-3)^{n-2}] - [2^{n-1} + (-3)^{n-1}] \\ &= 6[2^{n-2} + (-3)^{n-2}] - [2 * 2^{n-2} - 3 * (-3)^{n-2}] \\ &= 4 * 2^{n-2} + 9^{n-2} \\ &= 2^n + (-3)^n, \end{aligned}$$

provided $n \geqslant 2$ (for the second line). The punch line is that we need to check $\mathscr{P}(1)$ separately, which is easily done: $2^1 + (-3)^1 = 2 - 3 = -1$. Thus, $\mathscr{P}(n)$ is true for all $n$.

*Notation.* To avoid confusion, we shall often indicate multiplication between actual numbers by $*$ as in common software.

## 1.2 Divisibility

*Notation.* Let $m, n \in \mathbb{Z}$. One says that $m$ divides $n$, abbreviated to $m \mid n$ if there exists an integer $q$ such that $mq = n$. For example,

$$13 \mid 0, \qquad \text{but} \qquad 0 \nmid 13.$$

Here is a formal

**Definition 1.4.** *A positive integer $p \geqslant 2$ is a* prime number *if $a \in \mathbb{N},\ a \mid p \Rightarrow a = 1$ or $a = p$.*

Let $a$ be any integer, and $b$ a *positive* integer. *There exist integers $q, r$ such that*

$$a = qb + r, \qquad 0 \leqslant r < b.$$

We shall take the validity of this statement for granted. It is sometimes called the *Division Algorithm*, and one can imagine a mechanical way of finding the *quotient* $q$ and the *remainder* $r$. Note that $b \mid a$ if and only if $r = 0$.

**Example 1.5.**

$$\begin{aligned} 23 &= 4 * 5 + 3 \\ -17 &= (-4) * 5 + 3 \\ 510510 &= 510 * 1001 + 0 \\ 104729 &= 104 * 999 + 833. \end{aligned}$$

One uses notation like

$$a = r \bmod b, \quad \text{or} \quad a \equiv r \ (b),$$

or a mixture. We shall adopt the former, so for example

$$104729 = 833 \bmod 999, \quad \text{also} \quad 104729 = 1 \bmod 104.$$

**Definition 1.6.** *Let $a, b$ be integers. Then, the* greatest common divisor *of $a$ and $b$, denoted by $\gcd(a,b)$, is the largest positive integer that divides both $a$ and $b$. It is undefined when $a = b = 0$. It is the same as* $\mathrm{hcf}(a,b)$, *the* highest common factor *of $a$ and $b$, and is sometimes abbreviated to $(a,b)$.*

If $\gcd(a,b) = 1$ then $a$ and $b$ are called *coprime.*

**Example 1.7.**

$$\begin{aligned} \gcd(24,15) &= 3 \\ \gcd(6,0) &= 6 \\ \gcd(-12,-24) &= 12 \\ \gcd(510510,44) &= 22 \\ \gcd(104729,10^9) &= 1. \end{aligned}$$

**Proposition 1.8.** *There exist integers $x, y$ such that $\gcd(a,b) = xa + by$.*

Later, we shall recall Euclid's algorithm that determines $x$ and $y$. The proposition has the following consequences:

**Corollary 1.9.** *If $m$ is* any *divisor of $a$ and $b$ and $n = \gcd(a,b)$ then $m$ divides $n$.*

*Proof.* This follows immediately from the formula $n = xa + yb$, since $m$ must divide the right-hand side. □

**Corollary 1.10.** *Let $p$ be a prime number. Then*

$$p \mid mn \implies p \mid m \quad \text{or} \quad p \mid n.$$

*Proof.* Suppose that $p \nmid m$. Then $(p,m) = 1$, since the only divisors of $p$ are 1 and $p$, but the latter does not divide $m$. So we can write $1 = xp + ym$. Thus

$$n = xpn + ymn,$$

and (since $p$ divides both terms on the right-hand side) $p \mid n$. □

## 1.3 Modular arithmetic

**Definition 1.11.** *We say that $a_1$ and $a_2$ are* congruent *(or* equal*)* modulo $n$ *if $n$ divides $a_1 - a_2$. In symbols,*

$$a_1 = a_2 \ \textit{mod}\ n \quad \Leftrightarrow \quad n \mid (a_1 - a_2).$$

*We'll sometimes write $a_1 \equiv a_2$ if $n$ has been fixed in advance.*

Because of the division algorithm (with $b = n$) we know that any integer is equal modulo $n$ to some remainder $r$ in

$$R = \{0, 1, 2, \ldots, n-1\}.$$

We can define addition and multiplication on this set by taking remainders modulo $n$, like on a clockface.

**Example 1.12.** With $n = 7$

$$\begin{aligned} 3 + 5 &= 1 \bmod 7 \\ 3 * 5 &= 1 \bmod 7 \\ 6 * 6 &= 1 \bmod 7 \\ 6 &= -1 \bmod 7 \end{aligned}$$

When we are working modulo $n$, an element $r \in R$ really represents *all* integers obtained from $r$ by adding or subtracting multiples of $n$, i.e. it represents the *set*

$$\{r + kn : k \in \mathbb{Z}\} = r + n\mathbb{Z}.$$

In the laguange of abstract algebra, $\mathbb{Z}$ is a ring, $n\mathbb{Z}$ is an *ideal*, and $R = \mathbb{Z}/n\mathbb{Z}$ is the *quotient ring* each of whose elements is a *coset* $r + n\mathbb{Z}$.

Since $R$ is a ring, almost all the usual laws of arithmetic apply: if $a = b \bmod n$ then

$$a + c = b + c, \quad ac = bc, \quad a^2 = b^2, \ \ldots \qquad \bmod n.$$

Beware though that one can have divisors of zero: the statement

$$ab = 0 \implies a = 0 \ \text{ or } \ b = 0 \ \bmod n$$

is *false* in general. For example, $2 * 3 = 0 \bmod 6$. But it is true if $n$ is a prime number:

**Proposition 1.13.** *Suppose that $n = p$ is a prime number, and that $p$ does not divide $a$. Then $a$ has an inverse modulo $p$.*

*Proof.* By assumption, $\gcd(a, p) = 1$ since the only factors of $p$ are 1 and $p$, and $p \nmid a$. By §1.2, we know that $xa + yp = 1$ for some $x, y \in \mathbb{Z}$. It follows that $xa = 1 \bmod n$, and we can suppose that $0 < x < n$. □

**Example 1.14.** To perform a sequence of operations, take remainders at each stage. Compute $15^8 \bmod 16$. Note that $15 = -1 \bmod 16$, so $15^8 = (-1)^8 = 1 \bmod 16$.

**Example 1.15.** Solve $2x = 2 \bmod 16$. This means

$$2x = 2 + 16k,$$

so $x = 1 + 8k$. There are two solutions modulo 16, namely 1 and $9 \equiv -7$.

Let $p \geqslant 2$ be a prime number. Then

$$R^* = R \setminus \{0\} = \{1, 2, \ldots p-1\}$$

is a *group* under multiplication modulo $p$, and $R$ itself is a *field* (a ring in which multiplication is commutative and has inverses). Let $a$ be an integer that is not a muliple of $p$. Its remainder modulo $p$ is an element of $R^*$, whose order (by Cauchy's theorem) divides $p - 1$. This implies

**Theorem 1.16** (**Fermat's little theorem**). *If $p$ is prime and $p \nmid a$, then $a^{p-1} \equiv 1$ mod $p$.*

We can include the possibility that $p \mid a$ by simply multiplying both sides by $a$:

$$a^p \equiv a \bmod p, \qquad \forall a \in \mathbb{Z}.$$

**Example 1.17.** Taking $p = 11$ and $a = 2$ gives

$$2^{10} = 1 \bmod 11,$$

which is easy to check immediately as $2^{10} = 1024$.

Note that

$$8^8 \equiv 1 \bmod 9,$$

because $8^8 \equiv (-1)^8 \bmod 9$, so taking $a = 8$ and $p = 9$ satisfies Fermat's little equation, even though $p$ is not prime. Even better:

Let $n = 561$, which is certainly not prime. Then it is known that

$$a^{561} = a \bmod n, \qquad \text{for } \textit{all } a \in \mathbb{Z},$$

which makes 561 a *Carmichael number* (it is the first).

**Proposition 1.18.** *If $p$ is prime, the only solutions of $x^2 = 1$ mod $p$ are $x \equiv 1$ and $x \equiv -1$.*

*Proof.* $x^2 = 1 \bmod p$ means $p \mid (x^2 - 1)$, so

$$p \mid (x-1)(x+1).$$

By an earlier corollary, $p$ must divide at least one of these factors. If $p \mid (x-1)$ then $x \equiv 1$, whereas $p \mid (x+1)$ implies $x \equiv -1$. □

For example, modulo 7, we know that $a^6 \equiv 1$. A solution of $x^2 = a^6$ is $x = a^3$ and we observe that

$$1^3 \equiv 1, \quad 2^3 \equiv 1, \quad 3^3 \equiv -1, \quad 4^3 \equiv 1, \quad 5^3 \equiv -1, \quad 6^3 \equiv -1.$$

## 1.4 Binary expansions

To find the decimal expansion of an integer, we repeatedly divide by 10, and read the remainders from bottom to top. For example,

$$\begin{aligned} 327 &= 32 * 10 + \boxed{7} \\ 32 &= 3 * 10 + \boxed{2} \\ 3 &= 0 * 10 + \boxed{3} \end{aligned}$$

The same process works in base 2 (binary)

$$\begin{aligned} 39 &= 19 * 2 + \boxed{1} \\ 19 &= 9 * 2 + \boxed{1} \\ 9 &= 4 * 2 + \boxed{1} \\ 4 &= 2 * 2 + \boxed{0} \\ 2 &= 1 * 2 + \boxed{0} \\ 1 &= 0 * 2 + \boxed{1}. \end{aligned}$$

Therefore

$$39 = 100111_2,$$

which is correct since $39 = 2^5 + 7 = 100000_2 + 111_2$. On a computer, 6 bits are needed to represent 39.

Recall the concept of *logarithm to base* $b$. It is the inverse to exponentiation:

$$\text{if } y = b^x \quad \text{then} \quad x = \log_b y.$$

We write

$$\ln y = \log_e y, \qquad \lg y = \log_2 y,$$

where

$$e = \lim_{n\to\infty} \left(1 + \frac{1}{n}\right)^n = \sum_{n=0}^{\infty} \frac{1}{n!} = 2.7182818\ldots$$

It is easy to show that

$$\lg y = \log_2 y = \frac{\ln y}{\ln 2} \approx 1.44 \ln y.$$

In this course, we shall only use logarithms to base 2. Here are the key properties:

- $\lg(2^x) = x$
- $2^{\lg y} = y$
- lg is strictly increasing: $a < b \Rightarrow \lg a < \lg b$.

Suppose that $n$ is trapped between two powers of 2:

$$\begin{aligned} 2^k \leqslant n < 2^{k+1}, &\qquad \text{so} \\ k \leqslant \lg n < k + 1. & \end{aligned}$$

It follows that the "floor" of $\lg n$ equals $k$:

$$\lfloor \lg n \rfloor = k.$$

Here "floor" means *the largest integer less than or equal to*. Observe that

$$2^{k+1} - 1 = \underbrace{11\cdots1}_{k+1}$$

is the largest binary number that can be represented with $k+1$ bits: we need $k + 1 = \lfloor \lg n \rfloor + 1$ bits to represent $n$.

**Example 1.19.** How many bits are needed to represent $n = 8293417$? We must trap $n$ between two powers of 2. For this purpose it is useful to know that

$$10^3 \simeq 2^{10}.$$

We can easily calculate

$$\begin{aligned} 2^{20} &= (2^{10})^2 \\ &= (1024)^2 \\ &= 1048576. \end{aligned}$$

It follows easily that

$$2^{22} < n < 2^{23},$$

and 23 bits are needed. In fact,

$$n = 11111101000110000101001_2.$$

**Example 1.20.** On the piano, 7 octaves are equivalent to 12 perfect fifths. We can write

$$2^7 \approx (3/2)^{12}, \quad \text{so} \quad 2^{19} \approx 3^{12},$$

where $\approx$ means 'approximately equal'. Which side is greater? Musically, the approximation can be resolved by making every semitone correspond to an interval of $2^{1/12}$, so that a "perfect" fifth corresponds to the ratio $2^{7/12} \approx 1.498\ldots$ This is the *equal temperament* system of tuning keyboard instruments, a concept dating back to 1584 or earlier.

# 2 Recurrence relations

## 2.1 Recursive functions

In this section, we shall be dealing with functions $f\colon \mathbb{N} \to \mathbb{N}$. We are used to having such functions defined explicitly, such as

$$f(n) = 3^n + (-2)^n - \tfrac{1}{6}n - \tfrac{13}{36}.$$

But one can also define functions in terms of earlier values, using a prescription like

$$f(n) = \begin{cases} 0 & \text{if} \quad n = 0 \\ 3f(n-1) + 1 & \text{if} \quad n \geqslant 1. \end{cases}$$

This gives the table

| $n$ | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| $f(n)$ | 0 | 1 | 4 | 13 | 40 |
| $2f(n)$ | 0 | 2 | 8 | 26 | 80 |
| $2f(n)+1$ | 1 | 3 | 9 | 27 | 81 |

from which we might guess that

$$f(n) = \tfrac{1}{2}(3^n - 1).$$

This can be proved by induction. But such explicit formulae are often not possible.

**Example 2.1.** Define $g\colon \mathbb{N} \to \mathbb{N}$ by

$$g(n) = \begin{cases} n & \text{if } n = 0, 1 \\ g(n/2) + 1 & \text{if } n \geqslant 2 \text{ is even} \\ g(3n+1) + 1 & \text{if } n \geqslant 3 \text{ is odd} \end{cases}$$

This function is related to the so-called *Collatz conjecture* or $3n+1$ *problem*. In fact, $g(n) - 1$ is the 'stopping time' for the Collatz map

$$f(n) = \begin{cases} n/2 & \text{if } n \geqslant 2 \text{ is even} \\ 3n+1 & \text{if } n \geqslant 3 \text{ is odd;} \end{cases}$$

it equals the number of applications of $f$ needed to reach 1. Note that one can only reach 1 through the sequence $16, 8, 4, 2, 1$ though one can reach 16 from both 32 and 5. It is unknown if the stopping time is finite for every integer $n \geqslant 2$, but it has been checked for all integers up to at least $2^{60}$. Of course, $g(2^k) = k + 1$ since $f$ keeps halving.

Let us compute $g(7)$; this is done by recording a somewhat tortuous succession of equations so as to arrive at $g(1)$, and then backtracking with the values:

$$\begin{array}{rrcl@{\qquad}rcll} & g(1) & = & 1 & & & & \\ & g(2) & = & g(1)+1 & \Rightarrow g(2) & = & 2 & \\ & g(4) & = & g(2)+1 & g(4) & = & 3 & \\ & g(8) & = & g(4)+1 & g(8) & = & 4 & \\ & g(16) & = & g(8)+1 & g(16) & = & 5 & \\ & g(5) & = & g(16)+1 & g(5) & = & 6 & \\ & g(10) & = & g(5)+1 & g(10) & = & 7 & \\ & g(20) & = & g(10)+1 & g(20) & = & 8 & \\ & g(40) & = & g(20)+1 & g(40) & = & 9 & \\ & g(13) & = & g(40)+1 & g(13) & = & 10 & \\ & g(26) & = & g(13)+1 & g(26) & = & 11 & \\ & g(52) & = & g(26)+1 & g(52) & = & 12 & \\ & g(17) & = & g(52)+1 & g(17) & = & 13 & \\ & g(34) & = & g(17)+1 & g(34) & = & 14 & \\ & g(11) & = & g(34)+1 & g(11) & = & 15 & \\ \uparrow & g(22) & = & g(11)+1 & g(22) & = & 16 & \downarrow \\ \text{start} & g(7) & = & g(22)+1 & g(7) & = & 17 & \text{end} \end{array}$$

We do not know the answer until we have got all the way to the top (and $g(1)$) and then back down again on the right, to find that $g(7) = 17$.

This set-up is called a *stack*, since the left-hand column resembles a stack of trays in a cafeteria (which is why we started at the bottom): $g(5)$ went in first, and $g(1)$ last. Then we could retrieve $g(1)$ first and $g(5)$ last. This illustrates the principle 'Last In First Out' or LIFO. Later on, we shall meet a different set-up, called a *queue*, in which the first item in is the first to be processed. So 'First In First Out' or FIFO (like most underground lifts).

Here is a table of values of $g(n)$ for $n = 0, 1, 2, \ldots, 104$:

$0, 1, 2, 8, 3, 6, 9, \underline{17}, 4, 20, 7, 15, 10, 10, 18, 18, 5, 13, 21, 21, 8, 8, 16, 16, 11, 24, 11, 112,$
$19, 19, 19, 107, 6, 27, 14, 14, 22, 22, 22, 35, 9, 110, 9, 30, 17, 17, 17, 105, 12, 25, 25, 25, 12, 12,$
$113, 113, 20, 33, 20, 33, 20, 20, 108, 108, 7, 28, 28, 28, 15, 15, 15, 103, 23, 116, 23, 15, 23, 23,$
$36, 36, 10, 23, 111, 111, 10, 10, 31, 31, 18, 31, 18, 93, 18, 18, 106, 106, 13, 119, 26, 26, 26, 26, 26, 88$

## 2.2 Fibonacci numbers

Leonardo di Pisa (c. 1175–1250) found his famous sequence of numbers in connection with the breeding of rabbits. One starts with a newly-born pair of rabbits, one male one female. The idealized assumption is that at one month they mature and become fertile, and at two months (and at each month thereafter) the female gives birth to another male-female pair. Let $F_n = F(n)$ denote the total number of rabbit pairs in the middle of the $n$th month, so $F_1 = F_2 = 1$. Then

$$\begin{array}{rcccc} F_n & = & \#\{\text{immature pairs}\} & + & \#\{\text{mature pairs}\} \\ & = & F_{n-2} & + & F_{n-1} \end{array}$$

for $n \geqslant 3$. This is because all pairs from a month ago will be mature, and the newly-born immature rabbits are offspring of parents from two months ago.

To extend this relation to $n = 2$, we can set $F_0 = 0$. We then have the *recurrence relation*

$$F_n = F_{n-1} + F_{n-2}, \qquad F_0 = 0, \quad F_1 = 1,$$

which can be solved recursively. The aim of this section is to show that there is a simple formula for the Fibonacci number $F_n$. For this purpose, define

$$\varphi = \tfrac{1}{2}(1 + \sqrt{5}) = 1.6180\ldots, \qquad \psi = \tfrac{1}{2}(1 - \sqrt{5}) = -0.6180\ldots$$

In §2.3, we shall show that

**Proposition 2.2.** $F_n = \dfrac{1}{\sqrt{5}}\,(\varphi^n - \psi^n)$.

Note that $\varphi$ and $\psi$ are the roots of $x^2 - x - 1 = 0$ or

$$\frac{x}{1} = \frac{1}{x-1},$$

and that $\varphi$ (the positive root) is the so-called *golden ratio*. We leave proofs of the following statements as exercises.

**Corollary 2.3.** *The ratio $F_{n+1}/F_n$ tends to $\varphi$ as $n \to \infty$.*

**Corollary 2.4.** *$F_n$ is the closest integer to $\varphi^n/\sqrt{5}$ for all $n$.*

One can compute the so-called *generating function* for the sequence $(F_n)$ directly from the recurrence relation. Indeed,

**Corollary 2.5.** *Suppose that* $|x| < 1/\varphi$. *Then*

$$\sum_{n=1}^{\infty} F_n x^n = \frac{x}{1-x-x^2}. \tag{1}$$

We can check this by setting $\lambda = x + x^2$ and using the binomial expansion

$$\begin{aligned}(1-\lambda)^{-1} &= 1 + \lambda + \lambda^2 + \lambda^3 + \cdots \\ &= 1 + x + x^2 + (x^2 + 2x^3 + x^4) + (x^3 + 3x^4 + 3x^5 + x^6) + (x^4 + \cdots) + \cdots \\ &= 1 + x + 2x^2 + 3x^3 + 5x^4 + \cdots\end{aligned}$$

In particular, $\displaystyle\sum_{n=1}^{\infty} \frac{F_n}{10^n} = \frac{10}{89} = 0.11235955.$

We shall not be concerned with questions of convergence in this course, but Corollary 1 implies that the radius of convergence of the series (1) equals $1/\varphi$.

**A curiosity.** Since 1 mile equals 1.609... kilometers, Fibonacci's numbers (if you can remember them) give a sufficiently accurate way of converting. For example, 144 km/h is almost exactly 89 mph.

**Example 2.6.** The sum $s_n$ of the first $n$ odd numbers satisfies an obvious recurrence relation:

$$\begin{cases} s_n = s_{n-1} + 2n - 1, \\ s_1 = 1 \end{cases}$$

We already know that the solution is $s_n = n^2$, but the aim will be to solve such relations systematically without knowing the answer by other means.

**Definition 2.7.** *A recurrence relation of* order $k$ *will specify*

$$a_n = f(n, a_{n-1}, a_{n-2}, \ldots, a_{n-k})$$

*as a function of the* $k$ *preceding values and possibly* $n$ *itself. One also needs to prescribe* $k$ *initial values of the function* $n \mapsto a_n$.

The relation is called *linear* if the right-hand side equals

$$c_0(n) + c_1(n)a_{n-1} + \cdots + c_k(n)a_{n-k},$$

for some functions $c_i(n)$ of $n$, as in the previous example. Such a linear relation is called *homogeneous* if $c_0(n)$ is absent, and it has *constant coefficients* if $c_1, \ldots, c_k$ are independent of $n$ (so constants). The usual relation described the Fibonacci numbers is therefore or order 2, linear, homogeneous with constant coefficients. By contrast,

$$a_n = a_{n-1} * a_{n-2}$$

also has order 2, but is is not linear (so the other qualifications are irrelevant).

## 2.3 Constant coefficients

Consider a recurrence relation with constant coefficients:

$$a_n = c_1 a_{n-1} + \cdots + c_k a_{n-k} + c_0(n), \tag{NH}$$

with $c_0(n)$ a non-zero function. The associated homogeneous relation is

$$a_n = c_1 a_{n-1} + \cdots + c_k a_{n-k}, \tag{H}$$

without the term at the end. We are likely to consider only $k \leqslant 3$.

**Proposition 2.8.** *(i) If $(a_n)$ and $(b_n)$ are sequences solving* (H) *(with $b_n$ in place of $a_n$) then $(Aa_n + Bb_n)$ will also solve* (H) *for any $A, B \in \mathbb{R}$.*

*(ii) here are $k$ linearly independent solutions to* (H)

*(iii) If $(a_n)$ and $(b_b)$ solve* (NH) *then $(a_n - b_n)$ solves* (H).

The proposition is also valid for linear relations, and there is an analogy with ordinary differential equations. We omit the proofs here.

For $k = 2$, 'linearly independent' simply means that one solution is not an overall multiple of the other: sequences with $a_n = n^2$ and $b_n = n^2 + 1$ are independent, but $a_n = n^2$ and $b_n = -7n^2$ are not.

(iii) implies that the general solution of (NH) is *any* particular solution to it plus the general solution of (H).

It is known that solutions of (H) are mostly linear combinations of $\lambda^n$, where $\lambda \in \mathbb{R}$ is constant. The next example will verify this.

**Example 2.9.** Solve

$$\begin{cases} a_n = -a_{n-1} + 6a_{n-2}, & n \geqslant 2 \\ a_0 = 2,\ a_1 = -1. \end{cases}$$

Try $a_n = \lambda^n$. Substituting into the recurrence relation,

$$\lambda^n = 6\lambda^{n-2} - \lambda^{n-1},$$

and (since we can assume $\lambda \neq 0$),

$$\lambda^2 + \lambda - 6 = 0 \implies (\lambda - 2)(\lambda + 3) = 0.$$

Taking $\lambda = 2$ and $\lambda = 3$ gives two independent solutions, and (from (i) and (ii) above) the genral solution is

$$a_n = A * 2^n + B * (-3)^n.$$

The constants $A, B$ are determined by the initial conditions, which give

$$2 = A * 1 + B * 1, \quad -1 = A * 2 + B * (-3) \implies A = B = 1.$$

The final answer is therefore $a_n = 2^n + (-3)^n$.

**Example 2.10.** For the Fibonacci sequence, the equation is $\lambda^2 = \lambda + 1$ or $\lambda^2 - \lambda - 1 = 0$, which has roots

$$\varphi = \tfrac{1}{2}(1 + \sqrt{5}), \qquad \psi = \tfrac{1}{2}(1 - \sqrt{5}),$$

giving a general solution $A\varphi^n + B\psi^n$. Then $A, B$ are found by solving $0 = F_0$ (which implies $B = -A$) and

$$1 = F_1 = A\varphi + B\psi = A(\varphi - \psi) = A\sqrt{5}.$$

This proves the previous proposition.

To summarize, here is the strategy for solving (H):

| |
|---|
| Substitute $a_n = \lambda^n$ |
| Obtain a polynomial equation of degree $k$ in $\lambda$ |
| Find its roots $\lambda_1, \ldots, \lambda_k$ |
| The general solution is $a_n = A_1\lambda_1^n + \cdots + A_k\lambda_k^n$ |
| Find the constants by solving the initial conditions |

There are two possible snags. The roots may be complex, though if the original equation is real, they will always come in complex conjugates, and the choice of constants $A_i$ will ensure that all solutions are real. Or, there may be repeated roots, in which case (by (ii)) there must exist additional solutions.

**Example 2.11.** Express the solution of $a_n = -a_{n-2}$ with $a_0 = 0$ and $a_1 = 1$ in closed form. Of course,

$$(a_n) = (0, 1, 0, -1, 0, -1, 0, \ldots),$$

but we are asked for a formula. We have $\lambda^2 + 1 = 0$ so the roots are $\pm i$ where $i = \sqrt{-1}$. So the solution is $Ai^n + B(-i)^n$, with $A + B = 0$ and $i(A - B) = 1$. Then $A = -B = -\frac{1}{2}i$, and the closed formula is

$$a_n = -\tfrac{1}{2}i(i^n - (-i)^n) = -\tfrac{1}{2}(i^{n+1} + (-i)^{n+1}).$$

In the case of repeated roots, let us consider what happens when the roots are $\lambda$ and $\lambda + \delta$ for $\delta > 0$. We know from (i) that

$$\frac{(\lambda + \delta)^n - \lambda^n}{\delta}$$

must be a solution. If we let $\delta \to 0$ then in the limit this becomes the derivative of $\lambda^n$, namely $n\lambda^{n-1}$. So we expect this (or equivalently $n\lambda^n$) to be a second solution. In fact, a repeated root of multiplicity $m$ will allow us to introduce solutions

$$\lambda^n, \quad n\lambda^n, \quad \ldots \quad , n^{m-1}\lambda^n.$$

**Example 2.12.** Find the general solution of $a_n = 2a_{n-1} - a_{n-2}$. Here, $\lambda^2 - 2\lambda + 1 = 0$ or $(\lambda - 1)^2 = 0$, so we get

$$a_n = A * 1^n + B * n * 1^n = A + Bn.$$

## 2.4 Particular solutions

To solve a non-homogeneous linear equation (NH), proceed as follows. The order is important:

| |
|---|
| Find the general solution of (H) with constants |
| Determine the form of a particular solution of (NH) |
| Substitute to determine any constants in the particular solution |
| Add the two solutions |
| Apply the initial values to determine the constants relating to (H) |

We shall mostly see assigned functions of the form

$$c_0(n) = p(n) * \mu^n,$$

where $p(n)$ is a polynomial such as 7 or $n^2$ or $n^3 - n + 7$. Given such a function, one guesses a solution $q(n) * \mu^n$, where $q(n)$ is now an *arbitrary* polynomial of the same degree as $p(n)$. Here are some examples:

| $c_0(n)$ | guess |
|---|---|
| $7$ | $\alpha$ |
| $n$ | $\alpha n + \beta$ |
| $2^n$ | $\alpha 2^n$ |
| $n^2\,3^n$ | $(\alpha n^2 + \beta n + \gamma)3^n$ |

One needs to substitute into (NH) to find the constants $\alpha, \beta, \gamma$. This will work provided no term in the guess is a solution of (H). In the letter case, one needs to multiply by one or more factors of $n$. A simple instance follows, though future exercises will clarify this.

**Example 2.13.** Find the general solution of

$$a_n = -a_{n-1} + 6a_{n-2} + 2^n.$$

Had 2 not been a root, we would have tried $\alpha\, 2^n$, but (since $2^n$ solves (H)) this would have given $0 = 2^n$. So we try $a_n = \alpha\, n\, 2^n$. This gives

$$\alpha n 2^n = -\alpha(n-1)2^{n-1} + 6\alpha(n-2)2^{n-2} + 2^n.$$

Dividing by $a^{n-2}$,

$$4n\alpha = -2\alpha(n-1) + 6\alpha(n-2) + 4;$$

the terms involving $n$ cancel out (as they must), and we are left with

$$0 = 2\alpha - 12\alpha + 4.$$

Thus, $\alpha = 2/5$ and we finish up with

$$a_n = A\, 2^n + B(-3)^n + \tfrac{2}{5} n 2^n.$$

**Example 2.14.** Solve

$$a_n = -a_{n-1} + 6a_{n-2} + n, \qquad a_0 = 2, \quad a_1 = -1.$$

The homogenous equations has general solution $A * 2^n + B * (-3)^n$. For a particular solution, we substitute $a_n = \alpha n + \beta$. Since the resulting equation must hold for *all* $n$, we can separate out the terms involving $n$ and those that do not. This gives two separate equations, which imply that $\alpha = -1/4$ and $\beta = -11/16$. Then we substitute

$$a_n = A * 2^n + B * (-3)^n - \tfrac{1}{4}n - \tfrac{11}{16}$$

to find that

$$2 = A + B - \tfrac{11}{16}, \qquad -1 = 2A - 3B - \tfrac{11}{16}$$

giving $A = 8/5$ and $B = 87/80$.

## 2.5 Derangements

In this section, we will work with some linear recurrence relations that do not have constant coefficients.

Fix a positive integer $n$. Recall that a *permutation* of the set $\Omega = \{1, 2, \ldots, n\}$ is a bijective mapping $f : \Omega \to \Omega$. There are $n!$ such permutations.

**Definition 2.15.** *A* derangement *of $\Omega$ is a permutation $f : \Omega \to \Omega$ such that no element $i \in \Omega$ has the property that $f(i) = i$. This means that no number 'stays put', or (in the language of analysis) $f$ has no fixed point.*

Let $d_n$ denote the number of derangements of $\{1, 2, \cdots, n\}$. It is obvious that $d_1 = 0$ (because there is only the identity permutation), $d_2 = 1$ (only swapping $1, 2$ works) and $d_3 = 2$ (because only the two 3-cycles do not have a fixed point).

Recall that a *cycle* of order $k$ is a permutation of the form

$$f: \quad i_1 \mapsto i_2 \mapsto \cdots \mapsto i_k \mapsto i_1;$$

here $i_1, \ldots, i_k$ are distinct positive integers. This permutation is denoted by the symbol $(i_1\, i_2\, \ldots\, i_k)$, which we regard as a function applied (on the left) to numbers. If $i_j \leqslant n$ for all $j$ then $f$ is an element of order $n$ inside the group of a permutations of $\{1, 2, \ldots, n\}$. Moreover,

$$f = \sigma \circ (1\, 2\, \ldots\, k) \circ \sigma^{-1},$$

where $\sigma$ is any permutation that maps $j$ to $i_j$ for all $j$. This establishes the well-known fact that *any two $k$-cycles are conjugate* in a group of permutations.

To compute $d_4$, we appeal to the description of permutations from group theory. *Any permutation can be expressed (uniquely, up to order of the factors) as a product of disjoiont cycles.*

| type of permutation | example | number of them |
|---|---|---|
| identity | | 1 |
| 4 cycle* | (1234) | 6* |
| 3 cycle | (123) | 8 |
| 2 cycle | (12) | 6 |
| pair of 2 cycles* | (12)(34) | 3* |
| | | 4! = 24 |

Only the asterisked permutations are derangements, so $d_4 = 9$. One can show that $d_5 = 44$ with a similar table, but then it becomes complicated because of the large variety of cycle decompositions.

Fortunately, we can easily find a way of computing $d_n$ using recurrence relations. It is convenient to define $d_0 = 1$.

**Theorem 2.16.** *(i) $d_n$ satisfies the second-order linear homogeneous recurrence relation*

$$d_n = (n-1)(d_{n-1} + d_{n-2}), \quad n \geqslant 2.$$

*(ii) $d_n$ satisfies the first-order linear non-homogeneous recurrence relation*

$$d_n = nd_{n-1} + (-1)^n.$$

*(iii) The ratio $d_n/n!$ tends to $1/e$ as $n \to \infty$.*

*Proof.* (i) The definition of $d_0$ makes the formula work for $n = 2$. Now let $f$ be a derangement of $\{1, 2, \ldots, n\}$ for $n \geqslant 3$. Suppose that $f(1) = k$, so $k \geqslant 2$. There are two subcases:
(a) $f(k) = 1$, so the cycle decomposition contains a 2-cycle $(1\,k)$. Forgetting $1, k$, what is left of $f$ is a derangement of $n-2$ objects, and there are $d_{n-2}$ of these. Together with the choice of $i$, this gives $(n-1)d_{n-2}$ possibilities.
(b) $f(k) = j \neq 1$. This time, remove $k$ and define a derangement $\tilde{f}$ of $n-1$ objects by setting

$$\tilde{f}(i) = \begin{cases} j & \text{if} \quad i = 1 \\ f(i) & \text{if} \quad i \neq 1,\ i \neq k. \end{cases}$$

There are $(n-1)d_{n-1}$ possibilities in this subcase.

(ii) Set $a_n = d_n - nd_{n-1}$. We want to show that $a_n = (-1)^n$. Well,

$$\begin{aligned} a_n + a_{n-1} &= (d_n - nd_{n-1}) + (d_{n-1} - (n-1)d_{n-2}) \\ &= d_n - (n-1)(d_{n-1} + d_{n-2}) \\ &= 0, \end{aligned}$$

the last equality by (1). Since $a_1 = -1$, it follows that $a_n = (-1)^n$.

(iii) Rewrite (2) as

$$\begin{aligned} \frac{d_n}{n!} &= \frac{d_{n-1}}{(n-1)!} + \frac{(-1)^n}{n!} \\ &= \frac{d_{n-2}}{(n-2)!} + \frac{(-1)^{n-1}}{(n-1)!} + \frac{(-1)^n}{n!} \\ &\quad \cdots\cdots \\ &= \frac{d_1}{1!} + \sum_{i=2}^{n} \frac{(-1)^i}{i!}. \end{aligned}$$

with a total of $n-1$ lines. Since $d_1 = 0$, it follows (strictly speaking, by induction) that

$$\frac{d_n}{n!} = \sum_{i=0}^{n} \frac{(-1)^i}{i!},$$

since the first two terms in the summation cancel (by convemtion, $0! = 1$). As $n \to \infty$, the summation tends to $e^{-1}$. □

Here is a table of values:

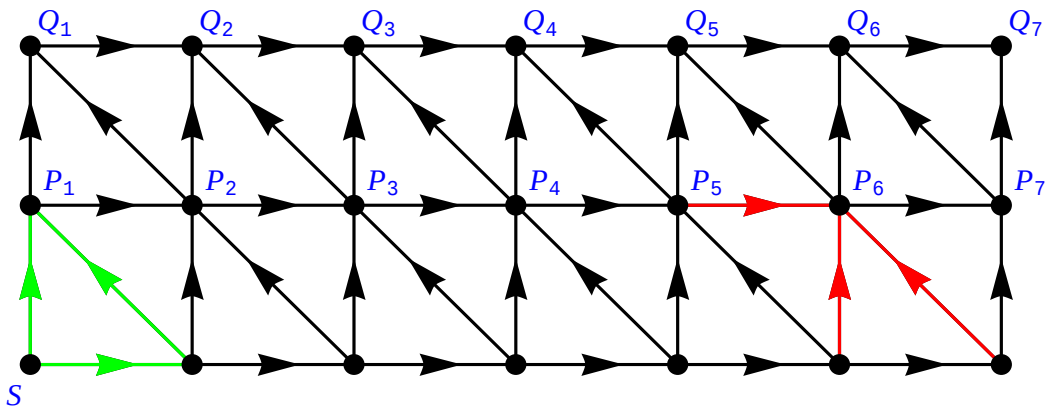
| $n$ | 1 | 2 | 3 | 4 | 5 | 6 | 7 | |
|---|---|---|---|---|---|---|---|---|
| $d_n$ | 0 | 1 | 2 | 9 | 44 | 265 | 1854 | $\cdots$ |
| $n!$ | 1 | 2 | 6 | 24 | 120 | 720 | 5040 | $\cdots$ |
| $d_n/n!$ | 0 | 0.5 | 0.333 | 0.375 | 0.367 | 0.368 | 0.368 | $\cdots$ |

Here the ratio is given to three significant figures, so the probability of 'no snap' playing with two decks of 6 shuffled cards is already a good approximation to $1/e$.

## 2.6 Other counting applications

This section highlights two situations in which recurrence equations also occur naturally.

**Example 2.17.** Consider the system of one-way roads illustrated:




Let $a_n$ denote the number of different routes from the starting point $S$ to $P_n$. (If $n \geqslant 7$ the diagram needs extending to the right in the obvious fashion.)

To illustrate the method, we first take $n = 6$. One can reach $P_n$ in one step from three directions, shown in red. Namely, travelling north or north-west from the bottom row, or travelling east from $P_5$. There is only one route to any point on the bottom row, so $a_6 = a_5 + 1 + 1$, since there are $a_5$ routes to $P_5$ which can be followed by the one step eastwards. The same argument shows us that

$$a_n = a_{n-1} + 2.$$

The green steps show that there are 2 routes to $P_1$, so $a_1 = 2$. The solution of this recurrence relation is obviously $a_n = 2n$.

A similar argument can now be used to count the number $b_n$ of routes from $S$ to $Q_n$ in the top row. Again, one should consider the *immediate* predecessors of $Q_n$, which are $P_n, P_{n+1}, Q_{n-1}$, provided $n \geqslant 2$. These furnish $a_n, a_{n+1}, b_{n-1}$ routes, so

$$b_n = a_n + a_{n+1} + b_{n-1} = b_{n-1} + 4n + 2, \qquad n \geqslant 2.$$

One can arrive at $Q_1$ from either $P_1$ or $P_2$, so $b_1 = a_1 + a_2 = 6$. The homogeneous relation (H) has general solution $b_n = C$ = constant, so for a particular solution of (NH) we try

$$b_n = An^2 + Bn,$$

giving

$$An^2 + Bn = A(n-1)^2 + B(n-1) + 4n + 2 \quad \Rightarrow \quad 0 = -2An + A - B + 4n + 2 = 0$$
$$\Rightarrow \quad A = 2, \ B = 4.$$

We also have $6 = b_1 = A + B + C$, so $C = 0$. Therefore

$$b_n = 2n^2 + 4n.$$

**Example 2.18.** A gardener has to plant a row of $n \geqslant 2$ rose bushes, which come in three varieties (red, artificially blue, yellow), observing the following rules:

(i) the first bush must be red;

(ii) the last ($n$th) bush must be red;

(iii) no two colours can be adjacent.

We seek the number $r_n$ of different ways of planting the bushes.

The lowest possible value of $n$ is 3 to avoid the two reds together. For $n = 3$ we just need to choose the middle colour, so $r_3 = 2$. More generally, once we know the colour of the $k$th bush then there are two choices of colour for bush $k+1$. So without condition 2., there are

$$1 * \underbrace{2 * 2 * \cdots * 2}_{n-1} = 2^{n-1}$$

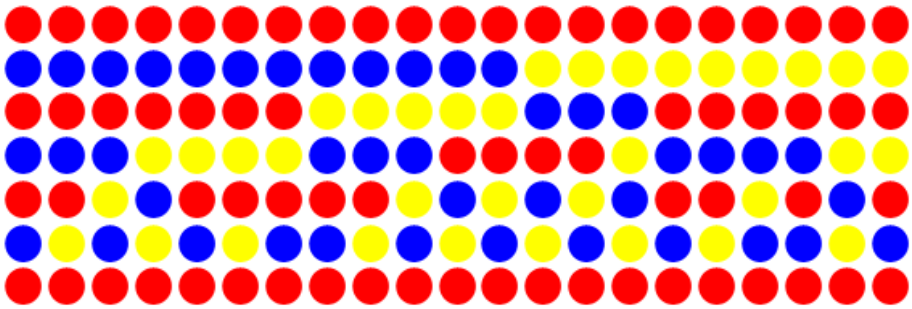
choices. With condition 2., we must insist that bush $n-1$ is not red, after which there is no more choice. Therefore

$$b_n = 2^{n-2} - b_{n-1}.$$

The solution is

$$r_n = \tfrac{1}{3} * 2^{n-1} - \tfrac{2}{3}(-1)^n, \qquad n \geqslant 3.$$

When $n = 7$ (as in the picture), there are 22 ways of planting, illustrated below with each row now vertical.



# 3 Arithmetical algorithms

## 3.1 First concepts

**Definition 3.1.** *An* algorithm *is a finite set of unambiguous instructions that when executed terminate in a finite number of steps.*

Named after Muhammad ibn Musa al-Khwarizmi (c. 780–850). A more formal specification (beyond the scope of this course) takes one into the area of recursive function theory, Turing machines and mathematical logic.

**Example 3.2.** Consider the factorial function $\mathbb{N} \to \mathbb{N}$. There are two distinct processes that can be used to compute $n!$

Firstly, by *iteration*. This is exemplified by the following SAGE code:

```
def fac(n):
    x = 1
    for i in range(2,n+1):
        x = x*i
    return x
```

In this code, the command range$(p, q)$ lists integers from $p$ to $q-1$ (rather than $q$). The penultimate line has the effect of replacing $x$ by $x$ times $i$. Note that the program correctly outputs $\mathrm{fac}(0) = \mathrm{fac}(1)$. By regarding $i$ more as a variable than a counter, we can replace the 'for/do loop' by a conditional:

```
def fac(n):
    x = 1
    i = 1
    while i < n+1:
        x = x*i
        i = i+1
    return x
```

The time needed to carry out the computation is estimated by counting the number $n-1$ of multiplications (the most 'expensive' operation). If we suppose that each multiplication takes one unit of time, then the total time $T = n-1$ satisfies

$$T = \Theta(n).$$

This equation is shorthand for saying that, for sufficiently large $n$, there exist constants $0 < c_1 < c_2$ such that

$$c_1 n \leqslant T \leqslant c_2 n.$$

Equivalently, $T = O(n)$ *and* $n = O(T)$, so both $T/n$ and $n/T$ are bounded as $n \to \infty$. Observe that it does not matter whether a unit of time is one millisecond or one minute.

Alternatively, one can use the recurrence relation $a_n = na_{n-1}$; this gives the definition of factorials by *recursion*:

```
def fac(n):
    if n == 0:
        return 1
    else:
        return n*fac(n-1)
```

This approach starts by asserting that $0! = 1$, which is true for convention, or rather convenience, as it seems to work in many formulae. Let $t_n$ denote the number of times multiplication is used to compute $\mathrm{fac}(n)$ using the last program. Then

$$t_n = t_{n-1} + 1,$$

so $t_n = n$. Once again, the total time equals $\Theta(n)$, but we also require memory that grows linearly with $n$ (unlike in the first case). Indeed, the equations stored expand and contract, like the stacking of trays. At some point of the process, we will have

$$\mathrm{fac}(5) = 5 * (4 * (3 * (2 * (1 * \mathrm{fac}(0))))),$$

and we are stuck if the process is interrupted.

## 3.2 Powers by squaring

**Example 3.3.** Consider computing $x^n$, where $n$ is a positive integer and $x$ is a number to a given precision. This could be carried out by iteration, mimicking what we first did for factorials:

```
def pow(x,n):
    y = 1
    for i in range(1,n+1):
        y = y*x
    return y
```

This method requires $n - 1$ multiplications, but we can find a much more efficient way by squaring at intermediate stages. For example, the calculation

$$\begin{aligned} x^{19} &= (x^9)^2\, x \\ &= ((x^4)^2\, x)^2\, x \\ &= (((x^2)^2)^2\, x)^2\, x \end{aligned}$$

uses only 6 multiplications. Here, we have effectively written the exponent $19 = 10011_2$ in binary, adding $x$ added on the right if and only if the remainder is 1. To convert the binary expansion of the exponent $n$ into a systematic procedure, read it from the left with initial value 1 in the 'register'. Then

- square for the privilege of processing the digit,
- multiply by $x$ for each digit '1' encountered.

Starting with 19, the first '1' on the left allows us to write $1^2 * x = x$. This is then squared three times, though in processing the next '1', we multiply by $x$. The final '1' causes us to square and multiply by $x$ again, and we are finished. It would be more efficient to ignore the first squaring (which is always $1 * 1$) and to process $x$ starting from the second binary digit, but the instructions are slightly neater to state using the full binary expansion. Here is the pseudocode:

```
pow(x,n):
    y = 1
    find the expansion n.binary()
    for each bit from left to right:
        y = y*y
        if bit == 1:
            y = y*x
    return y
```

To implement this properly in SAGE one would need to define 'bit' as a successive element in the string consisting of the binary digits of $n$, but this would obscure the instructions. As it stands, the process should be sufficiently clear by carrying it out by hand. The only values stored are $x$, $n$ and each current value of $y$. The only operations are squaring and multiplying by $x$.

**Example 3.4.** If $x = 3$ and $n = 11 = 1011_2$, we display each loop vertically.

| $n$ | 1 | 0 | 1 | 1 |
|---|---|---|---|---|
| $y$ in | 1 | 3 | 9 | 243 |
| $y$ squared | 1 | 9 | 81 | 59049 |
| $y$ out | 3 | 9 | 243 | 177147 |

Thus $3^{11} = 177147$ was computed with three squarings (ignoring $1*1$) and three multiplications. Here is the same example modulo 16:

| $y$ in | 1 | 3 | 9 | 3 |
|---|---|---|---|---|
| $y$ squared | 1 | 9 | 1 | 9 |
| $y$ out | 3 | 9 | 3 | 11 |

Therefore $3^{11} = 11 \bmod 16$.

Let's analyse the efficiency. Recall from §1 that the number of bits needed to encode $n$ in binary is $\lfloor \lg n \rfloor + 1$. For each bit, we need to square (a multiplication). Ignoring the first $1 * 1$ gives $\lfloor \lg n \rfloor + 1 - 1 = \lg n$ operations. Each '1' after the first gives an additional multiplication, so there are at most $\lfloor \lg n \rfloor$ of these. So we have a total of $2\lfloor \lg n \rfloor$ operations, and the algorithm requires $O(\lg n)$ operations, which is much more efficient than the iteration program.

## 3.3 Euclid's algorithm

Let $a, b$ be integers with $b > 0$. The aim is to compute their greatest common divisor $\gcd(a, b)$. Suppose that

$$a = qb + r, \qquad 0 \leqslant r < b,$$

which implies that

$$a/b = q + r/b. \qquad 0 \leqslant r/b < 1.$$

In this situation, we know that $r = a \bmod b$, but to emphasize that $r < b$ we can use the exact formula

$$r = a - \lfloor a/b \rfloor b.$$

This remainder is denoted $a\%b$ in C++ or SAGE.

**Lemma 3.5.** *With this notation,* $\gcd(a, b) = \gcd(b, r)$.

*Proof.* Suppose that $s = \gcd(b, r)$. Then $s|b$ and $s|r$. Thus $s|a$, and $s$ is a common divisor of $a$ and $b$. Suppose that $t$ is *another* common divisor of $a$ and $b$. Then $t|r$, so $t$ is also a common divisor of $b$ and $r$, and (since $s$ is the *greatest* such) $t \leqslant s$. Therefore $s$ is indeed the *greatest* common divisor of $a$ and $b$. □

Eulcid's algorithm now consists of starting from $(a, b)$, and then repeatedly performing divison and applying the lemma. Since $r_{i+1} < r_i$, we must have $r_n = 0$ for some $n$:

$$\begin{aligned} a &= q_0 b + r_1 \\ b &= q_1 r_1 + r_2 \\ r_1 &= q_2 r_2 + r_3 \\ &\cdots \\ r_{n-2} &= q_{n-1} r_{n-1} + 0. \end{aligned}$$

Applying the lemma, we are led to $\gcd(r_{n-1}, 0) = r_{n-1}$. So this equals $\gcd(a, b)$. The code is therefore quite simple:

```
def euc(a,b):
    if b == 0:
        return a
    else:
        return euc(b,a%b)
```

If we keep track of $r_i$ as a linear combination of $r_{i-1}$ and $r_{i-2}$, simplified at each stage, this we will obtain integers $x, y$ for which

$$\gcd(a, b) = xa + yb.$$

This is called *Euclid's Extended Algorithm.* There is a quick way of carrying this out by hand using matrices. We shall illustrate the method next. We first set up a matrix of the form

$$\begin{pmatrix} a & 1 & 0 \\ b & 0 & 1 \end{pmatrix}$$

in which the second column keeps track of coefficients of $a$ and the third column those of $b$: the first row is to be interpreted as telling us that $a = 1 * a + 0 * b$ and the second that $b = 0 * a + 1 * b$. One then subtracts the row with the smallest first entry (initially the second if $b < a$) from the other row. One repeats this step until one row row begins with a '0', in which case the entry above or below it equals $\gcd(a, b)$, and the remaining entries of that row give $x$ and $y$.

**Example 3.6.** To compute $\gcd(33, 93)$, we have

$$\begin{pmatrix} 93 & 1 & 0 \\ 33 & 0 & 1 \end{pmatrix} \rightarrow \begin{pmatrix} 27 & 1 & -2 \\ 33 & 0 & 1 \end{pmatrix} \rightarrow \begin{pmatrix} 27 & 1 & -2 \\ 6 & -1 & 3 \end{pmatrix}$$
$$\downarrow$$
$$\begin{pmatrix} 3 & 5 & -14 \\ 0 & -11 & 31 \end{pmatrix} \leftarrow \begin{pmatrix} 3 & 5 & -14 \\ 6 & -1 & 3 \end{pmatrix}$$

There is no need to swap the rows each time provided one is happy to subtract the first from the second. The final '0' tells us that $\gcd(93, 33) = 3$ (of course, this was obvious) and

$$3 = \gcd(33, 93) = 5 * 93 - 14 * 33$$

(the coefficients $x = 5$ and $y = -14$ were less obvious).

We shall not prove formally that this method *does* produce the correct result, but one can understand it as follows. The new numbers $27, 6, 3$ in the first columns are simply the remainders $r_1, r_2, r_3$, with $r_4 = 0$. And since we are performing elementary row operations, each row $(r_i \ x_i \ y_i)$ tells us that $r_i = x_i * 93 + y_i * 33$. The penultimate one gives us the desired expression for $r_3 = \gcd(93, 33)$.

**Example 3.7.** We compute $\gcd(33, 93)$ on the left below, and using the steps on the right we express it as a linear combination of 33 and 93.

$$\begin{aligned}
33 &= 0 * 93 + 33 \\
93 &= 2 * 33 + 27 && 27 = 1 * 93 - 2 * 33 \\
33 &= 1 * 27 + 6 && 6 = 1 * 33 - 1 * 27 = 33 - (93 - 2 * 33) = -93 + 3 * 33 \\
27 &= 4 * 6 + 3 && 3 = 1 * 27 - 4 * 6 = (93 - 2 * 33) - 4 * (-93 + 3 * 33) = 5 * 93 - 14 * 33 \\
6 &= 2 * 3 + 0
\end{aligned}$$

The conclusion is

$$3 = \gcd(33, 93) = 5 * 93 - 14 * 33.$$

Note that one saves one line if initially $|a| > b$. On the right-hand side, once an equation (like $27 = \cdots$) has been used twice, it can be discarded.

We can avoid all the subscripts above, and express it more succinctly with the following pseudocode:

```
EEUCLID(a,b)
if b=0
    then return (a,1,0)
(d',x',y') ← EEUCLID(b, a mod b)
(d,x,y) ← (d',y',x'-⌊a/b⌋y')
return (d,x,y)
```

## 3.4 Consolidation

The aim of this section is to draw together many of the topics we have seen so far, namely Induction (§1,1), the Fibonacci numbers (§2.2), and logarithms (§1.4 and §3.2), in order to analyse the efficiency of Euclid's algorithm (§3.3). We shall show that, like our method of exponentiation by repeated squaring, it executes in 'log time'.

**Example 3.8.** Consider the function

$$f(x) = 11x + 5.$$

As it stands, it defines both a mapping $\mathbb{R} \to \mathbb{R}$ and a mapping $\mathbb{Z} \to \mathbb{Z}$. The first is a bijection, the second is not (because the multiplicative inverse $11^{-1}$ does not exist in $\mathbb{Z}$). We now want to work modulo 26, this being the number of charcaters in the English alphabet. Write $x_1 \equiv x_2$ to mean

$$x_1 = x_2 \bmod 26, \quad \text{i.e.} \quad 26 \mid (x_1 - x_2).$$

Then

$$x_1 \equiv x_2 \implies f(x_1) \equiv f(x_2),$$

and because of this it makes sense to regard $f$ as a mapping between congruence classes modulo 26. To do this properly, we define

$$\tilde{f}\colon R \to R, \qquad R = \{0, 1, 2, \ldots, 26\}$$

by

$$\tilde{f}(x) = f(x) \bmod 26 \in R, \quad \text{explicitly} \quad f(x) - 26\lfloor f(x)/26 \rfloor.$$

Then $\tilde{f}\colon \mathbb{R} \to R$ is a bijection, i.e. a *permutation* of $R$. This is because $11, 26$ are coprime,

$$1 = \gcd(26, 11) = 3 * 26 - 7 * 11,$$

and $11^{-1} \equiv -7$ exists modulo 26. Thus,

$$y \equiv 11x + 5 \implies x \equiv 11^{-1}(y - 5) \equiv -7y + 9.$$

One could use $\tilde{f}\colon x \mapsto y$ and its inverse $y \mapsto x$ as a simple way to encrypt/decrypt words in the alphabet $\{A, B, C, \ldots, Z\}$, but it could be broken by frequency analysis of letters (of the message is large enough). A more sophisticated method based on similar principles is the so-called *Vigenère cipher*.

**Definition 3.9.** *Suppose that $a > b > 0$. Let $s(a, b)$ denote the number of steps needed in executing Euclid's algorithm so that the remainder becomes $0$ in the final step.*

It is reasonable to suppose that $s(a, b)$ estimates the time required to compute $\gcd(a, b)$. If $s(a, b) = n$ then we are stating that (in our previous notation) $r_n = 0$ and $r_{n-1} \neq 0$.

**Example 3.10.** One has

$$s(93, 33) = 4$$
$$s(33, 93) = 5.$$

Computer software quickly reveals that

$$\gcd(100! + 1,\ 100^{100} - 1) = 101$$
$$s(100! + 1,\ 100^{100} - 1) = 337.$$

Fibonacci numbers are relevant to the implementation of Euclid's algorithm. The following scheme implements the algorithm to determine the greatest common divisor of two adjacent Fibonacci numbers:

$$\begin{aligned} F_{n+2} &= F_{n+1} + F_n \\ F_{n+1} &= F_n + F_{n-1} \\ \vdots &= \quad \vdots \\ F_5 &= F_4 + F_3 \\ F_4 &= F_3 + F_2 \\ F_3 &= 2\,F_2 + 0 \end{aligned}$$

This is because each Fibonacci number like $F_{n+1}$ divides exactly once into the next higher one $F_{n+2}$ with remainder $F_n$ (because twice $F_{n+1}$ into $F_{n+2}$ won't go!). The conclusion is that

$$\gcd(F_{n+2}, F_{n+1}) = F_2 = 1,$$

so any two adjacent Fibonacci numbers are coprime. Here is an easy example:

$$\begin{aligned} 13 &= 1 * 8 + 5 \\ 8 &= 1 * 5 + 3 \\ 5 &= 1 * 3 + 2 \\ 3 &= 1 * 2 + 1 \\ 2 &= 2 * 1 + 0. \end{aligned}$$

Note that $s(13,8) = s(F_7, F_6) = 5$. More generally, one needs exactly $n$ steps to compute $\gcd(F_{n+2}, F_{n+1})$, because there are $n$ remainders (including $0$) in the general scheme above. The next results records this fact and generalizes it:

**Proposition 3.11.** *(i) We have $s(F_{n+2}, F_{n+1}) = n$ for all $n \geqslant 1$.*
*(ii) If $a > b > 0$ and $s(a,b) = n$ then*

$$a \geqslant F_{n+2} \quad \textit{and} \quad b \geqslant F_{n+1}.$$

*Proof.* It remains to prove the second statement, which we do by induction on $n$. It is obviously true for $n = 1$ since $F_3 = 2$ and $F_2 = 1$ and one only step is needed. Assume that the second statement is true when $n$ is replaced by $n - 1$. Let $r$ be the first remainder:

$$a = qb + r = q_0 b + r_1.$$

Since $s(a,b) = s(b,r) + 1$, we have $s(b,r) = n - 1$. By hypothesis,

$$b \geqslant F_{n+1} \quad \text{and} \quad r \geqslant F_n.$$

But then

$$a \geqslant b + r \geqslant F_{n+1} + F_n = F_{n+2}.$$

So the second statement is true for our fixed value of $n$. Therefore it is true for all $n$. □

There are other striking properties relating Fibonacci numbers to division and Euclid's algorithm. We quote without proof the

**Theorem 3.12.** *Let $m, n \geqslant 3$. Then $m|n$ if and only $F_m|F_n$.*

This means that the mapping $n \mapsto F_n$ 'respects' divisibility, and (as an exercise) it follows from the theorem that

$$\gcd(F_m, F_n) = F_{\gcd(m,n)}.$$

A simple example is $\gcd(F_8, F_{12}) = \gcd(21, 144) = 3 = F_4$, 4 being $\gcd(8, 12)$.

Another problem a bit beyond the scope of the course is to find the most efficient way of computing the Fibonacci numbers themselves. For example, how many steps and how much memory is needed to compute $F_{12}$?

Returning to Euclid's algorithm, we shall use the formula

$$F_n = \frac{1}{\sqrt{5}}(\varphi^n - \psi^n)$$

from §2.2, where (recall) $\varphi$ and $\psi$ are the roots of $x^2 - x - 1 = 0$ with $\psi < 0$. If $n$ is odd then $\psi^n < 0$; we deduce that

$$F_n > \frac{1}{\sqrt{5}}\varphi^n \quad \text{if } n \text{ is odd.}$$

Now suppose that $a > b > 0$ and $s(a, b) = n$. By part (ii) of the last Proposition,

$$b \geqslant F_{n+1} > \frac{1}{\sqrt{5}}\varphi^n.$$

The second equality is true for all $n$, because if $n$ is even we can actually replace $\varphi^n$ by $\varphi^{n+1}$, whereas if $n$ is odd we first use the fact that $F_{n+1} \geqslant F_n$. Therefore $\sqrt{5}\, b > \varphi^n$ and

$$\tfrac{1}{2}\lg 5 + \lg(b) > n \lg(\varphi).$$

Dividing by $\lg(b)$ shows and letting $b \to \infty$ shows that there exists a constant $c > 0$ such that

$$s(a, b) = n < c \lg b.$$

This can be expressed in 'big O' notation by the

**Theorem 3.13.** *Let $a > b > 0$. Then*

$$s(a, b) = O(\lg b) \quad \text{as} \quad b \to \infty.$$

We conclude on a more elementary note by constructing a counterpart of the greatest common divisor. Let $a, b \in \mathbb{N}$, and set $g = \gcd(a, b)$. In particular, $g$ is a common divisor and we can write

$$a = ga', \qquad b = gb'.$$

Consider the positive integer

$$\ell = \frac{ab}{g} = ga'b'.$$

**Proposition 3.14.** *$\ell$ is the lowest common multiple of $a$ and $b$. That is,*

1. *$a \mid \ell$ and $b \mid \ell$;*
2. *if $a \mid m$ and $b \mid m$ then $\ell \leqslant m$.*

*Proof.* Condition 1. is immediate.
For 2., write $m = am_1 = bm_2$, and recall that $g = xa + yb$ for some $x, y \in \mathbb{Z}$. Consider

$$gm = (xa + yb)m = xabm_2 + ybam_1 = ab(ym_1 + xm_2).$$

Since $ym_1 + xm_2 \in \mathbb{Z}$, we have $\ell \mid m$. In particular $\ell \leqslant m$. □

# 4 Basic graph theory

## 4.1 Definitions

A *graph* consists of a finite set $V$ of vertices and a finite family $E$ of pairs of elements of $V$, the edges. (The edges are defined as a *family* rather than a *set* so as to allow for multiple edges between two vertices. Moreover, an edge could consist of a loop from a vertex to itself, so the pair should be an ordered pair even though the order will not matter until we discuss digraphs.)

**Example 4.1.** A 'triangle' with three vertices:

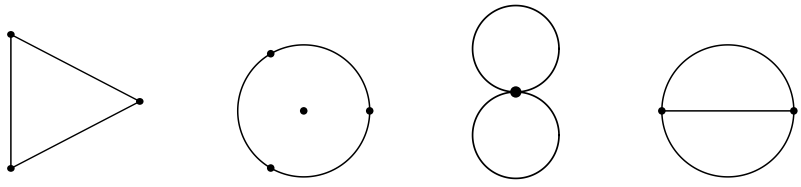
$$V = \{a, b, c\} \quad \text{or} \quad (a, b, c), \qquad E = (ab, bc, ca).$$

If we add an isolated vertex $d$, $V = \{a, b, c, d\}$ but $E$ stays the same.
A 'figure eight' with one vertex:

$$V = \{o\}, \qquad E = (oo, oo).$$

A lower case 'theta' with 2 vertices and 3 edges:

$$S = \{a, b\}, \qquad E = (ab, ab, ab).$$



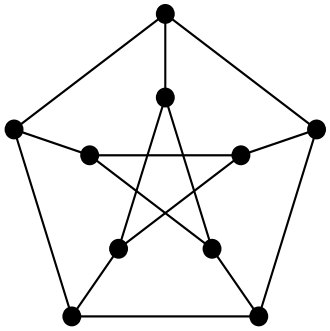
A graph is *simple* if there are no multiple edges and no loops. (In this case, $E$ can be defined as a *set* of unordered pairs of vertices, but it is still easier to write $ab$ or $v_1v_2$, or even 12, than $\{a, b\}$, $\{1, 2\}$ etc.)

The graph is *directed* or a *digraph* if each edge has an arrow, in which case each edge really is (in the logical sense) an ordered pair like $(a, b)$. To emphasize that the order is now important, one can denote the edge by $a \to b$, notation that may be closer to its meaning (like a one-way flow).

**Example 4.2.** Quite simple sets give rise to interesting graphs. Let $S = \{1, 2, 3, 4, 5\}$ and let $V$ be the set of all subsets of $S$ of size 2. So

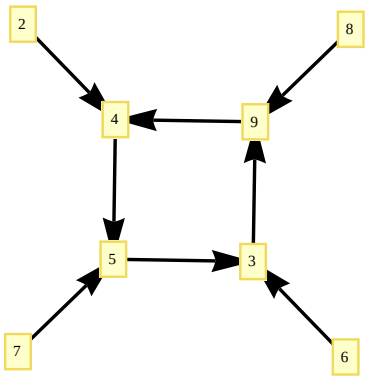
$$V = \{12, 13, 14, 15, 23, 24, 25, 34, 35, 45\}$$

(where 12 is shorthand for $\{1, 2\}$ etc., as explained above) and $|V| = \binom{5}{2} = 10$. We shall join two vertices (elements of $V$) by an edge if and only if the two subsets are *disjoint*. The result is called the *Petersen graph*. It is an example of a regular graph: the degree if every vertex is the same:



There are only 15 vertices, though it is impossible to represent the graph in 2 dimensions without edge crossings, since it is not a *planar* graph, a topic to be explored later.

**Example 4.3.** We shall define a digraph with vertex set $V = \{2, 3, 4, 5, 6, 7, 8, 9\}$ using modular arithmetic. Regard the elements of $V$ as congruence (or residue) classes modulo 11 (we have excluded 0, 1 and $10 \equiv -1$). The set of directed edges consists of pairs $(i, j)$ for which $j = i^2 \bmod 11$. The vertices $3, 4, 5, 9$ of the 'square' are the so-called *quadratic residues* modulo 11; they are elements admitting a square root mod 11:




The *degree* of a vertex $v$, written $d(v)$, is the number of occurrences of $v$ as an endpoint in the family of edges. Note that a loop will contribute 2 to the degree. If the graph $G$ is simple then the degree is also the number of vertices joined to $v$ by an edge. One often denotes the maximum degree of any vertex in $G$ by $\Delta(G)$, and the minimum by $\delta(G)$.

**Proposition 4.4.** *For* any *graph, the sum of the degrees of all vertices equals twice the number of edges:* $\sum_{v \in V} d(v) = 2|E|$.

*Proof.* We can prove this by induction on $|E|$. Given a graph with $n$ edges, remove any one. Either it joined two distinct vertices, or it was a loop at one vertex. In either case, we have reduced the sum of the degrees by 2. So assuming the result for $n-1$ edges (and it certainly holds for one edge), it remains true for $n$ edges. □

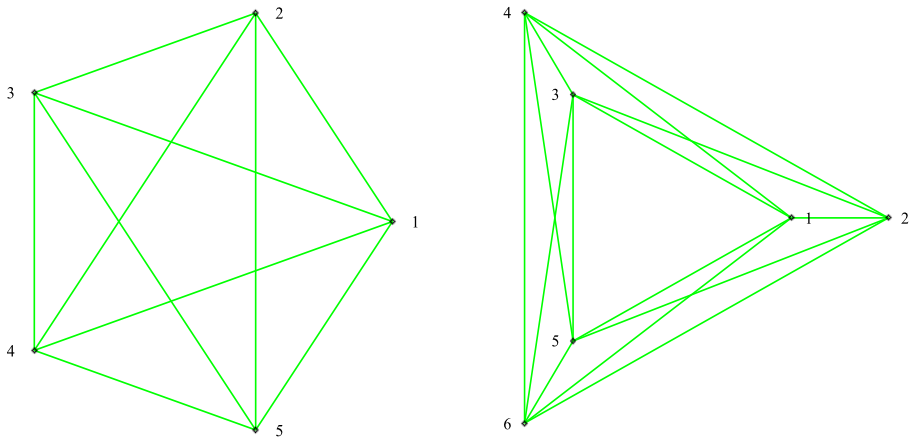
**Example 4.5.** There is no graph with vertex degrees $2, 3, 3, 5$.

**Definition 4.6.** *Two graphs* $G_1 = (V_1, E_1)$, $G_2 = (V_2, E_2)$ *are* isomorphic *if there exists a bijection* $f: V_1 \to V_2$ *between their vertex sets such that the number of edges between any two vertices* $a, b \in V_1$ *equals the number of edges between* $f(a), f(b) \in V_2$. *For simple graphs, this amounts to the assertion that*

$$(a, b) \in E_1 \iff (f(a), f(b)) \in E_2.$$

It is often easy to see that two graphs are *not* isomorphic, less easy to prove that they are. To show that two graphs are isomorphic, one must construct an isomorphism. To show that they are not isomorphic, one looks for some property which is different in the two graphs, such as the sequence of vertex degrees (if one is lucky), or the existence of cycles of a given length (see §4.2).

**Example 4.7.** A complete graph with $n$ vertices is a simple graph in which any two vertices are joined by an edge, so there are $\binom{n}{2}$ edges. Any two are isomorphic so we can speak of *the* complete graph with $n$ vertices. It is denoted $K_n$. The figures shown are representations of $K_5$ and $K_6$:



## 4.2 Connectivity

Two vertices $u, v$ of a graph $G$ are *adjacent* if there is an edge $uv \in E$ whose endpoints are $u$ and $v$. The edge is said to *join* $u$ and $v$.

A *walk* from $u$ to $v$ is a sequence of edges

$$v_0v_1,\ v_1v_2, \ldots, v_{n-1}v_n$$

with $u = v_0$ and $v = v_n$. The length of the walk is the number $n$ of edges (we also allow $u = v$ and $n = 0$).

A walk may be written $v_0v_1\cdots v_n$. Best not to write $v_0 \to v_1 \to \cdots \to v_n$ if $G$ is not a digraph.

A *trail* is a walk with no repeated edges.

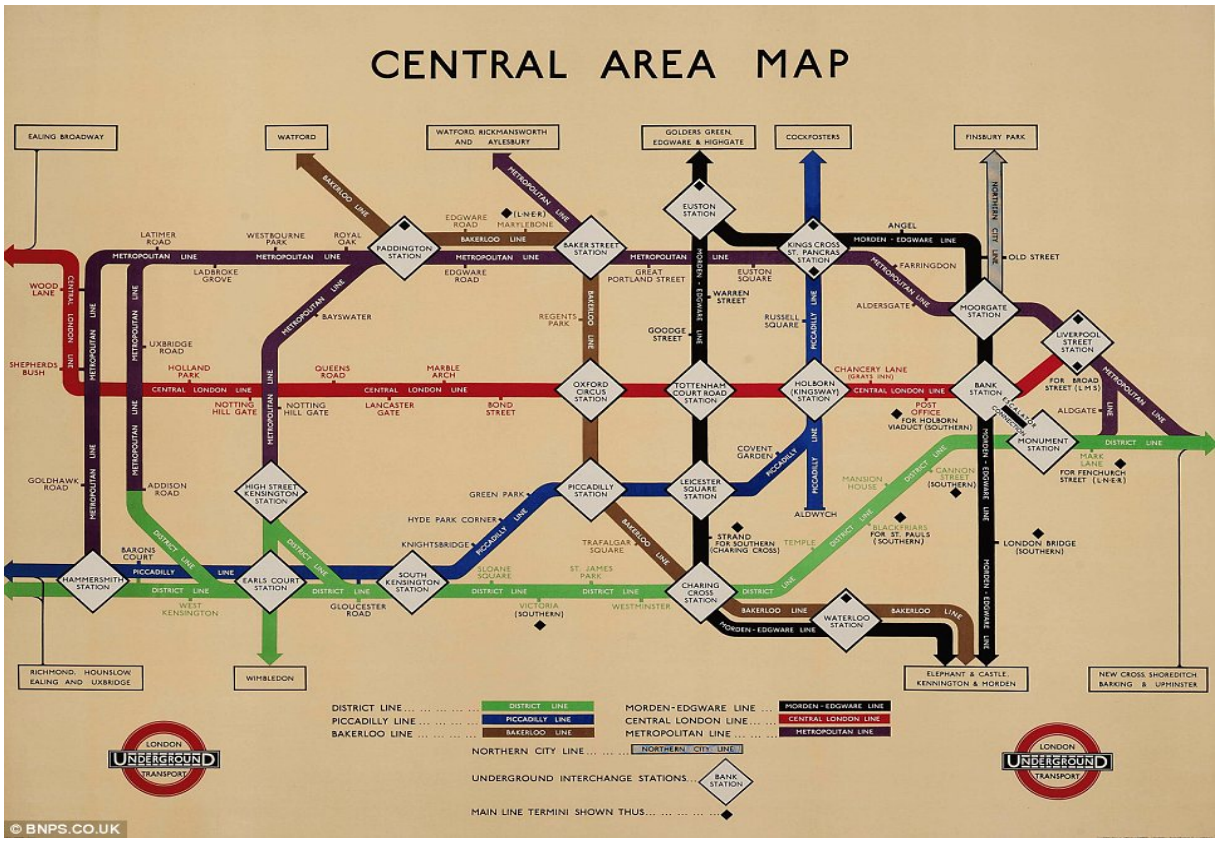
A *path* is a trail with no repeated vertices except possibly $v_0 = v_n$.

A walk/trail/path is *closed* if $v_0 = v_n$, i.e. it starts and finishes at the same vertex.

A *cycle* is a closed path with at least one edge. An $n$-cycle is a cycle of length $n$. Hence a loop is a 1-cycle and a 2-cycle only appears if there is a multiple edge.

*Examples.* Referring to the old underground map below, consider the stations

| | |
|---|---|
| A Aldwych! | |
| C Charing Cross | H Holborn |
| O Oxford Circus | P Picadilly Circus |
| S South Kensington | T Tottenham Court Road. |




Then

| | | | |
|---|---|---|---|
| LTOPL | is a | 4-cycle (and path) | |
| SPOTHA | | path of length 5 | |
| SPLTOPC | | trail (and walk) | [repeated vertex] |
| SPLTOP | | trail (and walk) | [P still repeated] |
| SPLTOPLH | | walk | [repeated edge] |

**Definition 4.8.** *Two vertices $u, v$ of a graph $G$ are* connected *if one can walk from one to the other and we can write $u \sim v$. (This implies there is a* path *between any two vertices, why?) Then $\sim$ is an equivalence relation, and it partitions $V$ into one or more subsets, the* components *of $G$. The graph $G$ itself is* connected *if there is just one component, in which case any two of its vertices are joined by a path.*

*A* subgraph *of a graph $G = G(V, E)$ is any graph $G' = G'(V', E')$ such that $V' \subseteq V$ and $E' \subseteq E$. Note that $E'$ might not include all the edges in $G$ that join vertices in $V'$, though if it does one says that $G'$ is* vertex-induced *(from $V'$).*

A component of a graph $G$ is then a maximal connected subgraph $G'$, i.e. $G'$ is connected but if one more vertex or edge from $G$ is added then the subgraph is no longer connected.

A *disconnecting set* of a connected graph $G$ is a set of edges whose removal makes the new graph disconnected. When an edge is removed the vertices which are its endpoints are retained. Of particular importance is the special case:

**Definition 4.9.** *A* cutset *of a connected graph $G$ is a disconnecting set, no proper subset of which is a disconnecting set.*

*Thus, if any one of the edges in a cutset is retained then the graph stays connected. If a cutset consists of a single edge then this edge is called a* cut-edge *or a* bridge*.*

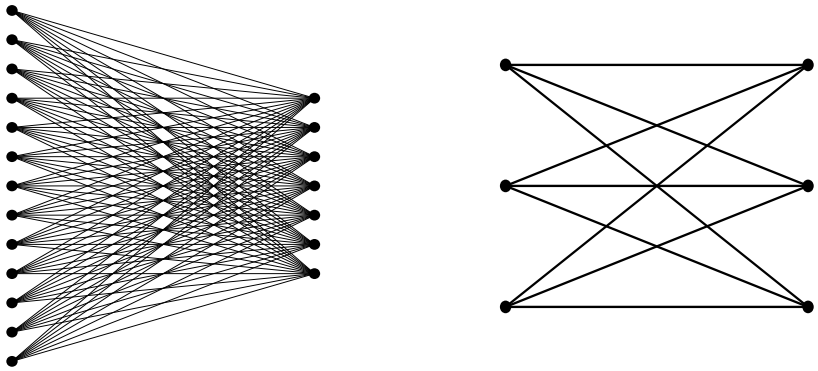
*A* separating set *of a connected graph $G$ is a set of vertices whose removal makes the new graph disconnected. When a vertex is removed all the edges which have it as an endpoint are also removed. If a separating set of a connected graph $G$ consists of a single vertex that vertex is called a* cut-vertex*.*

*A* tree *is a connected graph with no cycles, though there are alternative definitions (to be seen later).*

*A* bipartite graph *is a graph in which $V$, the set of vertices, is the union of two disjoint non-empty sets $V_1$ and $V_2$ and all the edges have one endpoint in $V_1$ and the other endpoint in $V_2$. (Hence no two vertices in $V_1$ are adjacent and no two vertices in $V_2$ are adjacent.) One can also prove the useful*

**Theorem 4.10.** *A graph is bipartite if and only if there are no cycles of odd length.*

**Example 4.11.** $K_{n,m}$ is the complete bipartite graph where $|V_1| = n$, $|V_2| = m$ and every vertex in $V_1$ is adjacent to every vertex in $V_2$. Here we see $(m, n) = (13, 7)$ and $(3, 3)$:



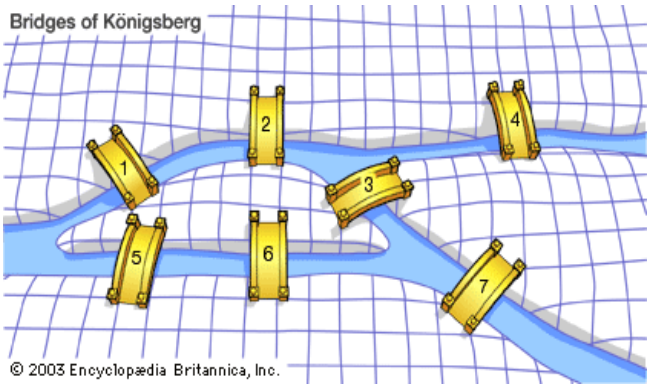
## 4.3 Eulerian graphs

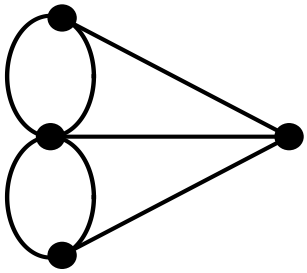
Recall that a cycle is a closed path. It is therefore a walk that passes through no edge twice and (apart from being joined up so that it finishes where it started) no vertex twice.

**Lemma 4.12.** *If a graph $G$ is connected and every vertex has degree at least 2 then $G$ contains a cycle.*

*Proof.* A loop defines a 1-cycle, and a multiple edge defines one or more 2-cycles, so we may assume that $G$ is simple. Choose any vertex $v_1$. Let $v_2$ be an adjacent vertex (meaning there is an edge $v_1v_2$). Since $v_2$ has degree at least 2, it must have an adjacent vertex $v_3$ distinct from $v_1$. Continuing in this way, since $V$ is a finite set, we must eventually find a vertex already in the list, say $v_j = v_i$ with $1 \leqslant i < j$, and $v_{i+1}, \ldots, v_j$ all distinct. Then $v_iv_{i+1}\cdots v_j$ is the desired cycle. □

**Example 4.13.** Graph theory is said to have begun with Euler's paper solving the problem of the seven bridges of Könisberg near the Baltic Sea (now the Russian city of Kaliningrad sandwiched between Poland and Lithuania and *disconnected* from Moscow). The question was whether there exists a *trail* that crosses each bridge exactly once.



There can't be one because when one converts the map into a graph with each land mass a vertex and each bridge an edge, all the vertices have odd degree, so it is impossible to 'come and go' without repeating an edge. Next we'll make this precise.

**Definition 4.14.** *A connected graph is* semi-Eulerian *if there is a trail which includes every edge of the graph. (Thus it passes along each edge exactly once and goes through every vertex, but some vertices can be visited more than once.) The graph is* Eulerian *if such a trail can be found that is closed.*

**Definition 4.15.** *Suppose that $G$ is a connected graph. Then $G$ is Eulerian if and only if every vertex has even degree.*

**Theorem 4.16.** *A connected graph is semi-Eulerian if and only if it has at most two vertices of odd degree.*

These results were known to Euler, who studied the bridges problem in 1736, though a formal proof was first given in 1871 by Hierholzer (who died that year aged 31).

*Note.* No graph can have an odd number of vertices of odd degree, because the 'total degree' must be even. We shall deduce the corollary from the theorem by a neat trick.

*Proof of the theorem.* If $G$ has a closed 'Eulerian trail', then follow along it from a starting vertex. At each new vertex, we can mark off the arriving edge and the departing edge. Both are traversed once and only once, so the degree of each vertex (including the start = finish one) must be even. This justifies the 'only if' part of the theorem.

The harder 'if' part can be proved by induction on the number of edges of $G$, using the lemma to first remove a cycle and work on what is left of $G$. However we shall explain how one can effectively construct a closed Eulerian trail using (what is now called) Hierholzer's algorithm.

Suppose then that $G(V, E)$ is a connected graph, all of whose vertices are of even degree. Fix a vertex $v_0 \in V$. A first observation (which is a refinement of the lemma) is that one always find a trail (with at least one edge) in $G$ that eventually returns to $v_0$. This is because, whichever

edge one chooses to walk along and whichever vertex one arrives at, there will always be an edge available to leave that vertex. (This is guaranteed by the even degree property, even if the trail has already passed through the same vertex.) But the graph is finite, so we must eventually return to $v_0$.

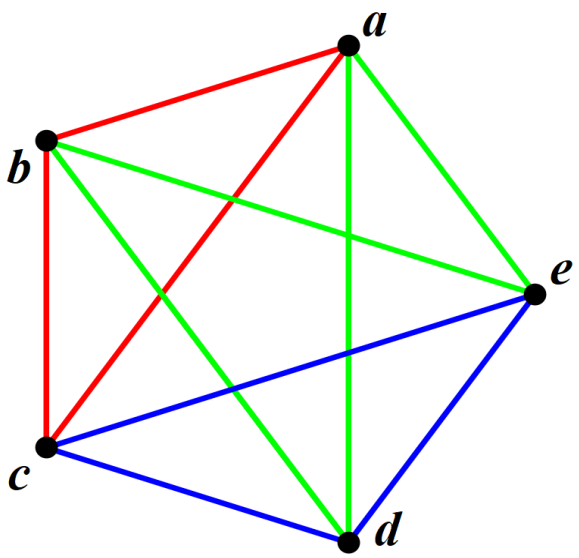
With this observation, we type out the algorithm in pseudocode. Each step can in fact by implemented by instructing a computer how to process data (lists of vertices and edges) representing the graph. Let $T$ be a closed trail of any length (even zero) starting and ending at $v_0$.

```
# Hierholzer's algorithm
def aug(T):
    if T contains all the edges of G:
        return T
    else:
        walk to the first vertex v1 in T with a spare edge
        construct new trail T1 from v1 to v1 using spare edges
        insert T1 as a detour into T to form a longer trail T+
    return aug(T+)
```

A 'spare edge' means an edge of $G$ that does not appear in $G$; there will always be one if $T$ is not already Eulerian. The argument above shows that the new trail $T_1$ can be found, and inserting it as a detour into $T$ yields the 'augmented' trail $T^+$. If $T$ is empty, $T_1$ will be a trail with at least one edge (it could be a loop) based at $v_0$. We could have asked the programme to output $T^+$, but in that case it would be necessary to start afresh with $T^+$ in place of $T$. By calling aug($T^+$), we have made the function recursive by requiring the program to keep applying the 'else' step until it finds a trail with all the edges of $G$.

**Example 4.17.** The complete graph $K_n$ is Eulerian if and only if $n$ is odd (for then all vertex degrees are even). Let us apply the algorithm to $K_5$ with vertices $a, b, c, d, e$. Take $T$ to be the empty trail (denoted $\varnothing$) at $v_0 = a$. When it comes to spare edges, let's always move to a vertex of least alphabetical order. This causes the program to manufacture three separate detours, illustrated below with the colours red, green, blue, and to return the Eulerian path *abdaebcdeca* with (as has to be) 10 edges.

$$
\begin{array}{lllll}
\text{input} \rightarrow & T & = & \varnothing & \\
 & & & \overbrace{\phantom{abca}} & \\
 & T_1 & = & abca & \\
 & T^+ & = & abca & \\
 & T & = & a\underline{b}ca & \\
 & & & \overbrace{\phantom{bdaeb}} & \\
 & T_1 & = & bdaeb & \\
 & T^+ & = & abdaebca & \\
 & T & = & abdaeb\underline{c}a & \\
 & & & \overbrace{\phantom{cdec}} & \\
 & T_1 & = & cdec & \\
 & T^+ & = & abdaebcdeca & \\
 & T & = & abdaebcdeca & \leftarrow \text{output}
\end{array}
$$




*Proof of the corollary assuming the theorem.* Suppose that $G$ has exactly two vertices $u, v$ of odd degree. Add an *extra* edge from $u$ to $v$. In this section, we are not assuming graphs are simple, so if there was already an edge from $u$ to $v$ (or more than one), we simply add another. The point is that the new graph has all its vertices of even degree so, by the theorem, it possesses a Eulerian trail, which must incorporate the extra edge. If the latter is removed, we end up with a Eulerian trail for $G$ that starts at $u$ and finishes at $v$. Conversely, if we had such a trail we can make it closed by again adding an extra edge, and conclude that the modified graph has all its other vertices of even degree. □

Now suppose that all vertices have even degree. If the graph has loops or extra edges connecting two vertices then (by first removing loops and pairs of multiple edges) we can modify any trail on the remaining simple graph to include what we removed. So we may suppose that $G$ is simple.

Argue by induction on the number $|E|$ of edges. If this is 3, then $G$ itself is a single 3-cycle.

Now take a graph with $|E| > 3$, and assume that the result is true for graphs with less edges. By the lemma, $G$ contains a cycle $C$, and we can assume this is not all of $G$ (otherwise we are finished). Remove all its edges and any isolated vertices; the resulting graph may have a number of components, $G_1, \ldots, G_k$ with $k \geqslant 1$. Each component $G_i$ is simple, has less edges than $G$, and its vertices have even degree (because if one was on $C$ we have removed two edges). By way of induction hypothesis, we may assume that $G_i$ has a closed trail $C_i$ passing along all the edges of $G_i$ exactly once.

If we now start from a vertex of $C$ and walk along it, the first time we encounter a component, we can insert the trail $C_i$, and then continue along $C$ until we meet a vertex of the next component of $G$ minus the cycle. And so on. This process gives a Eulerian trail for $G$, and the induction succeeds. □

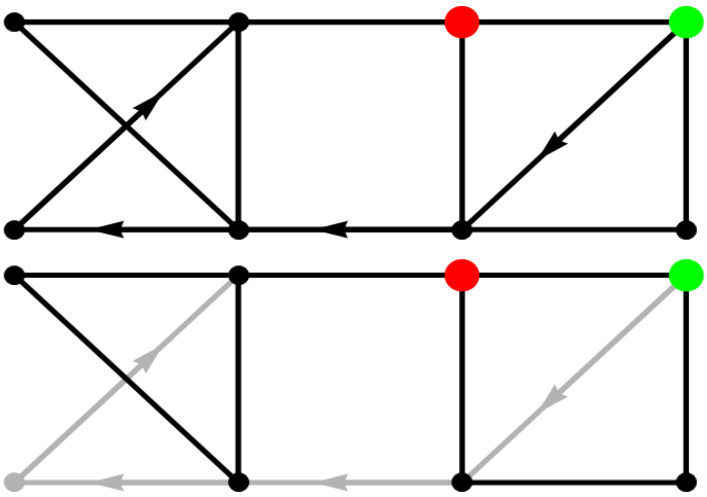
A more popular way of finding an Eulerian trail is *Fleury's algorithm*. It can be summed up by the slogan "do not burn bridges". Recall that a *bridge* in a connected graph is an edge that when removed will cause the new graph to be disconnected. We shall mimic our description of the previous algorithm to define Fleury's, but instead of proving that it always succeeds we make do with an example.

We shall implement the algorithm to construct a semi-Eulerian trail, so this time let $G$ be a connected graph with at most two vertices of odd degree. Fix a vertex $v_0$ of odd degree if there is one, and a trail $T$ starting at $v_0$, possibly empty.

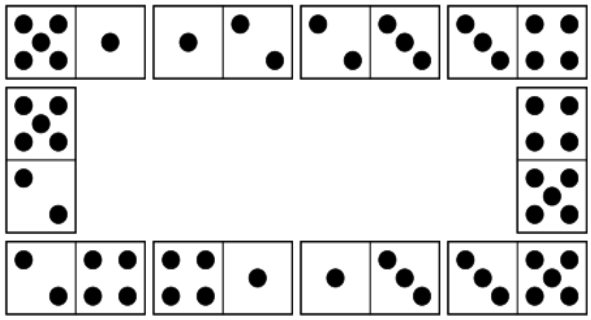
```
# Fleury's algorithm
def fl(T):
    G' = G with edges and isolated vertices of T removed
    w = last vertex in T
    choose an edge of G' at w avoiding a bridge if possible
    T+ = T with the new edge added
    return fl(T+)
```

Then $\mathrm{fl}(T)$ will be a semi-Eulerian trail for $G$.

**Example 4.18.** In the graph illustrated immediately below, there are two 'odd' vertices, so we start top right (green), and aim to finish at the adjacent (red) vertex. After traversing 4 edges, we have constructed a trail $T$ shown in grey. To obtain $G'$, we discard its 4 grey edges and 1 isolated vertex. The point then is that we must complete the 'left wing' before crossing the top bridge. (This is obvious to the human eye, but programming a computer to recognize a bridge would be an unwanted complication.) Thus, $T^+$ will be formed from $T$ by adding either the *left* horizontal edge or the vertical one.



**Example 4.19.** We noted that $K_n$ is Eulerian iff $n$ is odd (for then all vertex degrees are even). Each of the $\binom{7}{2} = 21$ edges of $K_7$ can be identified with a domino, and each domino with equal values side by side defines a loop at a vertex of $K_7$. A Eulerian trail in $K_7$ with its 7 loops is then a 'domino cycle', like the smaller $n = 5$ version:



## 4.4 Hamiltonian cycles

**Definition 4.20.** *A connected graph is* Hamiltonian *if there is a closed path (and so, a cycle) that visits every vertex exactly once.*

The closed path is called a *Hamiltonian cycle*. A Eulerian trail must, by its very nature visit every vertex, but it is allowed to do so more than once. By contrast, a Hamiltonian cycle must visit each vertex only once, and will not in general pass along every edge. Despite the analogy with Eulerian trail, deciding when a graph is Hamiltonian is much harder and remains the subject of current research.

There are no wide-ranging theorems that characterize Hamiltonian graphs, and most results that are known are rather restrictive and of limited value for the graphs encountered in this course. The following is one such result, which we state without proof.

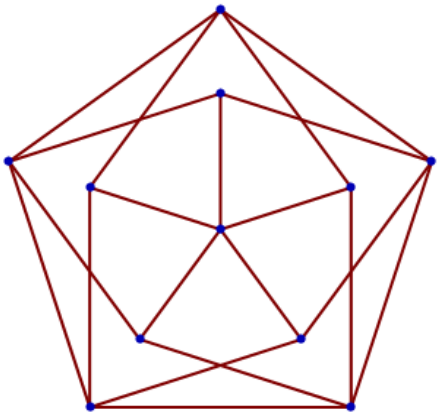
**Theorem 4.21** (Gabriel Dirac's Theorem)**.** *Let $G(V, E)$ be a simple graph with $|V| = n \geqslant 3$. If $d(v) \geqslant n/2$ for all $v \in V$ (equivalently, $n \leqslant 2\delta(G)$) then $G$ is Hamiltonian.*

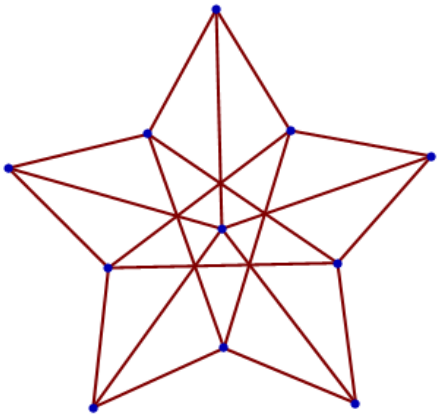
**Example 4.22.** Since $\delta(K_n) = n - 1$, the theorem does tells us that $K_n$ is Hamiltonian for all $n \geqslant 2$.

A bipartite graph with an odd number of vertices cannot be Hamiltonian, because any Hamiltonian cycle would be odd.

The 12 edges and 6 vertices of an octahedron form a graph that (like those arising from the other platonic solids) is Hamiltonian.

The Petersen graph is not Hamiltonian, but it does have a (non-closed) path passing though every vertex. The *Grötzsch graph* $\ddot{G}$ has 11 vertices and 20 edges, and by contrast *is* Hamiltonian. Here are two representations of it:





**Exercise 4.23.** For which values of $n$ does $\ddot{G}$ possess an $n$-cycle?

A celebrated example of a Hamiltonian graph is the *knight's graph* $N$ with 64 vertices, defined as follows. Consider a knight moving freely on a standard $(8 \times 8)$ chess board, without other peices to get in the way. Assign a vertex to each square, and declare two vertices to be adjacent if a knight can move legitimately between them (that is, two squares one way and one sideways). If a knight starts on one of the 16 squares near the centre of the board, it has 8 squares it can move to, but this choice is reduced closer to an edge of the board. The edges of the central $6 \times 6$ square allow 6 moves, except its corners that allow only 4. Overall, $N$ has

| | | |
|---|---|---|
| 4 | vertices of degree | 2 |
| 8 | ” | 3 |
| 20 | ” | 4 |
| 16 | ” | 6 |
| 16 | ” | 8 |

It follows that the total number of edges is

$$e = \tfrac{1}{2}(4 * 2 + 8 * 3 + 20 * 4 + 16 * 6 + 16 * 8) = 168,$$

and a Hamiltonian cycle uses 64 of them. Not surprisingly, it is difficult to find such a knight's tour from scratch. The following instance was constructed (by the lecturer for Module Selection)

by starting in a corner, and using Warnsdorff's (not infallible) rule: *move the knight to a square from which there is the least number of successive moves possible.* The knight therefore hugs the edges as long as it can; each number (from 1 to 64) labels the start of the corresponding edge of the cycle:

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 7 | 42 | 9 | 24 | 5 | 40 | 19 | 22 |
| 10 | 25 | 6 | 41 | 20 | 23 | 4 | 39 |
| 43 | 8 | 55 | 34 | 51 | 38 | 21 | 18 |
| 26 | 11 | 58 | 37 | 56 | 33 | 52 | 3 |
| 59 | 44 | 35 | 54 | 63 | 50 | 17 | 32 |
| 12 | 27 | 62 | 57 | 36 | 53 | 2 | 49 |
| 45 | 60 | 29 | 14 | 47 | 64 | 31 | 16 |
| 28 | 13 | 46 | 61 | 30 | 15 | 48 | 1 |

The knight's graph is bipartite because the vertices (squares) are divided into black and white, and a knight changes colour each move. From the remarks above, there cannot exist a knight's tour on (for example) a $7 \times 7$ chequer board.

# 5 Vertex colouring

## 5.1 Chromatic number

In this section, **all graphs will be simple**. The problem then is to assign a colour to each vertex of a graph so that no two adjacent vertices have the same colour, and to do this using the least number of colours.

In mathematical language, a vertex colouring of a graph $G(V, E)$ is a mapping $c\colon V \to \mathbb{N}$ with the property

$$uv \in E \implies c(u) \neq c(v).$$

This means that adjacent vertices have different colours (values of $c$), and to do this with as few colours as possible means reducing the image of $c$. We are using positive integers to label the colours, though in examples we shall use Greek letters in their alphabetical order

$$\alpha,\ \beta,\ \gamma,\ \delta,\ \varepsilon,\ \zeta,\ \ldots$$

On screen, we can use actual colours, such as red, green, blue, and yellow.

A graph is *$k$-colourable* if we can find a vertex colouring with $|\operatorname{Im} c| = k$. In this case, one of $k$ colours can be assigned to each vertex such that no two adjacent vertices have the same colour.

**Definition 5.1.** *The* chromatic number *of a simple graph $G$, denoted $\chi(G)$, is the least value of $k$ for which $G$ is $k$-colourable.*

*Observation.* It is impossible to find a vertex colouring if $G$ has a loop $uu$. Multiple edges do not affect the colouring problem, but in any case, we restrict to simple graphs.

$\chi(G) = 1$ if and only if all the vertices of $G$ are isolated, not an interesting scenario.

$\chi(G) = 2$ if and only if $G$ is bipartite (see the end of §4.2). This is really the definition of bipartite: one can regard the colours as 'positive' and 'negative' or (in the chess example in §4.4) white and black.

If $|V| = n$ then obviously $G$ is $n$-colourable and $\chi(G) \leqslant n$, but usually we can make do with *many fewer* colours because $\chi(G)$ is much more closely related to the degrees of vertices in $G$. In this sense, the next example is not typical.

**Example 5.2.** Since every vertex of $K_n$ is joined to every other, we have $\chi(K_n) = n$.

Let $C_n$ denote the 'cycle graph' consisting of the $n$ vertices and $n$ edges of a polygon. Then

$$\chi(C_n) = \begin{cases} 2 & \text{if } n \text{ is even,} \\ 3 & \text{if } n \text{ is odd.} \end{cases}$$

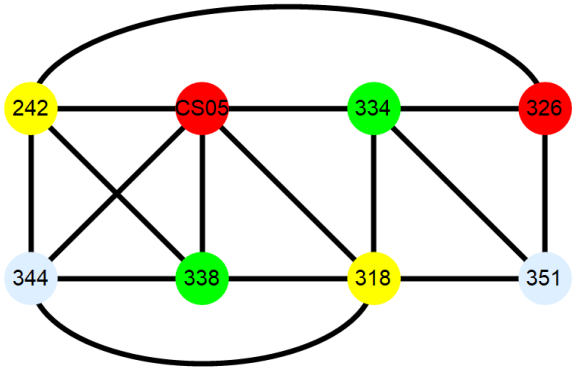
It follows that:

$$\begin{aligned} G \text{ contains } K_n \text{ as a subgraph} &\quad\Rightarrow\quad \chi(G) \geqslant n \\ G \text{ contains an odd cycle as a subgraph} &\quad\Rightarrow\quad \chi(G) \geqslant 3. \end{aligned}$$

*Application.* Vertex colouring can be used to solve the *timetabling problem* with:

- a set of modules (the vertices);
- groups of students who have selected pairs of modules (the edges);
- a limited number of time slots (the colours);
- an unlimited number of lecture rooms (to simplify the problem).

If no adjacent vertices have the same colour all students can attend all the lectures for the modules they have chosen!



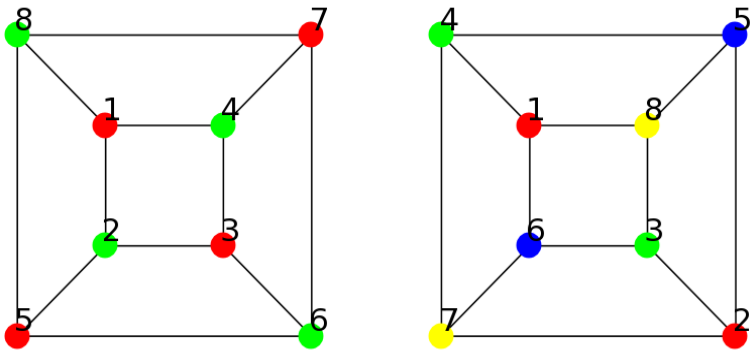

In the graph $G$ above, a popular module has 'degree' 5. Four modules (left) form a complete subgraph $K_4$, so no less than 4 colours will suffice. Thus $\chi(G) = 4$.

## 5.2 Colouring results

The whole theory of vertex colouring depends on the so-called *greedy algorithm*. This is a natural way of colouring the vertices when they are put in order, and (in very informal language) can be exporessed as follows:

```
label the vertices v_1,v_2,...,v_n
label the colours 1,2,...,n
assign to v_1 colour 1
for j in range(2,n+1):
    S = {colours assigned to vertices adjacent to v_j}
    assign to v_j the smallest colour not in S
return S and the assignments
```

**Example 5.3.** Different vertex orderings can give very different numbers of colours. The 'cube graph' is bipartite so $\chi = 2$, and this is illustrated on the left. But the greedy algorithm produces 4 colours (here $1 = R$, $2 = G$, $3 = B$, $4 = Y$) when applied to the ordering on the right:




For any $G$, it can be shown that there always exists some vertex ordering for which the greedy algorithm gives the minimum number (namely, $\chi(G)$) of colours.

Recall that

$$\Delta(G) = \max_{v \in V} d(v),$$

where $d(v)$ is the degree (valency) of the vertex $v$. For example, $\Delta(K_n) = n-1$ and $\chi(K_n) = n$. Here are the main results on vertex colouring, in increasing difficulty.

**Lemma 5.4.** *If $G$ is simple then $\chi(G) \leqslant \Delta(G) + 1$.*

**Proposition 5.5.** *If $G$ is simple, connected and not regular (not all vertices have degree $\Delta$) then $\chi(G) \leqslant \Delta(G)$.*

**Theorem 5.6** (Brooks' theorem)**.** *If $G$ is simple, connected and neither complete nor an odd cycle (so $G \not\cong K_n$ and $G \not\cong C_{2k+1}$) then again $\chi(G) \leqslant \Delta(G)$.*

**Question 5.7.** *Why must we add 'connected' in the last two statements?*

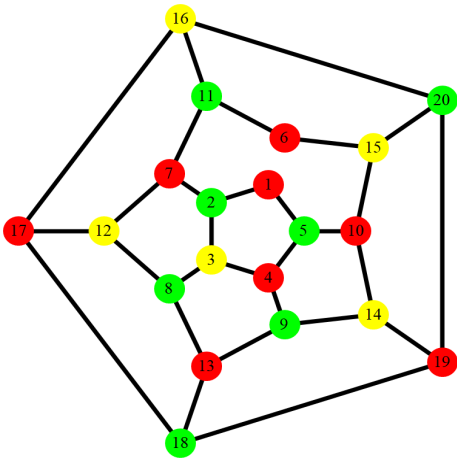
*Proof of the lemma.* Let $k = \Delta(G)$, and fix a set of $k+1$ colours. Take the vertices in *any* order. Suppose we have managed to colour some (or $\varnothing$) of them. The next (or first) vertex is surrounded by at most $n$ adjacent vertices, so we can colour it without a clash, and move on to the next vertex in the list. □

Note that the proposition reduces the proof of the theorem to the case of regular graphs. We shall prove the former, but not the latter. Incidentally, if we assume that $\Delta(G) \geqslant 3$, there is no need to mention the odd cycle in the theorem.

## 5.3 Brooks' algorithm

The proof of the proposition is accomplished by implementing a 2-step process that is sometimes called *Brooks' algorithm*. This produces an ordering of the vertices (or a re-ordering if we have one already), relative to which (as we shall explain) the greedy algorithm will *always* succeed with at most $\Delta$ colours.

We shall illustrate it with a modification of the dodecahedral graph with 12 vertices, in which we have removed one edge (leaving 29) so that the graph is no longer regular. We have labelled the vertices with numbers $1, 2, \ldots, 20$ in no special way, and the missing edge is $(1, 6)$.




Rather than type out the instructions, we shall explain the process with a table.

In general, let $G$ be a graph with $\Delta(G) = k$ but not regular. Choose a vertex with degree less than $k$ to start, and place the vertex in a 'queue'. At each stage, the first element in the queue is moved to the left-hand list, and any adjacent vertices not previously queued are added at the back of the queue (in any order, but for definiteness one can add them in increasing order). In our example, $k = 3$, and we can start with vertex 6. Later on, vertex 5 is adjacent to $1, 4, 10$, but $1, 10$ have already appeared in the queue, so only 4 is added (see the boxes).

At the end of the process, *the list of vertices must be read in reverse order*. In our case, we obtain

$$v_1 = 13, \quad v_2 = 9, \quad \ldots \quad , v_{20} = 6.$$

Now apply the greedy algorithm to this (reversed) list. By construction, each vertex in the list can be adjacent to at most $k-1$ previous elements in the list, because it is also adjacent

to one of the vertices above it in the table. For example, there is no problem colouring $v_{10}$ because it is only adjacent to $v_4$, and without checking we know it must be adjacent to some $v_j$ with $j > 10$ (it first entered the queue when 10 entered the list).

This scheme shows that we can colour $G$ with at most $k$ colours. In our case, we recover the colouring shown. If we restore the missing edge, this is not a valid colouring, though Brooks' theorem implies that the dodecahedral graph is also 3-colourable and has $\chi = 3$.

| | list | Q | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | | 6 | | | | | | |
| $v_{20}$ = | 6 | 11 | 15 | | | | | |
| $v_{19}$ = | 11 | 15 | 7 | 16 | | | | |
| | 15 | 7 | 16 | [10] | 20 | | | |
| | 7 | 16 | 10 | 20 | 2 | 12 | | |
| | 16 | 10 | 20 | 2 | 12 | 17 | | |
| | 10 | 20 | 2 | 12 | 17 | 5 | 14 | |
| | 20 | 2 | 12 | 17 | 5 | 14 | 19 | |
| | 2 | 12 | 17 | 5 | 14 | 19 | [1] | 3 |
| | 12 | 17 | 5 | 14 | 19 | 1 | 3 | 8 |
| | 17 | 5 | 14 | 19 | 1 | 3 | 8 | 18 |
| $v_{10}$ = | [5] | 14 | 19 | 1 | 3 | 8 | 18 | [4] |
| | 14 | 19 | 1 | 3 | 8 | 18 | 4 | 9 |
| | 19 | 1 | 3 | 8 | 18 | 4 | 9 | |
| | 1 | 3 | 8 | 18 | 4 | 9 | | |
| | 3 | 8 | 18 | 4 | 9 | | | |
| | 8 | 18 | 4 | 9 | 13 | | | |
| | 18 | 4 | 9 | 13 | | | | |
| $v_3$ = | [4] | 9 | 13 | | | | | |
| $v_2$ = | 9 | 13 | | | | | | |
| $v_1$ = | 13 | | | | | | | |

This table emphasizes the dynamic nature of the process, and how the data might be stored on a computer. Each vertex is processed separately and its 'new neighbours' put in the queue using the *First In First Out* (FIFO) principle, which contrasts with that of a stack (FILO) we saw in recursive relations.

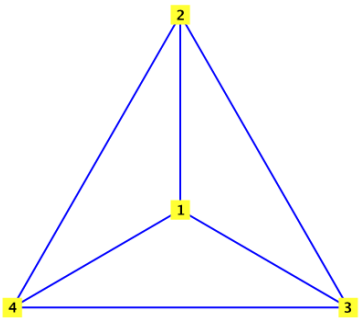
However, there is a lot of redundant information with the diagonals. In practice, one can form a long queue, ticking off the vertices as they are processed and crossing out vertices on the graph as soon as they enter the queue (as the initial one or neighbours of the current vertex).

**Exercise 5.8.** *Retain the edge* $(1,6)$ *and re-do the table starting with vertex 6 again. You will probably find that most vertex colours are the same but that at the final stage one requires a fourth colour. This is not a contradiction: even though the dodecahedral graph has* $\chi = \Delta = 3$, *it is important to remember that Brooks'* algorithm *is only guaranteed to use at most* $\Delta$ *colours when* $G$ *is* not *regular. We have not proved Brooks' theorem!*

# 6 Planarity

## 6.1 The Platonic graphs

We can represent $K_4$ the 'complete graph on four vertices' by the edges and vertices of a tetrahedron with transparent faces:




Unlike a square with its two diagonals added, this has the advantage that there are no 'false intersections' of its edges.

**Definition 6.1.** *A graph is called* planar *if it can be drawn in the plane without crossings, so that edges intersect only in vertices. When it has been drawn that way we shall call it a plane drawing.*

The phrase 'can be drawn' means 'is isomorphic to a graph', so 'planar' is a property of an *isomorphic class* — if it is true for one graph, it is true for any isomorphic graph. Our problem then is to understand how to decide whether such a class is planar.

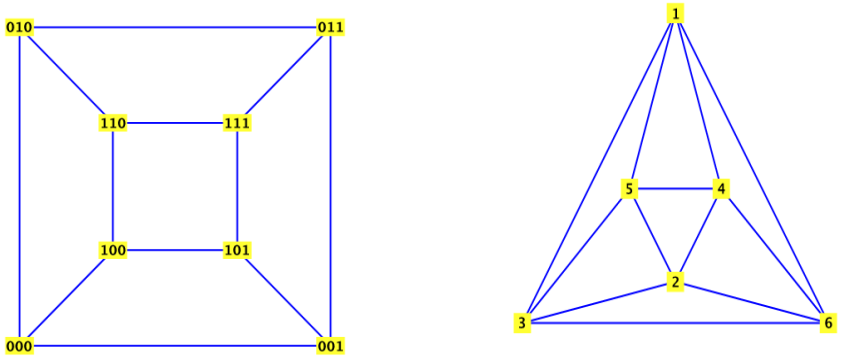
We shall begin with four more regular planar graphs, namely the ones determined by the vertices and edges of the remaining platonic solids. A *platonic solid* is a convex polyhedron (formed by intersecting a number of planes in space) with *congruent* faces each of which is a *regular polygon*. It is known that such a face must be a triangle, square or pentagon. Let

| | | |
|---|---|---|
| $n$ | denote the number of | vertices |
| $e$ | | edges |
| $f$ | | faces |
| $p$ | | edges bounding each face |
| $\Delta = q$ | | edges joined at each vertex |
| $\chi$ | chromatic number | |

Then the five Platonic solids and properties of the associated graphs are given by the table

| name | $n$ | $e$ | $f$ | $p$ | $q$ | $\chi$ | Eulerian? | Hamiltonian? |
|---|---|---|---|---|---|---|---|---|
| tetrahedron | 4 | 6 | 4 | 3 | 3 | 4 | no | yes |
| cube | 8 | 12 | 6 | 4 | 3 | 2 | no | yes |
| octahedron | 6 | 12 | 8 | 3 | 4 | 3 | yes | yes |
| dodecahedron | 20 | 30 | 12 | 5 | 3 | 3 | no | yes |
| icosahedron | 12 | 30 | 20 | 3 | 5 | 4 | no | yes |

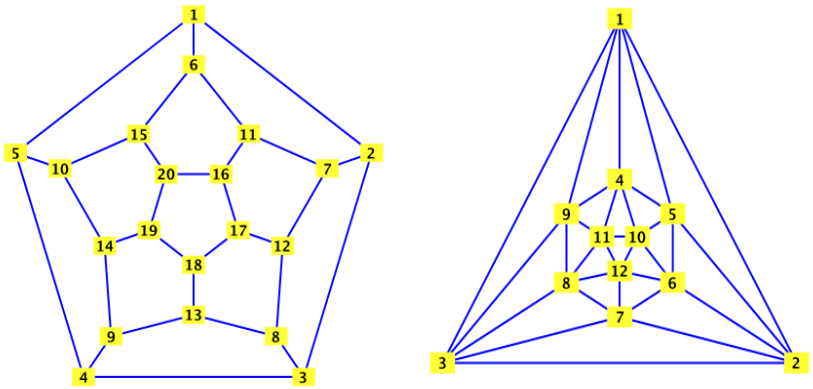
The Greek prefixes refer to the number of *faces*, for example *dodeca* means $2+10=12$. The last four come in pairs, in their data we swap $n \leftrightarrow f$ and $p \leftrightarrow q$. Here is the cube (or hexahedral) graph and the octahedral graph:




Recall that the cube graph is bipartite. Its $2^3$ vertices can be labelled by their Cartesian coordinates, which in the image are listed without commas:

$$000,\quad 001,\quad 010,\quad 011,\quad 100,\quad 101,\quad 110,\quad 111.$$

The octahedral graph is the only one of the five whose vertices all have even degree. The dodecahedral and icosahedral graphs are more complicated:




To convert $f$ into a number for a plane graph, we must count the outside as one face.

**Theorem 6.2** (Euler's formula)**.** *For any connected plane graph drawing,* $n - e + f = 2$.

*Proof.* This is a remarkably universal formula. It is valid when there is just 1 isolated vertex (and so 1 outside face). We can proceed by induction on $n$. Each time we add an edge joined to the previous graph, either its other end is 'free' (the vertex has degree 1) or it joins up an existing vertex. In the former case, we have added 1 edge and 1 vertex, in the latter case 1 edge and 1 face. Either way $n - e + f$ does not change. □

## 6.2 Detecting non-planarity

Recall that:

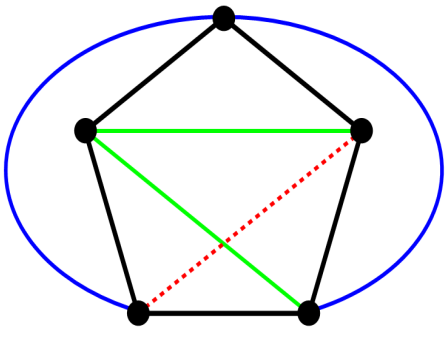
- $K_n$ is the complete graph with $n$ vertices and so $\binom{n}{2} = \frac{1}{2}n(n-1)$ edges;
- $K_{m,n}$ is the complete bipartite graph with $m+n$ vertices and $mn$ edges.

In particular, $K_5$ is the 'starred pentagon' and $K_{3,3}$ is the 'utility' graph representing the distribution of broadband, electricity, water to three consumers. Note that the edges and vertices of a cube form a subgraph of $K_{4,4}$ obtained by removing four edges.

**Proposition 6.3.** *Neither $K_5$ nor $K_{3,3}$ is planar.*

It follows that no planar graph can contain $K_5$ or $K_{3,3}$ as a *subgraph*.

*Proof.* This can be done by first principles (so no theory). Consider $K_5$; suppose it has a plane drawing with no redundant crossings. Since the abstract graph has a 5-cycle, that (taken on its own) will determine a pentagon in the plane. The positions of the other five edges in the drawing remain to be specified, and each one must either lie inside or outside the pentagon. They cannot all lie outside without a crossing, so let us suppose one lies inside. There is no way to distinguish any of the 5 remaining edges, so we pick one and assume it lies inside. After one more choice, the inside/outside positions are forced upon us, and we are stuck at the final step.



A similar argument works for $K_{3,3}$, though this has a 6-cycle. (Recall that any cycle in a bipartite graph must be even.) □

To formulate two important results, we need two new concepts for modifying graphs, namely *homeomorphism* and *contraction*.

**Definition 6.4.** *Two graphs $G_1, G_2$ are called* homeomorphic *if vertices of degree 2 can be added to one or both so that the resulting graphs are isomorphic.*

Roughly speaking, this means 'adding blobs' on one or more egdes so that the two graphs correspond. Adding a single vertex of degree 2 will split one edge into two, so adding several will generate more edges.

To compare $G_1$ and $G_2$, one could also 'strip' them of vertices of degree 2 one by one provided the result is still a graph. However, this stripping operation can lead to graphs that are not simple, so the notion of isomorphism is a little more complicated. If $G_1$ and $G_2$ are homeomorphic then their degree sequences can only differ by their numbers of entries that are '2'. This is important when one is looking for subgraphs that are homeomorphic to a specific graph like $K_5$, which has vertex sequence $(4, 4, 4, 4, 4)$.

**Example 6.5.** Let $G$ be a graph with degree sequence $(3, 3, 3, 3, 6, 6)$. Draw one that is simple. Any graph homeomorphic to $G$ must have degree sequence

$$(2, \ldots, 2, 3, 3, 3, 3, 6, 6),$$

with zero or more '2's. If there are no '2's then the graph will be isomorphic to $G$, which is a special case of homeomorphism.

Homeomorphism defines an equivalence relation on graphs in which one disregards vertices of degree 2. In particular, all cycle graphs $C_n$ with $n \geqslant 1$ are homeomorphic! But a 1-cycle (a single vertex with a loop) cannot have its vertex removed, because the result is not a graph!

Adding or removing vertices of degree 2 can be regarded as a 'trivial' operation that does not affect the essence of the graph. It is a topological notion. What we are really doing is concentrating on the set of points formed by the edges only (this set forms a 'topological space'), pretending they are made of stretchable wire. We are only interested in whether one set can be transformed into another by bending and stretching/compressing.

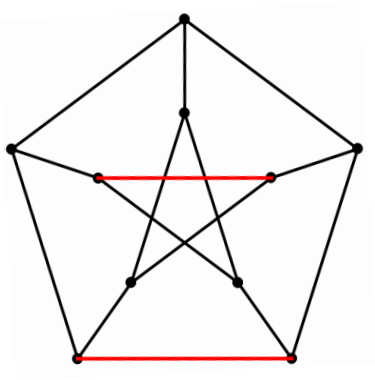
**Theorem 6.6** (Kuratowski, 1930)**.** *A graph is planar if and only if it does not contain a subgraph homeomorphic to $K_5$ or $K_{3,3}$.*

Expressed another way,

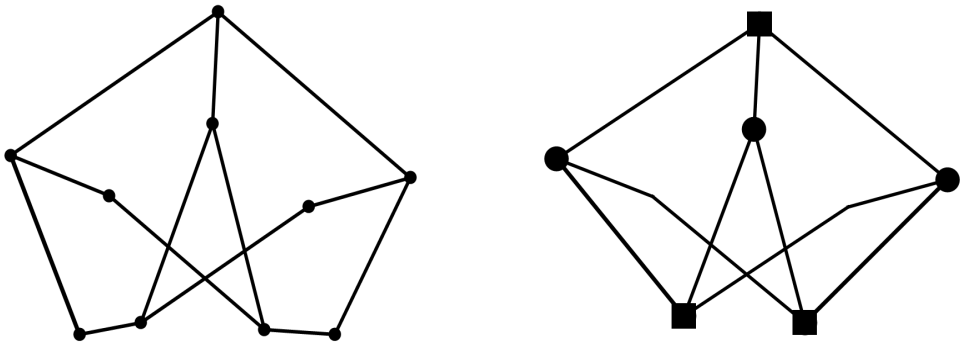
$$\text{non-planar} \iff \exists \text{ subgraph homeomorphic to } K_5 \text{ or } K_{3,3},$$

though (as remarked above) the implication $\Leftarrow$ is obvious. One can think of $K_5$ and $K_{3,3}$ as 'germs', one of which will *always* be present if the original graph is non-planar.

**Example 6.7.** Recall the Petersen graph $P$, which has 10 vertices and 15 edges. One guesses correctly that it is not planar. It is a regular graph with vertex degree 3, so there is no hope of finding a $K_5$ inside, but it does contain a subgraph homeomorphic to $K_{3,3}$. One needs to remove the two edges that are horizontal in the image below, leaving four vertices of degree 2. Note that the edges of a *subgraph* must be edges of $P$, so in order to talk of a subgraph we must leave the vertices in place, since they lie on other edges. But we can remove them and unite the edges so as to form a new graph homeomorphic to the subgraph:



The new graph has 6 vertices and 9 edges, since we removed 2 edges, and another four pairs of edges became four single ones. It is easy to see that the 6 vertices are partitioned into two groups of 3, with all possible edges going from one group to the other, so we are dealing with $K_{3,3}$.



This is all very well, but the operation we have performed is not very natural. Staring at $P$, it seems much closer to $K_5$ than $K_{3,3}$, and the next approach makes this precise.

**Definition 6.8.** *If $G$ is a graph, and $uv$ is an edge joining vertices $u$ and $v$, then the graph obtained from $G$ by* contracting *$uv$, written $G/uv$, is formed by making $u$ and $v$ coalesce so that any edges that arrived at either of them now arrive at the new common vertex. In this process, any loops are eliminated and any multiple edge just becomes a single one.*

A simple example would be a triangle with 3 vertices (i.e. a 3-cycle). If we contract any edge, we simply get a single edge with its two ends as vertices. Notice that the loop and extra edge are suppressed.

If we contract an edge in a plane diagram, it remains a plane diagram. One can contract a number of edges by doing one at a time. Contraction is a rough analogue of taking a *quotient* in other branches of mathematics.

Five contractions convert the Petersen graph $P$ to the complete graph $K_5$. In its pentagonal representation above, we contract the 5 spokes by joining each outer vertex to its nearest neighbour inside. This is a painless operation that does not even produce multiple edges to combine.

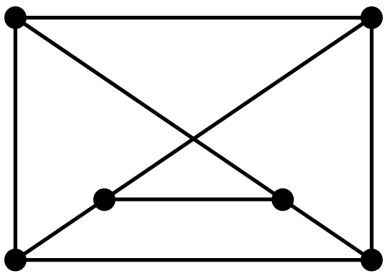
**Theorem 6.9** (Wagner, 1937)**.** *A graph is planar if and only if it does not contain a subgraph that can be contracted to (a graph isomorphic to)* $K_5$ *or* $K_{3,3}$*.*

Expressed another way,

$$\text{non-planar} \iff \exists \text{ subgraph contractible to } K_5 \text{ or } K_{3,3}.$$

But this time, neither of the implications is elementary. The forward direction ($\Rightarrow$) follows immediately from Kuratowski's theorem, because any subgraph *homeomorphic* to $K_5$ (resp. $K_{3,3}$) can itself be contracted to $K_5$ (resp. $K_{3,3}$) by contracting edges so as to delete the superfluous vertices of degree 2. But there are complications the other way – if $G$ is contractible to $K_5$ then it might not contain a subgraph homeomorphic to $K_5$, but one homeomorphic to $K_{3,3}$ instead.

**Exercise 6.10.** *Identify three edges of* $P$ *whose contraction yields* $K_{3,3}$*. It might help to use the following representation of* $K_{3,3}$*:*



## 6.3 Further results

Recall Euler's formula for a plane graph drawing:

$$n - e + f = 2.$$

We gave an informal proof in §5.3 by showing that the left-hand side does not change when the graph is 'grown' by adding one edge at a time. A more rigorous proof can be given by induction on the number $e$ of edges. We shall illustrate this for the special case of trees.

Recall that a *tree* is a *connected* graph with *no cycles*.

**Proposition 6.11.** *If* $G$ *is a tree then* $e = n - 1$*.*

*Proof.* If $e = 1$ then $n = 2$, the result is valid. Assume the result is true for $e = N - 1$. Let $G$ be a graph with $N$ edges. Now any edge $uv$ of a tree is a *bridge* – its removal disconnects the graph (because if not, there must be a path from $u$ to $v$ which becomes a cycle with $uv$). So if an edge is removed we obtain a graph with exactly two components (no more than two, because a single edge cannot connect three components). Both of the components must be trees, by definition. By assumption,

$$e = 1 + e_1 + e_2 = 1 + (n_1 - 1) + (n_2 - 1) = n - 1,$$

as stated. □

**Remark 6.12.** (i) We can use a similar induction argument to prove that any tree has a plane diagram (i.e. without crossings) with just an outside face. Thus $f = 1$, and the proposition is compatible with Euler's formula.
(ii) For a fixed number of vertices $n$, no connected graph can have less than $e - 1$ edges (exercise). Using this one can show that if $G$ is connected then $e = n - 1$ if *and only if* $G$ is a tree.

**Proposition 6.13.** *If $G$ is a simple planar graph with $e \geqslant 3$ then $e \leqslant 3n - 6$.*

*Proof.* Represent $G$ by a plane diagram. Each face (even if there is only one) borders at least 3 edges. Each such edge will be counted twice if it has different faces either side of it, otherwise counted once. Thus

$$2e \geqslant 3f = 3(2 + e - n),$$

which gives the result. □

**Lemma 6.14.** *Any simple planar graph has a vertex of degree at most 5, and is 6-colourable.*

*Proof.* Recall that the sum of the vertex degrees equals $2e$. If all the degrees are at least 6, then

$$6n \leqslant \sum_{v \in V} d(v) = 2e \leqslant 6n - 12,$$

a contradiction. □

We prove that $\chi \leqslant 6$ by induction on the number of vertices, $n$. Obviously $\chi \leqslant n$, so the result is true when $n \leqslant 6$. Suppose it is true for $n = N - 1$. If $G$ now has $N$ vertices, remove one (call it $v$) of degree at most 5 and its associated edges (at most 5 of them). By assumption, the remaining graph is 6-colourable. Moreover, $v$ only had 5 neighbouring vertices, so when we replace $v$ and its edges we still have a 6th colour left for $v$. □

The lemma is analogous to saying that $\chi \leqslant \Delta + 1$, and it is not too hard to refine the proof to conclude that $\chi \leqslant 5$. This is effectively the five colour theorem, whose equivalent statement for maps was proved by Heawood in 1890. The four colour theorem was not resolved until almost a century later, but not without a 'proof' that involved substantial computer verification of special cases:

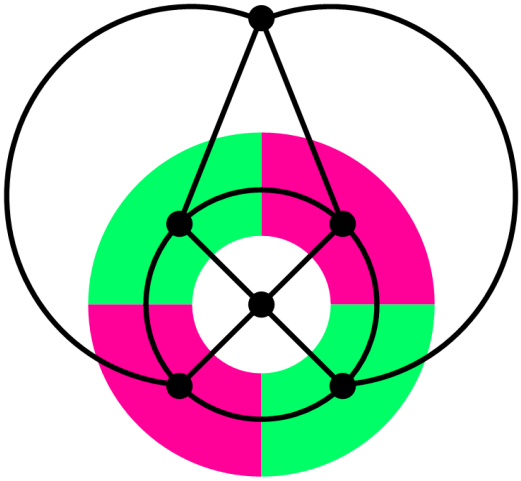
**Theorem 6.15** (Appel-Haken, 1976)**.** *Any simple* plane *graph is 4-colourable, i.e. $\chi \leqslant 4$.*

## 6.4 The four-colour theorem*

The last theorem is more familiar as a statement about *maps* – only four colours are needed in such a way that contiguous countries are distinguished.

Here is the idea. A *map* can be defined as a plane graph drawing for which removing one or two edges will not disconnect the graph. (This excludes vertices of degree 2 and an outside face reaching both sides of an edge.) Given a map $M$, we can form its 'dual', which is a plane graph diagram denoted $M^*$, which has a vertex for each face of $M$ and an edge joining two vertices whenever the corresponding faces are contiguous (and this edge crosses only that

common border). Because of our assumptions, $M^*$ will have no loops or multiple edges. Then a vertex colouring of $M^*$ corresponds to a valid colouring of the map, so the theorem implies the map colouring result.



The concept of duality is well known in the context of polyhedra – as we remarked, the cube (6 faces, 8 vertices) and octahedron (8 faces, 6 vertices) are dual pairs, as are the dodecahedron (12 faces, 20 vertices) and icosahedron (20 faces, 12 vertices). The dual of a tetrehedron is another tetrahedron (the graph being $K_4$ with 4 faces, 4 vertices).

We conclude with some comments that link this topic to *Geometry of Surfaces*. Planar graph diagrams can be regarded as graphs on the surface of a sphere in which one point of the sphere (say the north pole $p$) corresponds to infinity in the plane. The outside face in the plane becomes a normal face containing $p$ on the sphere, so this is more natural.

One can show that both $K_5$ and $K_{3,3}$ can be drawn on the torus without artificial crossings. One can also try to draw graphs on more complicated surfaces, in particular on a surface of *genus* $g$, which means a 'torus with $g$ holes'. Provided there are no crossings, it is well known that if $f$ now counts 'faces' on the surface, then

$$n - e + f = 2 - 2g,$$

this quantity being the so-called *Euler characteristic* of the surface. A graph that can be drawn on a surface of genus $g$ but not on one of genus $g-1$ is called a *graph of genus* $g$. Thus $K_4$ is a graph of genus 0, and $K_5$ and $K_{3,3}$ are graphs of genus 1.

It is also known that a map drawn on a surface of genus $g$ can always be coloured with a maximum of $k$ colours, where

$$k = \lfloor \frac{7 + \sqrt{1 + 48g}}{2} \rfloor.$$

Taking $g = 0$ gives the four colour theorem as a special case. Taking $g = 1$ shows that 7 colours suffice to colour the vertices of a graph inscribed on a doughnut.

# 7 Navigation in graphs

## 7.1 Adjacency data

There are two common ways of representing graphs non-pictorially – by means of an *adjacency matrix* or an *adjacency table*. We consider these in turn.

Without assuming that it is simple,

**Definition 7.1.** *Let $G = (V, E)$ be a graph $G$, where $V = \{v_1, \ldots, v_n\}$ is the set of vertices, and $E$ the set of edges. Let us label the vertices $v_1 \ldots, v_n$. The* adjacency matrix *of the graph $G$ is the matrix $A = (a_{ij})_{1 \leqslant i,j \leqslant n}$, where $a_{ij}$ is the number of edges joining $v_i$ to $v_j$.*

The adjacency matrix $A$ of a graph $G = (V, E)$, with $|V| = n$ is an $n \times n$ symmetric matrix. One can reconstruct $G$ from $A$.

**Example 7.2.**

$$A = \begin{pmatrix} 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 2 \\ 0 & 1 & 0 & 1 \\ 1 & 2 & 1 & 0 \end{pmatrix}$$

We can work out the degree of any vertex by taking the sum of the entries in the corresponding row (or column, since $A$ is symmetric). This gives the following lemma.

**Lemma 7.3.** *Let $G = (V, E)$ be a graph $G$, where $V = \{v_1, \ldots, v_n\}$ is the set of vertices, and $E$ the set of edges. Let $A$ be the adjacency matrix of $G$. Then, we have*

$$d(v_i) = 2a_{ii} + \sum_{\substack{1 \leqslant j \leqslant n \\ i \neq j}} a_{ij}. \tag{2}$$

*The graph is simple if and only if $a_{ii} = 0$ for all $i$, and every other entry is 0 or 1.*

**Example 7.4.** Returning to Example 7.2, we see that

$$\begin{aligned} d(v_1) &= 2 \\ d(v_2) &= 4 \\ d(v_3) &= 2 \\ d(v_4) &= 4. \end{aligned}$$

**Remark 7.5.** Our convention in Definition 7.1 is that a loop around a vertex is counted *once*, thus our formula (2) for the degree of a vertex. Another common approach is to assume that a loop is counted *twice*. In that case, the formula for the degree of a vertex would be

$$d(v_i) = \sum_{1 \leqslant j \leqslant n} a_{ij}. \tag{3}$$

**Definition 7.6.** *Let $G = (V, E)$ be a graph $G$, where $V = \{v_1, \ldots, v_n\}$ is the set of vertices, and $E$ the set of edges. Let $A$ be the adjacency matrix of $G$. Then, the* Laplacian *(or* Riskoff matrix*) of $G$ is the matrix $L$ given by $L = D - A$, where $D$ is the diagonal matrix*

$$D = \begin{pmatrix} d(v_1) & 0 & \ldots & 0 \\ 0 & d(v_2) & & 0 \\ \vdots & & \ddots & \vdots \\ 0 & \ldots & 0 & d(v_n) \end{pmatrix}$$

Many properties of the graph $G$ can be determined by studying its adjacency matrix. For example, we have the following theorem.

**Theorem 7.7.** *Let $G = (V, E)$ be a graph $G$, where $V = \{v_1, \ldots, v_n\}$ is the set of vertices, and $E$ the set of edges. Let $A$ be the adjacency matrix of $G$, and $L$ its Laplacian. Let $c(G)$ be the number of connected components of $G$. Then, we have*

$$c(G) = n - \operatorname{rank}(L) = \dim(\ker(L)).$$

**Example 7.8.** Return to Example 7.4, we obtain that the Laplacian of the graph is

$$L = \begin{pmatrix} 2 & -1 & 0 & -1 \\ -1 & 4 & -1 & -2 \\ 0 & -1 & 2 & -1 \\ -1 & -2 & -1 & 4 \end{pmatrix}$$

The echelon form of this matrix is given by

$$L' = \begin{pmatrix} 1 & 0 & 0 & -1 \\ 0 & 1 & 0 & -1 \\ 0 & 0 & 1 & -1 \\ 0 & 0 & 0 & 0 \end{pmatrix}.$$

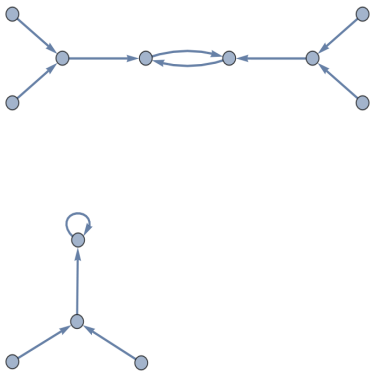
From this, we see that $\operatorname{rank}(L) = \operatorname{rank}(L') = 3$. Hence, $c(G) = 4 - 3 = 1$. This is unsurprising since by inspection we can see that $G$ is indeed connected.

Two advantages of the matrix approach are that it can be generalized:

- to deal with digraphs by relaxing the condition that $a_{ij} = a_{ji}$, so $a_{ij} = 1$ means that $i \to j$ is a directed edge (with a loop now counting 1), see the next example;
- to deal with *simple* weighted graphs or digraphs, in which each edge is assigned a non-negative number. The lower triangular part of the matrix then resembles a table of distances between cities of the type that used to be common in motoring atlases, except that only *adjacent* nodes have non-zero entries.

**Example 7.9.** Below is the sparse adjacency matrix of the disconnected digraph that represents squaring modulo 13, with vertices representing the 12 non-zero residue classes:

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



The smaller component consists of the residue classes 1 (with the loop), $12 = -1$, 5 and $8 = -5$ (the last two square to $-1$ mod 13). If we re-label the vertices so that $1, 5, 21, 26$ come at the start then the matrix will have a block form, making it obvious that there are (at least) two components.

Another method of representing an ordinary graph is by its adjacency table.

**Definition 7.10.** *Let $G = (V, E)$ be a graph $G$, where $V$ is the set of vertices, and $E$ the set of edges. The* adjacency table *$T$ of $G$ is the table*

$$T = \left[\begin{array}{c} v \\ \hline T_v \end{array}\right]_{v\in V},$$

*where $T_v$ is the set of vertices adjacent to $v$.*

We will also work with the transpose of this table when it is more convenient:

$$\left[\ v \ \middle|\ T_v\ \right]_{v\in V},$$

**Example 7.11.** The adjacency table of the graph $G$ in Example 7.2 is given by

| 1 | 2 | 3 | 4 |
|---|---|---|---|
| 2 | 1 | 2 | 1 |
| 4 | 3 | 4 | 2 |
| | 4 | | 3 |

In row representation, we can write this as:

$$\begin{array}{ll} 1: & [2,4] \\ 2: & [1,3,4] \\ 3: & [2,4] \\ 4: & [1,2,3] \end{array}$$

(This is how the adjacency table is given in `Sage`.)

One can easily adapt the matrix methods, for example to detect the existence of cycles

$$1 \to 2 \rightleftarrows 4 \to 1, \qquad 1 \to 2 \to 3 \to 4 \to 1$$

by passing from column to column.

## 7.2 Search trees

We discuss two different methods of searching through the vertices of graphs. These are

- *Depth first search* (DFS): In a *depth first search*, one follows paths as far as possible: From a vertex $v$ already reached, we proceed to any vertex $w$ adjacent to $v$ which was not yet visited; then we go on directly from $w$ to another vertex not yet reached etc., as long as this is possible. (If we cannot go on, we backtrack just as much as necessary.) In this way, one constructs *maximal* paths starting at some initial vertex $s$.
- *Breadth first seach* (BFS): In a *breadth first search*, vertices which have larger distance to the starting vertex $s$ are examined later than those with smaller distance to $s$.

Let $G = (V, E)$ be a graph. We define the following functions:

(a) For each $w \in V$, $p(w)$ denotes the vertex from which $w$ has been accessed.

(b) For each $w \in V$, $\mathrm{nr}(w)$ denotes the numbering of $w$ in the order in which it has been reached.

(c) For each $e \in E$, $u(e) = \texttt{False}$ if the edge $e$ has not been used to connect two vertices that have already been visited, and $u(e) = \texttt{True}$ otherwise.

**Algorithm 7.12** (Depth First Search (DFS)). *Let $G = (V, E)$ be a graph, and $s \in V$ a vertex.*

*(1) For each $v \in V$, initialize $p(v) = 0$ and $\mathrm{nr}(v) = 0$;*

*(2) For each $e \in E$, initialize $u(e) =$* `False`*;*

*(3) Set $i \leftarrow 1$; $v \leftarrow s$; $\mathrm{nr}(v) \leftarrow 1$;* `//The first vertex visited is s.`

*(4) Choose a vertex $w \in T_v$, and set $u(e) =$* `True`*, where $e = \{v, w\}$:*

*(a) If $\mathrm{nr}(w) = 0$, then set*

$$p(w) \leftarrow v;\ i \leftarrow i + 1;\ \mathrm{nr}(w) \leftarrow i;\ v \leftarrow w.$$

*(b) If $\mathrm{nr}(w) \neq 0$, then return to Step (4);*

*(5) Set $v \leftarrow p(v)$, and return to Step (4).*

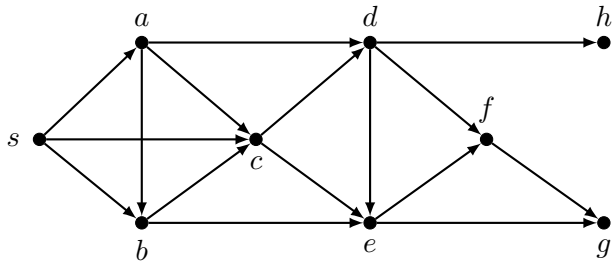
The DFS algorithm can be thought of as a walk in a maze. We illustrate this with an ucoming example.

**Theorem 7.13.** *Let $G = (V, E)$ be a graph.*

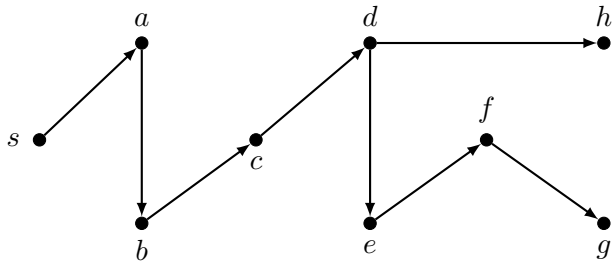
*(a) For $s \in V$, each edge in the connected component of $s$ is used exactly once in each direction during the execution of Algorithm 7.12.*

*(b) If $G$ is connected, then Algorithm 7.12 has complexity $O(|E|)$.*

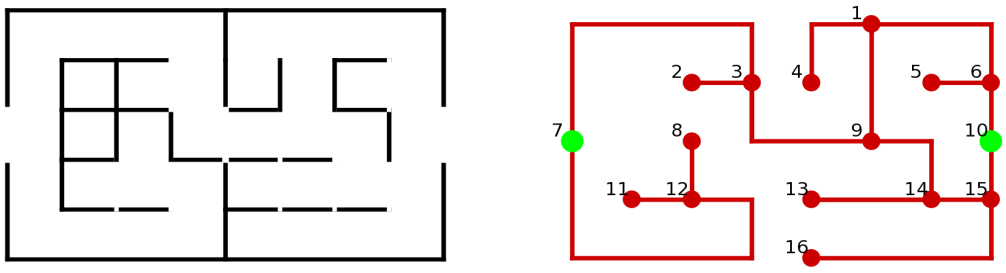
**Example 7.14.** Let us consider the following graph.




We perform a DFS search beginning at $s$. To make the algorithm deterministic, we visit the vertices in Step (4) according to their alphabetical order. This means that the vertices are reached in the following order: $s, a, b, c, d, e, f, g$. After that, the algorithm backtracks from $g$ to $f$, to $e$, and then to $d$. Now $h$ is reached and the algorithm backtracks to $d$, $c$, $b$, $a$, and finally to $s$. The directed tree $\mathcal{T}$ constructed by the DFS is given below:




**Example 7.15.** We can convert a maze (left) into a graph (right) by placing a vertex at each position where a choice is needed (including the start and end) and at a dead-end. A walk in this graph is a walk in the maze.



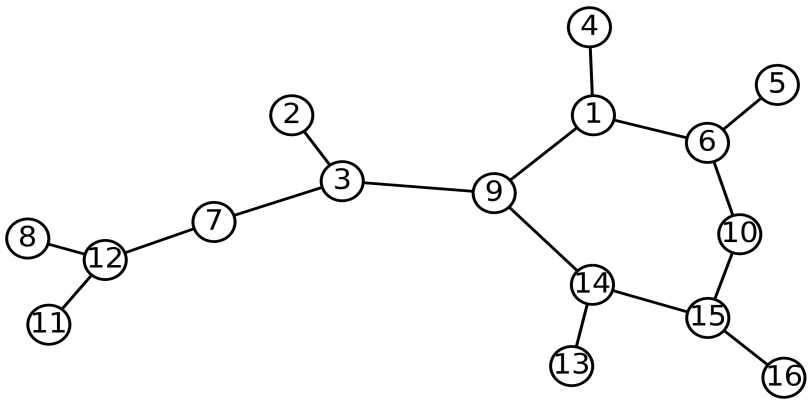

This graph can be input into a computer by typing its adjacency table. In SAGE, this is

```
T  =  { 1:  [4,6],
        2:  [3],
        3:  [2,7,9],
        4:  [1],
        5:  [6],
        6:  [1,5,10],
        7:  [3,12],
        8:  [12],
        9:  [1,3,14],
       10:  [6,15],
       11:  [12],
       12:  [7,8,11],
       13:  [14],
       14:  [9,13,15],
       15:  [10,14,16],
       16:  [15] }
```

We create the graph using `Sage` using the following script:

```
T = { 8: [12], 11: [12], 12: [7,8,11], 1: [4,6], 2: [3],
         3: [2,7,9], 4: [1], 5: [6], 6: [1,5,10], 10: [6,15],
         7: [3,12], 9: [1,3,14], 13: [14], 14: [9,13,15],
         15: [10,14,16], 16: [15] };
G = Graph(T);
H = G.plot(vertex_colors='white');
H.show(figsize=[4, 6], dpi=700)
```

This gives the graph




We perform a DFS search starting at $s = 7$. In Step (4), we visit the vertices in numerical order. We proceed in the following steps:

1. We visit the vertices $3, 2$ in that order; then we backtrack to $3$.
2. We visit the vertices $3$, $9$, $1$ and $4$. Then, we backtrack to $1$, then we go to $6$ and $5$.

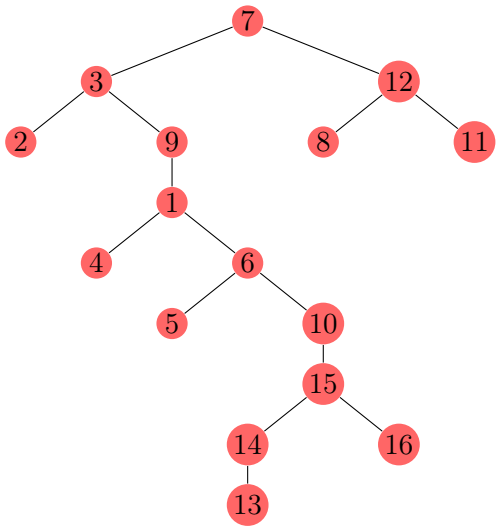



Figure 1: DFS search tree for Example 7.15

3. We visit the vertices 10, 15, 14 and 13. (Note, from 14 we cannot visit 9, otherwise we will have a loop, which is *not* permitted.) Then, we backtrack to 14, then to 15.
4. We visit the vertex 16. Then, we backtrack to 15, then to 10, 6, 1, 9, 3 and 7.
5. We visit 12 and 8. Then, we backtrack to 12.
6. We visit 11, and we backtrack to 12 and 7.

This gives the search tree in Figure 1.

One way to carry out Algorithm 7.12 is by processing the data from the adjacency table in a **stack**. At each stage, the vertex being processed is the one on the right. This represents the 'top' of the stack, which we are allowed to 'peek'. We seek to add or 'push' an *adjacent* vertex to the stack if there exists an adjacent vertex that has not already been processed.

We implement the DFS stack approach for Example 7.15 in Table 1. We add at most one adjacent vertex to the stack (for definiteness, the smallest in our list), and we keep a separate record of those vertices that have (at some point) entered the stack. If the vertex being processed has no new neighbours (either because it has degree one, or because its neighbours are in the stack), we remove or 'pop' it from the stack. Each vertex can appear at most once in our stack, and each vertex appears twice in our table, once added, once removed. This means that the table must contain $2n$ rows, where $n$ is the number of vertices (here $n = 16$).

*DFS abbreviated version.* Use of a table is advised to gain confidence in the method, but for some purposes it suffices to list the vertices in the order in which they are encountered, and add 'ties' joining neighbouring vertices that are not already adjacent in the list:

$$\overbrace{7,\ \overbrace{3,\ 2,\ 9},\ \overbrace{1,\ 4},\ \overbrace{6,\ 5,\ 10},\ \overbrace{15,\ 14,\ 13,\ 16},\ \overbrace{12,\ 8,\ 11}}.$$

| DFS stack ⇆ | added | removed |
|---|---|---|
| 7 | 7 | |
| 7, 3 | 3 | |
| 7, 3, 2 | 2 | |
| 7, 3 | | 2 |
| 7, 3, 9 | 9 | |
| 7, 3, 9, 1 | 1 | |
| 7, 3, 9, 1, 4 | 4 | |
| 7, 3, 9, 1 | | 4 |
| 7, 3, 9, 1, 6 | 6 | |
| 7, 3, 9, 1, 6, 5 | 5 | |
| 7, 3, 9, 1, 6 | | 5 |
| 7, 3, 9, 1, 6, 10 | 10 | |
| 7, 3, 9, 1, 6, 10, 15 | 15 | |
| 7, 3, 9, 1, 6, 10, 15, 14 | 14 | |
| 7, 3, 9, 1, 6, 10, 15, 14, 13 | 13 | |
| 7, 3, 9, 1, 6, 10, 15, 14 | | 13 |
| 7, 3, 9, 1, 6, 10, 15 | | 14 |
| 7, 3, 9, 1, 6, 10, 15, 16 | 16 | |
| 7, 3, 9, 1, 6, 10, 15 | | 16 |
| 7, 3, 9, 1, 6, 10 | | 15 |
| 7, 3, 9, 1, 6 | | 10 |
| 7, 3, 9, 1 | | 6 |
| 7, 3, 9 | | 1 |
| 7, 3 | | 9 |
| 7 | | 3 |
| 7, 12 | 12 | |
| 7, 12, 8 | 8 | |
| 7, 12 | | 8 |
| 7, 12, 11 | 11 | |
| 7, 12 | | 11 |
| 7 | | 12 |
| ∅ | | 7 |

Table 1: DFS stack for Example 7.15

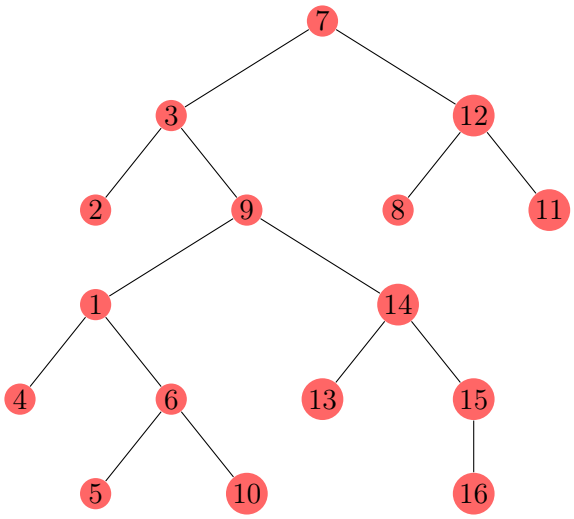



Figure 2: BFS search tree for Example 7.15

It is a consequence of the methods that all such ties are 'nested', with no crossings.

**Breadth first search.** We recall that a *queue* is a data structure in which items are added on the right, and removed from the left.

**Algorithm 7.16** (Breadth First Search (BFS)). *Let $G = (V, E)$ be a graph, and $s$ a vertex.*

*(1) Initialise the empty queue $Q = \varnothing$;*

*(2) Set $d(s) \leftarrow 0$;*

*(3) Append $s$ to $Q$;*

*(4) Remove the first vertex $v$ from $Q$;*

*(5) For each vertex $w \in T_v$, do:*

*If $d(v)$ is undefined, then set $d(w) \leftarrow d(v) + 1$; Append $w$ to $Q$.*

*(6) Return to Step (4).*

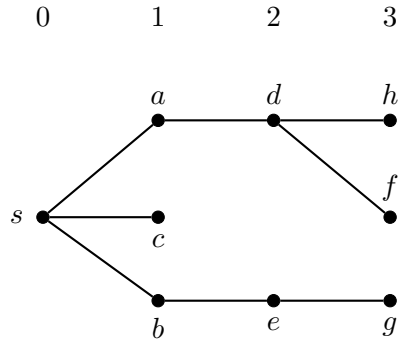
**Theorem 7.17.** *Algorithm 7.16 has complexity $O(|E|)$. At the end of the algorithm, every vertex $t$ of $G$ satisfies*

$$d(s,t) = \begin{cases} d(t), & \text{if } d(t) \text{ is defined;} \\ \infty, & \text{otherwise.} \end{cases}$$

**Corollary 7.18.** *Let $s$ be a vertex of a graph $G$. Then $G$ is connected if and only if $d(t)$ is defined for each vertex $t$ at the end of* BFS *search starting at $s$.*

**Example 7.19.** Let us consider the graph $G$ in Example 7.14. We a BFS search on this graph starting at the vertex $s$. To make the algorithm deterministic, we select the vertices in

alphabetical order in Step (5) of the BFS. To make things clearer, we have drawn the vertices in levels according to their distance to $s$; also, we have omitted all edges leading to vertices already labelled. Thus all we see of |G| is a spanning tree, that is, a spanning subgraph of G which is a tree. This kind of tree will be studied more closely in Chap. 4.




This is carried out by inserting data into a **queue**. When processing a vertex in a BFS, it is important to add *all* its adjacent vertices before moving on. This is exactly what we did in the 2-part algorithm to re-order vertices prior to colouring them. Although our table should show each addition and removal step by step, we can save time by using one row for each vertex being processed (having just been removed). This way, we only have $n$ rows, excluding the first:

| removed | ← BFS queue ← |
|---|---|
| | 7 |
| 7 | 3, 12 |
| 3 | 12, 2, 9 |
| 12 | 2, 9, 8, 11 |
| 2 | 9, 8, 11 |
| 9 | 8, 11, 1, 14 |
| 8 | 11, 1, 14 |
| 11 | 1, 14 |
| 1 | 14, 4, 6 |
| 14 | 4, 6, 13, 15 |
| 4 | 6, 13, 15 |
| 6 | 13, 15, 5, 10 |
| 13 | 15, 5, 10 |
| 15 | 5, 10, 16 |
| 5 | 10, 16 |
| 10 | 16 |
| 16 | ∅ |

*BFS abbreviated version.* As in the 2-part colouring algorithm, some of the information can be presented by a single long list:

| | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| vertices: | 7 | 3 | 12 | 2 | 9 | 8 | 11 | 1 | 14 | 4 | 6 | 13 | 15 | 5 | 10 | 16 |
| level: | 0 | 1 | 1 | 2 | 2 | 2 | 2 | 3 | 3 | 4 | 4 | 4 | 4 | 5 | 5 | 5 |

Here we have processed up to and including vertex 14, in the latter case by adding $13, 15$. It is a consequence of the method that vertices are added level by level, and the last row indicates their 'distance' to the start.

**Definition 7.20.** *Given a connected graph $G$, a* spanning tree *is a subgraph of $G$ without cycles that includes all the vertices of $G$.*

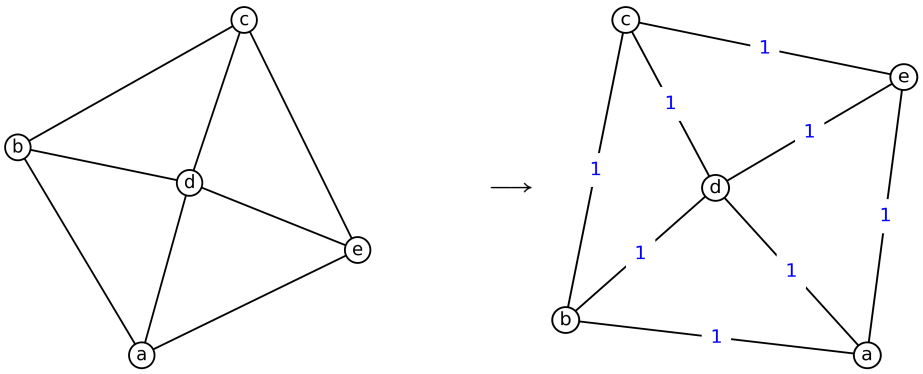
Both searches determine such a spanning tree, although the way they do this reflects their names. The trees are *rooted*, because we have distinguished a starting point – vertex 7 is the root. We can now re-draw the tree growing (by convention) downwards, level by level. The 'dead-end' vertices $2, 4, 5, 8, 11, 13, 16$ all define *leaves* of the trees, but the BFS tree (right) happens to have an extra leaf at the finish.

In our example, $G$ was not itself a tree, having a cycle $(9, 1, 6, 10, 15, 14, 9)$ and the two trees break this in different ways. The DFS tree omits the edge $(14, 9)$ whereas the BFS one omits $(10, 15)$. The DFS tree (left) has *height* $8$, this being the maximum number of edges from root to leaf, achieved by arriving at the dead-end $13$. By contrast, the BFS tree is more spread out and has height $5$.

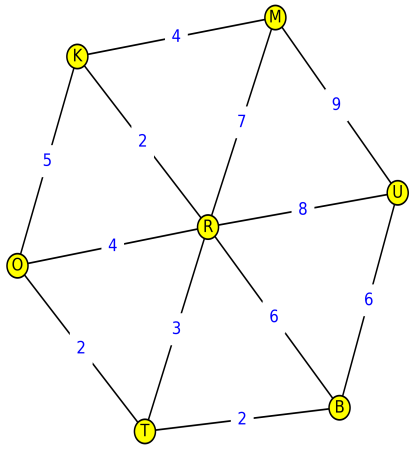
## 7.3 Shortest paths

**Definition 7.21.** *A* weighted graph *is a simple graph $G = G(V, E)$ together with a function $d : E \longrightarrow (0, \infty)$ called* weight function or map*, which take positive integer values. (We use $d$ for "distance" because $w$ could be confused with a vertex.)*

**Example 7.22.** Let $G = (V, E)$ be a simple graph. We can turn $G$ into a weighted graph by setting $d(e) = 1$ for all $e \in E$.




**Example 7.23.** In the graph below, the vertices can be thought of as representing cities. The weight can be thought of as the time spent on the train between each two cities connected by an edge. The weight can also represent the distance in km.




**Definition 7.24.** *Let $G = (V, E)$ be a weighted graph. Let $W = e_1 \sqcup \ldots \sqcup e_s$ be a walk (or path) in $G$. The* length *of the $W$ is given by*

$$d(W) = \sum_{i=1}^{s} d(e_i).$$

*If $u, v$ are vertices, the* distance *between $u$ and $v$ is defined by by*

$$d(u,v) := \begin{cases} \min\{d(W) : W \in \mathrm{Paths}(u,v)\}, & \text{if } v \text{ is accessible from } u \\ \infty & \text{otherwise.} \end{cases}$$

We note that $d(u,u) = 0$.

**Applications:** Weighted graphs have many important applications. For example, here are two such applications.

1. *Train scheduling:* An example consists of points on a transport network with distances or travel times between nodes. The TfL map is arguably less practical – it displays walking times between stations – but is a good illustration.
2. *Projects scheduling:* If we want to carry out a complex project – such as, for example, building a dam, a shopping centre or an airplane – the various tasks ought to be well coordinated to avoid loss of time and money. This kind of tools are used in operations research.

In such applications, one would like to find the shortest distance between two vertices. In practice, this could represent the shortest distance or time between to cities. The smallest cost of a project, etc. The aim of this section is to present and justify Dijkstra's algorithm for finding shortest paths from some fixed root vertex to all the others in a weighted graph.

The BFS algorithm is a special case of the Dijkstra's algorithm; in that case, one simply sets the weight of each edge to be $d(e) = 1$ (see Example 7.22).

If $G = (V, E)$ is a weighted graph, and $u, v \in V$, we will denote the *shortest path* between $u$ and $v$ by $\mathrm{SP}(u, v)$.

The following is known as the Dijkstra algorithm. It is one of the most important algorithm is graph theory.

**Algorithm 7.25** (Dijkstra's algorithm)**.** *This takes as input: a weighted graph* $G(V,E)$ *and a distinguished vertex* $v_0$*. Let* $V=\{v_0,v_1,\dots,v_n\}$*. Its output consists of a list of the shortest distances* $L_i=\mathrm{SD}(v_0,v_i)$ *for each vertex* $v_i$*. Obviously* $L_0=0$*.*

*(1) Initialize* $V_{\text{temp}}\leftarrow V$*, and set* $L_0=0$*;*

*(2) For* $j>0$*, intialize* $L_j\leftarrow\infty$*;*

*(3) while* $V_{\text{temp}}\neq\varnothing$*:*

*Choose* $v_i\in V_{\text{temp}}$ *with* $L_i$ minimal*, and set* $V_{\text{temp}}\leftarrow V_{\text{temp}}\setminus\{v_i\}$*.*

*(4) For* $v_j\in V_{\text{temp}}$ adjacent *to* $v_i$*, set* $L_j\leftarrow\min(L_i+d(v_iv_j),L_j)$*;*

*(5) Return* $(L_0,L_1,\dots,L_n)$*;*

**Example 7.26.** Let's return to Example 7.23. We would like to find the shortest distance between $B$ and $M$, i.e. $\mathrm{SP}(B,M)$.

- Initial values: $L_B=0$, $L_v=\infty$, for $v\in\{K,M,O,R,T,U\}$, $V_{\text{temp}}=V$;
- Iteration $k=1$:

  $v=T$, $V_{\text{temp}}=\{K,M,O,R,U\}$, $L_T=2$;
  $L_O=4$ and $L_R=5$;
- Iteration $k=2$:

  $v=O$, $V_{\text{temp}}=\{K,M,R,U\}$, $L_O=4$;
  $L_R=5$ and $L_K=7$;
- Iteration $k=3$:

  $v=R$, $V_{\text{temp}}=\{K,M,U\}$, $L_R=5$;
  $L_K=7$, $L_M=11$ and $L_U=6$;
- Iteration $k=4$:

  $v=U$, $V_{\text{temp}}=\{K,M\}$, $L_U=6$;
  $L_M=11$;
- Iteration $k=5$:

  $v=M$, $V_{\text{temp}}=\{K\}$, $L_M=11$;
  $L_K=7$;
- Iteration $k=6$:

  $v=K$, $V_{\text{temp}}=\varnothing$, $L_K=7$;

So the algorithm returns the vector $(0,7,11,4,5,2,6)$.

# 8 Optimality

## 8.1 Shortest paths

The aim of this section is to prove that Dijkstra's algorithm is 'correct'. The next result is phrased using the notation of Subsection 7.3.

**Lemma 8.1.** *Suppose that a shortest path $v_0 \rightsquigarrow u \rightsquigarrow v_i$ between two vertices $v_0, v_i$ in a weighted graph passes via an intermediate vertex $u$. Then both subpaths $v_0 \rightsquigarrow u$ and $u \rightsquigarrow v_i$ are shortest paths between their respective vertices, and in particular*

$$\mathrm{SD}(v_0, v_i) = \mathrm{SD}(v_0, u) + \mathrm{SD}(u, v_i).$$

*Proof.* If one of the 'subpaths' is not shortest, then we could substitute it with a shorter one, giving a shorter path $v_0 \rightsquigarrow v_i$. It's that simple! □

For the lemma, we are assuming that the path between $v_0$ and $v_i$ is a shortest one. If this were not the case, we only have the triangle inequality

$$\mathrm{SD}(v_0, v_i) \leqslant \mathrm{SD}(v_0, u) + \mathrm{SD}(u, v_i).$$

The lemma is an instance of an important argument called 'Bellman's optimality principle' that crops up in different guises in many problems in optimization theory.

Let us return to Dijkstra's algorithm. We wish to prove that all the permanent labels of vertices in $V_{\mathrm{perm}}$ are correct shortest distances:

**Theorem 8.2.** *Algorithm 7.25 determines with complexity $O(|V|^2)$ the distances with respect to some vertex $s$ in $(G, d)$. More precisely, at the end of the algorithm*

$$\mathrm{SD}(v_0, v_i) = L_i, \text{ for all } i = 0, \ldots, n.$$

*Proof.* We shall prove this by induction on $i$. The statement is certainly true when $i = 0$ since $\mathrm{SD}(v_0, v_0) = L_0 = 0$. Let $i > 0$, and assume that

$$\mathrm{SD}(v_0, v_k) = L_k \text{ for all } 0 \leqslant k \leqslant i - 1.$$

Now let $v_i \in V_{\mathrm{temp}}$ denote the temporary vertex chosen at a later stage because its label $L_i$ is minimal. We want to show that

$$L_i = \mathrm{SD}(v_0, v_i).$$

Suppose that

$$v_0 \rightsquigarrow v_k \to v_m \rightsquigarrow v_i$$

is a shortest path from $v_0$ to $v_i$. Here we have chosen intermediate and *adjacent* vertices $v_k$, with $k < i$ and $v_m \in V_{\mathrm{temp}}$; this is clearly possible and it may be that $m = i$. Our inductive hypothesis is that $\mathrm{SD}(v_0, v_j) = L_j$ for all $0 \leqslant j \leqslant i - 1$, so in particular $L_k = \mathrm{SD}(v_0, v_k)$. Then,

we have

$$\begin{aligned} L_i &\leqslant L_m, \text{ by minimality} \\ &\leqslant L_k + d(v_k, v_m), \text{ by definition of } L_m \text{ when } v_k \text{ scanned } v_m \\ &\leqslant L_k + \mathrm{SD}(v_k, v_i), \text{ since the edge is part of the SP} \\ &= \mathrm{SD}(v_0, v_k) + \mathrm{SD}(v_k, v_i), \text{ by hypothesis} \\ &= \mathrm{SD}(v_0, v_i), \text{ by the previous lemma.} \end{aligned}$$

But $L_i$ is the distance to $v_i$ via *some* path, so it must *equal* $\mathrm{SD}(v_0, v_i)$, and (incidentally) all the inequalities are equalities.

This completes the induction. □

**Remark 8.3.** The crucial technique in Dijkstra's algorithm is the act of relabelling: once $v_i$ becomes permanent we scan its adjacent vertices and reduce their labels if passing through $v_i$ gives a shorter path:

$$L_j = \min(L_i + d(v_i, v_j),\ L_j).$$

The act of relabelling is called *relaxation* of the edge $v_i v_j$, and its repeated use allows one to decrease the estimated shortest distances until they become correct. Dijkstra's algorithm has the characteristic that it grows a tree of shortest paths from the root. There are other shortest path algorithms that apply relaxations in a more brute force (and thus, simpler) way, whilst still eventually achieving a shortest path.

## 8.2 Kruskal's algorithm

In this section, $G$ is always a *simple connected* weighted graph. Let $n$ denote the number of its vertices.

We have seen a number of algorithms that, when applied to $G$, construct a *spanning tree*. This is a subgraph with the same vertex set as $G$, and since it is a tree, it will have $n-1$ edges. In particular, Dijkstra's algorithm does this, provided we give it a starting vertex to act as root.

A different connectivity problem concerns the construction of a *minimal spanning tree* or MST.

**Definition 8.4.** *Let $(G, d)$ be a weighted graph. For any subset $E'$ of the set of edges $E$, we defined the* total weight *of $E'$ to be*

$$d(E') = \sum_{e \in E'} d(e).$$

**Definition 8.5.** *Let $(G, d)$ be a weighted simply connected graph. A* minimal spanning tree *or* MST *for $G$ is a tree $T$ whose total degree is minimal among those of all spanning trees.*

Any connected graph has a spanning tree, because whenever there is a cycle one can remove any edge in that cycle, leaving all the vertices connected, and continue until there are no more cycles. Therefore a *minimal* spanning tree must exist, and (if there is more than one) the total weight of any two are equal by definition.

**Algorithm 8.6** (Kruskal's algorithm)**.** *Let $G = (V, E)$ be a weighted simple connected graph, with $|V| = n$. We order the edges of $G$ according to their weight, so that $E = \{e_1, \ldots, e_m\}$ with $d(e_1) \leqslant \ldots \leqslant d(e_m)$.*

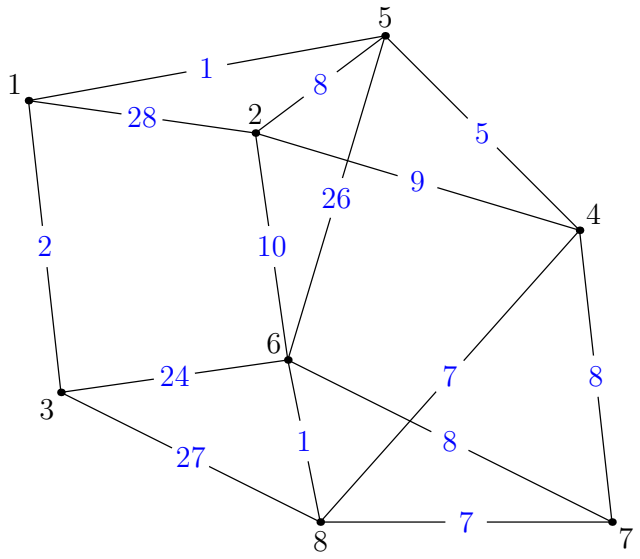
*(1) Initialize* $T \leftarrow \varnothing$*;*

*(2) For* $k = 1, \ldots, m$*, do:*

*if* $e_k$ *does not form a cycle together with some edges of* $T$*, then append* $e_k$ *to* $T$*;*

*(3) Return to Step (2);*

**Example 8.7.** Let us consider the graph below. We want to find a spanning tree.

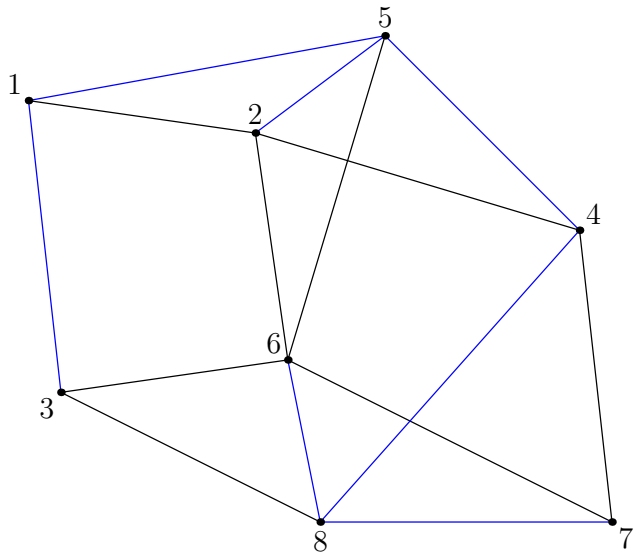



We label the edges first according to their weights, then to the numbering of their vertices. This gives the list:

$$e_1 = \{1,5\}, e_2 = \{6,8\}, e_3 = \{1,3\}, e_4 = \{4,5\}, e_5 = \{4,8\}, e_6 = \{7,8\},$$
$$e_7 = \{2,5\}, e_8 = \{4,7\}, e_9 = \{6,7\}, e_{10} = \{2,4\}, e_{11} = \{2,6\}, e_{12} = \{3,6\},$$
$$e_{13} = \{5,6\}, e_{14} = \{3,8\}, e_{15} = \{1,2\}.$$

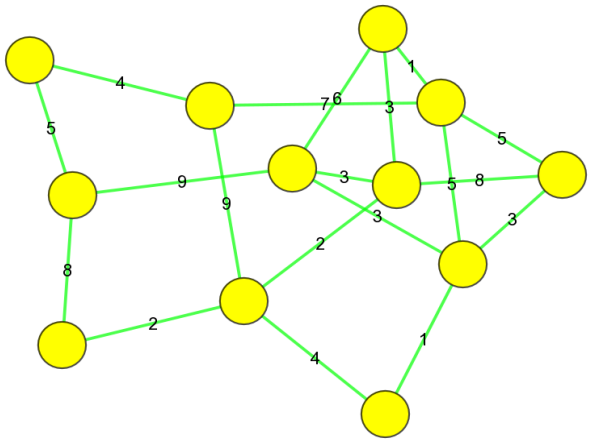
Now, we apply the Kruskal algorithm (Algorithm 8.6):

| Iteration $k$ | $T$ |
|---|---|
| 1 | $\{e_1\}$ |
| 2 | $\{e_1, e_2\}$ |
| 3 | $\{e_1, e_2, e_3\}$ |
| 4 | $\{e_1, e_2, e_3\}$ |
| 5 | $\{e_1, e_2, e_3, e_4\}$ |
| 6 | $\{e_1, e_2, e_3, e_4, e_5\}$ |
| 7 | $\{e_1, e_2, e_3, e_4, e_5, e_6\}$ |
| 8 | $\{e_1, e_2, e_3, e_4, e_5, e_6, e_7\}$ (After the 8th iteration, every additional edge creates a cycle) |

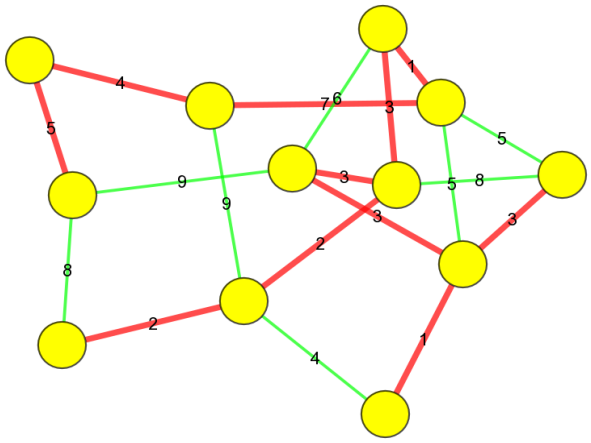



We see that $T$ is indeed a spanning tree for $G$. By Theorem 8.9, it is a minimal one.

**Example 8.8.** Applying Dijkstra's algorithm with root bottom left in the following graph gives a tree of weight 41. Applying Kruskal's algorithm gives a MST with weight 34.




The vertices are drawn with large circles to allow one to record the labels (temporary and permanent) in applying Dijkstra's algorithm




Kruskal's is a prototype 'greedy algorithm' since it executes what seems to be the optimal choice at each step. One could imagine (for example) that at each stage one should only add edges that are connected to ones already chosen, so that the MST is 'grown' branch by branch. (It turns out that this procedure is also valid – it is called Prim's algorithm and works well when the graph is defined by an adjacency matrix.) In any case, it is far from obvious that Kruskal's procedure works, but this is what the next result assures us:

**Theorem 8.9.** *Algorithm 8.6 outputs a spanning tree $T$ that is a minimal spanning tree.*

We need some lemmas before we can prove this theorem.

**Lemma 8.10.** *Let $G$ be a graph. Then, the following are equivalent:*

*(a) $G$ is a tree;*

*(b) $G$ does not have any cycles, but adding any further edge yields a* unique *cycle;*

*(c) Any two vertices of $G$ are connected by a unique path.*

*(d) $G$ is connected, and every edge of $G$ is a bridge.*

*Proof.* Give proof for first 3 items. □

**Remark 8.11.** Let $G = (V, E)$ be a graph and $T \subset G$ a tree. Let $e$ be an edge of $G$ *not* contained in $T$. Then, by Lemma 8.10, the graph $T \cup \{e\}$ contains a unique cycle. We will denote this cycle by $C_T(e)$.

*Proof of Theorem 8.9.* Note that at each intermediate stage, $F$ is a 'forest' consisting of one or more trees, and $|F| < n$. Since $|T| = n - 1$ it can have only one connected component, and must include all $n$ vertices. We need to prove that its total weight is *minimal* amongst all spanning trees.

Let $T'$ be a *minimal* spanning tree (one certainly exists!) such that $T' \cap T$ is as large as possible.

Let $e$ be an edge in $T \setminus T'$ of least weight. Let $C_{T'}(e)$ be the unique cycle given by Lemma 8.10 and Remark 8.11. The cycle $C_{T'}(e)$ must have an edge $e' \in T' \setminus T$. Otherwise, we would

have $C_{T'}(e) \subset T$. Again, by Lemma 8.10 and and Remark 8.11, there is unique cycle $C_T(e')$ contained in $T \cup \{e'\}$. The same argument as above implies that $C_T(e')$ has an edge $e'' \in T \setminus T'$.

Assume that $e = e_j$, $e' = e_k$ and $e'' = e_l$, with $1 \leqslant j, k, l \leqslant m$. By assumption on $e$, we have $d(e) \leqslant d(e'')$, hence $j \leqslant l$. We claim that $j \leqslant k$, meaning that $d(e) \leqslant d(e')$. Otherwise, we would have $k < j$, and $d(e') < d(e)$. So $e'$ would have been added to $F \subseteq T$ before $e$:

At the stage the algorithm is applied to any edges of weight $d(e')$, the edge $e''$ would not have been part of $F$ since $d(e') < d(e) \leqslant d(e'')$, nor could adding $e'$ have created a cycle in $F$, because in that case there would already be a path from $e'_+$ to $e'_-$ in $T$ preventing the later addition of $e''$. So $e'$ would have been added to $F \subseteq T$. But this is a contradiction.

Finally, consider

$$(T' \setminus \{e'\}) \sqcup \{e\};$$

this is an MST with a *larger* intersection with $T$, which is a contradiction. □

## 8.3 Back to matrices

The aim of this section is to link our study of spanning trees to some matrix algebra. We'll be using the prefix 'ADJ' for both the 'ADJacency' matrix and the 'ADJugate' (sometimes called 'ADJoint') matrix.

**Revision on matrix algebra.** Let $A = (a_{ij})$ be a square $n \times n$ matrix, and recall the notion of *cofactor*, used in the computation of inverses. Namely, define

$$c_{ij} = (-1)^{i+j}(\text{sub-determinant formed from } A \text{ by deleting row } i \text{ and col } j).$$

The transpose $C^\top$ is the so-called *adjugate* matrix $\widetilde{A} = \operatorname{adj} A$:

$$\widetilde{A}_{ij} = c_{ji},$$

and

$$A\widetilde{A} = (\det A)I = \widetilde{A}A.$$

Of course,

$$A = \begin{pmatrix} a & b \\ c & d \end{pmatrix} \implies \operatorname{adj} A = \begin{pmatrix} d & -b \\ -c & a \end{pmatrix}.$$

If $A$ is invertible then

$$A^{-1} = \frac{1}{\det A}\widetilde{A}.$$

However, $\widetilde{A}$ can still be useful when $A$ is not invertible; here's an enticing example:

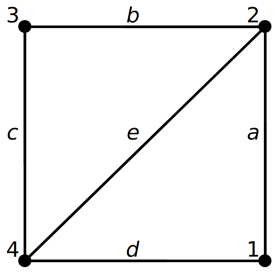
$$A = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix} \implies \widetilde{A} = -3\begin{pmatrix} 1 & -2 & 1 \\ -2 & 4 & -2 \\ 1 & -2 & 1 \end{pmatrix}.$$

Let $G$ be a simple graph with $n$ vertices. Consider the following $n \times n$ matrices:

$$\begin{aligned} A &= \text{the adjacency matrix of } G \\ D &= \text{the diagonal matrix of vertex degrees} \\ L &= D - A. \end{aligned}$$

$L$ is called the *Laplacian matrix* of th graph $G$. It is obviously symmetric, and all its rows (or columns) add up to zero. In fact, *the rank of $L$ equals $n$ minus the number of components of $G$.*

**Example 8.12.** Consider this graph with 4 vertices and 5 edges:




$$A = \begin{pmatrix} 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{pmatrix} \Longrightarrow L = \begin{pmatrix} 2 & -1 & 0 & -1 \\ -1 & 3 & -1 & -1 \\ 0 & -1 & 2 & -1 \\ -1 & -1 & -1 & 3 \end{pmatrix}.$$

**Exercise 8.13.** *Compute $A^2$ and check that its $(i,j)$th entry is the number of walks of length 2 from vertex $i$ to vertex $j$. Explain why, more generally, $(A^n)_{ij}$ is the number of walks of length $n$ from $i$ to $j$.*

## 8.4 The graph Laplacian*

Now we turn attention to the matrix $L$. Consider the characteristic polynomial

$$\det(L - xI) = (\lambda_1 - x)(\lambda_2 - x)\cdots(\lambda_4 - x).$$

Since $L$ is not invertible, at least one eigenvalue must vanish, say $\lambda_4 = 0$. Then

$$\det(L - xI) = x^4 - 10x^3 + (\lambda_1\lambda_2 + \lambda_2\lambda_3 + \lambda_3\lambda_1)x^2 - \lambda_1\lambda_2\lambda_3 x.$$

One the other hand, this equals

$$\begin{aligned} \det\begin{pmatrix} 2-x & -1 & 0 & -1 \\ -1 & 3-x & -1 & -1 \\ 0 & -1 & 2-x & -1 \\ -1 & -1 & -1 & 3-x \end{pmatrix} &= \det\begin{pmatrix} 2-x & -1 & 0 & -1 \\ -1 & 3-x & -1 & -1 \\ 0 & -1 & 2-x & -1 \\ -x & -x & -x & -x \end{pmatrix} \\ &= \det\begin{pmatrix} 2-x & -1 & 0 & -x \\ -1 & 3-x & -1 & -x \\ 0 & -1 & 2-x & -x \\ -x & -x & -x & -4x \end{pmatrix} \\ &= -4x\, c_{44} + O(x^2), \end{aligned}$$

where $O(x^2)$ gathers all the terms in $x^2, x^3, x^4$. (The first step above was to add the first three rows to the last one to give a row of $-x$'s, the second was to add the first three columns to the last one.) Therefore,

$$\lambda_1\lambda_2\lambda_3 = 4c_{44} = 32.$$

More to the point, by crossing out other rows/columns, we can see that *all* the cofactors of $L$ are are equal! The same argument gives

**Lemma 8.14.** *For a simple connected graph, all the cofactors of $L$ are equal (to $1/n$ times the product of its non-zero eigenvalues).*

**Theorem 8.15** (Kirchoff's matrix tree theorem)**.** *This number equals the number of spanning trees in the simple connected graph $G$.*

*Sketch of proof.* This is based on another matrix associated to a graph, its incidence matrix. Or rather, the incidence matrix $M$ associated to a digraph. First, we need to 'orient' the edges of $G$ arbitrarily, as in the picture on the previous page. Then rows of $M$ represent vertices, columns edges, and a 1 (resp. $-1$) in a column means that the edge leaves (resp. enters) the vertex associated to that row. With vertices labelled $1, 2, 3, 4$ and edges labelled $a, b, c, d, e$, this gives

$$M = \begin{pmatrix} 1 & 0 & 0 & -1 & 0 \\ -1 & 1 & 0 & 0 & -1 \\ 0 & -1 & 1 & 0 & 0 \\ 0 & 0 & -1 & 1 & 1 \end{pmatrix}.$$

It is easy to understand that

$$L = MM^\top.$$

The sum of the rows of $M$ is also zero. In general, for a connected graph, the rank of $M$ equals $n-1$, and (this is the key point) one can show that *a subset of $n-1$ edges forms a tree if and only if the determinant of that submatrix is non-zero.* We do not lose information by deleting any row of $M$, say the last, to define the *reduced incidence matrix* $R$.

The proof of Kirchoff's theorem is now a matter of computing

$$c_{nn}(L) = \det(RR^\top)$$

as a sum of products of sub-determinants of $R$. A generalization of the usual rule for $\det(AB)$ says how to do that. □

**Example 8.16.** The complete graph $K_3$ obviously has 3 spanning trees, and $K_4$ has $\binom{6}{3} - 4 = 16$. Using Kirchoff's theorem (and the trick of adding rows to simplify the cofactor calculation), one quickly obtains

**Corollary 8.17.** *The complete graph $K_n$ has $n^{n-2}$ spanning trees.*

Actually, we can forget about $K_n$, and $n^{n-2}$ counts *labelled trees with $n$ vertices.* This fact was known to Cayley, and Prüfer explained how such trees can be described by sequences of numbers $(a_1, \ldots, a_{n-2})$ with $a_i \in \{1, \ldots, n\}$.

# 9 Networks and flows

In this section, we discuss flows in *networks*: How much can be transported in a network from a source $s$ to a sink $t$ if the capacities of the connections are given? Such a network might model a system of pipelines, a water supply system, or a system of roads. The theory of flows has many applications, and is one of the most important parts of combinatorial optimization.

## 9.1 Network flow

Throughout this section, we work with directed graph. We recall some of the terminology that will be needed.

Let $G = (V, E)$ is a directed graph. For an edge $e \in E$, we denote the *start*, *source* or *origin* of $e$ by $e_-$, and the *end*, *sink* or *target* of $e$ by $e_+$. We say that $W \subset E$ is a *path* in $G$ if $W$ an undirected path is a path in the undirected graph $G$. In that case, we can write

$$W = v_0 \rightleftharpoons v_1 \rightleftharpoons v_2 \rightleftharpoons \cdots \rightleftharpoons v_k$$

We say that $e_i = v_i \rightleftharpoons v_{i+1}$ is a *forward edge* if $e_i = v_i \to v_{i+1}$, and a *backward edge* if $e_i = v_i \leftarrow v_{i+1}$. We say that $W$ is a *directed path* or an *aligned path* if each edge $e_i$ is a forward edge, so that

$$W = v_0 \to v_1 \to v_2 \to \cdots \to v_k.$$

**Definition 9.1.** *Let $G = (V, E)$ be a directed graph. A* capacity *on $G$ is a map $c : E \to \mathbb{R}_{\geqslant 0}$. The* capacity *of an edge $e \in E$ is the quantity $c(e)$.*

*Let $s, t \in V$ be two vertices such that $t$ is accessible from $s$. We call $N = (G, c, s, t)$ a* network *with* source *$s$ and* sink *$t$.*

*A* flow *on $N$ is a map $f : E \to \mathbb{R}_{\geqslant 0}$ which satisfies the following conditions:*

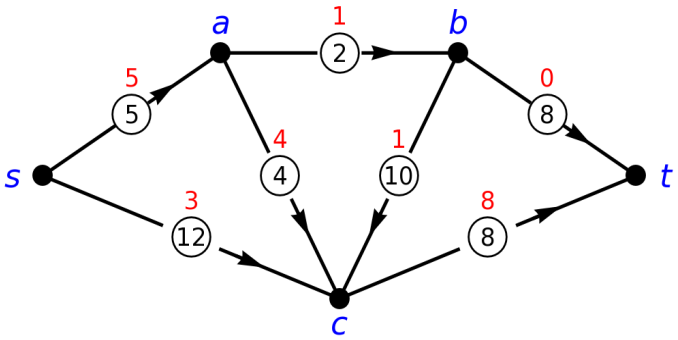
*(F1) For each edge $e \in E$, we have $0 \leqslant f(e) \leqslant c(e)$. We call $e$* saturated *if $f(e) = c(e)$, and* void *if $f(e) = 0$.*

*(F2) For each vertex $v \in V \setminus \{s, t\}$, we have*

$$\sum_{e_+ = v} f(e) = \sum_{e_- = v} f(e),$$

*where $e_-$ is the source of $e$ and $e_+$ its target. This is known as* Kirchoff's Law.

**Example 9.2.** The diagram illustrates a flow of value 8 on a network whose capacities are indicated by the ringed numbers. There are three saturated edges: $s \to a$, $a \to c$, $c \to t$. The edge $b \to t$ is void.




**Example 9.3.** As we explained in the introduction, networks can be used to represent of one-way roads (like motorway carriageways and sliproads) carrying traffic, pipes with carrying fluid or gas at pressure, or a national electricity grid.

**Lemma 9.4.** *Let $N = (G, c, s, t)$ be a network with a flow $f : E \to \mathbb{R}_{\geqslant 0}$. Then, we have*

$$\sum_{s=e_-} f(e) - \sum_{s=e_+} f(e) = \sum_{t=e_+} f(e) - \sum_{t=e_-} f(e). \tag{4}$$

*Proof.* We have

$$\sum_{s=e_-} f(e) + \sum_{t=e_-} f(e) + \sum_{\substack{v\neq s,t \\ v=e_-}} f(e) = \sum_{e\in V} f(e) = \sum_{s=e_+} f(e) + \sum_{t=e_+} f(e) + \sum_{\substack{v\neq s,t \\ v=e_+}} f(e)$$

The result then follows from Condition (F2). □

**Definition 9.5.** *Let $N = (G, c, s, t)$ be a network with a flow $f : E \to \mathbb{R}_{\geqslant 0}$.*

*(a) The quantity given by (4) is called the* value *of the flow $f$, and is denoted by $w(f)$.*

*(b) We say that $f$ is* maximal *if for any flow $f'$ on $N$, we have $w(f') \leqslant w(f)$.*

**Problem.** Given a network with source and sink, find a flow with the maximum possible value, a so-called *maximum flow*.

## 9.2 The max flow, min cut theorem

The main result of this section is known as the *Max Flow, Min Cut Theorem*. It states that the maximal value of a flow always equals the minimal capacity of a cut. We first start with some definition, then we give a characterization of maximal flows.

**Definition 9.6.** *Let $N = (G, c, s, t)$ be a network.*

*(a) A* cut *is a partition of the set $V$ of vertices of into two disjoint subsets $S$ and $T$, one of which contains $s$ and the other $t$:*

$$V = S \sqcup T, \qquad s \in S,\ t \in T.$$

*We denote such a cut by $(S, T)$.*

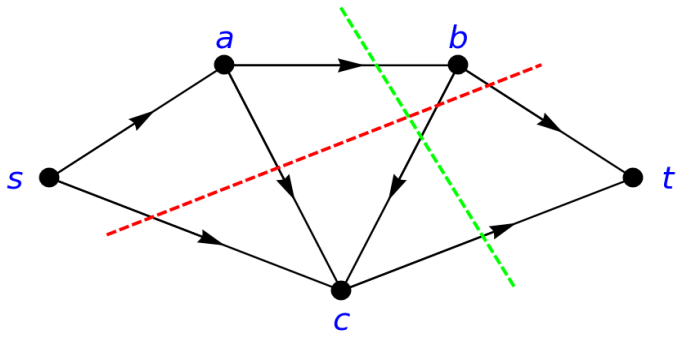
*(b) The* capacity *of a cut $(S, T)$ is the quantity*

$$c(S, T) := \sum_{e_-\in S, e_+\in T} c(e). \tag{5}$$

*(c) We say that the cut $(S, T)$ is* minimal *if for any cut $(S', T')$, we have $c(S, T) \leqslant c(S', T')$.*

**Remark 9.7.** A cut is completely specified by $S$ since $T = V \setminus S$. It is also specified by the arcs that need to be removed to separate $S$ from $T$. If we remove these arcs, we are assuming that the resulting subgraphs are both connected. An obvious special case is always $S = \{s\}$.

**Example 9.8.** A cut can be visualized by means of a line or curve cutting through the edges joining $S$ to $T$. Returning to Example 9.2, we see that the red and the green lines represent two distinct cuts. The red one has capacity 34, and the green one only 10.




**Lemma 9.9.** *Let $N = (G, c, s, t)$ be a network, with a flow $f : E \to \mathbb{R}_{\geqslant 0}$. Let $(S, T)$ be a cut of $N$. Then, we have*

$$w(f) = \sum_{e_- \in S, e_+ \in T} f(e) - \sum_{e_+ \in S, e_- \in T} f(e). \tag{6}$$

*In particular, we have $w(f) \leqslant c(S, T)$. Equality holds if and only if*

- *Every edge $e \in E$, with $e_- \in S$ and $e_+ \in T$, is saturated;*
- *Every edge $e \in E$, with $e_- \in T$ and $e_+ \in S$, is void.*

*Proof.* By summing Equation (4) in Condition (F2), we have

$$\begin{aligned} w(f) &= \sum_{v \in S} \left( \sum_{v = e_-} f(e) - \sum_{v = e_+} f(e) \right) \\ &= \sum_{e_- \in S, e_+ \in S} f(e) + \sum_{e_- \in S, e_+ \in T} f(e) - \sum_{e_+ \in S, e_- \in S} f(e) - \sum_{e_+ \in S, e_- \in T} f(e) \\ &= \sum_{e_- \in S, e_+ \in T} f(e) - \sum_{e_+ \in S, e_- \in T} f(e). \end{aligned}$$

The last equality follows from the fact that the first and second terms of the second equality cancel out. □

**Definition 9.10.** *Let $N = (G, c, s, t)$ be a network, with a flow $f : E \to \mathbb{R}_{\geqslant 0}$. A path $W$ from $s$ to $t$ is called an* augmenting path *with respect to $f$ if*

*(i) $f(e) < c(e)$ holds for every forward edge $e \in W$;*

*(ii) $f(e) > 0$ for every backward edge $e \in W$.*

**Example 9.11.** In Example 9.2, the path $s \to c \leftarrow b \to t$ is an augmenting path.

The following theorem is a characterisation of maximal flows on networks.

**Theorem 9.12** (Augmenting path theorem)**.** *Let $N = (G, c, s, t)$ be a network and $f : E \to \mathbb{R}_{\geqslant 0}$ a flow on $N$. Then, $f$ is maximal if and only if there are no augmenting paths with respect to $f$.*

*Proof.* First, assume that $f$ is a maximal flow. Suppose there is an augmenting path $W$. Let $d$ be the minimum of all values $c(e) - f(e)$ (taken over all forward edges $e$ in $W$) and all values $f(e)$ (taken over the backward edges in $W$). Then $d > 0$, by definition of an augmenting path. Now we define a mapping $f' : E \to \mathbb{R}_{\geqslant 0}$ as follows:

$$f'(e) = \begin{cases} f(e) + d & \text{if } e \text{ is a forward edge in } W, \\ f(e) - d & \text{if } e \text{ is a backward edge in } W, \\ f(e) & \text{otherwise.} \end{cases}$$

It is easily checked that $f'$ is a flow on $N$ with value $w(f') = w(f) + d > w(f)$, contradicting the maximality of $f$.

Conversely, suppose there are no augmenting paths in $N$ with respect to $f$. Let $S$ be the set of all vertices $v$ such that there exists an augmenting path from $s$ to $v$ (including $s$ itself), and put $T = V \setminus S$. By hypothesis, $(S,T)$ is a cut of $N$. Note that each edge $e = uv$ with $e_- = u \in S$ and $e_+ = v \in T$ has to be saturated: otherwise, it could be appended to an augmenting path from $s$ to $u$ to reach the point $v \in T$, a contradiction. Similarly, each edge $e$ with $e_- \in T$ and $e_+ \in S$ has to be void. Then Lemma 9.9 gives $w(f) = c(S,T)$, so that $f$ is maximal. □

**Theorem 9.13** (Integral flow theorem)**.** *Let $N = (G, c, s, t)$ be a network where all capacities $c(e)$ are integers. Then there is a maximal flow $f$ on $N$ such that all values $f(e)$ are integral.*

*Proof.* By setting $f_0(e) = 0$ for all $e$, we obtain an integral flow $f_0$ on $N$ with value $0$. If this trivial flow is not maximal, then there exists an augmenting path with respect to $f_0$. In that case the number $d$ appearing in the proof of Theorem 9.12 is a positive integer, and we can construct an integral flow $f_1$ of value $d$ as in the proof of Theorem 9.12. We continue in the same manner. As the value of the flow is increased in each step by a positive integer and as the capacity of any cut is an upper bound on the value of the flow (by Lemma 9.9), after a finite number of steps we reach an integral flow $f$ for which no augmenting path exists. By Theorem 9.12, this flow $f$ is maximal. □

**Corollary 9.14.** *Let $f$ be a flow on a flow network $N = (G, c, s, t)$, denote by $S_f$ the set of all vertices accessible from $s$ on an augmenting path with respect to $f$, and put $T_f = V \setminus S_f$. Then $f$ is a maximal flow if and only if $t \in T_f$. In this case, $(S_f, T_f)$ is a minimal cut: $w(f) = c(S_f, T_f)$.*

**Theorem 9.15** (Max-flow min-cut)**.** *Let $N = (G, c, s, t)$ be a network, and $f : E \to \mathbb{R}_{\geqslant 0}$ be a flow on $N$. The maximal value of $f$ is equal to the minimal capacity of a cut for $N$.*

*Proof.* The assertion follows from Theorem 9.13 and Corollary 9.14. □

## 9.3 Labelling Algorithm

In this section, we describe more carefully the algorithm that provides an infallible method for increasing the flow through a network, if such an increase is possible.

The proof of Theorem 9.13 suggests the following rough outline of such an algorithm:

**Algorithm 9.16** (Naive Labelling algorithm)**.** Let $N = (G, c, s, t)$ a network. The algorithm output a maximal flow $f$ on $N$.

(1) $f(e) \leftarrow 0$ for all edges $e$;

(2) **while** there exists an augmenting path with respect to $f$ **do**:

(3) let $W = (e_1, \ldots, e_k)$ be an augmenting path from $s$ to $t$, and set

$$\begin{aligned} d \leftarrow \min(&\{c(e_i) - f(e_i) : e_i \text{ is a forward edge in } W\} \\ &\cup \{f(e_i) : e_i \text{ is a backward edge in } W\}); \\ f(e_i) &\leftarrow f(e_i) + d \text{ for each forward edge } e_i; \\ f(e_i) &\leftarrow f(e_i) - d \text{ for each backward edge } e_i; \end{aligned}$$

(4) **od**

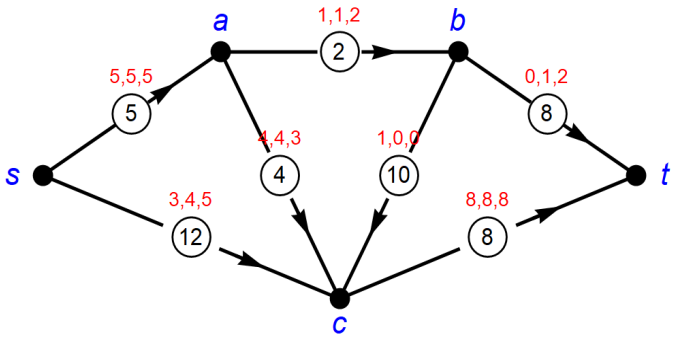
**Example 9.17.** Returning to Example 9.2, we want to find a maximal flow starting with the flow $f$ above. In the first step, we use the augmenting path: $s \to c \leftarrow b \to t$. The possible increment for the flow $f$ is

$$d_0 = \min\{c(s \to c) - f(s \to c), f(c \to b), c(b \to t) - f(b \to t)\} = \min\{12 - 3, 1, 8 - 0\} = 1.$$

So the augmented flow $f_1$ only changes the values of the flow $f$ along the edges $s \to c$, $c \leftarrow b$ and $b \to t$. We get that

$$f_1(e) = \begin{cases} f(e) + 1 & \text{if } \in \{s \to c, b \to t\}; \\ f(e) - 1 & \text{if } \in \{c \leftarrow b\}; \\ f(e) & \text{if } e \notin \{s \to c, c \leftarrow b, b \to t\}. \end{cases}$$

The values of the flow $f_1$ are given by the second entry (in red) above the capacity of each edge in the diagram below.




Next, the path $s \to c \leftarrow a \to b \to t$ is an augmenting path with respect to $f_1$. The new increment with respect to that path is

$$\begin{aligned} d_1 &= \min\{c(s \to c) - f_1(s \to c), f_1(c \to a), c(a \to b) - f_1(a \to b), c(b \to t) - f_1(b \to t)\} \\ &= \min\{12 - 4, 2 - 1, 8 - 0\} = 1. \end{aligned}$$

We get that

$$f_2(e) = \begin{cases} f_1(e)+1 & \text{if } \in \{s \to c, a \to b, b \to t\}; \\ f_1(e)-1 & \text{if } \in \{c \leftarrow a\}; \\ f_1(e) & \text{if } e \notin \{s \to c, c \leftarrow a, a \to b, b \to t\}. \end{cases}$$

The values of the flow $f_2$ are given by the third entry (in red) above the capacity of each edge in the diagram above.

There is no augmenting path with respect to the flow $f_2$ since all the forward edges from $a$ and $c$ are saturated, and all the backward ones have flow $0$. Therefore, by Theorem 9.12, $f_2$ is maximal. The vertices that are accessible from $s$ on an augmenting path with respect to $f_2$ are $a$ and $c$. So setting $S_2 = \{s, a, c\}$ and $T_2 = V \setminus S_2 = \{b, t\}$, we get a minimal cut by Corollary 9.14. We verify that

$$c(S_2, T_2) = 10 = w(f_2).$$

A more refined version of Algorihm 9.16 is the Ford and Fulkerson Labelling algorithm given below. It uses an analogue of the BFS algorithm to find a maximal flow.

**Algorithm 9.18** (Labelling algorithm)**.** Let $N = (G, c, s, t)$ a network. The algorithm output a maximal flow $f$ on $N$. The algorithm iterates the following subroutine.

(1) $\varepsilon = 0$;

(2) $Q = (s)$;

(3) $L(s) = \infty$;

(4) Add $(s, L(s))$ to the table;

(5) **while** $Q$ is non-empty and has first element $x$ **do**:

(6) **while** there exists $y$ adjacent to $x$ not already in $Q$ **do**:

(7) **if** $(x, y) \in E$ and $f(x, y) < c(x, y)$ **then**:

(8) add $y$ to $Q$;

(9) $L(y) = \min\{L(x),\ c(x, y) - f(x, y)\}$;

(10) add $(y, L(y), x, +)$ to the table;

(11) **if** $y = t$ **then**:

(12) $\varepsilon = L(t)$;

(13) stop;

(14) **elif** $(y, x) \in E$ and $f(y, x) > 0$:

(15) add $y$ to $Q$;

(16) $L(y) = \min\{L(x),\ f(y, x)\}$;

(17) add $(y, L(y), x, -)$ to the table;

(18) remove $x$ from $Q$;

(19) return $\varepsilon$ and the table

**Example 9.19.** We rework in Example 9.17, using the Relabelling Algorithm 9.18. Again, we want to find a maximal flow starting with the flow $f_0 = f$. We'll apply the labels in a table to avoid further complicating the diagram

| | $s$ | $a$ | $b$ | $c$ | $t$ | |
|---|---|---|---|---|---|---|
| 1st iteration: | $\infty$ | $4c^-$ | $1c^-$ | $9s^+$ | $1b^+$ | Queue is $scabt$ |

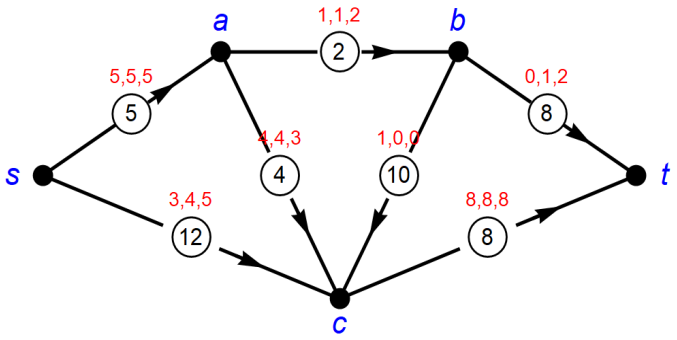
Below each vertex is a number indicating the amount of flow that can be transferred towards that vertex, from which vertex it was transferred, and in which direction.

We can now update the flow by $d = 1$ (the amount reaching $t$) along the augmenting path

$$s \to c \leftarrow b \to t,$$

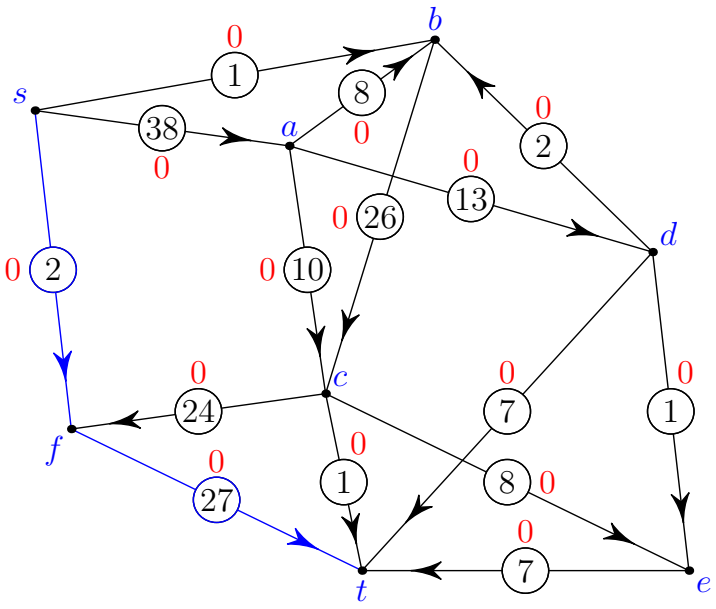
which is remembered with the aid of the symbols in the last row. Forward edges have the flow increased by $d$, backwards ones have it reduced by $d$. We can now remove all the labels and apply the same procedure to the updated flow to perform a second iteration:

| | $s$ | $a$ | $b$ | $c$ | $t$ | |
|---|---|---|---|---|---|---|
| 2nd iteration: | $\infty$ | $4c^-$ | $1a^+$ | $8s^+$ | $1b^+$ | Q is $scabt$ |
| 3rd iteration: | $\infty$ | $3c^-$ | | $7s^+$ | | Q is $sca$ |




In the third iteration, we cannot augment any edges beyond $a$ or $c$ because forward ones are saturated and backward ones have flow 0. This means that the second iteration produced a *maximum flow*. The diagram shows the all the flow numbers after $n = 0, 1, 2$ iterations, and the maximum flow has value 10.

With hindsight, it was obvious that our network admits a flow with value 10 – we can send 2 units along the path $sabt$ and 8 units along the path $sct$. However, the Labelling Algorithm has the advantage that it can be applied to any *any* flow, including the one with all numbers 0, which would have produced the more obvious maximum flow.




| $k$ | $x$ | $y \in A_x \setminus Q$ | Edge | Label $L(y)$ | Queue |
|---|---|---|---|---|---|
| 1 | $s$ | $a$ | + | $\min\{L(s), c(s \to a) - f_0(s \to a)\} = 38$ | $sabf$ |
| | | $b$ | + | $\min\{L(s), c(s \to b) - f_0(s \to b)\} = 1$ | |
| | | $f$ | + | $\min\{L(s), c(s \to f) - f_0(s \to f)\} = 2$ | |
| | $a$ | $c$ | + | $\min\{L(a), c(a \to c) - f_0(a \to c)\} = 10$ | $sabfcd$ |
| | | $d$ | + | $\min\{L(a), c(a \to c) - f_0(a \to c)\} = 13$ | |
| | $b$ | $\{\}$ | | | $sabfcd$ |
| | $f$ | $t$ | + | $\min\{L(f), c(f \to t) - f_0(f \to t)\} = 2$ | $sabfcdt$ |

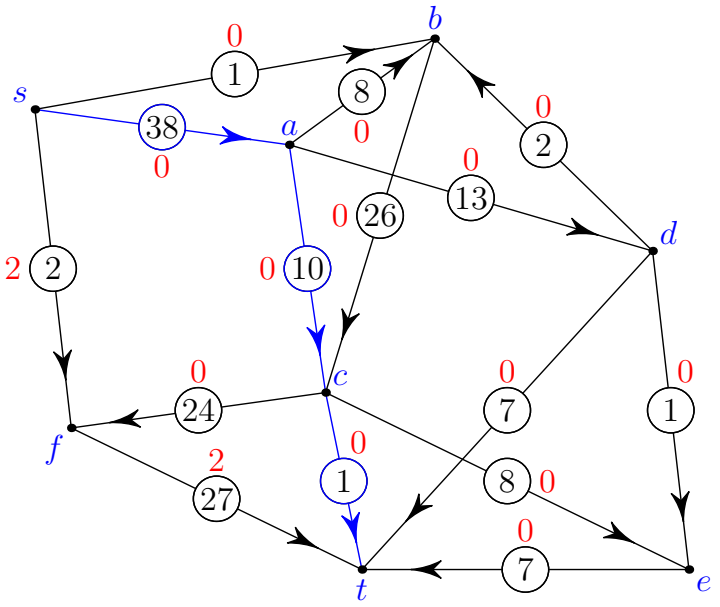
Figure 3: First iteration: $w(f_0) = 0$

**Remark 9.20.** Algorithm 9.18 also returns the cut associated to the maximal flow by Corollary 9.14. In Example 9.19, we obtain the cut $S = \{s, a, c\}$ and $T = \{b, t\}$. The capacity of this cut is

$$c(S, T) = c(a \to b) + c(c \to t) = 2 + 8 = 10.$$

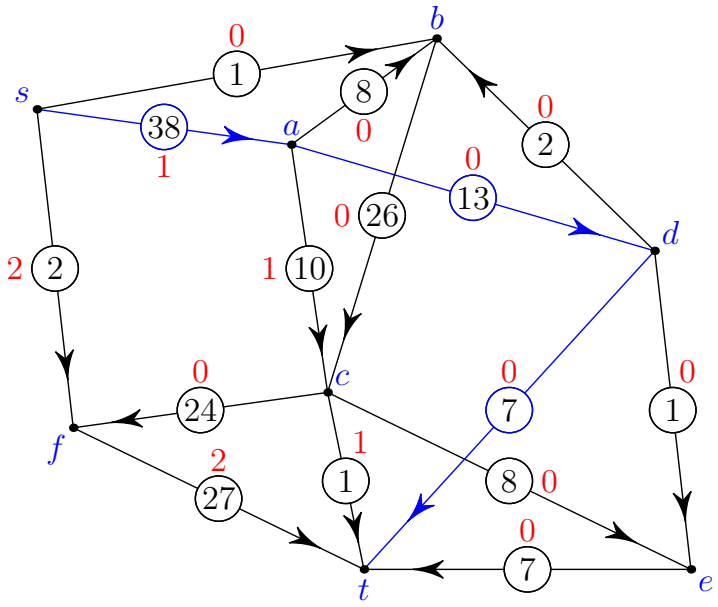
**Example 9.21.** We want to find a maximal flow on the network in Figure 3 using Algorithm 9.18. The details of the calculations are given in the Figures 3, 4, 5, 6, 7, 8, 9, 10, 11 and 12, and the summary in Table 2. In the column *Edge*, we indicate whether an edge is forward by $+$, and backward by $-$; and by $s$ that it is saturated.

The maximal flow is obtained after 9 iterations. At each step, one determines the augmenting path (winning path) by using the second and third columns of the table and working from bottom to top. For example, in the first step, we see that the augmenting path is $s \to f \to t$–it is shown in blue. We use the data of the $k$-th iteration to obtain the $k$-th row of Table 2. For example, in the first step, we see that the flow into $a$ comes from $s$, so the label for $a$ is $38s^+$,



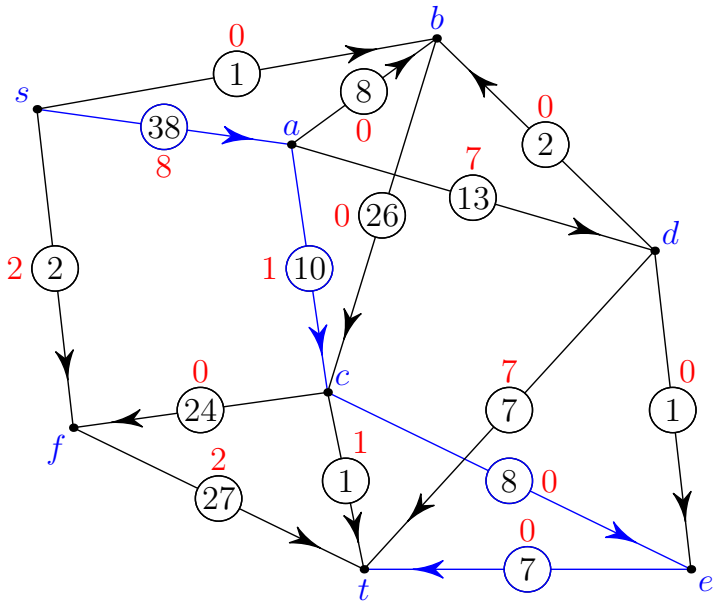

| $k$ | $x$ | $y \in A_x \setminus Q$ | Edge | Label $L(y)$ | Queue |
|---|---|---|---|---|---|
| 2 | $s$ | $a$ | + | $\min\{L(s), c(s \to a) - f_1(s \to a)\} = 38$ | $sab$ |
| | | $b$ | + | $\min\{L(s), c(s \to b) - f_1(s \to b)\} = 1$ | |
| | | $f$ | +/s | | |
| | $a$ | $c$ | + | $\min\{L(a), c(c \to a) - f_1(c \to a)\} = 10$ | $sabcd$ |
| | | $d$ | + | $\min\{L(a), c(d \to a) - f_1(d \to a)\} = 13$ | |
| | $b$ | $\{\}$ | | | $sabcd$ |
| | $c$ | $t$ | + | $\min\{L(c), c(c \to t) - f_1(c \to t)\} = 1$ | $sabcdt$ |
| | | $e, f$ | + | | |

Figure 4: Second iteration: $w(f_1) = 2$



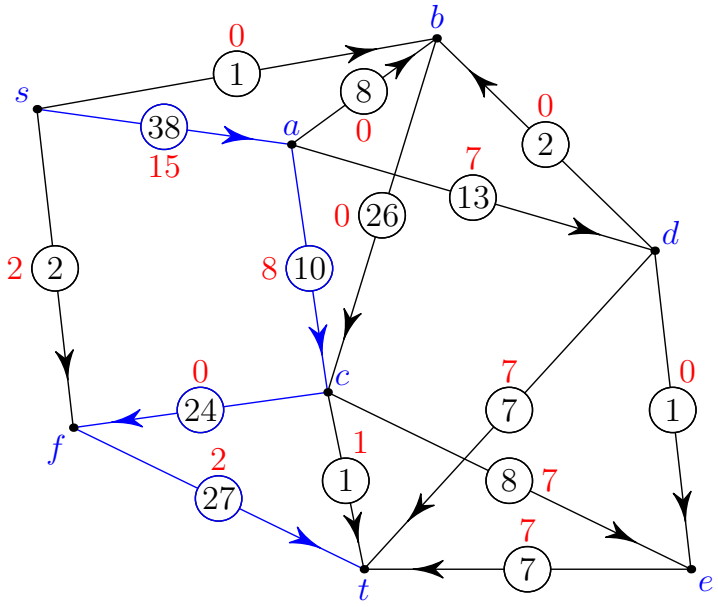

| $k$ | $x$ | $y \in A_x \setminus Q$ | Edge | Label $L(y)$ | Queue |
|---|---|---|---|---|---|
| 3 | $s$ | $a$ | + | $\min\{L(s), c(s \to a) - f_2(s \to a)\} = 37$ | $sab$ |
| | | $b$ | + | $\min\{L(s), c(s \to b) - f_2(s \to b)\} = 1$ | |
| | | $f$ | +/s | | |
| | $a$ | $c$ | + | $\min\{L(a), c(a \to c) - f_2(a \to c)\} = 9$ | $sabcd$ |
| | | $d$ | + | $\min\{L(a), c(a \to d) - f_2(a \to d)\} = 13$ | |
| | $b$ | $\{\}$ | | | $sabcd$ |
| | $c$ | $e$ | + | $\min\{L(c), c(c \to e) - f_2(c \to e)\} = 8$ | $sabcdef$ |
| | | $f$ | + | $\min\{L(c), c(c \to f) - f_2(c \to f)\} = 9$ | |
| | | $t$ | +/s | | |
| | $d$ | $t$ | + | $\min\{L(d), c(d \to t) - f_2(d \to t)\} = 7$ | $sabcdeft$ |

Figure 5: The third iteration: $w(f_2) = 3$



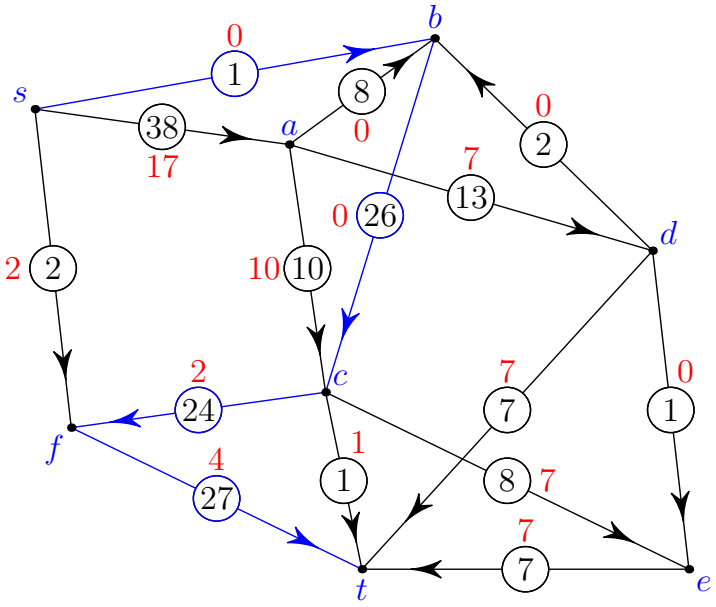

| $k$ | $x$ | $y \in A_x \setminus Q$ | Edge | Label $L(y)$ | Queue |
|---|---|---|---|---|---|
| 4 | $s$ | $a$ | + | $\min\{L(s), c(s \to a) - f_3(s \to a)\} = 30$ | $sab$ |
| | | $b$ | + | $\min\{L(s), c(s \to b) - f_3(s \to b)\} = 1$ | |
| | | $f$ | +/s | | |
| | $a$ | $c$ | + | $\min\{L(a), c(a \to c) - f_3(a \to c)\} = 9$ | $sabcd$ |
| | | $d$ | + | $\min\{L(a), c(a \to d) - f_3(a \to d)\} = 6$ | |
| | $b$ | $\{\}$ | | | $sabcd$ |
| | $c$ | $e$ | + | $\min\{L(c), c(c \to e) - f_3(c \to e)\} = 8$ | $sabcdef$ |
| | | $f$ | + | $\min\{L(c), c(c \to f) - f_3(c \to f)\} = 9$ | |
| | | $t$ | +/s | | |
| | $d$ | $t$ | +/s | | $sabcdef$ |
| | $e$ | $t$ | + | $\min\{L(e), c(e \to t) - f_3(e \to t)\} = 7$ | $sabcdeft$ |

Figure 6: The fourth iteration: $w(f_3) = 10$



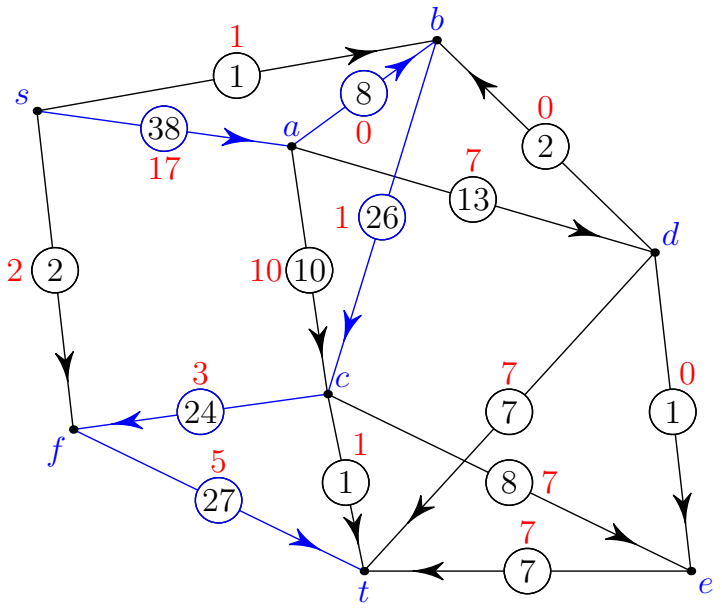

| $k$ | $x$ | $y \in A_x \setminus Q$ | Edge | Label $L(y)$ | Queue |
|---|---|---|---|---|---|
| 5 | $s$ | $a$ | + | $\min\{L(s), c(s \to a) - f_4(s \to a)\} = 23$ | $sab$ |
| | | $b$ | + | $\min\{L(s), c(s \to b) - f_4(s \to b)\} = 1$ | |
| | | $f$ | +/s | | |
| | $a$ | $c$ | + | $\min\{L(a), c(a \to c) - f_4(a \to c)\} = 2$ | $sabcd$ |
| | | $d$ | + | $\min\{L(a), c(a \to d) - f_4(a \to d)\} = 6$ | |
| | $b$ | $\{\}$ | | | $sabcd$ |
| | $c$ | $e$ | + | $\min\{L(c), c(c \to e) - f_4(c \to e)\} = 1$ | $sabcdef$ |
| | | $f$ | + | $\min\{L(c), c(c \to f) - f_4(c \to f)\} = 2$ | |
| | | $t$ | +/s | | |
| | $d$ | $t$ | +/s | | $sabcdef$ |
| | $e$ | $t$ | +/s | | $sabcdef$ |
| | $f$ | $t$ | + | $\min\{L(f), c(f \to t) - f_4(f \to t)\} = 2$ | $sabcdeft$ |

Figure 7: The fifth iteration: $w(f_4) = 17$



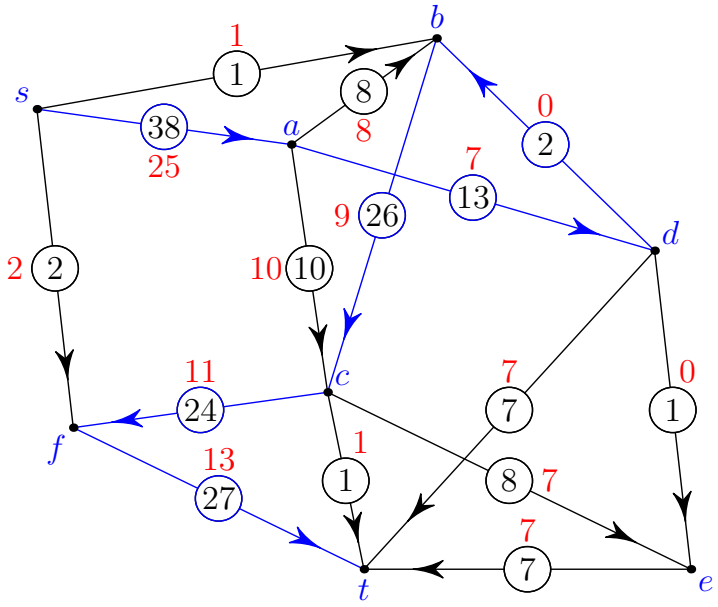

| $k$ | $x$ | $y \in A_x \setminus Q$ | Edge | Label $L(y)$ | Queue |
|---|---|---|---|---|---|
| 6 | $s$ | $a$ | + | $\min\{L(s), c(s \to a) - f_5(s \to a)\} = 21$ | $sab$ |
| | | $b$ | + | $\min\{L(s), c(s \to b) - f_5(s \to b)\} = 1$ | |
| | | $f$ | +/s | | |
| | $a$ | $d$ | + | $\min\{L(a), c(d \to a) - f_5(d \to a)\} = 6$ | $sabd$ |
| | | $c$ | +/s | | |
| | $b$ | $c$ | + | $\min\{L(b), c(b \to c) - f_5(b \to c)\} = 1$ | $sabdc$ |
| | $c$ | $e$ | + | $\min\{L(c), c(c \to e) - f_5(c \to e)\} = 1$ | $sabdcef$ |
| | | $f$ | + | $\min\{L(c), c(c \to f) - f_5(c \to f)\} = 1$ | |
| | | $t$ | +/s | | |
| | $e$ | $t$ | +/s | | $sabdcef$ |
| | $d$ | $t$ | +/s | | $sabdcef$ |
| | $f$ | $t$ | + | $\min\{L(f), c(f \to t) - f_5(f \to t)\} = 1$ | $sabdceft$ |

Figure 8: The sixth iteration: $w(f_5) = 19$



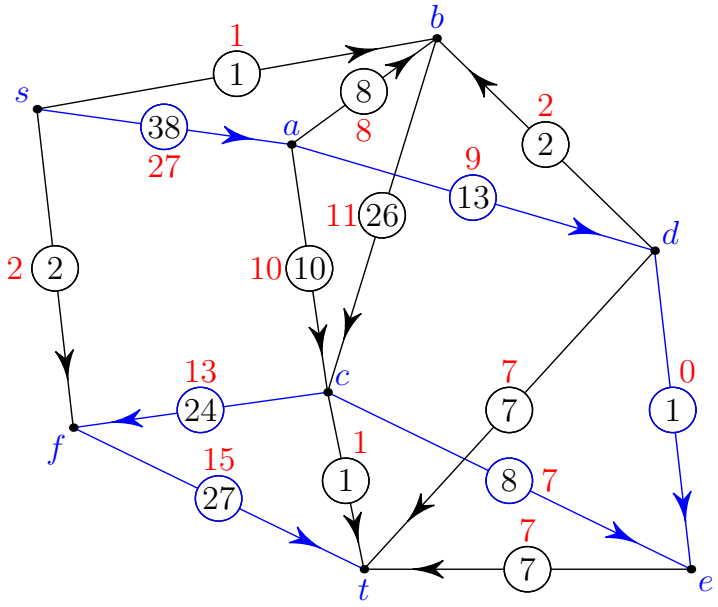

| $k$ | $x$ | $y \in A_x \setminus Q$ | Edge | Label $L(y)$ | Queue |
|---|---|---|---|---|---|
| 7 | $s$ | $a$ | + | $\min\{L(s), c(s \to a) - f_6(s \to a)\} = 21$ | $sa$ |
| | | $b$ | +/s | | |
| | | $f$ | +/s | | |
| | $a$ | $b$ | + | $\min\{L(a), c(b \to a) - f_6(b \to a)\} = 8$ | $sabd$ |
| | | $d$ | + | $\min\{L(a), c(a \to d) - f_6(a \to d)\} = 6$ | |
| | | $c$ | +/s | | |
| | $b$ | $c$ | + | $\min\{L(c), c(b \to c) - f_6(b \to c)\} = 8$ | $sabdc$ |
| | $d$ | $e$ | | $\min\{L(f), c(d \to e) - f_6(d \to e)\} = 1$ | $sabdce$ |
| | | $t$ | +/s | | |
| | $c$ | $e$ | + | $\min\{L(c), c(c \to e) - f_6(c \to e)\} = 1$ | $sabdcef$ |
| | | $f$ | + | $\min\{L(c), c(c \to f) - f_6(c \to f)\} = 8$ | |
| | | $t$ | +/s | | |
| | $e$ | $t$ | +/s | | $sabdcef$ |
| | $f$ | $t$ | + | $\min\{L(f), c(f \to t) - f_6(f \to t)\} = 8$ | $sabdceft$ |

Figure 9: The seventh iteration: $w(f_6) = 20$



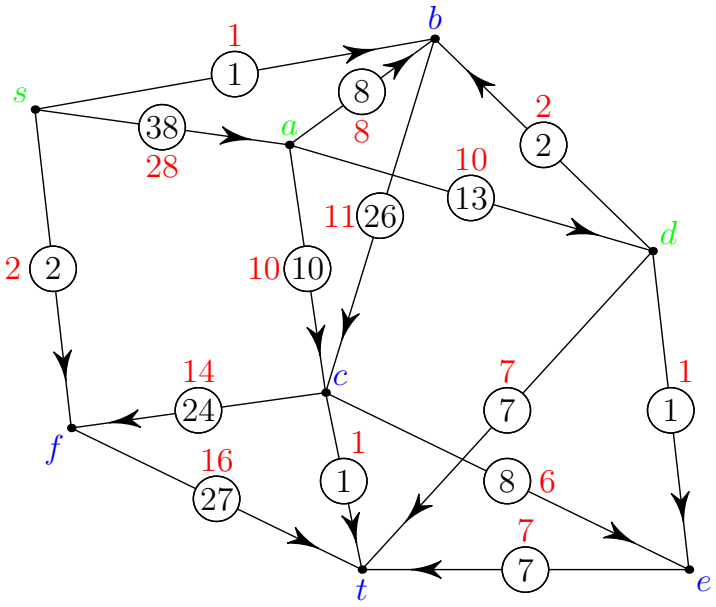

| $k$ | $x$ | $y \in A_x \setminus Q$ | Edge | Label $L(y)$ | Queue |
|---|---|---|---|---|---|
| 8 | $s$ | $a$ | + | $\min\{L(s), c(s \to a) - f_7(s \to a)\} = 13$ | $sa$ |
| | | $b$ | +/s | | |
| | | $f$ | +/s | | |
| | $a$ | $d$ | + | $\min\{L(a), c(d \to a) - f_7(d \to a)\} = 6$ | $sad$ |
| | | $b, c$ | +/s | | |
| | $d$ | $e$ | | $\min\{L(d), c(d \to e) - f_7(d \to e)\} = 1$ | $sade$ |
| | | $t$ | +/s | | |
| | $e$ | $c$ | − | $\min\{L(e), f_7(c \to e)\} = 1$ | $sadec$ |
| | | $t$ | +/s | | |
| | $c$ | $f$ | + | $\min\{L(c), c(c \to f) - f_7(c \to f)\} = 1$ | $sadecf$ |
| | | $t$ | +/s | | |
| | $f$ | $t$ | + | $\min\{L(f), c(f \to t) - f_7(f \to t)\} = 1$ | $sadecft$ |

Figure 10: The eighth iteration: $w(f_7) = 28$




| $k$ | $x$ | $y \in A_x \setminus Q$ | Edge | Label $L(y)$ | Queue |
|---|---|---|---|---|---|
| 9 | $s$ | $a$ | + | $\min\{L(s), c(s \to a) - f_8(s \to a)\} = 11$ | $sa$ |
| | | $b$ | +/s | | |
| | | $f$ | +/s | | |
| | $a$ | $d$ | + | $\min\{L(a), c(d \to a) - f_8(d \to a)\} = 4$ | $sad$ |
| | | $b, c$ | +/s | | |
| | $d$ | $e$ | | $\min\{L(d), c(d \to e) - f_8(d \to e)\} = 1$ | $sade$ |
| | | $t$ | +/s | | |
| | $e$ | $c$ | − | $\min\{L(e), f_8(c \to e)\} = 1$ | $sadec$ |
| | | $t$ | +/s | | |
| | $c$ | $f$ | + | $\min\{L(c), c(c \to f) - f_8(c \to f)\} = 1$ | $sadecf$ |
| | | $t$ | +/s | | |
| | $f$ | $t$ | + | $\min\{L(f), c(f \to t) - f_8(f \to t)\} = 1$ | $sadecft$ |

Figure 11: The ninth iteration: $w(f_8) = 30$




| $k$ | $x$ | $y \in A_x \setminus Q$ | Edge | Label $L(y)$ | Queue |
|---|---|---|---|---|---|
| 10 | $s$ | $a$ | + | $\min\{L(s), c(s \to a) - f_9(s \to a)\} = 10$ | $sa$ |
| | | $b$ | +/s | | |
| | | $f$ | +/s | | |
| | $a$ | $d$ | + | $\min\{L(a), c(d \to a) - f_9(d \to a)\} = 3$ | $sad$ |
| | | $b, c$ | +/s | | |
| | $d$ | $e$ | +/s | | $sad$ |
| | | $t$ | +/s | | |

Figure 12: The tenth iteration: $w(f_9) = 31$

| $k$ | $s$ | $a$ | $b$ | $c$ | $d$ | $e$ | $f$ | $t$ | $Q$ |
|---|---|---|---|---|---|---|---|---|---|
| 1 | $\infty$ | $38s^+$ | $1s^+$ | $10a^+$ | $13a^+$ | | $2s^+$ | $2f^+$ | $sabcdt$ |
| 2 | $\infty$ | $38s^+$ | $1s^+$ | $10a^+$ | $13a^+$ | | | $1c^+$ | $sabcdt$ |
| 3 | $\infty$ | $37s^+$ | $1s^+$ | $9a^+$ | $13a^+$ | $8c^+$ | $9c^+$ | $7d^+$ | $sabcdeft$ |
| 4 | $\infty$ | $30s^+$ | $1s^+$ | $9a^+$ | $6a^+$ | $8c^+$ | $9c^+$ | $7e^+$ | $sabcdeft$ |
| 5 | $\infty$ | $23s^+$ | $1s^+$ | $2a^+$ | $6a^+$ | $1c^+$ | $2c^+$ | $2f^+$ | $sabdceft$ |
| 6 | $\infty$ | $21s^+$ | $1s^+$ | $1b^+$ | $6a^+$ | $1c^+$ | $1c^+$ | $1f^+$ | $sabdceft$ |
| 7 | $\infty$ | $21s^+$ | $8a^+$ | $8b^+$ | $6a^+$ | $1c^+$ | $8c^+$ | $8f^+$ | $sadceft$ |
| 8 | $\infty$ | $13s^+$ | | $1e^-$ | $6a^+$ | $1d^+$ | $1c^+$ | $1f^+$ | $sadceft$ |
| 9 | $\infty$ | $11s^+$ | | $1e^-$ | $4a^+$ | $1d^+$ | $1c^+$ | $1f^+$ | $sadceft$ |
| 10 | $\infty$ | | | $3a^+$ | | | | | $sad$ |

Table 2: Summary of the Labelling algorithm for Example 9.23

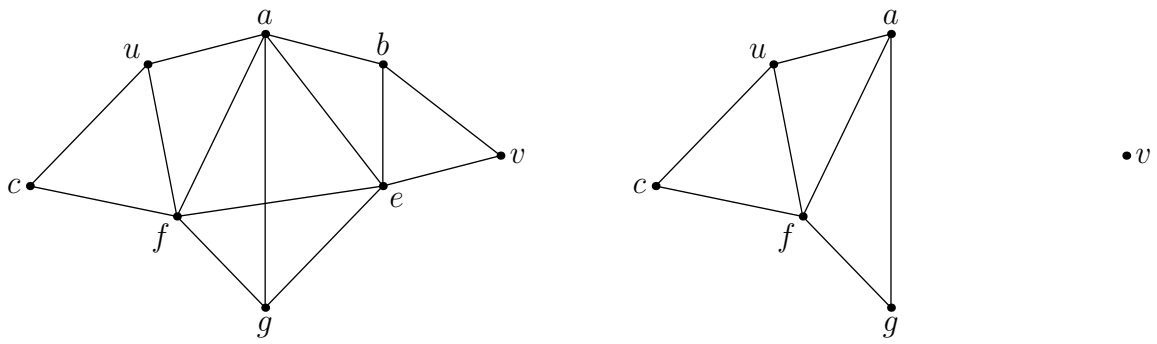



Figure 13: The sets $\{e, a\}$ and $\{e, b\}$ are vertex separator for $u$ and $v$

the flow into $b$ also comes from $s$, hence the label $1b^+$. But the flow into $c$ comes from $a$, hence the label $10a^+$. And so on.

The cut associate to the maximal flow is $S = \{s, a, d\}$ and $T = \{b, c, e, f, t\}$. The value of the flow is $w(f_9) = 31$.

## 9.4 Menger's theorem*

**Definition 9.22.** *Let $G = (V, E)$ be a connected graph. Let $u, v \in V$ be non-adjacent, and $X \setminus \{u, v\}$. We say that $X$ is a* vertex separator *for $u$ and $v$ (or a $(u, v)$-*separating set*) if we can disconnect both $G$, and $u$ and $v$, by deleting the vertices in $X$. We say that $X$ is a* minimal vertex separator *for $u$ and $v$ or (a* minimal $(u, v)$-separating set*) if it is not properly contained in another $(u, v)$-vertex separator.*

**Example 9.23.** In the graph in Figure 13, the two sets $\{e, a\}$ and $\{e, b\}$ are $(u, v)$-separators. Both sets have minimal size as $(u, v)$-separating sets.

**Example 9.24.** In the graph in Figure 14, we see that we can disconnected both $G$ and $u, v$ by deleting $v_1$, $v_2$ and $v_3$. This is a minimal size for a $(u, v)$-separating set.

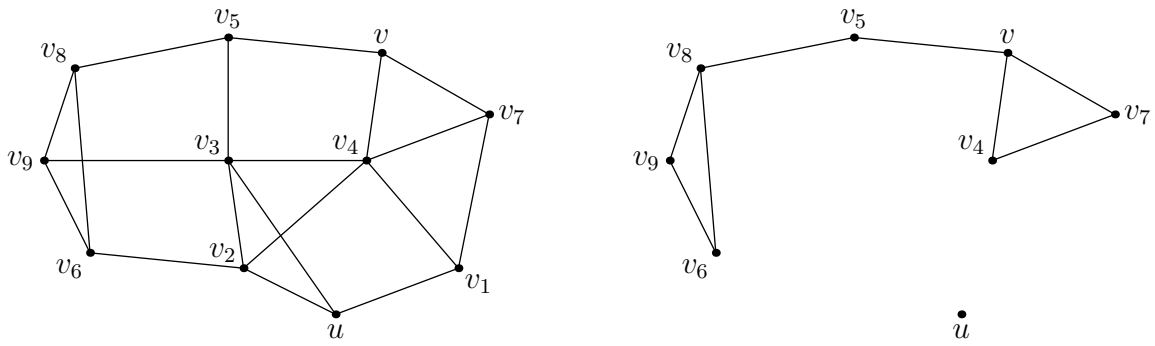



Figure 14: A $(u, v)$-separating set for the graph is $\{v_1, v_2, v_3\}$

**Definition 9.25.** *Let $G = (V, E)$ be a connected graph. Let $u, v \in V$ be non-adjacent. A set of $(u, v)$-path $S = \{P_1, \ldots, P_k\}$ is called an* edge separator *(or* internally disjoint*) if no two paths in $S$ have a common vertex other than $u$ and $v$.*

**Example 9.26.** The paths

$$P_1 = v_1v_3v_5v_7v_6,$$
$$P_2 = v_1v_2v_8v_6,$$
$$P_3 = v_1v_4v_7v_6$$

are *not* internally disjoint since $P_1$ and $P_3$ have the vertex $v_7$ in common. However, the paths

$$Q_1 = v_1v_4v_{10}v_2,$$
$$Q_2 = v_1v_6v_8v_9v_5v_2,$$
$$Q_3 = v_1v_3v_7v_2$$

are internally disjoint.

**Theorem 9.27** (Menger's theorem)**.** *Let $G = (V, E)$ be a connected graph and $u, v \in V$ be non-adjacent. The size of a minimal $(u, v)$-separating set is equal to the maximum number of internally disjoint $(u, v)$-paths in $G$.*

**Example 9.28.** In Example 9.24, $\{v_4, v_5\}$ is a minimal $(u, v)$-separating set. So by Menger's theorem, this is the maximal number of internally disjoint $(u, v)$-paths in $G$.

**Example 9.29.** Let $A$ and $B$ two Earth's locations, and $G = (V, E)$ be a network of satellites relaying communications between $A$ and $B$. Assume there is a magnetic storm from the sun.

**Question:** How many satellites have to be disabled by the storm in order to *totally* disrupt communications between $A$ and $B$? This example pertains to a recent news article about the lost of 40 satellites by Starlink https://www.bbc.co.uk/news/world-60317806.

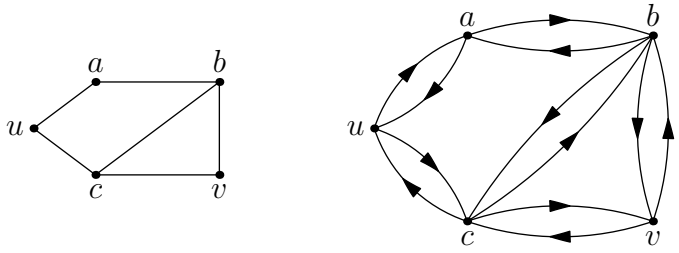



Figure 15: Replacing every edge $xy$ with directed arcs $x \to y$ and $y \to x$

The Falcon 9 rocket launch on 3 February 2022 which carried 49 Starlink satellites, most of which were caught by the storm

Before we prove Menger's theorem, we will do some preparation.

We saw the notion of cut for networks. Now we extend this to graph.

We construct a graph $G^*$ in the following two steps:

(a) Replace every edge $xy$ with directed arcs $x \to y$ and $y \to x$ (see Figure 15).

(b) Let $G^* = (V^*, E^*)$ be the *directed* graph obtained as follows (see Figure 16):

(i) If $x \notin \{u, v\}$, we split $x$ into two vertices $x^-$ and $x^+$, and put a new directed edge $x^- \to x^+$ between the two. We call $x^- \to x^+$ and *internal arc*.

(ii) If $x \to y$ is an edge for the graph obtained in (a), we replace it by

- $x^+ \to v$ if $x, y \notin \{u, v\}$;
- $x \to y^-$ if $x \in \{u, v\}$;
- $x^+ \to y$ if $y \in \{u, v\}$.

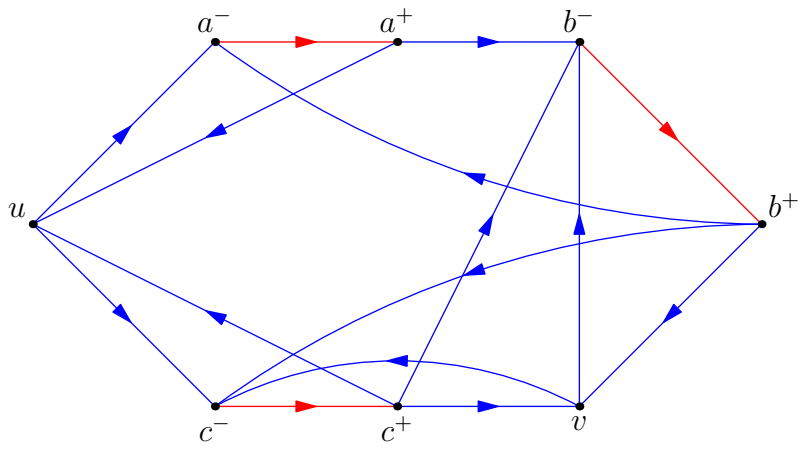



Figure 16: Splitting vertices into internal arcs

**Definition 9.30.** *Let $G = (V, E)$ be a connected simple graph, and $u, v \in V$ two distinct vertices. We call $G^* = (V^*, E^*)$ the* augmented directed graph with source $u$ and sink $v$ *obtained from $G$.*

**Lemma 9.31.** *Let $G = (V, E)$ be a connected simple graph, and $u, v \in V$ two distinct vertices. Let $X \subset V$ be a $(u, v)$-vertex separator for $G$. Then there exists a $(u, v)$-cut $(S^*, T^*)$ for $G^*$ such that every arc that connects $S^*$ to $T^*$ is an internal arc $x^- \to x^+$ for some $x \in X$.*

*Proof.* If $P$ is a path in $G$, we denote by $V(P)$ the vertices supporting $P$. Similarly, if $Q$ is a path in $G^*$, we denote by $V(Q)$ the vertices supporting $Q$. We define $S^*$ and $T^*$ by

$$S^* := \{w \in V^* : w \in V(Q) \text{ where } Q \text{ is a path between } u \text{ and } x^- \text{ for } x \in X\};$$
$$T^* := V^* \setminus S^*.$$

Since $X$ is a $(u, v)$-separator, every path $P$ that connects $u$ to $v$ can be written as

$$P = u - x_1 - x_2 - \cdots - x_k - x - y_1 - y_2 - \cdots - y_m - v,$$

with $x \in X$, and such that $\{x_1, x_2, \ldots x_k\} \cap \{y_1, y_2, \ldots y_m\} = \varnothing$. We can convert $P$ into a *directed* path $\overrightarrow{P} = \overrightarrow{P}_{u,x^-} \sqcup \overrightarrow{P}_{x^+,v}$ in $G^*$, where

$$\overrightarrow{P}_{u,x^-} = u \to x_1^- \to x_1^+ \to x_2^- \to x_2^+ \to \cdots \to x_k^- \to x_k^+ \to x^-;$$
$$\overrightarrow{P}_{x^+,v} = x^+ \to y_1^- \to y_1^+ \to y_2^- \to y_2^+ \to \cdots \to y_m^- \to y_m^+ \to v.$$

This decomposition is unique.

□

*Proof of Menger's theorem.* Let $c(u, v)$ be the size of a minimal $(u, v)$-separating set, and $\lambda(u, v)$ the maximum number of an internally disjoint $(u, v)$-paths in $G$. We want to show that $c(u, v) = \lambda(u, v)$.

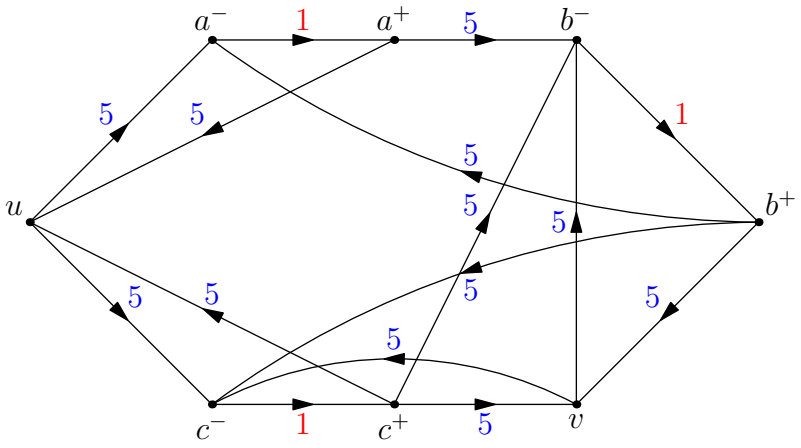
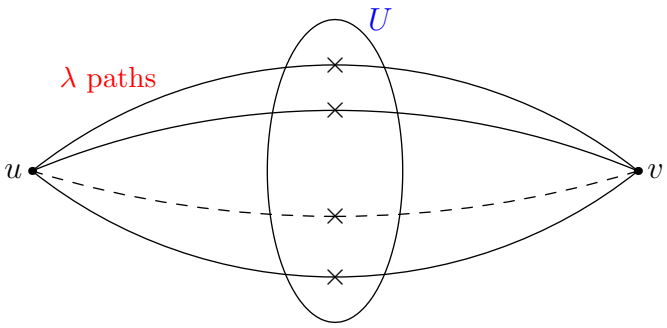



Figure 17: Adding capacities to the network

Let $U$ be a minimal $(u, v)$-separating set for $G$. By definition of this set, *every* $(u, v)$-path must go through $U$.




So the number of internally disjoint paths is at most $|U|$; hence we must have

$$\lambda(u, v) \leqslant |U| = c(u, v).$$

To complete the proof of the theorem, we must show that $c(u, v) \leqslant \lambda(u, v)$.

We define a network $N^* = (G^*, c^*, u, v)$, with source $u$, sink $v$ and capacity $c^*$. The capacities are assigned as follows (see Figure 17):

(a) If $e \in E^*$ is an internal arc, we let $c^*(e) = 1$;

(b) If $e \in E^*$ is that is *not* an internal arc, we let $c^*(e) = n$, where $n = |V|$.

**Goal:** We want to show that

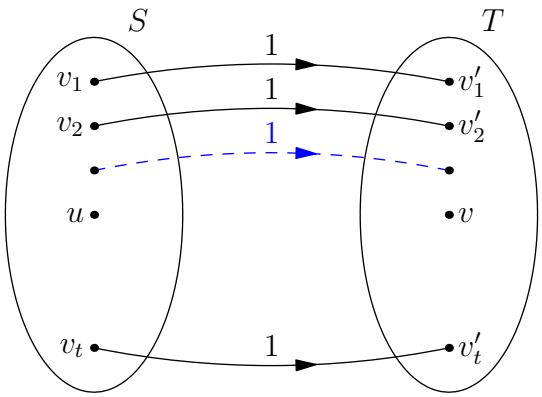
- max-flow of $N^* \leqslant \lambda(u, v)$;
- $c(u, v) \leqslant$ min-cut of $N^*$.

Combining these two facts, with the Max-Flow Min-Cut Theorem (Theorem 9.15), we obtain

$$c(u,v) \leqslant \text{min-cut of } N^* = \text{max-flow of } N^* \leqslant \lambda(u,v).$$

To show that max-flow of $N^* \leqslant \lambda(u,v)$, let $f$ be a maximum flow, with value $w(f) = m$. If there is a flow into $x^-$, the value of that flow has to be 1 since there is only one directed arc from $x^-$. This unit flow travels from $u$ to $v$. The $m$ unit flows will transform into an internally disjoint set of $(u,v)$-paths, hence max-flow of $N^* \leqslant \lambda(u,v)$.

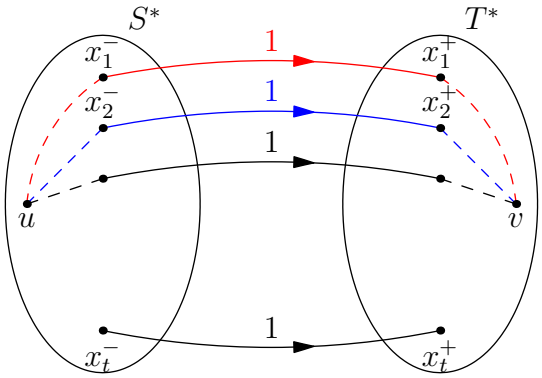
To show that $c(u,v) \leqslant$ min-cut of $N^*$, take a $(u,v)$-cut $(S,T)$ for the network $N^*$, assume for a contradiction that there is a *non-internal* directed arc from $S$ to $T$. Then, by construction of $N^*$, we have $c(S,T) \geqslant n$.




Now consider the flow with $(u,v)$-cut $(S^*,T^*)$, where

$$S^* = \{u\} \cup \{x^- : ux \in E\};$$
$$T^* = V \setminus S^*.$$

Every path that crosses from $S^*$ to $T^*$ contains an internal arc.




This implies that

$$c(S^*,T^*) \leqslant n-2 < n.$$

However, this is a contradiction. Therefore, all arcs must be internal arcs.

Let $(S,T)$ be a minimal cut for $N^*$, and consider the set

$$U = \{x \in V : x^- \in S, x^+ \in T\}.$$

Since all the arcs in $G^*$ are internal and $(S,T)$ is a minimal cut for $N^*$, we see that $|U| = c(S,T)$.

We will show that $U$ is a minimal $(u,v)$-separating set for $G$. From this it follows that

$$c(u,v) \leqslant |U| = c(S,T) = \text{min-cut for } N^*.$$

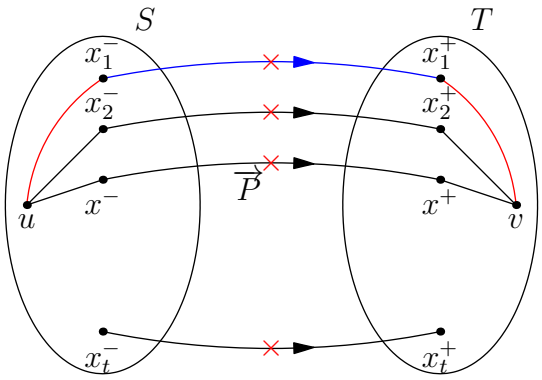
To prove the claim, consider a path

$$P = u - v_1 - \cdots - v_k - v$$

in $G$. We can convert it into a path in $N^*$, given by

$$\overrightarrow{P} = u \to v_1^- \to v_1^+ \to \cdots \to x^- \to x^+ \to \cdots \to v_k^- \to v_k^+ \to v,$$

with $x^- \in S$ and $x^+ \in T$.




Deleting these internal arcs disconnects the graph $N^*$. But the internal arcs come from the vertices in $U$. So $U$ is a $(u,v)$-separating set for $G$. □

## 9.5 Dynamic programming*

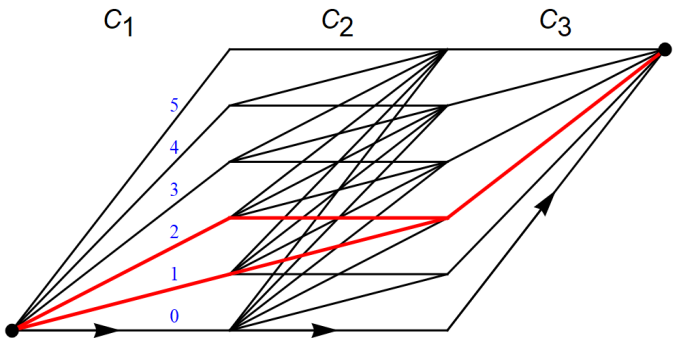
In this section, we shall once again encounter the optimality principle that underlies Dijkstra's algorithm for finding shortest paths in a weighted graph, and the labelling of an activity network to find critical paths.

**Example 9.32.** Deborah has £50K to invest in multiples of £10K in three companies $C_1, C_2, C_3$, and wants to maximise the return. All the money is to be used, but she is not allowed to invest more than once in any company. The following table shows the expected returns on investment, with the amount invested along the top row. All numbers are in units of £10K.

| | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| $C_1$ | 0 | 3 | 5 | 6 | 7 | 8 |
| $C_2$ | 0 | 2 | 4 | 8 | 10 | 11 |
| $C_3$ | 0 | 0 | 2 | 10 | 11 | 11 |

The problem amounts to finding an aligned path in the following network that runs from bottom left to top right with *maximum* total weight. Although only three arrows are shown, all the arcs are oriented from left to right, and this is akin to the activity network, in which duration is replaced by financial return.

The first stage consists of deciding how to invest in $C_1$, the second how much to invest in $C_2$. The difference has to be invested in $C_3$: if $x_1$ units are invested in $C_1$ and $x_2$ units in $C_2$ then $0 \leqslant x_2 \leqslant 5 - x_1$, and $x_3 = 5 - x_1 - x_2$ units are invested in $C_3$:




With reference to the graph, each event has coordinates $(i, y_i)$ where $0 \leqslant i \leqslant 3$ and $0 \leqslant y_i \leqslant 5$. At this event, one has concluded a total investment of $y = y_i$ in companies up to and including $C_i$. There are

$$6 + 5 + 4 + 3 + 2 + 1 = \tfrac{1}{2} * 6 * (6 + 1) = 21$$

aligned paths from start $(0, 0)$ to finish $(3, 5)$, but the problem can easily be generalized to more companies and investment choices.

Our solution below reveals that there are in fact two longest paths (shown red with a final arc in common) realizing a total return of £150K.

The special feature of this network is that although there are $6 + 21 + 6 = 33$ arcs, there are only 18 different weights. These are determined by the functions

$$r_i(x) = \text{return on investment of } x \text{ units in } C_i$$

from the table, with $0 \leqslant x \leqslant 5$. From these, we shall construct functions

$$f_i(y) = \text{ best return at the } i\text{th stage for a total investment } y,$$

where 'best' is taken over all possible investment strategies $x_1, x_2, \ldots, x_i$ up to the $i$th stage, and $y$ denotes the sum $x_1 + \cdots + x_i$. Our aim is to find $f_3(5)$.

Fortunately we do not need to consider all choices. The optimality principle implies that the best investment (which is a *longest* path) at each stage will *necessarily* arise from a best investment (longest path) at all previous stages. Hence the

**Corollary.** The 'best return' function satisfies

$$f_i(y) = \max_{0 \leqslant x \leqslant y} \left\{ f_{i-1}(y - x) + r_i(x) \right\} \quad \text{or} \quad f_i(y_i) = \max_{x_i} \left\{ f_{i-1}(y_{i-1}) + r_i(x_i) \right\},$$

with $f_0 = 0$. In practice, it may help to use the second equation in which the values of $x = x_i$ and $y = x_1 + \cdots + x_i$ at each stage make the discrete nature of $y$ more explicit.

The underlying logic of this formula is identical to that of the expression

$$E(u) = \max_{x \to u} \Big\{ E(x) + d(x,u) \Big\}$$

in §9.1 for finding latest start times. This is in turn a version of relabelling formula

$$L_j = \min(L_i + d(v_i, v_j),\ L_j)$$

in §8.1 to relax the temporary labels in Dijkstra's algorithm, 'min' here because we were seeking a *shortest* path.

In dynamic programming, the graph overcomplicates the situation, so the work is best organized into a table, which is headed by the values of $y$ in (traditionally) reverse order. There is really a separate table for each stage, which applies the corollary to determine $f_i(y_i)$, though the tables can be joined together. Strictly speaking, *every* stage needs a triangular table to cater for all the combinations of $x = x_i$ and $y_i = x_1 + \cdots + x_i$. However, the first and last are simpler, and in our example only the second one illustrates the general technique (this is apparent from the structure of the graph!).

| | | 5 | 4 | 3 | 2 | 1 | 0 | $\leftarrow y$ |
|---|---|---|---|---|---|---|---|---|
| | | 8 | 7 | 6 | 5 | 3 | 0 | $\leftarrow f_1(y_1)$ |
| $x_2\downarrow$ | $r_2(x_2)\downarrow$ | | | | | | | |
| 5 | 11 | | | | | | 11 | |
| 4 | 10 | | | | | 13 | 10 | |
| 3 | 8 | | | | 13 | 11 | 8 | $\leftarrow f_1(y_1) + r_2(x_2)$ |
| 2 | 4 | | | 10 | 9 | 7 | 4 | |
| 1 | 2 | | 9 | 8 | 7 | **5** | 2 | |
| 0 | 0 | 8 | 7 | 6 | **5** | 3 | 0 | |
| | | 13 | 11 | 8 | 5 | 3 | 0 | $\leftarrow f_2(y_2) = \max \text{in } \Delta$ |
| | | 13 | 11 | 10 | **15** | 14 | 11 | $\leftarrow f_2(y_2) + r_3(x_3)$ |
| | | 15 | | | | | | $\leftarrow f_3(y_3) = \max$ |

The first stage is straightforward: we set $x = x_1 = y$ and $f_1(y_1) = r_1(x_1)$.

At the second stage, for each value of $y$, we are allowed to choose any value of $x = x_2$ with $y + x_2 \leqslant 5$, and for each such value we add $r_2(x_2)$ to our return. The total investment $y + x_2$ is constant along each diagonal $\Delta$, which we scan in order to find the maximum return. For example, the diagonal returns $7, 8, 9, 11, 10$ is associated to a total inestment of 4 units, and its best return is

$$f_2(4) = \max_{0 \leqslant x \leqslant 4} \Big\{ f_1(4-x) + r_2(x) \Big\} = 11.$$

There are no entries above the main diagonal since those would represent current total investments of over £50K.

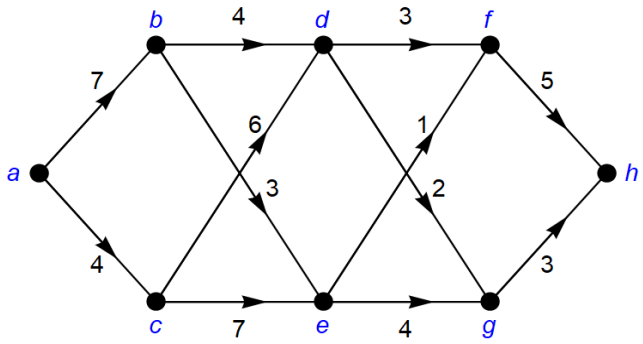
As we pass to the next stage, we associate the maximum return to the new value $y_i = y_{i-1} + x_i$, and place it under the bottom-left entry of the diagonal. In theory, we are ready for another triangular table, but in our example, there is *no choice left* since $x_3$ is determined by $x_1 + x_2$. Thus, there would only be one useful diagonal, which we have shown horizontally to save space.

The best return has value 15 coming from $x_3 = 3$, which in turn comes from an earlier best return of 5 with *either* $x_2 = 2$ *or* $x_2 = 1$. (These entries have been shown in bold.)

*Conclusion.* Wise investment produces a best possible return of £150K, arising in two ways: (i) £20K in $C_1$ plus £30K in $C_3$, or (ii) £10K in $C_1$ and $C_2$ plus £30K in $C_3$. These two solutions are the red paths, but the return is not good enough for our Dragon.

Dynamic programming is really a BFS with 'pruning' – one can forget results from previous stages. Here is an example in which one can compare the technique to Dijkstra's algorithm, though in this case the latter is probably quicker.

**Example 9.33.** Find the shortest path from $a$ to $h$ in the weighted digraph below:




Dijkstra's algorithm will work provided one takes account of the fact that the edges now directed; for example, $d$ is adjacent to $b$ but not vice versa, so one only scans from left to right. This problem is therefore layered like the investment one – for each vertex all paths from the start have the same length. The solution can be again be obtained in 'vertical' stages. The key point in problems like these is that *the best solution at the* $(i+1)$*th stage must arise from the best solutions at the* $i$*th stage.*

The shortest path can be obtained from the following table:

| start at: | $a$ | |
|---|---|---|
| | dist=0 | |
| to get to: | $b$ | $c$ |
| | dist =7 from $a$ | dist =4 from $a$ |
| to get to: | $d$ | $e$ |
| | best dist =10 from $c$ | best dist =10 from $b$ |
| to get to: | $f$ | $g$ |
| | best dist =11 from $e$ | best dist =12 from $d$ |
| to get to: | $h$ | |
| | dist=15 from $g$ | |

**Exercise 9.34.** *Suppose that the weights in the above network now represent capacities, $a$ is the source and $h$ is the sink. Show that the value of a maximum flow is 7 and identify a minimum cut.*

# 10 Cryptography

## 10.1 Euler's totient function

Let $n \geqslant 1$ be an integer, and recall the set of residue classes

$$R = \mathbb{Z}/n\mathbb{Z} = \{\bar{x} : x \in \mathbb{Z}\} = \{x + n\mathbb{Z} : x \in \mathbb{Z}\}.$$

In Section 1.3, we saw that $R$ is a ring under the following rules:

(a) $\bar{x} + \bar{y} = \overline{x+y}$, i.e. $(x + n\mathbb{Z}) + (y + n\mathbb{Z}) = (x + y) + n\mathbb{Z}$;

(b) $\bar{x} * \bar{y} = \overline{x*y}$, i.e. $(x + n\mathbb{Z}) * (y + n\mathbb{Z}) = (x * y) + n\mathbb{Z}$.

**Definition 10.1.** *Let $n \geqslant 1$ be an integer, and $x \in \mathbb{Z}$. We say that the residue class $\bar{x}$ is* unit *in $\mathbb{Z}/n\mathbb{Z}$ if there exists $y \in \mathbb{Z}$ such that $\bar{x} * \bar{y} = \bar{1}$. We denote the group of units in $\mathbb{Z}/n\mathbb{Z}$ by $(\mathbb{Z}/n\mathbb{Z})^*$ (or simply $(\mathbb{Z}/n\mathbb{Z})^\times$.*

The following result characterizes units in residue rings.

**Theorem 10.2.** *Let $n \geqslant 1$ be an integer and $x \in \mathbb{Z}$. Then, $\bar{x}$ is a unit if and only if there exists $x$ is coprime with $n$, which is the same as saying that* $\gcd(x, n) = 1$.

*Proof.* Take $x \in \mathbb{Z}/n\mathbb{Z}$. Then, we have

$$\begin{aligned}
\bar{x} \text{ is a unit in } \mathbb{Z}/n\mathbb{Z} &\iff \exists y \in \mathbb{Z} \text{ such that } x * y \equiv 1 \bmod n \\
&\iff \exists y \in \mathbb{Z} \text{ such that } n \mid (x * y - 1) \\
&\iff \exists y \in \mathbb{Z} \text{ such that } x * y - 1 = a * n \text{ for some } a \in \mathbb{Z} \\
&\iff \exists y \in \mathbb{Z} \text{ such that } x * y - a * n = 1 \text{ for some } a \in \mathbb{Z} \\
&\iff \exists y \in \mathbb{Z} \text{ such that } \gcd(x, n) = 1.
\end{aligned}$$

□

**Definition 10.3.** *The function* $\varphi : \mathbb{N} \to \mathbb{N}$*, defined by*

$$\varphi(n) := |(\mathbb{Z}/n\mathbb{Z})^*|,$$

*is called* Euler's totient function.

**Theorem 10.4.** *The Euler totient function satisfies the following properties:*

*(a) For every positive integer* $n \in \mathbb{Z}$*, we have*

$$\varphi(n) = n \prod_{p|n} \left(1 - \frac{1}{p}\right).$$

*(b) For every positive integers* $m, n$ *such that* $\gcd(m, n) = 1$*, we have*

$$\varphi(m * n) = \varphi(m) * \varphi(n).$$

**Example 10.5.**

$$\varphi(39) = 39\left(1 - \frac{1}{3}\right)\left(1 - \frac{1}{13}\right) = 24;$$
$$\varphi(100) = 100\left(1 - \frac{1}{2}\right)\left(1 - \frac{1}{5}\right) = 40.$$

**Theorem 10.6** (Euler)**.** *Let* $n \geqslant 1$ *be an integer, and* $x \in \mathbb{Z}$*. Then, we have*

$$\gcd(x, n) = 1 \implies x^{\varphi(n)} = 1 \ mod\ n.$$

*Proof.* The set $(\mathbb{Z}/n\mathbb{Z})^*$ is a group of order $\varphi(n)$ by definition of Euler's totient function. By Lagrange's theorem, the order of any element divides the size of this group, so $x^{\varphi(n)}$ is the identity in $\mathbb{Z}/n\mathbb{Z}$. □

## 10.2 Introduction

Human beings have always had a constant desire to protect information or secrets from others. There are plenty of famous historical examples where people tried to keep secrets from adversaries. Kings and generals have communicated with their troops using basic cryptographic methods to prevent enemies from getting access to sensitive military information. In fact, it is often reported that Julius Caesar used a simple cipher, which now carries his name.

As society has evolved, the need for more sophisticated methods to protect data has increased. Now, with the information era at hand, the need is more pronounced than ever. As the world becomes more connected, the demand for more information and electronic services is growing, and with increased demand comes increased dependency on electronic systems. Already, the exchange of sensitive information, such as credit card numbers, over the Internet is common practice. Protecting data and electronic systems is crucial to our ways of life. The techniques needs to protect data belong to the field of cryptography.

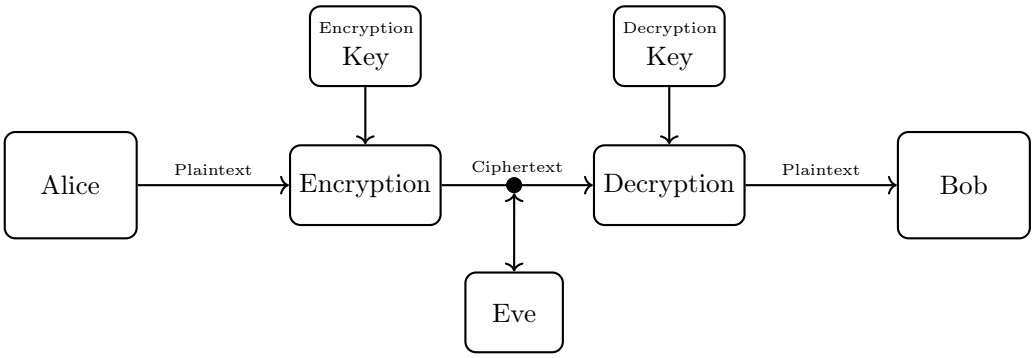



Figure 18: Basic communication scenario for cryptography

## 10.3 Basic setup

We have the following basic scenario: Alice wants to send a secret message, the *plaintext*, to Bob. She does this by *encrypting* the message into a *ciphertext* using a secret *key*, and transmitting the message over a non-secure channel. When Bob receives the message, he *decrypts* it using the key. Alice and Bob must agree on their keys (in secret). There is an eavesdropper, called *Eve*, attempting to find the plaintext.

To formalise the basic scenario in Figure 18, let $\mathcal{P}$ be the *plaintext alphabet*, the set from which the plaintext is written. E.g. $\mathcal{P} = \{\mathrm{A}, \mathrm{B}, \ldots, \mathrm{Z}\}$. Let $\mathcal{C}$ denote the *ciphertext alphabet*; sometimes we have $\mathcal{C} = \mathcal{P}$. Let $\mathcal{K}$ denotes the set of possible keys. The *encryption function* is a function

$$\begin{aligned} E : \mathcal{K} \times \mathcal{P} &\to \mathcal{C} \\ (k, x) &\mapsto E_k(x). \end{aligned}$$

Likewise, the *decryption function* is a function

$$\begin{aligned} D : \mathcal{K} \times \mathcal{C} &\to \mathcal{P} \\ (k, x) &\mapsto D_k(x). \end{aligned}$$

For all $x \in \mathcal{P}$ and all $k \in \mathcal{K}$, we want that $D_k(E_k(x)) = x$.

$$\mathcal{P} \xrightarrow{E_k} \mathcal{C} \xrightarrow{D_k} \mathcal{P}, \qquad \mathcal{P} \xrightarrow{id} \mathcal{P}$$

**Example 10.7** (Caesar's cipher)**.** One of the earliest ciphers is due to Julius Caesar. To encrypt a message using Caesar's cipher, we simply shift each letter by three places forward. To decrypt, we simply shift three letter back.

We view this mathematically as follows. Let $\mathcal{P} = \{0, 1, \ldots, 25\} = \mathbb{Z}/26\mathbb{Z}$, and put $\mathcal{P}$ in correspondence with $\{A, B, \ldots, Z\}$ by

$$0 \leftrightarrow A,\ 1 \leftrightarrow B,\ \ldots,\ 25 \leftrightarrow Z.$$

Take $\mathcal{C} = \mathbb{Z}/26\mathbb{Z}$. Then the encryption function for the Caesar cipher simply adds 3 in $\mathbb{Z}/26\mathbb{Z}$:

$$\begin{aligned} E : \mathcal{P} &\to \mathcal{C} \\ x &\mapsto x + 3, \end{aligned}$$

and the decryption function is

$$\begin{aligned} D : \mathcal{C} &\to \mathcal{P} \\ y &\mapsto y - 3. \end{aligned}$$

We want to encrypt the message "VENI, VIDI, VICI" with the Caesar cipher. First, we get rid of spaces (as they yield so much information on the structure of the message that decryption becomes easier) to get "VENIVIDIVICI". Looking up these letters in our plaintext alphabet $\mathcal{P}$, we get the sequence

$$21, 4, 13, 8, 21, 8, 3, 8, 21, 8, 2, 8.$$

By adding 3 in $\mathbb{Z}/26\mathbb{Z}$, we get

$$24, 7, 16, 11, 24, 11, 6, 11, 24, 11, 5, 11.$$

By identifying this numbers with the corresponding letters, we get "YHQLYLGLYLFL".

## 10.4 Vigenère cipher

The *Vigenère cipher* is a variation of the shift cipher. It is attributed to the sixteenth century French cryptographer Vigenère. However, many believe that Vigenère own ciphers were more sophisticated. This cipher was thought to be unbreakable until the nineteenth century: "Le chiffre indéchiffrable."

First, choose a key length $m = 6$, say. Then choose a vector of length $m$ from $\mathbb{Z}/26\mathbb{Z}$, e.g. $k = (21, 4, 2, 19, 14, 17)$. To encrypt the message, we shift the first letter by 21, the second by 4, and so on, then the seventh by 21, and so on.

Here the key space for length $m$ is

$$\mathcal{K} = (\mathbb{Z}/26\mathbb{Z})^m = \{(k_1, \ldots, k_m) : k_i \in \mathbb{Z}/26\mathbb{Z}\}.$$

We also have $\mathcal{P} = \mathcal{C} = \mathcal{K}$. The encryption map is

$$\begin{aligned} E_k : \mathcal{P} &\to \mathcal{C} \\ (x_1, \ldots, x_m) &\mapsto (x_1 + k_1, \ldots, x_m + k_m). \end{aligned}$$

The decryption map is

$$\begin{aligned} D_k : \mathcal{C} &\to \mathcal{P} \\ (y_1, \ldots, y_m) &\mapsto (y_1 - k_1, \ldots, y_m - k_m). \end{aligned}$$

**Example 10.8.** The encryption of the sentence "Divert troops East" using the key word "White" is as follows:

| | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Plaintext | DIVERTTROOPSEAST | 3 | 8 | 21 | 4 | 17 | 19 | 19 | 17 | 14 | 14 | 15 | 18 | 4 | 0 | 18 | 19 |
| Keyword | WHITEWHITEWHITEW | 22 | 7 | 8 | 19 | 4 | 22 | 7 | 8 | 19 | 4 | 22 | 7 | 8 | 19 | 4 | 22 |
| Ciphertext | ZPDXVPAZHSLZMTWP | 25 | 15 | 3 | 23 | 21 | 15 | 0 | 25 | 7 | 18 | 11 | 25 | 12 | 19 | 22 | 15 |

The security of the Vigenère cipher depends on the fact that the key and its length are unknown. There is also another additional advantage over a simple shift cipher in that a letter can be encoded in different ways. In Example 10.8, both the letters "I" and "T" encrypted as "P". Up until the 20th century, one needed a Vigenère table in order to do the encryption.

## 10.5 The RSA algorithm

Until the 1970's encrypting messages required both sender and receiver to use the same key ('codebook') to encrypt and decrypt. Such crypto systems are known as *symmetric key* crypto systems. The use of such crypto systems is not practical for interchange of secret data on the internet. The concept that led to the introduction of all modern forms of cryptography is that of an *asymmetrical key* crypto systems based on *trapdoors*, in the terminology of a famous paper by Diffie & Hellman (1976). A *trapdoor* is a mathematical function $f : X \to Y$ such that it is easy to compute $y = f(x)$, given $x \in X$, but *hard* to solve $y = f(x)$ for $x \in X$, given $y \in Y$. In other words, it is hard to invert $f$ without using extra information. This idea was successively implemented in the celebrated RSA algorithm, named after Rivest, Shamir & Adleman, who discovered it in April 1977 and patented it that year.

Years afterwards, it was revealed that the same algorithm had been described by Clifford Cocks in a GCHQ memo in 1973, working with James Ellis, who had already conceived of the trapdoor mechanism.

In RSA, one chooses:

- Two *large* primes $p \neq q$ and form $n = pq$, the *modulus*;
- An integer $2 \leqslant e < \varphi(n)$ such that $\gcd(e, \varphi(n)) = 1$ and compute $2 \leqslant s < \varphi(n)$ such that $es = 1 \bmod \varphi(n)$; $e$ is called the *encryption* or *public exponent* and $s$ the *decryption* or *secret exponent*.

The pair $k^p = (n, e)$ is called the *public key*, and the pair $k^s = (n, s)$ is called the *secret key*. One must keep $p, q$ and $s$ secret at all time. The plaintext and ciphertext alphabets are the same $\mathcal{P} = \mathcal{C} = \mathbb{Z}/n\mathbb{Z}$, and the key space is $\mathcal{K} = (\mathbb{Z}/\varphi(n)\mathbb{Z})^\times$. The encryption function is

$$\begin{aligned} E : \mathcal{P} \times \mathcal{K} &\to \mathcal{C} \\ (m, e) &\mapsto E_e(m) = m^e \bmod n. \end{aligned}$$

The decryption function is

$$\begin{aligned} D : \mathcal{C} \times \mathcal{K} &\to \mathcal{P} \\ (c, s) &\mapsto D_s(c) = c^s \bmod n. \end{aligned}$$

**Proposition 10.9.** *$D_s(E_e(m, e), s) = m$ for all $m \in \mathcal{P}$ and $e \in \mathcal{K}$.*

*Proof.* This follows from Fermat's little Theorem (Theorem 1.16) or Euler's Theorem 10.6. □

From Proposition 10.9, to send the plaintext $m$ to Bob, Alice does the following:

1. Compute $c = m^e \bmod n$;

2. Send the ciphertext $c$ to Bob.

To decipher the message $c$ sent by Alice, Bob simply computes

$$c^s = (m^e)^s = m^{es} = m \bmod n,$$

by Fermat's little Theorem or Euler's Theorem. And hence, Bob recovers the plaintext $m$ sent by Alice. Bob can do these computations since he's the only one to know $s$, $p$ and $q$. Eve also knows $e, c = m^e$, and $n$, but doesn't know $m$. Her challenge is to recover $m$ from $c$ and $k^p = (n, e)$.

**Example 10.10.** Suppose that $p = 29$, $q = 53$ and $n = 29 \cdot 53 = 1537$, and Bob's public key is $k^p = (n, e) = (1537, 3)$. Then, $\varphi(n) = (p-1)(q-1) = 1456$. The decryption key is $k^s = (n, s)$, where $s$ is the multiplicative inverse of $e$ modulo $\varphi(n)$. The Extended Euclid Algorithm yields $s = 971$ and $(n, s) = (1537, 971)$. The public key $(1537, 3)$ is published by Bob. Assume that Alice wants to send the message $m = 101$ to Bob. She can use Bob's public key to encrypt it as

$$c = E_e(m) = m^e \bmod n = 101^3 \bmod 1537 = 511 \bmod 1537.$$

She then sends $c = 511$ to Bob. Bob decrypts as

$$D_s(c) = c^s \bmod n = 511^{971} \bmod 1537 = 101 = m.$$

**Example 10.11.** In our second example, Bob chooses

$$p = 1999, \quad q = 2029,$$

so the 'key length' equals

$$n = pq = 4\ 055\ 971,$$

and

$$\phi = \varphi(n) = (p-1)(q-1) = 4\ 051\ 944.$$

Bob takes

$$e = 5$$

so as to make it easy for Alice for work out $x^e$ with her primitive calculator. It is obviously coprime to $\phi$; indeed choosing $e$ to be a prime obviates the need to check that $\gcd(e, \phi) = 1$. In addition $\phi + 1$ is a multiple of 5, and in fact

$$\phi + 1 = 5 * 810389,$$

so Bob's private key is $s = 810389$. He considers swapping $s$ with $e$ but decides against it.

Alice's message $x$ is in fact only 3 digits long, nonetheless $x^e$ is about $996 * 10^9$, just less than one trillion. But she did the modular calculation almost by hand:

$$\begin{aligned} c = 251^5 &= 251 * (251^2)^2 \bmod n \\ &= 251 * (63001)^2 \bmod n \\ &= 251 * (3\ 969\ 126\ 001) \bmod n \\ &= 251 * (2\ 386\ 363) \bmod n \\ &= 2\ 749\ 376 \bmod n. \end{aligned}$$

Bob now uses a computer to discover that

$$c^s \bmod n = 251,$$

so Alice had encrypted her module code.

## 10.6 Security of RSA

The security of RSA relies on the assumption that factoring (big) composite integers is hard. If Eve knows $p$ and $q$, then she knows $\varphi(n) = (p-1)(q-1)$, and she can easily compute the private exponent $s$ such that $se \equiv 1 \bmod \varphi(n)$. Moreover, given $n = pq$ and a multiple of $\operatorname{lcm}(p-1, q-1)$, one can compute $p$ and $q$. Indeed, suppose we are given $\varphi(n) = (p-1)(q-1) = pq - (p+q) + 1$. Then, we can recover $p, q$ as the roots of the quadratic polynomial

$$(X-p)(X-q) = X^2 - (n+1-\varphi(n))X + n.$$

So one must keep $\varphi(n)$ secret as well.

In practice, $p$ and $q$ should have a similar length but differ by a few powers of 10. Prime numbers can be found using primality tests, like a probabilistic version of one we shall consider briefly in §10.6. The effectiveness of the algorithm in this respect cannot be *proved* mathematically, and there is a serious concern from experts that within a couple of decades quantum computers could crack the current public keys.

## 10.7 The efficiency of RSA

In current RSA schemes, $n$ has 2048 bits typically, i.e. more that 600 decimal digits. So how do we compute $a^e \bmod n$ efficiently? If $n \approx 10^{200}$ and $e \approx 10^{50}$, $a = 2$, we couldn't even write down $a^e = 2^{10^{50}}$. Even working in $\mathbb{Z}/n\mathbb{Z}$ (i.e. modulo $n$) would still require $e$ multiplications. So to compute $a^e$, we use fast exponentiation as we saw in previous sections.

When $e = 2^i$, then $a^e = a^{2^i}$ can be computed using $i$ squaring in $\mathbb{Z}/n\mathbb{Z}$; this requires $C(\log n)^2 i = C(\log e)(\log^2 n)$ basic arithmetic operations, where $C > 0$ is a constant. For general $e$, write $e = (b_k b_{k-1} \dots b_0)_2$ and $k = \lfloor \log_2 e \rfloor$. Then, we have

$$a^e = \prod_{\substack{i=0 \\ b_i = 1}}^{k} a^{2^i}.$$

The number of squaring is $k = \lfloor \log_2 e \rfloor = C \log e$, and the number of multiplications required is $\#\{i : b_i = 1\} - 1 = C \log_2 e$. So to compute $a^e \bmod n$ requires $C(\log e)(\log^2 n)$ basic arithmetic operations, where $C > 0$.

In practice, one choose the public exponent $e$ to be short, and the private exponent $s$ has full length. In fact, a short exponent $s$ would be insecure. To perform the decryption, one combines fast exponentiation with the Chinese Remainder Theorem.

A popular choice of $e$ is the Fermat prime $2^{16} + 1 = 65537$. In practice, the numbers $p, q, e, s$ will be converted to base 2, and then divided into 64-bit blocks. These are then displayed using the 64 characters `A...Z`, `a...z`, `0...9`, `+` `/` as well as `=` and a series of check digits.

The RSA algorithm can be also be used to *authenticate* the sender of ciphertext by providing a digital signature linked to the message, and to enable *non-repudiation* – Alice can't deny she was the author of a command send to Bob.

A list of public keys in a typical `known _hosts` folder reveals a mix of 'ssh-rsa' and 'ecdsa-sha2-nistp256' algorithms. The latter are all based on the elliptic curve

$y^2 = x^3 - 3x + 41058363725152142129326129780047268409114441015993725554835256314039467401291$
whose study belongs to the realm of number theory and geometry.

## 10.8 Miller's test

We have seen that the RSA is based on the principle that factoring *large* composite integers is hard. Current RSA algorithms use integers $n$ that that have around 600 decimals. So, it is natural to ask, how does one find *big* prime numbers? How does one know whether a *big* number is *prime*?

To determine whether a given integer $n$ is prime, one uses a *primality test*. There are *deterministic* and *probabilistic* primality tests. A deterministic primality test is an algorithm which decides whether a given number $n$ is a prime, while a probabilistic primality test is an algorithm decides whether there is a *high probability* that a given number is prime.

In practice, probabilistic primality tests are faster than deterministic ones, and for everyday cryptographic purposes probabilistic tests suffice: if the probability that my banking application is corrupted is less than $10^{-1000}$, I shouldn't be worried.

In this section we present the so-called Miller-Rabin primality test. It is a probabilistic one. There are other primality tests but we will not see them.

Let $n = 217$. It is patently obvious that $n$ is divisible by $7$. Indeed, $n = 7 * 31$.

Observe that $n - 1$ is divisible by $2^3$, so set $k = 2^j * 27$ with $0 \leqslant j \leqslant 3$. The following table displays the values of $a^k \bmod n$, for $1 \leqslant a \leqslant 10$ in the range $[-108, 108]$:

| $a$ | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|
| $k = 27$ | 1 | -27 | -8 | 78 | -92 | -1 | -77 | 64 | 64 | 97 |
| $k = 54$ | 1 | 78 | 64 | 8 | 1 | 1 | 70 | -27 | -27 | 78 |
| $k = 108$ | 1 | 8 | -27 | 64 | 1 | 1 | -91 | 78 | 78 | 8 |
| $k = 216$ | 1 | 64 | 78 | -27 | 1 | 1 | 35 | 8 | 8 | 64 |

If $n$ were prime, by Fermat's little theorem we would have $a^{n-1} = 1 \bmod n$ if $0 < a < n$ and so the last row would be all 1's. Moreover, if $n$ is prime and $n - 1 = 2m$ then $x = a^m$ satisfies $x^2 = 1 \bmod n$, i.e. $(x - 1)(x + 1) = 0 \bmod n$, and $n$ must divide $x - 1$ or $x + 1$ so either $a^m = 1 \bmod n$ or $a^m = -1 \bmod n$. Continuing in this way establishes one implication of the following proposition.

**Proposition 10.12.** *Let $n > 1$ be an odd integer. Write $n - 1 = 2^k * u$ with $u$ odd. Then the following statements are equivalent.*

*(i) $n$ is a prime number.*

*(ii) For all $a \in (\mathbb{Z}/n\mathbb{Z})^*$ one of the following statements holds:*

- *$a^u \equiv 1 \bmod n$, or*
- *there is $0 \leqslant i \leqslant k - 1$ such that $a^{2^i * u} \equiv -1 \bmod n$.*

*Proof.* Since $n-1=2^k u$ with $u>1$ odd, $k\geqslant 1$, we can write

$$\begin{aligned} a^n-a &= a(a^{n-1}-1)=a(a^{(n-1)/2}+1)(a^{(n-1)/2}-1)\\ &= a(a^{(n-1)/2}+1)(a^{(n-1)/4}-1)(a^{(n-1)/4}+1)\\ &\quad\cdots\\ &= a(a^u-1)\prod_{i=0}^{k-1}(a^{2^i u}+1). \end{aligned}$$

So, by forgetting the intermediate equalities, we get

$$a^n-a=a(a^{n-1}-1)=a(a^u-1)\prod_{i=0}^{k-1}(a^{2^i u}+1). \tag{7}$$

If $n$ is an odd prime, then by Fermat's Little Theorem, $a^n-a=0\in\mathbb{Z}/n\mathbb{Z}$, so we have at least one of the following:

$$\begin{aligned} a &\equiv 0 \bmod n,\\ a^u &\equiv 1 \bmod n, \text{ or}\\ a^{2^i * u} &\equiv -1 \bmod n, \text{ for } i=0,1,\ldots,k-1. \end{aligned}$$

From this, it follows that (i) $\Rightarrow$ (ii).

□

**Example 10.13.** Let $n=561$. Then, $n-1=560=16*35=2^4*5*7$. Let $a=2$. Then

$$\begin{aligned} b_0 &= 2^{35}=263 \bmod 561,\\ b_1 &= b_0^2=166 \bmod 561,\\ b_2 &= b_1^2=67 \bmod 561,\\ b_3 &= b_2^2=1 \bmod 561. \end{aligned}$$

Since $b_3\equiv 1 \bmod 561$, we conclude that $561$ is composite. Moreover, $\gcd(b_2-1,561)=33$ is a non-trivial factor of $561$. Indeed, one can verify that $561=3*11*17$.

Proposition 10.12 leads to the following algorithm.

**Algorithm 10.14** (Miller-Rabin primality test)**.**
**Input:** $n\geqslant 3$ *an integer.*
**Output:** `probably prime` *or* `composite`.

*(1) If $n$ is divisible by 2, 3 or 5, then return* `composite` *and stop.*

*(2) Compute $n-1=2^k*u$ with $u\in\mathbb{N}$ odd.*

*(3) Choose $a\in(\mathbb{Z}/n\mathbb{Z})^*$ $(a\neq 1)$.*

*(4) Calculate $b=a^u \bmod n$.*

*(5) If $b=1 \bmod n$, return* `probably prime`.

(6) *Loop* $i = 0, \ldots, k-1$*:*

- *If* $b^2 = -1 \bmod n$*, return* `probably prime`*.*
- *Otherwise, set* $b = b^2 \bmod n$*.*

(7) *Increment* $i$*, and return to Step 6.*

(8) *Return* `composite`*.*

We need the following definition in order to discuss the accuracy of the Miller-Rabin primality test.

**Definition 10.15.** *Let* $n > 1$ *be an odd integer such that* $n - 1 = 2^k * u$*, with* $u$ *odd. An odd integer* $n > 1$ *such that* $\gcd(a, n) = 1$ *is called a* (Miller-Rabin) witness *to the compositeness of* $n$*, if the following two conditions are true:*

(a) $a^u \not\equiv 1 \bmod n$*; or*

(b) $a^{2^i * u} \not\equiv -1 \bmod n$*, for* $i = 0, 1, \ldots, k-1$*.*

The following theorem states that the Miller-Rabin test has a very high probability of deciding whether a given integer is composite.

**Theorem 10.16.** *Let* $n \in \mathbb{Z}_{>1}$ *be odd and composite. Then, we have*

$$\frac{\#\{a : 1 \leqslant a \leqslant n-1, a \text{ is a witness for } n\}}{n-1} \geqslant \frac{3}{4}.$$

So a random $a$ has $\geqslant 75\%$ chance predicting that $n$ is composite. So if you run the compositeness test with $r$ random choices for $a$, and $n$ passes, the probability that $n$ is prime is $\geqslant 1 - \frac{1}{4^r}$.

In most circumstances, we are happy with such a probable prime. So the Miller-Rabin test is a very good probabilistic primality test.

**Example 10.17.** Let $n = 172947529$. Then $n - 1 = 2^3 * 21618441$. Let $a = 17$. Then, we find that

$$17^{21618441} = 1 \bmod 172947529.$$

So 17 is *not* a Miller-Rabin witness. Next, letting $a = 3$, we have

$$3^{21618441} = -1 \bmod 172947529.$$

At this point, one might suspect that $n$ is prime. But if we try another value, say $a = 23$, we find that

$$\begin{aligned} 23^{21618441} &= 40063806 \bmod 172947529, \\ 23^{2*21618441} &= 2257065 \bmod 172947529, \\ 23^{4*21618441} &= 1 \bmod 172947529. \end{aligned}$$

Thus 23 is a Miller-Rabin witness and $n$ is actually a composite. In fact, $n$ is called a Carmichael number, but it is not easy to factor by hand: $n = 307 * 613 * 919$.

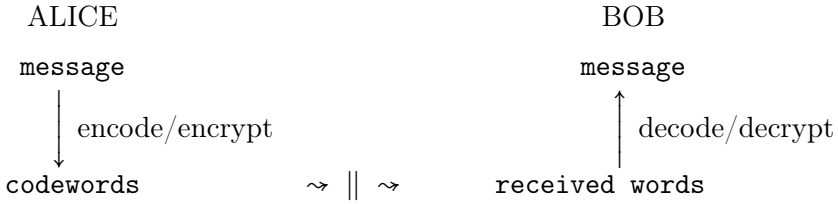
**Exercise 10.18.** *Determine whether* 353 *passes Miller's test to base* 5*. [Answer: Yes, since* $352 = 2 * 176 = 2^5 * 11$ *and one finds that* $5^{176} = -1$ *mod* 353*. In fact,* 353 *is prime so it must pass to any base!]*

# 11 Codes*

When sending a message, one might wish to:

- transmit it efficiently – achieved with error-correcting codes;
- keep the message private and authenticate the sender – the role of cryptography.

Here is the set-up to keep in mind:




We shall consider codes first. Bob might receive

`104727 IS A MEMORABLE QRIME`

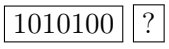
but the underlined characters are not what Alice wrote. The second error needs both a dictionary (or spell-checker) and a realization that the number is not salty, not criminal, not dirty, nor is it 3 cents. The first needs an analysis of primes that differ 'minimally' from 104727 (which is itself divisible by 3)

## 11.1 Check digits

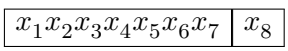
The idea here is to assign a check digit or digits to each block of numerical data:

The check should involve the whole block, but ideally be small compared to it and easy to compute. It should be able to detect when a common error has occurred, even if it will not correct the error. Examples of some common systems follow.

*Parity bit.* Each block might consist of 7 bits, to which one check bit is added. For example




where '?' is chosen so that the overall number of 1's is even. Here it is 1. More generally




must satisfy $\sum_{i=1}^{8} x_i = 0 \bmod 2$. This system defines a set of $2^7 = 128$ code words requiring 8 bits for transmission. It will succeed in detecting an error if exactly one digit $x_1, \ldots, x_7$ is transcribed wrongly, but it does not 'see' the transposition of two bits.

*ISBN 10 (International Standard Book Number, pre 2007).* This is a 10-digit number, which we can type as $x_1x_2x_3x_4x_5x_6x_7x_8x_9x_{10}$, in which the last digit $x_{10}$ is a check. For a number to be valid

$$x_1 + 2x_2 + 3x_3 + \cdots + 9x_9 + 10x_{10} = 0 \bmod 11.$$

The check digit is easily determined by the formula

$$x_{10} = -10x_1 = x_1 + 2x_2 + 3x_3 + \cdots + 9x_9 \bmod 11.$$

If $x_{10} = 10 \bmod 11$, the character 'X' is used. For example, the ISBN 10 number of Iris Murdoch's novel "The Sea, the Sea" (the 1978 Booker prizewinner) is

`014118616X`.

ISBN 10 detects the two commonest errors:

(i) a single wrong digit, like 3491234287 instead of 3491242287.

(ii) a single adjacent transposition, like 3491224287 instead 3491242287.

Let's verify (i). Suppose that the original 10-digit 'word' is $\mathbf{x} = x_1x_2x_3\cdots x_{10}$, but that this is received as $\mathbf{y}$ with a error in (say) the second position: $\mathbf{y} = x_1y_2x_3\cdots x_{10}$. Set

$$f(\mathbf{x}) = \sum_{i=1}^{10} i\,x_i,$$

so that $f(\mathbf{x}) = 0 \bmod 11$. Then

$$f(\mathbf{y}) = f(\mathbf{x}) + 2(y_2 - x_2) = 2(y_2 - x_2) \bmod 11$$

can't be zero because 11 is prime.

*IBAN (International Bank Account Number).* The IBANs of a given country have the same number of digits: for example, the UK and Germany have 22, whilst France and Italy have 27. Here is a UK example:

`GB27 LOYD 3011 2700 1268 86`

The two digits (here, 27) after the country code form the check. In general, these two digits represent an integer $c$ satisfying $2 \leqslant c \leqslant 98$ (with $c = 2$ giving 02 etc). The next four characters obviously indicates the bank (here Lloyds). What follows is the sort-code (30-11-27) and account number (00126886).

*Digression.* The sort-code and account number must pass a separate validation called 'modulus checking' that varies from bank to bank (and whether the account it a sterling or euro one!). Lloyds uses a system akin to ISBN 10 with (in our case) the weights shown in the first row:

| | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 3 | 2 | 9 | 8 | 5 | 2 | 6 | 5 | 4 | 3 | 2 | 1 |
| 3 | 0 | 1 | 1 | 2 | 7 | 0 | 0 | 1 | 2 | 6 | 8 | 8 | 6 |
| 0 | 0 | 3 | 2 | 18 | 56 | 0 | 0 | 6 | 10 | 24 | 24 | 16 | 6 |

The sum of the numbers in the last row is 165, which is a mltiple of 11, so the account number is valid.

Returning to the IBAN, move the first four characters to the back and remove spaces:

LOYD30112700126886GB76

Now replace any alphabetic letters by its position in the alphabet plus 9 (so A→10,..., L→11,..., Y→34). This gives the 26-digit number

21243413301127001268861611 76

which (as it stands, expressed to base 10) must equal 1 modulo 97, which it does! This condition uniquely specifies the check digits, with the assumption $2 \leqslant c \leqslant 98$. Because 97 is prime, any one of these 97 possibilities can occur. The drawback is that validation requires a computer.

*ISBN 13 (post 2007).* First, some general theory. Suppose that we want to add a single check digit $x_n$ to a string $x_1x_2\cdots x_{n-1}$, using the rule

$$c_0 + c_1x_1 + \cdots + c_nx_n = 0 \bmod N.$$

In order that $x_n$ is determined, we need $c_n$ to be coprime to $N$. For single errors to be detected we also need $c_i$ to be coprime to $N$ for all for all $i < n$. For transpositions to be detected, we need $c_i - c_j$ coprime to $N$ for all $i \neq j$.

ISBN 13 is validated by the 'check function'

$$f(\mathbf{x}) = x_1 + 3x_2 + x_3 + 3x_4 + \cdots + 3x_{12} + x_{13} \bmod 10,$$

but this fails to detect transpositions in which the digits differ by 5, like $27 \leftrightarrow 72$. For

$$3x_i + x_{i+1}, \quad 3x_{i+1} + x_i, \quad x_i + 3x_{i+1}, \quad x_{i+1} + 3x_i$$

are all equal modulo 10, and the check digit will be the same.

*Luhn algorithm.* This is used to determine the final digit of credit card numbers. First define

$$\widehat{2x} = \begin{cases} 2x & \text{if } x \in \{0,1,2,3,4\}, \\ 2x-9 & \text{if } x \in \{5,6,7,8,9\}. \end{cases}$$

The map $x \mapsto \widehat{2x}$ is the permuation

$$(0,1,2,3,4,5,6,7,8,9) \longmapsto (0,2,4,6,8,1,3,5,7,9)$$

with fixed points 0 and 9 (so $\widehat{2x}$ is not quite the obvious residue of $2x$ modulo 9). For a 16-digit number $\mathbf{x} = x_1\cdots x_{16}$ we define

$$f(\mathbf{x}) = \widehat{2x_1} + x_2 + \widehat{2x_3} + x_4 + \cdots + \widehat{2x_{15}} + x_{16}.$$

We then require that $f(\mathbf{x}) = 0 \bmod 10$. Without the 'hats', the function $f$ would not detect transcriptions differing by 5. But with the hats its detects all single transcriptions, and all adjacent transpositions except for $09 \leftrightarrow 90$ (thought to be less of a problem since 0 and 9 are far apart on the numerical keypad). It also corrects most twin errors $ii \leftrightarrow jj$, but not $22 \leftrightarrow 55$, $33 \leftrightarrow 66$ or $44 \leftrightarrow 77$, since (e.g.) $2 + \widehat{2} = 5 + \widehat{5}$.

## 11.2 Binary codes

These are based on the alphabet $\mathbb{B} = \{0, 1\}$. Later, it will be important to realize that (equipped with addition modulo 2 and multiplication) this set becomes a field. It is commonly denoted $\mathbb{Z}_2$, $\mathbb{Z}/2\mathbb{Z}$ or $\mathbb{Z}/(2)$. The problem with the first notation is that $\mathbb{Z}_2$ also stands for the (infinite) set of $p$-adic integers with prime $p = 2$. The other notations are clumsy, so we shall use $\mathbb{B}$ or (maybe later) $\mathbb{F}_2$.

The Cartesian product

$$\mathbb{B}^n = \mathbb{B} \times \cdots \times \mathbb{B}$$

is a vector space over the field $\mathbb{B}$ of dimension $n$ with an obvious basis. We abbreviate $(x_1, \ldots, x_n)$ to $x_1x_2\cdots x_n$.

**Definition 11.1.** *A* binary code *$C$ is a set of strings of 0's and 1's of length $n$, i.e. it is a subset of $\mathbb{B}^n$.*

*We shall call an element of $\mathbb{B}^n$ a* string *or* word*, and each element of $C$ a* codeword*. We regard $\mathbf{x} \in \mathbb{B}^n$ as 'valid' if $\mathbf{x}$ belongs to $C$, which we can think of (for the moment) as a set of valid account numbers expressed in binary.*

**Example 11.2.** Take $n = 4$ and define $C = \{0000,\ 0101,\ 1010,\ 1111\}$. You receive 0111. This is not in $< C$, so there must be an error. You can compare it to each element of $C$:

| received | codeword | # erroneous digits |
|---|---|---|
| 0111 | 0000 | 3 |
| 0111 | 0101 | 1 |
| 0111 | 1010 | 3 |
| 0111 | 1111 | 1 |

The original message was likely to have been 0101 or 1111. But that is still two choices – we want to design codes so that there is only one choice.

**Definition 11.3.** *The* Hamming distance *between two words $\mathbf{x}, \mathbf{y} \in \mathbb{B}^n$ is the number of bits by which they differ. It is denoted $\partial(\mathbf{x}, \mathbf{y})$.*

This function satisfies the properties of a *metric* in the sense of metric space, including the triangle inequality (for a proof of the latter, see §10.3):

$$\begin{cases} \partial(\mathbf{x}, \mathbf{y}) = 0 \quad \Leftrightarrow \quad \mathbf{x} = \mathbf{y} \\ \partial(\mathbf{x}, \mathbf{y}) = \partial(\mathbf{y}, \mathbf{x}) \\ \partial(\mathbf{x}, \mathbf{y}) \leqslant \partial(\mathbf{x}, \mathbf{z}) + \partial(\mathbf{z}, \mathbf{y}). \end{cases}$$

We shall always adopt the

**Minimum distance (MD) or nearest neighbour principle.** If an invalid word $\mathbf{x}$ is received, assume that the codeword $\mathbf{y}$ transmitted was one for which $\partial(\mathbf{x}, \mathbf{y})$ is as small as possible.

**Example 11.4.** Let $C = \{\mathbf{a} = 01101,\ \mathbf{b} = 10110,\ \mathbf{c} = 00011\}$. Then

$$\partial(\mathbf{b}, \mathbf{c}) = 3, \quad \partial(\mathbf{c}, \mathbf{a}) = 3, \quad \partial(\mathbf{a}, \mathbf{b}) = 4.$$

If we receive $\mathbf{x} = 01011$, we test

$$\partial(\mathbf{x}, \mathbf{a}) = 2, \quad \partial(\mathbf{x}, \mathbf{b}) = 4, \quad \partial(\mathbf{x}, \mathbf{c}) = 1,$$

so the MD principle tells us to assume that $\mathbf{c}$ was transmitted.

We want to design $C$ so that each codeword has a unique nearest neighbour (as measured with $\partial$). One might expect to achieve this if the codewords are well dispersed, which amounts to requiring that the distance between any two is sufficiently large:

**Definition 11.5.** *Let $C$ be a binary code. Its* minimum distance *is given by*

$$\delta = \min\{\partial(\mathbf{x}, \mathbf{y}) : \mathbf{x}, \mathbf{y} \in C,\ \mathbf{x} \neq \mathbf{y}\}.$$

Now suppose that $\delta \geqslant 2e + 1$. If $\mathbf{x} \in \mathbb{B}^n$ and $\mathbf{y}, \mathbf{y}' \in C$ then

$$\partial(\mathbf{x}, \mathbf{y}) \leqslant e, \ \ \partial(\mathbf{x}, \mathbf{y}') \leqslant e \implies \mathbf{y} = \mathbf{y}'.$$

This is an immediate consequence of the triangle inequality:

$$\partial(\mathbf{y}, \mathbf{y}') \leqslant \partial(\mathbf{y}, \mathbf{x}) + \partial(\mathbf{x}, \mathbf{y}') \leqslant 2e < \delta,$$

so the definition of $\delta$ implies that $\mathbf{y} = \mathbf{y}'$. As a consequence, we obtain the well-known

**Lemma 11.6.** *A binary code with $\delta \geqslant 2e + 1$ will correct $e$ errors using the MD principle.*

**Example 11.7.** In Example 1, $\delta = 2$ and this is not enough to correct any errors. In Example 2, $\delta = 3$ so one can detect and correct single errors.

**Lemma 11.8.** *Let $C \subset \mathbb{B}^n$ be a binary code with $\delta \geqslant 2e + 1$. Then*

$$|C|\Big(1 + n + \tbinom{n}{2} + \cdots + \tbinom{n}{e}\Big) \leqslant 2^n.$$

*Proof.* The expression in parentheses on the left-hand side equals the number of elements in $\mathbb{B}^n$ that are within distance $e$ of a given codeword $\mathbf{y}$. For example, there are $n$ words that differ from $\mathbf{y}$ by exactly one digit, and $\binom{n}{2}$ that differ by exactly two digits. If we surround each codeword $\mathbf{y}$ by the 'ball' or neighbourhood

$$N_e(\mathbf{y}) = \{\mathbf{y} \in \mathbb{B}^n : \partial(\mathbf{x}, \mathbf{y}) \leqslant e\},$$

no two balls can intersect, for Lemma 1 tells us that $N_e(\mathbf{y}) \cap N_e(\mathbf{y}') = \varnothing$. □

In the next section, we shall study the case $e = 1$ of '1-error correcting codes' in more detail, for which we need to assume that $\delta \geqslant 3$. Lemma 2 implies that $|C|(1 + n) \leqslant 2^n$, and equality here would imply that both $|C|$ and $n + 1$ are powers of 2. We shall show (in §10.4) that such codes do in fact exist.

## 11.3 Binary linear codes

We now specialize the set-up of the previous section to the case in which $C$ is a sub*space* of $\mathbb{B}^n$. This condition makes sense because $\mathbb{B}^n$ is a vector space over the finite field $\mathbb{B} = \{0,1\} = \mathbb{F}_2$, with coordinate-wise addition.

Remember that a word like 010101 really stands for the vector $(0,1,0,1,0,1)$. We need not worry about scalar multiplication since $2 = 0$ and $-1 = 1$ in $\mathbb{B}$! So we just need to verify that

$$\mathbf{x}, \mathbf{y} \in C \implies \mathbf{x} + \mathbf{y} \in C.$$

Any linear code must contain the zero vector $\mathbf{0} = 00\cdots0$. Moreover, it has a dimension $k$ with $k \leqslant n$, and a basis $\{\mathbf{x}_1, \cdots, \mathbf{x}_k\}$ consisting of $k$ elements. It then follows that

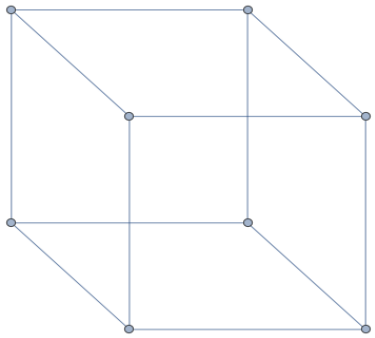
$$C = \Big\{ \sum_{i=1}^{k} a_k \mathbf{x}_k : a_k \in \mathbb{B} \Big\}$$

has $2^k$ elements. The space $\mathbb{B}^n$ itself has dimension $n$ and a basis consisting of the vectors $\{\mathbf{e}_i\}$ where $\mathbf{e}_i$ is the vector or word with a 1 in the $i$th position and zeros elsewhere.

**Example 11.9.** Let $C = \{000, 111\} \subset \mathbb{B}^3$. The two codewords can be visualized as the opposite vertices of a cube. Notice that $\delta = 3$ and

$$\mathbb{B}^3 = N_1(000) \sqcup N_1(111)$$

is partitioned into two subsets of size 4. This is an example of a *repeat code* in which each of two messages (0 and 1) is repeated twice to enable correction of 1 error.



Words in $\mathbb{B}^n$ can be thought of as vertices of an $n$-dimensional hypercube, but this is hard to visualize (at least for $n > 4$!). Here is a linear code with $(n, k, \delta) = (5, 2, 3)$ that we shall return to:

$$C = \{00000,\ 10110,\ 01011,\ 11101\}.$$

Any two of the nonzero elements form a basis of $C$.

**Definition 11.10.** *The* weight *of a word* $\mathbf{x} \in \mathbb{B}^n$ *equals the number of* 1*'s it has:*

$$\mathbf{x} = x_1 \cdots x_n \implies w(\mathbf{x}) = \sum_{i=1}^{n} x_i.$$

Recall the previous definition (of $\delta$).

**Lemma 11.11.** *Given a* linear *code* $C$,

$$\delta = \min\{w(\mathbf{x}) : \mathbf{x} \in C,\ \mathbf{x} \neq \mathbf{0}\}.$$

*So the minimum distance is also minimum* nonzero *weight in* $C$.

*Proof.* Denote (temporarily) the right-handf side by $\delta'$. The point is that

$$\partial(\mathbf{x}, \mathbf{y}) = \partial(\mathbf{x} - \mathbf{y}, \mathbf{0}) = \mathbf{w}(\mathbf{x} - \mathbf{y}),$$

which holds for any $\mathbf{x}, \mathbf{y} \in \mathbb{B}^n$ (we could equally well write + in place of −). If the latter belong to $C$ then so does $\mathbf{x} - \mathbf{y}$, so $\delta \geqslant \delta'$. But $w(\mathbf{z})$ is itself the distance of $\mathbf{z}$ from the zero vector $\mathbf{0} \in \mathbf{C}$, so $\delta' \geqslant \delta$. □

We can also use $w$ to prove the triangle inequality for $\partial$. If $\mathbf{x}, \mathbf{y} \in \mathbb{B}^n$ then

$$w(\mathbf{x} + \mathbf{y}) = \sum_{i=1}^{n} \widehat{x_i + y_i} \ \leqslant\ \sum_{i=1}^{n} (x_i + y_i) = w(\mathbf{x}) + w(\mathbf{y}),$$

where $\widehat{x_i + y_i} \in \{0, 1\}$ stands for the reduction of $x_i + y_i$ modulo 2. Thus $w$ behaves like a *norm* on a real vector space, and

$$\begin{aligned} \partial(\mathbf{x}, \mathbf{y}) \ = \ w(\mathbf{x} - \mathbf{y}) \ &= \ w(\mathbf{x} - \mathbf{z} + \mathbf{z} - \mathbf{y}) \\ &\leqslant \ w(\mathbf{x} - \mathbf{z}) + w(\mathbf{z} - \mathbf{y}) \\ &= \ \partial(\mathbf{x}, \mathbf{z}) + \partial(\mathbf{z}, \mathbf{y}). \end{aligned}$$

*Key example to illustrate the theory.* The first two nonzero elements of the previous example $C \subset \mathbb{B}^5$ correspond to the columns of the matrix

$$E = \begin{pmatrix} 1 & 0 \\ 0 & 1 \\ 1 & 0 \\ 1 & 1 \\ 0 & 1 \end{pmatrix} = \begin{pmatrix} I_2 \\ A \end{pmatrix},$$

where $I_n$ denotes the $n \times n$ identity matrix. Our convention is that matrices always act on the left on column vectors, so this defines a linear transformation

$$E\colon \quad \mathbb{B}^2 \longrightarrow \mathbb{B}^5.$$

It follows that $C = \operatorname{Im} E$, and each element of $C$ has the form

$$E\mathbf{v} = \begin{pmatrix} \mathbf{v} \\ A\mathbf{v} \end{pmatrix},$$

where $\mathbf{v}$ is one of

$$\text{west} = \begin{pmatrix} 0 \\ 0 \end{pmatrix}, \quad \text{north} = \begin{pmatrix} 1 \\ 0 \end{pmatrix}, \quad \text{south} = \begin{pmatrix} 0 \\ 1 \end{pmatrix}, \quad \text{east} = \begin{pmatrix} 1 \\ 1 \end{pmatrix}.$$

We shall freely transpose from rows to columns, using the latter when we need to act on them by matrices. With this confusion,

$$E\mathbf{v} = \boxed{\mathbf{v}}\boxed{A\mathbf{v}}.$$

Seen this way, $A\mathbf{v}$ plays the role of a check block for each of the four possibilities for $\mathbf{v}$, which might be commands for a robot to move. Observe that here the check is longer than the original message. This is to enable error *correction* rather than mere *detection*: since $\delta = 3$, the block 'protects' the direction in the event it is corrupted by 90 degrees.

We can describe $C$ in an equivalent way, using the matrix

$$H = \left(\begin{array}{cc|ccc} 1 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 \end{array}\right) = \left(\begin{array}{c|c} A & I_3 \end{array}\right).$$

For let

$$\mathbf{x} = \begin{pmatrix} \mathbf{b} \\ \mathbf{c} \end{pmatrix} \in \mathbb{B}^5,$$

with $\mathbf{b} \in \mathbb{B}^2$ and $\mathbf{c} \in \mathbb{B}^3$. Then

$$\begin{aligned} H\mathbf{x} = \mathbf{0} &\Leftrightarrow A\mathbf{b} + \mathbf{c} = \mathbf{0} \\ &\Leftrightarrow A\mathbf{b} = \mathbf{c} \\ &\Leftrightarrow \mathbf{x} \in \operatorname{Im} E \;=\; C. \end{aligned}$$

The matrix $H$ is called the *parity-check* or *check* matrix of the linear code $C$.

**Definition 11.12.** *Suppose that $r < n$. Let $H$ be a matrix of size $r \times n$ and entries in $\mathbb{B}$ (one can express this by writing $H \in \mathbb{B}^{r,n}$ and $r$ is the number of rows). Then the subspace*

$$\ker H = \{\mathbf{x} \in \mathbb{B}^n : H\mathbf{x} = \mathbf{0}\}$$

*of $\mathbb{B}^n$ is called the linear code* with check matrix $H$.

We shall always assume that $r \Longrightarrow nkH$, since if not we can delete one or more rows without affecting the kernel.

**Example 11.13.** Take $r = 1$ and $H$ to be the single row with all 1's. Then $C$ consists of all the elements of $\mathbb{B}^n$ of even weight. We could regard the first $n-1$ bits of $\mathbf{x} \in B^n$ as the 'message', and the final bit $x_n$ as a parity check digit, as in §10.1.

## 11.4 Correcting one error

We have already used three parameters to help describe a linear code:

$$\begin{aligned} n &= \text{number of bits in transmission} \\ k = n - r &= \text{dimension, so } |C| = 2^k \\ \delta &= \text{minimum distance between codewords.} \end{aligned}$$

Suppose that we need to correct one error in a transmitted message block. This requires a code (linear or not) with $\delta \geqslant 2e + 1 = 3$, and by Lemma 2 from §10.2,

$$s(1 + n) \leqslant 2^n,$$

where $s = |C|$ ($s$ for *size*). A big question is

*Given $s, n$ satisfying this inequality, does there exist $C \subset \mathbb{B}^n$ with $\delta = 3$ and $|C| = s$?*

*Easy exercise.* There is no code (linear or not) with $|C| = 3$, $n = 4$ and $\delta = 3$.

At the risk of repetition, let's summarize the definition of linear codes using matrices.

Such a code is often defined by a check matrix $H$ of size $r \times n$ with $r < n$. Then the set of codewords is

$$C = \{\mathbf{x}^\top : H\mathbf{x} = \mathbf{0}\} \subset \mathbb{B}^{\mathbf{n}}.$$

We naturally regard a *word* as a string written as a row, but it is always transposed to a column vector for the check matrix to test it. We shall usually omit the transpose symbol $^\top$, since context makes it clear whether one is dealing with a row or a column. So $C = \ker H$, i.e. $C$ is the kernel of the linear transformation

$$\begin{array}{ccc} \mathbb{B}^n & \longrightarrow & \mathbb{B}^r \\ \mathbf{x} & \longmapsto & H\mathbf{x}. \end{array}$$

We assume that $\Rightarrow nkH = r$, so that $\dim C = n - r$. We call this dimension $k$, so that there are $2^k$ codewords.

To make clearer the analogy with check digits, one often takes

$$H = \left( \begin{array}{c|c} A & I_r \end{array} \right)$$

so that the last block is the identity matrix. In this case, $H$ is said to be in *standard form*, but this is not always convenient. Note that $A$ has $n - r = k$ columns. We can define an 'encoding matrix'

$$E = \begin{pmatrix} I_k \\ A \end{pmatrix}.$$

By multiplying the blocks, we see that

$$HE = AI_k + I_r A = A + A = \mathbf{0}$$

is the zero matrix (of size $r \times k$). This means that $H$ annihilates the $k$ columns of $E$, which must therefore lie in $C$. But these $k$ columns are linearly independent because they include the columns of $I_k$, and they span the image of $E{:}\,\mathbb{B}^k \to \mathbb{B}^n$.

**Conclusion.** $C = \ker H = \operatorname{Im} E$, so as a row any codeword can be written

$$\boxed{\mathbf{v}} \; \boxed{A\mathbf{v}}.$$

Some authors would (correctly) express this as $(\mathbf{r},\ \mathbf{r}A^\top)$ having preferred to make explicit the row vector $\mathbf{r} = \mathbf{v}^\top$ and having choden to use $E^\top$ instead of $E$.

*Exercise.* Suppose that $C = \ker H$, where

$$H = \left(\begin{array}{cc|c} 1 & 0 & \\ 1 & 0 & \\ 1 & 0 & \\ 1 & 1 & I_7 \\ 0 & 1 & \\ 0 & 1 & \\ 0 & 1 & \end{array}\right).$$

What is the size of $C$? How many errors does it correct?.

**Lemma 11.14.** *Let $H$ be the check matrix of a linear code $C$. Then $\delta \geqslant 3$ (so $C$ corrects at least one error) provided no column of $H$ is zero and no two columns are equal. Moreover, if $\mathbf{x}$ differs from a codeword $\mathbf{y}$ by just one bit in the $i$th position (i.e. $\mathbf{x} = \mathbf{y} + \mathbf{e}_i$), then $H\mathbf{x}$ is the $i$th column of $H$.*

*Proof.* We need to ensure that $C$ has no words of *weight* 1 or 2. A word of weight one means it is $\mathbf{e}_i$ for some $i$, and $H_i = H\mathbf{e}_i$ is the $i$th column of $H$. So this must be nonzero. Similarly, a word $\mathbf{x}$ of weight 2 must equal $\mathbf{e}_i + \mathbf{e}_j$ with $i \neq j$, and so

$$H\mathbf{x} = H\mathbf{e}_i + H\mathbf{e}_j = H_i + H_j = H_i - H_j$$

must be nonzero. Finally, if $\mathbf{x} = \mathbf{y} + \mathbf{e}_i$ with $\mathbf{y} \in C$ then $H\mathbf{x} = H_i$. □

**Example 11.15.** The matrix

$$H = \begin{pmatrix} 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 & 1 & 0 & 1 \end{pmatrix}$$

obviously has rank 3, so defines a linear code of dimension $7 - 3 = 4$. Its parameters are $(n, k, \delta) = (7, 4, 3)$. If $\mathbf{x}$ differs from a codeword only in the $i$th position then $H\mathbf{x}$ (transposed to a row) is conveniently the binary expansion of $i$! If $H\mathbf{x}$ is nozero, it is called the *syndrome* of the word $\mathbf{x}$.

The best way to modify $H$ so that the identity matrix appears on the right is to perform row operations (as for echelon form) because this will not change the kernel of the matrix. We take

$$\begin{cases} \mathbf{r}'_1 &= \mathbf{r}_1 + \mathbf{r}_2 \\ \mathbf{r}'_2 &= \mathbf{r}_1 + \mathbf{r}_3 \\ \mathbf{r}'_3 &= \mathbf{r}_1 + \mathbf{r}_2 + \mathbf{r}_3 \end{cases}$$

to form

$$H' = \left(\begin{array}{cccc|ccc} 0 & 1 & 1 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 & 0 & 0 & 1 \end{array}\right).$$

The encoding matrix associated to $H'$ is

$$E = \left(\begin{array}{cccc} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ \hline 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \end{array}\right),$$

and any codeword then has the form $\boxed{\mathbf{v}}\,\boxed{A\mathbf{v}} \in \mathbb{B}^{4+3}$.

This time, the check block is smaller than the original message $\mathbf{v}$. To quantify this fact, one defines the *information rate* of the code as

$$\rho = \frac{k}{n} \quad \left(= \frac{\lg|C|}{n} \text{ if } C \text{ is not linear}\right).$$

Here we have $\rho = 4/7 \sim 0.57$.

**Exercise 11.16.** *Explain in what sense the code in the previous example can* detect *up to two errors, if it does not have to* correct *one!*

**Definition 11.17.** *Let $H$ be a matrix whose columns are all $2^r-1$ nonzero words formed from $k$ bits. The linear code $C = \ker H$ is called a* Hamming code*; it has parameters $(2^r-1, 2^r-r-1, 3)$.*

Any two Hamming codes of the same size ($|C| = 2^{2^r-r-1}$) are essentially equivalent, because permuting the columns will merely permute all the bits in $C$. When $r = 2$, we can take

$$H = \left(\begin{array}{c|cc} 1 & 1 & 0 \\ 1 & 0 & 1 \end{array}\right), \qquad E = \left(\begin{array}{c} 1 \\ 1 \\ 1 \end{array}\right),$$

so as to recover the repeat code $C = \{000, 111\} \subset \mathbb{B}^3$.

Hamming codes are *perfect*, meaning that we have equality in Lemma 2 in §10.2. Another way of saying this is that the balls

$$N_e(\mathbf{y}) = \{\mathbf{x} \in \mathbb{B}^n : \partial(\mathbf{x}, \mathbf{y}) \leqslant e\}$$

partition $C$ as $\mathbf{y}$ ranges over $C$. For the Hamming code in $\mathbb{B}^n$, we have $2^k$ balls each of size $1 + n = 2^r$. This was alluded to at the end of §10.2.

The information rate of a Hamming code is

$$\rho = \frac{k}{n} = \frac{2^r - r - 1}{2^r - 1} = \frac{1 - (r+1)2^{-1}}{1 - 2^{-r}} \to 1 \quad \text{as} \quad r \to \infty.$$

Already for $r = 6$ $(n = 63)$ we have $\rho > 0.9$.

Apart from the Hamming codes, codes of size 1 and repeat codes

$$\{0\cdots0,\ 1\cdots1\} \subset \mathbb{B}^n$$

of size 2 with $n$ odd, there is just one other perfect code. This is the mysterious *Golay code* $G_{23}$, a binary linear code with parameters $(23, 12, 7)$.