Computability, Algorithms, and Complexity

Course 240

Contents

Introduction

This course has three parts:

I: computability,

II: algorithms,

III: complexity.

In Part I we develop a model of computing, and use it to examine the fundamental properties and limitations of computers in principle (notwithstanding future advances in hardware or software). Part II examines some algorithms of interest and use, and Part III develops a classification of problems according to how hard they are to solve.

Parts I and III are fairly theoretical in approach, the aim being to foster understanding of the intrinsic capabilities of computers, real and imagined. Some of the material was crucial for the development of modern computers, and all of it has interest beyond its applications. But there are also practical reasons for teaching it:

- It is a good thing, perhaps sobering for computer scientists, to understand more about what computers can and can't do.
- You can honourably admit defeat if you know a problem is impossible or hopelessly difficult to solve. It saves your time. E.g., it is an urban myth that a programmer in a large British company was asked to write a program to check whether some communications software would 'loop' or not. We will see in section 5 that this is an impossible task, in general.
- The material we cover, especially in Part I, is part of the 'computing culture', and all computer scientists should have at least a nodding acquaintance with it.
- The subject is is of wide, indeed interdisciplinary interest. Popular books like Penrose's (see list above) and Hofstadter's 'Gödel, Escher, Bach' cover our subject, and there was quite a famous West End play ('Breaking the Code') about Turing's work a few years ago. The 'Independent' printed a long article on Gödel's theorem on 20 June 1992, in which it was said:

  > It is a measure of the achievement of Kurt Gödel that his Incompleteness Theorem, while still not considered the ideal subject with which to open a dinner party conversation, is fast becoming one of those scientific landmarks — like Einstein's Theory of Relativity and Heisenberg's Uncertainty Principle — that educated people,

even those with no scientific training, feel obliged to know something about.

Lucky you: we do Gödel's theorem in section 5.

Books and other reading

Texts (should be in the bookshop & library)

- V. J. Rayward-Smith, *A first course in computability,* McGraw Hill, 1995?

  An introductory paperback that covers Parts I and III of the course, and some of Part II. More detailed than this course.
- D. Harel, *The Science of Computing,* Addison-Wesley, 1989.

  A good book for background and motivation, with fair coverage of this course and a great deal more. Some may find the style diffuse. Less detailed than this course.

Advanced/reference texts See also the books on algorithms listed on page 96.

- Robert Sedgewick, *Algorithms,* Addison-Wesley, 2nd ed., 1988.

  A practical guide to many useful algorithms and their implementation. A reference for Part II of the course.
- J. Bell, M. Machover, *A course in mathematical logic,* North Holland, 1977.

  A good mathematical text, for those who wish to read beyond the course.
- G. Boolos, R. Jeffrey, *Computability and Logic,* Cambridge University Press, 1974.

  A thorough text, but mathematically demanding.
- M. R. Garey, D. S. Johnson, *Computers and intractability — a guide to NP-completeness,* Freeman, 1979.

  The 'NP-completeness bible'. For reference in Part III.
- J. E. Hopcroft & J. D. Ullman, *Introduction to automata theory, languages and computation,* Addison-Wesley, 1979. 2nd edn., J. E. Hopcroft, R. Motwani, J. D. Ullman, Addison-Wesley, 2001, ISBN: 0-201-44124-1.

  A classic text with a wealth of detail; but it concentrates on abstract languages and so has a different approach from ours.

Papers See also Stephen Cook's paper listed on page 154.

- A.M. Turing, *On computable numbers with an application to the Entscheidungsproblem,* Proceedings of the London Mathematical Society (Series 2), vol. 42 (1936), pp. 230–265.

  One of the founding papers of computer science, but very readable. Contains interesting philosophical reflections on the subject, the first description of the Turing machine, and a proof that some problems are unsolvable.
- G. Boolos, Notices of American Mathematical Society, vol. 36 no. 4 (April 1989), pp. 388–390. A new proof of Gödel's incompleteness theorem.

Popular material See also Chown's article mentioned on page 21.

- A. Hodges, *Enigma,* Vintage, 2nd edition, 1992.

  A biography of Alan Turing. Readable and explains some key ideas from this course (e.g., the halting problem) in clear terms.
- R. Penrose, *The Emperor's New Mind,* Vintage. Mainly physics but describes Turing machines in enough rigour to cover most of Part I of this course (e.g., halting problem). Enjoyable, in any case.

Notes on the text

The text has been used by Ian Hodkinson and Margaret Cunningham as coursenotes in the 20-lecture second-year undergraduate course '240 Computability, algorithms, and complexity' in the Department of Computing at Imperial College, London, UK, since 1991.

Italic font is used for emphasis, and **bold** to highlight some technical terms. '**E.g.**' means 'for example, **viz.** means 'namely', '**i.e.**' means 'that is', and **iff** means 'if and only if'. § means 'section' — for example, '§5.3.3' refers to the section called 'The Turing machine *EDIT*' starting on page 74.

There are bibliographic references on pages 6, 21, 96, and 154.

Part I

Computability

1. What is an algorithm?

We begin Part I with a problem that could pose difficulties for those who think computers are 'all-powerful'. To analyse the problem, we then discuss the general notion of an algorithm (as opposed to particular algorithms), and why it is important.

1.1 The problem

At root, Part I of this course is about paradoxes, such as:

> *The least number that is not definable by an English sentence having fewer than 100 letters.*

(The paradox is that we have just defined this number by such a sentence. Think about it!) C.C. Chang and H.J. Keisler kindly dedicated their book 'Model Theory' to all model theorists who have never dedicated a book to themselves. (Is it dedicated to Chang and Keisler, or not?)

Paradoxes like this often arise because of *self-reference* within the statement. The first one implicitly refers to all (short) English sentences, including itself. The second refers implicitly to all books, including 'Model Theory'. Now computing also uses languages — formal programming languages — that are capable of self-reference (for example, programs can alter, debug, compile or run other programs). Are there similar paradoxes in computing?

Here is a candidate. Take a high-level imperative programming language such as Java. Each program is a string of English characters (letters, numbers, punctuation, etc). So we can list all the syntactically correct programs in alphabetical order, as $P_1, P_2, P_3, \ldots$ Every program occurs in this list.

Each P_n will output some string of symbols, possibly the empty string. We can treat it as outputting a string of binary bits (0 or 1). Most computers work this way — if the output appears to us as English text, this is because the binary output has been treated as ASCII (for example), and *decoded* into English.

Now consider the following program P:

```
1  repeat forever
2      generate the next program $P_n$ in the list
3      run $P_n$ as far as the $n$th bit of the output
4      if $P_n$ terminates or prompts for input before the $n$th bit is output then
5          output 1
6      else if the nth bit of $P_n$'s output is 0 then
7          output 1
8      else if the nth bit of $P_n$'s output is 1 then
9          output 0
10     end if
11 end repeat
```

- This language is not quite Java, but the idea is the same — certainly we could write it formally in Java.
- Generating and running the next program (lines 2 and 3) is easy — we generate all strings of text in alphabetical order, and use an interpreter to check each string in turn for syntactic errors. If there are none, the string is our next program, and the interpreter can run it. This is slow, but it works.
- We assume that we can write an interpreter in our language — certainly we can write a Java interpreter in Java.
- In each trip round the loop, the interpreter is provided with the text of the next program, P_n, and the number n. The interpreter runs P_n, halting execution if (a) P_n itself halts, (b) P_n prompts for input or tries to read a file, or (c) P_n has produced n bits of output.
- All other steps of P are easy to implement.

So P is a legitimate program. So P is in the list of P_ns. Which P_n is P?

Suppose that P is P_7, say. Then P has the same output as P_7. Now on the seventh loop of P, P_7 (i.e., P) will be generated, and run as far as its seventh output bit. The possibilities are:

1. P_7 halts or prompts for input before it outputs 7 bits (impossible, as the code for $P = P_7$ has no HALT or READ statement!)
2. P_7 does output bit 7, and it's 0. Then P outputs 1 (look at the code above). But this 1 will be the 7th output bit of $P = P_7$, a contradiction!
3. P_7 does output bit 7, and it's 1. Then P outputs 0 (look at the code again). But this 0 will be P's 7th output bit, and $P = P_7$!

This is a contradiction: if P_7 outputs 0 then P outputs 1, and vice versa; yet P was supposed to be P_7. So P is not P_7 after all.

In the same way we can show that P is not P_n for any n, because P differs from P_n at the nth place of its output. So P is not in our list of programs. This is *impossible,* as the list contains all programs of our language!

Exercise 1.1 What is wrong?

Paradoxes might not be too worrying in a natural language like English. We might suppose that English is vague, or the speaker is talking nonsense. But we think of computing as a precise engineering-mathematical discipline. It is used for safety-critical applications. Certainly it should not admit any paradoxes. We should therefore examine our 'paradox' very carefully.

It may be that it comes from some quirk of the programming language. Perhaps a better version of Java or whatever would avoid it. In Part I of the course our aim is first to show that the 'paradox' above is extremely general and occurs in all reasonable models of computing. We will do this by examining a very simple model of a computer. In spite of its simplicity we will give evidence for its being fully general, able to do anything that a computer — real or imagined — could.

We will then rediscover the 'paradox' in our simple model. I have to say at this point that there is no real paradox here. The argument above contained an implicit assumption. [What?] Nonetheless, there is still a problem: the implicit assumption cannot be avoided, because if it could, we really would have a paradox. So we cannot 'patch' our program P to remove the assumption!

But now, because our simple model is so general, we are forced to draw fundamental conclusions about the limitations of computing itself. Certain precisely-stated problems are unsolvable by a computer even in principle. (We cannot write a patch for P.)

There are lots of unsolvable problems! They include:

- checking mechanically whether an arbitrary program will halt on a given input (the 'halting problem')
- printing out all the true statements about arithmetic and no false ones (Gödel's incompleteness theorem).
- deciding whether a given sentence of first-order predicate logic is valid or not (Church's theorem).

Undeniably these are problems for which solutions would be very useful.

In Part III of the course we will apply the idea of self-reference again to NP-complete problems — not now to the question of what we can compute, but to how fast can we compute it. Here our results will be more positive in tone.

1.2 What is an algorithm?

To show that our 'paradox' is not the fault of bad language design we must take a very general view of computing. Our view is that computers (of any kind) implement **algorithms.** So we will examine what an algorithm is.

First, a definition from Chambers Dictionary.

> **algorithm, al'go-ridhm,** n. a rule for solving a mathematical problem in a finite number of steps. [Root: Late Latin **algorismus,** from the Arabic name **al-Khwārazmi,** the native of Khwārazm (Khiva), i.e., the 9th century mathematician Abu Ja'far Mohammed ben Musa.]

We will improve on this, as we'll see.

1.2.1 Early algorithms

One of the earliest algorithms was devised between 400 and 300 B.C. by Euclid: it finds the highest common factor of two numbers, and is still used. The sieve of Eratosthenes is another old algorithm. Mohammed al-Khwārazmi is credited with devising the well-known rules for addition, subtraction, multiplication and division of ordinary decimal numbers.

Later examples of machines controlled by algorithms include weaving looms (1801, the work of J. M. Jacquard, 1752–1834), the player piano or piano-roll (the pianola, 1897 — arguable, as there is an analogue aspect (what?)), and the 1890 census tabulating machine of Herman Hollerith, immortalised as the 'H' of the 'format' statement in the early programming language Fortran (e.g., `FORMAT 4Habcd`). These machines all used holes punched in cards. In the 19th century Charles Babbage planned a multi-purpose calculating machine, the 'analytical engine', also controlled by punched cards.

1.2.2 Formalising Algorithm

In 1900, the great mathematician David Hilbert asked whether there is an algorithm that answers every mathematical problem. So people tried to find such an algorithm, without success. Soon they began to think it couldn't be done! Eventually some asked: can we *prove* that there's no such algorithm? This question involved issues quite different from those needed to devise algorithms. It raised the need to be precise about what an algorithm actually is: to formalise the notion of **'algorithm'**.

Why did no-one give a precise definition of **algorithm** in the preceding two thousand years? Perhaps because most questions on algorithms are of the form **find one to solve this problem I've got.** This can be done without a formal definition of algorithm, because we know an algorithm when we see one. Just as an elephant is easy to recognise but hard to define, you can write a program to sort a list without knowing *exactly* what an algorithm is. It is enough to invent something that intuitively is an algorithm, and that solves the problem in question. We do this all the time.

But suppose we had a problem (like Hilbert's) for which many attempts to find an algorithmic solution had failed. Then we might suspect that the task is impossible, so we would like to *prove* that no algorithm solves the problem. To have any hope of doing this, it is clearly *essential* to define precisely what an algorithm is, because we've got to know what counts as an algorithm. Similarly, to answer questions concerning *all* algorithms we need to know *exactly* what an algorithm is. Otherwise, how could we proceed at all?

1.2.3 Why formalise Algorithm?

As we said, we formalise **algorithm** so that we can reason about algorithms in general, and (maybe) prove that some problems have no algorithmic solution. Any formalisation of the idea of an **algorithm** should be:

- *precise* and unambiguous, with no implicit assumptions, so we know what we are talking about. For maximum precision, it should be phrased in the language of mathematics.
- *simple* and without extraneous details, so we can reason easily with it.
- *general*, so that all algorithms are covered.

Once formalised, an idea can be explored with rigour, using high-powered mathematical techniques. This can pay huge dividends. Once gravity was formalised by Newton as $F = Gm_1m_2/r^2$, calculations of orbits, tides, etc., became possible, with all that that implies. Pay-offs from the formalisation of **algorithm** included the modern programmable computer itself.[1] This is quite a spectacular pay-off! Others include the answer to Hilbert's question, related work in complexity (see Part III) and more besides.

1.2.4 Algorithm formalised

The notion of an algorithm was not in fact made formal until the mid-1930s, by mathematicians such as Alan Turing in England and (independently) Alonzo Church in America. Church and Turing used their formalisations to show that some mathematical problems have no algorithmic solution — they are unsolvable. (Turing used our 'paradox' to do this.) Thus, after 35 years, Hilbert's question got the answer 'NO'.

Turing's formalisation was by the primitive computer called (nowadays!) the **Turing machine.** The Turing machine first appeared in his paper in the reading list, in 1936, some ten years before 'real' computers were invented.[2] Turing's formalisation of the notion of an algorithm was: **an algorithm is what a Turing machine implements.**

We will describe the Turing machine at length below. We will see that it is *precise* and *simple*, just as a formalisation should be. However, to claim that it is *fully general* — covering all known and indeed all conceivable algorithms — may seem rash, especially when we see how primitive a Turing machine is. But Turing gave substantial evidence for this in his paper, evidence which has strengthened over the years, and the usual view nowadays is that the Turing machine is fully general. For historical reasons, this view is known as **Church's thesis,** or sometimes (better) as the **Church–Turing thesis.** We will examine the evidence for it after we have seen what a Turing machine is.

[1] This is, of course, an arguable historical point; Hodges' book (listed on p. 7) examines the historical background.

[2] Turing later became one of the pioneers in their development.

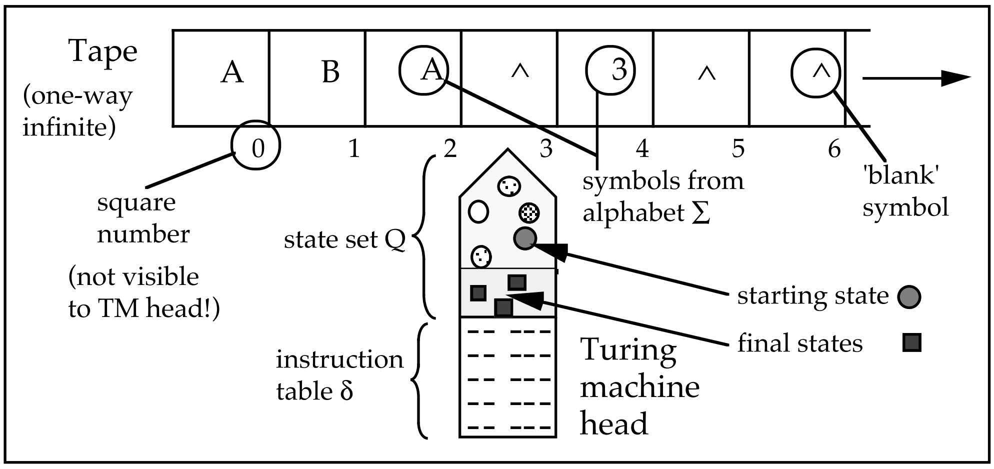


Figure 1.1: A Turing machine

1.3 Turing machines

We, also, will use Turing machines to formalise the concept of **algorithm.** Here we explain in outline what a Turing machine (TM) is; we'll do it formally in section 2. As we go through, think about how the Turing machine, our formalisation of **algorithm,** fits our requirements of *precision* and *simplicity*. Afterwards, we'll say more about its *generality* and why we use it in this course.

1.3.1 Naming of parts

There are several, mildly different but equally powerful, versions of the TM in the textbooks. We now explain what our chosen version of the TM *is*, and what it *does*.

In a nutshell, a Turing machine consists of a **head** that moves up and down a **tape,** reading and writing as it goes. At each stage it's in one of finitely many '**states**'. It has an **instruction table** that tells it what to do at each step, depending on what it's reading and what state it's in.

The tape The main memory of a TM is a 1-way-infinite **tape**, viewed as laid out from left to right. The tape goes off to the right, forever. It is divided into **squares,** numbered 0, 1, 2, ...; these numbers are for our convenience and are *not* seen by the Turing machine.

The alphabets In each square of the tape is written a single **symbol**. These symbols are taken from some finite **alphabet.** We will use the Greek letter **sigma** (Σ) to denote the alphabet. The alphabet Σ is part of the Turing machine. Σ is just a set of symbols, but it will always be finite with at least two symbols, one of which is a special **blank** symbol which we always write as '∧'. Subject to these restrictions, a Turing machine can have any finite alphabet Σ we like.

A blank in a square really means that the square is empty. Having a symbol for ‘empty’ is convenient — we don’t have to have a special case for empty squares, so things are kept *simple*.

The read/write head The TM has a single **head,** as on a tape recorder. The head can read and write to the tape.

At any given moment the head of the TM is positioned over a particular square of the tape — the **current square.** At the start, the head is over square 0.

The set of states The TM has a finite set Q of **states.** There is a special state q_0 in Q, called the **starting state** or **initial state.** The machine begins in the starting state, and changes state as it goes along. At any given stage, the machine will be ‘in’ some particular state in Q, called the **current state.** The current state is one of the two factors that determine, at each stage, what it does next (the other is the symbol in the square where the head is). The state of the TM corresponds roughly to the current instruction together with the contents of the registers in a conventional computer. It gives our ‘current position’ within the algorithm.

1.3.2 Starting a TM; input

A Turing machine starts off in the initial state, with its head over square 0. At the beginning, the tape will contain a finite number (possibly zero) of non-blank symbols, left-justified; this string of non-blank symbols constitutes the **input** to the Turing machine. The rest of the tape squares will be blank (i.e., they contain $\wedge$).

1.3.3 The run of the TM

A **run** is a step-by-step computation of the TM. At each step of a run:

(a) the head **reads** the symbol on the current tape square (the square where the head now is).

Then the TM does three things.

(b) First, the head **writes** some symbol from Σ to the current tape square.

Then:

(c) the TM may **move** its head left or right along the tape by one square,

(d) the TM goes into a new state.

Now the next step begins: it does (a)–(d) again, perhaps making different choices in (b)–(d) this time. And so on, step by step.

Note that:

- The TM writes *before* moving the head.

- In (b), the TM could write the same symbol as it just read. So in this case, the tape contents will not change.
- Similarly, in (d) the state of the TM may not change (as perhaps in a loop). In (c), the head may not move.

Also notice that the tape will always contain only finitely many non-blank symbols, because at the start only finitely many squares are not blank, and at most one square is altered at each step.

1.3.4 The instruction table

At each step (b)–(d) above, there are 'choices' to be made. Which symbol to write? Which way to move? And which state to enter? The answers depend *only* on:

(i) which symbol the machine reads from the current tape square;

(ii) the current state of the machine.

The machine has an **instruction table,** telling it what to do when, in any given state, a given symbol is read. We write the instruction table as δ, the Greek letter **delta.** δ corresponds to the *program* of a conventional computer. It is in effect just a list with five columns:

```
current_state;  current_symbol;  new_state;  new_symbol;  move
current_state;  current_symbol;  new_state;  new_symbol;  move
current_state;  current_symbol;  new_state;  new_symbol;  move
.....           .....            .....       .....        ....
```

Knowing the current state and symbol, the Turing machine can read down the list to find the relevant line, and take action according to what it finds. To avoid ambiguity, no pair (current-state; current-symbol) should occur in more than one line of the list.[3] (You might think that every such pair should occur somewhere in the list, but in fact we don't insist on this: see **Halting** below.)

Clearly, the 'programming language' is very low-level, like assembler. This fits our wish to keep things simple. But we will see some higher-level constructs for TMs later.

1.3.5 Stopping a TM; output

The run of a TM can terminate in just three different ways.

1. Some states of Q are designated special states, called **final states** or **halting states.** We write F for the set of final states. F is a subset of Q. If the machine gets into a state in F, then it stops there and then. In this case we say it **halts and succeeds,** and the **output** is whatever is left on the tape, from square 0 up to (but not including) the first blank.

[3]In Part III we drop this condition!

2. Sometimes there may be **no applicable instruction** in a given state when a particular symbol is read, because the pair (current-state; current-symbol) does not occur in the instruction table δ. If so, the TM is stuck: we say that it **halts and fails.** The output is **undefined** — that is, *there isn't an output.*

3. If the head is over square 0 of the tape and tries to **move left from square 0** along the tape, we count it as 'no applicable instruction' (because there is no tape square to the left of square 0, so the TM is stuck again). So in this case the machine also halts and fails. Again, the output is undefined.

Of course the machine may never halt — it may go on running forever. If so, the output is again undefined. E.g., it may be writing the decimal expansion of π on the tape 'to the last place' (there is a Turing machine that does this). Or it may get into a loop: i.e., at some point of the run, its 'configuration' (state, tape and head position) are exactly the same as at some earlier point, so that from then on, the same configurations will recycle again, forever. (A machine computing π never does this, as the tape keeps changing as more digits are printed. It never halts, but it doesn't loop, either.)

1.3.6 Summary

The Turing machine has a 1-way infinite tape, a read/write head, and a finite set of states. It looks at its state and reads the current square, and then writes, moves and changes state according to its instruction table. *Get-State, Read, Write, Move, Next-State*. It does this over and over again, until it halts, if at all. And that's it!

1.4 Why use Turing machines?

Although the Turing machine is based on 1930s technology, we will use it in this course because:

- It fits the requirements that the formalisation of algorithm should be *precise* and *simple*. (We'll make it even more precise in section 2.) Its *generality* will be discussed when we come to Church's thesis — the architecture of the Turing machine allows strong intuitive arguments here.

- It remains the most common formalisation of **algorithm**. Researchers, research literature and textbooks usually use Turing machines when a formal definition of computability is needed, so after this course you'll be able to understand them better.

- It is the standard benchmark for reasoning about the time or space used by an algorithm (see Part III).

- It crops up in popular material such as articles in New Scientist and Scientific American, and books by the likes of D. Hofstadter.

- It is now part of the computing culture. Its historical importance is great and every computer scientist should be familiar with it.

Why not adopt (say) a Cray YMP as our model? We could, but it would be too complex for our work. Our aim here is to study the concept of computability. We are concerned with which problems can be solved *in principle,* and not (yet) with practicality. So a very simple model of a computer, whose workings can be explained fully in a page or two, is better for us than one that takes many manuals to describe, and may have unknown bugs in it. And one can prove that a TM can solve exactly the same problems as an **idealised** Cray with unlimited memory!

1.4.1 How and why is a Turing machine *idealised*?

A TM is an **idealised computer,** because the amounts of time and tape memory that it is allowed to use are *unbounded*. This is not to say that it can use *infinitely* much time or memory. It can't (unless it runs forever — e.g., when it 'loops'). Think of a computer with infinitely many disk drives and RAM chips, which we allow to work on a problem for many years or even centuries. However long it runs for, at the end it will have executed only finitely many instructions. Because it can access only a finite amount of memory per instruction, on termination it will only have used a finite amount of disk space and RAM. But if we only gave it a fixed finite number of disks, if it ran for long enough it might fill them all up and run out of memory.

So our idealisation is this: only finitely much memory and time will get to be used in any given calculation, or run; but we set no limit on how much can be used.

We make these idealisations because our notion of **algorithm** should not depend on the state of technology, or on our budgets. For example, the function $f(x) = x^2$ on integers is intuitively computable by al-Khwārazmi's multiplication algorithm, although no existing computer could compute it for $x > 10^{10^{20}}$ (say). A TM can compute x^2 for all integers x, because it can use as much time and memory as it needs for the x in question. So idealising gives us a better model.

Nonetheless, the notion of being computable using only so much time or space is an important refinement of the notion of **computable.** It gives us a formal measure of the **complexity** (difficulty) of a problem. In Part III we will examine this in detail.

1.5 Church's thesis

Why should we believe — with Church and Turing — that such a primitive device as a Turing machine is a good formalisation of **algorithm** and could calculate not only all that a modern computer can, but anything that is in principle calculable?

First, is there anything to formalise at all? Maybe *any* definition of algorithm has exceptions, and there are exceptions to the exceptions, and so on. It is a notable fact about our world that this seems not to be so. Though the Turing machine looks very different to Church's alternative formalisation of **algorithm,**[4] *exactly the same things*

[4]Alonzo Church (c. 1935) used the lambda calculus — the basis of LISP.

turned out to be algorithms under either definition! Their definitions were **equivalent.**

Now if two people independently put forward quite different-looking definitions of **algorithm** that turn out to be equivalent, we may take it as evidence for the correctness of both. Such a ‘coincidence’ hints that there is in nature a genuine class of things that are algorithms, one that fits most definitions we offer.

This and other considerations (below) made Church put forward his famous *Thesis,* which says that the definition is the correct one.

This is also known as the *Church-Turing thesis,* and, when phrased in terms of Turing machines, it is certainly argued for in Turing’s 1936 paper, which was written without knowing Church’s work. But the shorter title is probably more common, though less just.

1.5.1 What does Church’s thesis say?

Roughly, it says: *A problem can be solved by an algorithm if and only if it can be solved by a Turing machine.* More formally, it says that a *function* is *computable* if and only if it is computable by a Turing machine.

1.5.2 What does it mean?

When we see Turing machines in action below, it will be clear that each one implements an algorithm (because we know an algorithm when we see one). So few people would reject the if direction ($\Leftarrow$) of the thesis.[5] The heart of the thesis lies in the only if ($\Rightarrow$) direction: *every algorithmically-solvable problem can be solved by a Turing machine.*

It is important to understand the status of this statement. It is not a *theorem*. It cannot be *proved*: that’s why it’s called a thesis.

Why can’t we prove it? Is it that there are (obviously) infinitely many algorithms, so to check that each of them can be implemented by a Turing machine would take infinitely long and so is impossible? No! I agree that if there were finitely many algorithms, we *could* in principle check that each one can be implemented by a Turing machine. But the fact that there are infinitely many is not of itself a fatal problem, as there might be other ways of showing that every algorithm can be implemented by a Turing machine than just checking them one by one. *It is not impossible to reason about infinite collections.* Compare: there are infinitely many right triangles; but we are still able to establish (some!) properties of all of them, such as ‘the square of the hypotenuse is equal to the sum of the squares of the other two sides’.

No; the real problem is that, although the notion of a Turing machine is completely precise (we will give a mathematical definition below), we have seen that the notion of an algorithm is an *intuitive, informal* one, with roots going back two thousand years. We can’t prove Church’s thesis, because it is not — cannot be — stated precisely enough.

[5]Some would say that a Turing machine only implements an algorithm if we can be sure that its computation will terminate, or even that we know how long it will take.

Instead, Church's thesis is more like a *definition* of **algorithm.** It says: 'Here is a mathematical model', and it asks us to accept — and in this course we do accept this — that any algorithm that we could possibly imagine fits the model and could be implemented by a Turing machine.

So Church's thesis is the claim that the Turing machine is a fully general formalisation of *algorithm.*

This is rather analogous to a scientific theory. For example, Newton's theory of gravity says that gravity is an attractive force that acts between any two bodies and depends on their masses and the square of the distance separating them. This formalises our intuitive idea of gravity, and the formalisation has been immensely useful. But we could not prove it correct.

Of course, Newton's theory of gravity was falsified by experiment. In the same way, Church's thesis could in a sense be *disproved,* if we found something that intuitively was an algorithm but that we could prove was not implementable by a Turing machine. We would then have to revise the thesis.

1.5.3 Evidence for the thesis

Given a new scientific theory, we would check its predictions by experimenting, and conduct 'thought experiments' to study its consequences. Since Church's thesis formalises the notion of **algorithm,** which is absolutely central to computer science, we had better examine carefully the evidence for its correctness. This evidence also depends on 'observations' and 'thought experiments'. In Turing's original 1936 paper, listed on p. 7, three kinds of evidence are suggested:

(a) Giving examples of large classes of numbers which are computable.

(b) A proof of the equivalence of two definitions (in case the new definition has a greater intuitive appeal).

(c) A direct appeal to intuition.

Let us examine these.

(a) Turing machines can do a wide range of algorithmic-like activities. They can compute arithmetical and logical functions, partial derivatives, do recursion, etc. In fact, no-one has yet found an algorithm that cannot be implemented by a Turing machine.

(b) All other suggested definitions of algorithm have turned out to be equivalent (in a precise sense) to Turing machines. These include:

- Software: the lambda-calculus (due to Church), production systems (Emil Post), partial recursive functions (Stephen Kleene), present-day computer languages.
- Hardware: register or counter machines, idealisations of our present-day computers, idealised parallel machines, and idealised neural nets.

They all look very different, but can solve (at best) precisely the same problems as Turing machines. As we will see, various souped-up versions of the Turing machine itself — even **non-deterministic** variants — are also equivalent to the basic model.

The essential features of the Turing machine are:

- its computations work in a discrete way, step by step, acting on only a finite amount of information at each stage,
- it uses finite but unbounded storage.

Any model with these two features will probably lead to an equivalent definition of algorithm.

(c) There are intuitive arguments that any algorithm could be implemented by a Turing machine. In his paper, Turing imagines someone calculating (computing) by hand.

> It is always possible for the computer to break off from his work, to go away and forget all about it, and later to come back and go on with it. If he does this he must leave a note of instructions (written in some standard form) explaining how the work is to be continued. . . . We will suppose that the computer works in such a desultory manner that he never does more than one step at a sitting. The note of instructions must enable him to carry out one step and write the next note. Thus the state of progress of the computation at any stage is completely determined by the note of instructions and the symbols on the tape. . . . This [combination] may be called the "state formula". We know that the state formula at any given stage is determined by the state formula before the last step was made, and we assume that the relation of these two formulae is expressible. In other words we assume that there is an axiom A which expresses the rules governing the behaviour of the computer, in terms of the relation of the state formula at any stage to the state formula at the preceding stage. If this is so, we can construct a machine to write down the successive state formulae, and hence to compute the required number. (pp. 253–4).

So for Turing, any calculation that a person can do on paper could be done by a Turing machine: type (c) (i.e., intuitive) evidence for Church's thesis. He also showed that π, e, etc., can be printed out by a TM (type (a) evidence), and in an appendix proved the equivalence to Church's lambda calculus formalisation (type (b)).

1.5.4 Other paradigms of computing

We can vary the notion of **algorithm** by dropping the requirement that it must take only finitely many steps. This leads to new notions of a **problem,** such as the 'problem' an operating system or word processor tries to solve, and has given rise to work on **reactive systems.** These are not supposed to terminate with an answer, but to keep running forever; their behaviour over time is what is of interest.

Is Church's thesis really true then? Can Turing machines do interactive work? Well, as the 'specification' for an interactive system corresponds to a function whose input and output are 'infinite' (the interaction can go on forever), the Turing machine model needs modifying. *But the basic Turing machine hardware is still adequate — it's only how we use it that changes.* For example, every time the Turing machine reaches a halting state, we might look at its output, overwrite it with a new input of our choice (depending on the previous output), and set it off again from the initial state. We could model a word processor like this. The collection of all the inputs and outputs (the 'behaviour over time/at infinity') is what counts now. This is research material and beyond the scope of the course. See Harel's book for more information on reactive systems.

More recent challenges to Church's thesis include quantum computers — whether they violate the thesis depends on who you read (go to the third-year course on quantum computing). Another is a Turing machine dropped into a rotating black hole. Theoretically, such a 'Marvin machine' could run forever, yet we could still read the 'answer' after its infinitely long computation. Recent research (still in progress) suggests this might be possible in principle in certain kinds of solution to Einstein's equations of general relativity. Whether it could ever be practically possible is quite another question, and whether it would violate Church's thesis is debated among philosophers.

Those who want to find out more could start with the article *Smash and grab* by Marcus Chown, New Scientist vol 174 issue 2337, 6 April 2002, page 24, online via `http://archive.newscientist.com/`

1.6 Summary of section

We viewed computers as implementing (running) algorithms. We gave a worrying 'paradox' in a Java-like language. To find out how serious it is for computing, we needed to make the notion of **algorithm** completely precise (formal). We discussed early algorithms, and Hilbert's question which prompted the formalising of the vague, intuitive notion of **algorithm.** Turing's formalisation was via **Turing machines,** and we explained what a Turing machine is. We finally discussed **Church's thesis,** saying that Turing machines can implement *any* algorithm. Since this is really a definition so can't be proved, we looked at evidence for it.

2. Turing machines and examples

We must now define Turing machines more precisely, using mathematical notation. Then we will see some examples and programming tricks.

2.1 What exactly is a Turing machine?

Definition 2.1 A **Turing machine** is a 6-tuple $M = (Q, \Sigma, I, q_0, \delta, F)$, where:

- Q is a finite non-empty set. The elements of Q are called **states.**
- Σ is a finite set of at least two elements or symbols. Σ is called the **alphabet** of M. We require that $\wedge \in \Sigma$.
- I is a non-empty subset of Σ, with $\wedge \notin I$. I is called the **input alphabet** of M.
- $q_0 \in Q$. q_0 is called the **starting state,** or **initial state.**
- $\delta : (Q \setminus F) \times \Sigma \to Q \times \Sigma \times \{-1, 0, 1\}$ is a partial function, called the **instruction table** of M. ($Q \setminus F$ is the set of all states in Q but not in F.)
- F is a subset of Q. F is called the set of **final** or **halting** states of M.

2.1.1 Explanation

Q, Σ, q_0, and F are self-explanatory, and we'll explain I in §2.2.1 below. Let us examine the instruction table δ. If q is the current state and s the character of Σ in the current square, $\delta(q, s)$ (if defined) will be a triple $(q', s', \delta) \in Q \times \Sigma \times \{-1, 0, 1\}$. This represents the *instruction* to make q' the next state of M, to write s' in the old square, and to move the head in direction d: -1 for left, 0 for no move, $+1$ for right. So the line

```
q    s    q'    s'    d
```

of the 'instruction table' of §1.3.4 is represented formally as

$$\delta(q, s) = (q', s', d).$$

We can represent the entire table as a partial function δ in this way, by letting δ(first two symbols) = last three symbols, for each line of the table. The table and δ carry the same information. Functions are more familiar mathematical objects than 'tables', so it is now standard to use a function for the instruction table. But it is not essential: Turing used tables in his original paper.

Note that $\delta(q, s)$ is undefined if $q \in F$ (why?). Also, δ is a *partial* function: it is undefined for those arguments (q, s) that didn't occur in the table. So it's OK to write $\delta : Q \times \Sigma \to Q \times \Sigma \times \{-1, 0, 1\}$, rather than $\delta : (Q \setminus F) \times \Sigma \to Q \times \Sigma \times \{-1, 0, 1\}$, since δ is partial anyway.

2.2 Input and output of a Turing machine

We now have to discuss the *tape contents* of a TM. First some notation to help us.

Definition 2.2 (Words)

1. A **word** is a finite string of symbols. Example: $w = abaa\wedge\wedge aab\wedge$ is a word. The length of w is 10 (note that the blanks '$\wedge$' count as part of the word and contribute to its length).

2. If Σ is a set, a **word of** Σ is a finite string of elements of Σ. We write Σ^* for the set of all words of Σ. So the above word w is in $\{a,b,c,\wedge\}^*$, even though c is not used. Remember: a **word** of Σ is an **element** of the set Σ^*.

3. There is a unique word of length 0, and it lies in any Σ^*; we write this **empty word** as ε. Also, each symbol in Σ is already a word of Σ, of length 1.

4. Clearly, if w, w' are words of Σ then we can form a new word of Σ by writing w' straight after w. We denote this concatenation by ww', or, when it is clearer, $w.w'$.

5. We also define well-known functions $\mathsf{head} : \Sigma^* \to \Sigma$ and $\mathsf{tail} : \Sigma^* \to \Sigma^*$ by:

   - if $s \in \Sigma$ and $w \in \Sigma^*$ then $\mathsf{head}(s.w) = s$
   - $\mathsf{tail}(s.w) = w$
   - $\mathsf{head}(\varepsilon) = \mathsf{tail}(\varepsilon) = \varepsilon$

   So, e.g.,

$$\begin{aligned}\mathsf{head}(abaa\wedge\wedge aab\wedge) &= a\\ \mathsf{tail}(abaa\wedge\wedge aab\wedge) &= baa\wedge\wedge aab\wedge\end{aligned}$$

2.2.1 The input word

A Turing machine $M = (Q, \Sigma, I, q_0, \delta, F)$ starts a run with its head positioned over square 0 of the tape. Left-justified on the tape is some word w of I. Recall that I is the input alphabet of M, and does not contain $\wedge$. So w contains no blanks.

So for example, if the word is $w = w_0, \ldots, w_{n-1} \in I^*$, then w_0 goes in square 0, w_1 in square 1, and so on, up to square $n-1$. The rest of the tape (squares $n, n+1, n+2, \ldots$) contains only blanks. The contents of the tape are shown in figure 2.1.

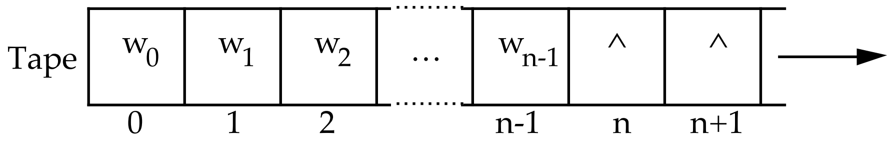


Figure 2.1: Tape with contents w

The word w is the **input** of M for the coming run. It is the initial data that we have provided for M.

Note that w can have any finite length ≥ 0. M will probably want to read all of w. How does M know where w ends? Well, M can just move its head rightwards until its head reads a blank '$\wedge$' on the tape. Then it knows it has reached the end of the input. This is why w must contain no blanks, and why the remainder of the tape is filled up with blanks ($\wedge$). If w were allowed to have blanks in it, M could not tell whether it had reached the end of w. Effectively, $\wedge$ is the 'end-of-data' character for the input.

Of course, M might put blanks anywhere on the tape when it is running. In fact it can write any letters from Σ. The extra letters of $\Sigma \setminus I$ are used for rough (or 'scratch') work, and we call them **scratch characters.**

2.2.2 Run of a Turing machine

This is as explained in §1.3.3. At stage 0, the TM $M = (Q, \Sigma, I, q_0, \delta, F)$ is in state q_0 with its head over square 0 of the tape. Let $n \geq 0$ and suppose (inductively) that at stage n, M is in state q (where $q \in Q$), with its head over square s (where $s \geq 0$), and the symbol in square s is a (where $a \in \Sigma$).

1. If $q \in F$ then M halts & succeeds.
2. Otherwise, if $\delta(q, a)$ is undefined, M halts & fails.
3. Otherwise, suppose that $\delta(q, a) = (q', a', d)$.
   (a) If $s + d < 0$ then M halts & fails.
   (b) Otherwise, at stage $n + 1$, the contents of square s will be a', all other squares of the tape will be the same as at stage n, the state of M will be q', and its head will be over square $s + d$.

2.2.3 Output of a Turing machine

Like the input, the **output** of a Turing machine $M = (Q, \Sigma, I, q_0, \delta, F)$ is a word in Σ^*. The output depends on the input. Just as the input is what is on the tape to begin with, so the output is what is on the tape at the end of the run, up to but not including the first blank on the tape — *assuming M halts successfully.* If, on a certain input, M **halts and fails,** or does not halt, then the output for that input is *undefined* — that is, there isn't one.

Recall that at each stage, only finitely many characters on the tape are non-blank. So the output is a *finite* word of Σ^*. It can be the empty word, or involve symbols from Σ that are not in I, but it never contains $\wedge$.

Exercise 2.3 Consider the Turing machine $M = (\{q_0, q_1, q_2\}, \{1, \wedge\}, \{1\}, q_0, \delta, \{q_2\})$, with instruction table:

q_0	1	q_1	$\wedge$	1
q_0	$\wedge$	q_2	$\wedge$	0
q_1	1	q_0	1	1

So δ is given by: $\delta(q_0,1) = (q_1,\wedge,1)$, $\delta(q_0,\wedge) = (q_2,\wedge,0)$, and $\delta(q_1,1) = (q_0,1,1)$.

List the successive configurations of the machine and tape until M halts, for inputs 1111, 11111 respectively. What is the output of M in each case?

Definition 2.4 (Input-output function of M**)** Given a Turing machine $M = (Q,\Sigma,I,q_0,\delta,F)$, we can define a partial function $f_M : I^* \to \Sigma^*$ by: $f_M(w)$ is the output of M when given input w.

The function f_M is called the **input-output function of** M, or the **function computed by** M. f_M is a *partial* function — it is not defined on any word w of I^* such that M halts and fails or does not halt when given input w.

Exercise 2.5 Let M be in exercise 2.3. Let 1^n abbreviate $1111\ldots1$ (n times). For which n is $f_M(1^n)$ defined?

2.2.4 Church's thesis formally

Let I,J be any alphabets (finite and non-empty). Let A be some algorithm all of whose inputs come from I^* and whose outputs are always in J^*. (For example, if A is al-Khwārazmi's decimal addition algorithm, then we can take I and J to be $\{0,1,\ldots,9\}$.) Consider a Turing machine $M = (Q,\Sigma,I,q_0,\delta,F)$, for some Σ containing I and J. We say that M **implements** A if for any word $w \in I^*$, if w is given to A and to M as input, then A has an output if and only if M does, and in that case their output is the same. If you like, M computes the same function as A.

We can now state Church's thesis formally as follows:

- 'Given any algorithm, there is some Turing machine that implements it.' Or:
- 'Any algorithmically computable function is Turing-computable — computable by some Turing machine.' Or:
- 'For any finite Σ and any function $f : \Sigma^* \to \Sigma^*$, f is computable iff there is a Turing machine M such that $f = f_M$.'

This is formal, but it is still imprecise, as the intuitive notion of 'algorithm' is still (and has to be) involved.

2.3 Representing Turing machines

2.3.1 Flowcharts of Turing machines

Written as a list of 5-tuples, the instruction table δ of a TM M can be hard to understand. We will often find it easier to represent M as a **graph** or **flowchart.** The nodes of the flowchart are the states of M. We use square boxes for the final states, and round ones for other states. An example is shown in figure 2.2.

The arrows between states represent the instruction table δ. Each arrow is labelled with one or more triples from $\Sigma \times \Sigma \times \{-1,0,1\}$. If one of the labels on the arrow

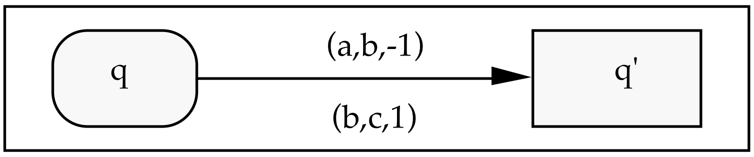


Figure 2.2: part of a flowchart of a TM

from state q to state q' is (a,a',d), this means that if M reads a from the current square while in state q, it must write a', then take q' as its new state, and move the head by d (+1 for right, 0 for 'no move', and -1 for left). Thus, for each $(q,a) \in Q \times \Sigma$, if $\delta(q,a) = (q',a',d)$ then we draw an arrow from state q to state q', labelled with (a,a',d).

By allowing multiple labels on an arrow, as in figure 2.2, we can combine all arrows from q to q' into one. We can attach more than one label to an arrow either by listing them all, or (shorthand) by using a **variable** (s,t,x,y,z, etc.), and perhaps attaching conditions. So for example, the label '$(x,a,1)$ if $x \neq \wedge, a$' from state q to state q' in figure 2.3 below means that when in state q, if any symbol other than $\wedge$ or a is read, then the head writes a and moves right, and the state changes to q'. It is equivalent to adding lots of labels $(b,a,1)$, one for each $b \in \Sigma$ with $b \neq \wedge, a$.

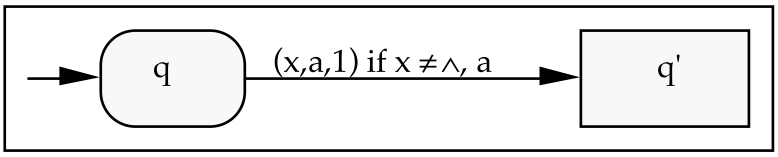


Figure 2.3: labels with variables

The starting state is indicated by an (unlabelled) arrow leading to it from nowhere (so q is the initial state in figure 2.3). All other arrows must have labels.

Exercises 2.6

1. No arrows leave any final state. How does this follow from definition 2.1? Can there be a non-final (i.e., round) state from which no arrows come, and what would happen if the TM got into such a state?
2. Figure 2.4 is a flowchart of the Turing machine of exercises 2.3 and 2.5 above. Try doing the exercises using the flowchart. Is it easier?

Warning Because δ is a function, each state of a flowchart should have *no more than one* arrow labelled $(a,?,?)$ leaving it, for any $a \in \Sigma$ and any values ?, ?. And if you forget an arrow or label, the machine might halt and fail wrongly.

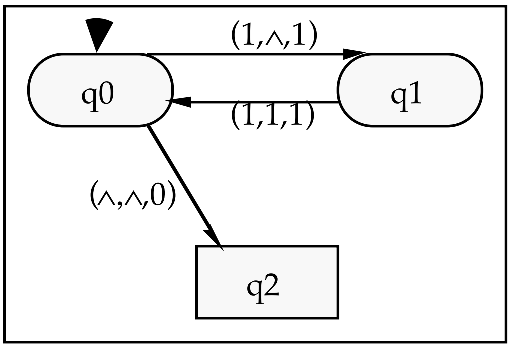


Figure 2.4: flowchart of TM of exercises 2.3, 2.5

2.3.2 Turing machines as pseudo-code

Another way of representing a Turing machine is in an imperative **pseudo-computer language.** The language is not a formal one: its syntax is usually made up as appropriate for the problem in hand.[1] The permitted basic operations are only Turing machine reads, writes and head movements. However, rather complicated control structures are allowed, such as **if-then** statements and **while** and **until** loops. A Turing machine usually implements if-then statements by using different states. It implements loops by repeatedly returning to the same state.

Warning Pseudo-code makes programming Turing machines less repetitive, as if-then structures etc. are needed very frequently. Many-track and many-tape machines (see later) are represented more easily.

However, there is a risk when writing pseudo-code that we depart too far from the basic state-changing idea of the Turing machine. The code must represent a real Turing machine. Whatever code we write, we must always be sure that it can *easily* be turned into an actual Turing machine. Assuming Church's thesis, this will always be possible; but *it should always be obvious how to do it.* For example,

```
solve the problem
halt & succeed
```

is not acceptable pseudocode, nor is

```
count the number of input symbols
if it is even then halt and succeed else halt and fail
```

(It is not obvious how the counting is done.) Nested loops are also risky — how are they implemented?

Halting: include a statement for halt & succeed, as above. For halt & fail, include a 'halt & fail' statement explicitly, or just arrange that no instruction is applicable.

[1]It has been formalised in some final-year and group projects.

2.3.3 Illustration

Example 2.7 (Deleting characters) Fix an alphabet I. Let us define a TM M with

$$f_M(w) = \mathsf{head}(w) \text{ for each } w \in I^*,$$

where the function head is as in definition 2.2. M will have three states: *skip*, *erase*, and *stop*. So $Q = \{skip, erase, stop\}$. *Skip* is the start state, and *stop* is the only halting state. We can take the alphabet Σ to be $I \cup \{\wedge\}$. δ is given by:

- $\delta(skip, x) = (erase, x, 1)$ for all $x \in \Sigma$,
- $\delta(erase, x) = (stop, \wedge, 0)$ for all $x \in \Sigma$.

So $M = (Q, \Sigma, I, skip, \delta, \{stop\})$. M is pictured in figure 2.5.

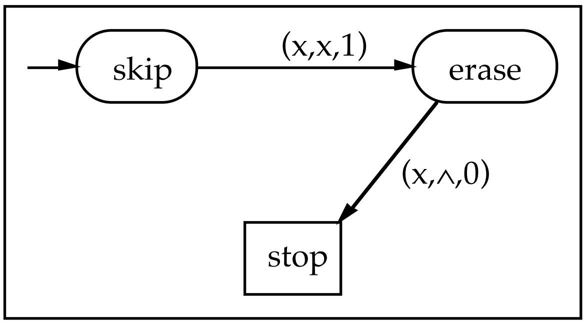


Figure 2.5: machine for $\mathsf{head}(w)$

The *names* of the states are not really needed in a flowchart, but they can make it more readable. In pseudo-code:

```
move right
write ∧
halt & succeed
```

Note the *close* correspondence between the two versions.

All M does is erase square 1. We did not need to erase the entire input word, because the output of a Turing machine is defined (§1.3.5, §2.2.3) to be the characters on the tape *up to one square before the first blank.* Here, we made square 1 blank, so the output will consist of the symbol in square 0, if it is not blank, or ε if it is.

Exercises 2.8 (Unary notation, unary addition) We can represent the number n on the tape by $111\ldots1$ (n times). This is **unary notation**. So 0 is represented by a blank tape, 2 by two 1s followed by blanks, etc. For short, we write the string $111\ldots1$ of n 1's as 1^n. In this course, 1^n will NOT mean $1 \times 1 \times 1 \ldots \times 1$ (n times). Note: 1^0 is ε.

1. Suppose $I = \{1, +\}$. Draw a flowchart for a Turing machine M with input alphabet I, such that $f_M(1^n. + .1^m) = 1^{n+m}$. (Remember that '.' means concatenation. E.g., if the input is '111+11', the output is '11111'.) So M adds unary numbers. (There is a suitable machine with 4 states. Beware of the case $n = 0$ and/or $m = 0$.)

2. Write a pseudo-code version of M.

2.4 Examples of Turing machines

We will now see more examples of Turing machines. Because Turing machines are so simple, programming them can be a tedious matter. Fortunately, over the years TM hackers have hit upon several useful labour-saving devices. The examples will illustrate some of these 'programming techniques' for TMs. They are:

- storing finite amounts of data in the state,
- multi-track tapes,
- subroutines.

Warning These devices are to help the programmer. *They involve no change to the definition of a TM.* (In section 3 we will consider genuine variants of the TM that make for even easier programming — though these are no more powerful in theory, as we would expect from Church's thesis.)

2.4.1 Storing a finite amount of information in the state

This is a very useful technique. First an example.

2.4.1.1 Shifting machines

Example 2.9 (Shifting a word to the right) We want a Turing machine M such that $f_M(w) = \mathsf{head}(w).w$ for all $w \in \{0,1\}^*$. So M shifts its input one square to the right, leaving the first character alone. E.g., $f_M(1011) = 11011$. See figure 2.6 for a solution.

The M above only works for inputs in $\{0,1\}^*$, but we could design a similar machine $M_I = (Q_I, I \cup \{\wedge\}, I, q_0, \delta_I, F_I)$ to shift a word of I^* to the right, where I is any finite alphabet. If I has more than 2 symbols then M_I would need more states than M above (how many?). But the idea will be the same for each I, so we would like to express M_I *uniformly* in I.

Suppose we could introduce into Q_I a special state $seen(x)$ with a **parameter**, x, that can take any value in I. We could then use x to remember the symbol just read. Using $seen(x)$, the table δ_I can be given very simply as follows:

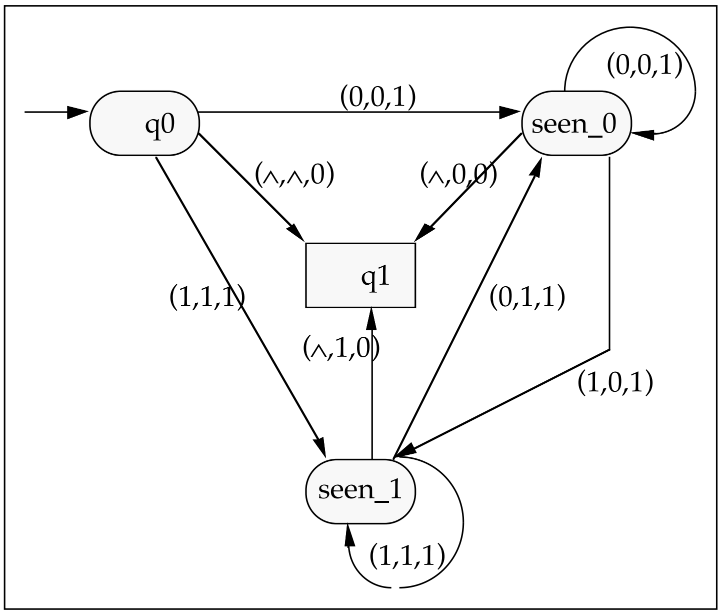


Figure 2.6: a shifter: $f_M(w) = \mathsf{head}(w).w$

- $\delta_I(q_0, a) = (seen(a), a, 1)$ for all a in I,
- $\delta_I(seen(a), b) = (seen(b), a, 1)$ for all a, b in I,
- $\delta_I(seen(a), \wedge) = (q_1, a, 0)$ for all a in I.

For an equivalent flowchart, see figure 2.7.

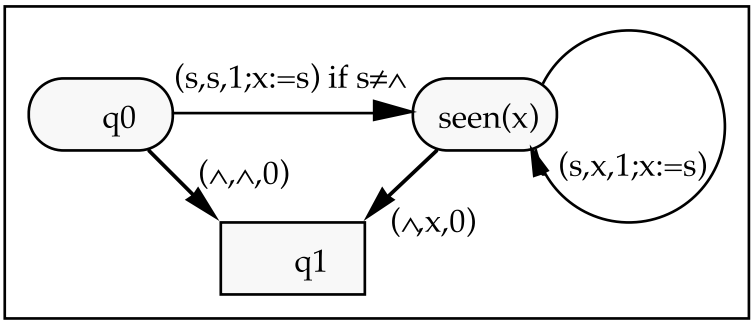


Figure 2.7: the ‘shifter’ TM drawn using parameters in states

Each arrow leading to $seen(x)$ is labelled with one or more 4-tuples. The last entry of each 4-tuple is an ‘assignment statement’, saying what x becomes when $seen(x)$ is entered.

The pseudo-code will use a variable x. x can take only finitely many values. We need not mention the initial write, as we only need specify writes that actually alter the tape.

```
read current symbol and put it into $x$
move right
repeat until $x = \wedge$
    swap current symbol with $x$
    move right
end repeat
halt & succeed
```

This will work for any I.

2.4.1.2 Using parameters in states is legal

In fact we can use states like $seen(x)$ without changing the formal definition of the Turing machine at all! We just observe that whilst it's convenient to view $seen(x)$ as a single state with a parameter x, we could equally get away with the collection $seen(a), seen(b), \ldots$ of states, one for each letter in I, if we are prepared to draw them all in and connect all the arrows correctly. This is a bit like multiplication: 3×4 is convenient, but if we only have addition we can view this as shorthand for $3+3+3+3$.

What we do is this. For each letter a of I we introduce a *single state,* called $seen(a)$, or if you prefer, $seen_a$. Because I is finite, this introduces only finitely many states. So the resulting state set is finite, and so is allowed by the definition of a Turing machine. In fact, if $I = \{a_1, \ldots, a_n\}$ then $Q_I = \{q_0, q_1, seen(a_1), \ldots, seen(a_n)\}$: i.e., $n+2$ states in all. Then δ_I as above is just a partial function from $Q_I \times (I \cup \{\wedge\})$ into $Q_I \times (I \cup \{\wedge\}) \times \{0, 1, -1\}$. So our machine is $M_I = (Q_I, I \cup \{\wedge\}, I, q_0, \delta_I, F)$ — a genuine Turing machine!

So although $seen(x)$ is conveniently viewed by us as a single state with a parameter ranging over I, for the Turing machine it is really many states, namely $seen(a_1)$, $seen(a_2)$, … $seen(a_n)$, one for each element of I.

So we can in effect allow parameters x in the states of Turing machines, *so long as x can take only finitely many values.* Doing so is just a useful piece of notation, to help us write programs. This notation represents the idea of storing a *bounded finite* amount of information in the state (as in the registers on a computer).

Warning We cannot store any parameter x that can take infinitely many values, or even an unbounded finite number of values. That would force the underlying genuine state set Q to be infinite, in contravention of the definition of a Turing machine. So, e.g., for any I, we get a Turing machine M_I that works for I. M_I is built in a uniform way, but we do *not* (cannot) get a *single* Turing machine M that works for any I! Similarly, we cannot use a parameter in a state to count the length of the input word, since even though the length of the input is always finite, there is no finite upper bound on it.

2.4.1.3 What we can do with parameters in states

Example 2.10 (Testing whether two strings are equal) We will design a Turing machine M with input alphabet I, such that $f_M(w_1,w_2)$ is defined if $w_1 = w_2$ (but we don't care what value it has), and undefined otherwise. That is, M halts & succeeds if $w_1 = w_2$, and halts & fails, or never halts, if $w_1 \neq w_2$.

First, how can a TM take more than one argument as input? We saw in exercise 2.8 a TM to calculate $n+m$ in unary. Its arguments were 1^n and 1^m, separated by '$+$'. So here we assume that I contains a delimiter, '$*$', say, and w_1, w_2 are words of I not containing '$*$'. That is, the input tape to M looks like figure 2.8.

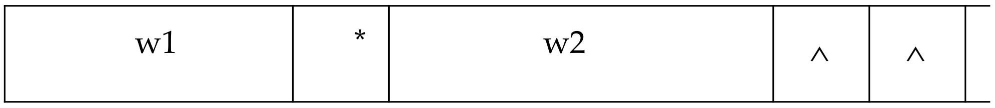


Figure 2.8: initial tape of M

We will use a parameter to remember the last character seen. We will also need to tick off characters once we have checked them. So we let M have full alphabet $\Sigma = I \cup \{\wedge, \surd\}$, where $\surd$ ('tick') is a new character not in I. We will overwrite each character with $\surd$, once we've checked it. Figure 2.9 shows a flowchart for M.

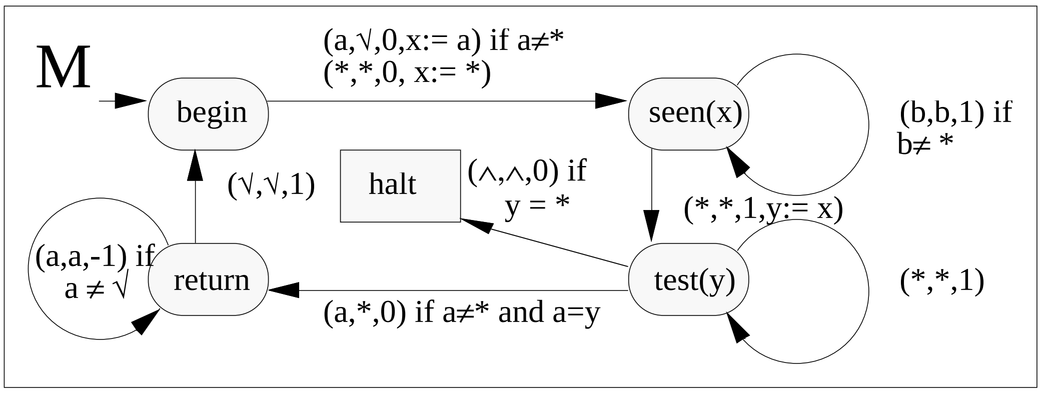


Figure 2.9: TM to check if $w_1 = w_2$

M overwrites the leftmost unchecked character of w_1 with $\surd$, passing to state *seen*(x) and remembering what the character was using the parameter x of 'seen'. (But if x is $*$, this means it has checked all of w_1, so it only remains to make sure there are no more uncompared characters of w_2.) Then it moves right until it sees $*$, when it jumps to state *test*(y), remembering x as y. In this state it moves past all $*$'s (which are the checked characters of w_2). It stops when it finds a character — a, say — that isn't $*$ (i.e., a is the first unchecked character of w_2). It compares a with y, the remembered character of w_1. There are three possibilities:

1. $a = \wedge$ and $y = *$. So all characters of w_1 have been checked against w_2 without finding a difference, and w_2 has the same length as w_1. Hence $w_1 = w_2$, so M halts and succeeds (state halt).
2. $a \neq y$. M has found a difference, so it halts & fails (there's no applicable instruction in state $test(y)$).
3. $a = y$ and $y \neq *$. So the characters match. M overwrites the current character (a) with $*$, and returns left until it sees a $\surd$. One move right then puts it over the next character of w_1 to check, and the process repeats.

Exercises 2.11

1. Try M on the inputs 123∗123, 12∗13, 1∗11, 12∗1, ∗1, 1∗, and ∗ (in the last three, w_1, w_2, or both are empty (ε)). What is the output of M in each case?
2. What would go wrong if the 'begin $\rightarrow$ seen' arrow was just labelled $(a, \surd, 0, x := a)$?

Please don't worry if you found that hard; Turing machines that need as many as five states (not counting any parameters) are fairly rare, and anyway we'll soon see ways to make things easier. By the way, it's a good idea to write your Turing machines using as few states as you can.

3. Design a Turing machine T_I to calculate the function $\mathsf{tail} : I^* \rightarrow I^*$.
4. Design a Turing machine M that checks that the first character of its input does not appear elsewhere in the input. How will you make M output the answer?

2.4.2 Multiple tracks

Above, we found it convenient to put (finite amounts of) data in the state of a Turing machine. So a state took the form $q(x)$ or (q,x), where x could take any of finitely many values. Then we could specify the instruction table more easily.

In the same way, many problems would be simpler to solve with Turing machines if we were allowed to use a tape with more than one track — as on a stereo cassette, which has four tracks all told. The string comparison example shows how useful this can be. The 'M' of figure 2.9 above was pretty complex. Wouldn't it be easier to use two tracks?

As before, let's cheat for a moment and do this. We would like the tape of M to have two tracks, with w_1 on the first track and w_2 on the second track, as shown in figure 2.10.

Then M can simply move its head along the tape, testing at each stage whether the characters in tracks 1 and 2 are the same. See figure 2.11. We write (x,y) as notation for a square having x in track 1 and y in track 2. M halts and fails if it finds a square with different symbols in tracks 1 and 2.

The two-track M is much easier to design. So it might be useful for Turing machines in general to be able to have a multi-track tape.

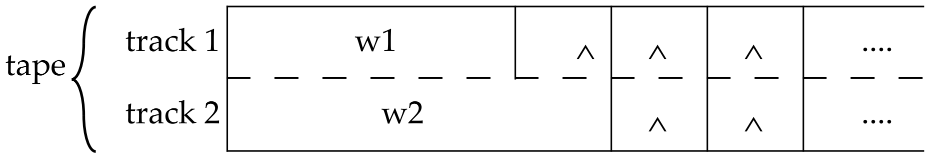


Figure 2.10: two-track tape for word-comparison TM, *M*

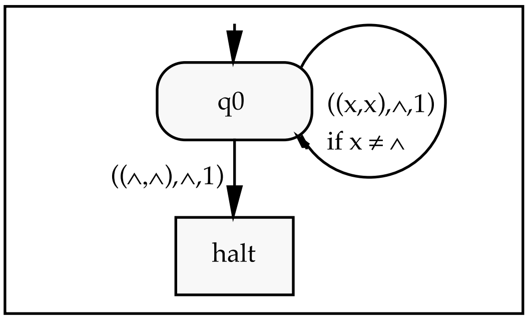


Figure 2.11: flowchart for *M*

2.4.2.1 Using tracks is legal

In fact, as with states, *we can effectively divide the tape into tracks without modifying the formal definition of the Turing machine.* To divide the tape into n tracks, we add a finite number of new individual symbols of the form $(a_1,\ldots,a_n)$ to Σ, where $a_1,\ldots,$ a_n are any symbols. Each $(a_1,\ldots,a_n)$ is a single symbol, in Σ, and may be written to and read from the tape as usual. But whenever $(a_1,\ldots,a_n)$ is in a square, we can *view* this square as divided into n parts, the ith part containing the 'single' symbol a_i. So if $n = 2$ say, and many squares have pairs of the form (x,y) in them, the tape begins to look as though it is divided into two tracks (figure 2.12):

If the only symbols on the tape are $\wedge$ and symbols of the form $(a_1,\ldots,a_n)$, we can consider the tape as actually divided into n tracks. Note that $\wedge \neq (\wedge,\wedge)$.

Warning The tuples $(a_1,\ldots,a_n)$ are just single symbols in the Turing machine's alphabet. The tracks only help us to think about Turing machine operations — they exist only in the mind of the programmer. *No change to the definition of a Turing machine has been made.* Compare arrays in an ordinary computer. The array $A(5,6)$ will usually be implemented as a 1-dimensional memory area of 30 contiguous cells. The division into a 2-dimensional array is done in software.

Warning We cannot divide the tape into infinitely many tracks — this would violate the requirement that Σ be finite. (But see 2-dimensional-tape Turing machines in §3.4.1.)

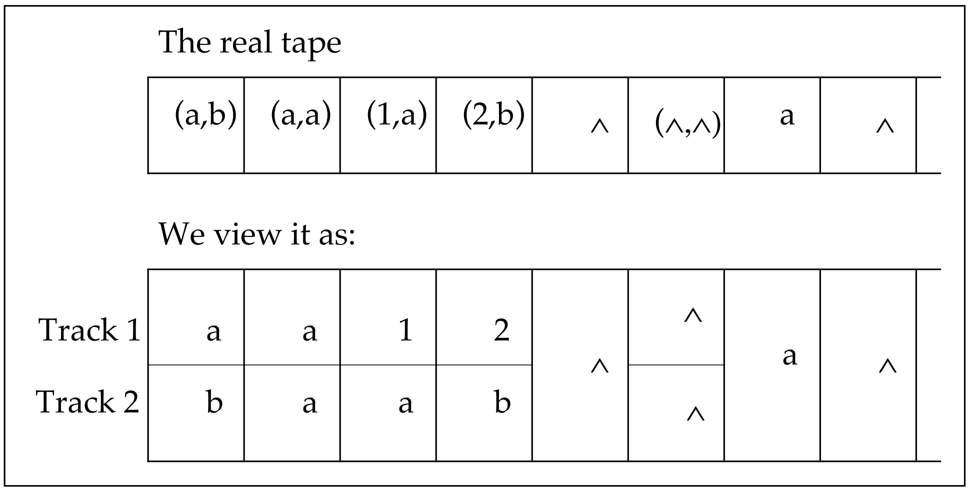


Figure 2.12: tracks on tape

2.4.2.2 What we can do with tracks

Because a Turing machine can write and move according to exactly what it reads, it can effectively read from and write to the tracks independently. Thus e.g., it can shift a single track right by one square (cf. example 2.9). In fact, anything we can do with a 1-track machine we can also do on any given track of a multi-track machine.

Cross-track operations are also possible. For example, this Turing machine copies track 1 as far as its first blank to track 2 (figure 2.13):

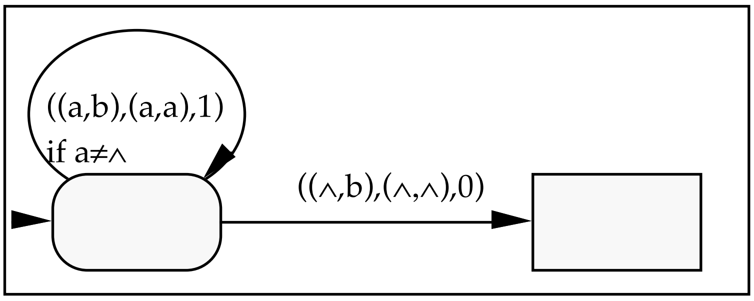


Figure 2.13: track copier

2.4.2.3 String comparison revisited

Now let's see in detail how to solve the string comparison problem using 2 tracks. The input is $w_1 * w_2$ as before: all on one track. See figure 2.14.

The Turing machine we want will have three stages:

Stage 1: replace the 1-track input by 2 tracks, with w_1 left-justified on track 1, and w_2, with $len(w_1)+1$ blanks before it, on track 2). This part can be done much

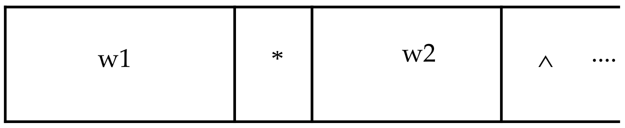


Figure 2.14: initial tape contents

as in figure 2.13 (how exactly?) Then return to square 0. The resulting tape has 2 tracks as far as the input went; after that, it has only one. Also, while we're at it, we mark square 0 with a '$*$' in track 2 (figure 2.15).

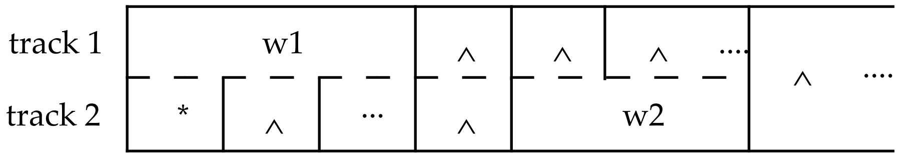


Figure 2.15: tape after Stage 1

Stage 2: shift w_2 left to align it with w_1. E.g., use some version of tail (exercise 2.11)(3) repeatedly, until the $*$ is gone (see figure 2.16).

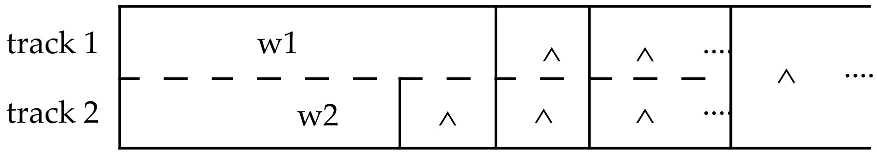


Figure 2.16: tape after Stage 2

Stage 3: compare the tracks as far as their first $\wedge$'s, halting & failing if a difference is found. This is easy — see figure 2.11.

Exercise: work out the details.

So comparing two words is easier with two tracks. But tapes with more than one track are useful even if there's only one input. An example is implicit marking of square 0 (§2.4.3 below); we'll see others in section 3.

2.4.2.4 Setting up and removing tracks

In the string comparison example 2.10 above, the two arguments w_1, w_2 were provided on a 1-track tape, one after the other (figure 2.8/2.14). We then put them on different tracks (figure 2.15). If there were 16 arguments, we could put them left-justified on a 16-track tape in a similar way (think about how to do it).

But often it is best to set up tracks *dynamically* — as we go along. This saves doing it all at the beginning. (Besides, however much of the tape we set up as 2 tracks initially, we might want to use even more of the tape later, so every so often we'd have to divide more of the tape into tracks, which is messy.)

So, each time our machine enters a square that is not divided into 2 tracks (i.e., doesn't have a symbol of the form (a,b) in it), it immediately replaces the symbol found — a, say — by the pair $(a,\wedge)$, and then carries on. This is so easy to do (just add instructions of the form $(q,a,q,(a,\wedge),0)$ to δ, for all non-pairs $a \in \Sigma$) that we won't often mention the setting up of tracks explicitly.

Similarly, when M has finished its calculations using many tracks, the output will have to be presented on a single track tape, as per the definition of output in §2.2.3. Assuming that the answer is on track 1, M will erase all tracks but the first, so that the tape on termination has a single track that looks like track 1 of the 'scratch' tape. It need only do this as far as the first $\wedge$ in track 1. See figure 2.17 for how to do this with a three-track scratch tape, assuming M has brought its head to square 0:

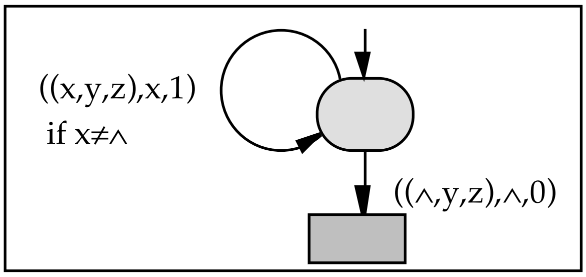


Figure 2.17: returning to a single track

2.4.3 Implicit marking of square 0 of the tape

Our Turing machines halt and fail if they try to leave the left hand end of the tape. As we may wish to avoid a halt & fail, it helps when programming to be able to tell when the head is in square 0. We have seen the need for this in the examples. We want to **mark square 0** with a special symbol, '$*$', say. Then when the head reads '$*$', we know it is in square 0.

But square 0 may contain an important symbol already, which would be lost if we simply overwrote it with '$*$'.

There are several ways to manage here:

1. Create an extra track, with '$*$' in square 0 and blanks in the remaining squares. To see if the head is in square 0, just read the new track.

2. For each a in Σ add a new character '$a*$' (or '$(a,*)$') to Σ. To initialise, replace the character b in square 0 by $b*$. From then on, write a starred character iff you read one. So square 0 is always the only square with a starred character. This is much the same as adding an extra track.

3. Include $*$ as a special character of Σ. To initialise, shift the input right one place and insert $*$ in square 0. Then carry out all operations on squares 1,2,..., using $*$ as a left end marker. This works OK, but involves some tedious copying, so is not recommended when designing actual TMs!

4. Write your TM carefully so it doesn't need to return to square 0. This is possible surprisingly often, but few can be bothered to do it.

2.4.3.1 Convention

Because we can always know when in square 0 (by using one of these ways), we will assume that a Turing machine always knows when its head is over square 0 of the tape. square 0 is assumed to be **implicitly marked.** This saves us having to mention the extra track explicitly when describing the machine, and so keeps things simple.

2.4.3.2 Examples of implicit marking (fragments of TMs)

```
repeat until read a or square 0 reached
    write a
    move left
end repeat
halt & succeed
```

For a flowchart, see figure 2.18.

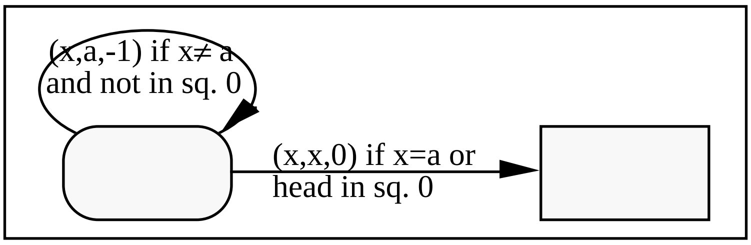


Figure 2.18: implicit marking of square 0 in flowcharts

We often need to return the head to square 0. This can be done very simply, using a loop:

```
move left until in square 0
```

Nonetheless, my own view is that '`return to square 0`' is too high-level pseudo-code (see the *warning* in §2.3.2), and should not be used.

2.4.4 Subroutines

It is quite in order to string several Turing machines together. Informally, when a final state of one is reached, the state changes to the initial state of the next in the chain. This can be done formally by collecting up the states and instruction tables of all the machines in the chain, and for each final state of one machine, adding a new instruction that changes the state to the initial state of the next machine in the chain, without altering the tape or moving the head. The number of states in the 'chain' machine is the sum of the numbers of states for the individual machines, so is finite. Thus we obtain a single Turing machine from the machines in the chain; again we have not changed the basic definition of the Turing machine. We will use this technique repeatedly.

Warning When control passes to the next Turing machine in the chain, the head may not be over square 0. Moreover, the tape following the 'input' may contain the previous machine's scratchwork and so not be entirely blank. Each machine's design should allow for these possibilities, e.g., by returning the head to square 0 before starting, or otherwise.

2.4.5 Exercises

We end this section with some problems that illustrate the techniques we have seen here.

Exercises 2.12

1. (subtraction) Suppose that $I = \{1, -\}$. Design a Turing machine $M = (Q, \Sigma, I, q_0, \delta, F)$ such that

$$f_M(1^n.-.1^m) = \begin{cases} 1^{n-m} & \text{if } n \geq m, \\ \varepsilon, & \text{otherwise.} \end{cases}$$

   M performs subtraction on unary numbers.

2. (unary multiplication) Suppose $I = \{1, *\}$. Design a Turing machine M such that $f_M(1^n.*.1^m) = 1^{nm}$. Hint: use 3 tracks and repeated addition and subtraction.

3. (inverting words) Let I be an alphabet. Find a Turing machine $M = (Q, \Sigma, I, q_0, \delta, F)$ such that $f_M(w)$ is the reverse of w. E.g.: use storage in states and marking of square 0.

4. (unary-binary conversion)

   (a) Design a machine M to add 1 to a binary number (much easier than in decimal!). That is, if if $n > 0$ is a number let 'n' $\in \{0,1\}^*$ be the binary expansion of n, without leading zeros, written with the least significant digits on the left. (E.g., '13' = 1011. This makes things easier.) Define '0' to be a single zero. We then require $f_M(\text{'}n\text{'}) = \text{'}n+1\text{'}$ for all $n \geq 0$.

   (b) Extend M to a machine that converts a unary number to its binary equivalent. Hint: use two tracks.
   (c) Design a Turing machine that converts a binary number to unary.

5. (primality testing) Design a TM that, given as input some binary number, tests whether it is prime. [Again, a 3-track machine is useful. See Hopcroft & Ullman, p. 154].

2.5 Summary of section

We defined a Turing machine formally, as a finite state machine with a finite symbol alphabet and a 1-way infinite tape. We explained why we chose it as our formalisation of algorithm, and how it is idealised from a real computer. We discussed Church's thesis.

We explained input and output for Turing machines and defined the input-output function f_M for a Turing machine M. We saw that a Turing machine can be represented as a flowchart or by pseudo-code. We gave examples of Turing machines that solve particular problems: unary-binary conversion, arithmetical operations, etc. We considered ways of programming Turing machines more easily: storing finite amounts of data in the state set (cf. registers), using many tracks on the tape (cf. arrays), and chaining Turing machines (cf. subroutines).

3. Variants of Turing machines

In this section we examine some variants of the TM we considered before. The main examples we have in mind are machines with a two-way infinite tape, or more than one tape. We will see that in computational power they are all the same as the ordinary model. This is in line with Church's thesis, and provides some evidence for the 'truth' of the thesis.

Nonetheless, variants of the basic Turing machine are still useful. Just as in real life, the more complex (expensive) versions can be easier to program (user-friendly), whilst the simpler, cheaper models can be easier to understand and so prove things about. For example, suppose we wanted to prove that 'register machines' are equivalent in power to Turing machines. This amounts to showing that Turing machines are no better and no worse than register machines (with respect to computational power). We could show this directly. But clearly it might be easier to prove that a *cheap* Turing machine is no better than a register machine, which in turn is no better than an *expensive* Turing machine. As in fact both kinds of Turing machine have equal computational power, this is good enough.

First we need to make precise what we mean by equal computational power.

3.1 Computational power

Comparing two different kinds of machine can be like comparing a car and a cooker. How can we begin? But a function $f : I^* \to \Sigma^*$ is a more abstract notion than a machine. We will compare different kinds of computing machine by comparing their *input-output functions,* as with the formal version of Church's thesis (§2.2.4). To show that cheap and expensive Turing machines are equivalent, we will show that any function computable by one kind is computable by the other.

Formally:

Definition 3.1 Let M_1, M_2 be Turing machines, possibly of different kinds, with the same input alphabet. We say that M_1 and M_2 are **equivalent** if $f_{M_1} = f_{M_2}$. That is, M_1 and M_2 compute the same function. They have the same input-output function.

So to show that two kinds of Turing machine have equal computational power, we will show that for any machine of one kind there is an equivalent one of the other kind.

3.1.1 Proving different machines have equal computational power

As one might expect, it is usually easy to show that an expensive machine M^+ can compute any function that a cheap machine M can. We must work harder to prove that any algorithm performable by an expensive M^+ can done by a cheaper machine M. We will see several examples below.

The details of the proofs are not so important. The *important point* is that in each case, although M^+ is (presumably) solving some problem, we do *not* try to make the cheap machine M solve the problem directly. Instead we cheat and make M mimic or **simulate** M^+, parrot fashion. The same happens when a Macintosh emulates a PC. We need no deep understanding of what kind of algorithms M^+ can perform, but only of the nuts-and-bolts design of M^+ itself. This is a very profound idea, and we will see it again later (UTMs in section 4, and reduction, in section 5 and Part III of the course).

Aside: it helps if M^+ is not too much more complex than M. So getting a 1-tape Turing machine M to imitate a Cray YMP would be best done by going through several increasingly complex machine designs $M_1, \ldots, M_n$. We would show that an ordinary Turing machine M is equivalent to M_1 (each can simulate the other), M_1 is equivalent to M_2, …, M_{n-1} is equivalent to M_n, and that M_n is equivalent to the Cray. This will show that a Turing machine is equivalent to a Cray.

3.2 Two-way-tape Turing machines

We could easily (with a little more cash) allow the tape of a Turing machine to be infinite in both directions. In fact this is a common *definition* of 'Turing machine', and is used in Rayward-Smith's book (our definition, using a one-way infinite tape, is used in Hopcroft & Ullman's).

Here's a picture of a 2-way infinite tape Turing machine:

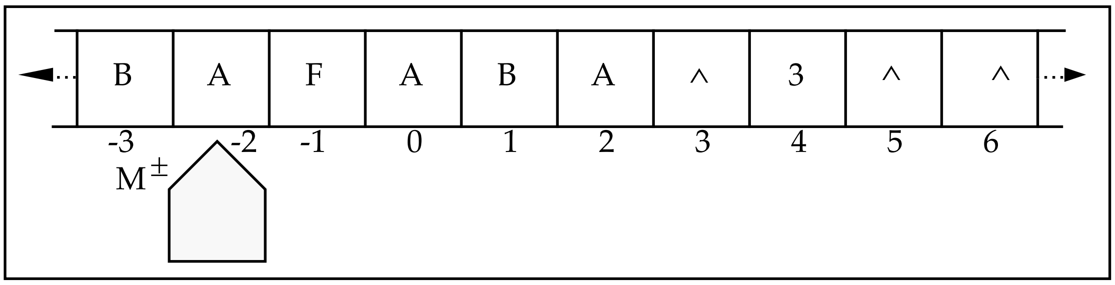


Figure 3.1: a two-way infinite tape TM, $M^{\pm}$

Definition 3.2 (Two-way-infinite tape TM) A **two-way infinite tape Turing machine** has the form $M^{\pm} = (Q, \Sigma, I, q_0, \delta, F)$, exactly as before. The tape now goes right and left forever, so the head of $M^{\pm}$ can move left from square 0 to squares $-1, -2$, etc., without halting and failing. (*You can't tell* from the 6-tuple definition what kind of tape the machine has; this information must be added as a rider. Of course, by default the tape is 1-way infinite, as that's our definition of a Turing machine.)

The *input* to $M^{\pm}$ is written initially in squares $0, 1, \ldots, n$. All squares $> n$ and < 0 are blank. If $M^{\pm}$ terminates, the output is taken to be whatever is in squares $0, 1, \ldots, m-1$, where the first blank square ≥ 0 is in square m. So we can define the input-output function $f_{M^{\pm}}$ for a two-way infinite tape machine as before.

Exercise 3.3 (This is too easy!) Since we can't tell from the 6-tuple definition what kind of tape the machine has, we can alter an ordinary TM by giving it a two-way infinite tape to run on: the result is a working machine. Find a 1-way infinite tape Turing machine that has a different input-output function if we give it a two-way infinite tape in this way.

Now 2-way infinite tape Turing machines still seem algorithmic in nature, so if Church's thesis is true, they should be able to compute exactly the same functions as ordinary Turing machines. Indeed they can: *for every two-way infinite Turing machine there is an equivalent ordinary Turing machine, and vice versa.* But we can't just quote Church's thesis for this, as we are still gathering evidence for the thesis! We must prove it. If we can do this, it will provide some type (b) evidence (see §1.5.3) for the correctness of Church's thesis as a definition of **algorithm.**

Two-way machines seem intuitively more powerful than ordinary ones. So it should be easy to prove:

Theorem 3.4 *If $M = (Q, \Sigma, I, q_0, \delta, F)$ is an ordinary Turing machine, then there is a two-way infinite Turing machine $M^{\pm}$ equivalent to M.*

And it is. We take $M^{\pm} = (Q, \Sigma \cup \{\text{fail}\}, I, q_0, \delta^{\pm}, F)$, where 'fail' is a new symbol not in Σ. $M^{\pm}$ begins by moving left to square -1, writing 'fail' there, and moving right to square 0 again. Then it behaves exactly as M, except that if it ever reads 'fail' it halts and fails. Clearly $f_M = f_{M^\pm}$. QED.

As we might expect, the converse is a little harder.

Theorem 3.5 *Let $M^\pm = (Q, \Sigma, I, q_0, \delta, F)$ be a two-way infinite Turing machine. Then there is an ordinary Turing machine M equivalent to $M^\pm$.*

PROOF. The idea is to curl the *two-way* tape round in a U-shape, making it *1-way infinite but with two tracks.* The top track will have the same contents as squares 0,1,2,...of the two-way infinite tape of $M^\pm$. The bottom track will have a special symbol '$*$' in square 0, to mark the end of the tape, and squares $1, 2, \ldots$ will contain the contents of squares $-1, -2, \ldots$ of $M^\pm$'s tape. See figure 3.2.

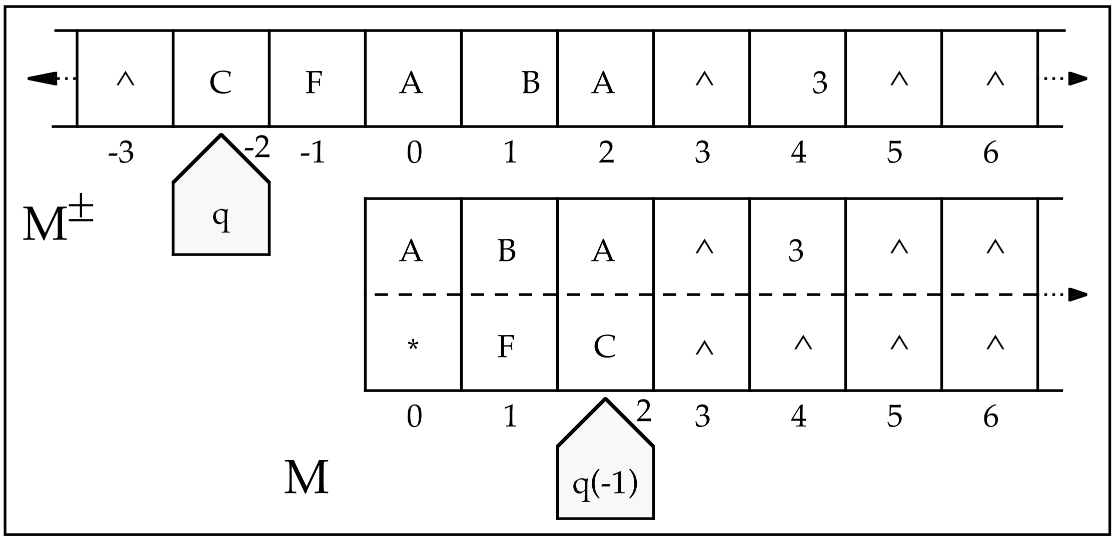


Figure 3.2: tape contents of $M^\pm$ and its shadow

The 1-way tape of M holds the same information as $M^\pm$'s 2-way tape. M will use it to follow $M^\pm$, move for move. It keeps track of whether $M^\pm$ is currently left or right of square 0, by remembering this (a finite amount of information!) in its state, as in §2.4.1.

In pseudo-code, it is quite easy to specify M. The variable `track` will hold 1 if $M^\pm$ is now reading a positive square or 0, and -1 if $M^\pm$ is reading a negative square. For M, $+1$ means 'top track' and -1 means 'bottom track'. The variable $M^\pm$`-state` holds the state of $M^\pm$. Note that these variables can only take finitely many values, so they can be implemented as a parameter in the states of M, as in §2.4.1. Remember (§2.4.2.1) that in reality, two-track squares of M's tape hold pairs (a, b), where we view a as the contents of the top track, and b the contents of the bottom track.

```
track := 1;   M±-state := q_0            % initially M± reads square 0 in state q_0
if reading a then write (a,*)             % initialise square 0
repeat until M±-state is a halting state of M±
```

```
    if the current square contains $a$ (say), and $a$ is not a pair of symbols, then
        if track $= 1$ then write $(a, \wedge)$ else write $(\wedge, a)$ end if        % dynamic track set-up
    end if
    if track $= 1$ then                          % $M^{\pm}$ is reading a square $\geq 0$
        case [we are reading $(a,b)$ and $\delta(M^{\pm}$-state$,a) = (q,a',d)$]:        % $\delta$ from $M^{\pm}$
            write $(a',b)$                       % write to top track
            $M^{\pm}$-state $:= q$
            if $b = *$ and $d = -1$ then         % $M^{\pm}$ in square 0 and moving left...
                move right; track $:= -1$
            else
                move in direction $d$            % track $= 1$ so $M$ moves the same way as M
            end if
        end case
    else if track $= -1$ then                    % $M^{\pm}$ is reading a square $< 0$
        case [we are reading $(a,b)$ and $\delta(M^{\pm}$-state$,b) = (q,b',d)$]:
            write $(a,b')$                       % write to bottom track
            $M^{\pm}$-state $:= q$
            move in direction $-d$               % track $= -1$, so $M$ moves 'wrong' way
            if now reading $*$ in track 2 then track $:= 1$        % $M^{\pm}$ now in square 0
        end case
    end if
end repeat
% $M^{\pm}$ has halted & succeeded, so clean up & output
move left until read $*$ in track 2              % return to square 0
repeat while not reading $\wedge$ in track 1
    if reading $(a,b)$ (say) then write $a$      % replace two tracks with one
    move right
end repeat
write $\wedge$; halt & succeed                   % blank to mark end of output
```

So M mimics $M^{\pm}$, move for move. Note that the case statements involve a fixed finite number of options, one case for each triple (q,a,b) where $q \in Q$ and $a,b \in \Sigma$. So we can implement them by 'hard-wiring', using finitely many states of M. We stipulate that if no option applies, the case statement halts & fails.

When $M^{\pm}$ halts and succeeds (if ever!), M removes the bottom track, the old top track up to its first $\wedge$ (which is $M^{\pm}$'s output) becoming the whole width of the tape. Thus, the output of M is the same as $M^{\pm}$ in all cases, and so M is equivalent to $M^{\pm}$. QED.

Try this on a simple example, following M's attempts to keep up with $M^{\pm}$. What happens if $M^{\pm}$ halts and fails?

3.3 Multi-tape Turing machines

With a lot more cash we could allow our machines to have more than one tape. Figure 3.3 shows a picture of a 3-tape Turing machine. Notice how it differs from a 1-tape machine with 3 tracks. Here, there are *three heads* which can move *independently* on their own tapes. At each step:

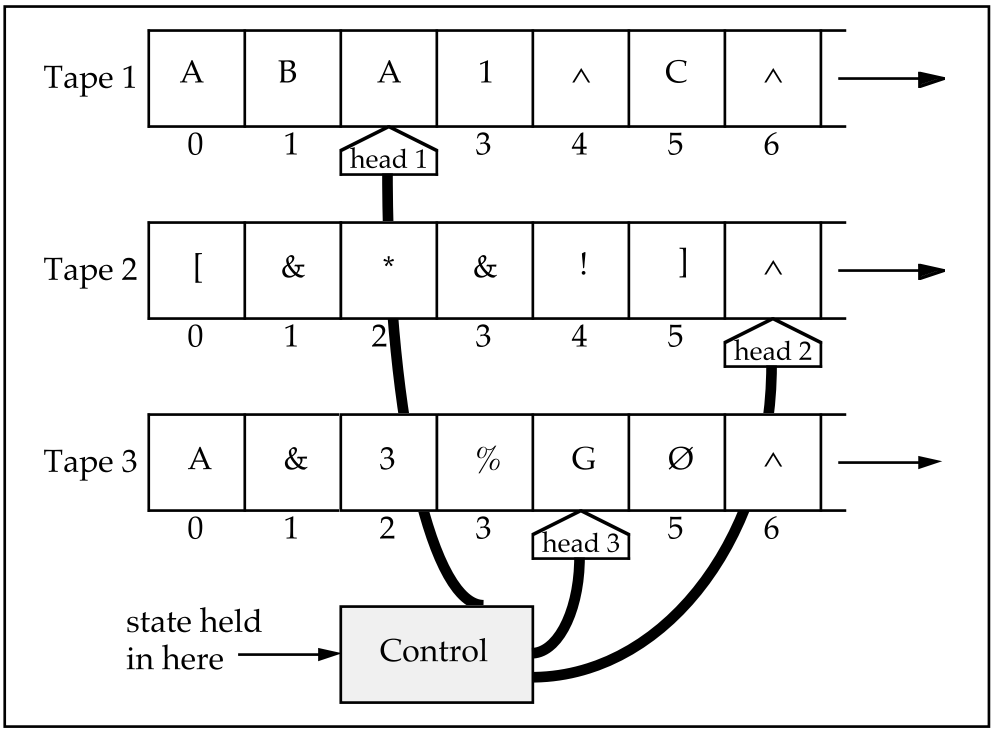


Figure 3.3: a three-tape TM caught in action

- All 3 heads read their squares; the 3 symbols found are passed to control.
- Depending on these 3 symbols and on its current state, control then:
  - tells each head what to write;
  - tells each head which way to move;
  - moves into a new state.
- The process repeats.

The moves, new symbols and state are determined by the **instruction table,** and depend on the old state and *all three* old symbols. So what one head writes can depend on what the other heads just read. In many ways, a many-tape Turing machine is analogous to *concurrent execution*, as in effect we have several communicating Turing machines running together.

Warning Do not confuse **multi-tape** TMs with **multi-track** TMs. I know they are similar as English words, but these names are in general use and we are stuck with them. They mean quite different things. Think of tape recorders. Two old mono tape recorders (2 tapes, with 1 track each) are *not the same* as one stereo tape recorder (1 tape, 2 tracks). They are in principle *better,* because (a) we could synchronise them to get the capabilities of a single stereo machine, but (b) we can use them in other ways too, e.g., for editing. Similar considerations apply to Turing machines.

3.3.1 The multi-tape machine formally

A 3-tape machine can be written formally as $M = (Q,\Sigma,I,q_0,\delta,F)$, where all components except solely for the instruction table δ are as for the usual Turing machine. In the 3-tape machine, δ is a partial function

$$\delta : Q\times\Sigma\times\Sigma\times\Sigma \to Q\times\Sigma\times\Sigma\times\Sigma\times\{-1,0,1\}\times\{-1,0,1\}\times\{-1,0,1\}.$$

The definition of the n-tape machine is the same, except that we have:

$$\delta : Q\times\Sigma^n \to Q\times\Sigma^n\times\{-1,0,1\}^n.$$

Here, and below, if S is any set then S^n is the set $\{(a_1,\ldots,a_n) : a_1,\ldots,a_n \in S\}$.

3.3.1.1 Remarks

1. For $n = 1$, this is the same Turing machine as in definition 2.1.

2. How can you tell how many tapes the Turing machine $(Q,\Sigma,I,q_0,\delta,F)$ has? Only by looking at δ. If δ takes 1 state argument and n symbol arguments, there are n tapes.

3. Note that it is NOT correct to write, e.g., $(Q^3,\Sigma^3,I^3,q_0,\delta,F^3)$ for a 3-tape machine. *One Turing machine, one state set, one alphabet.* There is a single state set in a 3-tape Turing machine, and so we write it as Q. If each of the three heads were in its own state from Q, then the state of the whole machine would indeed be a triple in Q^3. But remember, everything is linked up, and one head's actions depend on what the other heads read. With so much instantaneous communication between heads, it is meaningless to say that they have individual states. *The machine as a whole* is in some state — some q in Q.

   And there is one alphabet: the set of characters that can occur in the squares on the tapes. We used Σ^3 when there were 3 tracks on a single tape, because this involved changing the symbols we were allowed to write in a single square of a tape. So if you write Σ^3, you should be thinking of a 3-*track* machine. In fact, after using 3-tape machines for a while you won't want to bother with tracks any more, except to mark square 0.

3.3.1.2 Computations

How does a many-tape TM operate? Consider (for instance) a 3-tape machine $M = (Q, \Sigma, I, q_0, \delta, F)$. At the beginning, each head is over square 0 of its tape. Assume that at some stage, M is in state q and reads the symbol a_i from head number i (for each $i = 1,2,3$). Suppose that

$$\delta(q, a_1, a_2, a_3) = (q', b_1, b_2, b_3, d_1, d_2, d_3).$$

Then for each $i = 1,2,3$, head i will write the symbol b_i in its current square and then move in direction d_i (0 or ± 1 as usual), and M will go into state q'. M halts and succeeds if q' is a halting state. It halts and fails if there is no applicable instruction, or if any of the three heads tries to move left from square 0. The definition of an n-tape machine is similar.

3.3.1.3 Input/output. The function computed by M

The input, a word w of I, is placed left-justified on tape 1, with only $\wedge$s afterwards. All other tapes are blank. If M halts and succeeds, the output $f_M(w)$ is taken to be whatever is on tape 1 from square 0 up to the character before the first blank. If M doesn't halt & succeed, its output on w is undefined. So as before, f_M is a partial function from I^* into Σ^*: the **function computed by M.**

Notice that the input-output conventions are as for an ordinary Turing machine. So an n-tape machine is just an ordinary Turing machine if $n = 1$.

3.3.2 Old tricks on the new machine

We can write many-tape machines as flowcharts. For a 2-tape machine, the labels on arrows will be 6-tuples from $\Sigma^4 \times \{0,1,-1\}^2$, written

$$((a,b),(a',b'),(d,d')),$$

which means **take this arrow if head 1 reads a and head 2 b; get head 1 to write a' and head 2 b', and move head 1 in direction d $(0$ or $\pm 1)$ and head 2 in direction d'.** Some people prefer to write this label as (a,a',d,b,b',d'), dealing with tape 1 first, then tape 2. This obscures the fact that each tape's write depends on what *all* the heads read. Whichever notation you use, you should explain it.

We can also use pseudo-code; this is very suitable as it can easily refer to individual tapes (as for tracks). Indeed, all the programming techniques we saw in section 2 can still be used for multi-tape machines. For example, we can store information in states, and can divide each tape into tracks (each tape can have a different number of tracks). So e.g., by adding one extra track to each tape and putting $*$ in square 0 of that track, a many-tape machine can tell at any stage whether one of its heads is in square 0. Therefore, we continue to allow square 0 of each tape to be implicitly marked.

However, having many tapes is itself one of the most useful 'tricks' of all for programming Turing machines. For this reason we will use many-tape machines very often. An example may illustrate how they can help.

Example 3.6 (detecting palindromes (a yes/no problem)) Let I be an alphabet with at least 2 symbols. A **palindrome** of I is a word w of I such that w is the reverse of itself (if $w = a_1 a_2 \ldots a_n$ then reverse$(w) = a_n a_{n-1} \ldots a_1$). E.g., '**abracadabra**' is (sadly) not a palindrome; Napoleon's '**able was I ere I saw Elba**' is.

We now describe a 2-tape Turing machine $M = (Q, \Sigma, I, q_0, \delta, F)$ that halts and succeeds if w is a palindrome, and halts and fails otherwise. (The actual output $f_M(w)$ of M is unimportant; what matters is whether M halts and succeeds, because this is enough to tell us the answer — whether w is a palindrome or not. Problems like palindrome detection, with only two possible answers for each instance, are called **yes/no problems,** or **decision problems.** We'll see more of these in Part III.)

The idea is very simple. Initially, the word w is on tape 1. M first copies w to tape 2. Then it moves head 1 back to the beginning of the original copy of w on tape 1, and head 2 to the end of the new copy on tape 2. It now executes the following:

```
if the symbols read by the two heads are different then halt and fail
repeat until head 2 reaches square 0
    move head 1 right and move head 2 left
    if the symbols read by the two heads are different then halt and fail
end repeat
```

Note that a 2-*track* machine could not do this, as it has only one head. Note also that we can't take a shortcut by stopping when the heads meet in the middle: a 2-tape Turing machine doesn't know when its heads cross, unless it has arranged to count moves.

Exercises 3.7

1. Draw a flowchart for this machine. The main bit is shown in figure 3.4. (This

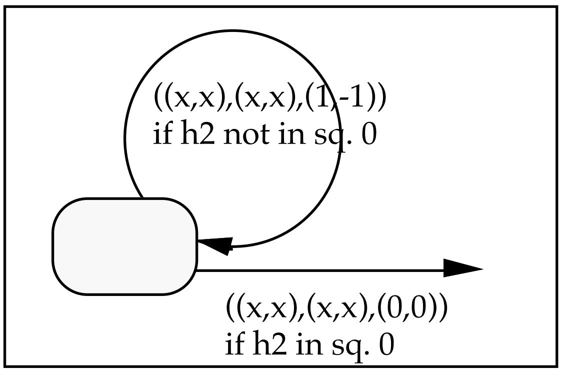


Figure 3.4: main part of palindrome tester

halts & fails if the heads read different characters — there's no applicable instruction.) Now try to design a single-tape Turing machine that does the same job. That should convince you that many-tape machines can help programming considerably. (For a solution see Harel's book, p.202.)

2. Let I be an alphabet. Design a 2-tape Turing machine M such that $f_M(w) = \text{reverse}(w)$ for all $w \in I^*$. Can you design an M such that $f_M(w) = w{*}w$? (The output is w, followed by a $*$, followed by a copy of w. E.g.: $abc{*}abc$)

3.3.3 Many-tape versus 1-tape Turing machines

In the exercise, the 1-tape Turing machine that you came up with probably didn't look much like the original 2-tape machine. If we tried to find 1-tape equivalents for more and more complex Turing machines with more and more tapes, our solutions (if any) would probably look less and less like the original. Can we be *sure* that *any n*-tape Turing machine has a 1-tape equivalent — as it should have by Church's thesis?

In the following important theorem, we *prove* that for any $n \geq 1$, n-tape Turing machines have exactly the same computational power as 1-tape machines. Thus Church's thesis survives, and in fact the theorem provides further evidence for the thesis.

Just as in the two-way-infinite tape case (theorem 3.4), we will design a 1-tape machine that *mimics* or *simulates* the n-tape machine itself, rather than trying to solve the same problem directly, perhaps in a very different way. But now, each *single* step of the n-tape machine will be mimicked by *many steps* of the 1-tape machine. We are really proving that *we can simulate a bounded concurrent system on a sequential machine*, albeit slowly.

Theorem 3.8 *Let n be any whole number, with $n \geq 2$. Then:*

1. *For any ordinary, 1-tape Turing machine M, there is an n-tape Turing machine M_n that is equivalent to M.*

2. *For any n-tape Turing machine M_n, there is an ordinary, 1-tape Turing machine M that is equivalent to M_n.*

PROOF. To show (1) is easy (because expensive is 'obviously better' than cheap). Given an ordinary 1-tape Turing machine M, we can make it into an n-tape Turing machine by adding extra tapes and heads but telling it not to use them. In short, it ignores the extra tapes and goes on ignoring them!

Formally, if $M = (Q, \Sigma, I, q_0, \delta, F)$, we define $M_n = (Q, \Sigma, I, q_0, \delta', F)$ by:

$$\delta' : Q \times \Sigma^n \to Q \times \Sigma^n \times \{-1, 0, 1\}^n$$
$$\delta'(q, a_1, \ldots, a_n) = (q', b_1, \wedge, \wedge, \ldots, \wedge, d_1, 0, 0, \ldots, 0) \text{ where } \delta(q, a_1) = (q', b_1, d_1).$$

(Recall that δ is the only formal difference between TMs with different numbers of tapes.) Clearly, M_n computes the same function as M, so it's equivalent to M.

The converse (2), showing that cheap is really just as good as expensive, is of course harder to prove. For simplicity, we only do it for $n = 2$, but the idea for larger n is the same.

So let M_2 be a 2-tape Turing machine. We will construct a 1-tape Turing machine M that **simulates** M_2. As we said, each M_2-instruction will correspond to an entire subroutine for M. The idea is very simple: M simulates M_2 by *drawing a diagram or* *picture of M_2's initial configuration, and then updating the picture to keep track of the moves M_2 makes.*

M has a single *4-track* tape (cf. §2.4.2). At each stage, track 1 will have the same contents as tape 1 of M_2. Track 2 will show the location of head 1 of M_2, by having an X in the current location of head 1 and a blank in the other squares. Tracks 3 and 4 will do the same for tape 2 and head 2 of M_2.

Example 3.9 Suppose that at some point of execution, the tapes and heads of M_2 are as in figure 3.5. Then the tape of M will currently be looking like figure 3.6.

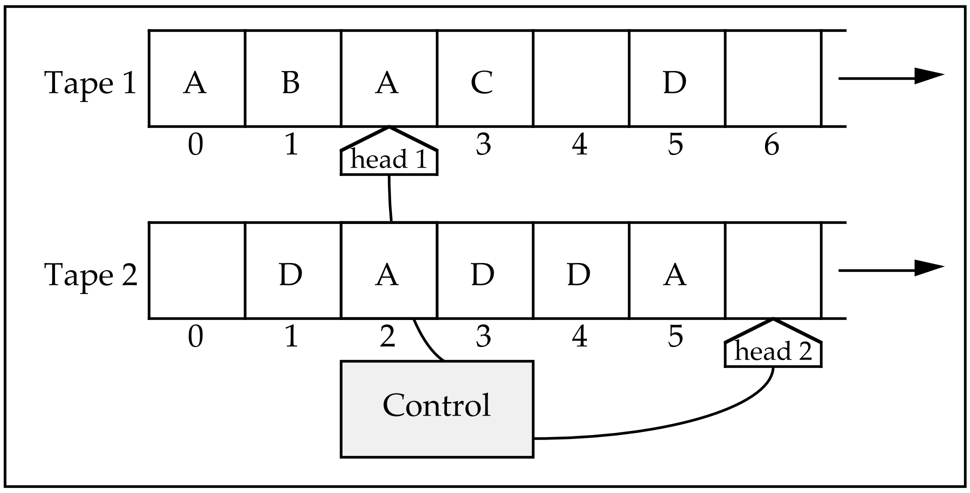


Figure 3.5: M_2, with 2 tapes

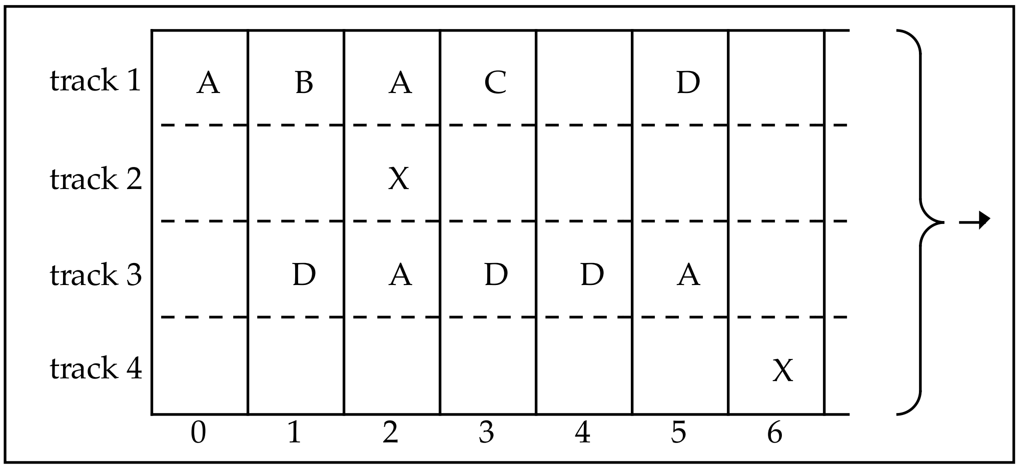


Figure 3.6: a single 4-track tape with the same data

So the tape of M will always show the current layout of M_2's tapes and heads. We have to show how M can update its tape to keep track of M_2. Let us describe M's operation from start to finish, beginning with the setting-up of the tracks.

Initialisation Recall that initially both heads of M_2 are over square 0; tape 1 carries the input of M_2, and tape 2 of M_2 is blank. M is trying to compute the same function as M_2, so we can assume its input is the same. That is, initially M's (single) tape is the same as tape 1 of M_2.

First, M sets up square 0. Suppose that tape 1 of M_2 (and so also M's tape) has the symbol a in square 0. Then M writes $(a,X,\wedge,X)$ in square 0. This is because it knows that both heads of M_2 start in square 0 — so that's where the Xs should be! And it knows tape 2 of M_2 is blank.

But these Xs will move around later, with the heads of M_2, so also, square 0 should be marked. M can mark square 0 with an extra track — cf. §2.4.3. So really M's tape has *five* tracks; but we agreed in §2.4.3.1 not to mention this track, for simplicity.

We'll assume **dynamic track set-up,** as in §2.4.2.4. So whenever M moves into a square whose contents are not of the form (a,b,c,d), but just a, it immediately overwrites the square with $(a,\wedge,\wedge,\wedge)$, and then continues. This is because to begin with, M's tape is the same as tape 1 of M_2, and tape 2 of M_2 is blank. So $(a,\wedge,\wedge,\wedge)$ is the right thing to write.

We assume also that M always knows the current state q of M_2 (initially q_0). It can keep this information in its state set as well, because the state set of M_2 is also finite.

M must now update the tape after each move of M_2, repeating the process until M_2 halts. Suppose that M_2 is about to execute an instruction (i.e., to read from and write to the tapes, move the heads, and change state). When M_2 has done this, its head positions and tape contents may be different. M updates its own tape to reflect this, in two stages:

Stage 1: Finding out what M_2 knows First M's head sweeps from square 0 to the right. As it does so, it will come across the X markers in tracks 2 and 4. When it hits the X in track 2, it looks at the symbol in track 1 of the same square. This is the symbol that head 1 of M_2 is currently scanning. Suppose it is a_1, say. *M remembers* this symbol 'a_1' in its own internal states — cf. §2.4.1. It can do this because there are only finitely many possible symbols that a_1 could be (Σ is finite). Similarly, M will eventually find the X in track 4, and then it also remembers the symbol — a_2, say — in track 3 of the same square. a_2 is the symbol that head 2 of M_2 is currently scanning. Of course, M might find the X in track 4 before or even at the same time as the X in track 2. In any event, once it has found both Xs, *M knows both the current symbols a_1, a_2 that M_2 is scanning.*

Stage 2: Updating the tape We assume that M 'knows' the *instruction table* δ of M_2. This never changes so can be 'hard-coded' in the instruction table of M. As with the 2-way-infinite tape simulation (theorem 3.5), M does not have to *compute* δ — δ is built into M in the sense that the instruction table of M is based on it.

E.g., a pseudo-code representation of M would involve a long case statement, one case for each line of the instruction table of M_2.

M now has enough information to work out what M_2 will do next. For, once M knows a_1, a_2, and q, then since it knows δ, it also knows the value

$$\delta(q, a_1, a_2) = (q', b_1, b_2, d_1, d_2) \in Q \times \Sigma^2 \times \{-1, 0, 1\}^2,$$

if defined. If it is not defined (because M_2 has no applicable instruction), M halts and fails (as M_2 does). Assume it is defined. Then M's head sweeps back leftwards, updating the tape to reflect what M_2 does. That is:

```
Procedure SweepLeft  % pre: head starts off over the rightmost X
set done1, done2 to false
repeat
    if track 2 has X and not done1 then
        set done1 to true
        write $b_1$ in track 1 and $\wedge$ in track 2 % so erasing the X
        move in direction $d_1$                  % follow the move of head 1 of $M_2$
        write X in track 2 (leaving the other tracks alone) % new position of head 1
        move in direction $-d_1$                 % back to where we were
    end if
    ⟨a similar routine for head 2 of $M_2$, using tracks 3 and 4 and variable done2⟩
    move left
until in square 0
```

M ends up in square 0 with the correct tape picturing M_2's new pattern.

If q' is not a halting state for M_2, M now forgets M_2's old state q, remembers the new state q', and begins the next sweep at Stage 1 above.

The output Suppose then that q' is a halting state for M_2. So at this point, M_2 will halt and succeed with the output on tape 1. As track 1 of M's tape always looks the same as tape 1 of M_2, this same output word must now be on track 1 of M's tape. M now demolishes the other three tracks in the usual way, leaving a single track tape containing the contents of the old track 1, up to the first blank.

M has simulated every move of M_2. So for all inputs in I^*, the output of M is the same as that of M_2. Thus $f_M = f_{M_2}$, and M is equivalent to M_2, as required. QED.

Summary We showed that any algorithm implementable by a 2-tape machine M_2 is implementable by a 1-tape machine M.

Exercises 3.10

1. Write out the pseudo-code routine to handle tracks 3 and 4 in SweepLeft.

2. Draw flowcharts of the parts of M that handle stages 1 and 2 above. It's not too complicated if you use parameters to store $a_1, a_2, q, q', b_1, b_2, d_1$, and d_2, as in §2.4.1.

3. Why do we 'move $-d_1$' after writing X in track 2? (Hint: what if the heads are both in square 6?)

4. Why do we need the variable `done1` in SweepLeft? What might happen if we omitted it?

5. What alterations would be needed to simulate an n-tape machine?

6. Suppose that at some point M_2 tries to move one of its heads left from square 0. M_2 halts and fails in this situation. What will M do?

7. Suppose that on some input, M_2 never halts. What will M do?

8. How could we make M more efficient?

9. (Quite long.) Let M_2 be the 'reverser' 2-tape Turing machine of exercise 3.7. Suppose M_2 is given as input a word of length n. How many steps will M_2 take before it halts? If M is the 1-tape machine that simulates M_2, as above, how many steps (roughly!) will it take?

3.3.4 Exam questions on Turing machines

1. (a) Design a Turing machine M with input alphabet $\{a, b, c\}$, which, given as input a word w of this alphabet, outputs the word obtained from w by deleting all occurrences of 'a'. For example, $f_M(bcaba) = bcb$

   You may use pseudo-code or a flow-chart diagram; in the latter case, you should explain your notation for instructions. You may use several tapes, and you can assume that square 0 of each tape is implicitly marked.

   (b) *Briefly* explain how you would design a Turing machine N, with the same input alphabet as M, that moves all occurrences of 'a' in its input word to the front (left), leaving the order of the other characters unchanged. Thus, $f_N(bcaba) = aabcb$

   The two parts carry, respectively, 65% and 35% of the marks.

2. (a) Explain the difference between a 2-*track* and a 2-*tape* Turing machine.

   Below, the notation 1^n *denotes a string 111...1 of n 1's. The symbol* $*$ *is used as a delimiter. You may assume that square 0 of each Turing machine tape is implicitly marked.*

   (b) Design a 2-tape Turing machine M with input alphabet $\{1\}$, such that if the initial contents of tape 1 are 1^n (for some $n \geq 0$) and the initial contents of tape 2 are 1^m (for some $m > 0$), then M halts and succeeds if and only if m divides n without remainder. You may use pseudo-code or a flow-chart diagram; in the latter case you should explain your notation for instructions.

(c) By modifying M or otherwise, briefly explain how you would design a (2-tape) Turing machine M^* with input alphabet $\{1,*\}$, such that for any $n \geq 0$ and $m > 0$, $f_{M^*}(1^n*1^m) = 1^r$, where r is the remainder when dividing n by m.

The three parts carry, respectively, 20%, 45% and 35% of the marks.

3. (a) What is **Church's thesis?** Explain why it cannot be proved but could possibly be disproved. What kinds of evidence for the thesis are there?

(b) Design a 2-tape Turing machine M with input alphabet I, such that if w_1 and w_2 are words of I of equal length, the initial contents of tape 1 are w_1 and the initial contents of tape 2 are w_2, then M halts and succeeds if w_1 is an **anagram** (i.e., a rearrangement of the letters) of w_2, and halts and fails otherwise.

For example, if $w_1 = abca$ and $w_2 = caba$, M halts and succeeds; if $w_1 = abca$ and $w_2 = cabb$, M halts and fails.

You may use pseudo-code or a flow-chart diagram; in the latter case you should explain your notation for instructions. You may assume that square zero of each Turing machine tape is implicitly marked.

The two parts carry, respectively, 40% and 60% of the marks. [1993]

3.4 Other variants

We briefly mention some other kinds of Turing machine, and how they are proved equivalent to the original version.

3.4.1 Two-dimensional tapes

We can have a Turing machine with a 2-dimensional 'tape' with squares labelled by pairs of whole numbers (n,m) (for all $n,m \geq 0$). Reading, writing and state changing are as before, but at each step the head can move left, right, up, or down. So δ is a partial function : $Q \times \Sigma \to Q \times \Sigma \times \{L,R,U,D,0\}$.

When the run starts, the input word is on the 'x-axis' — in squares $(0,0),(1,0),\ldots,$ $(k,0)$ for some $k \geq 0$ — and all other squares of the tape contain $\wedge$. The machine must leave the output on the x-axis as well, but it can use the rest of the plane as work space. Thus it is like a 1-tape machine with an unbounded number of tracks on the tape, except that access to other tracks is not instantaneous, as the head must move there first. As the input word is finite and only 1 symbol is written at each step, at all times there will only be finitely many non-blank symbols on the 2-dimensional tape.

3.4.1.1 Simulating a 2-dimensional Turing machine

We will show how a ‘big’ 2-dimensional machine can be simulated by a ‘little’ 2-tape machine. Then we will know by theorem 3.8 that the big machine can be simulated by a 1-tape machine too.

The contents of the big 2-dimensional tape are kept on little tape 1 in the following format. The data on tape 1 is divided into segments of equal length, separated by a marker, ‘*’. The segments list in order the non-blank rows of the big tape. For example, suppose the non-blank part of the big tape is as in figure 3.7:

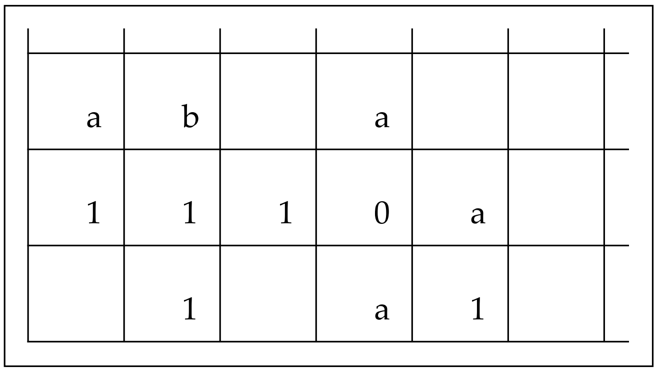


Figure 3.7: 2-dimensional tape

Then tape 1 of the little machine will contain the three segments

$$\wedge 1 \wedge a 1 * 1110 a * ab \wedge a \wedge **$$

— or the same but with longer segments filled out by blanks. Note the double $**$ at the end. Tape 2 of the little machine is used for scratch work.

Head 1 of the little machine is over the symbol corresponding to where the big machine’s head is. If big head moves left or right, so does little head 1. If however, big head moves up, little head 1 must move to the corresponding symbol in the next segment to the right. So the little machine must remember the offset of head 1 within its current segment. This offset is put on tape 2, e.g., in unary notation. So in this case, little head 1 moves left until it sees ‘$*$’. For each move left, little head 2 writes a 1 to tape 2. When ‘$*$’ is hit, head 1 moves right to the next ‘$*$’. Then for each further move of head 1 right, head 2 deletes a 1 from tape 2. When all the 1’s have gone, head 1 is over the correct square, and the next cycle commences.

Sometimes the little machine must add a segment to tape 1 (if big head moves higher on the big tape than ever before), or lengthen each segment (if big head moves further right than before). It is easy to add an extra segment of blanks on the end of tape 1 of the right length — tape 2 is used to count out the length. Adding a blank at the end of each segment can be done by shifting, as in example 2.9. The little machine can do all this, return head 1 to the correct position (how?), and then implement the move of the big head as above — there is now room for it to do so.

This bears out Turing's remark in his pioneering paper that whilst people use paper to calculate, the 2-dimensional character of the paper is never strictly necessary. Again we have found evidence of type (b) for Church's thesis. A similar construction can be used to show that for any $n \geq 1$, n-dimensional Turing machines are equivalent to ordinary ones.

3.4.2 Turing machines with limited alphabet

We can imagine Turing machines with alphabet $\Sigma_0 = \{0, 1, \wedge\}$ and $I = \{0, 1\}$. Unlike the previous variants, these are seemingly *less* powerful (cheaper) than the basic model. But they can compute any function $f_M : I \to I$ for any Turing machine M. The idea is to simulate a given Turing machine $(Q, \Sigma, I, q_0, \delta, F)$ by coding its scratch characters (those of $\Sigma \setminus I$) as strings of 1s. E.g., we list Σ as $\{s_1, \ldots, s_n\}$ and represent s_i by a string 1^i of i 1s. Exercise: work out the details. We will develop this idea considerably in the next section.

3.4.3 Non-deterministic Turing machines

We will define these and show that they're equivalent to ordinary machines in Part III of the course.

3.4.4 Other machines and formalisms

Ordinary Turing machines have the same computational power as **register machines,** and also more abstract systems such as the **lambda calculus** and **partial recursive functions.** No-one has found a formalism that is intuitively algorithmic in nature but has more computational power. This fact provides further evidence for Church's thesis.

3.5 Summary of section

We considered what it means for two different kinds of machine to have the same computational power, deciding that it meant that they could compute the same class of functions. Examples such as palindrome detection showed how useful many-tape TMs can be. We proved or indicated that the ordinary Turing machine has the same computational power as the variants: 2-way infinite tape machines, multi-tape machines, 2-dimensional tape machines, limited character machines, and non-deterministic machines. This provided evidence for Church's thesis.

4. Universal Turing machines

We nowadays accept that a single computer can solve a vast range of problems, ranging from astronomical calculations to graphics and process control. But before computers were invented there were many kinds of problem-solving machines, with quite different 'hardware'. Turing himself helped to design code-breaking equipment with dedicated hardware during the second world war. These machines could do nothing but break codes. Turing machines themselves come in different kinds, with different alphabets, state sets, and even hardware (many tapes, etc).

It was Turing's great insight that this proliferation is unnecessary. In his 1936 paper he described a single general-purpose Turing machine, that can solve all problems that any Turing machine could solve.

This machine is called a **universal Turing machine.** We call it U. *U is not magic — it is an ordinary Turing machine,* with a state set, alphabet, etc, as usual. If we want U to calculate $f_M(w)$ for some arbitrary Turing machine M and input w to M, we give U the input *w plus a description of M.* We can do this because $M = (Q, \Sigma, I, q_0, \delta, F)$ can be described by a finite amount of information. U then evaluates $f_M(w)$ by calculating what M would do, given input w — rather in the way that the 1-tape Turing machine simulated a 2-tape Turing machine in theorem 3.8.

So really, U is *programmable:* it is an *interpreter* for arbitrary Turing machines. In this section, we will show how to build U.

4.1 Standard Turing machines

In fact we have been lying! U will not be able to handle *arbitrary* Turing machines. For example, if M has a bigger input alphabet than U does, then some legitimate input words for M cannot be given to U at all. There's a similar problem with the output.

So when we build U, we will only deal with the restricted case of **standard Turing machines.** This just means that their alphabet is fixed. Though the 'computer alphabet' $\{0, 1\}$ is often used for this purpose, we will use the following, more convenient **standard character set.** In §4.4 we will indicate why using a fixed alphabet is not really a restriction at all.

Definition 4.1 We let C be the alphabet $\{$a,b,c,...,A,B,...,0,1,2,...,!,@,£,...$\}$ of characters that you would find on any typewriter (about 88 in all; note that $\wedge$ is not included in C).

Definition 4.2 A Turing machine S is said to be **standard** if:

1. it conforms *exactly* to definition 2.1, and
2. its input alphabet is C and its full alphabet is $C \cup \{\wedge\}$.

Warning By (1) of this definition, we know that:

- S has a single one-way infinite tape (a multi-tape TM is a **variant** of the TM of definition 2.1).
- the tape of S has only one track
- any marking of square 0 is done explicitly.

Extra tracks and implicit marking of square 0 are implemented by adding symbols to the alphabet (see §2.4.2.1). There is no point in fixing our alphabet as C, and then changing it by adding these extra symbols.

4.2 Codes for standard Turing machines

We need a way of describing a standard Turing machine to U. So we introduce a key notion, that of **coding a Turing machine,** so we can represent it as *data*. We will code each standard Turing machine S by a word $code(S)$ of C, in such a way that the operations of S can be reconstructed from $code(S)$ by an algorithm. So:

- S is a standard Turing machine;
- $code(S)$ will be a *word of* C, representing S.

Then we will design U so that $f_U(code(S) * w) = f_S(w)$ for all standard Turing machines S and all words $w \in C^*$.[1]

4.2.1 Details of the coding

Let $S = (Q, C \cup \{\wedge\}, C, q_0, \delta, F)$ be any standard Turing machine. Let us suppose that $Q = \{0, 1, \ldots, n\}$, $q_0 = 0$, and $F = \{f, f+1, \ldots, n\}$ for some $n \geq 0$ and some $f \leq n$. (There is no loss of generality in making this assumption: see §4.2.3.1 below.)

Much as in §2.1.1, we think of the instruction table δ as a list of 5-tuples, of the form

$$(q, s, q', s', d)$$

where $\delta(q,s) = (q', s', d)$. For each 5-tuple in the list we have:

$$0 \leq q < f, \quad 0 \leq q' \leq n, \quad s, s' \in C \cup \{\wedge\}, \quad d \in \{-1, 0, 1\}.$$

S is then specified completely by this list, together with the numbers n and f.

[1]We will input the pair $(code(S), w)$ to U in the usual way, by giving it the string $code(S)$ concatenated with the string w, with a delimiting character, say $*$, in between.

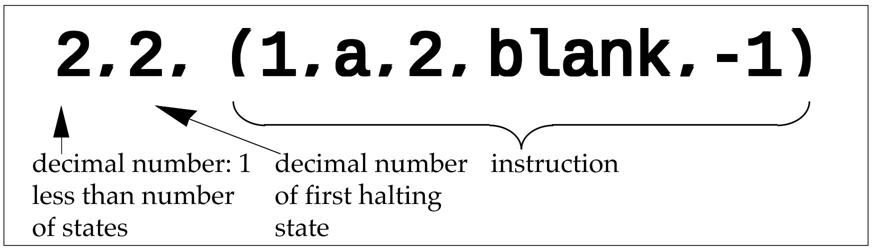
There are many ways of coding this information. We will use a simple one. Consider the *word*

$$n, f, t_1, t_2, \ldots, t_N$$

where the list of 5-tuples is $t_1, t_2, \ldots, t_N$ in some arbitrary order,[2] and all numbers (n, f and the numbers q, q' and d in the 5-tuples) are written in decimal (say). This is a word of our coding alphabet $C \cup \{\wedge\}$. We let $code(S)$ be the word of C obtained from this by replacing every '$\wedge$' by the five-letter word '`blank`'. (As we will be giving $code(S)$ as input to Turing machines, we don't want $\wedge$ to appear in it.)

4.2.2 Checking whether a word codes a TM

If we have a word w of C, we can check by an algorithm if w is the code for a Turing machine. E.g., ';;()101,y%-)' is no good, whilst the code shown in figure 4.1 is OK (3 states, 0, 1, 2; state 2 is halting; if in state 1 and read 'a', go to state 2, write '$\wedge$' and move left).



2,2,(1,a,2,blank,-1)
decimal number: 1 less than number of states
decimal number of first halting state
instruction

Figure 4.1: code(a very simple TM)

In general, w must have the form

$$n, f, (x, y, x, y, d), (x, y, x, y, d), \ldots, (x, y, x, y, d)$$

where n, f, x, and d are decimal numbers with $0 \leq f \leq n$, $0 \leq x \leq n$, and $-1 \leq d \leq 1$, and y is some single character of C, or '`blank`'.

Exercises 4.3

1. There are several other checks (to do with final states, functionality of δ, and more) to be made before we are sure w is a code for a genuine TM. Explain these.

2. In a 5-tuple (x, y, x, y, d), each y could be any of:

   0 1 2 3 4 5 6 7 8 9   ,   –   (   )   `blank`

   Is the code of S really unambiguous?

[2]Unlike in a conventional computer, the order of the instructions t_i is not part of S and clearly does not affect the way S works.

4.2.3 Remarks on the coding

4.2.3.1 Generality of coding

We assumed that $Q = \{0,1,\ldots,n\}$ and $F = \{f, f+1,\ldots,n\}$ for some $f \leq n$. This is not really a restriction, because given any old standard Turing machine $S = (Q, C \cup \{\wedge\}, C, q_0, \delta, F)$, we can always rename its states without changing its behaviour, so long as we then adjust q_0, δ, and F accordingly.

We can therefore rename the states in Q to $0,1,\ldots,n$ (where S has $n+1$ states), so that

- the initial state is 0
- the final states come at the end of the listing — i.e., they are $f, f+1,\ldots,n$ for some $f \leq n$.

Note: if q_0 were also a final state of S, we could not assume that F consists of the states at the *end* of the list. E.g., we might have $F = \{q_0, q_{273}\}$. So our coding would not work. But such a TM would halt immediately, so we can take its code to be '0,0,' or the code of any other Turing machine that halts immediately, as all such machines have the same input-output function (namely the identity). Similarly, if $F = \emptyset$ then the Turing machine never halts and succeeds, so it never outputs anything, and we can take its code to be '1,1,', or the code of any other Turing machine that never halts and succeeds.

4.2.3.2 (Non-)uniqueness of code of S

There are many ways of renaming the states Q of S to $0,1,\ldots,n$. And for a given S, we can list the instruction 5-tuples $t_1,\ldots,t_N$ in many different orders. We get a different word $code(S)$ representing the same S, for each possible renaming and ordering.

So $code(S)$ is really a **relation,** because it is not uniquely defined. We don't mind this. Below, where $code(S)$ comes up, it will stand for *any* code of S. We don't care which actual code is used; any order of instructions or states will do. There is still an algorithm to tell whether any given w is or is not the code of some Turing machine S (with the instructions listed in *some* order). If we really wanted a *function* from standard Turing machines to words, we could let $code(S)$ be the *first* word of C (in alphabetical order) that codes S.

Exercise 4.4 Could we design a TM to decide whether words w_1, w_2 of C code the *same* TM?

4.2.3.3 Standard is needed

We could not code a non-standard TM without more definitions. First, an arbitrary M may have alphabet $\Sigma \neq C$ — or it may have alphabet C but uses many tracks, which comes to the same thing. As $code(M)$ must be a word of C, not of Σ, we'd need to represent each symbol in Σ by a symbol or word of C. If M had alphabet C but used more than one tape, we'd have to change the instruction format: e.g., the instruction table of a 3-tape machine is a list of 11-tuples!

4.2.3.4 Other codings

Our coding has some *redundancy*. E.g., we don't really need the brackets '(' and ')', or the number n at the front (why not?). There are also other, rather different codings available. For example, Rayward-Smith's book gives one using prime factorisation, in which the code of S is always a number, usually called the **Gödel number** of S.

Exercise 4.5 How could we turn our $code(S)$ into a number?

We stress that these are only details. U only needs to be able to recover the workings of S from $code(S)$. We can use any coding that allows this.

4.2.4 Summary

The point behind these details is that each standard Turing machine S can be represented by a finite piece of information, and hence can be coded by a word $code(S)$ of C, in such a way that *we can reconstruct S from $code(S)$*. The word $code(S) \in C^*$ carries all the information about S. It is really a *name* or *plan* of S.

4.3 The universal Turing machine

Now we can build the universal machine U. It has the following specification:

- If the input to U is $code(S) * w$,[3] where S is a standard Turing machine and $w \in C^*$, then U will output $f_S(w)$. (If $f_S(w)$ is undefined, then so is the output of U.)
- If the input is not of this form, we don't care what U does.

That is,

$$f_U(code(S) * w) = f_S(w)$$

for all standard Turing machines S and all $w \in C^*$.

The input alphabet of U will be C, but U will not be standard, as it will have 3 tapes with square 0 (implicitly) marked.[4]

How does U work? Suppose that

$$S = (\{0, 1, \ldots, n\}, C \cup \{\wedge\}, C, 0, \delta, \{f, f+1, \ldots, n\})$$

for some $f \leq n$. Assume the input to U is $code(S) * w$. U will *simulate the run of S on input w*. We will ensure that at each stage during the simulation:

- tape 1 keeps its original contents $code(S) * w$, for reference;

[3]Recall that $code(S)$ is not unique. But any code for S carries all the information about S. In fact, it will be clear that U will output $f_S(w)$ given input $s * w$, where s is *any* code for S.

[4]If we wish, we can use theorem 3.8 to find a one-tape equivalent of U, and then, as the output of U will be a word of C (why?), apply theorem 4.7 below to find a standard TM equivalent to U.

- tape 2 is always the same as the current tape of S;
- head 2 of U is always in the same position as the head of S;
- tape 3 holds the current state of S, in the same decimal format as in the instructions on tape 1. So e.g., if S is in state 56, the contents of tape 3 of U are '5' in square 0 and '6' in square 1, the rest being blank.

Step 1: Initialisation: U begins by writing 0 in square 0 of tape 3. The rest of tape 3 is already blank, so it now represents the initial state 0 of S. U then copies w from tape 1 to tape 2. (The word w is whatever is after the pair of characters ')*' or a string of the form '$n, f, *$' on tape 1, so U can find it.) It then returns all three of its heads to square 0. The three tapes (and head 2) are now set up as above.

Step 2: Simulation: For each execution step of S, U does several things.

1. Maybe the current state q of S is a halting state. To find out, U first compares the number q on tape 3 with the number f in $code(S)$. U can find what f is by looking just after the first ',' on tape 1. They are in the same decimal format, so U can use a simple string comparison to check whether $q < f$ or $q \geq f$.
2. If $q \geq f$, this means that S is now in a halting state. Because tape 2 of U is *always* the same as the tape of S, the output of S is now on tape 2 of U. U now copies tape 2 to tape 1, terminated by a blank, and halts & succeeds.
3. If $q < f$ then q is not a halting state, and S is about to execute its next instruction. So head 1 of U scans through the list of instructions (the rest of $code(S)$, still on tape 1) until it finds a 5-tuple of the form (q, s, q', s', d) where:
   - q (as above) is S's current state as held on tape 3. Head 3 repeatedly moves along in parallel with head 1, to check this.
   - s is the symbol that head 2 is now scanning — i.e., S's current symbol. A direct comparison of the symbols read by heads 1 and 2 will check this. (If s is '`blank`', U tests whether head 2 is reading $\wedge$.)
4. If no such tuple is found on tape 1, this means that S has *no applicable instruction,* and will halt and fail. Hence U halts and fails too (e.g., by moving heads left until they run off the tape).
5. So assume that U has found on tape 1 the part '$(q, s$' of the instruction (q, s, q', s', d) that S is about to execute. S will write s', move its head by d, and change state to q'. To match this, U needs to know what s', q', and d are. It finds out by looking further along the instruction 5-tuple it just found on tape 1, using the delimiter ',' to keep track of where it is in the 5-tuple.[5] Head 2 of U can now write s' at its current location (by just

[5]The awful possibility that s and/or s' is the delimiter ',' can be got round by careful counting.

copying it from tape 1, except that if it is `blank`, head 2 writes $\wedge$), and then move by d (d is also got from tape 1). Finally, U copies the decimal number q' from tape 1 to tape 3, replacing tape 3's current contents. After returning head 1 to square 0, U is ready for the next step of the run of S. It now repeats Step 2 again.

Thus, every move of S is simulated by U. Clearly, U halts and succeeds if and only if S does, and in that case, the output of U is just $f_S(w)$. Hence, $f_U(code(S) * w) = f_S(w)$, and U is the universal machine we wanted.

Exercises 4.6

1. What does U do if S tries at some step to move its head left from square 0 of its tape?

2. (Important) Why do we not hold the state of S in the state of U (cf. storing a finite amount of information in the states, as in §2.4.1 and theorems 3.5 and 3.8)? After all, the state set of S is finite!

3. By using theorem 3.8, and then theorem 4.7 below, we can replace U with an equivalent standard TM. So we can assume that U is standard, so that $code(U)$ exists and is a word of C.

   Let S be a standard TM, and let $w \in C^*$. What is $f_U(code(U) * code(S) * w)$?

Using an interpreter was a key step in our original paradox, and so we are now well on the way to rediscovering it in the TM setting. In fact we will not use U to do this, but will give a direct argument. Nonetheless, U is an ingenious and fascinating construction — and historically it led to the modern programmable computer.

4.4 Coding

A Turing machine can have any finite alphabet, but the machine U built above can only 'interpret' standard Turing machines, with alphabet C. This is not a serious restriction. Computers use only 0 and 1 internally, yet they can work with English text, Chinese, graphics, sound, etc. They do this by **coding.**

Coding is not the secret art of spies — that is called **cryptography. Coding** means turning information into a different (e.g., condensed) format, but in such a way that nothing is lost, so that we can **decode** it to recover the original form. (Cryptography seeks codings having decodings that are hard to find without knowing them.) Examples of codings are ASCII, Braille, hashing and some compression techniques, Morse code, etc., (think of some more). A computer stores graphics in coded (e.g., bit-mapped) form.

Here, we will indicate briefly how to use coding to get round the restriction that U can only 'do' standard machines.

4.4.1 Using the alphabet C for coding

Just as ASCII codes English text into words of $\{0,1\}$, so the characters and words of any finite alphabet Σ can be coded as words of C.

1. C has about 88 characters, so we choose a whole number k such that:

   $$(\text{number of words of } C \text{ of length } k) = 88^k \geq \text{ size of } \Sigma.$$

2. We can then assign to each $a \in \Sigma$ a unique word of C of length k. We write this word as $code(a)$. There are enough words of C of length k to ensure that no two different symbols of Σ get the same code. (Formally we choose a 1–1 function $code : \Sigma \to C^k$; exactly what function we choose is not important).

3. We can now code any *word* $w = a_1a_2 \ldots a_n$ of Σ, by concatenating the codes of the letters of w:

   $$\begin{aligned} code(\varepsilon) &= \varepsilon \\ code(a_1a_2\ldots a_n) &= code(a_1).code(a_2). \cdots .code(a_n) \in C^* \end{aligned}$$

   This is just as in ASCII, Morse, etc. We see that $code(a_1a_2\ldots a_n)$ is a word of C of length kn.

4. We also need to **decode** the codes. There is a unique partial function $decode : C^* \to \Sigma^*$ given by:

   - $decode(code(w)) = w$ for all $w \in \Sigma^*$,
   - $decode(v)$ is undefined if v is a word of C^* that is not of the form $code(w)$.

   For any finite Σ, we can choose k and a function *code* as above, and define *decode* accordingly.

As an example of the same idea, we can code words of C itself as words of $\{0,1\}$, using ASCII. We have e.g., $code(\langle space \rangle) = 01000000$, $code(AB) = 0100000101000010$, and $decode(01000100) = D$.

4.4.2 Scratch characters

Coding helps us in two ways. First, a Turing machine will often need to use **scratch characters:** characters that are used only in the machine's calculations, and do not appear in its input or output. Examples are characters like $(a_1,\ldots,a_n)$ used in multi-track work (§2.4.2); we also use scratch characters for marking square 0 (§2.4.3). We now show that in fact, *scratch characters are never strictly needed.* We do this only for standard Turing machines, but the idea in general is just the same.

Theorem 4.7 (elimination of scratch characters) *Let $M = (Q,\Sigma,C,q_0,\delta,F)$ be a Turing machine with input alphabet C and any full alphabet Σ. Suppose that $f_M : C^* \to C^*$ — i.e., the output of M is always a word of C. Then there is a standard Turing machine S that is equivalent to M.*

PROOF. (sketch; cf. Rayward-Smith, Theorem 2.6) The idea is to get S to mimic M by working with codes throughout. Choose an encoding function $code : \Sigma \to C^*$. The input word w is a word of C. But as Σ is contained in C, w is also a word of Σ, so w itself can be coded! S begins by encoding w itself, to obtain a (longer) word $code(w)$ of C. S can then simulate the action of M, working with codes of characters all along. Whatever M does with the symbols of Σ, M^* does with their codes.

If the simulation halts, S can decode the information on the tape to obtain the required output. The decoding only has to cope with codes of characters in $C \cup \{\wedge\}$, as we are told that the output consists only of characters in C. Because S simulates all operations of M, we have $f_S = f_M$, so S and M are equivalent. At no stage does S need to use any other characters than $\wedge$ or those in C. So S can be taken to be standard. QED.

We can now use U to interpret any Turing machine M with input alphabet C and such that $f_M : C^* \to C^*$. We first apply the theorem to obtain an equivalent standard Turing machine S, and then pass $code(S)$ to U.

4.4.3 Replacing a Turing machine by a standard one

But what if the *input alphabet* of M is bigger than C? Maybe

$$\alpha\beta\gamma\otimes\partial \to \int \perp\{p \vee q \to \neg r\}$$

is a possible input word for M; we are *not allowed* to pass this to U, as it's not a word of U's input alphabet. But M is presumably executing some algorithm, so we'd like U to have a crack at simulating M.

Well, coding can help here, too. Just as computers can do English (or Chinese) word processing with their limited 0-1 alphabet, so we can design a new Turing machine M^* that parallels the action of M, but working with codes of the characters that M actually uses. We'll describe briefly how to do this; it's like eliminating scratch characters.

Assume M has full alphabet Σ. Σ could be very large, but it is finite (because the definition of Turing machine only allows finite alphabets). Choose a coding function $code : \Sigma \to C^*$. Where M is given input $w \in C^*$, we'll give $code(w)$ to M^*. From then on, M^* will work with the codes of characters from Σ just as in theorem 4.7 above. M will halt and succeed on input w if and only if M^* halts and succeeds on input $code(w)$. The output of M^* in this case will be $code(f_M(w))$, the code of M's output, and this carries the same information as the actual output $f_M(w)$ of M.

4.5 Summary of section

We have built a universal Turing machine U. U can simulate any standard Turing machine S (i.e., one with input alphabet C and full alphabet $C \cup \{\wedge\}$), yielding the same result as S on the same input. We only have to give U the additional information $code(S)$ — i.e., the program of S. So U serves as an interpreter for TMs.

To this end we explained how to specify a standard TM as a coded word. Standard TMs use the standard alphabet C. We showed how standard machines are in effect as good as any kind of TM, by coding words of an arbitrary alphabet as words of C and having a standard Turing machine work directly on the codes. We used this idea to eliminate the need for scratch characters.

5. Unsolvable problems

5.1 Introduction

In this section, we will show that some problems, although not vague in any way, are inherently *unsolvable* by a Turing machine. Church's thesis then applies, and we conclude that there is no algorithm to solve the problem.

5.1.1 Why is this of interest?

- Many of these problems are not artificial, 'cooked-up' examples, but fundamental questions such as 'will my program halt?' whose solutions would be of great practical use.
- Increasingly, advanced computer systems employ techniques (such as theorem provers and Prolog) based on **logic.** As logic is an area rich in unsolvable problems, it is important for workers in these application areas to be aware of them.
- The methods for proving unsolvability can be exploited further, in complexity theory (Part III). This is an active area of current research, also very relevant to advanced systems.
- It shows the fundamental limitations of computer science. If we accept Church's thesis, these problems will never be solved, whatever advances in hardware or software are made.
- Even if hardware advances, etc., cause Church's thesis to be updated in the fullness of time, the unsolvable problems are probably not going to go away. Their unsolvability arises not because the algorithms we have are not powerful enough, but because they are too powerful! We saw in section 1 how *paradoxes* cause unsolvability. Paradoxes usually arise because of **self-reference,** and algorithms are powerful enough to allow self-reference. (We saw in section 4 that a Turing machine can be coded as data, and so given as input to another Turing machine such as U. Compilers take programs as input — they can even compile

themselves!) As any amendment to Church's thesis would probably mean that algorithms are even more powerful than was previously thought, the unsolvable problems would likely remain in some form, and even proliferate.

5.1.2 Proof methods

Our first (algorithmically) unsolvable problems are problems about Turing machines themselves (and so — by Church's thesis — about algorithms themselves). Their unsolvability is proved by *assuming* that some Turing machine solves the problem, and then obtaining a **contradiction** (e.g., $0 = 1$, black = white, etc). *A contradiction is impossible!* Such an impossibility shows that our assumption was wrong, since all other steps in the argument are (hopefully) OK. So there's no Turing machine that solves the problem, after all.

We can then use the method of **reduction** to show that further problems are also unsolvable. The old, unsolvable problem is **reduced** to the new one, by showing that any (Turing machine) solution to the new problem would yield a Turing machine solution to the old. As the old problem is known to be unsolvable, this is impossible; so the new problem has no Turing machine solution either.

A sophisticated example of reduction is used in the proof of Gödel's first incompleteness theorem (§5.4).

5.2 The halting problem

This is the most famous example of an unsolvable problem. The halting problem (or 'HP') is the problem of *whether a given Turing machine will halt on a given input.* For the same reasons as in section 4, we will restrict attention to standard Turing machines (we saw in §4.3 that this is not really a restriction!) In this setting, the halting problem asks, given the input

- $code(S)$, for a standard Turing machine S;
- a word w of C (see definition 4.1 for the alphabet C),

whether or not S halts and succeeds when given input w.

Question Why can we not just use U of section 4 to do this, by getting U to simulate S running on w, and seeing whether it halts or not?

5.2.1 The halting problem formally

Formally, let $h : C^* \to C^*$ be the partial function given by

- $h(x) = 1$ if $x = code(S) * w$ for some standard Turing machine S, and S halts and succeeds on input w

- $h(x) = 0$ if $x = code(S) * w$ for some standard Turing machine S, and S does not halt and succeed on input w
- $h(x)$ is **arbitrary** (e.g., **undefined**) if x is not of the form $code(S) * w$ for any standard Turing machine S and word $w \in C^*$.

Big question: is this function h Turing-computable? Is there a Turing machine H such that $f_H = h$? Such an H would **solve the halting problem.**

Warning Our choice of values 1, 0 for h is *not important.* Any two different words of C would do. What matters is that, on input $code(S) * w$, H always halts & succeeds, and we can tell from its output whether or not S would halt & succeed on input w.

The halting problem is not a toy problem. Such an H would be very useful. As we now demonstrate, regrettably there is no such H. This fact has serious repercussions.

Theorem 5.1 (Turing, 1936) *The halting problem is unsolvable.*

This means that there is no Turing machine H such that $f_H = h$. Informally, it means that there's no Turing machine that will decide, for arbitrary S and w, whether S halts & succeeds on input w or not.

PROOF. Assume for contradiction that the partial function h (as above) is Turing computable. Clearly, if h is computable it is trivial to compute the partial function $g : C^* \to C^*$ given by:

$$g(w) = \begin{cases} 1, & \text{if } h(w * w) = 0, \\ \text{undefined}, & \text{otherwise} \end{cases}$$

(Here, '$w * w$' is just w followed by a '*', followed by w.) So let M be a Turing machine with $f_M = g$. By theorem 4.7 (scratch character elimination) we can assume that M is standard, so it has a code, namely $code(M)$.

There are two cases, according to whether $g(code(M))$ is defined or not.

Case 1: $g(code(M))$ **is defined.** Then $g(code(M)) = 1$ [by def. of g],

so $h(code(M) * code(M)) = 0$ [also by def. of g],

so M does not halt & succeed on input $code(M)$ [by def. of h],

so $f_M(code(M))$ is undefined [by def. of Turing machines],

so $g(code(M))$ is undefined [because $f_M = g$].

This contradicts the case assumption (which was '$g(code(M))$ is defined'). So we can't be in case 1.

Case 2: $g(code(M))$ **is not defined.** Then $f_M(code(M))$ is undefined [because $f_M = g$],

so M does not halt & succeed on input $code(M)$ [by def. of TMs],

so $h(code(M) * code(M)) = 0$ [by def. of h], so $g(code(M)) = 1$ [by def. of g],

so $g(code(M))$ is defined! This contradicts the case assumption, too, so we cannot be in case 2 either.

But clearly either $g(code(M))$ is defined, or it isn't. So we must be in one of the two cases. This is a contradiction. So h is not Turing computable. QED.

Another way of seeing the proof: Suppose for contradiction that H is a Turing machine that solves HP. We don't know how H operates. We only know that $f_H = h$. Consider the simple modification M of H shown in figure 5.1. If the input to M is w,

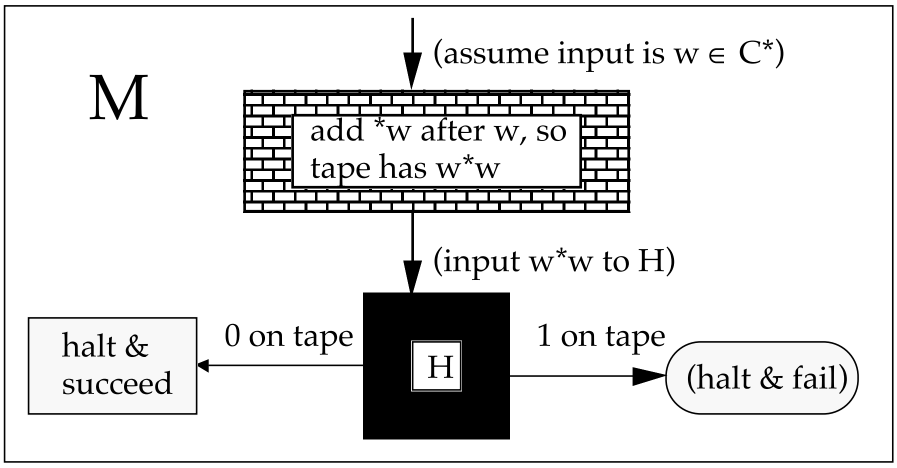


Figure 5.1: an impossible TM

then M adds a $*$ after w, then adds a copy of w after it, leaving '$w * w$' on the tape. It then returns to square 0, calls H as a subroutine, and halts & succeeds/fails according to the output of H, as in the figure. Note that these extra operations (copying, etc.,) are easy to do with a TM. So if H exists, so does M.

Clearly M outputs only 0, if anything. So $f_M : C^* \to C^*$, and by theorem 4.7 (elimination of scratch characters) we can assume that M is standard. So M has a code, viz. $code(M)$.

Consider the run of M when its input is $code(M)$. M will send input '$code(M) * code(M)$' to H. Now as we assumed H solves HP, the output of H on input $code(M) * code(M)$ says whether M halts and succeeds when given input $code(M)$.

But we are now in the middle of this very run — of M on input $code(M)$*!* H is saying whether the *current run* will halt & succeed or not! The run hasn't finished yet, but H is supposed to predict how it will end — in success or failure. This is clearly a difficult task for H! In fact, M is designed to find out what H predicts, *and then do the exact opposite!* Let us continue and see what happens.

The input to H was $code(M) * code(M)$, and $code(M)$ is the code for a standard Turing machine (M itself). So H will definitely halt and succeed. Suppose H outputs 1 (saying that M halts and succeeds on input $code(M)$). M now moves to a state with no applicable instruction (look at figure 5.1). M has now halted and failed on input $code(M)$, so H was wrong.

Alternatively, H decides that M halts and fails on input $code(M)$. So H outputs 0. In this case, M gleefully halts and succeeds: again, H was wrong.

But H was assumed to be correct *for all inputs.* This is a contradiction. So H does not exist. QED.

5.2.2 The halting problem is hard

Warning: do not think that HP is an easy problem. It is not (and in general, no algorithmically unsolvable problems are easy). I've heard the following argument:

1. We proved that there's no Turing machine that solves the halting problem.
2. So by Church's thesis, the halting problem is unsolvable by an algorithm.
3. But our brains are algorithmic — just complicated computers running a complex algorithm.
4. We can solve the halting problem, as we can tell whether a program will halt or not. So there is an algorithm to solve the halting problem — us!

(2) and (4) are in conflict. So what's going on?

Firstly, many people would not agree with (3). See Penrose's book, listed on page 6. But in any case, I don't believe (4). Consider the following pseudo-code program:

```
$n, p$: integer. stp: Boolean       % '$n$ is the sum of two primes'
set $n$ to 4
set stp to true
repeat while stp
    set stp to false
    repeat with $p = 2$ to $n-2$
        if prime($p$) and prime($n-p$) then set stp to true
    end repeat
    add 2 to $n$
end repeat

function prime($p$)       % assume $p \geq 2$
    $i, p$: integer
    repeat with $i = 2$ to $p-1$
        if $i$ divides $p$ without remainder then return false
    end repeat
    return true
end prime
```

The function prime returns true if the argument is a prime number, and false otherwise. The main program halts if some even number > 2 is not the sum of two primes. Otherwise it runs forever. As far as I know, no-one knows whether it halts or not. See Goldbach's conjecture (§5.4). (And of course we could design a Turing machine doing the same job, and no-one would know whether it halts or not.)

Exercises 5.2

1. Write a program that halts iff Fermat's last theorem is false. (This theorem was only proved in around 1995, after 300 years of effort. So telling if your program halts can be quite hard!)

2. What happens if we rewire the Turing machine M of figure 5.1, swapping the 0 and 1, so that M halts and succeeds if H outputs 1, and halts and fails if H's output is 0? What if we omit the duplicator that adds '$*w$' after w? [Try the resulting machines on some sample inputs.]

3. Show that there is no Turing machine X such that for all standard Turing machines S and words w of C, $f_X(code(S)*w) = 1$ if S halts *(successfully or not)* on input w, and 0 otherwise.

4. Let the function $f : C^* \to C^*$ be given by: $f(w) = a.f_M(w)$ if $w = code(M)$ for some standard Turing machine M and $f_M(w)$ is defined, and a otherwise (here, $a \in C$ is just the letter a!). Prove that f is not Turing computable.

5. (similar to part of exam question, 1991) Let X be a Turing machine such that $f_X(w) = w*w$ for all $w \in C^*$. Let Y be a *hypothetical* Turing machine such that for every standard Turing machine S and word w of C,

$$f_Y(code(S)*w) = \begin{cases} 1 & \text{if } f_S(w) = 0, \\ 0 & \text{otherwise} \end{cases}$$

   So Y tells us whether or not S outputs 0 on input w.

   (a) How might we build a standard Turing machine M such that for all $w \in C^*$, we have $f_M(w) = f_Y(f_X(w))$?

   (b) By evaluating $f_M(code(M))$, or otherwise, deduce that Y does not exist.

6. Prove that HP is unsolvable by using the 'Java-style' diagonal paradox of section 1. [Use the universal machine of section 4.]

7. A **super Turing machine** is like an ordinary TM except that I and Σ are allowed to be infinite. Find a super TM that solves HP for ordinary TMs. [Hint: take the alphabet to be C^*.] Deduce that super TMs can 'compute' non-algorithmic functions. What if instead we let Q be infinite?

5.3 Reduction

So the halting problem, HP, is not solvable by a Turing machine. There is no machine H as above. (By Church's thesis, HP has no algorithmic solution.) We can use this fact to show that a range of other problems have no solution by Turing machines.

The method is to reduce HP to a special case of the new problem. The idea is very simple. We just show that *in order to solve HP (by a Turing machine), it is enough to solve the new problem.* We could say that the task of solving HP **reduces** to the task of solving the new problem, or that HP 'is' a **special case** of the new problem. So if the new problem had a TM solution, so would HP, contradicting theorem 5.1. This means that the new problem doesn't have a TM solution.

5.3.1 Reduction and unsolvability

In general, we say that a problem A **reduces** to another problem, B, if we can convert any Turing machine solution to B into a Turing machine solution to A.[1]

Thus, if we knew somehow that A had no Turing machine solution (as we do for A = HP), we could deduce that B had no Turing machine solution either.

Example 5.3 Multiplication reduces to **addition**,[2] because we could easily modify an addition algorithm to do multiplication. So if we knew that multiplication could not be done by an algorithm, we couldn't hope to find an algorithm that does addition. Another example: we reduced Goldbach's conjecture to HP in §5.2.2 above.

Warning A reduces to B = you can use B to solve A. Please get it the right way round!

Warning It is vital to realise that we can reduce A to B *whether or not A and B are solvable.* The point is that *if* we were *given* a solution to B, we could use it to solve A, so that A is 'no harder' than B. (These free, magic solutions to problems like B are called **oracles.**) Not all unsolvable problems reduce to each other.[3] Some unsolvable problems are more unsolvable than others! We'll see more of this idea in Part III.

5.3.2 The Turing machine $M[w]$

Reduction is useful in showing that problems are unsolvable. We will see some examples in a moment. The following Turing machine will be useful in doing them.

[1]This is a bit vague, but it will do for now. (The main problem is what 'convert' means.) We will treat reduction more formally in section 11, but I can tell you in advance that we will be saying something like this: A reduces to B if there exists a Turing machine M that converts inputs to A into inputs to B in such a way that if M_B is a Turing machine solving B, then the Turing machine given by 'first run M, then run M_B' solves A. This is what happens in most of the examples below.

[2]In this sense, addition is *harder* than multiplication, because if you can add you can easily multiply, but not vice versa.

[3]E.g.: the problem 'does the standard TM S return infinitely often to its initial state when run on w?' is unsolvable but does not reduce to HP.

Suppose M is any Turing machine, and w a word of its input alphabet, I. We write $M[w]$ for the new Turing machine[4] that does the following:

1. First, it overwrites its input with w;
2. then it returns to square 0;
3. then it runs M.

It's easy to see that there always is a TM doing this, whatever M and w are. Here's an example.

Example 5.4 Suppose *TAIL* is a Turing machine such that $f_{TAIL}(w) = \mathsf{tail}(w)$ (for all $w \in C^*$). So *TAIL* deletes the first character of w, and shifts the rest one square to the left. See exercise 2.11(3) on page 33.

The machine *TAIL*[hello_world] first writes 'hello_world' on the tape, overwriting whatever was there already. Then it returns to square 0 and calls *TAIL* as a subroutine. This means that its output will be the rather coarse 'ello_world' *on any input.* The input is immediately overwritten, so it doesn't matter what it is.

Figure 5.2 shows the Turing machine *TAIL*[hello] as a flowchart.

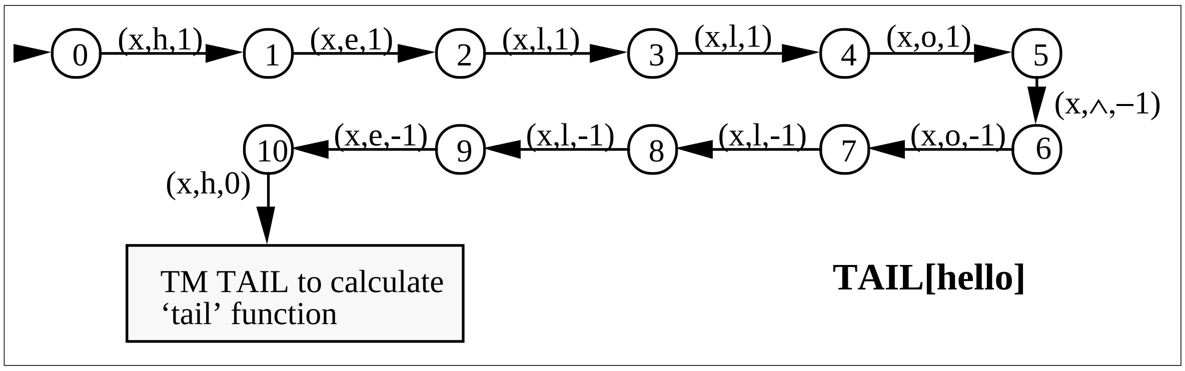


Figure 5.2: the TM *TAIL*$[hello]$

5.3.2.1 Important properties of $M[w]$

Because the input to $M[w]$ doesn't matter, it is clear that for any Turing machine M and any word w of I, we have:

1. $f_{M[w]}(v) = f_M(w)$ for any word v of I.
2. M halts and succeeds on input w, iff $M[w]$ halts and succeeds on input v for any or all words v of I.

[4]A less succinct notation for $M[w]$ would be 'w, then M'.

5.3.2.2 Making $M[w]$ from M and w

Given M and w, it's very easy to make $M[w]$. The part that writes w is always like the 'hello_world' machine — it has w hard-wired in, with a state for each character of w. Note that it adds a blank after w, to kill long inputs (see states 5–6 in figure 5.2). Returning to square 0 is easy; and then M is called as a subroutine. In figure 5.2, $M = \mathit{TAIL}$, $w = \mathit{hello}$.

5.3.2.3 Making $M[w]$ standard

Moreover, if M is standard, then $M[w]$ can be made standard, too.[5] (See §4.1.)

- The part of $M[w]$ that writes w is certainly standard.
- If we do the return to square 0 by implicitly marking square 0, we will NOT get a standard Turing machine. So we don't. Instead, we do a **hard-wired return to square 0.** Notice how $\mathit{TAIL}[\mathit{hello}]$ in figure 5.2 returns to square 0: by writing 'olleh' backwards! In general, $M[w]$ writes w forwards, and then returns to square 0 by writing w again, backwards, but with the head moving left.
- As M is known to be standard, there's no problem with that part of $M[w]$.

5.3.3 The Turing machine *EDIT*

It's not only easy for us to get $M[w]$ from M and w; we can even design a Turing machine to do it! There is a Turing machine that takes as input $code(S) * w$, for any standard Turing machine S and word w of C, and outputs $code(S[w])$. We call this machine *EDIT* — it edits codes of TMs.

How does *EDIT* work? It adds instructions (i.e., 5-tuples) on to the end of $code(S)$. The new instructions are for writing w and returning to square 0. They will involve some number N of new states: a new initial state, and a state for writing each character of w, both ways. So $N = 1 + 2 \cdot length(w)$. (For example, in figure 5.2 we added new states $0, 1, \ldots, 10$, so $N = 11$, i.e., one more than twice the number of characters in 'hello'.) The states of the M-part of $M[w]$ were numbered $0, 1, 2, \ldots$ in $code(S)$. Now they will be numbered $N, N+1, N+2, \ldots$. So *EDIT* must also increase all state numbers in the old $code(S)$ by N.

This sounds complicated, but it is really very simple. *Rough* pseudo-code for *EDIT* is as follows. Remember, *EDIT*'s input is $code(S) * w$, and its output should be $code(S[w])$.

```
% The states of $S[w]$ are those of $S$ plus $1 + 2 \cdot length(w)$ new ones, numbered
% $0, 1, \ldots, 2 \cdot length(w)$. So first, renumber the states mentioned in $code(S)$
% (currently $0, 1, \ldots$) as $1 + 2 \cdot length(w), 2 + 2 \cdot length(w), \ldots$.
Add $1 + 2 \cdot length(w)$ to all state numbers in $code(S)$, including $n$ and $f$ at the front
% $S[w]$ overwrites its input with '$w\wedge$' so add instructions for this at the end of the code.
```

[5]To show this, we could just use scratch character elimination. But it would make the machine *EDIT* (below) more complicated. So we try to make $M[w]$ standard in the first place.

```
% In figure 5.2 we'd get $(0,a,1,h,1),(1,a,2,e,1),(2,a,3,l,1),\ldots$, for all $a$ in $C\cup\{\texttt{blank}\}$.
s := 0      % s will be current state number ($s=0,1,\ldots,2\cdot length(w)$).
repeat with $q=1$ to $length(w)$
    for each $a\in C\cup\{\texttt{blank}\}$, add an instruction 5-tuple '$(s,a,s+1,\langle q$th char of $w\rangle,1)$'
    $s:=s+1$
end repeat
for each $a\in C\cup\{\texttt{blank}\}$, add an instruction 5-tuple '$(s,a,s+1,\texttt{blank},-1)$'
$s:=s+1$
% $S[w]$ returns to square 0 and hands over to $S$.
% Add instructions for this, in the way we said.
repeat with $q=length(w)$ down to 2
    for each $a\in C\cup\{\texttt{blank}\}$, add an instruction 5-tuple '$(s,a,s+1,\langle q$th char of $w\rangle,-1)$'
    s:= s+1
end repeat
for each $a\in C\cup\{\texttt{blank}\}$, add an instruction 5-tuple '$(s,a,s+1,\langle$1st char of $w\rangle,0)$'
halt & succeed
```

Of course, this does not show all the details of how to transform the input word $code(S)*w$ into the output word $code(S[w])$.

Exercises 5.5

1. Write out the code of the head-calculating machine M shown in figure 2.5 (p. 28), assuming to keep it short that the alphabet is only $\{a,b,\wedge\}$. Then write out the code of $M[ab]$. Do a flowchart for it. Does it have the number of states that I claimed above? What is its output on inputs (i) *bab*, (ii) $\wedge$? (Don't just guess; run your machine and see what it outputs!)

2. Would you implement the variable q in the pseudocode for *EDIT* using a parameter in one of *EDIT*'s states, or by an extra tape? Why?

3. Write proper pseudocode (or even a flowchart!) for *EDIT*.

5.3.4 The empty-input halting problem, EIHP

This is the problem of whether or not a Turing machine halts and succeeds on the empty input, ε. As before, we only consider standard Turing machines.

You might think EIHP looks easier than HP, as it only asks about the halting of S on a single fixed input. Well done — you noticed that EIHP reduces to HP! Easier it may be, but EIHP is still so hard as to be **unsolvable** — it has no algorithmic solution. We show this by reducing HP to EIHP.

To say that EIHP is **solvable** is to say that there is a Turing machine EI such that for any standard Turing machine S,

$$f_{EI}(code(S)) = \begin{cases} 1 & \text{if } S \text{ halts and succeeds on input } \varepsilon \\ 0 & \text{otherwise} \end{cases}$$

Theorem 5.6 *EIHP is unsolvable — there is no such Turing machine EI.*

PROOF. *We will prove this by showing that HP reduces to EIHP.* So assume we're given a Turing machine EI that solves EIHP. We convert it into a Turing machine H, as shown in figure 5.3. H first runs *EDIT* (see §5.3.3), then returns to square 0, then

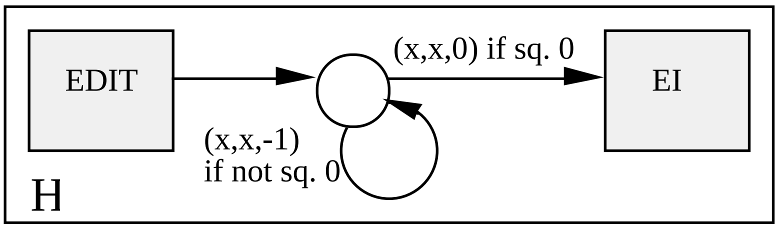


Figure 5.3: EIHP solution gives HP solution

calls EI as a subroutine. We showed how to make *EDIT*, we're given EI, and the rest is easy. So we can really make H as above.

We claim that H solves HP. Let's feed into H an input $code(S) * w$ of HP. H runs *EDIT*, which converts $code(S) * w$ into $code(S[w])$. This word $code(S[w])$ is then fed into EI, which (we are told) outputs 1 if $S[w]$ halts and succeeds on input ε, and 0 otherwise.

But (cf. §5.3.2.1, with $v = \varepsilon$) $S[w]$ halts and succeeds on input ε iff S halts and succeeds on w.

So the output of H is 1 if S halts and succeeds on w, and 0 otherwise. Thus, H solves HP, as claimed, and *we have reduced HP to EIHP.*

So by the argument at the beginning of §5.3, EIHP has no Turing machine solution. EI does not exist. QED.

Exercises 5.7

1. Reduce EIHP to HP (easy).

2. The **uniform halting problem,** UHP, is the problem of whether a (standard) Turing machine halts & succeeds on every possible input. Show by reduction of HP that UHP is unsolvable. [Hint: use the machine of figure 5.3.]

3. The **sometimes-halts problem,** SHP, is the problem of whether a (standard) Turing machine halts & succeeds on at least one input. Show by reduction of HP that SHP is unsolvable.

4. [Challenge!] Show that the problem of deciding whether or not two arbitrary standard Turing machines S_1, S_2 are **equivalent** (definition 3.1) is not solvable by a Turing machine.

5.3.5 Real-life example of reduction

As we can simulate Turing machines on ordinary computers (if enough memory is available), and vice versa, it follows that the halting problem for Turing machines reduces to the HP for Java. For if we had a Java program to tell whether an *arbitrary* Java program halts, we can apply it to the TM simulator, so Java could solve HP for TMs. But (cf. Church's thesis) the Java halting program could be implemented on a TM, so we'd have a TM that could solve the HP for Turing machines, contradicting theorem 5.1.

So there is no Java program that will take as input an arbitrary Java program P and arbitrary input x, and tell whether P will halt on input x. For a particular P and x you may be able to tell, but there is no general strategy (algorithm) that will work. The paradox of section 1 can also be used to show this. Thus it is better to write well-structured programs that can easily be seen to halt as required!

5.3.6 Sample exam questions on HP etc.

1. (a) Explain what the halting problem is. What does it mean to say that the halting problem is unsolvable?

   (b) Explain what the technique of reduction is, and how it can be used to show that a problem is unsolvable.

   (c) Let C be the standard typewriter alphabet. The symbol * is used as a delimiter. Let the partial function $f : C^* \to C$ be given by

$$f(code(S) * w) = \begin{cases} 1 & \text{if } S \text{ halts and succeeds on input } w \\ & \quad \text{and its output contains the symbol 0,} \\ 0 & \text{otherwise,} \end{cases}$$

   for any standard Turing machine S and word w of C.

   Prove, either directly or by reduction of the halting problem, that there is no Turing machine M such that $f_M = f$.

2. (a) Explain what the empty-input halting problem is. What does it mean to say that the empty-input halting problem is unsolvable?

   (b) The empty-output problem asks, given a standard Turing machine M and input word w to M, whether the output $f_M(w)$ of M on w is defined and is empty (ε).

   Let EO be a hypothetical Turing machine solving the empty-output problem:

$$f_{EO}(code(M) * w) = \begin{cases} 1 & \text{if } f_M(w) = \varepsilon, \\ 0 & \text{otherwise,} \end{cases}$$

   for any standard Turing machine M and word w of the typewriter alphabet C.

Let DUPLICATE be a Turing machine such that $f_{DUPLICATE}(w) = w * w$ for any word w of C.

By considering the Turing machine X partially described by figure 5.4, prove that EO cannot exist. (You must decide how to fill in the shaded parts.)

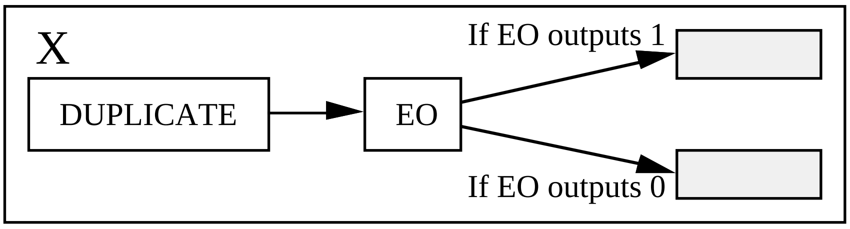


Figure 5.4: the TM X

(c) Can there exist a Turing machine EO' such that

$$f_{EO'}(code(M) * w) = \begin{cases} \varepsilon, & \text{if } f_M(w) = \varepsilon, \\ 1 & \text{otherwise} \end{cases}$$

for all standard Turing machines M and words w of C? Justify your answer.

The three parts carry, respectively, 30%, 50%, 20% of the marks.

3. In this question, C denotes the standard typewriter alphabet.

(a) What does it mean to say that a partial function $g : C^* \to C^*$ is (Turing-)computable?

(b) Let $g : C^* \to C^*$ be a partial function that 'tells us whether the output of a standard Turing machine on a given input is "hello" or not'. That is, for any standard Turing machine S and word w of C,

$$g(code(S) * w) = \begin{cases} y & \text{if } f_S(w) = hello \\ n & \text{otherwise.} \end{cases}$$

Show that g is not computable.

(c) Let U be the universal Turing machine. Let $h : C^* \to C^*$ be a partial function such that for any word x of C,

$$h(x) = \begin{cases} y & \text{if } f_U(x) = hello, \\ n & \text{otherwise.} \end{cases}$$

Given that g as in part b is not computable, deduce that h is not computable either.

The three parts carry, respectively, 25%, 50% and 25% of the marks.

4. In this question:

- C denotes the standard typewriter alphabet.
- If S is a standard Turing machine and w a word of C, $S[w]$ is a standard Turing machine that overwrites its input with w and then runs S.
  So $f_S[w](x) = f_S(w)$ for any word x of C.
- *EDIT* is a standard Turing machine such that for any standard Turing machine S and word w of C, $f_{EDIT}(code(S) * w) = code(S[w])$.
- *REV* is a standard Turing machine that reverses its input (so, for example, $f_{REV}(abc) = cba$).
- U is a (standard) universal Turing machine.

(a) Define what it means to say that a partial function $g : C^* \to C^*$ is (Turing-) computable.

(b) Let $g : C^* \to C^*$ be a partial function that "tells us whether or not a standard Turing machine halts and succeeds *on input w*w*". That is, for any standard Turing machine S and word w of C,

$$g(code(S) * w) = \begin{cases} y & \text{if } S \text{ halts \& succeeds on input } w * w, \\ n & \text{otherwise.} \end{cases}$$

Show that g is not computable.

(c) Evaluate:

i. $f_U(code(REV) * deal)$
ii. $f_U(f_{EDIT}(code(REV) * stock) * share)$
iii. $f_U(f_{EDIT}(code(U) * code(REV) * buy) * sell)$

The three parts carry, respectively, 20%, 40% and 40% of the marks. [1994]

5.4 Gödel's incompleteness theorem

We sketch a final unsolvability result, first proved by the great Austrian logician Kurt Gödel in 1931. One form of the theorem states that *there is no Turing machine that prints out all true statements ('sentences') about arithmetic and no false ones.*[6] Arithmetic is the study of the whole numbers $\{0, 1, 2, \ldots\}$ with addition and multiplication. A 'Gödel machine' is a hypothetical Turing machine that prints out all the true sentences (and no false ones) of the **language of arithmetic.** This is a first-order language with:

- The function symbols '$+$' and '$\cdot$' (plus and times), and the successor function S
- The relation symbol $<$ (and $=$ of course)
- The constant symbol 0

[6] Such a machine would never halt. It would print out successive truths, separated on the tape by $*$, say. We just have to keep looking at the output every so often.

- The variables $x, x', x'', x''', \ldots$ (infinitely many)
- The connectives $\wedge$ (and), $\vee$ (or), $\rightarrow$ (implies), $\neg$ (not), $\leftrightarrow$ (iff)
- The quantifiers $\forall, \exists$
- The brackets (and)

There are 19 symbols here. Think of them as forming an alphabet, Σ. Note that x' is two symbols, x and $'$. (We'll cheat and use $y, z, \ldots$ as abbreviations for variables $x', x'', \ldots$) The successor function S represents addition of 1. So $SSS0$ has value 3.

This is an *extremely* powerful language. E.g., x **is prime** is expressible by the formula

$$\pi(x) =_{def} (x > S0) \wedge (\forall y \forall z (x = y \cdot z \rightarrow y = x \vee z = x)).$$

Goldbach's conjecture is expressible by the sentence

$$GC =_{def} \forall x (x > SS0 \wedge \exists y (x = y \cdot SS0) \rightarrow \exists y \exists z (\pi(y) \wedge \pi(z) \wedge x = y + z)).$$

Exercise 5.8 What does GC say? Is $\forall y \forall z (SS0 \cdot y \cdot y = z \cdot z \rightarrow y = 0)$ true?

Whether Goldbach's conjecture is true is still unknown, 200 years or so after Goldbach asked it. So if we had a 'Gödel machine', we could wait and see whether GC or $\neg GC$ was in its output. Thus it would solve Goldbach's conjecture (eventually).

Regrettably:

Theorem 5.9 *There is no Gödel machine.*[7]

PROOF. (sketch) We reduce HP to the problem a Gödel machine would be solving, if such a machine existed. In fact we show that for any standard Turing machine M and word w of the alphabet C, the statement 'M halts on input w' can be algorithmically written as a sentence X of arithmetic! (This shows some of the power of this language.) We could then solve the halting problem by waiting to see which of X or $\neg X$ occurs in the output of a Gödel machine G. But we proved that HP has no Turing machine solution, so this is a contradiction. Hence there is no such G.

We will only be able to *sketch* the argument for obtaining X from $code(S) * w$ — in full, the proof is quite long. The idea, though, is very simple:

- A **configuration** of M is the combination (current state, tape contents, head position).
- A run of M can be modelled by a sequence of configurations of M. Given any configuration in a run, we can work out the next configuration in a well-defined way using the instruction table δ of M.
- We can **code** any configuration as a number. The relationship between successive configurations in a run of M then becomes an arithmetical relation between their codes. This relation is expressible in the language of arithmetic. (Cf. Turing's statement quoted on page 20.)

[7]Gödel's first incompleteness theorem. Equivalently (for mathematicians), there is no recursive axiomatisation of true arithmetic.

- We can write a formula coding entire sequences of numbers (configurations), of any length, as a single number. Thus the entire run of M can be represented as a single number, c. We can write a formula $R(c)$ expressing that the list of configurations coded by c forms a successful (halting) run of M on input w.

- We then write a sentence $X = \exists x R(x)$ of arithmetic saying that there exists ($\exists$) a run of M on w that ends in a halting state. So X is true if and only if M halts on w, as required.

QED.

The same idea comes up again in section 12 (Cook's theorem).

Gödel's theorem can also be proved using our old paradox **the least number not definable by a short English sentence:** see Boolos' paper in the reading list. For assume that there was such a Gödel machine: G, say. Because of G's mechanical Turing-machine nature, it turns out that properties of G are themselves statements about arithmetic. Crudely, the statement 'this statement is not in the output of G' can be written as a sentence S of arithmetic. This leads to a contradiction, since S is in the output of G iff S is false. Of course, Gödel's proof didn't use Turing machines — they hadn't been invented in 1931.

5.4.1 Details of the proof

The following couple of pages of details are for interest only; they're not likely to be examined!! There are eleven easy steps.

1. As in §4.2.1, let M have state set $Q = \{0, 1, 2, \ldots, q\}$, where 0 is the initial state and the halting states are $f, f+1, \ldots, q$.

2. Let's begin by **coding a configuration of M as a sequence of numbers.** We can code the state by itself, and the head position by the number of the tape square that the head is reading. And as the alphabet of M is C, we can code the symbols on the tape by their ASCII equivalents, using 0 for $\wedge$ (say). (Any ASCII code is a number: e.g., the ASCII code for 'A' is ASCII(A) = 01000001 in binary, i.e., 64+1 = 65. ASCII(B) = 66, etc.)

So the configuration 'in state k, with head over square r, the tape contents up to the last non-$\wedge$ being a_0 (square 0), a_1 (square 1), ..., a_m (square m)' can be represented by the sequence of numbers:

$$(k, r, \text{ASCII}(a_0), \text{ASCII}(a_1), \ldots, \text{ASCII}(a_m))$$

For example, if squares 5, 6, ... are blank, the configuration shown in figure 5.5 can be represented by the sequence (6,3,65,66,65,0,51), the ASCII codes for A, B, 3 being 65, 66, 51 respectively.

3. *Useful Technical Fact:* There is a formula $SEQ(x,y,z)$ of the language of arithmetic with the following useful property. Given any sequence $(a_0, a_1, \ldots, a_n)$ of numbers, there is a number c such that for all numbers z:

- $SEQ(c, 0, z)$ is true if and only if $z = a_0$
- $SEQ(c, 1, z)$ is true if and only if $z = a_1$

  $\ldots\ldots \ldots \ldots \ldots \ldots$

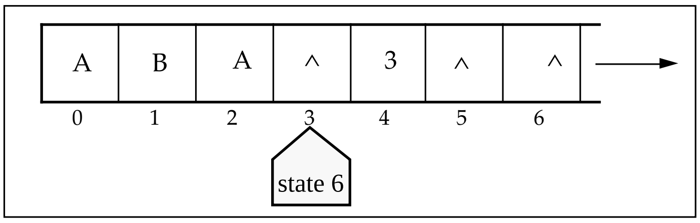


Figure 5.5: a Turing machine configuration

- $SEQ(c,n,z)$ is true if and only if $z = a_n$.

We can use such a formula *SEQ* to code the sequence $(a_0, a_1, \ldots, a_n)$ by the single number c. We can recover $(a_0, a_1, \ldots, a_n)$ from c using *SEQ*.

Finding such a formula *SEQ* is not easy, but it can be done. For example, we might try to code $(a_0, a_1, \ldots, a_n)$ by the number $c = 2^{a_0+1} \cdot 3^{a_1+1} \cdot \ldots \cdot p_n^{a_n+1}$, where the first $n+1$ primes are $p_0 = 2, p_1 = 3, \ldots, p_n$.[8] E.g., the sequence (2,0,3) would be coded by $2^{2+1} \cdot 3^{0+1} \cdot 5^{3+1} =$ 15,000. Because any whole number factors uniquely into primes, we can recover $a_0 + 1, \ldots, a_n +$ 1, and hence $(a_0, a_1, \ldots, a_n)$ itself, from the number c. So it is enough if we can write a formula $SEQ(x,y,z)$ saying 'the highest power of the yth prime that divides x is $z+1$'. *In fact we can write such a formula,* but there are simpler ones available (and usually the simple versions are used in proving that there is a formula *SEQ* like this!)

4. In fact we want to **code a configuration of M as a single number.** Using *SEQ*, we can code the configuration $(k, r, \text{ASCII}(a_0), \text{ASCII}(a_1), \ldots, \text{ASCII}(a_m))$ as a single number, c. If we do this, then $SEQ(c,0,k)$, $SEQ(c,1,r)$, $SEQ(c,2,\text{ASCII}(a_0))$, ..., $SEQ(c,m+2,\text{ASCII}(a_m))$ are true, and in each case the number in the third slot is the *only* one that makes the formula true.

5. *Relationship between successive configurations.* Suppose that M is in a configuration coded (as in (4)) by the number c. If M executes an instruction successfully, without halting & failing, it will move to a new configuration, coded by c', say. What is the arithmetical relationship between c and c'?

Let c code $(k, r, \text{ASCII}(a_0), \text{ASCII}(a_1), \ldots, \text{ASCII}(a_m))$. Assume that $r \leq m$.[9] So M is in state k, and its head is reading the symbol $a_r = a$, say. But we know the instruction table δ of M. Assume that $\delta(k,a) = (k', b, d)$, where k' is the new state, b is the symbol written, and d the move. So if c' codes the next configuration $(k', r', \text{ASCII}(a'_0), \text{ASCII}(a'_1), \ldots, \text{ASCII}(a'_m))$ of M, we know that (i) $r' = r + d$, and (ii) $a'_i = a_i$ unless $i = r$, in which case $a'_r = b$. So in this case, the arithmetical relationship between c and c' is expressible by the following formula

[8]We use $a_1 + 1$, etc., because a_1 may be 0; if we just used a_1 then 0 wouldn't show up as a power of 2 dividing c. So we'd have $code(2,0,3) = code(2,0,3,0,0,0)$ — not a good idea.

[9]If $r > m$, M is reading $\wedge$. This is a special case which we omit for simplicity. Think about what we need to do to allow for it.

$F(c,c',k,a,k',b,d)$:[10]

$$\begin{aligned}
&\forall r\Big([SEQ(c,0,k) \wedge SEQ(c,1,r) \wedge SEQ(c,r+2,\text{ASCII}(a)]\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{\% state } k\text{, head in sq. } r \text{ reads } a\\
&\quad \to [SEQ(c',0,k') \wedge SEQ(c',1,r+d) \wedge SEQ(c',r+2,\text{ASCII}(b))\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{\% new state, head pos \& char}\\
&\quad \wedge \forall i(i \geq 2 \wedge i \neq r+2 \to \forall x(SEQ(c,i,x) \leftrightarrow SEQ(c',i,x))]\Big)\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{\%rest of tape is unchanged}
\end{aligned}$$

Note that we obtain the values k', b, and d from k and a, via δ.

To express — for arbitrary codes c, c' of configurations — that c' codes the next configuration after c, we need one statement $F(c,c',k,a,k',b,d)$ like this for each line (k,a,k',b,d) of δ. Let $N(c,c')$ be the conjunction ('and') of all these Fs. N is the formula we want, because $N(c,c')$ *is true if and only if, whenever M is in the configuration coded by c then its next configuration (if it has one) will be the one coded by c'.*

6. *Coding a successful run.* A successful (halting) run of M is a certain finite sequence of configurations. We can code each of these configurations as a number c, so obtaining a sequence of codes, $c_0, c_1, \ldots, c_n$. For these to be the codes of a successful (halting) run of M on w, we require:

- c_0 *codes the starting configuration of M.* So we want $SEQ(c_0,0,0)$ [state is 0 initially], and $SEQ(c_0,1,0)$ [head over square 0 initially], and also some formulas expressing that the tape initially contains the input word $w = w_0w_1 \ldots w_m$. We can use the formulas $SEQ(c_0,2,\text{ASCII}(w_0))$, $SEQ(c_0,3,\text{ASCII}(w_1))$, …, and $SEQ(c_0,m+2,\text{ASCII}(w_m))$. We can write all this as a finite conjunction $I(c_0)$ (I for 'initial').

- c_{i+1} *is always the 'next' configuration of M after* c_i. We can write this as '$N(c_i,c_{i+1})$ holds for each $i < n$'.

- c_n *codes a halting configuration of M.* Recalling that $f, f+1, \ldots, q$ are the halting states of M, we can write this as a finite disjunction ('or'),

$$H(c_n) = SEQ(c_n,0,f) \vee SEQ(c_n,0,f+1) \vee \ldots \vee SEQ(c_n,0,q),$$

  saying that c_n is a configuration in which M is in a halting state.

7. *Coding a successful run as a single number.* If we now use the formula SEQ to code the entire $(n+2)$-sequence $(n,c_0,\ldots,c_n)$ as a single number, g, say, we can express the constraints in (6) as properties of g:

- $\forall x(SEQ(g,1,x) \to I(x))$
- $\forall n\forall i\forall x\forall y(SEQ(g,0,n) \wedge 1 \leq i < n+1 \wedge SEQ(g,i,x) \wedge SEQ(g,i+1,y) \to N(x,y))$
- $\forall n\forall x(SEQ(g,0,n) \wedge SEQ(g,n+1,x) \to H(x))$

[10] We've cheated and used 1,2,3 in this formula, rather than $S0$, $SS0$, and $SSS0$. We will continue to cheat like this, to simplify things. Also, the formula only works if $d \geq 0$, as there's no '$-$' in the language of arithmetic so if $d = -1$ we can't write $r+d$. If $d = -1$, we replace $SEQ(c,2,r)$ in line 1 by $SEQ(c,2,r+1)$ and $SEQ(c',2,r+d)$ in line 2 by $SEQ(c',2,r)$.

(Note that $c_0, \ldots, c_n$ are entries $1, 2, \ldots, n+1$ of g.)

8. Let the conjunction ($\wedge$) of these three formulas in (7) be $R(g)$. So for any number g, $R(g)$ holds just when g codes a successful run of M on input w. *So the statement 'M halts on input w' is equivalent to the truth of* $\exists x R(x)$.

9. *R can be constructed algorithmically.* Notice that what I've just described is an algorithm (implementable by a Turing machine) to construct $R(x)$, given the data: (a) how many states M has, (b) which states are halting states, (c) the instruction table δ of M, and (d) the input word w. This information is exactly what $code(M) * w$ contains! So there is an algorithm (or Turing machine) that constructs $\exists x R(x)$ from $code(M) * w$.

10. *Reducing HP to the Gödel machine.* If we had a Gödel machine, G, we could now solve the halting problem by an algorithm as follows.

1. Given $code(M)$ and w, where M is a standard Turing machine and w a word of C, we construct the sentence $\exists x R(x)$.

2. Then we wait and see whether $\exists x R(x)$ or $\neg\exists x R(x)$ turns up in the output of G. This tells us which of them is true. (One of them will turn up, because one of them is true, and G prints all and only true statements. So we won't have to wait forever.)

3. If $\exists x R(x)$ turns up, then it's true, so by (8) M must halt & succeed on input w. So we print 'halts' in this case — and we'd be right! We print 'doesn't halt' if $\exists x R(x)$ turns up; similarly, we'd be right in this case too.

So these algorithmic steps would solve the halting problem by a Turing machine.

11. *Conclusion.* But we know the halting problem can't be solved by a Turing machine. This is a contradiction. So G does not exist (because this is the only assumption we made).

5.4.2 Other unsolvable problems

- Deciding whether a sentence of first-order predicate logic is valid or not. Church showed that any algorithm to do this could be modified to print out (roughly) all true statements of arithmetic and no false ones. We've shown this is impossible.

- Post's correspondence problem. This has the stamp of a real problem about it — it doesn't mention algorithms or Turing machines. Given words $a_1, \ldots, a_n$, $b_1, \ldots, b_n$ of C, the question is: is there a non-empty sequence $i(1), \ldots, i(k)$ of numbers $\leq n$ such that the words $a_{i(1)}a_{i(2)}\cdots a_{i(k)}$ and $b_{i(1)}b_{i(2)}\cdots b_{i(k)}$ are the same? There is no algorithm to decide, in the general case, whether there is or not. This can be shown by reducing HP to this problem; see Rayward-Smith for details.

Exercises 5.10

1. Show that there is no algorithm to decide whether a sentence A of arithmetic is true or false. [Hint: any Turing machine to do this could be modified to give a Gödel machine.]

2. Show that there is no Turing machine N such that for all sentences A of arithmetic, $f_N(A) = 1$ if A is true, and undefined if A is false.

3. Complete the missing details in the proof of Gödel's theorem. Find out (from libraries) how to write the formula *SEQ*.

5.5 Summary of section

We saw the significance of unsolvable problems for computing. We proved that there is no Turing machine that says whether an arbitrary Turing machine will halt on a given input ('HP is unsolvable'). By showing that a solution to certain other problems would give a solution to HP (the technique of reduction), we concluded that they were also unsolvable. In this way, we proved that EIHP is unsolvable, and that there is no Turing machine that prints out exactly the true sentences of arithmetic. So doing this is another unsolvable problem.

5.6 Part I in a nutshell

Sections 1–2: A Turing machine (TM) is a 6-tuple $M = (Q, \Sigma, I, q_0, \delta, F)$ where Q is a finite set (of states), Σ is a finite set of symbols (the full alphabet), $I \neq \emptyset$ is the input alphabet, $\Sigma \supseteq I$, $\wedge \in \Sigma \setminus I$ is the blank symbol, $q_0 \in Q$ is the initial state, $\delta : Q \times \Sigma \to Q \times \Sigma \times \{0, 1, -1\}$ is a partial function (the instruction table), and F (a subset of Q) is the set of final states. We view M as having a 1-way infinite tape with squares numbered $0, 1, 2, \ldots$ from left to right. In each square is written a symbol from Σ; all but finitely many squares contain $\wedge$. Initially the tape contains a word w of I (a finite sequence of symbols from I), followed by blanks. w is the input to M. M has a read/write head, initially over square 0. At each point, M is in some state in Q, initially q_0. At each stage, if the head is over a square containing $a \in \Sigma$, and M is in state $q \in Q$, then if $q \in F$, M halts and succeeds. Otherwise, let $\delta(q, a) = (q', b, d)$; M writes b to the square, goes to state q', and the head moves left, right, or not at all, according as $d = -1, 1$, or 0, respectively. If $\delta(q, a)$ is undefined or if the move would take M's head off the tape, M halts and fails. The output of M is the final tape contents, terminated by the first $\wedge$; the output is only defined if M halts & succeeds. We write $f_M : I^* \to \Sigma^*$ for the partial function taking the input word to the output word; here, Σ^* is the set of words of Σ, and similarly for I. A partial function $f : I^* \to \Sigma^*$ is said to be Turing computable if it is of the form f_M for some Turing machine M.

Church's thesis (or better, the Church–Turing thesis) says that all algorithmically computable functions are Turing computable. As algorithm is a vague, intuitive concept, this can't be proved. But there is evidence for it, and it is generally accepted. The evidence has 3 forms:

1. A wide class of functions are Turing-computable. No known algorithm cannot be implemented by a TM.
2. The definition of computability provided by the TM is equivalent to all other definitions so far suggested.

3. Intuitively, any algorithm ought to be implementable by a TM.

Various tricks for simplifying TM design have been suggested. They help the user to design TMs, without changing the definition of a TM. We can divide the tape into finitely many tracks without changing the definition of a TM, since for n tracks, if a_i is in track i of a square (for each $i \leq n$) the tuple $(a_1, \ldots, a_n) \in \Sigma^n$ can be viewed as a single symbol occupying the square. As Σ^n is finite, it can be the alphabet of a legitimate TM. Using many tracks simplifies comparing and marking characters and square 0. Often we mark square 0 implicitly. Track copying operations, etc., are easy to do.

Similarly we can structure the states in Q. This amounts to augmenting Q by a set of the form $Q \times X$ for some non-empty X. Typically X will be Σ or Σ^n for some n. This allows M's behaviour to be more easily specified: when in a state $(q, a) \in Q \times \Sigma$, the behaviour depending on q can be specified separately from that depending on a. Since $Q \times \Sigma$ is a finite set, it can be the state set of a legitimate TM.

We often use 'flowcharts' to specify TMs. A pseudo-code representation is also possible but care is needed to ensure that the code can easily be transformed into a real TM.

Section 3: Variants of TMs have been suggested, making for easier TM design. In particular, the multi-tape TM is extremely useful: there are available n tapes, each with its own head, and the instruction table δ now has the form $\delta : Q \times \Sigma^n \to Q \times \Sigma^n \times \{0, 1, -1\}^n$, interpreted in the natural way. The input and output are by convention on tape 1. An n-tape machine M_n can be simulated by an ordinary Turing machine M. M divides its single tape into $2n$ tracks. For each i in the range $1 \leq i \leq n$, track $2i - 1$ contains the contents of tape i of M_n, and track $2i$ contains a marker denoting the position of head i of M_n over its tape. So M's tape contains a picture of the tapes and heads of M_n. For each step of M_n's run, M updates its tape (picture) accordingly, keeping the specified format. As M simulates every step of the computation of M_n, it can duplicate the input-output function of M_n.

In a similar way it an be shown that a 2-way infinite tape machine, a machine with a 2-dimensional tape, etc, can all be simulated by an ordinary TM. Thus they all turn out to be equivalent in computational power to the original TM. This provides evidence for Church's thesis.

Section 4: A 'universal' Turing machine U can be designed, which can simulate any ordinary Turing machine M. Clearly, for this to be possible we must restrict the alphabets Σ and I of M to a standard form; but essentially no loss of generality results, since we can code the symbols and words of any finite alphabet by words of a fixed standard alphabet. Given a word w and a description of a Turing machine S with standard alphabet, U will output $f_S(w)$. The description of S is also a word '$code(S)$' in the standard alphabet. U works by using the description '$code(S)$' to simulate the action of S on w.

Section 5: One advantage of formalising ‘computable’ is that theorems about ‘computable’ can be proved. We could not hope to show that some problems (functions) were not algorithmically solvable (computable) without a formal definition. In fact many problems are not solvable by a TM. The halting problem (HP), **will** M **halt on input** w**,** is not. This is shown by contradiction: assuming that the Turing machine H solves HP, we construct a Turing machine M that on a certain input (viz. $code(M)$) halts iff H says M will not halt on this input. This is impossible, so H does not exist.

Once a problem is known to be unsolvable, other problems can also be shown unsolvable, by reducing them to the first one. Or one can proceed from first principles in each case. One such is EIHP, the **empty-input halting problem.** Another is the famous Gödel theorem, that there is no algorithm to print all true statements of arithmetic. We proved this by reducing HP to the problem.

Part II

Algorithms

6. Use of algorithms

We now take a rest from Turing machines, and examine the use of algorithms in general practice. There are many: quicksort, mergesort, treesort, selection and insertion sort, etc., are just *some* of the *sorting* algorithms in common use, and for most problems there is more than one algorithm. How to choose the right algorithm? There is no 'best' search algorithm (say): it depends on the application, the implementation, the environment, the frequency of use, etc, etc. Comparing algorithms is a subtle issue with many sides.

6.1 Run time function of an algorithm

Consider some algorithm's implementation, in Java, say. It takes inputs of various sizes. For an input of size n, we want to assign a measure $f(n)$ of its use of resources, and, usually, minimise it. Usually $f(n)$ is the **time** taken by the algorithm on inputs of size n, in the worst or the average case (at our choice).

Calculating $f(n)$ exactly can be problematic:

- The time taken to execute an individual Java instruction can vary even on a single machine, if the environment is shared. So the run time of even the same program for the same data on the same machine may vary.
- There can be a wide variation of use of resources over all the inputs of a fixed size n. The worst case may be rare in practice, the average case unrepresentative.
- Some algorithms may run better on certain kinds of input. E.g., some text string searching algorithms prefer strings of English text (whose patterns can be utilised) to binary strings (which are essentially random).
- Often we do not understand the algorithm well enough to work out f.

So how to proceed? Much is known about some algorithms, and you can look up information. (We list some books at the end of the section; Sedgewick's is a good place to begin.) Other algorithms are still mysterious. Maybe you designed your own algorithm or improved someone else's, or you have a new implementation on a new system. Then you have to analyse it yourself.

Often it's not worth the effort to do a detailed analysis: rough rules of thumb are good enough. Suggestions:

- Identify the **abstract operations** used by the algorithm (read, if-then, +, etc). To maximise machine-independence, base the analysis on the abstract operations, rather than on individual Java instructions.
- Most of the (abstract) instructions in the algorithm will be unimportant resource-wise. Some may only be used once, in initialisation. Generally the algorithm will have an 'inner loop'. Instructions in the inner loop will be executed by far the most often, and so will be the most important. Where is the inner loop? A 'profiling' compilation option to count the different instructions that get executed can help to find it.
- By counting instructions, find a good upper bound for the worst case run time, and if possible an average case figure. It will usually not be worth finding an exact value here.
- Repeatedly refine the analysis until you are happy. Improvements to the code may be suggested by the analysis: see later.

6.1.1 Typical time functions

You can usually obtain the run-time function of your algorithm by a **recurrence relation** (see below). Most often, you will get a run time function

$$f(n) = c \cdot g(n) + \text{smaller terms},$$

where n is the input size and:

- c is a constant (with $c > 0$);
- the 'smaller terms' are significant only for small n or sophisticated algorithms;
- $g(n)$ is one of the following functions:
  1. constant (algorithm is said to run in **constant time**);
  2. $\log n$ (algorithm runs in **log time**);
  3. n (algorithm runs in **linear time**);
  4. $n \log n$ (algorithm runs in **log linear time**);
  5. n^2 (algorithm runs in **quadratic time**);
  6. n^3 (algorithm runs in **cubic time**);
  7. 2^n (or k^n for some $k > 1$) (algorithm runs in **exponential time**).

These functions are listed in order of growth: so n^2 grows faster than $n \log n$ as n increases. The graph in figure 6.1 shows some similar functions.

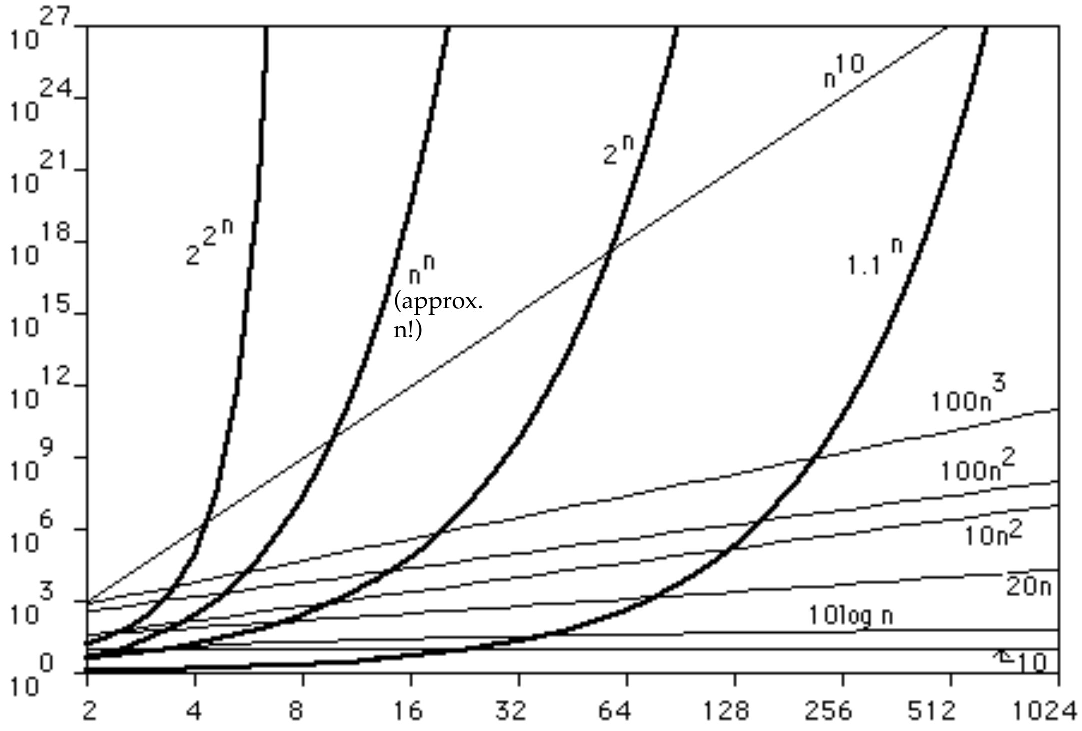


Figure 6.1: growth rates of functions (log scales)

6.1.2 Why these functions?

Why do the functions above come up so often? Because algorithms typically work in one of a few standard ways:

1. Simple stack pushes and pops will take **constant time,** assuming the data is of a small size (e.g., 32 bits for each entry in an integer stack). Algorithms running in low constant time are 'perfect'.

2. An algorithm may work by dealing with one character of its input at a time, taking the same time for each. Thus (roughly, and up to a choice of time units) we get

   $$f(n) = n \qquad \text{— \textbf{linear time.}}$$

   Algorithms running in linear time are usually very good.

3. It may repeatedly loop through the entire input to eliminate one item (e.g., the largest). In this case we'll have

   $$f(n) = n + f(n-1).$$

   So
   $$\begin{aligned} f(n) &= n + f(n-1) \\ &= n + (n-1+f(n-2)) \\ &= \dots \\ &= n + (n-1) + (n-2) + \cdots + 3 + 2 + f(1). \end{aligned}$$

This is an **arithmetic progression,** so we get $f(n) = n^2/2 + n/2 + k$ for some constant k. The k and $n/2$ are small compared with n^2 for large n, so this algorithm runs in **quadratic time.** An algorithm that considers all n^2 pairs of characters of the n-character input also takes quadratic time.

Algorithms running in quadratic time can be sluggish on larger inputs.

4. Maybe the algorithm throws away half the input in each step, as in binary search, or heap accessing (see §7.3.4). So

$$f(n) = 1 + f(n/2)$$

(this is only an approximation if n is not a power of 2). Then letting $n = 2^x$, we get

$$\begin{aligned} f(2^x) &= 1 + f(2^{x-1}) \\ &= 1 + (1 + f(2^{x-2})) \\ &= \ldots \\ &= x + f(2^0) \\ &= x + k \end{aligned}$$

for some constant k. As $f(2^x)$ is about x, $f(n)$ is about $\log_2(n)$: this algorithm runs in **log time.**

The log function grows very slowly and algorithms with this run-time are usually excellent in practice.

5. The algorithm might recursively divide the input into two halves, but make a pass through the entire input before, during or after splitting it. This is a very common 'divide and conquer' scheme, used in mergesort, quicksort (but see below), etc. We have

$$f(n) = n + 2f(n/2)$$

roughly — n to pass through the whole input, plus $f(n/2)$ to process each half. So, using a trick of *dividing by the argument,* and letting $n = 2^x$ again,

$$\begin{aligned} f(2^x)/2^x &= 2^x/2^x + 2 \cdot f(2^x/2)/2^x \\ &= 1 + f(2^{x-1})/2^{x-1} \\ &= 1 + (1 + f(2^{x-2})/2^{x-2}) \\ &= \ldots \\ &= x + f(2^0)/2^0 \\ &= x + c, \end{aligned}$$

for some constant c. So $f(2^x) = 2^x(x+c) = 2^x \cdot x +$ smaller terms. Thus, roughly, $f(n) = n\log_2 n +$ smaller terms. We have a **log linear algorithm.**

Log linear algorithms are usually good in practice.

Exercise 6.1 What if it divides the input into three?

6. Maybe the input is a set of n positive and negative whole numbers, and the algorithm must find a subset of the numbers that add up to 0. If it does an exhaustive search, in the worst case it has to check all possible subsets — 2^n of them. This takes **exponential time.** Algorithms with this run time are probably going to be appalling, unless their average-case performance is better (the **simplex algorithm** from optimisation is an example).

7. If the problem is to find all anagrams of the n-letter input word, it might try all $n!$ possible orderings of the n letters. The factorial function $n! = 1 \times 2 \times 3 \times \ldots n$ grows at about the rate of n^n, even faster than 2^n.

Quicksort This is a borderline case. In the average case, the hope is that in each recursive call quicksort divides its input into two roughly *equal* parts. By case 5 above, this means log linear time average-case performance. In the worst case, in each recursive call the division gives a big and a small part. Then case 3 is more appropriate, and it shows that quicksort runs in quadratic time in the worst case.

In practice, quicksort performs very well — and it can sort ‘in place’ without using much extra space, which is good for large sorting jobs.

6.1.3 The O-notation (revision)

This helps us make precise the notion ‘my algorithm runs in log time’ etc. It lets us talk about functions $f(n)$ **for large** n**,** and **up to a constant factor.**

Definition 6.2 Let f, g be real-valued functions on whole numbers (i.e., functions from $\{1,2,3,\ldots\}$ into the set of real numbers).

1. We say that f **is** $O(g)$ **(‘**f **is of the order of** g**’)** if there are numbers m and c such that $f(n) \leq c \cdot g(n)$ whenever $n \geq m$.

2. We say that f **is** $\theta(g)$ **(‘theta of** g**’)** if f is $O(g)$ and g is $O(f)$. This means that f and g have the same order of growth.

So f is of the order of g iff for all large enough n (i.e., $n \geq m$), $f(n)$ is at most $g(n)$ up to some constant factor, c. Taking logs, this means that *for all large enough* n, $\log f(n) \leq c' + \log g(n)$, where c' is a constant ($= \log c$). I.e., $\log f(n)$ is **eventually** no more than a constant amount above $\log g(n)$.

Similarly, f is $\theta(g)$ iff there is a constant c such that for all large enough n, $\log f(n)$ and $\log g(n)$ differ by at most c.

So in the graph of figure 6.1, f is $O(g)$ iff for large enough n, the line for f is at most a constant amount higher than that for g (it could be much lower, though!) And f is $\theta(g)$ if eventually (i.e., for large enough n) the lines for f and g are vertically separated by at most a fixed distance.

Definition 6.3 We can now say that an (implementation of an) algorithm runs in **log time** (or **linear time**) if its run-time function $f(n)$ is $\theta(\log n)$ (or $\theta(n)$, respectively). We define **runs in quadratic, log linear, exponential time,** etc, in the same way.

Exercises 6.4

1. Show that f is $\theta(g)$ iff there are m, c, d (with $d > 0$, possibly a fraction) such that $d \cdot g(n) \leq f(n) \leq c \cdot g(n)$ for all $n \geq m$.

2. Show that if f is $O(g)$ then there are c, d such that for all n, $f(n) \leq \max(c, d \cdot g(n))$. Is the converse true?

3. [Quite long.] Check that the functions in §6.1.1 are listed in increasing order of growth: if f is before g in the list then f is $O(g)$ but not $\theta(g)$. [Calculus, taking logs, etc., may help.]

4. Let $\mathcal{F}$ be the set of all real-valued functions on whole numbers. Define a binary relation E on $\mathcal{F}$ by: $E(f,g)$ holds iff f is $\theta(g)$. Show that E is an equivalence relation on $\mathcal{F}$ (see §7.1 if you've forgotten what an equivalence relation is). (Some people *define* $\theta(f)$ to be the E-class of f.)

   Show also that 'f is $O(g)$' is a pre-order (reflexive and transitive) on $\mathcal{F}$.

5. Show that for any $a, b, x > 0$, $\log_a(x) = \log_b(x) \cdot \log_a(b)$. Deduce that $\log_a(n)$ is $\theta(\log_b(n))$. Conclude that when we say an algorithm runs in log time, we don't need to say what base the log is taken to.

6.1.4 Merits of rough analysis

Note that the statement 'my algorithm runs in log time' (or whatever) will only be an accurate description of its actual performance *for large n* (so the smaller terms are insignificant), and *up to a constant factor* (c). A more detailed analysis can be done if these uncertainties are significant, but:

- A rough analysis is quicker to do, and often surprisingly accurate. E.g., most time will be spent in the inner loop ('90% of the time is spent in 10% of the code'), so you might even *ignore* the rest of your program.

- You may not know whether the algorithm will be run on a Cray or a PC (or both). Each instruction runs faster by roughly a constant factor on a Cray than on a PC, so we might prefer to keep the constant factor c above.

- You may not know how good the implementation of the algorithm is. If it uses more instructions than needed in the inner loop, this will increase its running time by roughly a constant factor.

- The run time of an algorithm will depend on the format of the data provided to it. E.g., hexadecimal addition may run faster than decimal addition. So if you don't know the data format, or it may change, an uncertainty of a constant factor is again introduced.

6.1.5 Demerits of rough analysis

In the analysis it's often easy to show f is O(one of the functions g above). To prove f is $\theta(g)$ is harder. If you can only show the worst case run time function f to be $O(g)$, so that $f(n) \leq c \cdot g(n)$ whenever $n \geq m$, then remember that g is an upper bound only.

In any case, whether you have an O- or a θ-estimate,

- the worst case may be rare;
- the constant c is unknown and could be large;
- the constant m is unknown and could be large.

This can be important in practice. For example, though for very large n we have $2n\log_2(n)^2 < n^{3/2}$, in fact $2n\log_2(n)^2 > n^{3/2}$ until n gets to about half a million. The moral is: although you should think twice before using an n^2 algorithm instead of an $n\log n$ one, nonetheless the n^2 algorithm may sometimes be the best choice.

6.1.6 The bottom line (almost...)

Generally, algorithms that run in linear or even log linear time are fine. Quadratic time algorithms are not so good for very large inputs, but algorithms with $f(n)$ up to n^5 are of some use. Exponential time algorithms are hopeless even for quite small inputs, unless their average-case performance is much better (e.g., the simplex algorithm).

6.1.7 Average case run time

The average case run time is harder to obtain and more machine-dependent. So your long, complex analyses may only be of any use on your current machine, and may not be worth the effort. Also the average case may not be easy to define mathematically or helpfully: what is 'average' English text? But average cases are useful in geometrical and sorting algorithms, etc.

6.2 Choice of algorithm

So how to choose, in the end? Don't ignore the run time function $f(n)$. Much better algorithms may not be much harder to implement.

But don't idolise it either, as the run time will only be estimated by $f(n)$ for large inputs (and other caveats above). Moreover, programmers' time is money, so it may be best to keep things simple. The constant factors in $f(n)$ may be unknown or wrong: a factor of 10 is easy to overlook in a rough calculation. So use empirical tests to check performance. But beware: empirical comparison of two algorithm implementations can be misleading unless done on similar machines, delays due to shared access are borne in mind, and equal attention has been paid to optimising the two implementations (e.g., by cutting redundant instructions and procedure calls in the inner loop).

Your choice of algorithm may also be influenced by other factors, such as the prevalent data structures (linked list, tree, etc.,) in your programming environment, as some algorithms take advantage of certain structures. The algorithm's space (memory) usage may also be important.

6.3 Implementation

We've seen some of the factors involved in *choosing* an algorithm. But the same algorithm can be *implemented* in many different ways, even in the same language. What advice is there here?

6.3.1 Keep it simple

Go for simplicity first. A brute force solution may be fine if the algorithm is only going to be used infrequently, or for small inputs. So why waste your expert time? (Of course, the usage may change, so be ready to re-implement.) If the result is too slow, it's still a good check of correctness for more sophisticated algorithms or implementations. There are algorithms that are prey to bugs that merely slow up performance, maintaining correctness. A naïve program can be used for speed comparisons, showing up such bugs.

6.3.2 Optimisation

Only do this if it's worth it: if the implementation will be used a lot, or if you know it can be improved. If it is, improve it incrementally:

- Get a simple version going first. It may do as it is!
- Check any known maths for the algorithm against your simple implementation. E.g., if a supposedly linear time algorithm takes ages to run, something's wrong.
- Find the inner loop and shorten it. Use a profiling option to find the heavily-used instructions. Are they in what you think is the inner loop? Look at every instruction in the inner loop. Is it necessary, or inefficient? Remove procedure calls from the loop, or even (last resort!) implement it in assembler. But try to preserve robustness and machine-independence, or more programmers' time may be needed later.
- Check improvements by empirical testing at each stage — this helps to eliminate the bad improvements. Watch out for diminishing returns: your time may nowadays be more expensive than the computer's.

An improvement by a factor of even 4 or 5 might be obtained in the end. You may even end up improving the algorithm itself.

If you're building a large system, try to keep it amenable to improvements in the algorithms it uses, as they can be crucial in performance.

6.4 Useful books on algorithms

1. Robert Sedgewick, *Algorithms,* Addison-Wesley, 2nd ed., 1988. A practical guide to many useful algorithms and their implementation.
2. A.V. Aho, J.E. Hopcroft, J.D. Ullman, *The design and analysis of algorithms,* Addison-Wesley, 1975. For asymptotic worst-case performance.
3. D.E, Knuth, *The art of computer programming,* 3 volumes, Addison-Wesley. Does more average-case analysis, and a full reference for particular algorithms.
4. G.H. Gonnet, *Handbook of algorithms and data structures,* Addison-Wesley, 1984. Worst- and average-case analysis, and covers more recent algorithms.

The last three are listed in Sedgewick. Maybe that's why all four are from the same publisher.

6.5 Summary of section

We examined some practical issues and advice to do with choice and implementation of algorithms. We introduced the run time function of an algorithm, in worst case or average case form. It is often most sensible to make do with a rough calculation of the run time function, obtaining it only for large input size (n) and up to a constant factor (c). In practice, more detailed calculations may be needed. The O-notation helps to compare functions in these terms. We saw how some common algorithm designs give rise to certain run time functions $(\log n, n, n \log n, n^2)$; these are calculated using recursive equations by considering the 'inner loop' of the algorithm.

7. Graph algorithms

We will now examine some useful algorithms. We concentrate on algorithms to handle graphs, as they are useful, quite challenging, easy to visualise, and will be needed in Part III.

7.1 Graphs: the basics

Relations (revision) Recall that a **binary relation** $R(x,y)$ on a set X is a subset of $X \times X$. We usually write '$R(x,y)$ holds', or just '$R(x,y)$', rather than '$(x,y) \in R$'. R is said to be:

- **reflexive,** if $R(x,x)$ holds for all x in X
- **irreflexive,** if $R(x,x)$ holds for no element x in X
- **symmetric,** if whenever $R(x,y)$ holds then so does $R(y,x)$
- **transitive,** if whenever $R(x,y)$ and $R(y,z)$ hold then so does $R(x,z)$
- an **equivalence relation,** if it is reflexive, symmetric, and transitive.

Definition 7.1 A **graph** is a pair (V,E), where V is a non-empty set of **vertices** or **nodes,** and E is a symmetric, irreflexive binary relation on V.

We can represent a graph by drawing the nodes as little circles, and putting a line ('edge') between nodes x,y iff $E(x,y)$ holds. In figure 7.1, the graph is $(\{1,2,3,4,5,6\}, \{(1,3),(3,1), (2,3),(3,2), (4,5),(5,4), (5,6),(6,5), (4,6),(6,4)\})$.

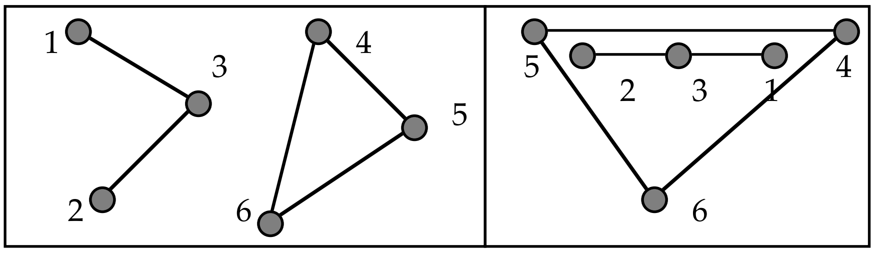


Figure 7.1: two drawings of the same graph (6 nodes, 5 edges)

If $G = (V,E)$ is a graph, and $x,y \in V$, we say that there's an **edge** from x to y iff $E(x,y)$ holds. We think of (x,y) and (y,x) as representing the *same* edge, so the number of edges in G is half the number of (x,y) for which E holds. So the graph in figure 7.1 has 5 edges, not 10. We'll usually write n for the number of nodes of a graph, and e for the number of edges.

Exercise 7.2 Show that any graph with n nodes has at most $n(n-1)/2$ edges.

Examples There are many examples of graphs, and many problems can be represented in terms of graphs. The London tube stations form the nodes of a graph whose edges are the stations (s,s') that are one stop apart. The world's airports form the nodes of a graph whose edge pairs consist of airports that one can fly between by Aer Lingus without changing planes. The problem of pairing university places to students can be considered as a graph: we have a node for each place and each student, and we ask for every student to be connected to a unique place by an edge. Electrical circuits: the wires form the edges between the nodes (transistors etc).

Other graphs There are more 'advanced' graphs. Directed graphs do not require that E is symmetric. In effect, we allow arrows on edges, giving each edge a direction. There may now be an edge from x to y, but no edge from y to x. The nodes could represent tasks, and an arrow from a to b could say that we should do a before b. They might be players in a tournament: an arrow from Merlin to Gandalf means Merlin won.

Weighted graphs (see section 8) have edges labelled with numbers, called weights or lengths. If the nodes represent cities, the weights might represent distances between them, or costs or times of travel.

Graph algorithms Many useful algorithms for dealing with graphs are known, but as we will see, some are not easy. For example, no fast way to tell whether a graph can be drawn on paper without edges crossing one another was known until 1974, when R.E. Tarjan developed an ingenious *linear time* algorithm. We'll soon see graph problems with no known efficient algorithmic solution.

7.2 Representing graphs

How to *input* a graph (V,E) into a computer? First rename the vertices so that they are called $1,2,\ldots,n$ for some n. (Maybe use a hashing technique to do this if the vertices originally have names, like London tube stations.) Typically you'll then input the number n of vertices, followed by a delimiter $*$, followed by a list of all pairs $(x,y) \in E$ — perhaps omitting the dual pair (y,x) unless the graph is directed. The same information can be input using an $n \times n$ Boolean array (the **edge matrix** of the graph). If the graph is weighted, the weight of an edge can be added after the edge. This format can be input to a Turing machine if all numbers are input in binary (say) and there's a terminating $\wedge$.

How to *represent* the graph in a computer? We could just use the edge matrix (see Sedgewick's book). But it's often better to use a **linked list,** especially if the graph has relatively few edges. Finding edges is then faster. The graph in figure 7.1 would be represented by:

$$\begin{array}{l}
1 \to 3 \\
2 \to 3 \\
3 \to 1 \to 2 \\
4 \to 5 \to 6 \\
5 \to 4 \to 6 \\
6 \to 5 \to 4
\end{array}$$

Each line begins with a header vertex (1–6), and lists all vertices connected to it by an edge. There's redundancy (e.g., the edge $(1,3)$ shows up in lines 1 and 3), but this is useful for queries such as 'which vertices are connected to x?' If operations such as deleting a node from a graph are important, it can help to add pointers from the head of each line of the list to the corresponding entries in bodies of the other lines. Here, 1 would get a special pointer to the second entry of line 3. The overhead cost of doing this should be borne in mind.

A linked list can represent directed graphs: the entries in a line headed by *x* are those nodes *y* such that there's an arrow from *x* to *y*. The weights in a weighted graph can be held in an integer field attached to each non-header entry in the list.

7.3 Algorithm for searching a graph

We want to devise a general purpose algorithm that will rapidly visit every node of a graph, travelling only along graph edges. Such an algorithm will be useful for graph searches, measurements, etc. For example, starting from Dublin (or anywhere reachable from Dublin, for that matter) it would trace out all airports reachable by Aer Lingus, even with plane changes.

7.3.1 The search strategy

The general idea will be this. At any stage of the search, some graph vertices will have been **visited.** Others will be accessible from the already-visited vertices in one step, by a single edge. They are on the **fringe** of the visited nodes, and are ripe for visiting next. The other vertices will be **far away,** neither visited nor (yet) on the fringe.

We will repeatedly:

- choose a fringe vertex,
- visit it and so promote it to 'visited' status
- replace it on the fringe by its immediate (but unvisited) neighbours: i.e., those unvisited nodes that are connected to it by an edge.

7.3.2 Depth-first and breadth-first search

At each stage, we must decide which fringe vertex to visit next. The choice depends on what we're trying to do. Two common choices are:

depth-first Visit the *newest* fringe vertex next: the one that most recently became a fringe vertex. (Last in, first out: implementation could use a stack.)

breadth-first Visit the *oldest* fringe vertex next. (First in, first out: implementation could use a queue.)

In the *breadth-first* approach, *all neighbours of the start node* get visited first, then the next nearest, and so on. This strategy would be good for a group of people searching for something in a maze. In contrast, the neighbours of the starting node tend to be visited later in the *depth-first* approach. As the next place to visit is usually close by, this approach is good for a single person searching a maze.

Compared with depth-first search (heavy arrows in figure 7.4 below), the edges traced out in breadth-first search (figure 7.5) tend to form a squatter and bushier pattern, with many short branches. Figure 7.2 shows the kind of shapes to expect. Note that unlike when searching a *tree* (e.g., in implementing Prolog), the difference between breadth first and depth-first search in a *graph* is not just the *order* in which vertices are visited. The *path actually taken* also differs.

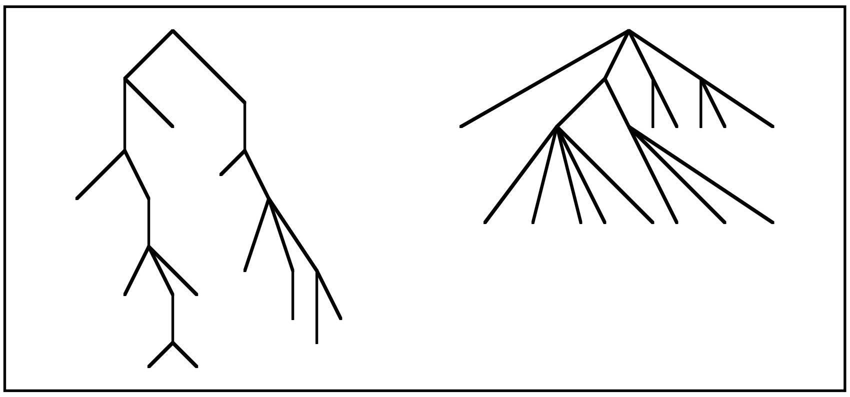


Figure 7.2: depth-first and (right) breadth-first search trees

7.3.3 More general priority schemes

There's a more general way of choosing a fringe vertex to visit next. Whenever we add a vertex to the fringe, we assign it a *priority*. At each stage, the fringe vertex with the highest priority will be visited next. If there are several fringe vertices with equal priorities, we can choose any of them; the algorithm is non-deterministic.

We can choose any scheme to assign priority. If we let highest priority = newest, we get depth-first search; if we let highest priority = oldest, we get breadth-first search. So both breadth-first and depth-first search can be done using priorities. We'll see the effects of other priority schemes in §8.3.

7.3.4 The data structure: priority queue

There is a data structure called a **priority queue** for implementing general, user-chosen priorities. It generalises stacks and queues. It's often implemented as a **heap**, and any access typically takes log time (with stacks and queues, access takes constant time).

For our purposes, we'll assume the priority queue has the following specification.

7.3.4.1 Specification of priority queue

The priority queue consists of *triples* (x, y, p), where:

- x is an **entry**;
- y is a **label** of x (it can be any extra information we want);
- p is the **priority** of x. It's a number.

It is x that's in the queue (with label y and priority p). So *at most one* x-entry is allowed in the queue at any time.

1. We can 'push' onto the priority queue any entry x, with any label y, and any priority p.
2. The push has no effect if x is already an entry in the queue with higher or equal priority than p. I.e., if the queue contains a triple (x,z,q), where z is any label and q is a priority higher than or equal to p, the push *doesn't do anything.*
3. Any x-entry already in the queue but with lower priority than p is *removed.* I.e., if the queue contains a triple (x,z,q), with any label z, and q a lower priority than p, then the push *replaces* it with (x,y,p).
4. A 'pop' operation always removes from the queue (and returns) an entry (x,y,p) with highest possible priority.

7.3.5 The visit algorithm in detail

In 'pseudo-code', our algorithm is as follows. The nodes of the graph are represented by the numbers 1 to n.

```
1   $visited(n)$: global Boolean array, initially all false; $x$: integer
2   repeat with $x = 1$ to $n$
3       if not $visited(x)$ then visit($x$)
4   end repeat
5   procedure visit($x$)
6   $x,y,z$ : integer                % to represent vertices
7   empty the fringe (priority queue)
8   push $x$ into fringe, with label $*$, and any priority
9   repeat until fringe is empty
10      pop $(x,y,p)$ from fringe  % So $x$ was the queue entry; $y$ was its label;
                                   % and $p$ was the (highest possible) priority.
11      set $visited(x)$ to true     % Anything else you want to do to the new
                                   % current node, $x$, such as printing it, do it here!
                                   % $y$ tells us the edge $(y,x)$ used to get to $x$.
12      repeat for all nodes $z$ connected to $x$ by an edge
13          if not $visited(z)$ then push $z$ into fringe, with label $x$ and chosen priority
14      end repeat
15  end repeat
16  end visit
```

Note that in line 10, there could be several fringe nodes of equal highest priority. The priority queue non-deterministically pops any such node. The repeat in line 12 does not need to test all nodes z of the graph: it just examines the body of line x of the linked list (§7.2). The order in which line 12 runs through the x is not specified. The label y in line 10 is useful because it tells us how we got to the current node, x. We usually want to know the *route* we took when searching the graph, as well as which nodes we visited and in what order. Knowing the order that we visited the nodes in is *not* enough to determine the route: see the example below.

7.3.6 Depth-first search: example

Let's see how the algorithm runs on an example graph. Visiting the *newest fringe vertex first* conducts a **depth-first search** of the graph, only moving along edges. Running visit(1) on the graph in figure 7.3, represented by the linked list below, visits nodes in the order 1,7,6,3,5,4,2. See figure 7.4.

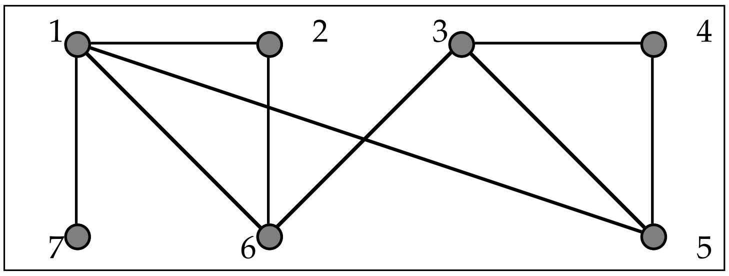


Figure 7.3: another graph

$1 \rightarrow 2 \rightarrow 5 \rightarrow 6 \rightarrow 7$
$2 \rightarrow 1 \rightarrow 6$
$3 \rightarrow 4 \rightarrow 5 \rightarrow 6$
$4 \rightarrow 3 \rightarrow 5$
$5 \rightarrow 1 \rightarrow 3 \rightarrow 4$
$6 \rightarrow 1 \rightarrow 2 \rightarrow 3$
$7 \rightarrow 1$

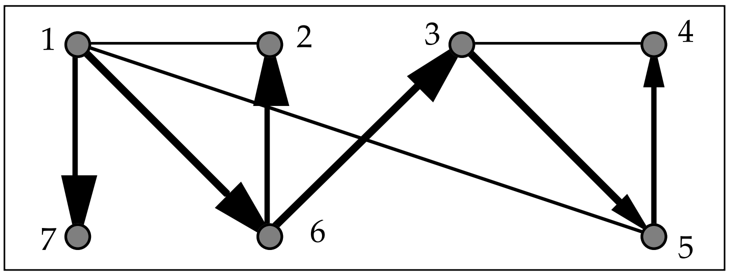


Figure 7.4: depth-first search

The 'tree' produced (heavy arrows in figure 7.4) tends to have only a few branches, which tend to be long. Cf. figure 7.2 (left).

7.3.6.1 Execution

We show the execution as a table. Initially, the fringe consists of $(1,*,0)$, where the label $*$ indicates we've just started, 1 is the starting node, and 0 is the (arbitrary) priority. We pop it from the fringe. The immediate neighbours of 1 are numbered 2, 5, 6 and 7. Assume we push them in this order. The fringe becomes $(2,1,1)$, $(5,1,2)$, $(6,1,3)$, $(7,1,4)$, in the format of §7.3.4.1; the third figure is the (increasing) priority.

fringe	pop	visited	print	push	comments
$(1,*,0)$	$(1,*,0)$	1		$(2,1,1)$ $(5,1,2)$ $(6,1,3)$ $(7,1,4)$	
$(2,1,1)$ $(5,1,2)$ $(6,1,3)$ $(7,1,4)$	$(7,1,4)$	7	edge '1,7'	–	No unvisited neighbours of 7, so no push.
$(2,1,1)$ $(5,1,2)$ $(6,1,3)$	$(6,1,3)$	6	edge '1,6'	$(2,6,5)$ $(3,6,6)$	'Backtrack' to visit 6 from 1. Push of 2 has better priority than the current fringe entry $(2,1,1)$, which is replaced.
$(5,1,2)$ $(2,6,5)$ $(3,6,6)$	$(3,6,6)$	3	edge '6,3'	$(4,3,7)$ $(5,3,8)$	The view of 5 from 3 replaces the older view from 1.
$(2,6,5)$ $(4,3,7)$ $(5,3,8)$	$(5,3,8)$	5	edge '3,5'	$(4,5,9)$	Again, this push involves updating the priority of node 4.
$(2,6,5)$ $(4,5,9)$	$(4,5,9)$	4	edge '5,4'	–	No unvisited neighbours of 4, so no push.
$(2,6,5)$	$(2,6,5)$	2	edge '6,2'	–	Another backtrack! No unvisited neighbours of 2, so no pushes.
empty					Terminate call of visit(1). Return.

7.3.6.2 Warning: significance of the label 'y'

Notice that the nodes were visited in the order 1,7,6,3,5,4,2, but that this does not determine the route. Did we go to 2 from 1 or from 6? Did we arrive at 4 from 3, or from 5? That's why we have to keep the label y in the queue, so that we can tell how we got to each node. This will be even more important in section 8, where the route taken is what we're actually looking for.

Exercises 7.3

1. What route do we get if we start at 2, or 4, instead of 1?
2. What alterations to the code in §7.3.5 would be needed to implement the fringe with an ordinary stack?

3. Work out how to deduce the path taken in depth-first search, if you know only (a) the graph, and (b) the order in which its nodes were visited. (The main problem, of course, is to handle backtracking.) Can you do the same for breadth-first search (see below)?

7.3.7 Breadth-first search: example

Visiting the *oldest fringe vertex first* conducts a **breadth-first search** of the graph, only moving along edges. Running visit(1) on the graph above leads to the sequence 1,2,5,6,7,3,4 of visits shown in figure 7.5. Note that the tree is squatter than in the depth-first case.

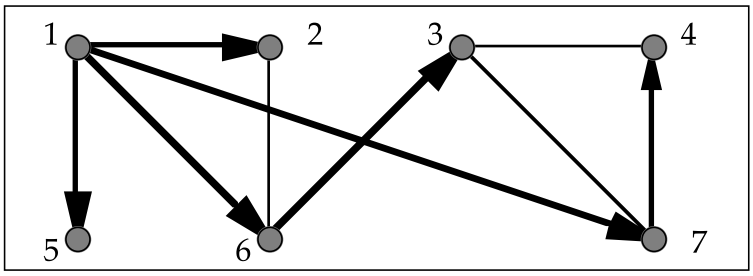


Figure 7.5: breadth-first search

Exercise 7.4 Work out the execution sequence and try it from different starting nodes.

7.3.8 Run time of the algorithm

Let's simplify by approximating, and only counting data accesses. The fringe is administered by a priority queue that stores nodes in priority order. As we said, this is often implemented by a heap: a kind of binary data structure. If a heap contains m entries, accessing it (read or write) takes time $\log m$ in the worst case (cf. binary search, §6.1.2). Suppose the graph has n nodes and e edges. Clearly the fringe never contains more than n entries, so let's assume each fringe access takes time $\log n$ (worst case). However, we count only 1 for emptying the fringe in line 7 of the code, and 1 for the first push in line 8. Obviously, each *visited* array access takes constant time: independent of n and e.

- Initialisation of the n elements of the *visited* array to false (line 1), the n reads from it in line 3, and initialisation of the fringe (lines 7–8): total $= 2n+2$.

- Every node is removed from the fringe exactly once (line 10). This involves n accesses, each taking time $\leq \log n$. Total: $n \log n$.

- For each node x visited, every neighbour z is obtained from the linked list (only count 1 for each access to this, since we just follow the links) and checked to see if it's been visited (lines 12–16; count 1). As each graph edge (x,z) gets checked twice in this way, once from x and once from z, the total time cost here is $2 \times 2e = 4e$.

- Not all z connected to x get written to the fringe, because of the test in line 13. If z is put on the fringe when at x, then x will not be put on the fringe later, when at z, as x will by then have been visited. So each edge results in at most one fringe-write. Hence the fringe is written to at most e times. Each write takes $\log n$. Total: $e \log n$.

Grand total: $2n+2+n\log n+4e+e\log n$. This is satisfactorily low, and the algorithm is useful in practice. Neglecting smaller terms, we conclude:

> The algorithm takes time $O((n+e)\log n)$ (i.e., log linear time) in the worst case.

The performance of graph algorithms is often stated as $f(n,e)$, not just $f(n)$.

7.4 Paths and connectedness

Definition 7.5 If x, y are nodes in a graph, a **path** from x to y in the graph is a sequence $v_0, v_1, \ldots, v_k$ of nodes, such that $k > 0$, $v_0 = x$, $v_k = y$, and (v_i, v_{i+1}) is an edge for each i with $0 \leq i < k$. The **length** of the path is k — i.e., the number of *edges* in it. The path is **non-backtracking** if the v_i are all different.

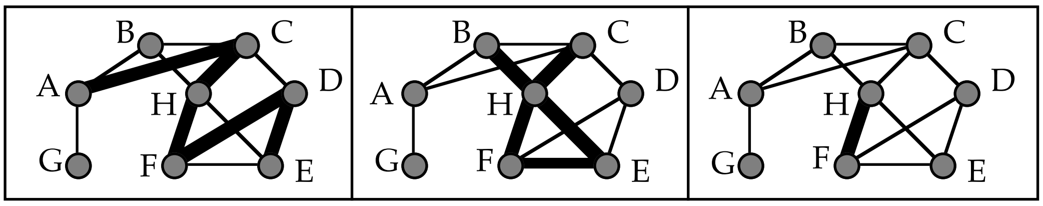


Figure 7.6: paths

In figure 7.6 (left), the heavy lines show the path ACHFDE from A to E. (They also represents the path EDFHCA from E to A — we can't tell the direction from the figure.) This path is non-backtracking. In the centre, BHFEHC (or BHEFHC?) is a path from B to C, but it's a *backtracking path* because H comes up twice. On the right, the heavy line is an attempt to represent the path HFH, which again is backtracking.

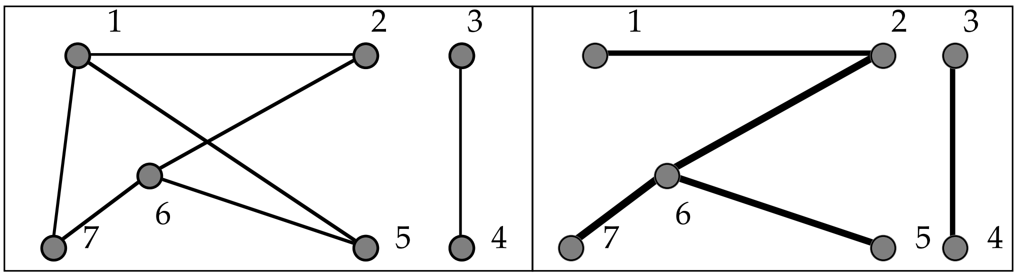


Figure 7.7: a disconnected graph; a depth-first search tree for it

7.4.1 Connectedness

The graph of figure 7.3 is **connected:** there's a path along edges between any two distinct (= different) vertices. In contrast, the graph on the left of figure 7.7 is disconnected. There's no path from 1 to 3.

What if we run the algorithm on a disconnected graph like this? In depth-first mode, it traces out the heavy lines on the right of figure 7.7. Visit(1) starts at 1 and worms its way round to 2,6,5 and 7. But then it terminates, and visit(3) is called (line 3 of the code in §7.3.5). Whatever priority scheme we adopt, visit(1) will only visit all nodes reachable from 1.

7.4.2 Connected components

The nodes reachable from a given node of a graph form a **connected component** of the graph. Any graph divides up into disjoint connected components; it's connected iff there's only one connected component. On any graph, a call of visit(x) visits all the nodes of the connected component containing x. Obviously, visit can't jump between connected components by using edges, so we have to set it off again on each component. Line 3 of the code does this: it will be executed once for each connected component. The number of times visit is called counts the connected components of the graph.

Exercises 7.6

1. Try the algorithm on figure 7.7, starting at 4, in depth- and breadth-first modes. How often is visit called in each case?

2. Let $G = (V,E)$ be a graph. Define a binary relation $\sim$ on V by: $x \sim y$ if $x = y$ or there is a path from x to y. Check that $\sim$ is an equivalence relation on V. (The equivalence classes are the connected components of G.)

7.5 Trees, spanning trees

We've already seen in the examples that our algorithm traces out a tree-like graph: the heavy lines in figures 7.4–7.5 and 7.7. We can say quite a lot about trees.

7.5.1 Trees

A tree is a special kind of graph:

Definition 7.7 (very important!) A **tree** is a connected graph with no cycles.

But what's a cycle?

Definition 7.8 A **cycle** in a graph is a path from a node back to itself without using a node or edge twice.

So paths of the form ABA, and figures-of-eight such as ABCAEDA (see figure 7.8), are not cycles.

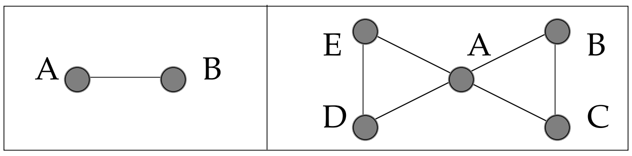


Figure 7.8: these are not cycles!

In figure 7.7, 1,2,6,5,1 is a cycle.

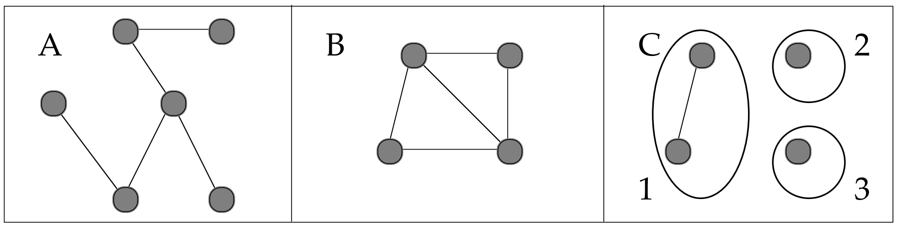


Figure 7.9: *A* (tree), *B*,*C* (not trees)

In figure 7.9, *A* is a tree. *B* has a cycle (several in fact), so isn't a tree. *C* has no cycles but isn't connected, so isn't a tree. It splits into three connected components, the ringed 1,2 and 3, which are trees. Such a 'disconnected tree' is called a **forest.**

7.5.2 Spanning trees

A call of the visit procedure always traces out a tree. For as it always visits new (unvisited) nodes, it never traces out a cycle. Moreover, if the graph is *connected*, a single visit call visits all the nodes, so the whole algorithm's trace is a tree. As it contains all the nodes, it's called a **spanning tree** for the graph. If the graph is disconnected we get a spanning tree for each connected component.

Definition 7.9 (very important!) A tree containing all the nodes of a graph (and only using edges from the graph) is called a **spanning tree** for the graph.

Only a connected graph can have a spanning tree, but it can have more than one spanning tree. The breadth-first and depth-first searches above gave different spanning trees (figures 7.4 and 7.5).

A spanning tree is the quickest way to visit all the nodes of a (connected) graph from a starting node, as no edge is wasted. The algorithm starts with the initial vertex and no edges. Every step adds a single node and edge, so the number of nodes visited is always one ahead of the number of edges. Because of this, the number of edges in the final spanning tree is *one less* than the number of nodes.

If we run the algorithm on a *tree,* it will trace out the entire tree, using all the edges in the tree. (A tree T is connected, so the algorithm generates a spanning tree T' of T. Every edge of T is in T'. For if $e = (x,y)$ were not an edge of T', then as x and y are in T', there's a (non-backtracking) path from x to y in T'; and this path, plus e, gives a cycle in the original tree T — impossible. So $T' = T$.) Thus we see:

Proposition 7.10 *Any tree with n vertices has $n-1$ edges.*

7.5.3 Testing for cycles

If the original *connected* graph (with n nodes) has $\geq n$ edges, it must have a cycle. For, any edge not in a spanning tree must connect two nodes that are already joined by a path in the tree. Adding the extra edge to this path gives a cycle.

We can use this to find out if a *connected* graph has a cycle. Just count its vertices and edges. There's a cycle iff (no. of edges) $\geq$ (no. of vertices). If the graph is disconnected, we could do this for every connected component in turn.

Another way to test for cycles is to modify the algorithm (§7.3.5) to check, each time round the main loop of lines 9–17, whether the test in line 13 is failed more than once. *If this ever happens, there's a cycle,* because the algorithm has found two 'visited' neighbours z of the current node x. One such z is the node one step higher in the tree than x (if any) — this is the node we arrived at x from. The other z indicates a cycle.

Example: when at node 5 during the depth-first search of figure 7.4, nodes 1 and 3 were rejected as fringe contenders because they had been visited earlier. Node 3 is the previous node in the search tree; but node 1's visibility from 5 indicates the presence of a cycle, as we can travel from 5 to 1 directly and then return to 5 via the tree (via 6 and 3). See figure 7.10.

Exercises 7.11

1. Let $U = (S,L)$ where S is the set of London tube stations, and $(x,y) \in L$ iff x is exactly one stop away from y. Is U a connected graph? Is it a tree? If not, find a cycle.

2. Show that any two distinct nodes x,y of a tree are connected by a *unique* non-backtracking path.

3. Show that *any* graph (even if disconnected) with at least as many edges as vertices must contain a cycle. [Hint: you could add some edges between components.]

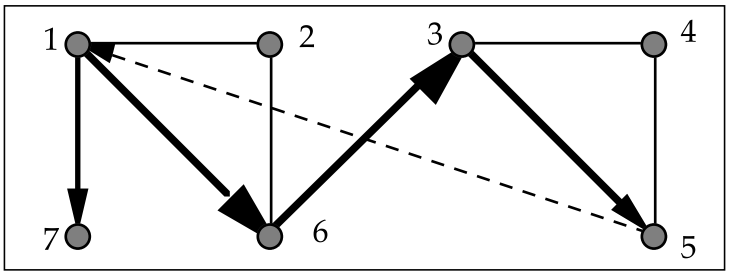


Figure 7.10: cycle 51635

4. Show that no graph with 10 nodes and 8 edges is connected.

5. Show that any connected graph with n nodes and $n-1$ edges (for some $n \geq 1$) is a tree. Find a graph with n nodes and $n-1$ edges (for some n) that's not a tree.

6. Let $G = (V, E)$ be a graph with n nodes, and $T = (V, P)$ a subgraph (so every edge of T is an edge of G). Show that T is a spanning tree of G iff it's connected and has $n-1$ edges.

7. Figure 7.11 is a picture of the maze at Hampton Court, on the river west of London, made by Messrs. Henry Wise and George London in 1692. Draw a graph for this maze. Put nodes at the entrance, the centre, and at all 'choice points' and dead ends. Join two nodes with an edge if you can walk directly between them without going through another node. Is the graph connected? Is it a tree? What information about the maze is not represented in the graph?

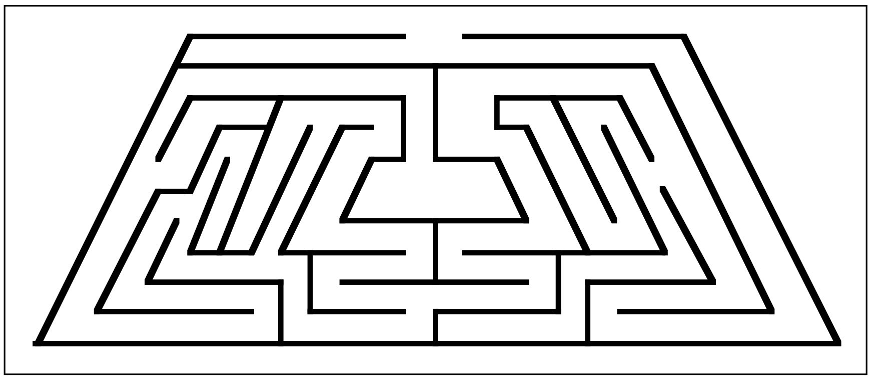


Figure 7.11: Hampton Court maze

7.6 Complete graphs

Graphs with the maximum possible number of edges are called **complete graphs.**

Definition 7.12 A graph (V,E) is said to be **complete** if $(x,y) \in E$ for all $x,y \in V$ with $x \neq y$.

Exercises 7.13

1. What spanning trees are obtained by depth first and breadth-first search in the complete graph of figure 7.12? How many writes to the fringe are there in each case?

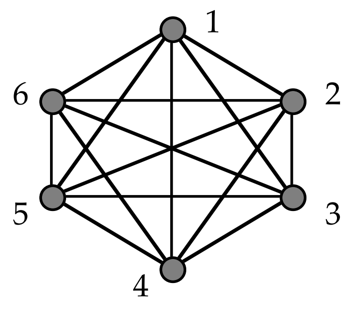


Figure 7.12: a complete graph on 6 vertices

2. How many edges does a complete graph on n vertices have?

7.7 Hamiltonian circuit problem (HCP)

Definition 7.14 A **Hamiltonian circuit** of a graph is a cycle containing all the nodes of the graph. See figure 7.13.

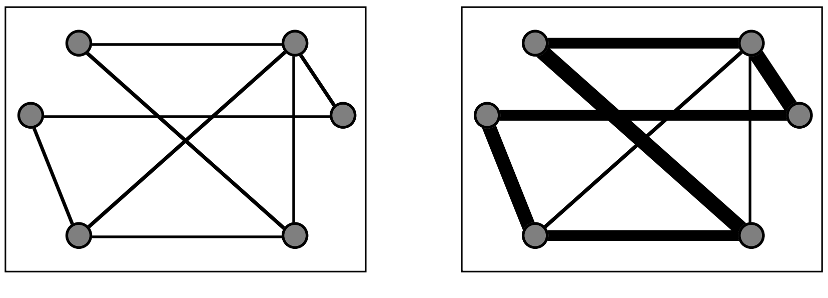


Figure 7.13: a graph (left) with a Hamiltonian circuit (right)

Definition 7.15 The **Hamiltonian circuit problem** (HCP) asks: does a given graph have a Hamiltonian circuit?

Warning HCP is, it seems, much harder than the previous problems. Our search algorithm is no use here: we want a 'spanning cycle', not a spanning tree. An algorithm could check every possible ordered list of the nodes in turn, stopping if one of them is a Hamiltonian circuit. If the graph has n nodes, there are essentially at most $(n-1)!/2$ such lists: $n!$ ways of ordering the nodes, but we don't care which is the start node (so divide by n), or which way we go round (so divide by 2). Whether a given combination is a Hamiltonian circuit can be checked quickly, so the $(n-1)!$ part dominates the time function of this algorithm. But $(n-1)!$ is not $O(n^k)$ for any number k. It is not even $O(2^n)$ (exponential time).

There is no known **polynomial time solution** to this problem: one with time function $O(n^k)$ for some k. We will look at it again later, as it is one of the important class of NP-complete problems.

Exercise 7.16 (Puzzle) Consider the squares on a chess-board as the nodes of a graph, and let two squares (nodes) be connected by an edge iff a knight can move from one square to the other in one move. Find a Hamiltonian circuit for this graph.

7.8 Summary of section

We examined some examples of graphs, and wrote a general purpose graph searching algorithm, which chooses the next node to examine according to its priority. Different priorities gave us depth-first and breadth-first search. We saw that it traced out a spanning tree of each connected component of the graph. We can use it to count or find the connected components, or to check for cycles. It runs in time $O((n+e)\log n)$ at worst (on a graph with n nodes and e edges). We defined a complete graph, and briefly looked at the (hard) Hamiltonian circuit problem.

8. Weighted graphs

Now we'll consider the more exotic but still useful **weighted graph.** We'll examine some weighted graph problems and algorithms to solve them. Sedgewick's book has more details.

8.1 Example of weighted graph

Imagine the nodes A–E in figure 8.1 are towns. An oil company wants to build a network of pipes to supply all the towns from a refinery at A. The numbers on the edges represent the cost of building an oil pipeline from one town to another: e.g., from A to D it's £5 million. The problem is to find the cheapest network.

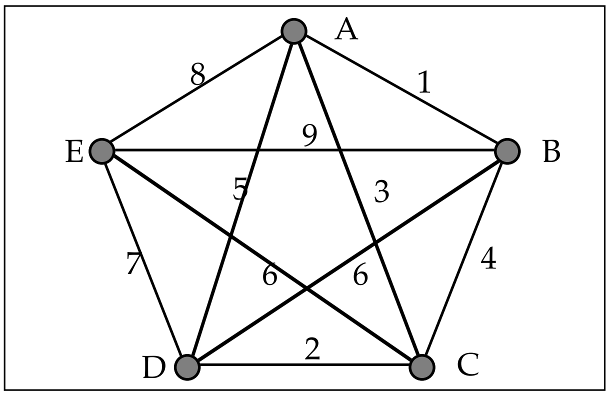


Figure 8.1: a weighted graph

8.1.1 What is a weighted graph?

We can represent the map above by a weighted graph. The nodes are the towns A–E, all edges are present, and the weight on each edge is the cost of building a pipe between the towns it connects.

Definition 8.1 Formally, a **weighted graph** is a triple (V,E,w), where:

- (V,E) is a graph
- $w : E \to \{1,2,\dots\}$ is a map providing a number (the weight) for each vertex pair ('edge').

We require that $w(x,y) = w(y,x)$ for all $(x,y) \in E$ (so that each edge gets a well-defined weight). We'll usually assume that weighted graphs (V,E,w) are *connected* (this means that (V,E) is connected).

So a weighted graph is just a graph with a number attached to each edge. The numbers might be distances, travel times or costs, electrical resistances, etc. Often, depending on the problem, (V,E) will be a *complete* graph, as we can easily represent a 'non-edge' by a very large (or small) weight. We can also use fractional or real-number weights if we want. Figure 8.1 represents a complete weighted graph, as all edges are present.

8.2 Minimal spanning trees

A proposed network of pipes can be represented by a *graph* (V,P). V is the set of towns as above, and P is the set of proposed pipelines. We put an edge (x,y) in P iff

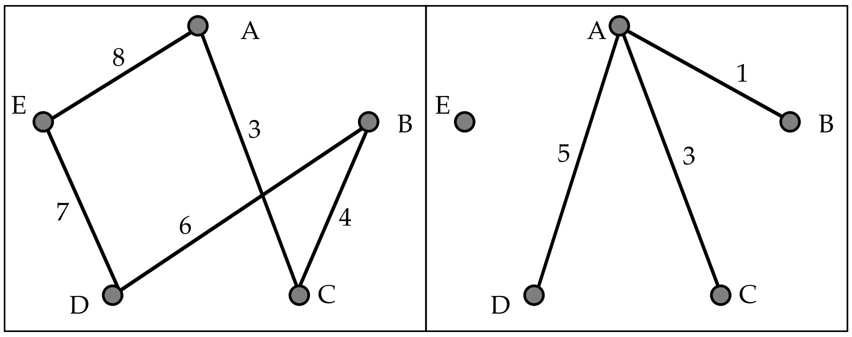


Figure 8.2: two possible pipelines

a pipe is to be built directly between x and y. Two possible pipe networks are given in figure 8.2.

Clearly, the cheapest network will have only one pipeline route from any town to any other. For if there were two different ways of getting oil from A to B (e.g., via C or via E and D, as on the left of figure 8.2), it would be cheaper to cut out one of the pipes (say the expensive one from A to E). The remaining network would still link all the towns, but would be cheaper. In general, if there is a *cycle* in the proposed network, we can remove an edge from it and get a cheaper network that still links up all the towns. So:

- the pipes the company builds should form a *tree.*

The right hand pipeline network in figure 8.2 does not connect all the towns. As every town should lie on the network,

- the tree should be a *spanning tree* (of the complete graph with vertices A–E).
- And its total cost should be least possible.

Definition 8.2 A **minimal spanning tree** (MST) of a (connected) weighted graph (V,E,w) is a graph (V,P) such that:

1. (V,P) is connected
2. $P \subseteq E$
3. the sum of the weights of the edges in P is as small as possible, subject to the two previous constraints.

A minimal spanning tree (V,P) will be a *tree,* for (as above) we could delete an edge from any cycle, leaving edges still connecting all the nodes but with smaller total weight. Because (V,P) must be connected, it must be a *spanning tree* of (V,E).

A MST will give the oil company a cheapest network. There may be more than one such tree: e.g., if all weights in (V,E,w) are equal, any spanning tree of (V,E) will do. Though one might find a MST in the graph of figure 8.1 by inspection, this will be harder if there are 100 towns, say. We need an algorithm to find a MST.

8.3 Prim's algorithm to find a MST

Our search algorithm gave a *spanning tree;* can we modify it to give a *minimal* one in a weighted graph? Let's try the following: when we push a node (town) onto the fringe, its priority will be the length of the edge joining it to the current node. A short length will mean high priority for popping, a long one low priority. See below for an example of this algorithm in action.

8.3.1 Proving correctness of Prim's algorithm

This idea seems intuitively correct. The graph will be explored using the shortest edges first, so the spanning tree produced *has a good chance* of being minimal. But how can we be *sure* that it *always* delivers a MST in *any* weighted graph? After all, the algorithm operates 'locally', working out from a start node; whereas a MST is defined 'globally', as a spanning tree of least weight. Maybe the best edges to use are at one side of the graph, but if we start the algorithm at the other side, it'll only find them at the end, when it's too late. (See §8.5.1 below for an apparently similar case, where these difficulties seem fatal.)

So we should *prove* its correctness. This is not so hard if we know the following property of MSTs. (If we don't, it can be seriously nasty!)

8.3.1.1 Useful 'separation' property

Let (V,E,w) be any connected weighted graph, and $T = (V,P)$ be any spanning tree of it. We say that T has the **'separation property'** if:

$(*)$ Given *any* division of the nodes in V into two sets, T contains *one of the shortest* (lowest weight) edges connecting a node in one set to a node in the other.

That is, there is no edge in E between the two sets that is shorter than every edge in T between them.

Example 8.3 Figure 8.3 shows a weighted graph, the weight $w(x,y)$ being the distance between the nodes x and y as measured in the diagram. The bold lines form a spanning

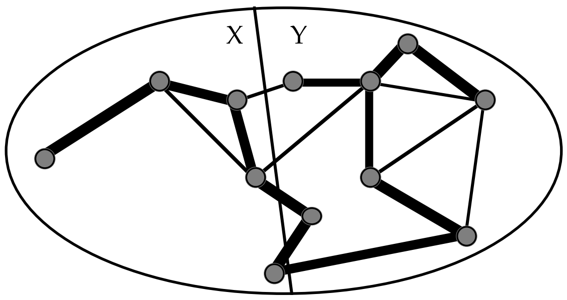


Figure 8.3: does this tree have the separation property? (weight $\approx$ distance)

tree; the light lines are the graph edges not in the tree. We've chosen an arbitrary division of the nodes into sets X, Y. If the spanning tree has the separation property, no graph edge from X to Y should be shorter than the three heavy tree edges crossing the $X-Y$ division.

Warning — what the separation property is not. There might be *more than one* shortest edge from X to Y. (They'll all be of equal length, of course. For example, this happens if all graph edges have the same length!) The separation property says that *at least one of them* is in the spanning tree.

The separation property is talking about shortest $X-Y$ *edges,* not shortest paths. *It is false* in general that the shortest *path* between any node of X and any node of Y is the path through the tree. Look for yourself. The top two nodes in figure 8.4 below are connected by an edge of length 12. But the path between them in the tree shown has length $4+7+6+5=22$ — and the tree does in fact have the separation property.

8.3.1.2 Separation property for MSTs?

What's all this got to do with Prim's algorithm? Well, we will show that *any MST has the separation property.* Let's see an example first.

Example 8.4 The heavy edges in figure 8.4 form an MST (you can check this later). On the left, X is the set of nodes in the circle, and Y is the rest. There are two least $X-Y$

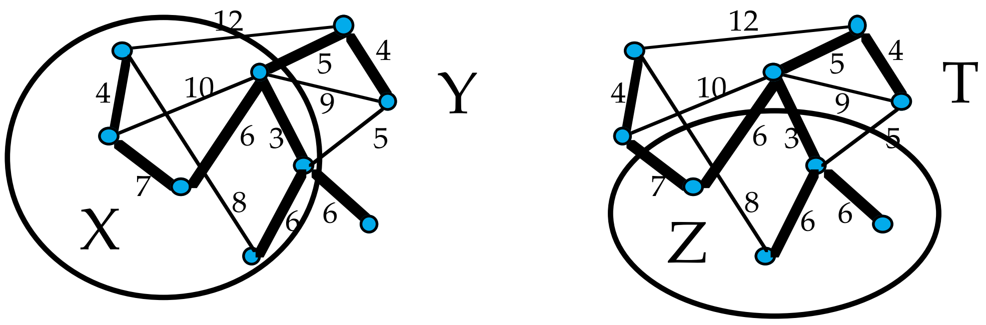


Figure 8.4: the MST has one of the least weight X–Y (and Z–T) edges

edges (of weight 5), and one of them is indeed in the MST shown, as the separation property says. On the right, I used a different division, $Z-T$, of the same weighted graph. The shortest $Z-T$ edge is of length 3 — and again, it's in the MST.

So the separation property might just hold for this MST, if we checked all sets X, Y. But in general? In fact, any MST has the separation property. But we can't establish this by checking all possible MSTs of all weighted graphs — there are infinitely many of them, and we wouldn't have the time. We will have to prove it.

Theorem 8.5 *Any MST has the separation property.*

PROOF. We will show that any spanning tree that does not have the separation property is not an MST.

Suppose then that:

- T is a spanning tree of the weighted graph (V,E,w).
- there's a division of V into two sets X,Y
- there's an edge $e = (x,y) \in X \times Y$ that's shorter than any edge of T connecting X and Y.

(For an example, see the spanning tree shown in figure 8.3.) We'll show that T is not a MST.

As T is a spanning tree, there's a unique path in T connecting x to y (the dotted line in figure 8.5). This path must cross over from X to Y by some edge $e' = (x',y') \in E$. (We let e' be any cross-over edge if there's more than one.)

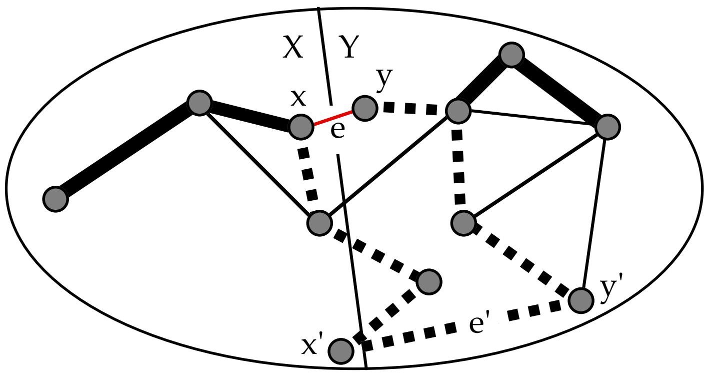


Figure 8.5: a short edge e from X to Y

Let's replace e' by e in T. We get $T^* = (V, (P \cup \{e\}) \setminus \{e'\})$ (see figure 8.6). Then:

- As e is shorter than e', T^* has smaller total weight than T.
- T^* has *no cycles.* Although adding e to T produces a unique cycle, taking e' out destroys it again.
- T^* is *connected.* For if $z,t \in V$ are different nodes, there was a path from z to t in T. If this path didn't use the edge e', it's still a path in T^*. If it did use e', then the path needed to get from x' to y'. But we can get from x' to y' in T^*, by going via e. So we can still get from z to t in T^*.

So T^* is a spanning tree. But T^* has smaller total weight than T. So T was not a MST. The separation property is proved. QED.

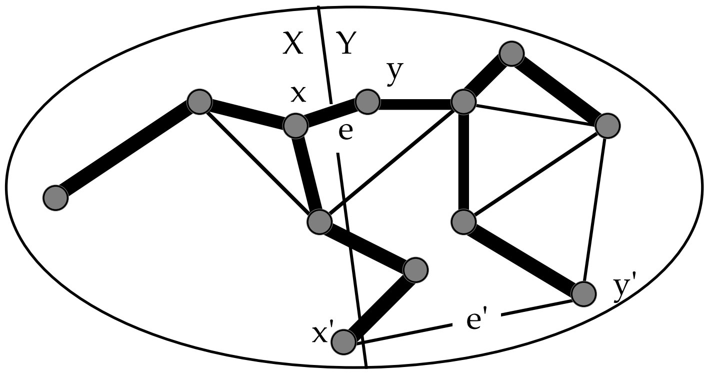


Figure 8.6: new spanning tree T^* (e' replaced by e)

8.3.1.3 Proof of correctness of Prim's algorithm

We're not finished yet. We showed any MST has the separation property; but we still have to show our algorithm builds a MST.

Theorem 8.6 *Prim's algorithm always finds a MST.*

PROOF. Assume for simplicity that all graph edges have different weights (lengths). (The algorithm finds a MST even if they don't: proving this is a tutorial exercise.) Let T be any MST. At each stage, our proposed algorithm adds to its half-built tree X the shortest possible edge connecting the nodes of X with the remaining nodes Y. (As all edges have different weights, there is exactly one such edge.) By the 'separation' property (theorem 8.5), the MST also includes this edge. So *every edge of the tree built by the algorithm is in* T.

Example 8.7 Figure 8.7 shows Prim's algorithm half way through building a MST for the graph in figure 8.4.

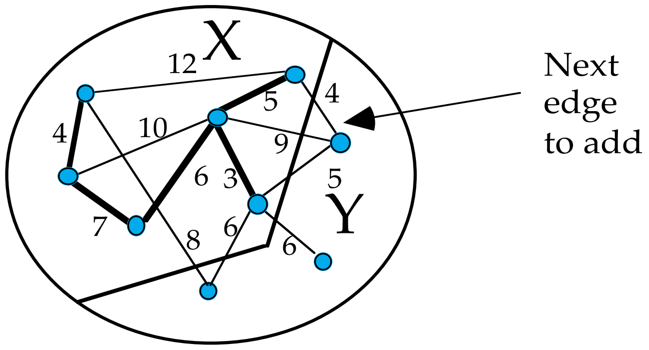


Figure 8.7: Prim's algorithm in progress

X is the half-built tree — the nodes already visited. Y is the rest. In the next step, the algorithm will add the edge shown, as it has highest priority on the fringe at the moment (check this!) But by the separation property, this edge is *the* shortest X–Y edge. So it is also in any MST — e.g., it's in the one shown in figure 8.4.

But all spanning trees have the same number of edges ($n-1$, where the whole graph has n nodes; see proposition 7.10). We know the algorithm always builds a spanning tree — so it chooses $n-1$ edges. But T is a MST, so also has $n-1$ edges. Since the algorithm only chooses edges in the MST T, and it chooses the same number of edges ($n-1$) as T has, it follows that the tree built by the algorithm *is* T. So Prim's algorithm does indeed produce a MST. This is true even with fractional or real-number weights. QED.

Exercises 8.8 (challenge!)

1. Deduce that if all edges in a weighted graph have different weights, then it has a *unique* MST. *Must* this still be true if some edges have equal weight? *Can* it still be true?
2. Is it true that any spanning tree (of a weighted graph) that has the separation property is a MST of that graph? (This, 'separation property $\Rightarrow$ MST', is the converse of theorem 8.5.)
3. Here's a proposed algorithm to find a MST of a connected weighted graph G.

```
1 Start with any spanning tree $T$ of $G$.
2 Pick any $X$–$Y$ division of the nodes of $G$.
3 If $T$ doesn't have a shortest $X$–$Y$ edge, replace an $X$–$Y$ edge of $T$
  with a shorter one [as in the proof in §8.3.1.3, especially figures 8.5
  and 8.6].
4 Repeat steps 2–3 until $T$ doesn't change any more, whichever $X$, $Y$
  are picked.
```

   (a) Does this terminate?
   (b) If it does, is the resulting tree T a MST of G?
   (c) If so, would you recommend this algorithm? Why?

Warning If we run Prim's algorithm on the graph in figure 8.1, starting from node A, we get a MST — we just proved this. If we start it from node D, we also get a MST — we proved that the algorithm *always* gives a MST. So wherever we start it from, it delivers a MST. Of course, we may not always get the same one. But if all edges had different weights, we would get the same MST wherever we started it from (by exercise 8.8(1) above).

So to get an MST, there is no need to run the algorithm from each node in turn, and take the smallest tree found. It gives an MST wherever we start it from.

Try the algorithm on figure 8.1, starting from each node in turn. What is the total weight of the tree found in each case? (They should all be the same!) Do you get the same tree?

8.3.2 Implementation and execution of Prim's algorithm

We can use the 'visit' algorithm of §7.3.5. When we push (x,y) onto the fringe (priority queue) we give it priority $w(x,y)$, where *low weight = high priority for popping.* (We can write this as 'push $(x,y,w(x,y))$ onto fringe'.) E.g., if edge (A,C) has weight 4 we push (A,C,4) onto the fringe. When we pop an edge (x,y) we pop one with *highest* priority — i.e., *lowest* weight.

A run of MST(A) for the weighted graph in figure 8.1 looks like this. First, push $(A,*,0)$ into queue. The run is then as shown in the table. The MST we get is 'AB, AC, CD, CE', of total length 12:

fringe	pop	visited	print	push	comments
$(A,*,0)$	$(A,*,0)$	A		$(B,A,1)$ $(C,A,3)$ $(D,A,5)$ $(E,A,8)$	
$(B,A,1)$ $(C,A,3)$ $(D,A,5)$ $(E,A,8)$	$(B,A,1)$	B	edge 'A,B'	$(C,B,4)$ $(D,B,6)$ $(E,B,9)$	C,D, and E are already in the fringe with better priority, so the pushes have no effect.
$(C,A,3)$ $(D,A,5)$ $(E,A,8)$	$(C,A,3)$	C	edge 'A,C'	$(D,C,2)$ $(E,C,6)$	Both pushes have better priority than the current fringe entries, which are replaced.
$(D,C,2)$ $(E,C,6)$	$(D,C,2)$	D	edge 'C,D'	$(E,D,7)$	This push has lower priority than current entry $(E,C,6)$ for E, so no dice.
$(E,C,6)$	$(E,C,6)$	E	edge 'C,E'	–	
empty					Terminate.

Figure 8.8 shows the MST found in this run.

8.3.3 Run time of Prim's algorithm

On graphs with few edges, the algorithm runs in time $O((n+e)\log n)$, where there are n nodes and e edges. Cf. §7.3.8.

Exercise 8.9 (Kruskal's algorithm for MST) Another algorithm to find an MST, due to Kruskal, runs in $O(e \log e)$. This exercise is to check that it works. Let (V,E,w) be a connected weighted graph with n nodes and e edges. The algorithm works as follows:

```
1   Sort the edges in $E$. Let the result be $E = \{s_1, \ldots, s_e\}$ in order of weight,
    so that $w(s_1) \leq w(s_2) \leq \cdots \leq w(s_e)$.
```

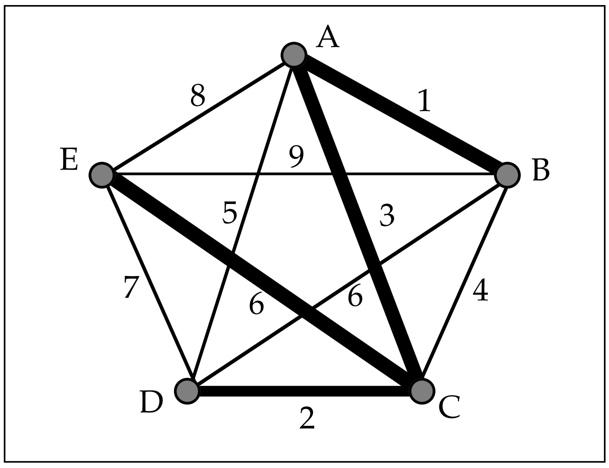


Figure 8.8: the MST found from figure 8.1

```
2    set $T := \emptyset$; set $i := 1$
3    repeat until $T$ contains $n-1$ edges
4        if the graph $(V, T \cup \{s_i\})$ has no cycles then set $T := T \cup \{s_i\}$
5        add 1 to $i$
6    end repeat
7    output $T$
```

Let T be the output. So $T \subseteq E$.

1. Show that (V,T) is a spanning tree of (V,E).
2. Show that (V,T) has the separation property.
3. Deduce that (V,T) is a MST of (V,E,w). (It may help to simplify by assuming all edges of E have different weights, but try to eliminate this assumption later.)
4. Show that, using a suitable sorting algorithm (suggest one), Kruskal's algorithm runs in time $O(e \log e)$.

8.4 Shortest path

Suppose in a weighted graph we want to find the path of least possible length from node x to node y. We use the algorithm to build a spanning tree, starting at x. For each node z added to the tree, we keep a tally of its distance $d(z)$ from x through the tree as built so far, and add to the fringe all neighbours t of z with priority $d(z) + w(z,t)$. We stop when y gets into the tree. The shortest path from x to y is then the unique path from x to y through the tree. This algorithm is essentially due to Dijkstra (a big cheese). Exercise: try it on an example. And prove it correct!!

8.5 Travelling salesman problem (TSP)

Example: Consider figure 8.1 again. Suppose the numbers on the edges represent road distances between the towns.[1] A salesperson lives in A and wants to make a round trip, visiting each city just once and returning home at the end. The manager conjures a figure, d, out of the air. If the whole trip is more than d miles, no expenses can be claimed.

Problem: Is there a route of length $\leq d$?

We can formalise this problem using weighted graphs.

Problem (TSP): Given a complete weighted graph (V,E,w) (that is, (V,E) is a complete graph), and a number d, is there a Hamiltonian circuit of (V,E) of total length at most d?[2]

As the graph is complete, there will be many Hamiltonian circuits, but they may all be longer than d.

This is not a toy problem: variants arise in network design, integrated circuit design, robotics, etc. See Harel's book, p.153. TSP is another hard problem. The exhaustive search algorithm for HCP also works for TSP. There are $(n-1)!/2$ possible routes to consider (see page 111). For each route, we find its length (this can be done in time $O(n)$) and compare it with d. As for HCP, this algorithm runs even slower than exponential time. There is no known polynomial time solution to TSP. Some heuristics and sub-optimal solutions in special cases are known.

8.5.1 Nearest neighbour heuristic for TSP

One might hope that the following algorithm would find the shortest Hamiltonian circuit in any weighted graph (V,E,w):

```
1   start by letting $current\_node$ be any node of $V$
2   repeat until all nodes have been visited
3       go to the nearest node to $current\_node$
              % [the node $x$ such that $w(x, current\_node)$ is least]
4   end repeat
```

This is called the **nearest neighbour heuristic.** It works locally, choosing the nearest neighbour to the current node every time. It's the nearest algorithm to Prim's algorithm for finding a MST (§8.3), and it is similarly fast. But while Prim's algorithm

[1]Important: roads between the towns may not be straight! I.e., there may be three towns, x,y, and z, with $w(x,z) > w(x,y) + w(y,z)$.

[2]This is the 'yes-no' version of TSP. The 'original' version is 'given a complete weighted graph (V,E,w), find a Hamiltonian circuit of minimal length'.

is correct, actually delivering a MST, the performance of the nearest neighbour heuristic is absolutely diabolical in many cases — it's one of the worst TSP heuristics of all. The energetic will find seriously incriminating evidence in Rayward-Smith's book; the rest of us may just try the heuristic on the graph shown in figure 8.9.

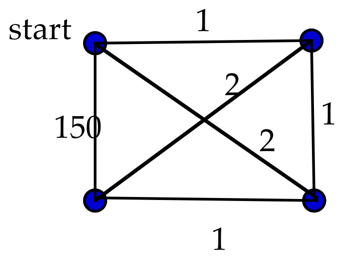


Figure 8.9: a bad case for nearest neighbour

One might easily think that the nearest neighbour heuristic was 'intuitively a correct solution' to TSP. It takes the best edge at each step, yes? But in fact, it is far from being correct. Intuition is surely very valuable. Here we have an *awful warning* against relying on it uncritically. Nonetheless, nearest neighbour is used as an initial stage in some more effective heuristics.

Exercise 8.10 Suppose we had a polynomial time algorithm that solved TSP. Show how it could be used to solve the 'original' version of TSP mentioned in footnote 2. (Creativity is called for.)

8.6 Polynomial time reduction

Though similar to TSP, HCP seems rather easier. We can *formalise* this using the **reduction** of section 5, with the new feature that now we want the reduction to be *fast.* Suppose we have a fast method F of transforming a graph G into a complete weighted graph G^* plus a number d, so that G has a Hamiltonian circuit iff G^* has a round trip of length $\leq d$. That is:

- G is an instance of HCP,
- $F(G) = \langle G^*, d\rangle$ is an instance of TSP,
- the answers (yes or no) are the same for G as for $\langle G^*, d\rangle$.

Then any fast method M of solving TSP could also be used to solve HCP quickly. For, given an instance G of HCP, we transform it quickly into $F(G)$ and apply M, which is also fast. Whatever answer M gives to $F(G)$ (yes or no) will also be the correct answer to G. By **fast** we mean **takes polynomial time (see §7.7) in the worst case.** This technique is called **polynomial time (p-time) reduction.** See Part III.

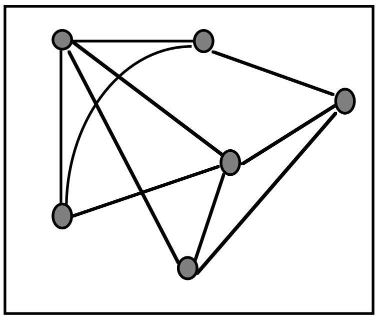


Figure 8.10: an instance G of HCP ... but is it a yes-instance?

Example 8.11 (p-time reduction of HCP to TSP) Suppose that we have an instance of HCP: a graph such as the one shown in figure 8.10.

We can turn it into an instance of TSP by:

- defining the distance between nodes x and y by

$$d(x,y) = \begin{cases} 1, & \text{if } x \text{ is joined to } y \text{ in the graph,} \\ 2, & \text{otherwise} \end{cases}$$

- defining the bound 'd' to be the number of nodes.

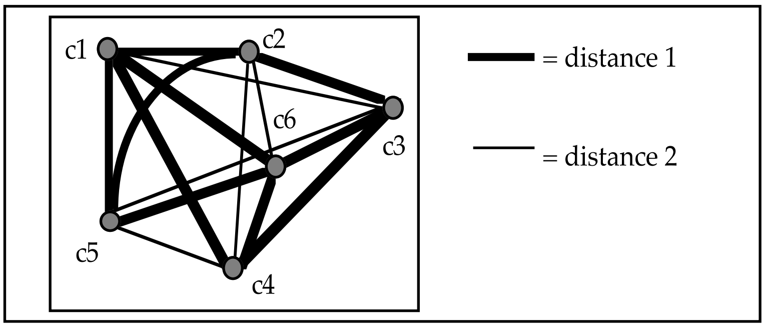


Figure 8.11: the instance $F(G)$ of TSP; $d = 6$

We get figure 8.11. This conversion takes time about n^2 if there are n nodes, so is p-time. Then

- any Hamiltonian circuit in the original graph yields a round trip of length n in the weighted graph.
- Conversely, any round trip in the weighted graph must obviously contain n edges; if it is of length $\leq n$ then all its edges must have length 1. So they must be real edges of the original graph.

So the original graph has a Hamiltonian circuit iff there's a route of length $\leq n$ in the corresponding weighted graph. E.g., in figure 8.11, the route $(c_1, c_2, c_3, c_4, c_6, c_5, c_1)$ has length 6.

8.7 NP-completeness taster

So HCP is 'no harder' than TSP. In fact they are about the same difficulty: one can also reduce TSP to HCP in p-time, though this is more tricky. Both TSP and HCP are examples of **NP-complete problems.** Around 1,000 problems are now known to be NP-complete, and they all reduce to each other in p-time. In practice, NP-complete problems are **intractable:** currently, even moderately large instances of them can't be handled in a reasonable time, and most people believe that no fast solution exists. We'll examine NP-completeness in Part III.

8.8 Summary of section

We discussed weighted graphs and applications. Using a 'short edge = high priority' fringe popping strategy, we found an algorithm for finding a minimal spanning tree (MST) in a weighted graph, and proved it correct. There's a unique MST if all edges have different weights. We gave an algorithm to find the shortest path between two nodes of a weighted graph. We mentioned the (hard) travelling salesman problem, and showed that any fast solution to it would provide a fast solution to the Hamiltonian circuit problem (§7.7). No polynomial time solution to either of these is known.

8.9 Part II in a nutshell

Section 6: When choosing an algorithm it helps to know roughly the time $f(n)$ that it'll take to run on an input of a given size n. As there are many inputs of a given size n, we usually consider the worst or the average case. Worst case run time estimates are easier to find. Usually a rough estimate will do: we get $f(n) = c \cdot g(n) +$ smaller terms. The uncertainties in the estimate may be important and should be borne in mind.

g above can often be calculated by a recurrence relation. Because many algorithms use one of a few standard techniques, g often has one of the following forms: constant, $\log n; n; n \log n; n^2$; or 2^n, and we say the algorithm runs in **constant time, log time,** etc.

Implementation should use careful experiments and may involve optimising the code. *Keep it simple* is a sound rule.

O notation is useful for comparing growth rates of functions. For functions f, g, if $f(n) \leq c \cdot g(n)$ for all large enough n then we say that f is $O(g)$. If f is $O(g)$ and g is $O(f)$, we say that f is $\theta(g)$.

Section 7: A **graph** is a collection of vertices or **nodes,** some of which are connected by **edges.** Many problems can be represented as problems about graphs. A common graph searching algorithm proceeds from a start node through the graph along edges to other nodes. At each point, the immediate neighbours of the current node are added to the 'fringe' of vertices to visit next. Which fringe vertex is actually picked for visiting depends on its **priority,** which can be assigned in any way. E.g., giving top priority to the most recent fringe entrant (stack), or the oldest (queue), leads to **depth-first** and **breadth-first search,** respectively.

Each call visits an entire **connected component** of the graph: those nodes accessible from the start node by going along edges. The algorithm traces out a **tree** (a **connected** graph with no **cycles**) made of graph edges and including every vertex (a **spanning tree**). If the graph is not connected (has > 1 connected component), the algorithm will have to be called more than once. So we can use it to count connected components. If it ever examines a node that was visited earlier (not counting the immediately previous node), the graph has a **cycle.** On a graph with n nodes and e edges, the algorithm runs in worst case time $O((n+e)\log n)$.

We saw that running the algorithm on a tree gives the whole tree, which therefore has 1 less edge than the number of nodes.

A **complete graph** is one with all possible edges. A **Hamiltonian circuit** in a graph is a cycle visiting all nodes. The problem of whether a given graph has such a circuit — the **Hamiltonian circuit problem (HCP)** — is hard. Exhaustive search can be used; there is no known polynomial time algorithm (i.e., one running in time $O(n^k)$ in the worst case, for some k) to detect whether a graph has such a cycle.

Section 8: In a **weighted graph** we attach a positive whole number (a weight, or length) to each edge. A common problem is to find a spanning tree of least possible total weight (a minimal spanning tree, or MST). We showed that the algorithm above will produce a MST if at each stage we always choose the fringe node closest to the visited nodes. We can use a similar method to find the shortest path between two nodes.

Given a weighted graph and a bound d, the **travelling salesman problem (TSP)** asks if there's a Hamiltonian circuit in the graph of total weight $\leq d$. This problem has many applications, but is hard. The position is similar to HCP.

We can **reduce** HCP to TSP rapidly (with worst case time function of the order of a polynomial). Any putative fast solution to TSP could then be used to give a fast solution to HCP. In fact one can also reduce TSP to HCP in p-time, so HCP and TSP are about equally hard. They are 'NP-complete' (see Part III). Currently they are intractable, and most people expect them to remain so.

Part III

Complexity

In Part I of the course we saw that some problems are algorithmically unsolvable. Examples:

- the halting problem (will a given TM halt on a given input?)
- deciding the truth of an arbitrary statement about arithmetic.

But there are wide variations in difficulty even amongst the *solvable* problems. In practice it's no use to know that a problem is solvable, if all solutions take an inordinately long time to run. So we need to refine our view of the solvable problems. In Part III we will classify them according to difficulty: how long they take to solve. *Note: the problems in Part III are solvable by an algorithm; but they may not be solvable in a reasonable time.*

Earlier, we formalised the notion of a **solvable problem** as one that can be solved by a Turing machine (Church's thesis). We did this to be able to reason about algorithms in general. We will now formalise the **complexity** of a problem, in terms of Turing machines, so that we can reason in general about the varying difficulty of problems.

We will classify problems into four levels of difficulty or complexity. (There are many finer divisions).

1. The class P of tractable problems that can be solved efficiently (in polynomial time: p-time).

2. The intractable problems. Even though these are algorithmically solvable, any algorithmic solution will run in exponential time (or slower) in the worst case. Such problems cannot be solved in a reasonable time, even for quite small inputs, and for practical purposes they are unsolvable for most inputs, unless the algorithm's average case performance is good. The exponential function dwarfs technological changes (figure 6.1), so hardware improvements will not help much (though quantum computers might).

3. The class NP of problems. These form a kind of half-way house between the tractable and intractable problems. They can be solved in p-time, but by a **non-deterministic algorithm.** Could they have p-time **deterministic** solutions? This is the famous question 'P = NP?' — is every NP-problem a P-problem? The answer is *thought* to be **no,** though no-one has *proved* it. So these problems are currently believed to be intractable, but haven't been proved so.

4. The class NPC of NP-complete problems. In a precise sense, these are the hardest problems in NP. Cook's theorem (section 12) shows that NP-complete problems exist (e.g., 'PSAT'); examples include the Hamiltonian circuit and travelling salesman problems we saw in sections 7–8, and around 1,000 others (so far). All NP-complete problems reduce to each other in polynomial time (see §8.6). So a fast solution to any NP-complete problem would immediately give fast solutions to all the others — in fact to all NP problems. This is one reason why most people believe NP-complete problems have no fast deterministic solution.

Why study complexity? It is useful in practice. It guides us towards the tractable problems that are solvable with fast algorithms. Conversely, NP-complete problems occur frequently in applications. Complexity theory tells us that when we meet one, it might be wise not to seek a fast solution, as many have tried to do this without success.

On a more philosophical level, Church's thesis defined an algorithm to be a Turing machine. So two Turing machines that differ even slightly represent two different algorithms. But if each reduces quickly to the other, as all NP-complete problems do, we might wish to regard them as the *same* algorithm — *even if they solve quite different problems!* So the notion of fast reducibility of one problem or algorithm to another gives us a higher-level view of the notion of algorithm.

So in Part III we will:

1. define the run time function of a Turing machine,
2. introduce non-deterministic Turing machines and define their run time function also,
3. formalise fast reduction of one problem to another,
4. examine NP- and NP-complete problems.

9. Basic complexity theory

We begin by introducing the notions needed to distinguish between tractable and intractable problems. The classes NP and NPC will be discussed in sections 10 and 12.

9.1 Yes/no problems

We will only deal with 'yes/no problems', so that we can ignore the output of a Turing machine, only considering whether it halts & succeeds or halts & fails. This simplification will be especially helpful when we consider non-deterministic Turing machines (section 10).

Definition 9.1 A yes/no problem is one with answer yes or no. Each yes/no problem has a set of **instances** — the set of valid inputs for that problem. The yes-instances of a problem are those instances for which the answer is 'yes'. The others are the no-instances.

Many problems can be put in yes/no form:

Problem	instances	yes-instances	no-instances
primality	binary representations of numbers	binary representations of primes	binary representations of non-primes
Halting problem: Does M halt on input w?	all pairs $(code(M), w)$ where M is a standard TM, and w a word of C	all those pairs $(code(M), w)$ such that M halts & succeeds on w	the pairs $(code(M), w)$ such that M doesn't halt & succeed on w
HCP	all (finite) graphs	graphs with a Hamiltonian circuit	graphs with no Hamiltonian circuit
TSP	all pairs (G, d), where G is a weighted graph, and $d \geq 0$	all pairs (G, d) such that G has a Hamiltonian circuit of length $\leq d$	all pairs (G, d) such that G has no Hamiltonian circuit of length $\leq d$

9.1.1 Acceptance, rejection, solving

Even if we ignore its output, a Turing machine can still 'communicate' with us by halting & succeeding ('yes'), or halting & failing ('no'), so it can answer yes/no problems. Of course, we may have to code the instances of the problem as words, so they can be input to a Turing machine. For HP, $code(M)$ is given to the TM, as M itself is a machine, not a word. The coding of instances into words should be **reasonable:** we do not allow unary representation of numbers, and cheating (such as coding all yes-instances as 'yes' and all no-instances as 'no') is not allowed.

Definition 9.2

1. A Turing machine M is said to **accept** a word w of its input alphabet if M halts and succeeds on input w.

2. M is said to **reject** w if M halts and fails on input w.

3. A Turing machine M is said to **solve** a yes/no problem A if:

- every instance of A is a word of the input alphabet of M (or can be coded as one in a reasonable way);
- M accepts all the yes-instances of A;
- M rejects all the no-instances of A.

Example 9.3

1. In example 3.6 we saw a Turing machine that halts and succeeds if its input word is a palindrome, and halts & fails if not. This machine solves the yes/no problem '**is $w \in I^*$ a palindrome?**', where I is its input alphabet.
2. The universal Turing machine U does *not* solve the halting problem. If we give it $code(M) * w$ for some standard M and word $w \in C^*$, then U does halt & succeed on the yes-instances (see the table above). *But* it does *not* halt & fail on all the no-instances: if M runs forever on w, U never halts on $code(M) * w$.

9.1.1.1 Our problems must have infinitely many y- and n-instances

We do not consider yes/no problems with only finitely many yes-instances, or only finitely many no-instances. They are too easy! E.g., if the yes-instances of a yes/no problem X are just $y_1, \ldots, y_n$, a Turing machine M can solve X by checking to see if the input word w is one of the y_i. (The y_i are 'hard-wired' into M; we can do this as there are only finitely many of them.) If it is, M halts and succeeds; otherwise it halts and fails. No 'calculation' is involved. E.g., 'Is 31 prime?' has no instances at all, and is solved by the trivial Turing machine whose initial state is halting.

This may seem odd. For example, one of the Turing machines in figure 9.1 (both with input alphabet C, say) solves the yes/no problem:

'Is (a) Goldbach's conjecture true, and (b) $w = w$?'

The instances of this problem are all words w of C. The yes-instances are those w such that (a) Goldbach's conjecture is true, and (b) $w = w$. The no-instances are the rest. So if the conjecture is true, every $w \in C^*$ is a yes-instance, so Y solves it (Y halts & succeeds on any input). If not, every $w \in C^*$ is a no-instance, so N solves it (N halts & fails on any input). Of course, we don't know which! *But the problem is solvable* — either by Y or by N.

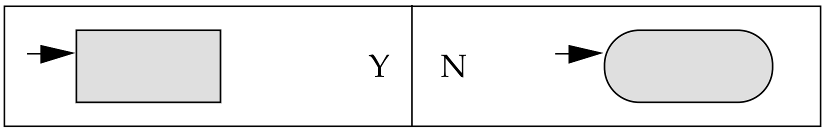


Figure 9.1: Which machine solves Goldbach's conjecture?

For a problem to be solvable according to our definition, we are only concerned that a Turing machine solution exists, not in how to find it. So Goldbach's conjecture really is too easy for us! We are only interested in problems with infinitely many yes- and infinitely many no-instances.

9.2 Polynomial time Turing machines

As for algorithms in section 6, we want to define how long a Turing machine takes to run. Of course, we have to bear in mind that a Turing machine can run for different numbers of steps on different words of any given length. Here, we consider the *worst case* only. The warnings in section 6 about doing this still apply: e.g., our Turing machine may usually be very quick, taking the maximum time on only a few inputs.

9.2.1 Run time function of Turing machines

Definition 9.4 Let $M = (Q, \Sigma, I, q_0, \delta, F)$ be a Turing machine. We write $\mathit{time}_M(n)$ for the the length of the *longest* run of M on any input of size n (we want the worst case). $\mathit{time}_M(n)$ will be ∞ if M does not halt on some input of length n. We call the function $\mathit{time}_M : \{0,1,2,\ldots\} \to \{0,1,2,\ldots,\infty\}$ the **run time function of** M**.**

9.2.2 p-time Turing machines

Definition 9.5 A Turing machine M is said to **run in polynomial time (p-time)** if there is some polynomial $p(n) = a_0 + a_1 n + a_2 n^2 + \ldots + a_k n^k$, where the coefficients $a_0, \ldots, a_k$ are non-negative whole numbers, such that:

$$\mathit{time}_M(n) \leq p(n) \qquad \text{for all } n = 0,1,2,\ldots.$$

That is, *no run of M on any word of length n lasts longer than $p(n)$ steps.*

Turing machines that run in p-time are considered to be fast. This is a broad but still useful categorisation — see the Cook–Karp thesis below.

Note We do not use the O-notation in the definition of M running in p-time (e.g., by saying '$\mathit{time}_M(n)$ is $O(n^k)$' for some k). We require that $\mathit{time}_M(n)$ should be at most $p(n)$, not just at most $c \cdot p(n)$ for some constant c. This is no restriction because $c \cdot p(n)$ is a polynomial anyway. But further, we require that $\mathit{time}_M(n) \leq p(n)$ *for all n, however small,* so that we are sure what happens for all n. We have to be a bit more careful than with the more liberal O-notation, but there are some benefits of this approach:

Proposition 9.6 *p-time Turing machines always halt.*

PROOF. If M is p-time then for some polynomial $p(n)$, M takes at most $p(n)$ steps to run on any word w of length n. But $p(n)$ is always finite, for any n. So M always halts (succeeding or failing) on any input; it can't run forever. QED.

The following exercise shows that insisting on a firm polynomial bound on run time *for all n* is not really a restriction.

Exercise 9.7 Let $f : I^* \to \Sigma^*$ be any partial function. Show that the following are equivalent:

1. $f = f_M$ for some Turing machine M running in p-time,
2. $f = f_{M^*}$ for some Turing machine M^* such that $time_{M^*}(n)$ is $O(n^k)$ for some k.

Hint: for '$\Uparrow$', tabulate all 'short' inputs for M^* as a look-up table.

9.3 Tractable problems — the class P

Our main interest is in when a problem is **tractable:** when it has a reasonably fast solution. Here we define 'tractability' formally, and look at some examples.

9.3.1 Cook–Karp thesis

Tractable problems can be solved in a reasonable time for instances of reasonable size. We would like to make this more precise.

The graph in figure 6.1 on page 90 showed that *polynomials* ($100n^2$, etc.,) grow at a manageable rate. The degree of the polynomial (k in definition 9.5) can usually be massaged down to 5 or better. As $\log n \leq n$ for all $n \geq 1$ (exercise: prove it!), even run time functions such as $n \log n$ are bounded by a polynomial (here, n^2). The problems we saw in sections 7–8 are all solvable in polynomial time, except (probably) the Hamiltonian circuit and travelling salesman problems, HCP and TSP. Al-Khwārazmi's algorithms for arithmetic are tractable, as are most sorting and other algorithms in common use.

On the other hand, problems that only have algorithms with *exponential* run time function (or worse) are effectively no better than unsolvable ones for even moderately large inputs, unless their average-case performance is better (here, we only consider worst-case).

So, rather as in Church's thesis, we will equate the *tractable problems* (a vague notion, since what is tractable depends on our technology and resources) with the *problems solvable in p-time* (a precise notion). Doing so is sometimes called the **Cook–Karp thesis,** after S. Cook (of whom more later), and R. Karp. The Cook–Karp thesis is useful, but a little crude; more people disagree with it than with Church's thesis. (For one thing, some people think average-case complexity is more important than worst-case in practice.)

9.3.2 P

Adopting the Cook–Karp thesis, we make the following important definition.

Definition 9.8

1. A yes/no problem is said to be **tractable** if it can be solved by a Turing machine running in p-time.
2. An algorithm is said to be tractable if it can be implemented by a Turing machine that runs in polynomial time.
3. We write P for the class of tractable yes/no problems: those that are solvable by a Turing machine running in polynomial time.

9.3.2.1 P is closed under complementation

The class P has some nice properties. We look at one now, to practice using P. We will see more nice properties of P in section 11.

Definition 9.9 The **complement** of a yes/no problem is got by exchanging the answers **yes** and **no.** What were the yes-instances now become the no-instances, and vice versa. E.g., the complement of 'is *n* prime?' is 'is *n* composite?'.

If $\mathcal{S}$ is a class of problems (e.g., P), we write **co-**$\mathcal{S}$ for the class consisting of the complements of the problems in $\mathcal{S}$. Clearly, $\mathcal{S} =$ co-co-$\mathcal{S}$.

Proposition 9.10 *P is closed under complementation.*

That is, if A is in P, the complementary problem to A is also in P. Or, co-P $\subseteq$ P.

PROOF. For if A is a yes/no problem solvable in p-time by a Turing machine M, we can rewire M so that (a) the halting states are no longer halting states, so that entering one now causes a halt and fail, and (b) whenever M is in a 'halt and fail' situation (no applicable instruction), control passes to a new state, which is halting. The rewired machine M' (see figure 9.2) also runs in p-time. As M' accepts exactly the words that M rejects, it solves the complementary problem to A, which is therefore in P. QED.

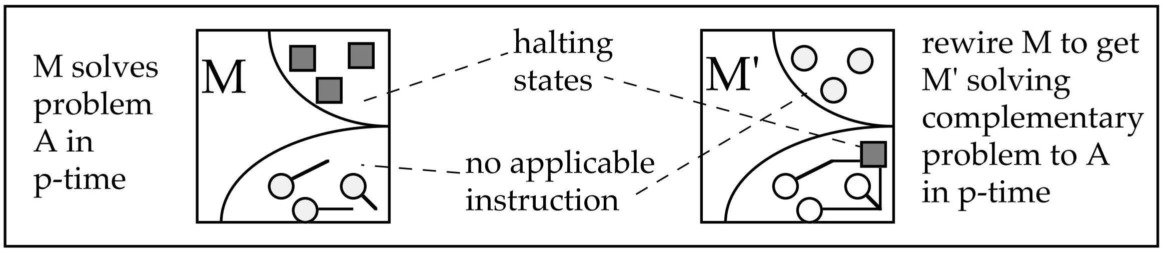

M solves problem A in p-time
M
halting states
no applicable instruction
M'
rewire M to get M' solving complementary problem to A in p-time

Figure 9.2: sketch of why P is closed under complementation

Exercise 9.11 The proof above doesn't explain in detail what happens if M tries to move left from square 0. How would you do this?

Show that co-P = P (i.e., not just '$\subseteq$').

9.3.2.2 Primality testing

It can be hard to tell if a problem is tractable. Here's one that was open for 2000 years:

Primality problem: 'given a whole number x, is it prime?' Non-primality (compositeness) problem: 'given a whole number x, is it composite (i.e., not prime)?'

One solution is to check all possible factors y of x (all integers y with $2 \leq y \leq \sqrt{x}$). If x is given in binary, then an input of length n could represent an x of up to about 2^n, which has $\sqrt{2^n} = (\sqrt{2})^n$ possible factors. This approach will therefore take exponential time.

Until quite recently, no p-time algorithm for primality or for non-primality testing was known, though people were hopeful that one would be found, and some superb probabilistic algorithms had been devised. But in 2002, Agrawal, Kayal, and Saxena, from Kampur, India, published an ingenious (deterministic) polynomial-time algorithm that determines whether an input number n is prime or composite. (They instantly became world-famous). So primality testing is now known to be tractable.

A great deal of work has been done in this area, but as it is a crucial field for cryptography (see Harel's book), some of the work is probably not published.

9.4 Intractable problems?

Definition 9.12 An algorithm is said to be **intractable** if it can't be implemented by a p-time Turing machine. A yes/no problem is said to be **intractable** if (i) it is algorithmically solvable, but (ii) it is not tractable. *All* solutions *necessarily* use intractable algorithms.

Some problems are known to be intractable. There are many examples from logic: one is deciding validity of sentences of first-order logic written with only two variables (possibly re-used: like $\exists x \exists y(x < y \wedge \exists x(y < x))$). This problem is solvable, but all algorithms must take at least exponential time.

But many common problems have not been proved to be either tractable or intractable! Typical examples are HCP and TSP. All known *algorithms* to solve these problems are intractable, but it is not known if the *problems* are themselves intractable. Maybe there's a fast algorithm that everyone's missed. For reasons to be seen in section 12, this is not thought likely.

Besides TSP and HCP, problems in this category include:

Propositional satisfaction (PSAT) Here we consider formulas of **propositional logic.** They are written using an alphabet I with atoms, $p_1, p_2, p_3, \ldots,$[1] connectives $\wedge$ (and), $\vee$ (or), $\neg$ (not), $\rightarrow$ (implies) and $\leftrightarrow$ (iff), and brackets), (. This is as for arithmetic (§5.4), but there are no quantifiers this time. Any formula is a word of I, and can be given as input to a Turing machine.

We can assign varying truth values (true or false) to the atoms. We write h for a particular assignment: so e.g., $h(p_1)$ = true, $h(p_2)$ = false is possible. Then we can

[1]As in section 5, p_{46} is 3 symbols, p, 4, and 6. So all formulas can be written with a finite alphabet I including p and the numbers 0–9 say, plus $\wedge, \neg$, etc.

recursively work out the truth value of any formula A:

$$\begin{array}{rll} h(\neg A) = \text{true} & \text{iff} & h(A) = \text{false} \\ h(A \wedge B) = \text{true} & \text{iff} & h(A) = h(B) = \text{true} \\ h(A \vee B) = \text{true} & \text{iff} & \text{at least one of } h(A), h(B) \text{ are true} \\ h(A \to B) = \text{true} & \text{iff} & \text{either } h(A) \text{ is false or } h(B) \text{ is true or both} \\ h(A \leftrightarrow B) = \text{true} & \text{iff} & h(A) = h(B). \end{array}$$

So for example, if h makes $p = p_1$ true and $q = p_2$ false, then:

- $h(p \to q)$ = false,
- $h(((p \to q) \to p) \to p)$ = true,
- $h((p \wedge q) \vee (\neg p \wedge \neg q)) = h(p \leftrightarrow q)$ = false.

Problem (PSAT, propositional satisfiability) Given a formula A, is there some assignment h to the atoms of A such that $h(A)$ = true? That is: is A **satisfiable**?

A Turing machine could check whether A is satisfiable by checking every valuation for the atoms in A — i.e., searching the 'truth table' of A. We want to find out how long this might take. For each atom p, there are 2 possible values of $h(p)$, so for n atoms there are 2^n possible valuations. So we need to estimate how many atom A has, in the worst case.

To simplify, we assume that every atom has the same length (say 1) as a word. Whilst this will be false in general (e.g., the 1,000,000th atom, $p_{1000000}$, will have length 8), it will be true e.g., for up to 10,000 atoms if we use base 10,000 (say) for arithmetic; and solving PSAT for an arbitrary formula with 10,000 different atoms is currently unthinkable.

Under this assumption, for any $r > 0$ we can easily find formulas with r atoms that have length at most $6r$. We can prove this by induction on r. If $r = 1$ then certainly both p and $\neg p$ have length ≤ 6. If it's true for r, let A have r atoms and be of length $\leq 6r$. Choose an atom p not occurring in A. Then $(p \wedge A)$, $(p \to A)$, $(\neg A \wedge \neg p)$, etc., have $r+1$ atoms. Their length is the length of A, plus 1 for p (thanks to our simplifying assumption), plus 2 for (,), plus 1 for the connective ($\wedge, \to$, etc.), and at most 2 for possible $\neg$s. The total is $\leq$ (length of $A) + 6 \leq 6r + 6 = 6(r+1)$. QED.[2]

Hence for any n (divisible by 6) there are formulas of length n with at least $n/6$ atoms. So in the worst case, there are at least $2^{n/6}$ different valuations h for A of length n. So the run time on an input of length n is at least $2^{n/6} = (\sqrt[6]{2})^n$ in the worst case — exponential. There are more sophisticated methods (tableaux etc.), but no known tractable one. PSAT is NP-complete — see section 12.

[2]In fact if B, C have s, t atoms (all different) and are of length $\leq 6s-5, 6t-5$, respectively, then $(B \wedge C)$, $(B \to C)$, $(\neg B \vee \neg C)$, etc., have $s+t=r$ atoms and are of length $\leq 6s-5+6t-5+5 = 6r-5$. This gives even more formulas.

9.5 Exhaustive search in algorithms

We now discuss a common obstacle to finding a fast solution: the need to conduct an exhaustive search for it.

Broadly, the yes/no problems we've seen fall into two types:

($\exists$) Is there a needle in the haystack? We want to show that there is a solution amongst the many possibilities. *One* way to do so (we cannot rule out the possibility that there are other ways) is to actually find a solution — e.g., a factor of a composite number, a (short) Hamiltonian circuit (HCP and TSP), or a valuation making a formula true (PSAT).

($\forall$) Is there no needle in the haystack? We want to show that none of the possible solutions is in fact a solution. **Complements** of type ($\exists$) problems are of this type. E.g., to show A is not satisfiable, we want to establish that there's *no* assignment h that makes A true.

Type ($\forall$) problems intuitively seem harder than type ($\exists$), just as finding an algorithm for a given problem is easier than proving there's no algorithm that works. But often it's not known whether they really are harder, in the sense that the time complexity of a Turing machine solution is necessarily higher.

Exhaustive search (try all possible solutions) can be used in ($\exists$) and ($\forall$), but it leads to intractability, as the number of possible solutions tends to rise at least exponentially with the input size. It could be called exhausting search. We very much want to avoid it.

But at least, for any possible solution that the search throws up, we generally have a fast (p-time) way of checking that it actually is a solution. This is certainly so in all the cases we've seen. For example, it's fast to check whether a given possible factor of a number really is a factor. It's fast to check whether a possible Hamiltonian circuit (= a listing of the nodes in some order) really is a Hamiltonian circuit, and whether a given Hamiltonian circuit has total length $< d$. It's fast to check whether a given valuation of the atoms of a formula actually makes the formula true.

So the real barrier to efficient solution of these problems is the search part.

Now for some problems, a clever search strategy has been invented, rendering them tractable. For example, consider the following yes-no problem.

Problem	instances	yes-instances	no-instances
Spanning tree weights	all pairs (G,d), where G is a connected weighted graph, and $d > 0$ is a number	those pairs (G,d) where G has a spanning tree of weight $< d$	those pairs (G,d) where all spanning trees of G have weight $\geq d$

This can be solved by finding a MST (section 8) and comparing its total weight with d. Not only is the length comparison is fast, but there's also a fast (log linear) algorithm to find a MST. It's as though we have a metal detector that guides us to the needle if there is one, so we don't need to examine the whole haystack.

For other problems such as HCP and TSP (type (∃)), no clever search strategy has yet been found, and no tractable solutions are known.[3]

So problems subdivide further:

(∃1) Problems of type (∃) for which a clever (i.e., p-time) search strategy is available.

(∃2) Problems of type (∃) for which no clever search strategy is known.

The type (∀) problems subdivide similarly. E.g., **is every spanning tree of length** $> d$**?** is type (∀1), as we can find a MST and see if it weighs in at more than d. The table gives more examples.

	∃) Is there a needle?	∀) Is there no needle?
1) a fast search strategy is known	'Does the weighted graph G have a spanning tree of weight $< d$?'	'Is every spanning tree of the weighted graph G of total weight $\geq d$?'
2) no fast search strategy is known as yet	TSP, HCP, PSAT (and all NP-complete problems)	'Is every Ham. circuit of length $> d$?' 'Is the given formula unsatisfiable?'

The point is that we know where to look for a tree of length $\leq d$ if there is one. We can narrow down the search space to a small size that can be searched tractably, given that we have a fast algorithm to check that a given possible solution to the problem is actually a solution.

Now, importantly, if we can narrow down the search space in this way, then it's just as easy to find a solution as to check that there isn't a solution! Both involve going through the same shrunken search space. So *type* (∃1) *and* (∀1) *problems are equally easy (they are tractable).* This really follows from the fact that P is closed under complement (proposition 9.10). Once we know a type (∃) problem is tractable (in P), its complement, a type (∀) problem, will also be tractable.

The type (2) problems *seem* to be intractable, but the only (!) source of intractability is our inability to find a clever strategy to replace exhaustive search. They would become tractable if we had a good search strategy.

Which problems become tractable if we discount[4] *the cost of exhaustive search?* This is a kind of science fiction question: what would it be like if . . . ? The answer is: over a thousand commonly encountered ones: the NP problems. They are of type (∃); their complements, of type (∀), would simultaneously become tractable, too.

And *could there really be a clever search strategy* for these problems, one that we've all missed? Most people think not, but no-one is sure. We explore these interesting questions in the next section, using a new kind of Turing machine.

[3]There are fast probabilistic and genetic 'solutions' to TSP that are sometimes very effective, but they are not guaranteed to solve the problem.

[4]There are several ways of doing the discounting, depending on what kind of information we want from the exhaustive search. The simplest way is to discount the cost of search in type (∃) problems — those involving simply seeing whether there exists (∃) a solution among many possibilities — and this is the approach we will take in section 10. Another way uses **oracles.**

10. Non-deterministic Turing machines

A **non-deterministic Turing machine** is one that can make *choices* of which ‘instruction’ to execute at various points in a run. So what happens during the run is *not determined* in advance. Such a machine gives us an exhaustive search for free, because by using a sequence of choices it can simply *guess* the solution (which part of the haystack to check). We don’t specify which choices are made, or how, because we are interested in solving problems when we’re given a search for free, not in the mechanism of the search. We can view the non-deterministic parts of a non-deterministic Turing machine N as ‘holes’, waiting to be filled by a clever search strategy if it’s ever invented. (Such holes are rather like variables in equations — e.g., x in $x^2+2x+1=0$ — and we know how useful variables can be.) In the meantime we can still study the behaviour of N — by studying non-determinism itself.[1]

So: a non-deterministic Turing machine is like an ordinary one, but more than one instruction may be applicable in a given state, reading a given symbol. If you like, the instruction table δ can have *more than one* entry for each pair $(q,a) \in Q \times \Sigma$. When in state q and reading a, the machine can choose which instruction to execute. This is why these machines are called non-deterministic: their behaviour on a given input is not determined in advance.

10.1 Definition of non-deterministic TM

Definition 10.1 Formally, a **non-deterministic Turing machine (NDTM)** is a 6-tuple $N = (Q,\Sigma,I,q_0,\delta,F)$ as before, but now, δ is a (total[2]) function

$$\delta : (Q \setminus F) \times \Sigma \longrightarrow 2^{Q\times\Sigma\times\{0,1,-1\}}.$$

Here, 2^X is the set of all subsets of X (the power set of X). E.g., if $X=\{1,2\}$ then $2^X=\{\emptyset,\{1\},\{2\},\{1,2\}\}$. If X has n elements then 2^X has 2^n elements, which explains the notation 2^X. So $\delta(q,a)$ is now a *set* of triples (q',a',d) in $Q\times\Sigma\times\{0,1,-1\}$.

10.1.1 Operation of NDTM

A non-deterministic Turing machine N has a one-way infinite tape and a single head, as usual. (We can consider 3-tape variants etc., if we want.) N begins in state q_0 with

[1] At first sight, a guessing machine may allow even more for free than exhaustive searches. We’ll see in §10.4 that in fact it doesn’t!

[2] We can take δ to be a **total** function, rather than a **partial** one, because if we want there to be no applicable instruction in state q when reading symbol a, we define $\delta(q,a)=\emptyset$ (empty set).

its head in square 0. In state q and reading symbol a, N works like this:

- If $q \in F$ then N halts and succeeds.
- Otherwise, N can go into state q', write symbol a', and move the head in direction $d \in \{0,1,-1\}$, for *any* $(q',a',d) \in \delta(q,a)$.
- N has free choice as to which $(q',a',d) \in \delta(q,a)$ to take.
- $\delta(q,a) = \emptyset$ means that there is no applicable instruction. In this case N halts and fails. N also halts and fails if its head tries to move left off the tape.

Of course, there are NDTMs N such that $\delta(q,a)$ always either contains a single triple (q',a',d) or is empty. Such an N will behave like an ordinary Turing machine — deterministically. So the ordinary Turing machine is a *special case* of a non-deterministic Turing machine. We have again *generalised* the definition of a Turing machine, as we did with the n-tape Turing machine in section 3 (the n-tape model generalises the ordinary one, as n could be 1).

10.1.2 Input and output of NDTM

As usual, the **input** of a NDTM N is taken to be the contents of the tape before the run, up to but not including the first blank.

However, we have a problem in defining the **output** of N for a given input. This is because N may well have more than one successful run (one in which it halts and succeeds) starting with input w. For each such run we may get a different ‘output’ — so which one is the ‘real’ output of N on w?

We could define N’s output to be the *set* of all words x of Σ such that after some successful run of N, the tape contents (up to the first blank) are x. I.e., $f_N(w)$ would be the set of all possible outputs of N on w. So f_N would be a (total[3]) function $f_N : I^* \to 2^{\Sigma^*}$.

This is getting complicated. Again we simplify matters by considering only yes/no problems. With these, N’s ‘output’ is just yes or no, according to whether N accepts or rejects the input. So we don’t need to consider its output word(s) at all.

Therefore we do not define the output of a NDTM. But we do need to revise definition 9.2 (acceptance/rejection of input) to cover NDTMs.

10.1.3 Accepting and rejecting for NDTMs

Definition 10.2 A Turing machine N (non-deterministic or otherwise) is said to **accept** the input $w \in I^*$ if *there exists* a successful run of N (one in which N halts and succeeds) given w as input. If all runs on input w end in a halt-and-fail, N is said to reject w. This agrees with our previous definition when N is deterministic.[4]

[3]If there is no successful run of N on w, then $f_N(w) = \emptyset$. So again we can take f_N to be a **total** function.

[4]If N is deterministic then it has only one run on w. So by definition, N accepts w if N halts and succeeds on input w, and rejects w if it halts & fails on w. This is just as before (definition 9.2).

The definition of **solving** a yes/no problem is the same as for deterministic Turing machines (definition 9.2): N should accept the yes-instances and reject the no-instances.

There is a lot of dubious mysticism surrounding the way NDTMs make their choices. Some writers talk about magic coins, others, lucky guesses, etc etc. In my view, there is no magic or luck involved. According to the above definition, a NDTM N accepts an input w if *it is possible* for N to halt & succeed on input w. If you remember this simple statement, you'll save yourself a lot of headaches. N would have to make all the right choices — somehow — but we don't need to say how! We are not claiming that NDTMs are (yet) a practical form of computation. They are a tool for studying complexity. As we said, non-determinism is a 'hole' waiting to be filled in by future discoveries.

Exercise 10.3 What's wrong with this argument: the non-deterministic Turing machine in figure 10.1 solves any yes/no problem, because given a yes-instance, it can move to state q_1, and given a no-instance it can move to state q_2. As it's non-deterministic, we don't need to say how it chooses! [Look at the definition of solving.]

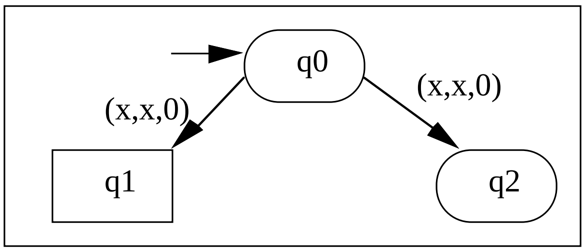


Figure 10.1: a TM to solve any problem??

10.2 Examples of NDTMs

Let's see some examples. In each one, notice how the machine first guesses a solution, and then checks that it really is a solution. Both processes are fast.

Example 10.4 (Non-primality testing) Given a number n, is it composite? We can make a 2-track NDTM N that:

1. *guesses* a number m with $1 < m < n$;
2. divides n by m (deterministically, in p-time);
3. if there's no remainder then it halts and succeeds ('yes'); otherwise it halts and fails ('no').

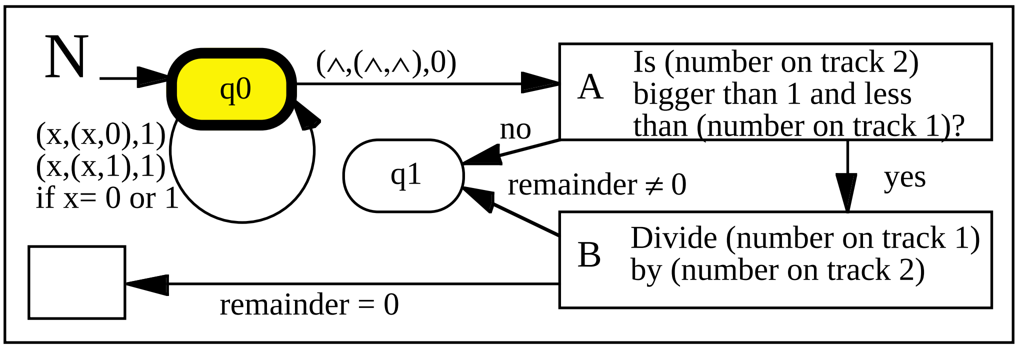

N
q0
(∧,(∧,∧),0)
(x,(x,0),1)
(x,(x,1),1)
if x= 0 or 1
A
Is (number on track 2) bigger than 1 and less than (number on track 1)?
no
yes
q1
remainder ≠ 0
B
Divide (number on track 1) by (number on track 2)
remainder = 0

Figure 10.2: a NDTM solving non-primality problem

See figure 10.2. The input is a binary number, on track 1. *N* non-deterministically writes out a number on track 2. Notice how *two* instructions are applicable in state q_0 when reading 0 or 1 on track 1: *N* can write either 0 or 1 in track 2. But if it's reading a ∧ in track 1, it *must* write ∧ and go to box A. This means that it writes 0s and 1s until the end of the input: it can't run forever. It ends with a ∧ and goes to box A.

Now it halts and succeeds iff the number it wrote was not 0, 1 or $\geq$ the input (subroutine A), and there's no remainder on dividing the input by the guessed number (subroutine B). A and B can be done deterministically in p-time, using ordinary long division, etc.

Following figure 10.2 through, we see that the only way that *N* can halt and succeed is by finding a factor. If the input is composite, then *N could possibly* halt and succeed — it has only (!) to guess the right factor. But if the input is prime, *N cannot* halt and succeed, whatever guesses it makes. It can only halt and fail. Hence *N* solves the non-primality yes/no problem.

In figure 10.3, *N* (in state q_0) has guessed '10?' (in binary) as a factor of 55 (110111 in binary). It's about to guess the last digit, '?'. If it chooses 0, leaving '100' = 4, it will halt & fail after B. But it could pick 1, leaving '101' = 5. Then it will halt and succeed. So it can possibly halt & succeed on this input, so 55 is not prime.

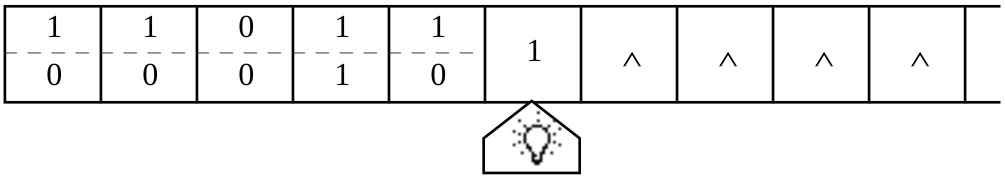


Figure 10.3: will *N* be lucky?

Exercise 10.5 Why doesn't this approach work for the problem 'is *n* prime?'

Example 10.6 (Travelling salesperson problem) An instance consists of the distances between the cities, and the bound d, all coded in some sensible manner (no unary notation, etc.). The answer is 'yes' if there is a route of length $\leq d$, and 'no' otherwise.

We can design a NDTM N that:

1. non-deterministically guesses a route around the cities;
2. works out its length (it can do this deterministically, in p-time);
3. if the length is $\leq d$ then it halts and succeeds ('yes'); otherwise it halts and fails ('no').

Clearly, if there is a route of length $\leq d$ then N *could possibly* halt and succeed — it only has to guess right in (i). But if there is no such route, N *cannot* halt and succeed. It can only halt and fail. Hence N solves TSP.

Example 10.7 (PSAT) Problem: Given a formula A of propositional logic, is A satisfiable? Using a NDTM, we can guess an assignment h, and then easily check in p-time whether or not $h(A) = \text{true}$.

Exercise 10.8 How can we solve the Hamiltonian circuit problem using a fast NDTM?

10.3 Speed of NDTMs

What does it mean to say that a non-deterministic Turing machine N runs in p-time? To answer this we have to define $time_N(n)$, as we did for deterministic TMs in §9.2.1. But because N can guess, there are now many possible runs of N on any given input $w \in I^*$. The runs are of three kinds (see figure 10.4):

1. successful (accepting) runs that end in a halt & succeed;
2. rejecting runs (ending in a halt & fail);
3. infinite runs in which N never halts.

There are several ways of defining the run time of N.[5] We will take the simplest:

Definition 10.9 (Run time function for NDTMs) $time_N(n)$ is the length of the longest possible run (i.e., the depth of the tree in figure 10.4) that N could make on any word of length n. So $time_N(n) \leq \infty$.

Again, this definition reduces to that of §9.2.1 when N is deterministic, as it then has only one run on any input.

Exercise 10.10 (for mathematicians: cf. König's tree lemma) Is definition 10.9 well-defined? What is $time_N(n)$ if N has arbitrarily long finite runs on words of length n, yet no infinite run? Then there'd be no longest run.

But in fact this can't happen! Show that for any n, if for all $r \geq 0$ there is a run of N of length at least r on some word w of length n, then N must also have an infinite run on some w of length n.

[5]We could follow Rayward-Smith and ignore the no-inputs. For most purposes (e.g., Cook's theorem) the choice doesn't matter very much.

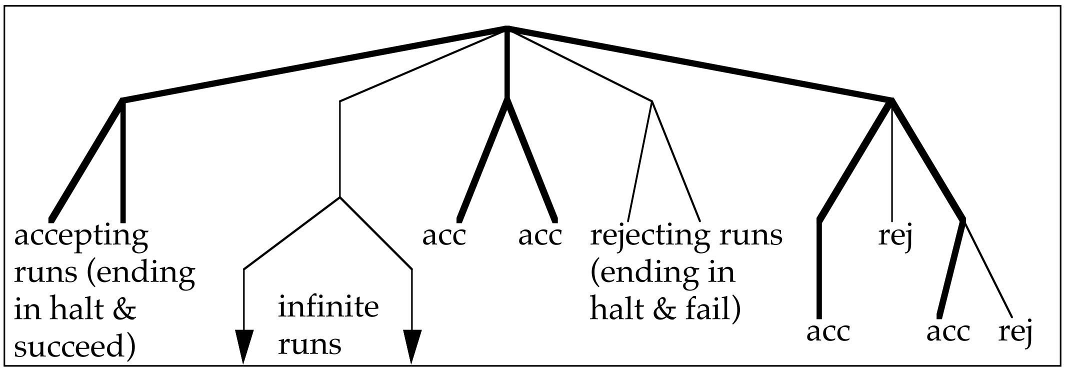

accepting runs (ending in halt & succeed)
infinite runs
acc
acc
rejecting runs (ending in halt & fail)
rej
acc
acc
rej

Figure 10.4: tree of runs of a NDTM on some input

10.4 The class NP

Definition 10.11 (p-time NDTMs) We say that a non-deterministic Turing machine N **runs in polynomial time (p-time)** if there is a polynomial $p(n)$ with positive coefficients such that $time_N(n) \le p(n)$ for all $n \ge 0$. That is, no run of N on any input of length n lasts longer than $p(n)$ steps.

This is as before. As for deterministic Turing machines, a p-time non-deterministic Turing machine always halts (cf. proposition 9.6).

Definition 10.12 (The class NP of problems) We define NP to be the class of all yes/no problems A such that there is some NDTM N that runs in p-time and solves A: N accepts all the yes-instances of A, and rejects all the no-instances.

The class NP is very important. It is the class of 'type ($\exists$) problems' (see §9.5) that would succumb to a clever search strategy. It contains many commonly occurring problems of practical importance. For instance, each NDTM described in the examples of §10.2 has p-time complexity, so PSAT, TSP, HCP, and compositeness testing are all in NP. As Pratt proved in 1977, so is primality testing (it is now known to be in P).

We said that an NDTM is a Turing machine with 'holes' waiting to be filled by a clever search strategy. In effect, all the hoped-for strategy needs do is to search the tree of figure 10.4, to find an accepting run. Thus the remark in footnote 1 on p. 137 is justified!

10.4.1 P = NP?

Clearly P $\subseteq$ NP (simply because p-time deterministic TMs are special cases of p-time NDTMs). It is not known whether P = NP — this is probably the most famous open question in computer science. Most computer scientists believe that P $\neq$ NP, but they may all be wrong. Unlike Church's thesis, the question P = NP is precisely stated. Whether P = NP or not shouldn't be a matter of belief. We want to prove it, one way or the other. A lot of work has been done, but no-one has yet published a proof.

10.5 Simulation of NDTMs by ordinary TMs

We now show that although NDTMs seem faster, they can't solve any more Y/N problems than deterministic Turing machines. This gives our last hefty chunk of evidence for Church's thesis.

The idea is very simple. We saw that NDTMs are just a quick way of doing an exhaustive search. So now we'll do the exhaustive search in full — deterministically.

As usual, we check the easy part first: NDTMs can do anything that ordinary TMs can. This is true because a deterministic Turing machine is a special case of a NDTM.

Formally, given an ordinary Turing machine $M = (Q, \Sigma, I, q_0, \delta, F)$, we can convert it into a NDTM, $N = (Q, \Sigma, I, q_0, \delta', F)$, as follows. For each $q \in Q$ and $a \in \Sigma$, let

$$\delta'(q,a) = \begin{cases} \{\delta(q,a)\}, & \text{if } \delta(q,a) \text{ is defined,} \\ \emptyset, & \text{otherwise.} \end{cases}$$

Then N behaves exactly as M does — deterministically! So N solves the same problem as M.

Now we check the hard part.

Theorem 10.13 *Any yes/no problem solvable by a NDTM can also be solved by a deterministic Turing machine.*

PROOF. [sketch] We will show that any NDTM N can be simulated by a 3-tape deterministic Turing machine M. M will accept/reject the same input words as N.

The idea is that M simulates *all possible runs* of N on the given input. At each point of the run of N there is possibly a fork, when N chooses which of several instructions to execute. So the runs of N form the branches of a tree (see figure 10.4). M will construct and search this tree in a breadth-first fashion, halting and succeeding as soon as it finds a branch that ends in a 'halt & succeed' for N.

The search must be breadth-first because branches corresponding to runs of N that never terminate (e.g., loops) are *infinite.* A depth-first search would take M off down such a branch forever, never to return. But the next branch along might represent a 'halt & succeed' run, which is what M is looking for.

Breadth-first searches are expensive in memory usage. Fortunately, Turing machines have lots of memory!

1. Each node of the tree (see figure 10.4) corresponds to the configuration of N when 'at' that node, namely the following information:

   - the current state of N,
   - the current contents of N's tape,
   - the position of N's head on its tape.

   The configuration determines the possible next moves of N, and so determines which nodes lie immediately below the current node in the tree. Note that the tree is *finitely branching* (i.e., there are only finitely many children of each node), because N has only finitely many choices in any configuration.

2. We can represent a configuration of N by a word $s * t$, where:

   - s is a word representing the current state of N (e.g., the decimal state number),
   - $*$ is a delimiter,
   - t is a '2-track word' with N's current tape contents (up to the last non-blank character) in track 1, and an X in track 2 marking the current head position (cf. section 3). Note that t could be arbitrarily long but is always finite.

   If q is a state of N, w a word of Σ, and $k \geq 0$ a number, let us write $\mathit{config}(q, w, k)$ for the word $s * t$ corresponding to the configuration where N is in state q, w is on the tape, and the head of N is over square k.

3. Any level of the tree of runs of N can be represented by a finite sequence of configurations of the form $s * t$, each separated from the next by another delimiter, say $**$. Using this data storage method, M can simulate N.

4. Initially, N's input word w is on tape 1 of M. M replaces it with $\mathit{config}(q_0, w, 0)$, using tape 2 for scratch work (copying etc.).

5. After n cycles, the tree has been explored to depth n (root = depth 0). Tape 1 of M will hold all those labelled configurations $s * t$ attached to nodes in level n of the tree.

6. Now, for each step of N, M updates tape 1.

7. First, M checks to see if any configuration so far on tape 1 is a 'halt & succeed' for N (i.e., it involves a halting state for N). If M finds any, it also halts and succeeds: the search is over.

8. Otherwise, M moves successively through the configurations $s * t$ on tape 1. Each $s * t$ corresponds to a node at level n of the tree. It may have several **child nodes** at level $n + 1$: there will be one for each possible move of N in the configuration $s * t$. For each such possible move of N in turn, M calculates the configuration of the corresponding 'child' node: the new label $s' * t'$. M can do this because it 'knows' the instruction table of N.[6] If $s' * t'$ would involve a negative head position, it is invalid and is ignored.[7]

   Otherwise, M appends it to the end of tape 2. Tape 3 is used for rough work.

9. So M works out *all* possible moves of N in each configuration $s * t$, one after the other. If there are no valid children (e.g., if N has no applicable instruction in this configuration) then no change is made to tape 2.

[6]Alternatively we could use the technique of section 4 and provide M with $\mathit{code}(N)$ on another tape. This would allow M to simulate any NDTM with the right alphabet!

[7]Note: M does *not* halt and fail as soon as it finds a single run of N that halts and fails, because *other runs* of N may be successful.

10. Having done all children of this configuration $s*t$, head 1 moves to the next configuration on tape 1, and the process repeats, the new children being appended to tape 2. And so on, through all configurations on tape 1.

11. When every configuration on tape 1 has been dealt with in this way, tape 2 holds the configurations corresponding to level $n+1$ of the tree. M can copy tape 2 back over tape 1 and go on to the next cycle (step 7 above). If tape 2 is empty, this means that there are no valid children. The tree has no level $n+1$, so M halts and fails.

Now if N accepts w then there is *some* successful run of N on w. So somewhere in the tree there's a configuration of the form

$$(\dagger) \qquad \mathit{config}(q, w, m), \text{where } q \text{ is a halting state of } N.$$

M will eventually find it, and will also halt and succeed. So M also accepts w.

On the other hand, if N rejects w, then every run of N on w ends in a halt-and-fail. So (cf. exercise 10.10) the tree will be finite, of depth n, say, with no configurations of the form $(\dagger)$. M will eventually try to construct level $n+1$ of the tree, which is empty. At that point, M halts & fails. So M rejects w too.

So N and M solve the same yes/no problem. This completes the proof. QED.

11. Reduction in p-time

We now use Turing machines to formalise the technique we saw in §8.6 of reducing one problem to another in p-time. This **p-time reduction** gives fast non-deterministic solutions to new yes/no problems from known fast non-deterministic solutions to old ones. It gives a measure of the relative hardness of yes/no problems.

11.1 Definition of p-time reduction '$\leq$'

Definition 11.1 (p-time reduction) Let A, B be any two yes/no problems (not necessarily in NP or even solvable by a Turing machine).

1. Let X be a deterministic[1] Turing machine. We say that X **reduces** A to B if:

   (a) for every yes-instance w of A, $f_X(w)$ is defined and is a yes-instance of B

   (b) for every no-instance w of A, $f_X(w)$ is defined and is a no-instance of B.

[1]We want X to be deterministic because it should be 'genuinely fast', and have an output.

2. We say that A **reduces to** B **in polynomial time** (or p-time) if there exists a deterministic Turing machine X running in p-time that reduces A to B.

3. We write A $\leq$ B if A reduces to B in polynomial time.

4. We write A $\sim$ B if A $\leq$ B and B $\leq$ A.

If A $\leq$ B then as in §8.6 we can use a fast solution to B to solve A quickly, by first reducing the instance of A to an instance of B of the same 'parity' (yes or no), and then applying the fast solution to B.

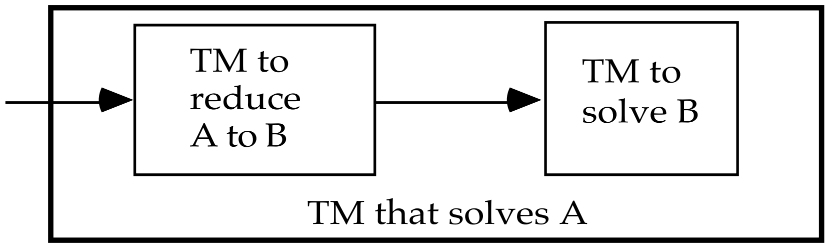
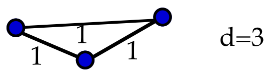


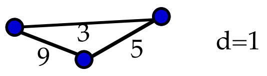
Figure 11.1: if A $\leq$ B, and we are given a solution to B, then we can solve A

Warning A $\leq$ B implies that, but is *not* the same as, **any fast solution to B can be used to solve A quickly.** There might be other ways of using B to solve A than via reduction (have a look at exercise 8.10 again).

Example 11.2 By example 8.11, HCP reduces to TSP, and the reduction can easily done by a deterministic Turing machine running in p-time. So HCP $\leq$ TSP.

Warning *Don't* try to reduce HCP to TSP in p-time as follows: given an instance G of HCP,

- If G is a yes-instance of HCP, output 

- If G is a no-instance of HCP, output 

This is a reduction (why?), but it involves determining whether G is a yes- or a no-instance. This is hard to do. There is no known p-time way to do it, so this reduction is (probably) not p-time.

A machine reducing a problem A to another, B, need not be able to solve A; and its design does not necessarily take account of whether it is given a yes- or a no-instance of A. It may be very hard (even impossible) to solve A, and yet quite easy to reduce A to B, by making simple changes to the instances in a way that preserves their yes–no parity.

But sometimes the reduction does solve the original problem. See theorem 11.11 below for an example.

Example 11.3 (change of base) Let $a,b \geq 2$. We can design a deterministic Turing machine $X_{a,b}$ running in p-time, such that for any number n, if w represents n in base a then $f_{X_{a,b}}(w)$ represents n in base b. *Change of base of arithmetic can be done in polynomial time.*

For any number $a \geq 2$, let C_a be the yes/no problem 'is w the representation in base a of a prime number?'. Then for any $a,b \geq 2$, $X_{a,b}$ reduces C_a to C_b. (Exercise: check this.) So $\mathrm{C}_a \leq \mathrm{C}_b$ (and by symmetry, $\mathrm{C}_b \leq \mathrm{C}_a$) for any $a,b \geq 2$. So $\mathrm{C}_a \sim \mathrm{C}_b$.

So with respect to the ordering $\leq$ of difficulty, changing the base makes no difference at all.

Why do we not allow $a = 1$ here — unary notation? Unary is a special case; the exercise below shows why.

Exercise 11.4 There is a deterministic Turing machine *BU* that, given the binary representation of a number as input, outputs the unary representation. (1) Design one. (2) Show that no such machine *BU* can have polynomial time complexity. [Hint: how long does *BU* take to output the answer if the input is the binary representation of n?]

11.2 $\leq$ is a pre-order

We saw that if A $\leq$ B then we can use a fast solution to B to solve A quickly. So if A $\leq$ B then in effect A is no harder than B. Thus the relation $\leq$ *orders* the yes/no problems by increasing difficulty.

But is $\leq$ really an ordering at all? In fact it's what's called a **pre-order:** a reflexive, transitive (see §7.1) binary relation. Other pre-orders include the ordering on numbers: $x \leq x$ for all x, and $x \leq y \leq z$ implies $x \leq z$. The relation T on students given by $s\ T\ t$ iff t is at least as tall as s is a pre-order. The well-known binary relation likes(x,y) may be a pre-order, if everyone likes themselves (so Bob likes Bob, for example), and whenever (say) Bob likes Chris and Chris likes Keith then also Bob likes Keith.

Theorem 11.5 *The relation $\leq$ defined above is a* pre-order *on the class of yes/no problems.*

PROOF. **$\leq$ is reflexive.** To prove this we must show that for any yes/no problem A, A $\leq$ A holds. To prove A $\leq$ A, we must find a deterministic p-time Turing machine reducing A to A.

Let I be a finite alphabet in which all instances of A can be written. Let X be the deterministic Turing machine

$$(q_0, I \cup \{\wedge\}, I, q_0, \emptyset, \{q_0\}).$$

(Cf. Y of figure 9.1.) X just halts & succeeds without action, so its output is the same as its input. Hence if w is a yes-instance of A then $f_X(w) = w$ is a yes-instance of A; and similarly for no-instances. So X reduces A to A. Moreover, X runs in polynomial time, since $time_X(n) = 0$ for all n.

$\le$ **is transitive.** Let A, B, C be yes/no problems and assume that A $\le$ B and B $\le$ C. We show that A $\le$ C.

As A $\le$ B, there is a deterministic p-time Turing machine X that reduces A to B. Similarly, as B $\le$ C, there is another deterministic p-time Turing machine Y reducing B to C. Then the Turing machine $X * Y$ obtained by running X then Y (figure 11.2) is deterministic. (We have to return to square 0 after X because Y expects to begin there.)

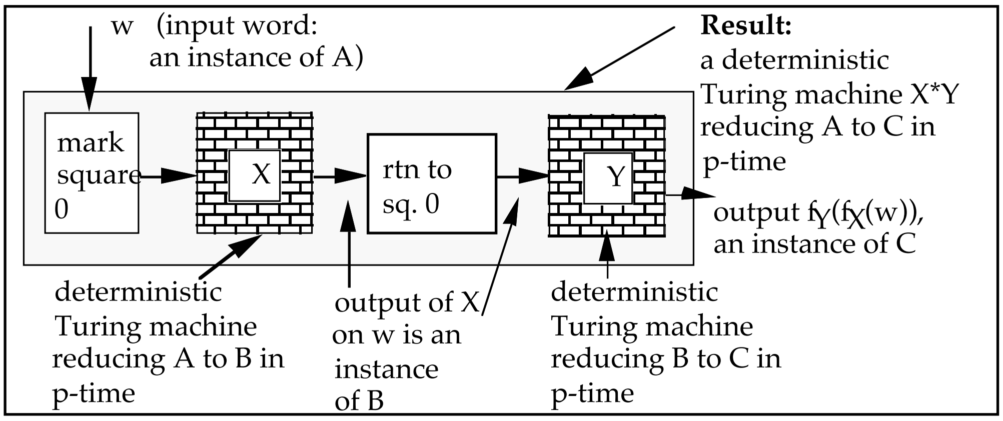

w (input word: an instance of A)
Result: a deterministic Turing machine X*Y reducing A to C in p-time
mark square 0
X
rtn to sq. 0
Y
output fY(fX(w)), an instance of C
deterministic Turing machine reducing A to B in p-time
output of X on w is an instance of B
deterministic Turing machine reducing B to C in p-time

Figure 11.2: reducing A to C (so $\le$ is transitive)

First we check that $X * Y$ reduces A to C. If w is a yes-instance of A then $f_X(w)$ is a yes-instance of B, and so $f_Y(f_X(w))$ is a yes-instance of C. Similarly if w is a no-instance of A, $f_Y(f_X(w))$ is a no-instance of C. So $X * Y$ reduces A to C, as required.

Now we check that $X * Y$ runs in p-time. Let $p(n), q(n)$ be polynomials with positive coefficients such that $time_X(n) \le p(n)$ and $time_Y(n) \le q(n)$ for all $n \ge 0$. If the input word w to $X * Y$ has length n, then:

1. Marking square 0 takes time 1.
2. Running X on w takes time $\le p(n)$.
3. Returning to and unmarking square 0 takes time k, where X's head is in square k when X halts. How big could k be? Well, X's head began in square 0, and moves at most 1 square per move. So k is at most the number of moves X made. But X made at most $p(n)$ moves. So $k \le p(n)$.
4. Running Y takes time $\le q(\text{length of } f_X(w))$. What is the length of $f_X(w)$? As above, X can write at most 1 output symbol per move. So

   $$(\text{length of } f_X(w)) \le (\text{no. of moves of } X \text{ on input } w) \le p(n).$$

   But q has positive coefficients, so if n increases then $q(n)$ can't decrease.[2] So

   $$q(\text{length of } f_X(w)) \le q(p(n)).$$

   Hence Y takes time $\le q(p(n))$.

[2]E.g., because $\frac{dq}{dn} \ge 0$ if $n \ge 0$.

The total run time of $X * Y$ is thus at most $1 + p(n) + p(n) + q(p(n))$, which works out to a polynomial. For example, if $p(n) = 2n^2 + n^3$ and $q(n) = 4 + 5n$, then the expression is $1 + 2(2n^2 + n^3) + 4 + 5(2n^2 + n^3)$, which works out to $5 + 14n^2 + 7n^3$, a polynomial. QED.

Remark 11.6

1. Steps 3 and 4 above are important.
2. We cannot just say: the symbol $\leq$ *looks like* the usual ordering $1 \leq 2 \leq 3 \ldots$ on numbers (it has a line under it: it is not $<$, but $\leq$), so therefore $\mathrm{A} \leq \mathrm{A}$ (etc). The symbol $\leq$ may look like the ordering of numbers, but it has a quite different meaning. *To prove that $\leq$ is reflexive and transitive we must use the definition of $\leq$.* No other way will do.
3. To prove that A reduces to A in p-time may seem a silly thing to do. It is not silly. It is just trivial.

Exercise 11.7 Show that the relation $\mathrm{A} \sim \mathrm{B}$ given by '$\mathrm{A} \leq \mathrm{B}$ and $\mathrm{B} \leq \mathrm{A}$' (definition 11.1) is an equivalence relation (see §7.1) on the class of all yes/no problems. (Use the theorem.)

11.3 Closure of NP under p-time reduction

We said that if $\mathrm{A} \leq \mathrm{B}$ then we can use any given fast solution to B to solve A rapidly. What if the fast 'solution' to B is by a p-time NDTM? We'd expect the resulting solution to A also to be a (hopefully p-time) NDTM. Thus we expect that if B is in NP then so is A. Let's check this.

Theorem 11.8 *Suppose A and B are yes/no problems, and* $\mathrm{A} \leq \mathrm{B}$. *If B is in NP, then A is also in NP.*

PROOF. Similar to before. As B is in NP, there is a non-deterministic Turing machine N that solves B. As $\mathrm{A} \leq \mathrm{B}$, there is a deterministic Turing machine X reducing A to B. We want to join up X to N as in figure 11.3 below (cf. figure 11.2).

Let us call the joined-up machine $X * N$. Then $X * N$ is a NDTM that solves A. For if w is a yes-instance of A, then $f_X(w)$, the input to N, is a yes-instance of B. So N accepts $f_X(w)$. Hence $X * N$ accepts w. Similarly, if w is a no-instance of A then $X * N$ rejects w.

So to show that A is in NP, it is enough to show:

Claim. The non-deterministic TM $X * N$ runs in p-time.
Proof of claim. We must show $time_{X*N}(n) \leq r(n)$ for some polynomial r (and for all n). The count goes much as before. Let w be an instance of A. We calculate how long $X * N$ can run for on input w. Assume $time_X(n) \leq p(n)$ and $time_N(n) \leq q(n)$ for polynomials p, q with positive coefficients. Let $\mathrm{length}(w) = n$. Then as in theorem 11.5:

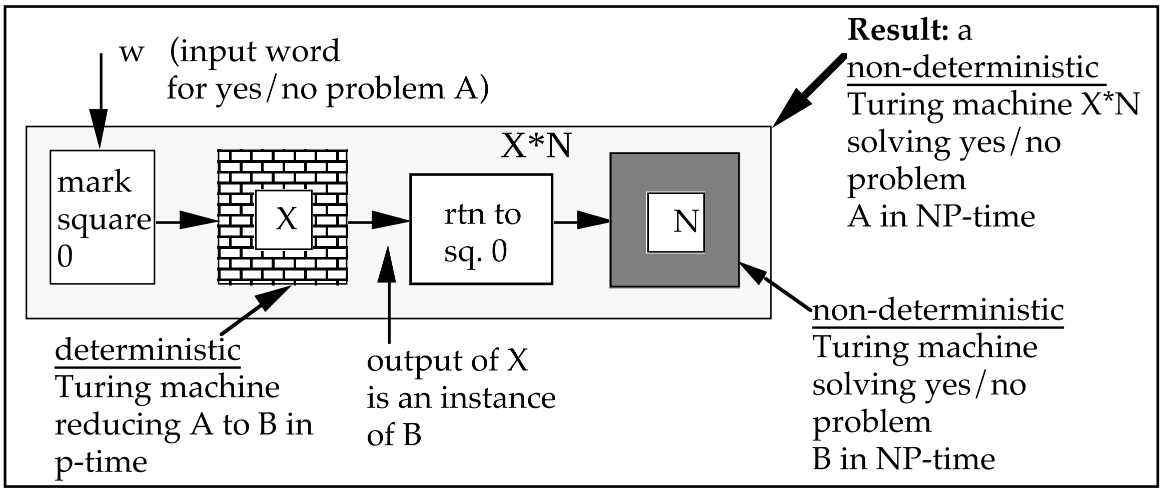


Figure 11.3: using solution to B to solve A

- Marking square 0 takes a single instruction: time 1.
- X runs for time at most $p(n)$ on input w. X halts with output $f_X(w)$.
- Returning to square 0 takes time at most $p(n)$, as before.
- Then the output $f_X(w)$ of X is given as input to N. As before, $f_X(w)$ has length $\leq p(n)$, so no run of N is longer than $q(p(n))$.

So no run of $X * N$ is longer than $r(n) = 1 + 2p(n) + q(p(n))$. This is a polynomial in n. Hence $X * N$ runs in p-time. This proves the claim.

As $X * N$ is a p-time Turing machine solving A, A is in NP. QED.

Remark We've showed that the concatenation (joining up) of p-time Turing machines X, Y by sending the output of X into Y as input, gives another p-time Turing machine $X * Y$. Y can be non-deterministic, in which case so is $X * Y$.

Corollary 11.9 *HCP is an NP problem.*

PROOF. TSP is in NP, by example 10.6. A simple formalisation of example 8.11 using Turing machines shows that HCP $\leq$ TSP. So by the theorem, HCP is in NP also. QED.

We already knew this (exercise 10.8), but reduction is useful for other things — see especially NP-complete problems in section 12. Some 1,000 other p-time reductions of problems to NP problems are known.

11.4 The P-problems are ≤-easiest

What are the *easiest* yes/no problems, with respect to our 'difficulty' ordering ≤ of problems? In fact they are those in P — the yes/no problems solvable deterministically in polynomial time.

First, we show that the *≤-easiest problems are in P.* This is similar to the proof of theorem 11.8, above. Essentially it shows that, like NP, P is closed downwards under ≤.

Theorem 11.10 *If* A*is a yes-no problem, and* A ≤ B *for all yes-no problems* B*, then* A ∈ P.

PROOF. Choose any B ∈ P. Then there is a deterministic Turing machine *M* solving B in p-time. Also, there's a deterministic p-time Turing machine *X* reducing A to B. Then the machine shown in figure 11.4 below solves A in p-time — the proof is similar to that of theorem 11.8. This shows that A ∈ P, as required. QED.

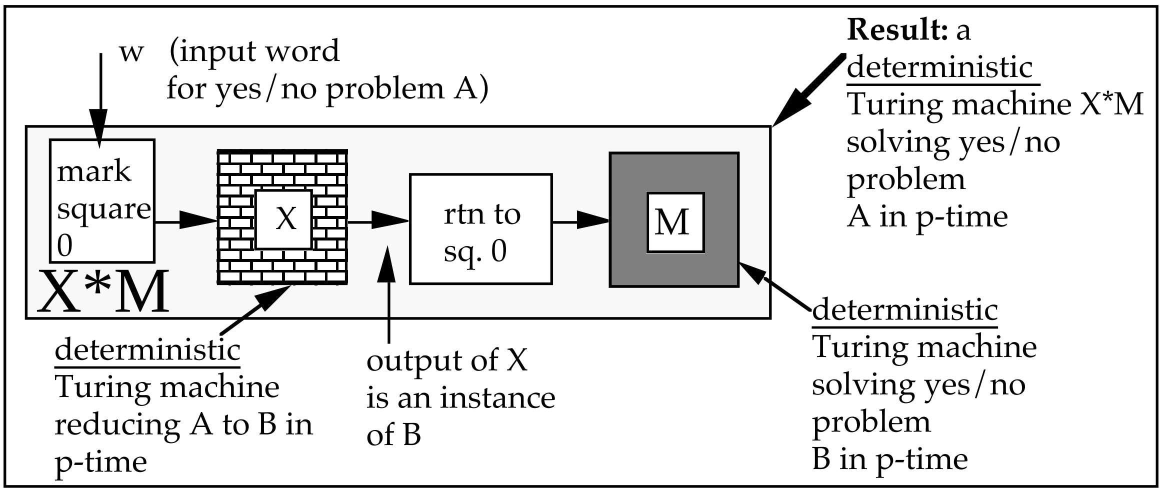

w (input word
for yes/no problem A)
mark square 0
X
rtn to sq. 0
M
X*M
Result: a deterministic Turing machine X*M solving yes/no problem A in p-time
deterministic Turing machine reducing A to B in p-time
output of X is an instance of B
deterministic Turing machine solving yes/no problem B in p-time

Figure 11.4: solving A in p-time, if A ≤ B ∈ P

Now we show the other half, that *the problems in P are ≤-easiest.* This is a new argument for us, and one that seems like a trick.

Theorem 11.11 *If* A *is any problem in* P*, and* B *is any yes/no problem at all, then* A ≤ B.

PROOF.

Crudely, the idea is this. We want to show that we can find a fast solution to A if we are allowed to use one for B. But we're told A is solvable in p-time, so we can solve A directly, without using the solution for B! This is crude because we have to show that A reduces to B in p-time, which is not quite the same (see the warning on page 146). But the same trick works.

Let M be a deterministic p-time Turing machine solving A. Choose any yes-instance w_1 and no-instance w_2 of B (remember our yes/no problems have infinitely many yes- and no-instances (see p. 129), so we can certainly find w_1, w_2). Then let X be the machine of figure 11.5.

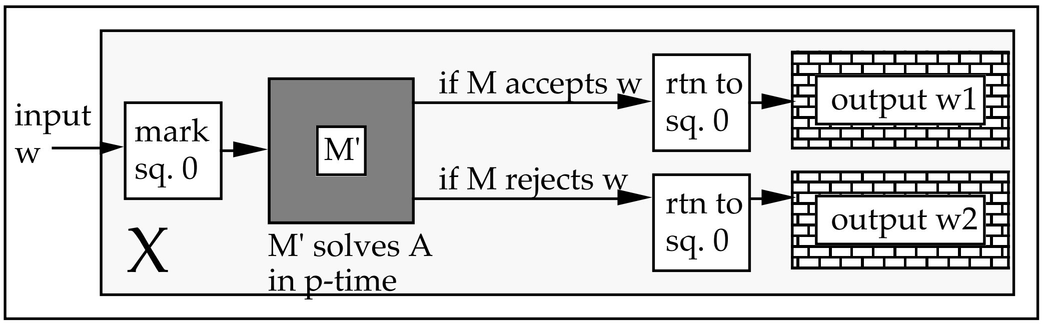


Figure 11.5: reducing a P-problem to any problem

In the figure, '**output** w_1' is a Turing machine that outputs the word w_1 as fixed text (as in the 'hello_world' example). The Turing machine '**output** w_2' is similar. X contains a likeness M' of M, slightly modified so that:

- if M halts and succeeds then control passes to **output** w_1,
- if M halts and fails then control passes to **output** w_2.

We require that M' eventually passes control to the rest of X, so that $f_X(w)$ is defined for any instance w of A. This is true, because as M runs in p-time, it always halts (see proposition 9.6).

Clearly, X is deterministic. By counting steps, as in theorems 11.5 and 11.8, we can check that M runs in p-time.

We show that X reduces A to B. If the input to X is a yes-instance w of A, then M will halt and succeed on w, so X outputs w_1, a yes-instance of B. Alternatively, if the input is a no-instance of A, then M halts and fails, and X outputs w_2, a no-instance of B. So by definition 11.1(1), X reduces A to B. So A $\leq$ B as required. QED.

Conclusion A yes-no problem A is in P iff A $\leq$ B for all yes-no problems B. Thus, P is indeed the class of $\leq$-minimal, or easiest, problems.

Exercises 11.12

1. Check that M above does run in p-time.
2. Let $\sim$ be the equivalence relation of definition 11.1(4). Let A be any yes/no problem in P. Show that for any yes/no problem B, A $\sim$ B iff B $\in$ P.
3. In theorem 5.6, we reduced HP to EIHP. Is this reduction p-time?

11.5 Summary of section

We defined the relation $\leq$ of p-time reduction between yes/no problems. The ordering $\leq$ is a pre-order, and we think of it as an ordering of difficulty: if A $\leq$ B then A is no harder than B. We write A $\sim$ B if A $\leq$ B and B $\leq$ A — they are of the same difficulty.

Figure 11.6 sketches the yes/no problems.

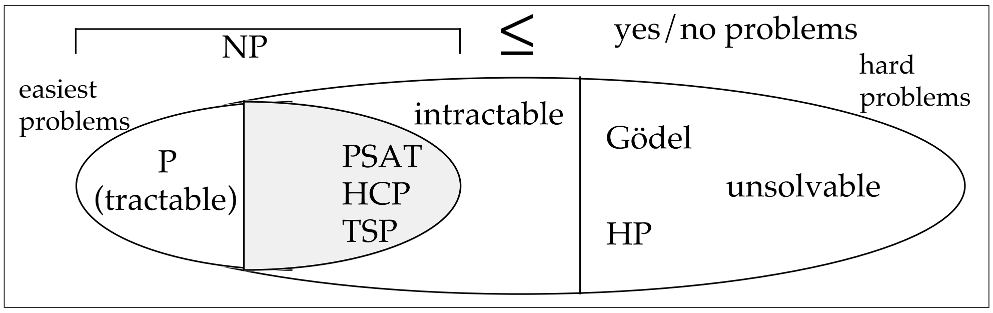


Figure 11.6: a view of y/n problems (if P $\neq$ NP)

Each $\sim$-class consist of all problems of a certain $\leq$-difficulty (of A, B such that A $\leq$ B and B $\leq$ A). By theorem 11.11 and exercise 11.12(2), P is an entire $\sim$-class, consisting of the $\leq$-easiest problems. If A $\in$ NP then any $\leq$-easier problem is also in NP. So NP is a union of classes: no $\sim$-class overlaps NP on both sides. Of course the shaded area in NP but outside P may be empty, so that PSAT and friends are all in P! Whether this is so is the question P = NP, which is unsolved.

You will probably survive if you remember that:

- $\leq$ is reflexive and transitive,
- NP is closed downwards with respect to $\leq$,
- P is the class of $\leq$-minimal problems.

The problems in P are the $\leq$-easiest. Is there is a $\leq$-hardest problem? We will investigate this — at least within NP — in the final section.

12. NP-completeness

Cheer up — this is the last section. The holiday approaches.

12.1 Introduction

Is there is a $\leq$-hardest problem? The answer is not at all obvious, even within NP. Could it be that for any NP-problem, there's always a harder one (still in NP)? If so, there'd be harder and harder problems in NP (with respect to $\leq$), forming an infinite sequence of increasingly hard problems, never stopping. Or maybe there are many different $\leq$-hardest NP-problems, all unrelated by $\leq$ — after all, why should we expect very different problems to reduce to each other? (Of course if P = NP then our question is irrelevant. But most likely, P $\neq$ NP. Ladner showed in 1975 that if P $\neq$ NP then there are infinitely many $\sim$-classes within NP.)

In fact this doesn't happen, at least within NP (and also within several other complexity classes which we won't discuss). In a famous paper of 1971, Stephen Cook proved that there are $\leq$-hardest problems in NP. Such problems are called NP**-complete problems.**

What do we mean by a $\leq$-hardest problem?

Definition 12.1 (NPC, Cook[1]) A yes/no problem A is said to be NP**-complete** if

1. A $\in$ NP,

2. B $\leq$ A for all problems B $\in$ NP.

The class of all NP-complete problems is denoted by NPC.

Exercise 12.2 Show that if A, B are NP-complete then A $\sim$ B. Show that if A is NP-complete and A $\sim$ B then B is also NP-complete. So the NP-complete problems form a single $\sim$-class.

12.1.1 NP-complete problems

But are there any NP-complete problems? Answer: yes.

Theorem 12.3 (Cook, 1971) NPC $\neq \emptyset$. *In particular,* PSAT *is* NP-*complete.*

We prove it in §12.3.

Some 1,000 examples of NP-complete problems are now known. They include PSAT, the Hamiltonian circuit problem (HCP), the travelling salesman problem (TSP), **partition** (given a set of integers, can it be divided into two sets with equal sum?) **scheduling** (can a given set of tasks of varying length be done on two identical machines to meet a given deadline?) and many other problems of great practical importance. See texts, e.g., Garey & Johnson, for a longer list. Non-primality and primality testing were known for a long time to be in NP, but are now known (since 2002) to be in P, which is even better (remember that P $\subseteq$ NP).

[1]S.A. Cook, **The complexity of theorem proving procedures,** Proceedings of Third Annual ACM Symposium on the Theory of Computing, 1971, pp. 151–158.

12.1.2 Significance of NP-completeness

At first sight, TSP and PSAT have little in common. But by exercise 12.2, any NP-complete problem is reducible in polynomial time to any other. Hence any solution to TSP can be converted into a solution to PSAT that runs in about the same time — and vice versa.

This means that we can regard an algorithm solving TSP as the same — in a sense — as one that solves PSAT. To identify algorithms that are p-time reducible to each other gives us a higher level view of algorithms.

If your boss gives you a hard-looking problem to solve, and you are feeling lazy, one way to avoid solving it is to show that it can't be solved by a Turing machine at all (as in section 5). Few bosses would then persist.

If this is not possible, it's almost as good to show that it is NP-complete. NPC problems are usually considered intractable, being the hardest in NP. If the boss insists that you write a program to solve it, you could respond that most boffins believe no such program would run in a reasonable time even for small inputs. You could produce the graph in figure 6.1.

So the boss says: 'You're a clever young person! Good degree and all that. You ought to be able to find a really clever search strategy, giving a program that runs in polynomial time — even n^5 or so!' But NP-complete problems are the $\leq$-hardest in NP. So not only would a clever program immediately give fast solutions to the 1,000 or so known NP-*complete problems* — a huge range including TSP, PSAT, college timetabling problems, map colouring problems, planning problems from AI, etc., etc., in totally different application areas — but to *all problems in* NP. Different groups of people have been looking for fast solutions to these for years, without success.[2] Some applications (e.g., in RSA cryptography) even *rely* on the assumption that their pet problem has no fast solution. So you would upset many apple-carts if you found a polynomial time solution. With the military after you, you might have to become a traveller, never visiting any city more than once.

No joy? You could hint that a p-time solution would solve P = NP and so make you more famous than your boss. If that doesn't work, you really are stuck, and if a fast solution is essential you should consider:

- optimising your algorithm as far as possible;
- restricting the problem to certain special simpler inputs;
- hoping for the best in the average case;
- looking for a sub-optimal, probabilistic or genetic 'solution';
- finding heuristics (an AI-style approach).

See Harel's or Sedgewick's book for these. For example, for the 'flat' version of TSP where the map is 'real' — for any 3 cities x, y, z, the distance from x to z is at most the sum of the distances from x to y and from y to z — there is a p-time algorithm that

[2]But they haven't *proved* there's no fast solution. Maybe this failure indicates there is a fast solution!

delivers a route round the cities of at most twice the optimal length. But for the general version of TSP, the existence of such a polynomial time algorithm would imply P = NP.

12.2 Proving NP-completeness by reduction

The best way to prove that a yes/no problem A is NP-complete is usually to show that:

1. A is in NP, and
2. A is $\geq$-harder than a known NP-complete problem B (i.e., A $\geq$ B).

For if A is $\geq$-harder than a $\geq$-hardest problem in NP, but is still in NP, then A must also be $\geq$-hardest in NP: i.e., NP-complete.

(1) is usually easy, but must not be forgotten. (For example, there are *unsolvable* problems A that satisfy (2). Satisfiability for **predicate logic** is unsolvable, but is clearly $\geq$ PSAT since PSAT is a special case of it. Such problems are not NP-complete, being outside NP.) One can either prove (1) directly, as in §10.2, or else show that A $\leq$ C for some C known to be in NP, and then use theorem 11.8.

To show (2), you must reduce a known NP-complete problem B to A in p-time. Any B in NPC will do. There are now about 1000 Bs to choose from. A popular choice is 3SAT:

3SAT: Given: a propositional formula F that is a conjunction of **clauses** of the form $X \vee Y \vee Z$, where X, Y, Z are atoms (propositional variables) or negations of atoms. E.g., $F = (p \vee q \vee \neg r) \wedge (\neg s \vee x \vee w)$.[3]

Question: is there an assignment h such that $h(F) =$ true?

Cook showed that 3SAT is NP-complete in his 1971 paper. Because the instances of 3SAT are more limited than those of PSAT, it is often simpler to reduce 3SAT to the original problem A than to reduce PSAT to it.

Exercises 12.4

1. Recall the definition of the class co-NP (definition 9.9). Show that NP = co-NP iff $NPC \cap \text{co-NP} \neq \emptyset$.[4]
2. [Quite hard; for mathematicians] Show PSAT $\sim$ 3SAT, and HCP $\sim$ PSAT. You can assume Cook's theorem (below).

[3]The '3' in 3SAT refers to there being 3 disjuncts (X, Y, Z above) in each clause. The formula F may use many more than 3 atoms.

[4]By exercise 9.11, if P = NP then NP = co-NP. It might be true that NP = co-NP and still P $\neq$ NP, but this is thought unlikely. Cf. the discussion in §9.5.

12.3 Cook's theorem

Of course, the first-discovered NP-complete problem, PSAT, wasn't shown NP-complete by reduction — no NPC problems to reduce to it were known! So to end, we'll sketch Stephen Cook's beautiful proof that PSAT is NP-complete.

We already showed (1) PSAT $\in$ NP in example 10.7. So it's enough to show (2) if A $\in$ NP then A $\leq$ PSAT. Fix any A $\in$ NP. We want to build a deterministic p-time Turing machine X such that:

- Given a yes-instance w of A, X outputs a satisfiable formula F_w of propositional logic;
- Given a no-instance w of A, X outputs an unsatisfiable formula F_w of propositional logic.

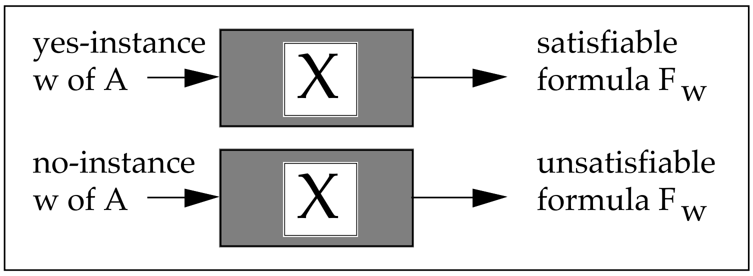


Figure 12.1: hoped-for X reducing A to PSAT

See figure 12.1. X must construct F_w from w deterministically, and in p-time.

All we know is that A $\in$ NP. So there's a non-deterministic Turing machine $N = (Q, \Sigma, I, q_0, \delta, F)$ solving A, and a polynomial $p(n)$ such that no run of N on any input of length n takes longer than $p(n)$ steps. As N solves A, the yes-instances of A are exactly those w for which N has an accepting run (one that ends in a halting state for N). Roughly, the formula F_w will directly express conditions for there to be an accepting run of N on input w! F_w will be satisfiable iff there is such a run. Compare Gödel's theorem (§5.4), where we said there was a formula $R(x)$ of arithmetic that said x coded an accepting run of a Turing machine M on an input w. This showed that we can describe Turing machines by logical formulas, and was known before Cook. But here we must use propositional logic, not arithmetic. Propositional logic is not usually powerful enough to cope, but Cook's insight was that it's OK here, as N is 'simple' (it runs in p-time, so essentially we can bound x in $R(x)$).

Table to represent a run of N **on** w When does N have an accepting run on w? We can easily (but verbosely) represent a run of N on w of length n by a *table* (see figure 12.2). The table has $p(n)+1$ big boxes, labelled 'time': $0, 1, \ldots, p(n)$. They will represent the configuration of N at each step: its state, head position, and the tape contents. Box 0 (magnified in figure 12.3 below) will represent N's configuration at

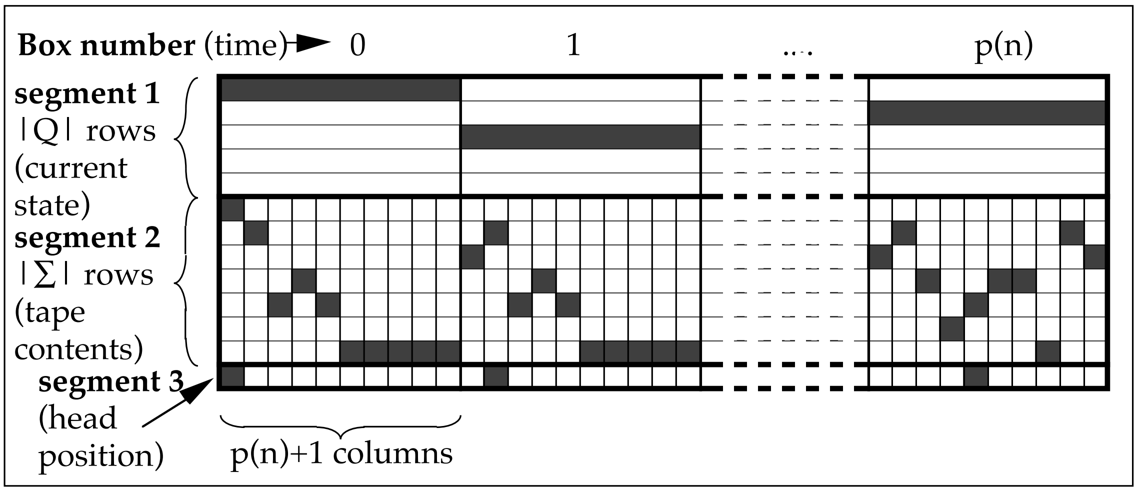


Figure 12.2: table representing a run of N on w of length n

time 0: initially. Box 1 will represent the configuration at time 1, and so on. We know N halts by time $p(n)$, so we don't need any more boxes than $0, 1, \ldots, p(n)$.

How does the table record a run of N on w? Each box is divided horizontally into three segments, to describe the configuration of N at the time that concerns the box (the 'current time'). The first segment indicates the state of N at that time. It's divided into as many rows as there are states, one row for each state. We shade the row of the current state. So if $Q = \{q_0, \ldots, q_s\}$ and the current state is q_i, row i alone will be shaded. In figure 12.3 (time 0) row 0 is shaded because initially N is in state q_0. The second segment describes the current tape contents. Now as N only has at most $p(n)$ moves, its head can never get beyond square $p(n)$. So all tape squares after $p(n)$ will always be blank, and we need only describe the contents up to square $p(n)$. We do it by chopping the segment up into rows and columns. The rows correspond to symbols from $\Sigma = \{a_0, \ldots, a_r\}$ say, where $a_r = \wedge$, and the columns correspond to squares $0, 1, \ldots, p(n)$ of the tape. We shade the intersection of row i, column j iff a_i is currently the character in square j. So the shading for time 0 describes the initial tape contents: w itself.[5]

We can read off w from figure 12.3: it is $a_0a_1a_4a_3a_4$. The rest of the tape is $a_r = \wedge$.

Finally the third segment describes N's current head position. We divide it into columns $0, 1, \ldots, p(n)$, and shade the column where the head is. The head never moves more than $p(n)$ away from square 0 in any run (it hasn't time), so we only need $p(n) + 1$ columns for this. For the time 0 box (figure 12.3) we shade column 0, as N's head begins in square 0. If N halts before $p(n)$, we can leave all later boxes blank.

Will a table like that of figure 12.2 but with *random shading* correspond to a real accepting run of N? No: there are four kinds of constraint.

1. For each $t \leq p(n)$, the box for time t must represent a genuine configuration $C(t)$

[5]Technical point: by replacing $p(n)$ by $p(n) + n$ if need be, we can assume that $p(n) \geq n$. We still have $time_M(n) \leq p(n)$. So we've room to record w itself.

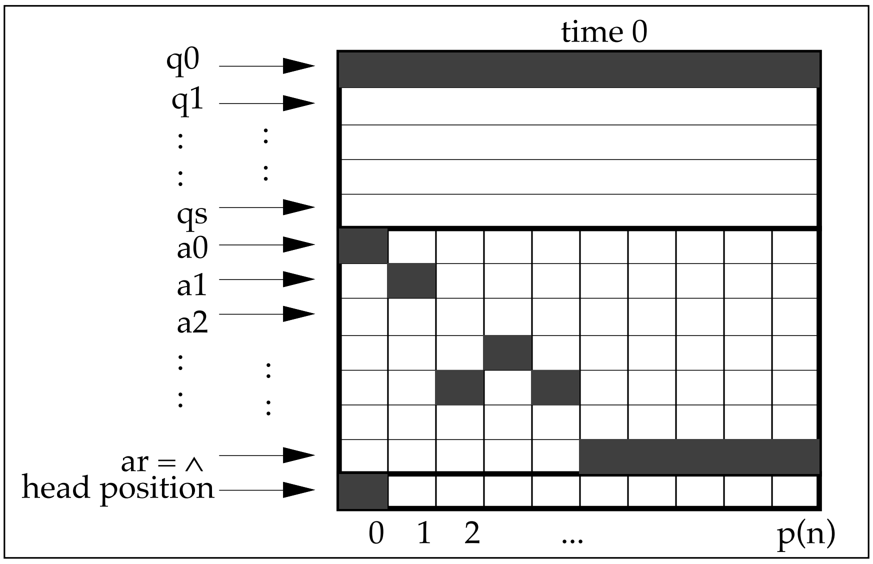


Figure 12.3: box 0 represents *config*$(q_0, a_0a_1a_4a_3a_4, 0)$ (cf. theorem 10.13)

of N. So exactly one row of segment 1 in each box should be shaded (because N is in exactly one state at each time). Similarly, just one column of segment 3 should be shaded, as the head is always in a unique position. And each column of segment 2 should have exactly one shaded row (as each tape square always has a single character).

2. $C(0)$ must be the initial configuration of N. So the box for time 0 should say that the head is in square 0 and the state is q_0, as in figure 12.3. Moreover, it should say that the initial tape contents are w.

3. The whole table must represent a run of N. The successive configurations $C(t)$ represented by the boxes should be related, as we have to be able to get from $C(t)$ to $C(t+1)$ by a single step of N. So there'll be further constraints. Only one tape character can change at a time; and the head position can vary by at most 1. Compare boxes 0 and 1 of figure 12.2. And the new character and position, and new state, are related to the old by δ. E.g., we can read off from figure 12.2 that $(q_0, a_0, q_2, a_2, 1)$ is an instruction of N.

4. The run indicated by the table must be accepting. This is a constraint on the final state, as shown in segment 1 of the last non-empty box. If the shaded state is in F, the run is accepting; otherwise, not.

Any accepting run of N on w meets these constraints and we can fill in a table for it. Conversely, if we fill in the table so as to meet the four constraints, we do get an accepting run of N on w. So the question **does** N **accept** w is the same as **can we fill in the table subject to the four constraints?**

Describing the table with logic Filling in squares of the table is really a logical Boolean operation — either a square is shaded (1), or not (0). Let's introduce a propositional atom for each little square of the table. The atom's being true will mean that its square is filled in. So a valuation v of the atoms corresponds exactly (in a 1–1 way) to a completed table (though it may not meet the constraints).

Describing the constraints with logic The constraints (1)–(4) above correspond to constraints on the truth values of the atoms. It is possible to write a propositional formula F_w that expresses these constraints. F_w does not say which squares are filled in: it only expresses the constraints. (The constraints do determine how box 0 is filled in, but boxes 1, 2, ..., $p(n)$ can be filled in in many different ways, corresponding to the different choices made by N during its run.) Given any valuation v of the atoms that makes F_w true, we can read off from v a shading for the table that meets the four constraints. And from any validly-completed table we can read off a valuation v to the atoms such that $v(F_w) = \text{true}$. So the question **can we fill in the table subject to the four constraints?** is the same as the question **is there a valuation v making F_w true?**

Writing F_w How do we write F_w? See Rayward-Smith for details. For example, suppose the atoms corresponding to the little squares in the first column of segment 2 of box 0 are $P_0, P_1, \ldots, P_r$. If the first symbol of w is a_0, as it was above, then only the first little square is shaded, so of the P_i, P_0 should be the only true atom. Therefore we include the clause $P_0 \wedge \neg P_1 \wedge \neg P_2 \wedge \ldots \wedge \neg P_r$ in F_w.

Writing F_w in full is tedious, but clearly it's an algorithmic process. And in fact, if we know the parts $Q, \Sigma, I, q_0, \delta$, and F of N, we can design a deterministic Turing machine X running in polynomial time[6] that writes out F_w when run on input w!

The end We have $f_X(w) = F_w$. But now we're finished. For, w is a yes-instance of A iff N accepts w, iff there's a way to fill in the table that meets the constraints, iff there's some valuation v making $f_X(w) = F_w$ true, iff F_w is a yes-instance of PSAT. Hence X reduces A to PSAT. Since X is deterministic and runs in p-time, we have A $\leq$ PSAT. But this holds for any A in NP. So PSAT is NP-complete. QED and goodnight.

12.4 Sample exam questions on Parts II, III

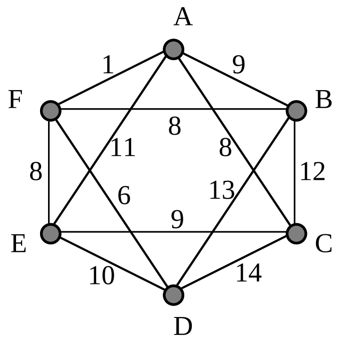
1. (a) Explain the emphasised terms:
   - i. *connected* graph
   - ii. *spanning tree* (of a connected graph)
   - iii. the *Hamiltonian circuit problem*
   - iv. the *travelling salesman problem.*

[6]To make this more plausible, let's ask how many *atoms* we need. There are $p(n)+1$ boxes. In each box, segment 1 has $|Q| = s+1$ rows, segment 2 has $(p(n)+1) \times |\Sigma| = (p(n)+1) \cdot (r+1)$ little squares, and segment 3 has $p(n)+1$ little squares. The total is $(p(n)+1)[(p(n)+1)(r+2)+s+1]$ — a polynomial!! So the number of atoms we need is a polynomial in n.

(b) A **maximal spanning tree** of a connected weighted graph *G* is a spanning tree *T* of *G*, such that the sum of the weights of the edges in *T* is as large as possible (i.e., no spanning tree of *G* has larger total weight).

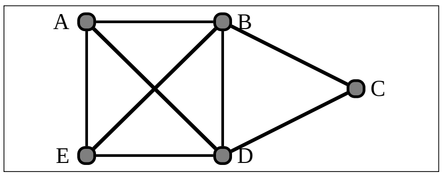
i. Suggest an algorithm to find a maximal spanning tree of a connected weighted graph. (Do not prove your algorithm correct.)

ii. Use your algorithm to find a maximal spanning tree of the following weighted graph:



2. (a) Briefly explain the difference between the **depth-first** and **breadth-first** methods of constructing a spanning tree of a connected graph.

(b) i. List the edges of a spanning tree of the following graph, using the depth-first method.



ii. Repeat b(i), using the breadth-first method.

(c) Explain the meaning of the emphasised terms:

i. A **Hamiltonian circuit** of a graph;

ii. A **minimal spanning tree** (MST) of a connected weighted graph.

(d) Let *G* be a connected weighted graph. Explain why any Hamiltonian circuit of *G* must have greater total weight than the total weight of any minimal spanning tree of *G*. [Hint: transform a Hamiltonian circuit into a spanning tree.]

3. Let A and B be yes/no problems.

(a) Explain what is meant when we say that:

i. A reduces to B in p-time (in symbols, A $\leq$ B)

ii. A is in P

iii. A is an NP-problem

iv. A is NP-complete.

For each of (i)-(iv), give an example of a problem A (or in (i), problems A and B) satisfying the condition.

(b) Show that the p-time reduction ordering $\leq$ of a(i) above is **transitive** on yes/no problems.

(c) i. Show that if A is an NP-problem, B is NP-complete and A $\geq$ B then A is also NP-complete.

ii. If A, B are both NP-complete, does it follow that A $\geq$ B? Justify your answer.

4. (a) Let A and B be arbitrary yes-no problems. Define what it means to say that A **reduces to** B **in p-time** (in symbols, A $\leq$ B).

(b) Let $\leq$ be the relation of part a. Prove that if A $\leq$ B and B $\in$ NP then A $\in$ NP.

(c) Let $\leq$ be the relation of part a.

i. Define the class NPC of NP-complete yes-no problems.

ii. Let HCP, PSAT be the Hamiltonian circuit and propositional satisfaction problems, respectively. Let A be a yes-no problem, and suppose that HCP $\leq$ A and A $\leq$ PSAT. Prove directly from your definition in part c(i) that A is NP-complete. [You can assume that HCP and PSAT are NP-complete.]

12.5 Part III in a nutshell

Section 9: We want to analyse the complexity of solvable problems in terms of how long they take to solve. We introduce yes/no problems to simplify our discussion. A Turing machine can solve such a problem by accepting (= halting & succeeding on) the yes-instances, and rejecting (= halting & failing on) the no-instances. So we don't need to consider its output. We define the **run time function** $time_M(n)$ of a Turing machine M to be the longest it can run for on any input of size n. A Turing machine **runs in p-time** if its run time function is bounded by some polynomial. P is the class of 'tractable' yes/no problems solvable by some Turing machine running in p-time. We showed that P is closed under complementation, and indeed, P $=$ co-P. Whilst no-one has proved it, many problems such as HCP (section 7), TSP (section 8) and PSAT (propositional satisfaction) do appear intractable, although they are solvable by exhaustive search. We are lacking an efficient search strategy.

Section 10: Whilst waiting for a strategy to be devised (though most think there isn't one) we can examine which problems would yield to such a strategy. To do this we use the **non-deterministic Turing machine (NDTM),** which can make choices during a run. More than one instruction may be applicable in a single configuration. (A strategy could be plugged into such a machine, narrowing the choice to one again.) A NDTM N **accepts** its input iff it has at least one accepting run on that input. It **rejects** an input if all its runs on that input end in failure. It **solves** a yes/no problem if it accepts the yes-instances and rejects the rest, as before. Its **run time function** $time_N(n)$ is the length of the longest run of N on any input of size n. The definition of N running in p-time is as before.

NDTMs can solve yes/no problems like PSAT, HCP, TSP, etc., in p-time, as they simply guess a possible solution x to the input instance w, accepting w if x is in fact a solution. Checking that x is a solution to w (e.g., x is a round trip of length $\leq d$ in TSP, or in PSAT, x is a valuation making the propositional formula A true) can be done in p-time. We let NP be the class of all yes/no problems solvable by some NDTM in p-time. So TSP, HCP and PSAT are all in NP. As deterministic Turing machines are a special case of NDTMs, any P problem is in NP.

However, though faster than deterministic Turing machines, NDTMs can solve no more yes/no problems. This gives more evidence for Church's thesis. A deterministic Turing machine can simulate any NDTM by constructing all possible runs in a breadth-first manner, and seeing if any is accepting. That is, it does a full exhaustive search of the tree of runs of the NDTM.

Section 11: We can formalise the notion of a yes/no problem A being no harder than another, B, by **p-time reduction.** To reduce A to B in p-time ('A $\leq$ B') is to find a (deterministic) p-time Turing machine X that converts yes-instances of A into yes-instances of B, and similarly for no-instances. Since X is fast, any given fast solution F to B can be used to solve A: first apply X, then F. If F solves B non-deterministically, the solution we get to A is also non-deterministic, so if B $\in$ NP and A $\leq$ B then also A $\in$ NP: that is, $\leq$-easier problems than NP-problems are also in NP.

It's easy to convert yes-instances of A into yes-instances of A and no-instances of A into no-instances of A in p-time (leave them alone!), so A $\leq$ A and $\leq$ is reflexive. $\leq$ is transitive, as if X converts A-instances to B-instances in p-time, and Y converts B-instances to C-instances in p-time, then running X then Y converts A-instances to C-instances (always preserving parity: yes goes to yes, no to no). Careful counting shows that this takes only p-time (remember to return heads to square 0, and that the input to Y may be (polynomially) longer than the original input to X). Hence A $\leq$ B $\leq$ C implies A $\leq$ C, so $\leq$ is transitive. $\leq$ is thus a pre-order.

Problems in P are $\leq$-easiest of all. For if A $\in$ P and B is arbitrary, we can convert instances w of A to instances of B in p-time and preserving parity) by the following trick. As we can solve A completely in p-time, we find out in p-time whether w is yes or no for A. Then we hand over a fixed instance of B of appropriate parity.

Section 12: The $\geq$-hardest problems in NP are called NP-complete. A yes/no problem A is NP-complete if A is in NP but A $\geq$ B for all NP-problems B. Cook proved in 1971 that PSAT is NP-complete, so NP-complete problems exist. His proof went like this. We know PSAT $\in$ NP. If A $\in$ *NP*, there's a p-time NDTM

N solving A. There's a deterministic p-time Turing machine X that given an instance w of A, outputs a propositional formula F_w expressing the constraints that must be met if N is to have an accepting run on w. Any valuation making F_w true shows that the constraints can be met, and gives an accepting run of N on w, and vice versa. Thus w is a yes-instance of A iff F_w is satisfiable. Hence X reduces A to PSAT in p-time, and so A $\leq$ PSAT. This holds for all A $\in$ NP. Thus PSAT is NP-complete.

As well as PSAT the NP-complete problems include 3SAT, HCP, TSP and some 1,000 other common problems. As they are all equally $\leq$-hard, a p-time solution to any would yield p-time solutions for all. But as many of them have been attacked seriously for years without success, it's probably not worth the effort trying to write a fast algorithm for any of them. Perhaps through frustration, people believe all NP-complete problems to be intractable: the famous question 'P = NP?' is thought almost universally to have the answer 'no'. However, no-one has proved it (or at least published a proof) either way. Unlike Church's thesis, which is by its nature unprovable, one day a proof of P $=$ NP or P $\neq$ NP may appear.

Index

In the index, the symbol '∼' denotes the current heading.