Notes on Complexity

Peter J. Cameron

Preface

These notes have been developed for the first part of the course MAS223, *Complexity and Optimisation in Operations Research*, at Queen Mary, University of London. The description for this part of the course reads:

> The course begins with an outline of complexity theory, which gives a more precise meaning to the statement that some problems (such as minimal spanning tree) are easy to solve whereas others (such as travelling salesman) are hard.

The key objectives for this part of the course are:

- Decision problems; how to express a problem with an integer solution as decision problem.
- The O and o notation; arranging functions in order of value for large argument.
- Input data representation and simple algorithms for arithmetic, matrix, and graph problems.
- Turing machines; ability to translate instructions into the action of the machine.
- Solutions of decision problems on deterministic and nondeterministic Turing machines. Definition of P and NP. Interpretation of NP in terms of certificates.
- Polynomial transformations, NP-completeness. Examples of NP-complete problems.
- Randomised and approximation algorithms. The class RP and its relation to P and NP.

The notes provide a self-contained introduction to decision, optimisation and counting problems, Turing machines, the definitions of complexity classes including P and NP and the relations between them. All of the above key objectives are covered here. The notes also include a number of worked exercises, many of which were set as homework problems in the course.

The textbook for this part of the course was

> M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, Freeman, 1979.

Another useful reference (with a different emphasis) is

Dominic Welsh, *Codes and Cryptography*, Oxford University Press 1988.

There is no shortage of material available on the World Wide Web. In addition to websites mentioned in the text, you may wish to look at an on-line course on computability and complexity by Paul Dunne (University of Liverpool) at

`http://www.csc.liv.ac.uk/~ped/teachadmin/algor/comput.html`

and some applets demonstrating various heuristics for the Travelling Salesman Problem by Stephan Mertens (University of Magdeburg) at

`http://itp.nat.uni-magdeburg.de/~mertens/TSP/TSP.html`

Peter J. Cameron
March 2001

Contents

1 Introduction **1**
1.1 Minimal connector, travelling salesman 1
1.2 Graphs, trees and circuits 6
1.3 Proofs 11

2 Problems, algorithms, computations **17**
2.1 Decision, counting, optimisation 18
2.2 Input and output 19
2.3 Orders of magnitude 21
2.4 Examples 22

3 Complexity: P and NP **37**
3.1 Turing machines 37
3.2 P and NP 43
3.3 Polynomial transformations 46
3.4 Cook's Theorem; NP-completeness 47
3.5 Examples 52

4 Other complexity classes **57**
4.1 Harder problems 57
4.2 Counting problems 59
4.3 Parallel algorithms 60
4.4 Randomised algorithms 60
4.5 Approximation algorithms 62
4.6 Quantum computation 64

Chapter 1

Introduction

Some problems are easy, some are hard.

In this course, we don't study problems which are conceptually hard, such as proving Fermat's Last Theorem. Instead, we look at problems which are simple in principle, but hard because of the amount of calculation required or the number of cases that have to be checked.

We are interested in the number of steps required to solve the problem, and, more particularly, how this number grows as a function of the "size" n of the problem. The importance of this can be seen from an example. Suppose that I can solve problems of size 100 on my current computer. Next year, I will get a new computer which is twice as fast and has twice as much memory. If the number of steps is proportional to n, I will be able to solve problems up to size 200; if it is proportional to n^2, up to size 140; but if it is proportional to 2^n, I will only be able to do the next case, 101. There is a big difference between *polynomial* and *exponential* growth!

1.1 Minimal connector, travelling salesman

We begin with two examples that will be used often during the course. Suppose that n towns are given, and we know the distance between each pair of towns. (An example with $n = 12$ is given on the next page.) Now here are two problems that we might want to solve:

- The *minimal connector problem*: we have to install a communication system linking all the towns. We want the total length of cable installed to be as small as possible.

- The *travelling salesman problem*: a salesman has to travel to all the towns, visiting each town once and returning to his starting point. We want the

Figure 1.1: Map of Great Britain

total distance travelled to be as small as possible.

Although these two problems look quite similar, we will see that the minimal connector problem is "easy", but the travelling salesman problem is "hard". There is no difficulty in principle in solving the travelling salesman problem – we could simply look at all possible cyclic tours through the towns – but the number of possibilities to check grows very rapidly, and for even a moderate number of towns it is not practicable to check all possibilities.

First we attack the minimal connector problem in a simple-minded way. We first choose the shortest possible link between any two towns. We continue doing this until all the towns are connected, except that, if two towns already have an indirect connection, we do not need to link them again. Thus, for example, if we have already chosen edges AB, BC and CD, there is no need to include AD.

	B	C	D	E	F	G	H	I	J	K	L
A	676	813	947	916	240	233	861	169	373	304	832
B		166	312	253	631	470	269	737	924	758	188
C			383	195	781	620	396	884	1094	908	253
D				399	959	786	201	1001	1201	1080	114
E					901	723	449	995	1197	1011	291
F						163	874	106	314	127	821
G							695	267	475	288	639
H								916	1116	983	122
I									208	135	885
J										304	1067
K											943

Table 1.1: Distances in km

More formally, the procedure (known as the *greedy algorithm for minimal connector*) works as follows:

- Arrange the pairs of towns in a list L in order of increasing distances. Take an empty list T.
- Repeat the following step until the edges in T connect all the towns:
  - Take the first pair in the list L, say $\{t_1, t_2\}$.
  - If the two towns t_1 and t_2 in this pair are not connected by a sequence of edges in T, add the edge $\{t_1, t_2\}$ to T.
  - Delete the pair $\{t_1, t_2\}$ from L.
- Return the list T.

Let's work this algorithm on our example. The list L begins
106 (FI), 114 (DL), 122 (HL), 127 (FK), 135 (IK), 163 (FG), 166 (BC),
169 (AI), 188 (BL), 195 (CE), 201 (DH), 208 (IJ), 233 (AG), 240 (AF),
253 (BE), 253 (CL), 267 (GI), 269 (BH), 291 (EL), 304 (AK), 304 (JK),
312 (BD), 314 (FJ), 373 (AJ), 383 (CD), 396 (CH), 399 (DE), 449 (EH),
470 (BG), 475 (JG), ...

So we choose first the edges FI, DL, HL, FK. We do not choose IK, since I and K are already connected via F. Continuing, we choose FG, BC, AI, BL, CE. We do not choose DH. We choose IJ. Then every additional edge is skipped over until we reach BG, at which point we have connected all the towns and the algorithm terminates.

What is the output of this algorithm? Clearly the final list T of edges connects all the towns. The solution contains no cycles, since to create a cycle we would have to add an edge joining two towns already connected. Thus, the solution is a *tree*. It is not obvious that it is a minimal connector, but in fact this is the case, as we prove later. It is also clear that this is an "efficient" algorithm (we will also make this more precise later).

We see that this algorithm begins by producing a number of disconnected pieces, which later coalesce. We can avoid this by a small modification to the algorithm, which also can be shown to produce a minimal connector. This is *Prim's algorithm*:

- Let L be the list of all pairs of towns (sorted by increasing distance), and T the empty list.
- Take the pair in L at least distance; add it to T.
- Repeat the following step until the edges in T connect all the towns:
  - Take the first pair in the list L having the property that one of its towns lies on an edge in T and the other does not; say $\{t_1, t_2\}$. Add $\{t_1, t_2\}$ to T.
- Return the list T.

Exercise 1.1.1 Work through this algorithm in the example, and show that it finds the same minimal connector as the greedy algorithm (though the edges are chosen in a different order).

Solution The distances in increasing order are
106 (FI), 114 (DL), 122 (HL), 127 (FK), 135 (IK), 163 (FG), 166 (BC), 169 (AI), 188 (BL), 195 (CE), 201 (DH), 208 (IJ), 233 (AG), 240 (AF), 253 (BE), 253 (CL), 267 (GI), 269 (BH), 291 (EL), 304 (AK), 304 (JK), 312 (BD), 314 (FJ), 373 (AJ), 383 (CD), 396 (CH), 399 (DE), 449 (EH), 470 (BG), 475 (JG), ...

The greedy algorithm chooses in order the edges FI, DL, HL, FK, FG, BC, AI, BL, CE, IJ, BG, at which point we have connected all the towns and the algorithm terminates.

Prim's algorithm chooses first FI, then FK, FG, AI, IJ, BG, BC, BL, DL, HL, and CE (each edge chosen is the shortest between one town already connected and one new town).

The edges are the same; only the order is different.

Let us try the same technique for the travelling salesman. Adapting the above, we have the *greedy algorithm for travelling salesman*. We assume that the number n of towns is greater than 2, else there is not much choice about the travelling salesman's itinerary.

- Let L be the list of all pairs of towns (sorted by increasing distance), and T the empty list.
- Take the first pair in L; add it to T. At this and all subsequent stages except the last, the edges in T will form a path, so we can talk about the ends of the path.
- Repeat the following step until the edges in T connect all the towns:
  - Take the first pair in the list L having the property that one of its towns is an end of the path T and the other is not on the path; say $\{t_1, t_2\}$. Add $\{t_1, t_2\}$ to T.
- Add to T the edge joining its two endpoints, creating a cycle. Return the list T.

Although this looks superficially similar to Prim's algorithm, and it does produce an itinerary for the travelling salesman, it does *not* produce a tour of smallest length.

In our example, the edges are chosen in the order

FI, FK, AI, AG, KJ, BG, BC, CE, EL, DL, DH, HJ,

giving the tour AGBCELDHJKFIA of length 3492.

However, the tour AHDLECBGFKJIA has length only 3269. (In fact this is the shortest possible tour, as can be confirmed by checking all the possibilities.)

Faced with the difficulty of the problem, we must be prepared to compromise. There are various kinds of compromise that we could make in an optimisation problem: we could be content with an efficient algorithm that does one of the following:

- it guarantees to find a solution which is not too far from the optimal;
- it makes some random choices, and guarantees to find the optimal with not-too-small probability;
- it makes some random choices, and guarantees to find a solution which is not too far from the optimal with not-too-small probability.

As an example of the first compromise, we give an algorithm which, under an assumption which is physically reasonable, finds a travelling salesman tour which is guaranteed to be "not too bad". This is the *twice-round-the-tree algorithm for the travelling salesman*:

- Find a minimal connector (e.g. using the greedy algorithm or Prim's algorithm).
- Find a tour visiting all the towns and returning to its starting point, using each edge of the tree twice. (We will discuss later how this is done.)
- Take this tour, and modify as follows: At each stage, go directly to the next town on the tour which has not yet been visited. Return the result.

In our example, the tour in the second stage of the algorithm can be chosen to be AIJIFKFGBCECBLDLHLBGFIA, and the final travelling salesman's tour is then AIJFKGBCELDHA, with length 3404, not too far from optimal!

A list of distances between pairs of towns is said to satisfy the *triangle inequality* if, for any three towns x,y,z, we have

$$d(x,y)+d(y,z) \geq d(x,z);$$

in other words, going directly from x to z is no further than detouring via y. We will show later that, if the distances satisfy the triangle inequality, then the length of the tour found by the algorithm is less than twice the minimum possible length.

Finally, we discuss how to construct the tour in the second stage of the algorithm, in the case where the towns are represented in the plane (for example, on a map). Take the minimal connector, and replace each edge by a pair of edges. Now, if we enter a town by an edge, we leave it by one of the edges immediately adjacent in the anticlockwise sense. Since the minimal connector is a tree, after exploring the branch along this edge, we return to the town along the other edge of this pair, when we again move to the next edges in the anticlockwise sense. So, when we leave the town along the pair of edges by which we originally entered, all edges through that town have been used twice.

1.2 Graphs, trees and circuits

In this section we introduce the notation and terminology of graph theory, in order to state the problems more precisely.

A *graph* consists of a set V of *vertices* and a set E of *edges*, each edge being *incident* with a pair of vertices. We denote the graph G with vertex set V and edge

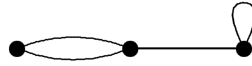


Figure 1.2: A graph

set E by the ordered pair (V,E). Figure 1.2 shows a graph, in the usual pictorial representation, with vertices represented by dots and edges by lines.

Note that our definition allows two features which are sometimes disallowed:

- There may be several edges incident with the same pair of vertices. We call such edges *multiple edges*. We saw in the twice-round-the-tree algorithm that multiple edges have their uses!
- There may be an edge with the property that the two vertices incident with it are equal. Such an edge is called a *loop*.

A graph without loops and multiple edges is called a *simple graph*. (A graph which may contain them is sometimes referred to as a *general graph* or *multi-graph*.) An important example of a simple graph is the *complete graph* K_n, the graph with n vertices and one edge incident with each pair of vertices. (That is, the edge set E consists of all 2-element subsets of the vertex set V.)

An *edge-weighted graph* is a graph with a number $d(e)$ associated with each edge e. The *total weight* of an edge-weighted graph is the sum of the weights of the edges. The weights $d(e)$ are non-negative real numbers, which may represent distances, capacities of pipelines, costs of building communication links, etc.

A *walk* in a graph is a sequence

$$(v_0,e_1,v_1,e_2,v_2,\ldots,v_{n-1},e_n,v_n),$$

where $v_0,v_1,\ldots,v_n$ are vertices, $e_1,\ldots,e_n$ are edges, and e_i is incident with v_{i-1} and v_i for $i=1,\ldots,n$. We say that it is a walk from v_0 to v_n. Two classes of walks are particularly important:

- If all the vertices are different, the walk is called a *path*.
- If all the vertices are different except that $v_n=v_0$ (and also, if $n=2$, the two edges are different), then the walk is called a *circuit*.

If two vertices are joined by a walk, then they are joined by a path. For, if the vertex v occurs more than once on the walk, we can delete the part of the walk between the first and last occurrence of v and obtain a shorter walk. After doing this finitely many times, we will obtain a path.

Proposition 1.2.1 *Let $G = (V,E)$ be a graph. Define a relation $\sim$ on V by the rule that $v \sim v'$ if there is a path from v to v'. Then $\sim$ is an equivalence relation.*

Proof The relation $\sim$ is reflexive (since v is joined to itself by the path with one vertex and no edges), and symmetric (since, if we have a path from v to w, then reversing it gives a path from w to v). We have to prove that it is transitive.

So suppose that $v \sim v'$ and $v' \sim v''$, so that there is a path $P = (v,X,v')$ from v to v', and a path $P' = (v',X',v'')$ from v' to v'', where X and X' are sequences of edges and vertices. Then (v,X,v',X',v'') is a walk from v to v''. By the remark before the proof, there is a path from v to v''; so $v \sim v'$. This completes the proof.

The equivalence classes of the relation in the preceding proposition are called the *connected components* of the graph G; and we say that G is *connected* if it has just one connected component. In other words, a graph is connected if there is a path between any two of its vertices.

A *forest* is a graph with no cycles; a *tree* is a connected forest. (So the connected components of forests are trees.) Note that a forest is a simple graph, since loops and multiple edges give rise to circuits of length 1 and 2 respectively.

Proposition 1.2.2 *Suppose that a forest has n vertices, m edges, and r connected components. Then $n = m + r$.*

Proof A forest has the property that, if one edge is removed, the number of connected components increases by 1 (see below). Using this fact, the proposition is easily proved by induction on m, the number of edges:

- If there are no edges, then each connected component is a single vertex, so $r = n$, $m = 0$, and the induction starts.

- Suppose that the proposition is true for forests with $m-1$ edges, and let G be a forest with m edges, n vertices, and r components. Removing an edge gives a graph with $m-1$ edges and $r+1$ components. By induction,

$$n = (m-1) + (r+1) = m + r,$$

and we are done.

Now let e be an edge of a forest, C the connected component containing e. Let v and w be the vertices incident with e. Now each vertex of C is joined to either v or w by a path not containing e. (Suppose that x is joined to v by a path including e. Then e must be the last vertex in the path, else v would occur twice. Deleting v and e gives a path from x to w not using e.) So when e is deleted, C splits into at most two components. But it must split: for if v and w were joined by a path not containing e, then adding e would produce a circuit, contradicting the assumption that G is a forest. The other components are unaffected by the deletion of e. So the number of components increases by one, as claimed.

Proposition 1.2.3 *Let $G = (V,E)$ be a connected graph. Then there is a subset S of E such that (V,S) is a tree.*

Proof We give an algorithm for finding such a tree.

- Initialise by setting $S = E$.
- While the graph (V,S) contains a circuit, delete from E any edge which lies in at least one circuit.
- Return the set S.

The algorithm clearly produces a graph containing no circuit. To show that it is connected, we observe that the original graph is connected, and prove that the edge-deletion step preserves connectedness. Let the deleted edge be incident with v and w. Since e lies in a circuit, there is a path P from v to w not using e. So, if a path from x to y uses e, we can replace e by P to find a walk from x to y not using e, and then shorten this walk to a path as usual.

A tree with the properties given in this proposition is called a *spanning tree* of the graph G. If G is a weighted graph, then a spanning tree of G with smallest possible total weight is a *minimal connector*.

If a graph consists of a circuit, then removing any edge gives rise to a spanning tree.

Let $G = (V,E)$ be a graph. A *Hamiltonian circuit* in G is a circuit containing all the vertices of V (each exactly once). Clearly a graph containing a Hamiltonian circuit is connected. The converse is false, and there is no simple test known for recognising *Hamiltonian graphs* (those containing Hamiltonian circuits). As we will see, this is a hard problem.

Exercise 1.2.1 Prove that K_n is Hamiltonian if and only if $n \neq 2$.

Solution Any circuit passing through all vertices in any order is Hamiltonian, since each pair of vertices is joined by an edge.

Exercise 1.2.2 Is the graph shown in Figure 1.3 (the so-called *Petersen graph*) Hamiltonian?

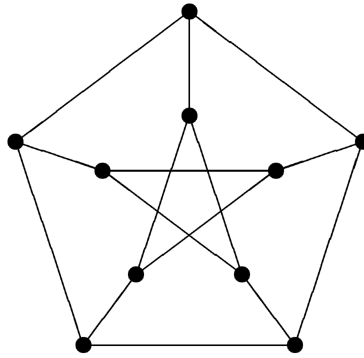


Figure 1.3: The Petersen graph

Solution The Petersen graph does not have a Hamiltonian cycle. You should follow the argument below on the drawing of the graph.

Consider the five edges joining the outer pentagon to the inner pentagram. Any Hamiltonian circuit must start and end either in the outer or in the inner cycle, and so must use an even number of these edges.

Suppose that a cycle uses two of the crossing edges. Then it must follow the outer cycle between their outer ends and the inner cycle between their inner ends. But if the outer ends are four steps apart, then the inner ends are not more than three steps apart. So no cycle can be formed using two crossing edges.

Suppose that a cycle uses four crossing edges. We can suppose that the one not used is the vertical edge in the figure. Then the two vertices on this edge must be reached by two edges of the outer and inner cycles. This gives us eight of the ten edges of the cycle, and clearly there is no way to join them up to form a cycle.

If the graph G is weighted, a Hamiltonian circuit is known as a *travelling salesman tour*, and the travelling salesman tour of minimum possible weight is the *minimal travelling salesman tour*. (We think of weights as distances between vertices in this context.)

In the first section, we looked for minimal connectors and travelling salesman tours in weighted complete graphs (that is, every pair of vertices forms an edge and has a weight).

We can reverse the procedure. Let G be an arbitrary connected simple graph with n vertices. Weight the edges of the complete graph K_n by the rule that $\{v,w\}$ has weight 1 if it is an edge of G, or 2 if not.

- A minimal connector for K_n has weight $n-1$ and is a spanning tree for G.
- G is Hamiltonian if and only if a minimal travelling salesman tour for K_n has length n.

If G is itself a weighted graph, we can use the same trick, choosing a very large weight for non-edges, to reduce questions about G to questions about K_n.

Exercise 1.2.3 Let G be a simple graph. Give each edge $\{v,w\}$ of K_n the weight 1 if it is an edge of G, and 2 if not.

(a) Prove that the weight of a minimal connector for K_n is $n+r-2$, where r is the number of connected components of G.

(b) Prove that the weight of a minimal travelling salesman tour for K_n is $n+s$, where s is the smallest number of edges whose addition to G gives a Hamiltonian graph.

Solution In this question we have two things to do in each part: construct a connector or travelling salesman tour of the specified weight, and show that no smaller weight is possible.

(a) Choose a spanning tree in each connected component of G. In each component we have one fewer edges than vertices, so altogether we will have $n-r$ edges, with total weight $n-r$. Now enlarge this to a spanning tree for K_n. We have to add $r-1$ more edges, each of weight 2 (since they do not belong to G), giving a connector of weight $n-r+2(r-1)=n+r-2$.

Now take any minimal connector for K_n. Since it is a tree, its edges which belong to G will form a forest, with at least as many components as G; say s components, where $s \geq r$. Thus, we use $n-s$ edges of G. The remaining $s-1$ edges are not in G, and have weight $2(s-1)$. The total weight is thus $n-s+2(s-1)=n+s-2 \geq n-r+2$. So the weight of a minimal connector is indeed $n-r+2$.

(b) Suppose that adding s edges to G gives a Hamiltonian graph. Then a travelling salesman tour can be constructed using at most s of these edges and at least $n-s$ edges of G; its total weight is at most $n-s+2s=n+s$.

On the other hand, a travelling salesman tour of weight $n+t$ would use at most t edges not in G, and clearly their addition to G would form a Hamiltonian graph.

1.3 Proofs

In this section we give the proofs of two results from the first section: the greedy algorithm always finds a minimal connector; and, if the triangle inequality holds,

then the twice-round-the-tree algorithm always finds a travelling salesman tour whose length is less than twice the minimum.

Theorem 1.3.1 *The greedy algorithm, applied to any weighted complete graph, always finds a minimal connector.*

Proof The greedy algorithm finds a subgraph which is connected (this is the termination condition) and has no cycles (this is the condition for an edge to be added), that is, a weighted spanning tree, with edge set S (say). We have to show that S is a spanning tree of smallest weight.

Let $e_1, e_2, \ldots, e_{n-1}$ be the edges in S, in the order in which the Greedy Algorithm chooses them. Note that

$$d(e_1) \leq \cdots \leq d(e_{n-1}),$$

since if $d(e_j) < d(e_i)$ for $j > i$, then at the ith stage, e_j would join points in different components, and should have been chosen in preference to e_i.

Suppose, for a contradiction, that there is a spanning tree of smaller weight, with edges $f_1, \ldots, f_{n-1}$, ordered so that

$$d(f_1) \leq \ldots \leq d(f_{n-1}).$$

Thus,

$$\sum_{i=1}^{n-1} d(f_i) < \sum_{i=1}^{n-1} d(e_i).$$

Choose k as small as possible so that

$$\sum_{i=1}^{k} d(f_i) < \sum_{i=1}^{k} d(e_i).$$

Note that $k > 1$, since the greedy algorithm chooses first an edge of smallest weight. Then we have

$$\sum_{i=1}^{k-1} d(f_i) \geq \sum_{i=1}^{k-1} d(e_i);$$

hence

$$d(f_1) \leq \cdots \leq d(f_k) < d(e_k).$$

Now, at stage k, the greedy algorithm chooses e_k rather than any of the edges $f_1, \ldots, f_k$ of strictly smaller weight; so all of these edges must fail the condition that they join points in different components of (V,S), where $S = \{e_1, \ldots, e_{k-1}\}$. It follows that the connected components of (V,S'), where $S' = \{f_1, \ldots, f_k\}$, are subsets of those of (V,S); so (V,S') has at least as many components as (V,S).

But this is a contradiction, since both (V,S) and (V,S') are forests, and their numbers of components are $n-(k-1)$ and $n-k$ respectively; it is false that $n-k \geq n-(k-1)$.

Theorem 1.3.2 *Suppose that the edge weights in a complete graph satisfy the triangle inequality. Then the twice-round-the-tree algorithm always finds a travelling salesman tour whose length is less than twice the minimum.*

Proof The algorithm produces a travelling salesman tour, as we already observed. Let l be the weight of a minimal connector, and L the weight of a minimal travelling salesman tour.

We have $l < L$, since deleting an edge from a Hamiltonian circuit gives a spanning tree.

In the second step of the algorithm, we double every edge of the tree, and produce a tour (with repeated vertices) of length $2l$.

In the last step, we take various "short cuts": instead of following the original tour from v through vertices $x_1, \ldots, x_m$ to w, we go straight from v to w. But an easy induction based on the Triangle Inequality shows that

$$d(v,x_1) + d(x_1,x_2) + \cdots + d(x_m,w) \geq d(v,w),$$

so these short cuts don't increase the weight of the tour. So if the final weight is L', we have $L' \leq 2l < 2L$, and we are done.

Exercise 1.3.1 Prove that, if the Triangle Inequality holds, then

$$d(v,x_1) + d(x_1,x_2) + \cdots + d(x_m,w) \geq d(v,w),$$

for any vertices $v, w, x_1, \ldots, x_m$.

Solution The proof is by induction on m. Assuming the result with $m-1$ replacing m, we have

$$d(v,x_1) + d(x_1,x_2) + \cdots + d(x_{m-1},x_m) \geq d(v,x_m),$$

and by hypothesis,

$$d(v,x_m) + d(x_m,w) \geq d(v,w).$$

The result follows.

Exercise 1.3.2 Show that, if L is the weight of a minimal travelling salesman tour, then the twice-round-the-tree algorithm produces a tour of weight at most $2(L-t)$, where we can take t to be either the nth smallest edge weight, or the second smallest weight of an edge through any particular vertex v.

Solution Let M be the weight of the minimal connector. Then the algorithm produces a travelling salesman tour of weight at most $2M$.

Now removing any edge from a travelling salesman tour gives a spanning tree, whose weight is thus not smaller than M. Suppose we remove the edge of largest weight x in the tour. Then at least n edges of the complete graph have weight smaller than x, so $x \geq t$, where t is the nth smallest edge weight. Thus $M \leq L - x \leq L - t$.

Similarly, if we pick a vertex v and remove the edge of the travelling salesman tour containing v and of larger weight x, then x is at least the second smallest weight of an edge through v, and the argument proceeds as before.

This chapter ends with something a bit different. We often use the principle that, if one of N possibilities can be determined uniquely as a result of n binary choices, then $N \leq 2^n$. (This is sometimes called the "Twenty Questions" principle, after the panel game in which the panellists are allowed to ask twenty questions with "yes" or "no" answers and have to identify some object. Since 2^{20} is a little greater than a million, in theory one of a million objects can be identified. The following exercise shows that there is a ternary version as well, where each question is allowed to have one of three possible answers.

Exercise 1.3.3 (a) I have twelve coins, which are identical except that one of the coins is either lighter or heavier than the others. I have a balance which can compare the weight of two sets of coins. Show that, in three weighings, I can determine which coin is different, and whether it is lighter or heavier than the others.

(b) Each weighing can have three results (left-hand side heavier, right-hand side heavier, or exact balance). So in three weighings I can distinguish $3^3 = 27$ possibilities. If I had 13 coins $C_1, \ldots, C_{13}$, I might expect to be able to determine which of the possible cases "C_i light", "C_i heavy" (for $1 \leq i \leq 13$) or " all coins the same", since there are $2 \cdot 13 + 1 = 27$ possibilities. (This argument shows that we certainly can't deal with more than 13 coins in just three weighings.)

Is there a scheme for determining which coin out of 13 is different in only three weighings?

Solution (a) The following three weighings can be checked to work:

$$\begin{array}{ccc} C_5, C_6, C_8, C_9 & \text{against} & C_6, C_{10}, C_{11}, C_{12}, \\ C_2, C_3, C_4, C_{10} & \text{against} & C_8, C_9, C_{11}, C_{12}, \\ C_1, C_3, C_6, C_7 & \text{against} & C_4, C_9, C_{10}, C_{12}. \end{array}$$

(b) If we have 13 coins, then there are 27 possibilities (each coin could be either light or heavy, or they might all be the same) to be determined by three weighings each with three possible outcomes. Since $3^3 = 27$, this would only be possible if the first weighing reduced the number of possibilities to 9, the second weighing to 3, and the third weighing to just one. But consider the first weighing, and suppose that we put m coins in each pan. If the left-hand pan is heavier, then we have $2m$ possibilities (a coin in the left-hand pan may be heavy, or a coin in the right-hand pan may be light). There is no integer m satisfying $2m = 9$, however, so the weighing is not possible.

Chapter 2

Problems, algorithms, computations

In this section, we make precise the notion that finding a minimal connector is easy while finding a minimal travelling salesman tour is hard. As said earlier, there is no difficulty in principle in either case, but we know how to find a solution quickly in the first case, and we don't know how (and suspect that it is not possible) in the second.

Our measure of the complexity of a problem will be the amount of computation resources required to solve it, or (more precisely) how this grows as a function of the amount of data required to specify the problem. Various resources can be considered; for example:

- *time*, the number of computational steps required to solve the problem on an idealised model of a computer;
- *space*, the maximum number of bits of information that have to be held in memory during the computation;
- *processors*, the number of processors used (in the case of a parallel algorithm);
- *randomness*, the number of coin tosses required by a "randomised" algorithm.

In this course we only consider time as a complexity measure.

Of course, the exact number of steps taken by a computation depends on the precise model of computation that we use. We will take a very simple model, a *Turing machine*. However, the most advanced chip ever made can only do the equivalent of a bounded number of Turing machine steps in a single clock cycle. Even in theory, the speed-up given by a more sophisticated model is only a polynomial factor. So, if we are not too precise about the exact number of steps, then the computational model used is not crucial.

Note: There is one computational model to which this remark does not apply. This is a *quantum computer*. Although a quantum computer cannot compute anything which could not be computed on a Turing machine, it can perform exponentially many Turing machine steps in a single cycle. However, quantum computers have not yet been built!

2.1 Decision, counting, optimisation

As we saw, the problems of deciding whether a graph is Hamiltonian and of finding the length of a minimal travelling salesman tour in a weighted complete graph are very similar. But there is one important difference between them:

- the first is a *decision problem*, "Is the graph Hamiltonian?", to which the answer is simply "yes" or "no" (one bit of information);
- the second is an *optimization problem*, "How long is the shortest travelling salesman tour?", to which the answer is a number.

We could also vary the decision problem to:

- a related *counting problem*, "How many Hamiltonian circuits does a given graph have?", to which the answer is again a number.

These three types of problems are more closely related than they appear. A decision problem is often a special case of a counting problem. (To decide whether G is Hamiltonian, we could count the number of Hamiltonian circuits in G and see whether the number is zero or not.) A decision problem may also be a special case of an optimization problem. (As we noted, to decide whether G is Hamiltonian or not, we give weight 1 to each edge of G and weight 2 to each non-edge; then G is Hamiltonian if and only if the shortest travelling salesman tour has length n.)

In the other direction, suppose that we have an optimization or counting problem, to which the solution is known to be a non-negative integer strictly smaller than N, for some number N. (The solution to a counting problem is always a non-negative integer; and, for example, the number of Hamiltonian circuits in a graph on n vertices will certainly not be greater than $n!$. In the travelling salesman problem, if all edge weights are non-negative integers not exceeding M, then the length of the shortest tour is an integer at most nM.)

In this situation, we can solve the optimization problem by solving at most $\log_2(N)$ decision problems of the form "Is the answer at least K?", for various integers K. For, when written in base 2, the solution has (at most) $\log_2(N)$ binary digits, and these can be determined one at a time (from largest to smallest) by a sequence of questions of the above type.

For example, if the answer is known to be at most 64 and is actually 27, we ask "Is the answer at least K?" for $K = 32$, 16, 24, 28, 26, 27, receiving the answers No, Yes, Yes, No, Yes,Yes; these answers reveal that the number is 011011 in base 2.

We will usually assume that the solution to any optimization problem is an integer. Usually there will be an *a priori* upper bound which is at most the exponential of a polynomial in the size of the input data, so we can reduce the problem to a polynomial number of decision problems.

2.2 Input and output

Problems with more input data, for example problems involving larger graphs, will take longer to solve; just reading the input will take longer. So we will measure the complexity of a problem by how long it takes to solve it as a function of the number of bits of input. (We arrange that the problem input is encoded as a string of bits.)

The precise way in which the encoding is done will affect the function. For example, a graph can be described in various ways. Let us assume that the vertices are numbered as $v_0, v_1, \dots, v_{n-1}$.

- If the graph is simple, we could give it as an *incidence matrix*, an $n \times n$ matrix with (i, j) entry 1 if $\{v_i, v_j\}$ is an edge, or 0 otherwise. This matrix contains n^2 bits of information.

- This can be improved slightly; since $a_{ij} = a_{ji}$ and $a_{ii} = 0$ for all i, j, it is enough to give the entries a_{ij} with $i < j$. This reduces the number of bits to $n(n-1)/2$, slightly less than half the number we had previously.

- We could give a list of n lists, the ith list consisting of all numbers j for which v_j is adjacent to v_i. This is slightly less efficient in general, since each number j lies in the interval $[0, n-1]$ and therefore takes $\log_2 n$ bits to write down in base 2 notation, so we might require as many as $n^2 \log_2 n$ bits in general.However, if the graph has only a few edges, then this method is better. For example, if each vertex is joined to exactly three others, then the number of bits required is $3n \log_2 n$.

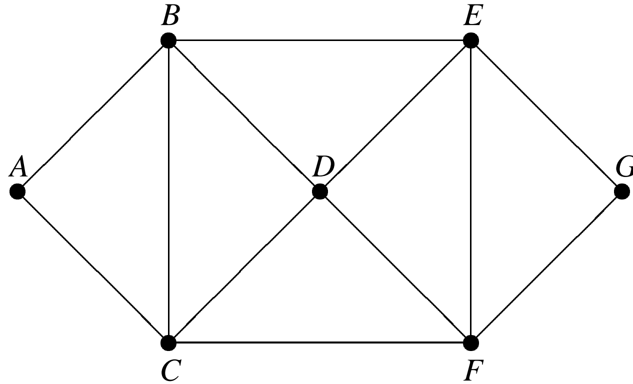


Figure 2.1: A graph

Consider the graph shown in Figure 2.1. The incidence matrix is

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

and the lists of neighbours are:

$$\begin{array}{rl} A: & BC \\ B: & ACDE \\ C: & ABDF \\ D: & BCEF \\ E: & BDFG \\ F: & CDEG \\ G: & EF \end{array}$$

You can see an example of a moderately large graph on the Web. This is a fragment of the mathematical *collaboration graph*, whose vertices are all mathematicians, two vertices adjacent if they have written a joint article. One very prolific mathematician was Paul Erdős, who died in 1996. He had over 500 co-authors. At the web site

http://www.acs.oakland.edu/~grossman/erdoshp.html

you can find the vertices of the collaboration graph at most two steps from Erdős (these number over 5000), with all edges involving at least one mathematician adjacent to Erdős. The graph is given by the "lists of neighbours" method. (A similar database for the Kevin Bacon game, at

http://www.cs.virginia.edu/oracle/

does not make the lists available but simply looks up shortest paths to Kevin Bacon.)

All that we require in the representation of input data is that it is not too inefficient. For example, an important problem (related to cryptography) is that of deciding whether a given positive integer n is prime. The input is the number n. It could be given as a string of n ones, but this is very inefficient; we could instead write n in base 2, needing only about $\log_2(n+1)$ bits. (It is simple to test in n steps whether n is prime; to do it in $(\log n)^k$ steps, for any fixed k, is much more challenging!)

2.3 Orders of magnitude

We introduce some standard notation for the order of magnitude of a function of a positive integer n. Let $f(n)$ and $g(n)$ be two such functions, where we assume that $g(n)$ is never zero.

- We say that $f(n) = O(g(n))$ (read "$f(n)$ is big Oh of $g(n)$") if there is a positive constant C such that $f(n) \leq Cg(n)$ for all sufficiently large n.
- We say that $f(n) = o(g(n))$ (read "$f(n)$ is little Oh of $g(n)$") if $f(n)/g(n) \to 0$ as $n \to \infty$.

Note that we could replace "all sufficiently large n" by "all n" in the definition of $f(n) = O(g(n))$, at the expense of increasing the constant a bit.

This notation is useful for comparing the rate of growth of functions in a simple way. For example, if

$$f(n) = \begin{cases} \frac{2}{27}n^3 + 6n^2 + 3n & \text{if } n \text{ is odd,} \\ \frac{2}{15}n^3 + 5n^2 + 7n + 4 & \text{if } n \text{ is even,} \end{cases}$$

then $f(n) = O(n^3)$.

Typically we use a very simple function for g. For example, if f is any polynomial of degree d, then $f(n) = O(n^d)$.

Exercise 2.3.1 Prove that there is no constant c such that $n! = O(c^n)$. (In other words, the function $n!$ grows faster than any exponential function.) Is $n! = O(n^n)$ true?

Solution We are asked to prove that the inequality $n! \leq Ac^n$ is false for large enough n, for any constants A and c. We observe that, if $c_1 < c_2$, then $A_1c_1^n < A_2c_2^n$ holds for large enough n for any positive constants A_1 and A_2. Taking logarithms, we require that

$$n \log c_1 + \log A_1 < n \log c_2 + \log A_2,$$

and this is true as long as

$$n > \frac{\log A_1 - \log A_2}{\log c_2 - \log c_1}.$$

So it is enough to prove that the inequality fails for given c and *some* value of A. We may assume that c is an integer.

But clearly, for any integer c, we have $n! > c!c^{n-c} = (c!/c^c)c^n$ for $n > c$, since all factors in the product apart from $1, \ldots, c$ are greater than c.

For the last part, note that

$$n! = 1 \cdot 2 \cdots n \leq n \cdot n \cdots n = n^n,$$

so certainly $n! = O(n^n)$ holds.

Exercise 2.3.2 A function g on the natural numbers is said to grow faster than another function f if $g(n) > f(n)$ for all sufficiently large n (that is, all $n > n_0$, for some number n_0). Arrange the following functions in increasing speed of growth:

$$10^{10^{10}}n,\ 10n^{10^{10}},\ 10n\log n,\ n^{\sqrt{n}},\ 10^{10}n^{10},\ 10^{10}\log n,\ n^{\log n},\ 10^{n^{10}},\ n!\ .$$

Solution We use the fact that $\log n$ grows slower than any power of n, and an exponential function of n grows faster than any power of n. Also, $n!$ grows faster than any exponential function of n. But $n!$ is smaller than $n^n = \mathrm{e}^{n \log n}$, so grows slower than $\mathrm{e}^{cn^{\alpha}}$ for any $\alpha > 1$.

For posers of n, we can ignore any constants, and order them by the exponent: thus, $10^{10^{10}}n$ comes before $10^{10}n^{10}$, which comes before $10n^{10^{10}}$.

How about a function like $n^{\sqrt{n}}$? This is equal to $\mathrm{e}^{\sqrt{n}\log n}$, so grows slower than e^{cn}, since the exponent $\sqrt{n}\log n$ grows slower than cn (because $\log n$ grows slower than $c\sqrt{n}$).

So finally the order is

$$10^{10}\log n,\ 10^{10^{10}}n,\ 10n\log n,\ 10^{10}n^{10},\ 10n^{10^{10}},\ n^{\log n},\ n^{\sqrt{n}},\ n!,\ 10^{n^{10}}.$$

2.4 Examples

In this section, we give a few examples in an informal style. These can of course be done more formally.

Addition of integers The usual operation of integer addition, applied to two n-digit integers requires only $O(n)$ operations. Here is description of how to add the integers whose base-b representations are $x_{n-1} \dots x_0$ and $y_{n-1} \dots y_0$. Here div and mod are functions giving us the quotient and remainder in an integer division, and $:=$ the assignment operator.

- Let $i := 0$ and $c := 0$. (Here i will be the number of the digit on which we are operating and c the "carry".)
- While $i \le n-1$, do the following:
  - Let $u := x_i + y_i + c$; let $z_i := u \bmod b$ and $c := u \operatorname{div} b$.
  - Let $i := i+1$.
- At the conclusion of this loop, we have $i = n$, and there are no more digits to add. If $c \neq 0$, then put $z_n := c$.

Although $x_i + y_i + c$ appears to involve two additions, it is easy to see that c is either 0 or 1. So the operation in this step can be done by looking up tables of "addition carrying zero" and "addition carrying one". Only n such lookups and assignments are required.

Multiplication of integers To multiply two n-digit integers by the usual method, we have to do n^2 multiplications and $O(n^2)$ additions, since a typical digit z_i of the product is given by

$$z_i = x_0 y_i + x_1 y_{i-1} + \cdots + x_i y_0 + \text{carry}$$

if $i \le n$. We can improve on this by breaking the integers into smaller parts. We will use base 2 here. For example, if we write

$$x = u_1 2^{n/2} + u_0, \qquad y = v_1 2^{n/2} + v_0,$$

then

$$xy = w_2 2^n + w_1 2^{n/2} + w_0,$$

where

$$\begin{aligned} w_2 &= u_1 v_1, \\ w_1 &= u_1 v_0 + u_0 v_1 = w_2 + w_0 - (u_1 - u_0)(v_1 - v_0), \\ w_0 &= u_0 v_0. \end{aligned}$$

Thus multiplication of two n-bit numbers requires three multiplications of $n/2$-bit numbers together with $O(n)$ additions and subtractions of digits.

Let $T(n)$ be the number of steps required to multiply two n-bit numbers by this method. We have

$$T(n) = 3T(n/2) + O(n).$$

From this recurrence relation we find that

$$T(n) = An^{\log 3/\log 2} + O(n \log n).$$

(If it happens that $n = 2^k$, then we find that $T(n) = 3^k T(1) + kO(n)$; now $k = \log_2 n$ and so $3^k = n^{\log 3/\log 2}$ and $kO(n) = O(n \log n)$. If n is not a power of 2, then round up to the next power of 2 above.)

Since $\log 3/\log 2 = 1.59\ldots$, this is considerably better than the $O(n^2)$ steps required by the elementary method. Still further improvements are possible. So the obvious algorithm is not always the best!

Finding the determinant Let A be an $n \times n$ matrix. How hard is it to find the determinant of A?

One way to find the determinant is to use the formula

$$\det(A) = \sum_{\sigma \in S_n} \operatorname{sign}(\sigma) a_{1\sigma(1)} a_{2\sigma(2)} \cdots a_{n\sigma(n)}, \tag{2.1}$$

where S_n is the set of all permutations of $\{1, \ldots, n\}$ and sign is the sign of the permutation. This is clearly very bad: there are $n!$ terms to be calculated, each term involving $n - 1$ multiplications, so the number of steps is more than exponential in n.

For example, let

$$A = \begin{vmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{vmatrix}$$

Equation 2.1 gives

$$\det(A) = 1.5.9 + 2.6.7 + 3.4.8 - 1.6.8 - 2.4.9 - 3.5.7 = 0.$$

The cofactor expansion is no better: the determinant is the sum of n cofactors, each an $(n-1) \times (n-1)$ determinant. In our example,

$$\det(A) = 1 \cdot \begin{vmatrix} 5 & 6 \\ 8 & 9 \end{vmatrix} - 2 \cdot \begin{vmatrix} 4 & 6 \\ 7 & 9 \end{vmatrix} + 3 \cdot \begin{vmatrix} 4 & 5 \\ 7 & 8 \end{vmatrix}$$

which involves calculating three smaller determinants.

But by using elementary row operations (Gaussian elimination), it is possible to calculate the determinant in $O(n^3)$ operations. In the example,

$$\det(A) = \begin{vmatrix} 1 & 2 & 3 \\ 0 & -3 & -6 \\ 0 & -6 & -12 \end{vmatrix} = \begin{vmatrix} 1 & 2 & 3 \\ 0 & -3 & -6 \\ 0 & 0 & 0 \end{vmatrix} = 0.$$

Of course, the advantage of n^3 over $n!$ is not clear for $n = 3$.

The $O(n^3)$ can be further improved using a trick somewhat like that for multiplication.

The *permanent* of a matrix is the function worked out by the formula for the determinant in Equation (2.1), but leaving out the sign factor:

$$\det(A) = \sum_{\sigma \in S_n} a_{1\sigma(1)} a_{2\sigma(2)} \cdots_{n\sigma(n)}, \tag{2.2}$$

For example, if A is as above, then

$$\text{per}(A) = 1.5.9 + 2.6.7 + 3.4.8 + 1.6.8 + 2.4.9 + 3.5.7 = 450.$$

The permanent is important in many matching problems. Here there is no linear algebra to help us, and nothing much better than evaluating all terms and summing is known. However, we will see that deciding whether the permanent is zero is sometimes easier.

Exercise 2.4.1 Show that any algorithm for the determinant of an $n \times n$ matrix (or even for deciding whether the matrix is non-singular) requires at least n^2 steps.

Solution It is easy to find two $n \times n$ matrices which agree in all positions except one, with the property that one has non-zero determinant and the other has zero determinant. So we cannot tell whether the determinant is zero or not without at least reading all n^2 entries!

Connectedness Given a graph, we can check efficiently whether it is connected. The algorithm given here doesn't quite do that. It begins with a directed graph (one in which each edge is given a direction, so that it has n initial vertex and a terminal vertex), and a vertex s of the graph (the source), and determines which vertices can be reached by directed paths (following the arrows) from s. If the graph is undirected, then we convert it into a directed graph by replacing each edge $\{x, y\}$ with a pair of edges (x, y) (from x to y) and (y, x) (from y to x). Now an undirected graph is connected if nd only if, from any given starting vertex s, there is a path to every vertex.

We say that w is an *out-neighbour* of v if there is an edge (v,w), and an *in-neighbour* if there is an edge (w,v).

The algorithm works as follows. During the course of the algorithm, we assign numbers to the vertices; the number assigned to v will be the least number of steps required to reach v from s. The algorithm is a version of "breadth-first search". There are two global variables, a non-negative integer i (recording the number of the stage) and a Boolean f (which tells us if we have finished).

- Begin by setting $i := 0$, $f :=$ false, and assigning the number 0 to s.
- While $f =$ false, do the following:
  - Set $f :=$ true.
  - Run through the vertices which have been assigned the number i. Whenever such a vertex has an out-neighbour which has not yet been assigned a number, assign $i+1$ to it and set $f :=$ false.
  - Set $i := i+1$.
- When we reach this stage, no new assignments have been made at the last pass. Terminate the algorithm and return the vertex assignments.

The number assigned to each vertex v is, as claimed, the least number of steps from s to v. We prove this as follows. First, there is a path from s to v of length i. This is clear if $i = 0$, since s is the only vertex assigned 0. If $i > 0$, then v is an out-neighbour of a vertex u to which $i-1$ was assigned. By induction, we can reach u in $i-1$ steps, whence we can reach v in i steps.

We must also show that there is no shorter path. Suppose that this is false, and let v be chosen so that the shortest path from s to v is smaller than the number i assigned to v, and (subject to this) that i is minimal. Clearly $i > 0$. But if u is the penultimate vertex on a path from v to w, then the number assigned to u is equal to its distance from s (which is less than $i-1$), and the algorithm assigns a number less than i to v, a contradiction.

The argument above shows that from the assignments we can find a shortest path from s to v by backtracking. If i is assigned to v, then choose any in-neighbour of v to which $i-1$ is assigned, and work back in this manner until s is reached.

Exercise 2.4.2 Apply this algorithm to the graph shown in Figure 2.1, with each edge oriented in both directions.

Solution The table shows assignments and vertex labels after each pass of the algorithm:

i	f	A	B	C	D	E	F	G
0	false	0						
1	false	0	1	1				
2	false	0	1	1	2	2	2	
3	false	0	1	1	2	2	2	3
4	true	0	1	1	2	2	2	3

Network flow A *network* consists of a weighted directed graph (having non-negative weights) with two distinguished vertices s (the *source* and t (the *target*). A *flow* in a network is a function f from the set E of edges to the non-negative real numbers satisfying

- $0 \le f(e) \le w(e)$ for each edge e;
- for any vertex $v \neq s,t$,

$$\sum_{(x,v)\in E} f(x,v) = \sum_{(v,y)\in E} f(v,y).$$

In other words, the flow in each edge cannot exceed its capacity, and the flow into and out of any vertex other than the source or target must balance (so that the net flow out of such a vertex is zero).

It is easily checked that, for any flow, the net flow out of s is equal to the net flow into t; this number is called the *value* of the flow. We are interested in finding a flow whose value is as large as possible.

If all the edge capacities are positive integers (and are not too large), then there is an efficient algorithm to solve the problem. It works as follows.

We define a *flow-augmenting path* to be a directed path which uses the following two types of edges:

Type 1: any edge $e = (x,y)$ for which the flow in e is less than the capacity;

Type 2: any edge e in which the flow is non-zero, *but used in the reverse direction* – that is, if $e = (x,y)$ has non-zero flow, then the path is allowed to use the "edge" (y,x).

Now suppose that we have a flow f with the property that each $f(e)$ is an integer. (We call such a flow *integral*.) The following step attempts to augment the flow (that is, to increase its value).

Calculate the set S of all vertices x for which there exists a flow-augmenting path from s to x. (That is, find the vertices which can be reached from s using the above two types of edges.) There are two possibilities:

(a) $t \in S$. In this case, take a flow-augmenting path from s to t, and modify the flow by increasing by one the flow in each edge of the first type on the path, and decreasing by one the flow in each edge of the second type. It can be checked that we obtain a new flow f' whose value is one greater than the value of f.

(b) $t \notin S$. In this case, let T be the complementary set to S. Then $s \in S$ and $t \in T$. Moreover, if C is the set of edges from S to T, then each edge in C carries its full capacity in the flow f. The set C is called a *cut*, since its removal leaves no path from s to t. Now no flow can have value larger than the capacity of any cut (see below). So the existence of the cut C whose capacity is equal to the value of f demonstrates that no flow with larger value is possible.

Suppose that C is any cut, consisting of the edges from S to the complementary set T, and f any flow, with value v. The flow out of s is equal to v; since all other vertices of S have equal flow in and out, the net flow out of S is equal to v. That is, the flow out of S (which must use the edges of C) minus the flow into S must be equal to v. So the capacity of C must be at least v.

Now the algorithm can be given.

- Start with the zero flow.
- Repeatedly attempt to augment the flow until no further augmentation is possible.
- At this point, return the flow f and the cut C defined in case (b) above.

Because the value of f is equal to the capacity of C, there cannot be a flow with larger value.

The number of times that we can augment the flow is not greater than the sum of the capacities of all the edges, so is at most $n_1 n_2$, where n_1 is the number of edges and n_2 the largest capacity of an edge. Searching for the flow-augmenting path takes at most n_1 steps. So the whole procedure runs in at most $n_1^2 n_2$ steps.

So we have given an algorithmic proof of the following two important theorems.

Theorem 2.4.1 (Max-Flow Min-Cut Theorem) *In any network with positive integer capacities, the maximum value of a flow is equal to the minimum capacity of a cut.*

Theorem 2.4.2 (Integrity Theorem) *In any network with positive integer capacities, there is an integral flow with maximum value. Such a flow can be found in a number of steps polynomial in the number of vertices and the maximum capacity of an edge.*

Example Consider the network shown in Figure 2.2, in which all edges have capacity 1.

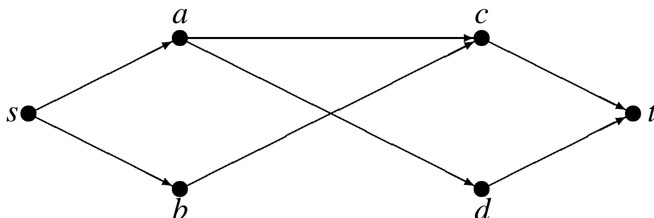


Figure 2.2: A network

In the first pass of the algorithm, we might choose the augmenting path *sact*, and introduce a flow of 1 in each of the edges (s,a), (a,c), (c,t). At the next stage, there is an augmenting path *sbcadt* (note that we use the edge (a,c) in the wrong direction, since this edge carries a positive flow). So we introduce a flow of 1 in (s,b), (b,c), (a,d) and (d,t), and reduce to zero the flow in the edge (a,c). Now there are no augmenting paths leaving s at all; so we have a flow of maximum value (namely 2), and the edges (s,a) and (s,c) form a cut with capacity 2.

In practice, rather than implementing the algorithm for Max-Flow as given, it is more efficient to begin by guessing a flow, reducing the capacities of edges accordingly, and then implementing the algorithm. The larger the value of the flow we guess, the fewer iterations of the algorithm are needed.

Exercise 2.4.3 Find a maximal flow and a minimal cut in the network in Figure 2.3. (The numbers written on the edges represent capacities, and arrows give directions.)

Solution We begin by guessing a flow. If we assign flow values

$$8,3,2,4,2,1,6,0,1,2,2,9$$

to the edges

$$sa,sb,ab,ad,ac,cb,be,dc,ce,de,dt,et$$

respectively, we have a flow of value 11. In searching for an augmenting path, we observe that *sacet* consists of edges all in the positive direction and all carrying less than capacity. So we increase the flow in these edges obtaining

$$9,3,2,4,3,1,6,0,2,2,2,10$$

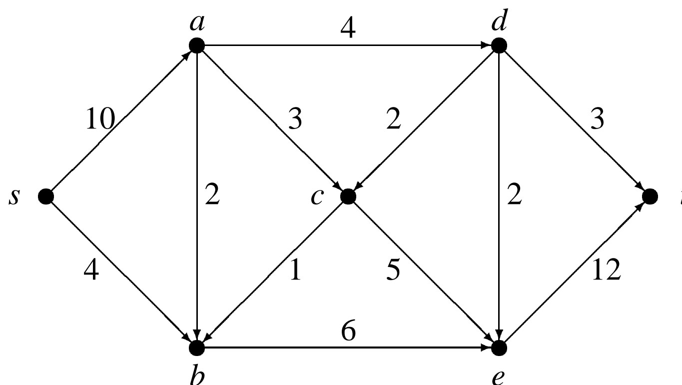


Figure 2.3: Another network

with value 12. Now, in searching for an augmenting path, we find that we can use the edge *sa*, but all outlets from *a* are blocked; we can use *sb*, then *bc* (which is carrying a flow of 1 in the reverse direction), then *cet*. This gives a new flow

$$9,4,2,4,3,0,6,0,3,2,2,11$$

with value 13. This time in searching for an augmenting path we find that we cannot leave the set $\{s,a,b\}$, so the edges leading out of this set, namely, $\{ad,ac,be\}$, form a cut with capacity 13, demonstrating that we have the maximum possible flow.

Vanishing permanent Let $A=(a_{ij})$ be an $n\times n$ matrix with non-negative entries. Remember that the permanent of A is the sum of all the terms in the determinant but without the alternating signs. We saw that, although calculating the determinant is "easy", calculating the permanent is thought to be "hard". We will give an efficient algorithm for the simpler question of deciding whether the permanent of A is zero, by reducing this question to a network flow problem.

Construct a network $N(A)$ as follows. The vertex set is

$$V=\{s,r_1,\ldots,r_n,c_1,\ldots,c_n,t\},$$

where s is the source and t the target; the edges are as follows:

- an edge (s,r_i) for $1\le i\le n$;

- an edge (r_i, c_j) if and only if $a_{ij} \neq 0$;
- an edge (c_j, t) for $1 \leq j \leq n$.

Each edge has capacity 1.

Proposition 2.4.3 *The permanent of A is non-zero if and only if the maximum value of a flow in $N(A)$ is equal to n.*

Proof Considering the edges out of s, we see that the value of a flow cannot be greater than n, and is equal to n if and only if it uses every edge out of s.

Suppose that the maximum value of a flow is equal to n. By the Integrity Theorem, the flow realising this value can be taken to be integral; and it uses every edge out of s and every edge into t. Now the flow has value 1 from each r_i to some c_j, and clearly the map σ taking i to j is a permutation of $\{1, \ldots, n\}$. Thus, $a_{i\sigma(i)} \neq 0$ for $i = 1, \ldots, n$, and we get a non-zero term in the expression for the permanent. Since all entries are non-negative, the permanent is non-zero.

Conversely, if the permanent is non-zero, then at least one term, say the term $a_{1\sigma(1)}a_{2\sigma(2)} \cdots a_{n\sigma(n)}$, is non-zero. Then there is a flow of value n, using the edges (s, r_i), $(r_i, c_{\sigma(i)})$, and (c_i, t) for $1 \leq i \leq n$.

Proposition 2.4.3 has an application to the famous "marriage problem". We are given n women and n men, with the information that some couples are compatible (that is, they would be prepared to marry), and others are not. Can we arrange the n marriages such that all married couples are compatible?

Some readers may be familiar with Hall's Marriage Theorem from Graph Theory or Combinatorics courses. According to this theorem, the compatible marriages can be arranged if and only if, for any set of k women, there are at least k men compatible with some woman in the set.

More formally, a *system of distinct representatives*, or SDR for short, for a family $(A_i : i = 1, \ldots, n)$ of sets is a family $(a_i : i = 1, \ldots, n)$ of elements having the properties

- $a_i \in A_i$ for $i = 1, \ldots, n$ (this says that a_i is a *representative* of the set A_i);
- $a_i \neq a_j$ for $i \neq j$ (this says that the a_i are *distinct*).

Now the statement of Hall's Theorem is as follows.

Theorem 2.4.4 *Let $A_1, \ldots, A_n$ be subsets of a set S. For any set I of indices (that is, for any subset I of $\{1, \ldots, n\}$), let*

$$A(I) = \bigcup_{i \in I} A_i.$$

Then the family $(A_i : i = 1, \ldots, n)$ of sets has a system of distinct representatives if and only if

$$|A(I)| \geq |I|$$

for every subset I of $\{1, \ldots, n\}$.

This is a very important and useful theorem. However, since there are 2^n subsets of $\{1, \ldots, n\}$, checking the conditions directly would be very slow! But we can do better. We restrict to the case where S has exactly n elements (that is, in the application, there are as many boys as girls).

Define an $n \times n$ matrix $M = (m_{ij})$, where

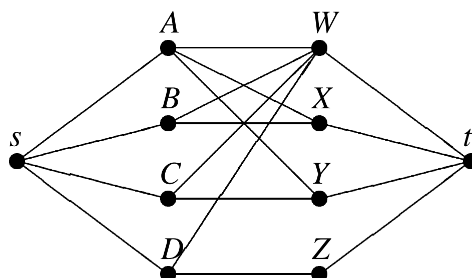
$$m_{ij} = \begin{cases} 1 & \text{if } i \in A_j, \\ 0 & \text{if not.} \end{cases}$$

This is sometimes called the *incidence matrix* of the family of sets. Then the permanent of M is non-zero if and only if there exists a SRD for the family; that is, if and only if compatible marriages can be arranged. By Proposition 2.4.3, there is an efficient algorithm to decide whether this holds or not. More generally, any SDR corresponds to a non-zero term in the permanent; so the permanent is equal to the number of SDRs.The next example illustrates.

Exercise 2.4.4 Four women A, B, C, D and four men W, X, Y, Z are friends. A would be happy to marry W, X or Y; B would be happy with W or X; C would be happy with W or Y; and D would be happy with W or Z. Use the network flow algorithm to decide whether it is possible to marry the women to the men subject to these constraints.

In how many different ways can the marriages be arranged?

Solution The network is as follows. All edges are directed from left to right and have capacity 1.



Now, either by running the network flow algorithm, or by inspection, there is a flow of value 4, using the edges sA, sB, sC, sD, AW, BX, CY, DZ, Wt, Xt, Yt, Zt, which is clearly maximal. So (A,W), (B,X), (C,Y) and (D,Z) form a compatible pairing or matching.

The number of matchings is the permanent of the matrix representing the compatibilities, namely,

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

It is easy to see that each of the ones in the first row can only occur in one non-zero term contributing to the permanent, as shown:

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

So the permanent is equal to 3, and there are three compatible matchings. These are exactly given by the three terms shown, namely

- (A,W), (B,X), (C,Y) and (D,Z)
- (A,X), (B,W), (C,Y) and (D,Z)
- (A,Y), (B,X), (C,W) and (D,Z).

Sorting One of the commonest jobs that computers do is *sorting data*: given a list of data items, in an unknown permutation of the correct order, the task is to restore the list to the correct order. We assume that the items in the list are integers and we are required to sort them into increasing order, but similar remarks apply to any sorting task. (Indeed, in discussing the Greedy Algorithm for the Minimal Connector and Travelling Salesman problems in Chapter 1, we saw the advantage of sorting the list of pairs of towns in order of increasing distance.)

Each comparison or movement of data can be broken down into more elementary machine steps; the number of such steps which is at most some polynomial in the number of digits of the numbers compared. So we will simply count the number of comparisons required by an algorithm in order to estimate the complexity of the task. Let n be the number of items in the list to be sorted.

There are $n!$ possible orderings of the list. By the end of a successful sort,we have effectively identified which one of these possibilities actually occurred. So

the number of comparisons required is not more than $\log_2 n! \sim n \log_2 n$. (As a result of k yes-no questions, we can identify one of at most 2^k possibilities.)

The simplest sorting algorithm is *Bubblesort*: we make repeated passes through the list, and whenever we find two elements out of order, we interchange them. Note that each element can move at most one place in each pass. So, if the largest element occurs in position i, we will need at least $n-i$ passes, requiring $(n-i)(n-1)$ comparisons. We see that

- there are many orderings that require at least $(n-1)^2$ comparisons;
- On average, at least $n(n-1)/2$ comparisons are required.

Clearly, this is not very good compared to our lower bound!

Several better sorting algorithms are known. One of the simplest to describe is *Quicksort*, which works as follows:

- Let x be the first element of the list, and split the remainder of the list into sublists L, R consisting of elements less than and greater than x respectively (this requires $n-1$ comparisons).
- Recursively sort L and R.
- Return $[L \text{ (sorted)}, x, R \text{ (sorted)}\ ]$.

It can be shown that this takes about $cn \log n$ comparisons on average. There are still some orders which require about cn^2 comparisons (paradoxically, if the list is already sorted, we require the maximum number $n(n-1)/2$ of comparisons!), but there are strategies to avoid this problem.

Sorting is a vitally important practical problem, and any advance which shaves a bit of time from a sorting algorithm is worth pursuing. However, from the point of view that we take in this course, the difference in complexity between Bubblesort and Quicksort is of no significance; either can be performed in time polynomial in the size of the input data. So we regard sorting as an “easy” problem from a computational point of view.

Exercise 2.4.5 Another sorting algorithm is *Mergesort*, which operates as follows:

- Divide the given list L into two nearly equal parts L_1 and L_2 (containing $\lfloor n/2 \rfloor$ and $\lceil n/2 \rceil$ items respectively).
- Sort L_1 and L_2.
- Merge the sorted lists, by starting with an empty list M and repeating the following operation:

Let a_1 and a_2 be the first entries in L_1 and L_2; remove the smaller of L_1 and L_2 from its list and add it to the final list;

until L_1 and L_2 are empty. Return M.

Show that the largest number $F(n)$ of comparisons required by Mergesort satisfies

$$F(n) = n - 1 + F(\lfloor n/2 \rfloor) + F(\lceil n/2 \rceil)$$

and deduce that $F(n) = O(n \log n)$.

Solution Sorting L_1 and L_2 takes at most $F(\lfloor n/2 \rfloor) + F(\lceil n/2 \rceil)$ comparisons, and there will be some orderings for which this number is required. The merge step requires at most $n-1$ comparisons, since once one of L_1 and L_2 is empty, no more comparisons are required. Clearly this bound is also attained. So we have proved the recurrence for F.

To show that $F(n) = O(n \log n)$, it suffices to prove this when n is a power of 2. For suppose that $F(n) \leq cn \log n$ when n is a power of 2, and let m be the least power of 2 not smaller than n. Then

$$F(n) \leq F(m) \leq cm \log m \leq 2cn \log(2n),$$

as required.

Now for powers of 2, we can prove by induction a more precise result, namely

$$F(2^k) = (k-1)2^k + 1$$

for $k \geq 1$. For we clearly have $F(1) = 1$, so the result holds for $k = 1$. Now, assuming that it holds for k, we have

$$F(2^{k+1}) = (2^{k+1} - 1) + ((k-1)2^k + 1) + ((k-1)2^k + 1) = k2^{k+1} + 1,$$

and the inductive step is done.

A more challenging exercise, which you might like to try yourself, is to show that, if $0 \leq t \leq 2^k$, then $F(2^k + t) = (k-1)2^k + 1 + t(k+1)$. In other words, F grows linearly between any two successive powers of 2.

Exercise 2.4.6 I am an industrialist who needs the solution to the “widget problem” for designing gizmos. I am a very busy man, and can only afford to wait for one week for a solution to the problem in any particular case.

(a) The standard algorithm for the widget problem solves an instance of size n of the widget problem by running through all $n!$ permutations of the constituent parts. My supercomputer can generate a permutation in one nanosecond and test it in four nanoseconds. (One nanosecond = 10^{-9} second.) How large an instance of the widget problem can I solve?

(b) A new algorithm for the problem has been devised which only involves looking through $1000n^3$ possible permutations instead of all $n!$. These configurations are more complicated to generate; each takes one microsecond ($= 10^{-6}$ second). They can still be tested in four nanoseconds. If I use the new algorithm, how large an instance can I solve?

Solution The number of seconds in a week is $7 \times 24 \times 60 \times 60 = 604800$.

(a) Generating and testing one permutation takes 5×10^{-9} seconds, so the number of permutations we can check is $604800/(5 \times 10^{-9}) = 1.2 \times 10^{14}$. So the number n must satisfy

$$n! \leq 1.2 \times 10^{14},$$

or $n \leq 16$.

(b) This time, generating and testing one permutation takes 1.004×10^{-6} seconds, so the number we can check is $604800/(1.004 \times 10^{-6}) = 6.02 \times 10^{11}$. So the number n must satisfy

$$1000n^3 \leq 6.02 \times 10^{11},$$

or $n \leq 844$.

Chapter 3

Complexity: P and NP

In this section we give a formal definition of a Turing machine, of the computational complexity of a decision problem, and the complexity classes P and NP.

3.1 Turing machines

In this section we describe Turing machines, our basic model of computation. Although Turing machines appear rather limited, it is believed that no method of computation is more powerful. This is the *Church–Turing thesis*, which states:

> Any problem which can be solved on any mechanical computational device can be solved on a Turing machine.

Of course this is not a mathematical theorem; rather, it is a statement of what computation means. Certainly no device ever constructed or even imagined (including a quantum computer) has ever violated this thesis.

Similarly, any computing device which currently exists has the property that it is "not much faster" than a Turing machine: more precisely, for any such device D, there is a polynomial p such that, if D solves the problem in n steps, a Turing machine solves it in $p(n)$ steps. (This is not an act of faith for the future; it would be false for a quantum computer if one were built.)

A *Turing machine* has two components: a read/write head and a tape. The tape is a line of unit squares, infinite in both directions, and there is a finite alphabet $A = \{a_1, \ldots, a_n\}$ such that each each square of the tape either is blank or has a symbol from A written on it. We assume that only a finite number of squares are not blank. Thus the information stored is finite, but we do not put any limit on the number of bits we can store. For ease of exposition we assume that "blank" is represented by a special symbol $\beta \in A$. Often we take the other symbols to be the binary digits 0 and 1. Figure 3.1 shows the idea.

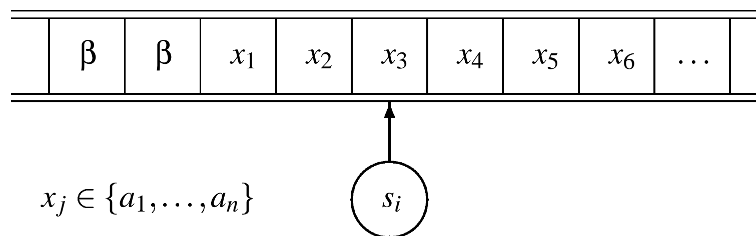


Figure 3.1: A Turing machine

The head moves over the tape, so that at any given clock cycle it is scanning one square of the tape. It also can be in any one of a finite set of *internal states* $S = \{s_1, \ldots, s_r\}$. Usually we assume that there is a distinguished *starting state*; there is no harm in also assuming a *stopping state*, as we shall see.

The action of the machine is specified by giving a list of *instructions*. Each instruction has the form

> If the head is in state s_i and it is scanning a square containing symbol a_j, then it should do one of the following actions:
>
> - move one square left;
> - move one square right;
> - change the symbol on the square to a_k;
>
> and change into state s_l.

Each such instruction can be represented by a quadruple $s_i a_j L s_l$, $s_i a_j R s_l$, or $s_i a_j a_k s_l$. The Turing machine is completely specified by the list of quadruples.

A Turing machine is *deterministic* (for short, a DTM) if, for any $s_i \in S$ and any $a_j \in A$, there is at most one quadruple in the list beginning $s_i a_j$; it is *non-deterministic* (for short, a NDTM) otherwise.

Suppose that a DTM is started with the head in state s_i scanning a square carrying symbol a_j. If there is a quadruple beginning $s_i a_j$, the machine takes the appropriate action. At the next clock cycle, it is in a new state scanning a new symbol, and the process repeats. If there is no relevant quadruple in the instruction list, the machine *halts*. Only computations which halt can actually be regarded as producing a result, so we are interested in these.

If the machine does halt, we may assume that when it does so it is in a distinguished stopping state s_H not used for any other purpose. This is achieved by listing all pairs $s_i a_j$ which do not occur at the start of any quadruple, and adding

for each a new quadruple $s_i a_j a_j s_H$ (that is, "leave the symbol scanned as it is and move into state s_H"). Since no quadruple begins s_H, the new machine will now halt in this state.

For a NDTM, the difference is that the machine may have a choice of instructions at some stage. In this case, in a particular computation, it chooses one instruction to obey. So we have a branching tree of possible computation paths. Some paths may lead to the machine halting, while others lead to it continuing forever. Again, we are only interested in computation paths which halt; again we may assume that the machine halts in a distinguished halting state.

Example The following list of quadruples defines a Turing machine which does the following job. If we write a number n on the tape in base 2, and put the head in an initial state s_0 scanning the blank square immediately to the right of the number, it replaces n by $n+1$ and returns to its starting square before halting in state s_H. Check this by tracking its operation on a number of your choice.

$$\begin{array}{l} s_0\beta L s_1 \\ s_1 1 0 s_2 \\ s_2 0 L s_1 \\ s_1 0 1 s_3 \\ s_1 \beta 1 s_3 \\ s_3 0 R s_3 \\ s_3 1 R s_3 \\ s_3 \beta\beta s_H \end{array}$$

Table 3.1: A Turing machine program

A couple of observations are in order. First, we have not specified how the machine should act if it is not set up according to the specification. Indeed, if it starts in state s_0 not scanning a blank, it does nothing; and if it starts scanning a blank not immediately to the right of a binary string, it changes the blank to its left to a 1 and then halts.

Second, not all state-symbol pairs occur in quadruples. If the machine is in state s_2, it expects to be scanning a zero: the only way state s_2 arises in normal operation is when the machine has changed a 1 to a 0 and is about to move left.

Exercise 3.1.1 Consider the Turing machine defined by the following seventeen quadruples. The states are $s_0, s_1, s_2, s_H, t_1, \ldots, t_7$, and the tape symbols are β (blank),

0 and 1.

$s_0\beta L t_1$	$s_1\beta R t_2$	$s_2\beta L t_6$
$t_1\beta 0 s_1$	$t_2\beta L s_H$	$t_6 1 0 t_7$
$s_1 0 R s_1$	$t_2 1 1 t_3$	$t_7 0 L t_6$
$s_1 1 R s_1$	$t_3 1 R t_3$	$t_6 0 1 s_1$
	$t_3\beta L t_4$	$t_6\beta 1 s_1$
	$t_4 1\beta t_5$	
	$t_5\beta L s_2$	
	$s_2 1 L s_2$	

(a) Show that, if the machine is in state s_0 scanning a blank square with a blank square to its left, then it writes 0 in the square to the left and returns to the starting square in state s_1 in three moves.

(b) Show that, if the machine is in state s_1 scanning a blank square with a blank square to its right, then it halts on the starting square in state s_H in two moves.

(c) Show that, if the machine is in state s_1 scanning a blank square with a string of 1s of length n (followed by a blank) to its right, then it erases the rightmost 1 and returns to its starting square in state s_2 in $O(n)$ moves.

(d) Show that, if the machine is in state s_2 scanning a blank square with the number n written in base 2 immediately to its left, then it replaces n with $n+1$ and returns to its starting square in state s_1 in $O(\log n)$ steps. (You may wish to compare the triples in the third column above with an example from lectures.)

(e) Now suppose that the machine starts in state s_0 scanning a blank square with n ones immediately to its right (and the rest of the tape blank). Show that it terminates on its starting square in state s_H in $O(n^2)$ steps. Describe the configuration on the tape when the machine halts.

Solution For simplicity I will write $\ldots a_1 a_2 [s_i] a_3 \ldots$ to denote that the tape has the symbols $\ldots a_1 a_2 a_3 \ldots$ written on it and the machine is in state s_i scanning the square with a_2 written.

(a) The first three instructions show that

$$\ldots\beta\beta[s_0]\ldots \to \ldots\beta[t_1]\beta\ldots \to \ldots 0[s_1]\beta\ldots \to \ldots 0\beta[s_1]\ldots$$

(b) The first two instructions in the second column show that

$$\ldots\beta[s_1]\beta\ldots \to \ldots\beta\beta[t_2]\ldots \to \ldots\beta[s_H]\beta\ldots$$

(c) In two steps, we have

$$\dots\beta[s_1]111\dots11\beta\dots\to\dots\beta1[t_2]11\dots11\beta\dots\to\dots\beta1[t_3]11\dots11\beta\dots$$

Now while the machine is in state t_3 scanning a 1, it moves right. This happens n times until we have

$$\dots\beta111\dots11\beta[t_3]\dots$$

The next steps are

$$\dots\beta111\dots11[t_4]\beta\dots\to\dots\beta111\dots1\beta[t_5]\beta\dots\to\dots\beta111\dots1[s_2]\beta\beta\dots$$

At this point there are $n-1$ ones on the tape, and the machine moves right for $n-1$ stages to reach

$$\dots\beta[s_2]111\dots1\beta\beta\dots$$

in $2n+4$ steps altogether.

(d) Consider the number n written in base 2. Suppose that the longest run of ones starting at the right (the units digit) is k (this includes the possibility that $k=0$, if the number has units digit zero. Then either the number in base 2 is $11\dots1$ (k ones) which is equal to

$$2^{k-1}+2^{k-2}+\cdots+2+1=2^k-1,$$

or it is $*011\dots1=N+2^k-1$, where N is represented by the string $*$. Note that $n\geq 2^k-1$, so that $k\leq\log_2 n+1$.

Now suppose that we start with the second case, namely $*011\dots11\beta[s_2]\dots$ One step takes us to $*011\dots11[t_6]\beta\dots$. Then we have

$$*011\dots11[t_6]\beta\dots\to *011\dots10[t_7]\beta\dots\to *011\dots1[t_6]o\beta\dots$$

In other words, in two steps the machine changes a 1 to a 0 and moves left. So after $2k$ steps we have $*0[t_6]0\dots00\beta\dots$. Next

$$*0[t_6]0\dots00\beta\dots\to *1[s_1]0\dots00\beta\dots$$

following which the machine moves right for k steps to reach $*10\dots00\beta[s_1]\dots$ The binary number $*10\dots00$ is equal to $N+2^k=n+1$, and the total number of steps taken is $1+2k+1+k=0(\log_2 n)$.

In the other case, where $n=2^k-1$, the operation is the same except that instead of changing a 0 to a 1 the machine changes a β to a 1 giving the number $10\dots00=2^k=n+1$. The number of steps is the same.

(e) Let us denote by $\{n\}$ the number n in base 2 written on the tape. Starting with $\beta\beta[s_0]1\dots1\beta\dots$ (with n ones in the string), the machine first writes a 0 to

the left (after three steps). We can write this configuration as $\{0\}\beta[s_1]1\dots1\beta\dots$. Now, after $O(n+\log_2 n)$ steps, it removes a 1 from the string to the right and increases $\{0\}$ to $\{1\}$, resulting in $\{1\}\beta[s_1]1\dots1\beta\dots$ (with $n-1$ ones in the string). After a similar time, it becomes $\{2\}\beta[s_1]1\dots1\beta\dots$ (with $n-2$ ones). That is, it repeatedly removes 1 from the string and increases the number written to the left by one. After doing this n times (which takes time $nO(n+\log_2 n) = O(n^2)$), we reach $\{n\}\beta[s_1]\beta\dots$. Then by (b), two more steps take us to $\{n\}\beta[s_H]\beta\dots$ and the machine halts. So the operation of the machine can be described as follows:

> If started in state s_0 on a blank square with a string of ones to the right and the rest of the tape blank, it counts the ones (and erases them) and writes the number of ones in base 2 to the left, then halts; all this in time $O(n^2)$.

Exercise 3.1.2 Outline the construction of a Turing machine which, when started on a blank square with the number n written in base 10 to its left on the tape, decides whether n is divisible by 3. [Hint: How would you decide whether n is divisible by 3? How would you do this if you could only remember a very small amount of information while you do the sum?] A detailed list of quadruples is not required, but you should explain the principles that your machine uses.

Solution The basic test is that a number n is divisible by 3 if and only if the sum of its digits is divisible by 3. To test this, we don't have to remember the sum of the digits, but only the sum modulo 3, and we can use three Turing machine states (say t_0, t_1, t_2) to do this. So, if the machine is in state t_i scanning the digit j, it should move one square to the left and move into state t_k, where $k \equiv i + j \pmod 3$. When it finishes reading the number and reaches a blank square, it returns the answer "yes" if it is in state t_0 and "no" otherwise.

The following set of quadruples will do the job. For a change, we use two halting states, s_{YES} and s_{NO}, to signify the answer.

$$s_0\beta L t_0$$

$$\begin{array}{cccccccccc}
t_0 0 L t_0 & t_0 1 L t_1 & t_0 2 L t_2 & t_0 3 L t_0 & t_0 4 L t_1 & t_0 5 L t_2 & t_0 6 L t_0 & t_0 7 L t_1 & t_0 8 L t_2 & t_0 9 L t_0 \\
t_1 0 L t_1 & t_1 1 L t_2 & t_1 2 L t_0 & t_1 3 L t_1 & t_1 4 L t_2 & t_1 5 L t_0 & t_1 6 L t_1 & t_1 7 L t_2 & t_1 8 L t_0 & t_1 9 L t_1 \\
t_2 0 L t_2 & t_2 1 L t_0 & t_2 2 L t_1 & t_2 3 L t_2 & t_2 4 L t_0 & t_2 5 L t_1 & t_2 6 L t_2 & t_2 7 L t_0 & t_2 8 L t_1 & t_2 9 L t_2
\end{array}$$

$$t_0\beta\beta s_{\mathrm{YES}} \qquad t_1\beta\beta s_{\mathrm{NO}} \qquad t_2\beta\beta s_{\mathrm{NO}}$$

This machine is clearly optimally efficient: by the time it has finished reading all the digits, it has worked out the answer.

We can imagine improving the efficiency of a Turing machine in various ways. We could allow it to change the symbol and move the head in a single operation.

We could allow it to move several squares left or right, instead of just one. (However, there are good reasons why the length of a jump should be bounded. The finite number of state-symbol pairs cannot encode infinitely many jump lengths, and physically the distance moved in one clock cycle is bounded because the head cannot move faster than the speed of light.) We could replace the one-dimensional tape with a multi-dimensional array; we could equip the head with its own memory in the form of a stack. We could even allow several communicating heads (with restrictions as for jumps). It can be shown that none of these improvements enlarges the class of computations which can be performed; and, although they speed up computations somewhat, they do not change the definitions of the classes P and NP, to which we turn next.

3.2 P and NP

We say that a deterministic Turing machine *solves* a decision problem P if the following is true. Suppose that the input for P is written (in binary notation) on the tape, and the machine is in the starting state scanning the blank square just to the left of the data. Then it halts when there is only one non-blank square; the head is scanning this square and is in the halting state; and the symbol in the square is 1 if the answer to the decision problem is "yes", or 0 if it is "no".

A non-deterministic Turing machine solves the problem if, with the same initial conditions, there is at least one computational path which leads to the same result.

It follows from the Church–Turing thesis that, if a decision problem has a mechanical or algorithmic solution, then there is a (deterministic) Turing machine which solves it. Now we are interested in how many steps such a Turing machine takes. It follows from our comments that, if we count the number of "elementary" steps taken by our algorithm, the the number of Turing machine steps is not too much greater.

We distinguish between a *problem* X and an *instance* of X. Just knowing that a particular question, specified by 100 bits of data, can be solved in 1000000 Turing machine steps, gives us no information about how hard the general question is: the complexity might be $10000n$, or n^3, or even $2^{2\sqrt{n}}$. Accordingly, we define the *complexity* of a decision problem X to be the function $f = f_X$ defined as follows:

- The *size* of an instance of X is the number of bits of data required to specify that instance.

- $f(n)$ is the smallest integer N such that there exists a deterministic Turing machine which solves any instance of X of size n in at most N steps.

Then we say that $\mathcal{X}$ is *polynomial-time solvable*, or belongs to the class P, if $f_{\mathcal{X}}(n) = O(n^k)$ for some integer k.

> We take the view here that problems in P are those which are "easy" or *tractable*, and problems not in P are "hard" or *intractable*.

Note that this definition refers to the "worst case" of the problem. It may be that a typical problem instance can be solved very quickly, but there are a few recalcitrant instances which take much longer. (Some people argue that an "average case" complexity is more meaningful. It is certainly true that there are many important problems where the average case is much easier than the worst case.)

Note also that the definition says "there exists a Turing machine", that is, "there exists an algorithm". So, to show that a problem $\mathcal{X}$ is polynomial-time solvable, all we have to do is to exhibit an algorithm which will solve the polynomial in a polynomial number of steps. Our earlier arguments show that it is not even necessary to translate the algorithm into a Turing machine; we can be quite informal about the definition of steps. So all the problems in the final section of the preceding chapter are in P. However, to show that a problem is not in P is usually much more difficult: we have to show that there is *no possible algorithm* which can guarantee to solve the problem in a polynomial number of steps.

Analogously, we say that a problem is *non-deterministic polynomial-time solvable*, or belongs to NP, if there is a non-deterministic Turing machine which has an accepting calculation for any positive instance of the problem and takes at most n^k steps for some k, where n is the size of the input.

Since non-deterministic computations are quite hard to think about, we give another interpretation. To specify a computation path of a non-deterministic Turing machine, we have to give some additional information which tells the machine which instruction to execute at each point where an ambiguity arises. We can turn this into a deterministic computation as follows. We give all the required information in advance, so that the machine is presented with both the data for the problem and some additional data forming a "certificate". Now we require that the machine can perform a deterministic computation, using information from the certificate as well as the problem data, and terminate with the answer "yes" precisely in the case where the solution to the problem is "yes".

For example, the problem "Given a graph, does it have a Hamiltonian circuit?" is in NP. The certificate is just the Hamiltonian circuit. You can think about it like this:

- a problem is in P if it can be solved quickly;

- a problem is in NP if a proposed solution can be checked quickly, using the certificate to do the checking. For example, I can quickly convince you that a graph is Hamiltonian, just by showing you a Hamiltonian circuit.

A feature of this definition is that, whereas the negation of a problem in P is also in P (since we can just perform the calculation for the original problem and then negate the final answer in one more step), the negation of a problem in NP is *not* necessarily in NP. Although I can quickly convince you that a graph is Hamiltonian, I will have a much harder job convincing you that a graph is not Hamiltonian!

Exercise 3.2.1 Show that the following decision problem in in NP. You may argue informally; you are not required to construct a Turing machine to solve the problem.

Composite number

Instance: A positive integer n in base 2 notation.

Problem: Is n composite?

Solution To show that the problem is in NP, we have to show that there is a certificate for any positive instance of the problem, such that given the certificate, the correctness of the positive answer can be verified in a polynomial number of steps.

Given that the number n is composite, we take the certificate to be a number m such that $1 < m < n$ and m divides n. The size of the input data is the number of bits necessary to write n in base 2, which is $\lceil \log_2 n + 1 \rceil$. The size of the certificate m is smaller than this, and the division sum can be done in a polynomial number of steps.

Any problem which is in P is in NP: just use the empty certificate. So $\mathsf{P} \subseteq \mathsf{NP}$. Since NP contains many problems (such as the Hamiltonian circuit problem) which are regarded as "hard" (and where no polynomial-time algorithm has ever been found, despite a lot of effort), it is widely believed that $\mathsf{P} \neq \mathsf{NP}$. This is the outstanding open problem of complexity theory.

On 24 May 2000, the Clay Mathematical Institute announced seven prizes, each worth one million U.S. dollars, for the solution of seven of the major problems in contemporary mathematics. The first problem on the list is that of deciding whether $\mathsf{P} \neq \mathsf{NP}$. See the Web page at

`http://www.claymath.org/prize_problems/p_vs_np.htm`

for more information.

3.3 Polynomial transformations

Let $\mathcal{X}$ and $\mathcal{Y}$ be decision problems.

We say that there is a *polynomial transformation* from $\mathcal{X}$ to $\mathcal{Y}$ if, for any instance X of $\mathcal{X}$ of size n, there is a Turing machine which calculates an instance Y of $\mathcal{Y}$ in a number of steps which is bounded by a polynomial in n, and has the property that the answers ("yes" or "no") to the two instances X and Y are the same.

More precisely, we can assume that the Turing machine begins scanning the square just to the left of the input data for the instance X; it writes the input data for the instance Y on the tape, deletes the data for X, and halts scanning the square immediately to the left of the data for Y.

Since the machine only takes a polynomial number of steps, the size of the instance Y is bounded by a polynomial in the size n of instance X.

As usual, we don't have to be so formal in practice. To show the existence of a polynomial transformation from $\mathcal{X}$ to $\mathcal{Y}$, it suffices to give an algorithm to translate any instance of X into an instance of Y with the same solution, and argue informally that the algorithm runs in polynomial time.

As an example, let us consider the two problems HC (Hamiltonian circuit) and TSP (Travelling Salesman Problem) specified as follows:

HC (Hamiltonian circuit)

Instance: A graph G.

Problem: Does G have a Hamiltonian circuit?

TSP (Travelling salesman)

Instance: A weighted complete graph, where weights are positive integers, and a positive integer L.

Problem: Is the length of a shortest travelling salesman tour at most L?

This is the way in which we will specify decision problems.

There is a polynomial transformation from HC to TSP, which we saw in the first chapter. Given a graph G on n vertices, we assign weights to the edges of the complete graph K_n by the rule that $w(e) = 1$ if e is an edge of G, and $w(e) = 2$ otherwise. Then the shortest travelling salesman tour has length (at most) n if and only if G is Hamiltonian. Clearly this transformation of graphs into travelling salesman data can be performed efficiently.

The most important consequence of the definition is the following.

Proposition 3.3.1 *Suppose that there is a polynomial transformation from $\mathcal{X}$ to $\mathcal{Y}$. If $\mathcal{Y}$ is in* P*, then $\mathcal{X}$ is in* P*; and if $\mathcal{Y}$ is in* NP*, then $\mathcal{X}$ is in* NP.

Proof We are given that there is a Turing machine T_1 which transforms an instance X of $\mathcal{X}$ of length n into an instance Y of $\mathcal{Y}$ with the same answer in time at most $p(n)$, where p is a polynomial. The size of Y is at most $q(n)$, where q is polynomial, as we remarked above.

If $\mathcal{Y}$ is in P, then we are also given that there is a Turing machine T_2 which solves Y in time polynomial in its size, that is, $r(q(n))$, where r is a polynomial. Now let T be the Turing machine which simulates the operation of T_1 until it halts, and then the operation of T_2. (Take T_1 and T_2 to have disjoint sets of states, and then identify the halting state of T_1 with the initial state of T_2.) The resulting machine solves X correctly in time $p(n)+r(q(n))$, which is polynomial in n.

The argument for NP is similar, using a non-deterministic Turing machine in place of T_2 to solve Y.

Intuitively, regarding polynomial-time as "easy", this means:

> If there is a polynomial transformation from $\mathcal{X}$ to $\mathcal{Y}$, then $\mathcal{X}$ is no harder than $\mathcal{Y}$.

Also, we note the following. If $\mathcal{X}$ is in P, and $\mathcal{Y}$ is any problem for which the answer is not always "yes" and not always "no", then there is a polynomial transformation from $\mathcal{X}$ to $\mathcal{Y}$. Simply take two instances Y_0 and Y_1 of $\mathcal{Y}$, one of which has answer "yes" and the other has answer "no". Given any instance X of $\mathcal{X}$, we can solve it in polynomial time and then write on the tape the data for either Y_0 or Y_1 depending on what the answer to X is.

In particular, any non-trivial problem in P has a polynomial transformation to any other.

3.4 Cook's Theorem; NP-completeness

As we have said, we regard problems in P as easy. The class NP contains many problems which are commonly regarded as hard, such as the Travelling Salesman. The remarks at the end of the last section show that the problems in P are all equivalent with respect to polynomial transformation, and form the "easiest" problems in NP.

Cook's Theorem shows that there is another subclass of NP whose members are the "hardest" problems in NP (and again are all equivalent). These problems

are called NP-complete. Among the NP-complete problems are our old favourites HC and TSP, as we will see. Here is the formal definition.

A problem $\mathcal{X}$ is said to be NP-*complete* if

- $\mathcal{X}$ is in NP;
- for any problem $\mathcal{Y}$ in NP, there is a polynomial transformation from $\mathcal{Y}$ to $\mathcal{X}$.

Immediately from the definition, we see that NP-complete problems are the hardest in NP. Indeed, the following holds.

Proposition 3.4.1 *Let $\mathcal{X}$ be any* NP*-complete problem. If there is a polynomial-time algorithm for $\mathcal{X}$, then* $\mathsf{P} = \mathsf{NP}$.

For, by definition, for any problem $\mathcal{Y}$ in NP, there is a polynomial transformation from $\mathcal{Y}$ to $\mathcal{X}$; so, if $\mathcal{X}$ is in P, then so is $\mathcal{Y}$. So finding an efficient algorithm for any one such problem would win the million dollars from the Clay Mathematical Institute!

How do we know that NP-complete exist? This is the content of Cook's Theorem. First, we define the problem Cook considered, the satisfiability of Boolean formulae, or SAT for short.

A Boolean formula is one built up from Boolean variables $x_1, \ldots, x_n$ (each of which can take the values true or false, by means of connectives: $\neg$ (negation, "not"), $\wedge$ (conjunction, "and"), and $\vee$ (disjunction, "or"). The connectives are evaluated according to the usual truth tables.

A Boolean formula F is said to be in *conjunctive normal form* (for short, CNF) if it has the form

$$F = C_1 \wedge C_2 \wedge \cdots \wedge C_m,$$

where each *clause* C_i has the form

$$C_i = (u_{i1} \vee u_{i2} \vee \cdots \vee u_{ir_i}),$$

and each *literal* u_{ij} is either a variable $x_{m_{ij}}$ or a negated variable $(\neg x_{m_{ij}})$ (which we write for short as $\overline{x}_{m_{ij}}$)). It is a theorem of Boolean logic that any formula is equivalent to one in conjunctive normal form.

An assignment of values to the Boolean variables is said to be a *satisfying assignment* for a formula F if the truth value of F with this assignment is true. If F is in CNF, then in a satisfying assignment, each clause C_i must get the value true; so at least one literal u_{ij} in each clause must get the value true (which means that $x_{m_{ij}}$ takes the value true if $u_{ij} = x_{m_{ij}}$, or the value false if $u_{ij} = \overline{x}_{m_{ij}}$)).

For example, the formula

$$(x_1 \vee x_2 \vee x_3) \wedge (\overline{x}_2 \wedge x_4)$$

is satisfied by the assignment

$$x_1 = \mathsf{false}, \quad x_2 = \mathsf{true}, \quad x_3 = \mathsf{false}, \quad x_4 = \mathsf{true}$$

(and indeed by many other assignments).

Exercise 3.4.1 For each of the following Boolean formulae, (i) is it in conjunctive normal form, (b) is it satisfiable? [Recall that $\overline{x}$ means the negation of x.]

(a) $F = x_1 \wedge x_3 \wedge x_{26} \wedge \overline{x_1}$.

(b) $G = (x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee \overline{x}_2 \vee x_3) \wedge (\overline{x}_1 \vee x_2 \vee x_3) \wedge (\overline{x}_1 \vee \overline{x}_2 \vee \overline{x}_4) \wedge (\overline{x_3} \vee x_4) \wedge (x_3 \vee x_4) \wedge (\overline{x}_3 \vee \overline{x}_4)$.

Solution

(a) This formula is in conjunctive normal form. (Note that each clause contains just one literal; this is permitted.) It is not satisfiable since no assignment of truth values can give both x_1 and $\overline{x}_1$ the value true.

(b) This formula is in conjunctive normal form; and it is not satisfiable. For the only way of satisfying the last three clauses is to put $x_3 = \mathsf{false}$ and $x_4 = \mathsf{true}$; then to satisfy the first four clauses requires that each of the four combinations $(x_1 \vee x_2)$, $(\overline{x}_1 \vee x_2)$, $(x_1 \vee \overline{x}_2)$ and $(\overline{x}_1 \vee \overline{x}_2)$ must be satisfied, which is clearly not possible.

Now the satisfiability problem is as follows.

SAT (Satisfiability of Boolean formula)

Instance: A Boolean formula F in CNF.

Problem: Does F have a satisfying assignment?

Theorem 3.4.2 (Cook's Theorem) *The problem* SAT *is* NP*-complete.*

Proof This is not a complete proof, more an illustration of what is required. We have to show two things: that SAT is in NP; and that any problem in NP has a polynomial transformation to SAT.

The first statement is easy. A suitable certificate for a satisfiable formula is just a satisfying assignment. If we know the assignment, we can quickly check that at least one literal in each clause is true.

It is the second statement that requires the work, because we have to start with any problem X in NP. All we know about such a problem is that, if X is any positive instance and C a certificate for X, then there is a Turing machine which begins with the data for X and the certificate C and reaches the accepting state after a polynomial number of steps. What we have to do is to encode the action of this Turing machine into a CNF formula. The formula will have clauses describing the configuration of the tape and the state of the head at any time, guaranteeing that the machine operates correctly, that it starts with the correct data, and that it finishes in the accepting state.

We will illustrate with a very simple example, using the following problem. The input consists of an unknown number n of ones on consecutive tape squares, and the problem is to decide whether n is even. This problem is actually in P, so no certificate is required, but the general principle is the same as for any problem in NP. A Turing machine to solve the problem could consist of the following quadruples:

$$s_0\beta R s_1$$
$$s_1 1\beta s_4$$
$$s_4\beta R s_2$$
$$s_2 1\beta s_3$$
$$s_3\beta R s_1$$
$$s_1\beta 1 s_5$$
$$s_2\beta 0 s_5$$

The machine moves right, erasing ones and alternating between states s_1 and s_2. When it reaches a blank square, it writes 1 or 0 according as the number of ones it has passed is even or odd, and enters the halting state s_5. So it terminates scanning a 1 if and only if n is even. We also see that the machine takes $2n+2$ steps if the input contains n ones.

Consider the case $n = 4$ for illustration. The program terminates in ten steps, so we can be sure that if it starts on tape square 0 then it cannot reach any square outside the range from -10 to $+10$. The propositional variables we use fall into three groups:

$x(i,s_j)$ will indicate that at time i the head is in state s_j, for $0 \le i \le 10$ and $0 \le j \le 5$.

$y(i,j)$ will indicate that at time i the head is scanning square j, for $0 \leq i \leq 10$ and $-10 \leq j \leq 10$.

$z(i,j,a_k)$ will indicate that at time i, square j has symbol a_k written in it, for $0 \leq i \leq 10$, $-10 \leq j \leq 10$ and $k = 0,1,2$ (where $a_0 = \beta$, $a_1 = 0$, and $a_2 = 1$.

The clauses of the formula reflect the correct action of the Turing machine, together with its initial and final configuration. They can be divided into six clauses, as follows.

First group: These express the fact that each time the head is in exactly one state. They are of two types:

$$x(i,s_0) \vee x(i,s_1) \vee \cdots \vee x(i,s_5)$$

is true if the head is in at least one state, and

$$\overline{x}(i,s_j) \vee \overline{x}(i,s_k)$$

for $k \neq j$, is true if the head is not both in state s_j and state s_k.We require these for $0 \leq i \leq 10$ and, for the second type, $0 \leq j,k \leq 5$.

Second group: These express the fact that, at each time, the head is scanning exactly one square. They are constructed like the first group but using the y variables.

Third group: These express the fact that each square contains only one symbol at any given time. Again similar, using the z variables.

Fourth group: These describe the initial configuration. Each clause consists of only a single literal. We include $x(0,s_0)$, $y(0,0)$, $z(0,i,1)$ for $i = 1,2,3,4$, and $z(0,i,\beta)$ for the other values of i.

Fifth group: These describe the operation of the machine. Note that a clause

$$\overline{x} \vee \overline{y} \vee z$$

is equivalent to the implication

$$(x \wedge y) \to z$$

which holds unless x and y are true and z false. Now we translate each machine instruction into several types of clauses. Thus, $s_i a_j a_k s_l$ becomes

$$\overline{x}(t,s_i) \vee \overline{y}(t,u) \vee \overline{z}(t,u,a_j) \vee p;$$

there are three such clauses, one having $p = z(t+1,u,a_k)$ (this will say that the machine writes a_k), one with $p = x(t+1,s_l)$ (this will say that the state changes to s_l), and one with $p = y(t+1,u)$ (to say that the head does not move). For the instruction $s_ia_jLs_l$, we have the above clauses with $p = y(t+1,u-1)$ in place of the last value, and the first one involving a_k deleted; for $s_ia_jRs_l$, use $p = y(t+1,u+1)$ instead. There are two or three clauses for each instruction, each value of t with $0 \le t \le 9$, and each value of u with $-9 \le u \le 10$. Unfortunately we are not finished yet: we need to say that squares not being scanned don't change their content. This can be done by clauses of the form

$$y(t,u) \vee \bar{z}(t,u,a_k) \vee z(t+1,u,a_k).$$

Sixth group: These assert that the head terminates scanning a square bearing the symbol 1 and in the halt state s_5. We can take a clause

$$\bar{y}(10,j) \vee z(10,j,1)$$

for $-10 \le j \le 10$, and a single clause $x(10,s_5)$.

Some thought shows that a satisfying assignment for the resulting conjunction of clauses exists if and only if the Turing machine accepts the given input.

It is fairly clear that this pattern will work for any problem in NP.

It is interesting to stop and think about what has been done here. We have shown that, given the description of a Turing machine and its input (all written on a tape), there is another Turing machine which takes this tape as input and produces as output a logical formula which is satisfiable if and only if the first Turing machine accepts its input!

3.5 Examples

In order to show that a problem $\mathcal{X}$ is NP-complete, we have to show two things:

- $\mathcal{X}$ is in NP; and
- there is a polynomial transformation from a known NP-complete problem to $\mathcal{X}$.

Usually the first step is easy. To begin with, the only example of an NP-complete problem which we can use in the second step is SAT, by Cook's Theorem. But, as we progress, we increase our stock of NP-complete problems, and this step becomes easier. In this section, we give a few examples of such proofs.

3-SAT The problem 3-SAT is a special case of SAT, where we consider only formulae in which every clause contains *exactly three* literals. Clearly a special case is no more difficult than the general case; we show that it is no easier either!

Theorem 3.5.1 *There is a polynomial transformation from* SAT *to* 3-SAT. *Hence* 3-SAT *is* NP*-complete.*

Proof We have to take a logical formula F which is a conjunction of arbitrary clauses, and produce a formula F' which is a conjunction of clauses each involving only three literals, which is satisfiable if and only if F is.

We translate the clauses of F one at a time.If we have a clause $u \vee v \vee w$ which already involves three literals, we can leave it as it is.

For a clause $u \vee v$ with only two literals, we take a new variable z (not occurring anywhere else in the formula), and build the two clauses $u \vee v \vee z$ and $u \vee v \vee \bar{z}$. Now, no matter what value is assigned to z, the corresponding literal in one clause will be false, so $u \vee v$ must be true if the clause is to be satisfied.

Similarly, for a single literal u, take two new variables z_1 and z_2, and form the four clauses $u \vee z_1 \vee z_2$, $u \vee z_1 \vee \bar{z}_2$, $u \vee \bar{z}_1 \vee z_2$, and $u \vee \bar{z}_1 \vee \bar{z}_2$.

Suppose finally that we have a clause with more than three literals, say $u_1 \vee \cdots \vee u_k$, with $k > 3$. Take $k-3$ new variables $z_1, \ldots, z_{k-3}$, and form the clauses

$$\begin{array}{lll} u_1 \vee u_2 \vee z_1, & u_3 \vee \bar{z}_1 \vee z_2, & u_4 \vee \bar{z}_2 \vee z_3, \\ \ldots & u_{k-2} \vee \bar{z}_{k-4} \vee z_{k-3}, & u_{k-1} \vee u_k \vee \bar{z}_{k-3}. \end{array}$$

For example, if $k = 4$, we replace the clause $u_1 \vee u_2 \vee u_3 \vee u_4$ by $u_1 \vee u_2 \vee z$ and $u_3 \vee u_4 \vee \bar{z}$. We have to show that an assignment satisfies the original clause if and only if there is a value for z such that the two new clauses are satisfied.

Suppose first that an assignment satisfies $u_1 \vee u_2 \vee u_3 \vee u_4$. If u_1 or u_2 is given the value true, then set z to be false; if u_3 or u_4 is true, set z to be true.

Conversely, suppose that both of the three-literal clauses are satisfied by an assignment. If we have set z to be true, then one of u_3 and u_4 must be true to satisfy the second clause; if we set z to be false, then one of u_1 and u_2 must be true to satisfy the first clause.

A similar argument works for larger k (see the next exercise).

Exercise 3.5.1 Check that all these clauses are satisfied by an assignment if and only if the original clause is satisfied by the assignment of values to the us.

Solution In the general case, the parts of clauses involving the new variables can be written as

$$z_1, z_1 \to z_2, \ldots, z_{k-4} \to z_{k-3}, \bar{z}_{k-3}.$$

In this form it is clear that no assignment can make all these subformulae true. But, by setting $z_1, \ldots, z_i$ to be true and $z_{i+1}, \ldots, z_{k-3}$ to be false, we satisfy all of them except $z_i \to z_{i+1}$. Similarly, we can satisfy all but the first (by putting all variables false), or all but the last (by putting them all true).

Now suppose we have an assignment of truth values satisfying the k-literal clause. Then some u_i is satisfied. We can now assign values to the zs to satisfy all the fragments except the one in the 3-literal clause containing u_i. So all the 3-literal clauses are satisfied.

Conversely, suppose that all the 3-literal clauses are satisfied. Then as we noted, not all the fragments involving the zs can be true, so at least one u_i must be true, and so the original k-literal clause is satisfied.

Why do we take 3-SAT here? It can be shown that the problem 2-SAT (satisfiability of Boolean formulae in CNF with two literals in each clause) is in P, that is, it can be solved efficiently. You may wish to try to prove this – it is not completely straightforward, and not immediately relevant, so we will not give the proof.

Vertex cover Let $G = (V, E)$ be a graph. A *vertex cover* of G is a set C of vertices with the property that every edge of G is incident with some vertex in C. We are interested in the size of the smallest vertex cover.

Exercise 3.5.2 What is the smallest size of a vertex-cover of the Petersen graph (Figure 1.3)?

Solution A vertex cover for a pentagon must contain at least three vertices. So we require at least three vertices from both the outer pentagon and the inner pentagram; so no smaller vertex cover than 6 is possible. But there is a vertex cover of size 6, given by the circled vertices in Figure 3.2.

Here is a formulation of the decision problem associated with this question.

VC (vertex cover)

Instance: A graph G, and a positive integer k.

Problem: Does G have a vertex cover of size at most k?

This problem is in NP, since the list of vertices in a vertex cover is a certificate for a positive instance.

Theorem 3.5.2 *There is a polynomial transformation from* 3-SAT *to* VC. *Hence* VC *is* NP*-complete.*

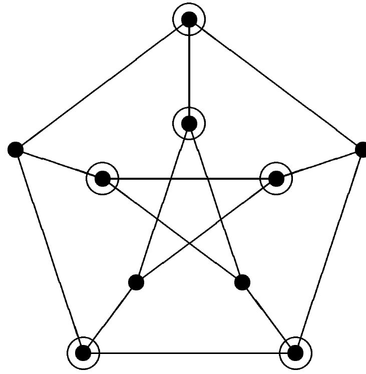


Figure 3.2: A vertex cover

Proof Take an instance of 3-SAT, a formula F in n Boolean variables which is a conjunction of m clauses each containing three literals.We construct a graph $G = (V,E)$ such that G has a vertex-cover of size $n+2m$ or less if and only if the formula is satisfiable.

The graph G has $2n+3m$ vertices, as follows:

- For each variable x_i, there are two vertices called x_i and $\overline{x}_i$, joined by an edge. We call these the *truth-setting vertices*.
- For each clause $u \vee v \vee w$, there are three vertices called u, v, w, any pair joined by an edge (that is, forming a triangle). We call these the *satisfaction-testing vertices*.

Now the name of each satisfaction-testing vertex also occurs as the name of a truth-setting vertex; we join these vertices. For example, if we have the clause $\overline{x}_1 \vee x_2 \vee \overline{x}_4$, we have a triangle of satisfaction-testing vertices named $\overline{x}_1$, x_2 and $\overline{x}_4$, and each is joined to exactly one truth-setting vertex (the one with the same name), as shown in Figure 3.3.

Now any vertex-cover must obviously contain at least one of each pair $\{x, \overline{x}\}$ of truth-setting vertices, and at least two of each triangle of satisfaction-testing vertices. So the size of a vertex-cover is at least $n+2m$. We have to show that there is a vertex-cover of this size if and only if the formula is satisfiable.

Suppose first that there is a satisfying assignment. This tells us how to choose one of each pair of truth-setting vertices: choose the one of x and $\overline{x}$ which is true. Now each clause contains a true literal, so one of the vertices in each triangle of

satisfaction-testing vertices is joined to one of the truth-setting vertices which we have chosen; we put the other two into our vertex-cover, getting $n+2m$ vertices altogether.

Conversely, suppose that there is a vertex-cover C with $n+2m$ vertices. The argument shows that it must contain one of each pair of truth-setting vertices and two of each triangle of satisfaction-testing vertices. We now define an assignment by putting x or $\overline{x}$ true depending on which is in C. Now in each clause $u \vee v \vee w$, only two of u, v, w are in C; so the third must be joined to a truth-setting vertex in C, which means it is assigned the value true, and so the clause is true. Thus, F is satisfied.

For example, the formula

$$(x_1 \vee x_2 \vee x_3) \wedge (\overline{x}_1 \vee x_2 \vee \overline{x}_4)$$

translates into the graph shown in Figure 3.3, and the satisfying assignment

$$x_1 = \mathsf{true}, \quad x_2 = \mathsf{false}, \quad x_3 = \mathsf{false}, \quad x_4 = \mathsf{false}$$

translates into the vertex cover formed by the circled vertices.

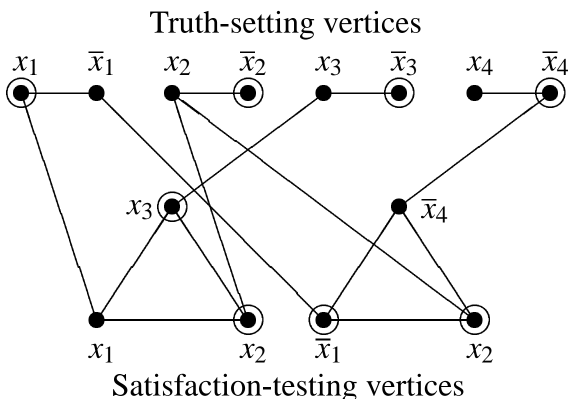


Figure 3.3: A vertex cover

Hamiltonian circuit We defined the problem HC (Hamiltonian circuit) earlier.

Theorem 3.5.3 *There is a polynomial transformation from* VC *to* HC. *Hence* HC *is* NP*-complete.*

The proof is similar to but a bit more complicated than the preceding one; we refer to Garey and Johnson for details.

We already saw that there is a polynomial transformation from HC to TSP (the Travelling Salesman problem). So TSP is also NP-complete.

Chapter 4

Other complexity classes

The complexity classes P and NP are the most important but not the only ones that have been considered. We look briefly at some of the others; this will be more informal, but the classes involving randomised algorithms or approximation algorithms lead on naturally to the second part of the course.

4.1 Harder problems

First, we introduce the classes PSPACE and EXPTIME, which contain problems thought to be even harder than NP-complete problems.

A problem $\mathcal{X}$ belongs to the class PSPACE if any instance of size n can be solved by a Turing machine in which, while the program is running, the head moves at most $O(n^k)$ steps away from its initial location. Informally, a problem is in the class PSPACE if it can be solved using only a polynomial amount of memory space.

Typical hard problems in PSPACE are finding a winning strategy in positional games like "generalised chess" played on an $n \times n$ board. We may have to look many moves ahead, but we only have to remember the configuration of the board. (Finding a winning strategy for ordinary chess is a single finite problem, though a very large one; we can't talk about the complexity unless have have a whole family of arbitrarily large problems.)

Theorem 4.1.1 NP $\subseteq$ PSPACE.

Needless to say, it is believed that these two classes are unequal but nobody can prove it! We will prove that P $\subseteq$ PSPACE and treat NP more informally.

The reason that P $\subseteq$ PSPACE is simply that, if a Turing machine runs for a polynomial number of steps, it obviously cannot move further than a polynomial

distance from its starting square. In general, the space complexity of a problem cannot be larger than the time complexity.

For NP, we argue as follows. Suppose that there is a nondeterministic Turing machine which solves an instance of the problem in polynomial time. Remember that there is at least one computational path which leads to the successful result. We can simulate this Turing machine by a deterministic machine which tries all the possible computational paths; we do not need any more memory space, except for enough to save a copy of the input data. (There is a difficulty, since there might be a computational path which doesn't terminate, or which takes a very long time, which the machine tries before reaching the successful path. So we have to equip the machine with a "clock" which tells it to abandon a particular attempt if it has not succeeded in a fixed polynomial number of steps.)

A problem $\mathcal{X}$ is in the class EXPTIME if an instance of the problem of size n can be solved by some Turing machine in time $O(2^{n^k})$ for some k. That is, EXPTIME really means "time which is the exponential of a polynomial".

Theorem 4.1.2 PSPACE $\subseteq$ EXPTIME.

To prove this, suppose that $\mathcal{X}$ is a problem in the class PSPACE, so that an instance of size n can be solved by a Turing machine whose head moves no more than $p(n)$ steps from its original position. Thus the only tape squares which are used are those in the range from $-p(n)$ to $p(n)$. Suppose that the number of tape symbols is k and the number of machine states is r.

We claim that the total number of possible configurations of machine and tape is at most

$$(2p(n)+1)rk^{2p(n)+1}.$$

For there are at most $2p(n)+1$ positions for the head, and at most r states; then each tape square has one of k symbols written in it, so the total number of strings that could be written on the tape is at most $k^{2p(n)+1}$.

Now we claim that the time taken by the computation is not greater than this number. For otherwise, some configuration of position and state of the head and string written on the tape must occur twice. But then, the second time, the computation will proceed exactly as it did on the first occasion, so the machine is stuck in a loop and will never terminate. This contradicts the assumption that the machine really does solve the problem!

Now $(2p(n)+1)rk^{2p(n)+1}$ is certainly bounded by the exponential of a polynomial: we have $2p(n)+1 < 2^{2p(n)+1}$, and so

$$(2p(n)+1)rk^{2p(n)+1} < 2^{(1+\log_2 k)(2p(n)+1)+\log_2 r}.$$

So the class $\mathcal{X}$ belongs to EXPTIME, as claimed.

4.2 Counting problems

Many decision problems can be extended in a natural way to counting problems. Here are a couple of examples.

SAT (Satisfiability)

Instance: A Boolean formula, in conjunctive normal form

Decision problem: Is there an assignment of truth values to the variables which satisfies the formula?

Counting problem: How many such satisfying assignments are there?

HC (Hamiltonian circuit)

Instance: A graph G.

Decision problem: Does G have a Hamiltonian circuit?

Counting problem: How many Hamiltonian circuits does G have?

In each case, the counting problem is harder than the decision problem. If we could solve the counting problem, we could immediately solve the decision problem by just testing whether the answer is zero or non-zero.

Sometimes, an easy decision problem is associated with an easy counting problem. We have seen that a graph G has a spanning tree if and only if G is connected (and this can be decided quickly). The number of spanning trees can be computed by evaluating a determinant (which can also be done quickly), according to *Kirchhoff's Matrix-Tree Theorem*, which we state without proof:

Theorem 4.2.1 *Let A be the adjacency matrix of a simple graph G. Let D be the diagonal matrix whose (i,i) entry is the valency of the ith vertex (the number of edges on which it lies. Let X be the matrix obtained from $D-A$ by deleting the first row and column. Then* $\det(X)$ *is equal to the number of spanning trees of G.*

For example, let $G = K_3$, the complete graph on 3 vertices. Then

$$D-A = \begin{pmatrix} 2 & -1 & -1 \\ -1 & 2 & -1 \\ -1 & -1 & 2 \end{pmatrix},$$

and

$$\det(X) = \det\begin{pmatrix} 2 & -1 \\ -1 & 2 \end{pmatrix} = 3,$$

so G has three spanning trees.

Sometimes, however, an easy decision problem is associated with a hard counting problem, as in the case of the permanent of a matrix with non-negative entries. As we saw, it is easy to decide whether the permanent is non-zero, but hard to evaluate it. In the case of Hall's Marriage Problem, the permanent counts the number of ways in which the compatible marriages can be arranged.

The natural complexity class for counting problems is #P (read "number-P"). It consists of problems which can be solved in polynomial time on a non-deterministic Turing machine, and we are asked to count the number of accepting computations. Alternatively, we have to count the number of "certificates" for a positive solution. The counting problems for **SAT** and **HC** above are examples. We do not discuss this further.

4.3 Parallel algorithms

Some problems can be solved much faster using a modified Turing machine containing a large number of heads, equipped with the capacity to communicate with one another and all using the same tape. We think of these heads as processors working in parallel and sharing memory.

A class $\mathcal{X}$ of problems is said to belong to NC if a problem in $\mathcal{X}$ of sise n can be solved in time $O((\log n)^k)$ by a machine with $O(n^k)$ heads, for some positive integer k. Some hard problems can be shown to lie in this class. The letters NC stand for "Nick's class", after Nick Pippinger who invented it. We do not discuss it further.

4.4 Randomised algorithms

So far we have considered time and space (that is, time to perform the computation and memory space used) as the resources which measure the complexity of a calculation. We now consider a different kind of resource: *randomness*.

A deterministic computer cannot generate a random number. The "random numbers" built into most programming systems are the result of applying some computation to a "seed", which may be the reading of the computer's clock. The resulting numbers vary in an apparently unpredictable way but each is uniquely determined by the one before. They are more properly called "pseudo-random numbers". The more complicated the calculation is, the more satisfactory the result will be. However, from a theoretical point of view it is better to regard randomness as a resource, which is paid for in time by a pseudo-random number generator.

Thus, we regard a Turing machine executing a randomised algorithm as being

equipped with a source of random bits, which in practice will be supplied by a pseudo-random number generator. The machine can have additional instructions which cause it to request a random bit, and the subsequent state of the head will depend on the value of this bit.

In the course of a polynomial-time calculation, of course, the machine can only ask for polynomially many random bits.

For example, a non-deterministic Turing machine can be made into a randomised Turing machine in the following way. Whenever the machine has a choice of instructions to follow, it asks for enough random bits to allow it to make the choice.

Now we define the class RP of problems which can be solved in "random polynomial time". First, for comparison, we repeat the definition of NP. A class $\mathcal{X}$ of decision problems belongs to NP if there is a non-deterministic Turing machine which runs in time polynomial in the input size such that

- for any positive instance of $\mathcal{X}$, there is at least one computation path which accepts the instance;
- for any negative instance, there is no accepting computation path,

Now we say that $\mathcal{X}$ belongs to RP if there is a non-deterministic Turing machine which runs in time polynomial in the input size such that

- for any positive instance of $\mathcal{X}$, at least half of all the computation paths accept the instance;
- for any negative instance, there is no accepting computation path,

It is clear that this is a stricter requirement than the definition of NP. Moreover, a deterministic Turing machine can be regarded as a non-deterministic machine in which there is only one computation path for any problem instance. So we have:

Theorem 4.4.1 $\mathsf{P} \subseteq \mathsf{RP} \subseteq \mathsf{NP}$.

There is nothing special about the probability $1/2$ in the definition: any positive constant would do, for the following reason:

Theorem 4.4.2 *Suppose that a problem can be solved in time T by a randomised Turing machine with probability c, where $0 < c < 1$. Then it can be solved in time $T\lceil \log\varepsilon/\log(1-c)\rceil$ with probability $1-\varepsilon$, for any ε.*

For $1-c$ is the probability that we do not obtain a result in time T. If we repeat the calculation n times, using independent random bits, the probability that

we do not obtain a result is $(1-c)^n$. We can make this smaller than ε by choosing n so that $(1-c)^n < \varepsilon$, or $n\log(1-c) < \log\varepsilon$; in other words,

$$n > \log\varepsilon/\log(1-c).$$

(Remember that $\log(1-c)$ is negative, since $1-c<1$.)

So, if a randomised Turing machine solves a problem with probability greater than $1/2$, then in just 100 repetitions of the calculation the chance of not having a result is less than $(1/2)^{100}$, which is smaller than the probability of a hardware error in the computer. (Of course, this assumes that the random bits are independent, which will not be the case if we use a pseudo-random number generator.)

What this means is that, if we are prepared to accept a minute probability of error, then the class RP is almost as satisfactory as P. In practical terms, as we will see, algorithms involving some random choices can give us a very powerful method to attack hard problems.

The most famous example of a randomized algorithm is a primality test developed by Solovay and Strassen and by Rabin. The algorithm (which depends on some advanced number theory, so we do not give details here) has the property that it answers "composite" or "probably prime": if the input is prime then the answer is "probably prime", if it is composite then the answer is "composite" with probability at least $1/2$. So, in terms of the definition, the problem "Is n composite?" is in RP. If the number n gets the answer "probably prime" in 100 independent runs of the algorithm, then we are justified in assuming that n is prime for practical purposes such as cryptography. But just one answer "composite" is enough to convince us.

The development of randomised algorithms does provide an interesting problem for the philosophy of mathematics. It is now possible to write down a number n with several hundred digits and state "n is probably prime"; if the computation has been done properly, our confidence in this statement may be greater than our confidence in a long and complex proof of a theorem proved by a mathematician. So can such a statement be a mathematical truth?

4.5 Approximation algorithms

We saw an example of an algorithm (the twice-round-the-tree algorithm for the Travelling Salesman Problem) which gives an answer which is at most twice the optimum value. Several other examples of such algorithms are known.

However, for what is technically known as an "approximation algorithm", we ask for more. We require that you can get within a factor arbitrarily close to 1 of the optimum value if you are prepared to spend enough time doing it.

Let us suppose that we have a problem with input data of size n, where we are required to compute some numerical function (such as a counting problem or the Travelling Salesman Problem). Let K be the true answer to the problem. We say that there is a *polynomial-time approximation algorithm* for the problem if, given any positive number ε, there is a Turing machine which computes a number k satisfying

- $K/(1+\varepsilon) < k < K(1+\varepsilon)$;
- the number of steps is bounded by a polynomial in n and $\log(1/\varepsilon)$.

We say that the algorithm estimates K to within ε if the first condition holds. (This agrees with the usage "to within 1%", for example.)

We have to explain why we use $\log(1/\varepsilon)$ here. Suppose that we have computed the answer, and we are required to improve our accuracy by one decimal place. That means, we have to reduce the possible error to one-tenth of its previous value, so $\log(1/\varepsilon)$ increases by a constant amount. If, for example, the time was a linear function of $\log(1/\varepsilon)$, then this would increase the time taken by a constant amount. If the time was proportional to $1/\varepsilon$ instead, then to get one extra decimal place would take ten times as long! In brief, we require that the time taken grows as a polynomial in the size of the input data and in the number of significant figures required in the answer.

We can combine the last two ideas and define a *randomised approximation algorithm*. Here, we use a randomised algorithm (one which makes choices between computation paths based on random bits). We prescribe both the accuracy of the computed answer (the number ε above) and the probability that the algorithm fails to meet the requirements (another positive number δ). If the running time is bounded by a polynomial in n (the size of the input data), $\log(1/\varepsilon)$, and $\log(1/\delta)$, we call the algorithm *fully polynomial*.

If there is an randomised algorithm which estimates K to within ε with probability at least $3/4$, say, then we can estimate K to within ε with arbitrarily high probability $1-\delta$, by the following simple trick: repeat the algorithm N times, where $N = 1 + 12\lceil\log(1/\delta)\rceil$, and take the median of the resulting N estimates. This depends on the following result about probability theory, of which we omit the proof (which just involves estimates for binomial coefficients):

Proposition 4.5.1 *Let $X_1,\ldots,X_N$ be independent random variables having the same distribution, where N is odd. Let a and b be real numbers. Suppose that*

$$P(a \le X_i \le b) \ge \frac{3}{4}$$

for all i. Then, if X denotes the median of $X_1,\ldots,X_N$, we have

$$P(a \le X \le b) \ge 1 - \mathrm{e}^{-N/12}.$$

Now, in our case, to make this probability greater than $1-\delta$, we just require $e^{-N/12} < \delta$, or $N \geq 12\log(1/\delta)$. The added 1 is to make N odd, so that the median is defined.

So we can give a simpler definition: a randomised approximation algorithm for a number K is *fully polynomial* if, for any positive number ε, it approximates K to within ε with probability at least $3/4$ in time polynomial in n and $\log(1/\varepsilon)$.

For example, consider the counting problem associated with **SAT**: we are given a Boolean formula F in conjunctive normal form, and we are asked to count the number of satisfying assignments. If F is a formula in n variables, then there are altogether 2^n assignments of truth values, so it will take exponentially long to try them all and count the successes.

We could proceed by *sampling*. In other words, choose a large enough integer N, and then choose N assignments of values at random; count the proportion M of these assignments which make F true. Provided that M is not too small, we would guess that about M/N of all assignments are satisfying assignments, and estimate the total number of satisfying assignments as $(M/N)2^n$. It can be shown that, if the actual number of satisfying assignments is $\mu \cdot 2^n$, and if we choose N large enough (precisely, $N \geq 4\log(2/\delta)/\mu\varepsilon^2$), then we will succeed in estimating the number to within ε with probability at least $1-\delta$.

But if the number of satisfying assignments is rather small, we are likely to find none in our sample. (Remember that there are 2^n assignments and we can only look at a small proportion of them.) Then we will not be able to give an accurate estimate.

There are more advanced sampling methods to get around this problem, but we don't consider them here.

4.6 Quantum computation

This topic is mentioned just for completeness here, though it may well play a more important role in complexity theory in the future. Theoretical models of quantum computers have led to definitions of the complexity class QP (problems which can be solved in a polynomial number of steps on a quantum computer). This class certainly includes problems which are not known to be in P and are thought to be "hard" in terms of classical computation.

The most striking example is the result of Peter Shor who gave a quantum polynomial-time algorithm for the problem of factorising an integer. Since this hard problem lies at the foundations of cryptographic systems (such as RSA) widely used on the internet and in commerce, the impact of an actual quantum computer would be very great indeed. But no serious quantum computer has yet

been constructed.

Further information on quantum computation can be obtained from the Web page

`http://www.theory.caltech.edu/people/preskill/ph229/`

the course information for John Preskill's course on Quantum Computation at Caltech.