11 Reduction in p-time

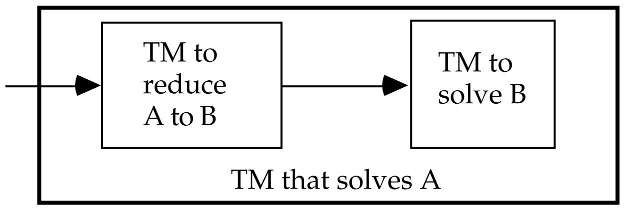
We now use Turing machines to formalise the technique we saw in §8.6 of reducing one problem to another in p-time. This *p-time reduction* gives fast non-deterministic solutions to new yes/no problems from known fast non-deterministic solutions to old ones. We'll study its properties.

11.1 DEFINITION OF P-TIME REDUCTION

Let A, B be any two yes/no problems (not necessarily in NP or even solvable by a Turing machine).

1. Let X be a deterministic[1] Turing machine. We say that X *reduces A to B* if:
   - for every yes-instance w of A, $f_X(w)$ is defined and is a yes-instance of B
   - for every no-instance w of A, $f_X(w)$ is defined and is a no-instance of B.
2. We say that A *reduces to B in polynomial time* (or *p-time)* if there exists a deterministic Turing machine X running in p-time that reduces A to B.
3. We write **A ≤ B** if A reduces to B in polynomial time.
4. We write **A ~ B** if A ≤ B and B ≤ A.

If A≤B then as in §8.6 we can use a fast solution to B to solve A quickly, by first reducing the instance of A to an instance of B of the same 'parity' (yes or no), and then applying the fast solution to B.



11.1 If A≤B, and we are given a solution to B, then we can solve A

[1] We want X to be deterministic because it should be 'genuinely fast', and have an output.

11.1.1. Warning

A≤B implies that, but is <u>not</u> the same as, *any fast solution to B can be used to solve A quickly.*

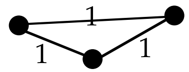
11.2 EXAMPLES

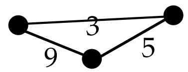
11.2.1 Example

By §8.6, HCP reduces to TSP, and the reduction can easily done by a deterministic Turing machine running in p-time. So HCP ≤ TSP.

11.2.1.1 Warning

<u>Don't</u> try to reduce HCP to TSP in p-time as follows: given an instance G of HCP,

- If G is a yes-instance of HCP, output 

d=3
- If G is a no-instance of HCP, output 

d=1

This involves determining whether G is a yes- or a no-instance. This is hard to do. There is no known p-time way to do it, so this reduction is (probably) not p-time.

A machine reducing a problem A to another, B, need not be able to solve A; and its design does not necessarily take account of whether it is given a yes- or a no-instance of A. It may be very hard to solve A, and yet quite easy to reduce A to B, by making simple changes to the instances in a way that preserves their yes–no parity.

For an apparent exception, see §11.5.2 below.

11.2.2 Example: change of base

Let a,b≥2. We can design a deterministic Turing machine $X_{a,b}$ running in p-time, such that for any number n, if w represents n in base a then $f_{X_{a,b}}(w)$ represents n in base b. <u>Change of base of arithmetic can be done in polynomial time</u>.

For any number a≥2, let C_a be the yes/no problem 'is w the representation in base a of a prime number?'. Then for any a, b ≥ 2, $X_{a,b}$ reduces C_a to C_b. (**Exercise**: check this.) So C_a≤C_b (and by symmetry, C_b≤C_a) for any a, b≥2. So C_a~C_b.

So with respect to the ordering ≤ of difficulty, changing the base makes no difference at all.

Why do we not allow a=1 here — unary notation? Unary is a special case; the exercise below shows why.

11.2.2.1 Exercise

There is a deterministic Turing machine BU that, given the binary representation of a number as input, outputs the unary representation. Show that BU can't have polynomial time complexity. [Hint: how long does BU take to output the answer if the input is the binary representation of n?]

11.3 ≤ IS A PRE-ORDER

We saw that if A≤B then we can use a fast solution to B to solve A quickly. So if A≤B then in effect A is no harder than B. Thus the relation ≤ orders the yes/no problems *by increasing difficulty*. But is ≤ really an ordering at all? In fact it's what's called a *pre-order*: a reflexive, transitive (see §7.1.1) binary relation. Other pre-orders include the ordering on numbers: x≤x for all x, and x≤y≤z implies x≤z. The well-known binary relation *likes(x,y)* may be a pre-order, if everyone likes themselves (so Bob likes Bob, for example), and whenever (say) Bob likes Chris and Chris likes Keith then also Bob likes Keith.

11.3.1 Theorem

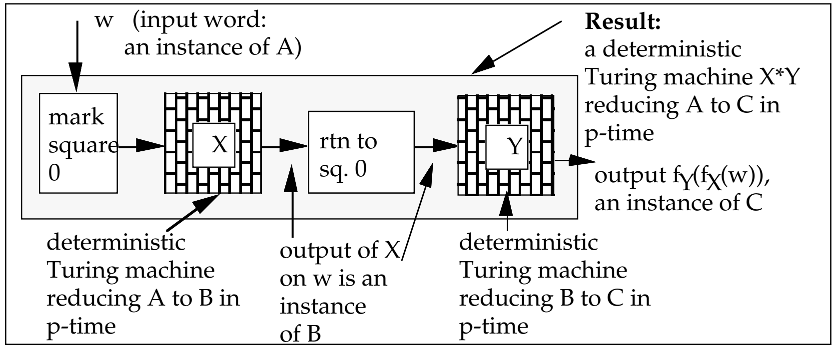
The relation ≤ defined above is a pre-order on the class of yes/no problems.

Proof (i): ≤ is reflexive. To prove this we must show that for any yes/no problem A, A≤A holds. To prove A≤A we must find a deterministic p-time Turing machine reducing A to A.

Let I be a finite alphabet in which all instances of A can be written. Let X be the deterministic Turing machine $(\{q_0\}, I\cup\{\wedge\}, I, q_0, \varnothing, \{q_0\})$. (Cf. Y of fig. 9.1.) X just halts & succeeds without action, so its output is the same as its input. Hence if w is a yes-instance of A then $f_X(w) = w$ is a yes-instance of A; and similarly for no-instances. So X reduces A to A. Moreover, X runs in polynomial time, since $time_X(n) = 0$ for all n.

(ii): ≤ is transitive. Let A, B, C be yes/no problems and assume that A≤B and B≤C. We show that A≤C.

As A≤B, there is a deterministic p-time Turing machine X that reduces A to B. Similarly, as B≤C there is another deterministic p-time Turing machine Y reducing B to C. Then X*Y obtained by running X then Y (Fig. 11.2) is deterministic. (We have to return to square 0 after X because Y expects to begin there.)



11.2 Reducing A to C (so ≤ is transitive)

First we check that X*Y reduces A to C. If w is a yes-instance of A then $f_X(w)$ is a yes-instance of B, and so $f_Y(f_X(w))$ is a yes-instance of C. Similarly if w is a no-instance of A, $f_Y(f_X(w))$ is a no-instance of C. So X*Y reduces A to C, as required.

Now we check that X*Y runs in p-time. Let p(n), q(n) be polynomials with positive coefficients such that $time_X(n) \le p(n)$ and $time_Y(n) \le q(n)$ for all n≥0. If the input word w to X*Y has length n, then:

1. Marking square 0 takes time 1.
2. Running X on w takes time ≤ p(n).
3. Returning to and unmarking square 0 takes time k, where X's head is in square k when X halts. How big could k be? Well, X's head began in square 0, and moves at most 1 square per move. So k is at most the number of moves X made. But X made at most p(n) moves. So k ≤ p(n).
4. Running Y takes time ≤ q(length of $f_X(w)$). What is the length of $f_X(w)$? As above, X can write at most 1 output symbol per move. So

(a) (length of $f_X(w)$) ≤ (no. of moves of X on input w) ≤ p(n).

But q has positive coefficients, so if n increases then q(n) can't decrease.[2] Thus

(b) q(length of $f_X(w)$) ≤ q(p(n)).

Hence Y takes time ≤ q(p(n)).

The total run time of X*Y is thus at most 1 + p(n) + p(n) + q(p(n)), which works out to a polynomial.

For example, if $p(n) = 2n^2 + n^3$ and $q(n) = 4 + 5n$, then the expression is $1 + 2(2n^2+n^3) + 4 + 5(2n^2+n^3)$, which works out to $5 + 14n^2+7n^3$, a polynomial. QED.

2 E.g., because $\frac{dq}{dn} \ge 0$ if n≥0.

11.3.2 Remarks

- Steps 3 and 4 above are important.
- We cannot just say: the symbol ≤ <u>looks like</u> the usual ordering 1≤2≤3… on numbers (it has a line under it: it is not **<**, but ≤), so therefore A≤A (etc). The symbol ≤ may look like the ordering of numbers, but it has a quite different meaning. <u>To prove that ≤ is reflexive and transitive we must use the definition of ≤</u>. No other way will do.
- To prove that A reduces to A in p-time may seem a silly thing to do. It is not silly. It is just trivial.

11.3.3 Exercise

Show that the relation A~B given by 'A≤B and B≤A' (Definition 11.1) is an equivalence relation (see §7.1.1) on the class of all yes/no problems. (Use the theorem.)

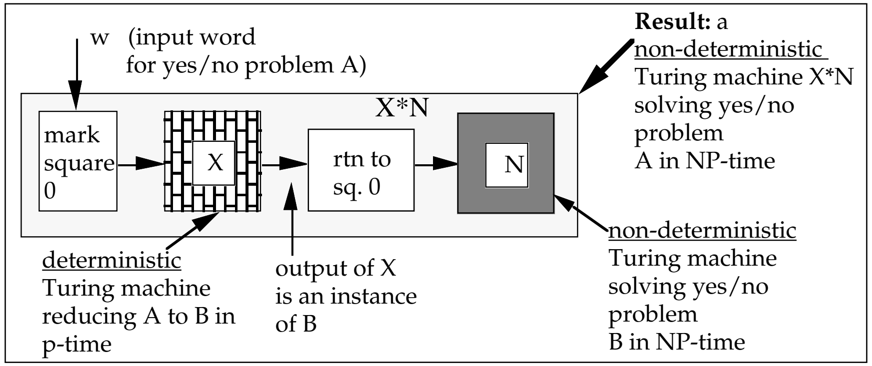
<u>11.4 CLOSURE OF NP UNDER P-TIME REDUCTION</u>

We said that if A≤B then we can use any given fast solution to B to solve A rapidly. What if the fast 'solution' to B is by a p-time NDTM? We'd expect the resulting solution to A also to be a (hopefully p-time) NDTM. Thus we expect that if B is in NP then so is A. Let's check this.

11.4.1 Theorem

Suppose A and B are yes/no problems, and A ≤ B. If B is in NP, then A is also in NP.

Proof: Similar to before. As B is in NP, there is a non-deterministic Turing machine N that solves B. As A≤B, there is a deterministic Turing machine X reducing A to B. We want to join up X to N as in Fig. 11.3 below (cf. Fig 11.2).



11.3 Using solution to B to solve A

Let us call the joined-up machine *X*N*. Then X*N is a NDTM that solves A. For if w is a yes-instance of A, then $f_X(w)$, the input to N, is a yes-instance of B. So N accepts $f_X(w)$. Hence X*N accepts w. Similarly, if w is a no-instance of A then X*N rejects w.

So to show that A is in NP, it is enough to show:

<u>Claim</u>: The non-deterministic TM X*N runs in p-time.

<u>Proof of Claim</u>: We must show $time_{X*N}(n) \le r(n)$ for some polynomial r (and for all n). The count goes much as before. Let w be an instance of A. We calculate how long X*N can run for on input w. Assume $time_X(n) \le p(n)$ and $time_N(n) \le q(n)$ for polynomials p, q with positive coefficients. Let length(w) = n. Then as in §11.3.1:

- Marking square 0 takes a single instruction: time 1.
- X runs for time at most p(n) on input w. X halts with output $f_X(w)$.
- Returning to square 0 takes time at most p(n), as before.
- Then the output $f_X(w)$ of X is given as input to N. As before, $f_X(w)$ has length ≤ p(n), so no run of N is longer than q(p(n)).

So no run of X*N is longer than r(n) = 1+2p(n) + q(p(n)). This is a polynomial in n. Hence X*N runs in p-time. This proves the claim.

As X*N is a p-time Turing machine solving A, A is in NP. QED.

11.4.1.1 Remark

We've showed that the concatenation (joining up) of p-time Turing machines X, Y by sending the output of X into Y as input, gives another p-time Turing machine X*Y. Y can be non-deterministic, in which case so is X*Y.

11.4.2 Corollary

HCP is an NP problem.

Proof TSP is in NP, by §10.2.2. A simple formalisation of §8.6 using Turing machines shows that HCP ≤ TSP. So by the theorem, HCP is in NP also. QED.

We already knew this (§10.2.4), but reduction is useful in other cases — see especially NP-complete problems in §12. Some 1,000 other p-time reductions of problems to NP problems are known.

11.5 THE P-PROBLEMS ARE ≤-EASIEST

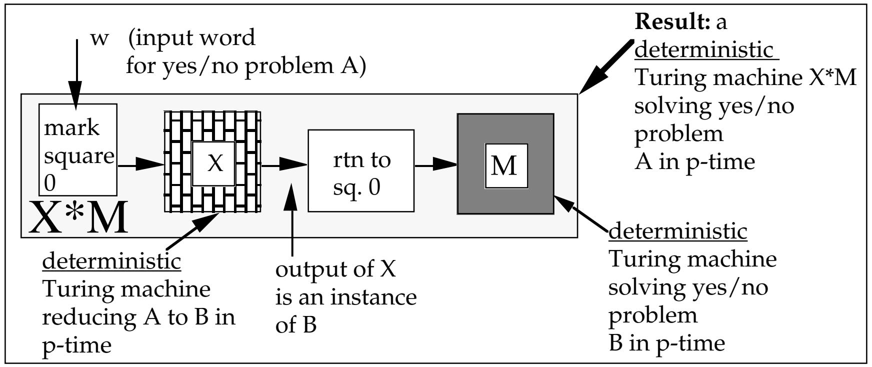
What are the easiest yes/no problems, with respect to our difficulty ordering ≤ of problems? In fact they are those in P — the yes/no problems solvable deterministically in polynomial time.

First, we show that the ≤-easiest problems are in P. This is similar to the proof of 11.4.1, above. Essentially it shows that, like NP, P is closed downwards under ≤.

11.5.1 Theorem

If A is a yes-no problem, and A≤B for all yes-no problems B, then A∈P.

Proof: Choose any B∈P. Then there is a deterministic Turing machine M solving B in p-time. Also, there's a deterministic p-time Turing machine X reducing A to B. Then the machine shown in figure 11.4 below solves A in p-time. This shows that A∈P, as required.



11.4 Solving A in p-time, if A≤B∈P.

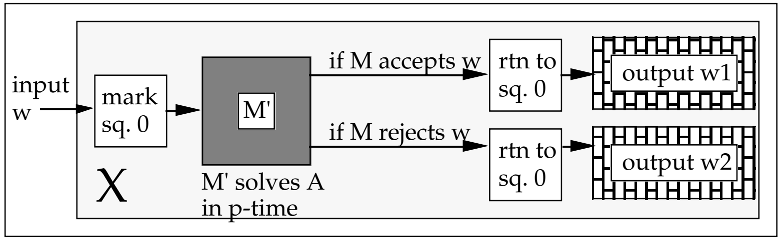
Now we show the other half, that the problems in P are ≤-easiest. This is a new argument for us, and one that seems like a trick.

11.5.2 Theorem

If A is any problem in P, and B is any yes/no problem at all, then A≤B.

Proof: Crudely, the idea is this. We want to show that we can find a fast solution to A if we are allowed to use one for B. But we're told A is solvable in p-time, so we can solve A directly, without using the solution for B! This is crude because we have to show that A reduces to B in p-time, which is not quite the same (see §11.1.1). But the same trick works.

Let M be a deterministic p-time Turing machine solving A. Choose any yes-instance w1 and no-instance w2 of B. Then let X be the machine of figure 11.5:



11.5 Reducing a P-problem to any problem

In the figure, *output w1* is a Turing machine that outputs the word w1 as fixed text (as in the 'hello world' example). The Turing machine *output w2* is similar. X contains a likeness M' of M, slightly modified so that:

- if M halts and succeeds then control passes to *output w1,*
- if M halts and fails then control passes to *output w2.*

We require that M' eventually passes control to the rest of X, so that $f_X(w)$ is defined for any instance w of A. This is true because as M runs in p-time, it always halts (see §9.2.2.3).

Clearly X is deterministic. By counting steps, as in §11.3.1 and §11.4.1, we can check that M runs in p-time.

We show that X reduces A to B. If the input to X is a yes-instance w of A, then M will halt and succeed on w, so X outputs w1, a yes-instance of B. Alternatively, if the input is a no-instance of A, then M halts and fails, and X outputs w2, a no-instance of B. So by Definition 8.2.1(1), X reduces A to B.

So A≤B as required. QED.

11.5.3 Conclusion

A yes-no problem A is in P iff A≤B for all yes-no problems B. Thus, P is indeed the class of ≤-minimal, or easiest, problems.

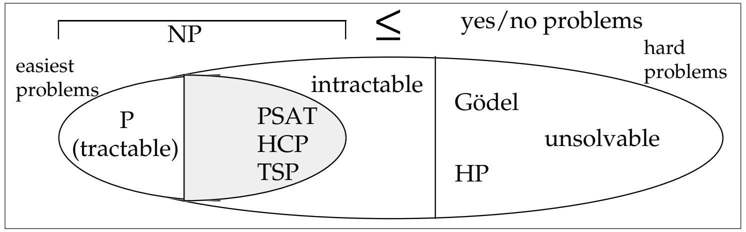
11.5.4 Exercise

Let ~ be the equivalence relation of Definition 11.1(4). Let A be any yes/no problem in P. Show that for any yes/no problem B, A~B iff B ∈ P.

11.6 SUMMARY

We defined the relation ≤ of p-time reduction between yes/no problems. The ordering ≤ is a pre-order, and we think of it as an ordering of difficulty: if A≤B then A is no harder than B. We write A~B if A≤B and B≤A — they are of the same difficulty.

Figure 11.6 sketches the yes/no problems.



11.6 A view of y/n problems (if P≠NP)

Each ~-class consist of all problems of a certain ≤-difficulty (of A, B such that A≤B and B≤A). By §11.5, P is an entire ~-class, the ≤-easiest problems. If A∈NP then any ≤-easier problem is also in NP. So NP is a union of classes: no ~-class overlaps NP on both sides. Of course the shaded area in NP but outside P may be empty, so that PSAT et. al. are all in P! Whether this is so is the question P=NP, which is unsolved.

You will probably survive if you remember that:

- ≤ is reflexive and transitive
- NP is closed downwards with respect to ≤
- P is the class of ≤-minimal problems

The problems in P are the ≤-easiest. Is there is a ≤-hardest problem? We will investigate this — at least within NP — in the final section.

12 NP-completeness

Cheer up — this is the last section. The holiday approaches.

12.1 INTRODUCTION

Is there is a ≤-hardest problem? The answer is not at all obvious, even within NP. Could it be that for any NP-problem, there's always a harder one (still in NP)? If so, there'd be harder and harder problems in NP (with respect to ≤), forming an infinite sequence of increasingly hard problems,

never stopping. Or maybe there are many different ≤-hardest NP-problems, all unrelated by ≤. (Of course if P=NP then our question is irrelevant. But most likely, P≠NP. Ladner showed in 1975 that if P≠NP then there are infinitely many ~-classes within NP.)

In fact this doesn't happen, at least within NP (and also within several other complexity classes which we won't discuss). In a famous paper of 1971, Stephen Cook proved that there are ≤-hardest problems in NP. Such problems are called *NP-complete problems.*

What do we mean by a ≤-hardest problem?

12.1.1 Definition (Cook[3])

A yes/no problem A is said to be *NP-complete* if

(i) A ∈ NP
(ii) B≤A for all problems B ∈ NP.

The class of all NP-complete problems is denoted by *NPC*.

12.1.1.1 Exercise

Show that if A, B are NP-complete then A~B. Show that if A is NP-complete and A~B then B is also NP-complete. So the NP-complete problems form a single ~-class.

But are there any NP-complete problems? Answer: yes.

12.1.2 Theorem (Cook, same paper)

NPC ≠ Ø. In particular, PSAT is NP-complete. We prove it in §12.3.

Some 1,000 examples of NP-complete problems are now known. They include PSAT, the Hamiltonian circuit problem (HCP), the travelling salesman problem (TSP), partition (given a set of integers, can it be divided into two sets with equal sum?) scheduling (can a given set of tasks of varying length be done on two identical machines to meet a given deadline?) and many other problems of great practical importance. See texts, e.g., Garey & Johnson, for a longer list. Non-primality and primality testing are in NP, but are not known to be NP-complete.

12.1.3 Significance of NP-completeness

1. At first sight, TSP and PSAT have little in common. But by Ex. 12.1.1.1, any NP-complete problem is reducible in polynomial time to any other.

[3] S.A. Cook, *The complexity of theorem proving procedures*, Proceedings of Third Annual ACM Symposium on the Theory of Computing, 1971, pp. 151-158.

Hence any solution to TSP can be converted into a solution to PSAT that runs in about the same time — and vice versa.

This means that we can regard an algorithm solving TSP as *the same* — in a sense — as one that solves PSAT. To identify algorithms that are p-time reducible to each other gives us a higher level view of algorithms.

2. If your boss gives you a hard-looking problem to solve, and you are feeling lazy, one way to avoid solving it is to show that it can't be solved by a Turing machine at all (§5). Few bosses would then persist.

If this is not possible, it's almost as good to show that it is NP-complete. NPC problems are usually considered intractable, being the hardest in NP. If the boss insists that you write a program to solve it, you could respond that most boffins believe no such program would run in a reasonable time even for small inputs. You could produce the graph in Fig. 6.1.

So the boss says: 'You're from Imperial College! You ought to be able to find a really clever search strategy, giving a program that runs in polynomial time — even n^5 or so!' But as NP-complete problems are the ≤-hardest in NP, not only would a clever program immediately give fast solutions to the 1,000 or so known NP-complete problems — a huge range including TSP, PSAT, college timetabling problems, map colouring problems, planning problems from AI, etc. etc, in totally different application areas — but to all problems in NP (e.g., primality and non-primality testing). Different groups of people have been looking for fast solutions to these for years, without success.[4] Some applications (e.g., in cryptography) even rely on the assumption that their pet problem has no fast solution. So you would upset many apple-carts if you found a polynomial time solution. With the military after you, you might have to become a traveller, never visiting any city more than once.

No joy? You could hint that a p-time solution would solve P=NP and so make you more famous than your boss. If that doesn't work, you really are stuck, and if a fast solution is essential you should consider:

- optimising your algorithm as far as possible;
- restricting the problem to certain special simpler inputs;
- hoping for the best in the average case;
- looking for a sub-optimal, probabilistic or genetic 'solution';
- finding heuristics (an AI-style approach).

See Harel's or Sedgewick's book for these. For example, for the 'flat' version of TSP where the map is 'real' — for any 3 cities x,y,z, the distance from x to z is at most the sum of the distances from x to y and y to z — there

4 But they haven't proved there's no fast solution. Maybe this failure indicates there is a fast solution!

is a p-time algorithm that delivers a route round the cities of at most twice the optimal length. But for the general version of TSP, the existence of such a polynomial time algorithm would imply P=NP.

12.2 PROVING NP-COMPLETENESS BY REDUCTION

The best way to prove that a yes/no problem A is NP-complete is usually to show that:

(a) A is in NP, and

(b) A is ≥-harder than a known NP-complete problem B (i.e., A≥B).

For if A is ≥-harder than a ≥-hardest problem in NP, but is still in NP, then A must also be ≥-hardest in NP: i.e., NP-complete.

(a) is usually easy, but must not be forgotten. (For example, there are unsolvable problems A that satisfy (b). Satisfiability for *predicate* logic is unsolvable, but is clearly ≥ PSAT since PSAT is a special case. Such problems are not NP-complete, being outside NP.) One can either prove (a) directly, as in §10.2, or else show that A ≤ C for some C known to be in NP, and then use §11.4.1.

To show (b) you must reduce a known NP-complete problem B to A in p-time. Any B in NPC will do. There are now about 1000 B's to choose from. A popular choice is 3SAT:

- 3SAT: Given: a propositional formula F that is a conjunction of *clauses* of the form $X \vee Y \vee Z$ where X, Y, Z are atoms or their negations. E.g., $F = (p \vee q \vee \neg r) \wedge (\neg s \vee x \vee w)$.[5] Question: is there a valuation v such that v(F) = true?

Cook showed that 3SAT ∈ NPC in his 1971 paper. Because the instances of 3SAT are more limited than those of PSAT, it is often simpler to reduce 3SAT to the original problem A than to reduce PSAT to it.

12.2.1 Exercise/problem

Recall the definition of the class co-NP (§9.3.3.1). Show that NP = co-NP iff NPC ∩ co-NP ≠ Ø.[6]

[Quite hard; for mathematicians] Show PSAT~ 3SAT, and HCP ~ PSAT. You can assume Cook's theorem (below).

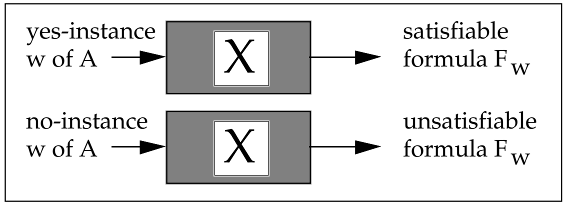
5 The '3' refers to there being 3 disjuncts (X,Y,Z above) in each clause. The formula F may use many more than 3 atoms.

6 By §9.3.5.1, if P=NP then NP = co-NP. It might be true that NP = co-NP and still P≠NP, but this is thought unlikely. Cf. the discussion in §9.3.5.

12.3 COOK'S THEOREM

Of course, the first-discovered NP-complete problem, PSAT, wasn't shown NP-complete by reduction — no NPC problems to reduce to it were known! So to end, we'll sketch Stephen Cook's beautiful proof that PSAT is NP-complete. We already showed (i) PSAT $\in$ NP in §10.2.3. So it's enough to show (ii) if A $\in$ NP then A $\leq$ PSAT. Fix any A $\in$ NP; we want to build a deterministic p-time Turing machine X such that:

- Given a yes-instance w of A, X outputs a satisfiable formula F_w of propositional logic;
- Given a no-instance w of A, X outputs an unsatisfiable formula F_w of propositional logic.



12.1 hoped-for X reducing A to PSAT

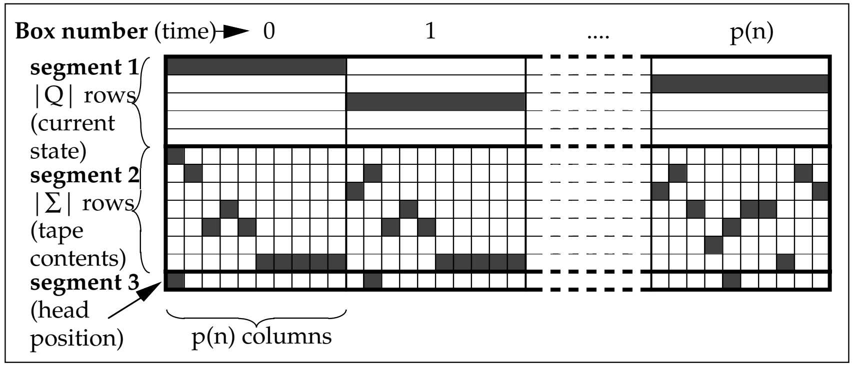
X must construct F_w from w deterministically, and in p-time.

All we know is that A $\in$ NP. So there's a non-deterministic Turing machine N = $(Q,\Sigma,I,q_0,\delta,F)$ solving A, and a polynomial p(n) such that no run of N on any input of length n takes longer than p(n) steps.

As N solves A, the yes-instances of A are exactly those w for which N has an *accepting run* (one that ends in a halting state for N). Roughly, the formula F_w will directly express conditions for there to be an accepting run of N on input w! F_w will be satisfiable iff there is such a run. Compare Gödel's theorem (§5.4), where we said there was a formula R(x) of arithmetic that said x coded an accepting run of a Turing machine M on an input w. This showed that we can describe Turing machines by logical formulas, and was known before Cook. But here we must use propositional logic, not arithmetic. Propositional logic is not usually powerful enough to cope, but Cook's insight was that it's OK here, as N is 'simple' (it runs in p-time, so essentially we can bound x in R(x)).

12.3.1 Table to represent a run of N on w

When does N have an accepting run on w? We can easily (but verbosely) represent a run of N on w of length n by a *table:*



12.2 Table representing a run of N on w of length n

The table has p(n)+1 big *boxes*, labelled 'time': 0,1,…,p(n). They will represent the *configuration* of N at each step: its state, head position, and the tape contents. Box 0 (magnified in Fig 12.3 below) will represent N's configuration at time 0: initially. Box 1 will represent the configuration at time 1, and so on. We know N halts by time p(n), so we don't need any more boxes than 0,1,…,p(n).

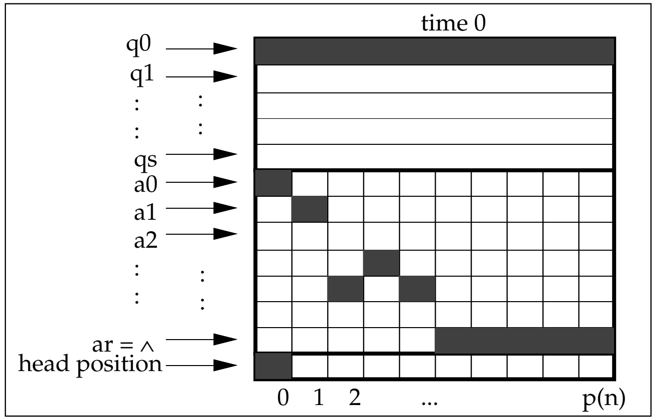
How does the table record a run of N on w? Each box is divided horizontally into three *segments*, to describe the configuration of N at the time that concerns the box (the 'current time'). The first segment indicates the state of N at that time. It's divided into as many rows as there are states, one row for each state. We shade the row of the current state. So if $Q = \{q_0,..,q_s\}$ and the current state is q_i, row i alone will be shaded. In Fig 12.3 (time 0) row 0 is shaded because initially N is in state q_0.

The second segment describes the current tape contents. Now as N only has at most p(n) moves, its head can never get beyond square p(n). So all tape squares after p(n) will always be blank, and we need only describe the contents up to square p(n). We do it by chopping the segment up into rows and columns. The *rows* correspond to symbols from $\Sigma = \{a_0,\ldots,a_r\}$ say, where $a_r = \wedge$, and the *columns* correspond to squares 0,1,…,p(n) of the tape. We shade the intersection of row i, column j iff a_i is currently the character in square j. So the shading for time 0 describes the initial tape contents: w itself[7]. We can read off w from Fig. 12.3: it is $a_0a_1a_4a_3a_4$. The rest of the tape is $a_r = \wedge$.

[7] Technical point: by replacing p(n) by p(n)+n if need be, we can assume that p(n)≥n. We still have $time_M(n) \leq p(n)$. So we've room to record w itself.

Finally the third segment describes N's current head position. We divide it into columns 0,1,…,p(n), and shade the column where the head is. The head never moves more than p(n) away from square 0 in any run (it hasn't time), so we only need p(n)+1 columns for this. For the time 0 box (Fig. 12.3) we shade column 0, as N's head begins in square 0.

If N halts before p(n) we can leave all later boxes blank.



12.3 Box 0 represents *config*(q_0,$a_0a_1a_4a_3a_4$,0) (cf. §10.4.1)

Will a table like that of Fig. 12.2 but with random shading correspond to a real accepting run of N? No: there are four kinds of constraint.

(i) For each t ≤ p(n), the box for time t must represent a genuine configuration C(t) of N. So exactly one row of segment 1 in each box should be shaded (because N is in exactly one state at each time). Similarly, just one column of segment 3 should be shaded, as the head is always in a unique position. And each column of segment 2 should have exactly one shaded row (as each tape square always has a single character).

(ii) C(0) must be the initial configuration of N. So the box for time 0 should say that the head is in square 0 and the state is q0, as in Fig. 12.3. Moreover, it should say that the initial tape contents are w.

(iii) The whole table must represent a run of N. The successive configurations C(t) represented by the boxes should be related, as we have to be able to get from C(t) to C(t+1) by a single step of N. So there'll be further constraints. Only one tape character can change at a time; and the head position can vary by at most 1. Compare boxes 0 and 1 of Fig. 12.2. And the new character and position, and new state, are related to the old by δ. E.g., we can read off from Fig 12.2 that (q_0, a_0, q_2, a_2, 1) is an instruction of N.

(iv) The run indicated by the table must be accepting. This is a constraint on the final state, as shown in segment 1 of the last non-empty box. If the shaded state is in F, the run is accepting; otherwise, not.

Any accepting run of N on w meets these constraints and we can fill in a table for it. Conversely, if we fill in the table so as to meet the four constraints, we do get an accepting run of N on w. So the question *does N accept w* is the same as *can we fill in the table subject to the four constraints?*

12.3.2 Describing the table with logic

Filling in squares of the table is really a logical Boolean operation — either a square is shaded (1), or not (0). Let's introduce a propositional atom for each little square of the table. The atom's being true will mean that its square is filled in. So a valuation v of the atoms corresponds exactly (in a 1–1 way) to a completed table (though it may not meet the constraints).

12.3.3 Describing the constraints with logic

The constraints (i)–(iv) can be written as constraints on the truth values of the atoms. It is possible to write a propositional formula F_w that expresses these constraints. F_w does not say which squares are filled in: it only expresses the constraints. Given any valuation v of the atoms that makes F_w true, we can read off from v a shading for the table that meets the four constraints. And from any validly-completed table we can read off a valuation v to the atoms such that v(F_w) = true.

So the question *can we fill in the table subject to the four constraints?* is the same as the question *is there a valuation v making F_w true?*

12.3.4 Writing F_w

How do we write F_w? See Rayward-Smith for details. For example, suppose the atoms corresponding to the little squares in the first column of segment 2 of box 0 are P_0, P_1, …, P_r. If the first symbol of w is a_0, as it was above, then only the first little square is shaded, so of the P_i, P_0 should be the only true atom. Therefore we include the clause $P_0 \wedge \neg P_1 \wedge \neg P_2 \wedge \ldots \wedge \neg P_r$ in F_w.

Writing F_w in full is tedious, but clearly it's an algorithmic process. And in fact, if we know the parameters Q,Σ,I,q_0,δ, and F of N, we can design a deterministic Turing machine X running in polynomial time[8] that writes out F_w when run on input w!

[8] To make this more plausible, let's ask how many atoms we need. There are p(n) boxes, and in each box, segment 1 has |Q| = s rows, segment 2 has p(n)x|Σ| = r.p(n) little squares, and segment 3 has p(n) little squares. The total is p(n)[(r+1).p(n)+s] — a polynomial!! So the number of atoms we need is a polynomial in n.

12.3.5 The end

We have $f_X(w) = F_w$.

But now we're finished. For w is a yes-instance of A iff N accepts w, iff there's some valuation v making $f_X(w) = F_w$ true, iff F_w is a yes-instance of PSAT. Hence X reduces A to PSAT. Since X is deterministic and runs in p-time, we have $A \leq PSAT$. But this holds for any A in NP. So PSAT is NP-complete. QED and goodnight.

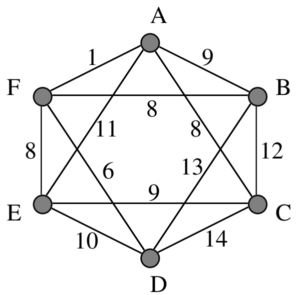
Sample exam questions on parts II,III

1. (a) Explain the underlined terms:
   (i) connected graph
   (ii) spanning tree (of a connected graph)
   (iii) the Hamiltonian circuit problem
   (iv) the travelling salesman problem.

(b) A *maximal spanning tree* of a connected weighted graph G is a spanning tree T of G, such that the sum of the weights of the edges in T is as large as possible (i.e., no spanning tree of G has larger total weight).

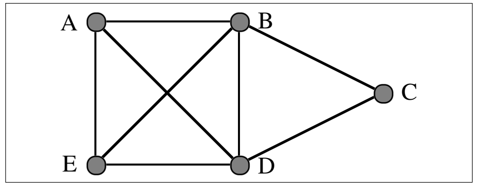
(i) Suggest an algorithm to find a maximal spanning tree of a connected weighted graph. (Do not prove your algorithm correct.)

(ii) Use your algorithm to find a maximal spanning tree of the following weighted graph:



3a Briefly explain the difference between the *depth-first* and *breadth-first* methods of constructing a spanning tree of a connected graph.

b i) List the edges of a spanning tree of the following graph, using the depth-first method.



ii) Repeat b(i), using the breadth-first method.

c Explain the meaning of the underlined terms:
i) A Hamiltonian circuit of a graph;
ii) A minimal spanning tree (MST) of a connected weighted graph.

d Let G be a connected weighted graph. Explain why any Hamiltonian circuit of G must have greater total weight than the total weight of any minimal spanning tree of G.
[Hint: transform a Hamiltonian circuit into a spanning tree.]

3. Let A and B be yes/no problems.
(a) Explain what is meant when we say that:
   (i) A reduces to B in p-time (in symbols, $A \leq B$)
   (ii) A is in P
   (iii) A is an NP-problem
   (iv) A is NP-complete.

For each of (i)-(iv), give an example of a problem A (or in (i), problems A and B) satisfying the condition.

(b) Show that the p-time reduction ordering $\leq$ of (a(i)) above is *transitive* on yes/no problems.

(c) (i) Show that if A is an NP-problem, B is NP-complete and $A \geq B$ then A is also NP-complete.
(ii) If A, B are both NP-complete, does it follow that $A \geq B$? Justify your answer.

4a Let A and B be arbitrary yes-no problems. Define what it means to say that *A reduces to B in p-time* (in symbols, $A \leq B$).

b Let $\leq$ be the relation of part a. Prove that if $A \leq B$ and $B \in NP$ then $A \in NP$.

c Let $\leq$ be the relation of part a.

i) Define the class NPC of NP-complete yes-no problems.

ii) Let HCP, PSAT be the Hamiltonian circuit and propositional satisfaction problems, respectively. Let A be a yes-no problem, and suppose that HCP ≤ A and A ≤ PSAT. Prove directly from your definition in part c(i) that A is NP-complete.
[You can assume that HCP and PSAT are NP-complete.]

Part III in a nutshell

§9: We want to analyse the complexity of solvable problems in terms of how long they take to solve. We introduce yes/no problems to simplify our discussion. A Turing machine can solve such a problem by accepting (= halting & succeeding on) precisely the yes-instances, and rejecting (= halting & failing on) the no-instances. So we don't need to consider its output. We define the *run time function* $time_M(n)$ of a Turing machine M to be the longest it can run for on any input of size n. A Turing machine *runs in p-time* if its run time function is bounded by some polynomial. P is the class of 'tractable' yes/no problems solvable by some Turing machine running in p-time. Whilst no-one has proved it, many problems such as HCP (§7), TSP (§8) and PSAT (propositional satisfaction) do appear intractable, although they are solvable by exhaustively searching for a solution. We are lacking an efficient search strategy.

§10: Whilst waiting for a strategy to be devised (though most think there isn't one) we can examine which problems would yield to such a strategy. To do this we use the *non-deterministic* Turing machine (NDTM), which can make choices during a run. More than one instruction may be applicable in a single configuration. (A strategy could be plugged into such a machine, narrowing the choice to one again.) A NDTM N *accepts* its input iff it has at least one accepting run on that input. It *rejects* an input if all its runs on that input end in failure. It *solves* a yes/no problem if it accepts the yes-instances and rejects the rest, as before. Its *run time function* $time_N(n)$ is the length of the longest run of N on any input of size n. The definition of N running in p-time is as before.

NDTMs can solve yes/no problems like PSAT, HCP, TSP etc. in p-time, as they simply guess a possible solution x to the input instance w, accepting w if x is in fact a solution. Checking that x is a solution to w (e.g., x is a round trip of length ≤d in TSP, or in PSAT, x is a valuation making the propositional formula A true) can be done in p-time. We let NP be the class of all yes/no problems solvable by some NDTM in p-time. So TSP, HCP and PSAT are all in NP. As deterministic Turing machines are a special case of NDTMs, any P problem is in NP.

However, though faster than deterministic Turing machines, NDTMs can solve no more yes/no problems. This gives more evidence for Church's thesis. A deterministic Turing machine can simulate any NDTM by constructing all possible runs in a breadth-first manner, and seeing if any is accepting. That is, it does a full exhaustive search of the tree of runs of the NDTM.

§11: We can formalise the notion of a yes/no problem A being no harder than another, B, by *p-time reduction.* To reduce A to B in p-time ('A≤B') is to find a (deterministic) p-time Turing machine X that converts yes-instances of A into yes-instances of B, and similarly for no-instances. Since X is fast, any given fast solution F to B can be used to solve A: first apply X, then F. If F solves B non-deterministically, the solution we get to A is also non-deterministic, so if B ∈ NP and A≤B then also A ∈ NP: that is, ≤-easier problems than NP-problems are also in NP.

It's easy to convert yes-instances of A into yes-instances of A in p-time (leave them alone!), so A≤A and ≤ is reflexive. ≤ is transitive, as if X converts A-instances to B-instances in p-time, and Y converts B-instances to C-instances in p-time, then running X then Y converts A-instances to C-instances (always preserving parity: yes goes to yes, no to no). Careful counting shows that this takes only p-time (remember to return heads to square 0, and that the length of input to Y may be longer than the original input to X). Hence A≤B≤C implies A≤C, so ≤ is transitive. ≤ is thus a pre-order.

Problems in P are ≤-easiest of all. For if A ∈ P and B is arbitrary, we can convert instances w of A to instances of B in p-time and preserving parity) by the following trick. As we can solve A completely in p-time, we find out in p-time whether w is yes or no for A. Then we hand over a fixed instance of B of appropriate parity.

§12: The ≥-hardest problems in NP are called NP-complete. A yes/no problem A is NP-complete if A is in NP but A ≥ B for all NP-problems B. Cook proved in 1971 that PSAT is NP-complete, so NP-complete problems exist. His proof went like this. We know PSAT ∈ NP. If A ∈ NP, there's a p-time NDTM N solving A. There's a deterministic p-time Turing machine X that given an instance w of A, outputs a propositional formula F_w expressing the constraints that must be met if N is to have an accepting run on w. Any valuation making F_w true shows that the constraints can be met, and gives an accepting run of N on w. Thus w is a yes-instance of A iff F_w is satisfiable. Hence X reduces A to PSAT in p-time, and so $A \leq PSAT$. This holds for all $A \in NP$. Thus PSAT is NP-complete.

As well as PSAT the NP-complete problems include 3SAT, HCP, TSP and some 1,000 other common problems. As they are all equally ≤-hard, a p-time solution to any would yield p-time solutions for all. But as many of them have been attacked seriously for years without success, it's probably not worth the effort trying to write a fast algorithm for any of them.

Perhaps through frustration, people believe all NP-complete problems to be intractable: the famous question P=NP? is thought almost universally to be false. However, no-one has proved it (or at least published a proof) either way. Unlike Church's thesis, which is by its nature unprovable, one day a proof of P=NP or P≠NP may appear.